# From Logs to Causal Analysis: Extracting Data to Diagnose Large Systems


## ABSTRACT

Causal analysis is an essential lens for understanding complex system dynamics in domains as varied as medicine, economics and law. Computer systems often exhibit a similar level of complexity, but much of the information that could help subject them to causal analysis is only available in long, messy, semi-structured log files.

In this work, we want to apply causal reasoning to make large systems debugging easier. In order to transform logs into a representation amenable to causal analysis, we employ methods drawn from the areas of data transformation, cleaning, and extraction. We also present algorithms for how to efficiently use this representation for causal discovery - the task of constructing a causal model of the system from available data. Our proposed framework gives log-derived variables human-understandable names and distills the information present in a log file around a user's chosen *causal units* (e.g. users or machines), generating appropriate aggregated variables for each causal unit. It then exposes these variables through an interactive interface that users can leverage to recover the portion of the causal model relevant to their question. This makes causal inference possible using off-the-shelf tools.

We evaluate our framework using Sawmill, a prototype implementation, on both real-world and synthetic log datasets and find it to be: (1) **Accurate**, achieving on average across datasets a mean reciprocal rank of 0.8018 for relevant causes (41.89% better than the next best baseline) and a mean ATE error of 20.27% (compared to 31.26% for the next best baseline); (2) **Computationally efficient**, requiring a mean of 346.92s (4.30s per MiB of log data) to go from a log file to a quantitative effect, while scaling linearly with log complexity; (3) **Humanly efficient**, requiring between 6-10 user interactions end-to-end; and (4) **Complete**, with each component of our framework shown to both *add value* and to *be efficient*.



## CCS CONCEPTS

• **Software and its engineering** → **System administration**; • **Computing methodologies** → **Causal reasoning and diagnostics**; Natural language generation.

## KEYWORDS

Logs, Fault Diagnosis, Causality, Large Language Models, Causal Discovery

**ACM Reference Format:**

. 2024. From Logs to Causal Analysis: Extracting Data to Diagnose Large Systems. In *Proceedings of International Conference on Management of Data (SIGMOD '25).* ACM, New York, NY, USA, 15 pages. https://doi.org/XXXXXXX.XXXXXXX





*SIGMOD '25, June XX-YY, 2025, Berlin, Germany*





ACM ISBN 978-1-4503-XXXX-X/18/06

https://doi.org/XXXXXXX.XXXXXXX


## 1 INTRODUCTION

The scale and complexity of today's computer systems makes failures a frequent phenomenon [32, 43, 72, 94] and their diagnosis a challenging endeavor, especially in production [45, 69, 82]. Traditional systematic debugging techniques, like testing [22, 51, 53, 57, 68], formal verification [17, 25, 40, 85] and simulation [44, 48, 87], are often a bad fit for operations teams faced with problems on a large complex system, who do not have the time and expertise (or often even the access permissions) to fully ascertain the correctness of the codebase [43, 74]. Instead, they have to work backwards from each failure towards its cause using observational data collected from the system. Informally, we have heard reports from operations teams spending tens of hours with tools such as Splunk [1] or Datadog [2] in order to diagnose a poorly-understood problem.

However, operations teams want to go beyond diagnosing the problem - they want to repair it by alerting the appropriate engineering team. Moreover, whenever there are *multiple ways* to fix a problem, they would like to identify the *most efficient* way to utilize engineering effort. This points to the growing field of causal reasoning [60], which aims to accurately describe and quantify the mechanisms driving each event. Causality has provided scientists with a common language to express and evaluate hypotheses about complex systems across diverse domains [12, 65, 77].

We aim for a general-purpose causal diagnosis system that can address a wide range of software and data systems problems. Existing log analysis tools offer an ideal starting point for general-purpose causal analysis, given their coverage of software and hardware components and the existing management infrastructure. If we could ask a Datadog [2]- or Splunk [1]-like system *why* an observed problem took place, then we might dramatically lower the Mean Time to Repair (MTTR). To achieve this, we aim to combine log management with the best theoretical machinery science can offer for reasoning about problems and causes.

**Episode** 1. *In theory, causality sounds promising for managing systems. But, in practice, the log data available to system administrators like Alex look like Figure 1a: semi-structured chronological accounts in text form. Unfortunately for hundreds of operations teams, system logs fall badly short of the input causal inference requires.*

To satisfy Pearl's theory of causality [60], which we adopt in this work and summarize in Section 2, causal inference toolkits require a representation like Figure 1b – a table with *one row per user*, including only *few, relevant variables* and *without missing data*. They also require a causal model covering the variables of interest [60], something non-trivial to recover from a log. In summary, Alex faces three challenges to answering causal questions based on logs:

```
20:24:44 INFO u0 q34 Running CREATE INDEX midx ON metrics (id);
20:32:25 INFO u0 q35 Running SELECT * FROM metrics WHERE id=562;
20:32:26 INFO u0 q35 Ran in 607.31ms
20:32:28 INFO u0 q36 Running SELECT * FROM metrics WHERE id=555;
20:33:28 INFO u0 q36 Query timed out
```

**(a) The log data available to Alex.**

| User | M: Free Memory | I:Index Presence | L:Latency Mean (ms) | D: Data Size (GB) | T: Timeouts per day |
|---|---|---|---|---|---|
| $U_0$ | 67.80 % | 1 | 637.02 | 64.41 | 56852 |
| $U_1$ | 80.96 % | 0 | 372.60 | 38.07 | 29164 |
| ... | ... | ... | ... | ... | ... |

**(b) The tabular data required for causal inference.**

**Figure 1: Log data is unsuitable for causal inference.**

**Challenge A: Deriving the Schema.** Logs include textual *log messages*, each printed on one or more *log lines*. This is a far cry from the tabular dataset with interpretable column names in Figure 1b. Log parsing algorithms can convert log data into *some* tabular representation [35, 96, 101], but this can yield hundreds of variables without a human-friendly way to navigate them. For Alex to engage with them, we need to derive a human-understandable schema from these variables. However, most of the variables are very context specific (e.g. the port number reported in a connection message), making it challenging even for a human to annotate them succinctly.

**Challenge B: Distilling the Data.** Causal inference requires the same data fields per causal unit - e.g. per user, for Alex's example. Simply parsing the log falls short of this goal in two ways. First, there will be a lot of missing values, because each log message only reports *some* variables: some messages report latency, others might flag a timeout etc. Missing values can worsen *selection bias* if removed from the dataset, invalidating conclusions [13] - i.e. the remaining observations may be a biased sample of the population. They can also mean trouble for the *positivity* assumption [60] (Section 2), required by causal inference. Second, logs record fine-grained information, but knowing, e.g., the latency of each individual query is unhelpful for an analyst like Alex. These values must be summarized, while preserving "causal usefulness".

**Challenge C: Obtaining the Causal Model.** Causal inference requires a causal model to adjust for confounders [60] - variables that influence both treatment and outcome, distorting their relationship. A model can be derived either from the data, using a causal discovery algorithm, or from variable semantics by a human expert. However, neither method would work well for log variables. Existing causal discovery algorithms would face significant complexity given the number of log-derived variables and the possible functional dependencies among them. Even if successful, they may yield a model too busy to inspect for correctness - but just a few incorrect assumptions can swing the results of our analysis. For a human expert, the problem size makes the prospect of hand-crafting a full model daunting. A large language model would also be ineffective and expensive, due to the esoteric nature of some log-derived variables and the quadratic possible causal relationships.

**Episode** 2. *Alex has access to the log files of 1,000 different users, some of which are facing high latency. Alex wants to identify how to reduce latency most effectively and notify the right engineering team, but is stumped given the causal inference challenges above.*

In this work, we propose a framework to address these challenges and transform logs into high-grade causal resources, on which we can apply known causal approaches to enable rapid diagnosis of system failures. Our framework can be an important tool for engineers managing complex systems, like Alex. To address **Challenge A**, we derive human-understandable variable tags by leveraging Large Language Models; for **Challenge B**, we distill log information around *causal units* and generate new variables for every causal unit to maximize "usefulness"; while in the face of **Challenge C**, we propose interfaces to recover a relevant, partial causal model of the system from log-derived data. The user can then use the resulting dataset and causal model for causal inference.

We are not the first to apply causal reasoning to software systems. CausalSim [9] learns a causal model of system dynamics from randomized controlled trial data. To localize failures in microservice architectures, Sage [29] uses modular causal graphs based on telemetry data, while Aggarwal et al. model causality among the *counts* of error-level log messages [7]. Other works use "causality" in a less technical sense - e.g. Falcon [55] and Horus [56] use temporal precedence as a proxy for causality. However, none of these works tackles the end-to-end problem of fully leveraging log files for principled causal analysis. We make the following contributions:

- We propose a framework for transforming logs to a representation satisfying the assumptions of causal inference.
- We introduce *exploration-based causal discovery*, in which curated data-driven suggestions meet expert user feedback.
- We evaluate Sawmill, a prototype implementation of our proposed framework and the first practical system that can go from log files to causal effects end-to-end, and find it is:
  - **Accurate**, achieving on average across datasets a mean reciprocal rank of 0.8018 for relevant causes (41.89% better than the next best baseline) and a mean ATE error of 20.27% (compared to 31.26% for the next best baseline).
  - **Computationally efficient**, requiring a mean of 346.92s (4.30s per MiB of log data) to go from a log file to a quantitative effect, while scaling linearly with log complexity.
  - **Humanly efficient**, requiring between 6-10 user interactions end-to-end.
  - **Complete**, with each component of our framework shown to both *add value* and to *be efficient*.

In the remainder of this paper, we will first provide some background on causality (Section 2) and an overview of the problems we tackle (Section 3). We will next describe individual components (Sections 4- 6) and evaluate a prototype (Sections 7- 8). We will then discuss related work (Section 9) before concluding (Section 10).

## 2 BACKGROUND ON CAUSALITY

We now discuss Pearl's causal model [60, 61], which we adopt.

**Interventional Causal Inference.** Causal inference aims to estimate the causal effect of a treatment variable $T$ on an outcome variable $Y$, over a population of *causal units* - e.g. users in Alex's case. This effect may be distorted by *confounders* $\mathcal{Z}$ which influence both $T$ and $Y$ [60, 61]. For example, if Alex estimates the effect of index presence on latency, a confounder may be the size of the table - a larger table can directly increase latency, but it may also mean that more attention has been paid to creating an index.

To avoid confounders, one can collect *interventional* data, where $T$ no longer depends on $\mathcal{Z}$ because it is randomly determined - e.g. through a randomized controlled trial (RCT). *Assigning* $T = t$ is denoted $\text{do}(T = t)$, to distinguish it from *observing* that $T = t$ [60]. We can then define the Average Treatment Effect [66] of $T$ on $Y$:

**Definition** 1. *The **Average Treatment Effect (ATE)** is*

$$ATE(T, Y) = \mathbb{E}[Y \mid \text{do}(T = 1)] - \mathbb{E}[Y \mid \text{do}(T = 0)]$$

Intuitively, the ATE formalizes a "dose-respose" mental model of causality: if we give users a "dose" of free memory (e.g., 1% more), by how much will query latency decrease? This quantifies the trade-offs involved when several interventions could have *some* impact, helping Alex pick the most efficient one. Importantly, this definition requires the Stable Unit Treatment Value Assumption (SUTVA) [67]: the outcome of each unit should not depend on the treatments of *other units*. For example, user $U_0$'s index creation should not affect the latency of user $U_1$'s queries. We will return to the SUTVA later.

**Observational Causal Inference.** RCTs are expensive and time-consuming, if not unethical or infeasible - Alex would have to convince users to randomly change their settings, potentially harming performance. We can use *observational* data instead, if we *adjust for the confounders*, i.e. compute the ATE per value of $\mathcal{Z}$ and weigh the results. On top of SUTVA, this requires two more assumptions [60]:

- **Unconfoundedness**: The entire set $\mathcal{Z}$ is observed, so that it can be adjusted for to yield the correct ATE.
- **Positivity** (or **Overlap**): For each value of $\mathcal{Z}$, we have observations for every value of $T$.

**Definition** 2. *The **Average Treatment Effect (ATE) from observational data** satisfying unconfoundedness and positivity is*

$$ATE(T, Y) = \mathbb{E}_Z\left[\mathbb{E}\left[Y|T = 1, \mathcal{Z} = z\right] - \mathbb{E}\left[Y|T = 0, \mathcal{Z} = z\right]\right]$$

**Causal Models and Graphs.** Definition 2 requires a sufficient set of confounders $\mathcal{Z}$ to adjust for, for given $T$ and $Y$. To find them, we can leverage a causal model [60] that captures causal relationships among the variables. These relationships can be arbitrary, but they are most often assumed to be linear [60, 80], which we also adopt:

**Definition** 3. *Given variables $V_i$ for $i = 1, ..., n$, with associated sets of* parents *$\mathcal{P}_i$ (i.e. sets of variables that directly affect their value) and error terms $E_i$, a **linear causal model** is a set of equations:*

$$V_i = E_i + \sum_{P \in \mathcal{P}_i} \lambda_P P, \quad \lambda_P \in \mathbb{R}, \quad i = 1, ..., n$$

A causal model can be visualized as a *causal graph* [59]: a directed acyclic graph with one node per variable, where a directed edge $V_i \rightarrow V_j$ implies $V_i \in \mathcal{P}_j$. Figure 2 shows a causal graph for the variables in Figure 1b, highlighting the confounding we mentioned.

**Causal Discovery.** To obtain a causal model, we can either rely on expert input, or use data-driven *causal discovery* algorithms [30, 84], which generally rely on either conditional independence calculations using the available data, or on various methods for scoring graphs based on the information they capture. However, these algorithms often impose additional restrictions on the dataset, such as requiring a non-singular correlation matrix among the variables.

**Episode** 3. *Alex can now phrase a crisper question: which intervention has the most negative ATE on query latency?*

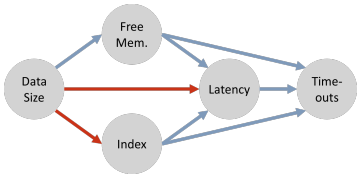



**Figure 2: A causal graph for Figure 1b. The size of the data may confound the effect of index presence on latency.**

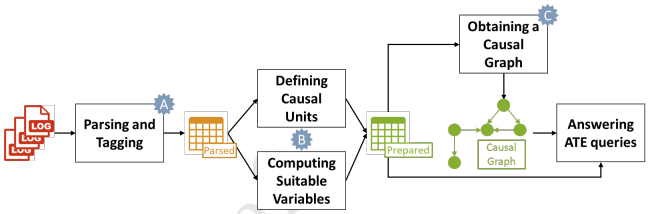



**Figure 3: Our framework, indicating the three challenges.**

## 3 A FRAMEWORK OF PROBLEMS

To answer the question from Episode 3, Alex needs a framework like the one in Figure 3, which transforms logs into a table like Figure 1b and derives a causal graph like Figure 2, ultimately enabling ATE calculation. The transformation begins with **Parsing and Tagging**, where the log is parsed into a preliminary table called the *parsed table* and meaningful human-understandable tags are derived for each variable in the table. The information in the parsed table is then reorganized to satisfy the requirements for causal inference, by **Defining Causal Units** and **Computing Suitable Variables**, leading to a new table which we call the *prepared table*. Finally, the prepared table is used for the purposes of **Obtaining a Causal Graph**. Our framework enables a user experience like the following:

**Episode** 4. *Alex discovers that our framework can create high-quality resources for causal analysis of log data. With a few interactions, Alex derives the prepared table and causal graph within minutes and finds that the highest absolute ATE on mean latency is exerted by the presence of an index. Alex forwards this to the database team.*

However, this framework contains three important challenges, as described in Section 1. We dive deeper into them below and define the problems that we will address in subsequent sections.

### 3.1 Challenge A: Deriving the Schema

Text logs must first be *parsed* into a machine-friendlier format: a table of variable observations per log message. This involves defining variables and collecting their values in the *parsed table*:

**Definition** 4 (Parsed Table). *For a log with $M$ log messages, the parsed table includes one row per log message and one column for each of the parsed variables $\mathcal{W} = \{W_i \mid 1 \leq i \leq W_{max}\}$. Let $W_{i,j}$ denote the value of $W_i$ for the $j$-th log message.*

Despite log parsing being challenging, both "classical" [26, 36, 37, 41] and learning-based [3, 11, 21, 24, 49, 52, 54, 88] log parsing algorithms abound [35, 96, 101]. However, these algorithms often focus on the log templates - the non-variable parts of each log message - with comparatively less attention paid to the variables.

In our framework, we hope to actively work with the *variables* to perform causal inference. At a minimum, this requires *naming* the variables in a human-understandable way, so that users can effectively reason about them. We will call these human-understandable names *tags*. We therefore obtain the following problem:

**Problem** 1 (Variable Tagging). *Given a collection of parsed variables $\mathcal{W}$ and example log messages that include $W_i$ for each $W_i \in \mathcal{W}$, derive a succinct human-understandable tag for each $W_i$.*

## 3.2 Challenge B: Distilling the Data

The rows of the parsed table, one per log message, are unhelpful for causal inference. We instead must consider groups of log messages together as a *causal unit*, with each group being "one data point" for the question at hand. For example, the right causal unit for some questions might be a user, since log messages referring to the same user collectively capture the user's activity. But not every choice of causal unit is permissible, because of the SUTVA (Section 2) - e.g., users may be unsuitable causal units if they share hardware: user $A$'s work could impact user $B$'s latency.

There can be a varying number of values for each parsed variable in each causal unit. For example, if Alex defines a causal unit per user, we may have a different number of query latency readings for each user. To make units comparable, we must replace such varying-size collections of values with *the same* value(s) per causal unit. This transformation will lead to the *prepared table*:

**Definition** 5 (Prepared Table). *For a log with $C$ causal units, the prepared table includes one row per causal unit and one column for each of the prepared variables $\mathcal{U} = \{Ui \mid 1 \leq i \leq U_{max}\}$. Let $U_{i,c}$ denote the value of $U_i$ for the $c$-th causal unit.*

Each prepared variable is derived from some parsed variable, its *base variable*, by using a different function (e.g. the mean) to reconcile the values within each causal unit. To capture the distribution of these values as accurately as possible, we could use several different functions, each yielding one prepared variable. However, this would make the prepared table large - expensive to process and confusing to interpret. We are faced with the following problem:

**Problem** 2 (Optimal Variable Selection). *Given observations of $W_i \in \mathcal{W}$ in some causal unit $c \in C$, obtain a single observation that maximizes usefulness for downstream causal analysis.*

## 3.3 Challenge C: Obtaining the Causal Model

We want to leverage the prepared table for causal inference. However, we also need a causal graph to adjust for the appropriate confounders, as explained in Section 2. A user could easily evaluate the plausibility and direction of the causal relationship between two prepared variables. To keep track of their decisions, we have:

**Definition** 6 (Edge State Matrix). *Let each directed edge $U_i \rightarrow U_j$ where $\{U_i, U_j\} \subset \mathcal{U}$ be in one of three states:*

- *Accepted into the causal graph: $S(U_i \rightarrow U_j) = 1$*
- *Rejected from the causal graph: $S(U_i \rightarrow U_j) = -1$*
- *Undecided: $S(U_i \rightarrow U_j) = 0$*

*Let accepted and rejected edges collectively be termed "decided". Then, let the edge state matrix $ESM = \{S(U_i \rightarrow U_j)\}$*

However, fully specifying the edge state matrix by hand is daunting, since the prepared table could contain hundreds of variables.

Could we simply use an existing algorithm to *discover* a causal graph from the prepared table? In certain small cases this may be possible; however, in most cases, it is out of the question. Given the large number of prepared variables, a full causal graph would be prohibitively expensive to build if using a constraint-based algorithm that tests for conditional independence like PC [80] or GES [80]. If attempting to use an algorithm based on a Functional Causal Model instead, like LiNGAM [76], one would run into issues due to functional dependencies among the prepared variables - for example, LiNGAM assumes that the error terms between pairs of variables exhibit nonzero variance [76], which is not true of many events that co-occur in logs. We confirm these claims in Section 8.5.3.

Could we instead consult a Large Language Model to detect causal relationships? Recent work has indeed shown that LLMs can be effective causal advisors [46]. However, LLMs rely on the *names* of the variables, rather than the associated data. In the context of log-derived variables, such names can be especially esoteric, prohibiting the model from giving a clear verdict. This approach would also require prompting the LLM for each pair of prepared variables, leading to prohibitive runtimes, as we will see in Section 8.5.3.

Since neither of these approaches is suitable alone, we would like to combine the data-driven causal hypotheses with the user's expertise. However, we need to ensure the amount of work expected from the user remains tractable. This leads to the final problem:

**Problem** 3 (Efficient User Consultation). *Obtain a causal model that correctly computes the ATE of interest, while minimizing processing time and user interactions (i.e. decisions on graph edges).*

# 4 DERIVING THE SCHEMA (CHALLENGE A)

## 4.1 Log Parsing

Log parsing in our framework is performed by calling the function Parse. This call involves three steps. First, we let the user optionally specify a log message prefix as an argument to Parse (for example, a timestamp regular expression), based on which multi-line log messages can be identified and treated as one "line" in all downstream analysis. This log message prefix is by default empty, meaning that each log line is by default interpreted as a separate log message. Then, if multiple logs need to be jointly analyzed, we collate them into a single log, which becomes the input to our framework; to each log message, we add an identifier of the log it came from.

Next, users can pass into Parse zero or more regular expressions to parse variables like timestamps. Our framework then uses an off-the-shelf parsing algorithm to identify the rest of the log variables. Our framework is flexible in terms of parsing algorithm, as long as it leads to a parsed table as per Definition 4. We call the non-variable part of each log line the *log template*. To the parsed table, we also add a binary indicator variable for each log template, which takes the value 1 only for rows that match the template. This lets us count the rate of incidence of each log template downstream.

Log parsing can go wrong in two ways: observations of the same variable may be modeled as observations of different variables, or vice-versa. In cases where the parsing algorithm has mapped semantically different variables to the same parsed variable, the

user can identify it in the parsed table and correct it by calling the function SEPARATE on the erroneously defined variable.

Some parsed variables are uninteresting for causal inference - e.g. identifiers. We discard any categorical parsed variable where the number of distinct values exceeds 15% of the variable's occurrences, an approach similar to past work for detecting identifiers [90].

### 4.2 Variable Tagging

After parsing, we address Problem 1 by automatically assigning a human-understandable tag to each parsed variable. Users can later edit these tags manually. For variables parsed using a regular expression, the user provides a tag together with the regular expression. For each of the remaining parsed variables, we consult three sources in sequence to generate a tag. As soon as one source produces a tag, we move on to the next parsed variable. If no source produces a tag, we assign the system-derived variable name as the tag, which is a unique string of symbols. The three sources are:

(1) Attempt to recover a tag from the 3 tokens preceding the variable in the log template. In particular, if the variable is preceded by ':' or '=' (possibly with an interspersed opening quote), use the token before the ':' or '=' as the tag.
(2) Consult GPT-3.5-Turbo [6], providing (a) an example message where the variable appears, (b) the 3 tokens that precede the variable (c) 4 additional example values for the variable from other log messages following the same template.
(3) Consult GPT-4 [58] using the same information as above.

Sometimes, distinct parsed variables may be assigned the same tag - e.g., several may be called "port", from different contexts. We use the tag only for the first parsed variable to obtain it and use a unique system-generated variable name for the rest.

### 4.3 Computational Complexity

The complexity of parsing depends on the chosen parsing algorithm and is external to our framework. For tagging, we require $O(M)$ time and $O(|\mathcal{W}|)$ space for the metadata fed into the GPT prompts, where $M$ is the number of log messages and $\mathcal{W}$ is the set of parsed variables. Depending on the choice of log parsing algorithm, this information can also be collected during parsing. We then require an additional $O(|\mathcal{W}|)$ time and up to $2|\mathcal{W}|$ GPT calls to compute the tags, which take $O(|\mathcal{W}|)$ space to store. Since tagging is a per-variable operation, linear complexity is optimal for this problem.

## 5 DISTILLING THE DATA (CHALLENGE B)

### 5.1 Defining Causal Units

To define causal units, a user selects a parsed variable on which the units will be based and then calls the function SETCAUSALUNIT - e.g. `user_ID` or `machine_ID`. Per Section 3.2, this choice must both reflect the user's intended analysis and respect the SUTVA. We defer to the user's knowledge of the system design for causal unit definition to ensure that this condition is satisfied. In the call to SETCAUSALUNIT, the user can optionally pass a discretization function if a continuous variable is chosen - e.g., a binning function.

**DEFINITION** 7 (DISCRETIZATION FUNCTION). *A discretization function $d$ maps the values of a parsed variable $W_i \in \mathcal{W}$ to a discrete set $\mathcal{V}(d, W_i) \cup \{MISSING\}$. Missing values are mapped to MISSING.*

**DEFINITION** 8 (CAUSAL UNIT). *Let a $W_i \in \mathcal{W}$ and a discretization function $d$. For each $v \in \mathcal{V}(d, W_i)$, we define a causal unit as all columns except $W_i$ for rows of the parsed table for which $d(W_i) = v$. Rows where $d(W_i) = MISSING$ are not mapped to a causal unit:*

$$CU(W_i, d, v) = \{W_{k,j} \mid 1 \le k \le W_{max}\,,\ k \ne i\,,\ d(W_{i,j}) = v\}$$

### 5.2 Aggregation

For each parsed variable, our framework generates a number of prepared variables using different aggregation functions as soon as the user calls the function PREPARE. If the user is confident that a certain aggregate function is best for a particular parsed variable, they can explicitly specify it as an argument to that call. Otherwise, we compute multiple aggregates of each parsed variable $W \in \mathcal{W}$, based on its type. For numerical variables, we consider $\mathcal{A}_W = [max, min, mean]$. For categorical variables, we consider $\mathcal{A}_W = [first, last, mode]$, where $first(W)$ and $last(W)$ return the value of $W$ that appeared earliest or latest within each causal unit, respectively. When calculating these functions, we ignore all *MISSING* values in their inputs. We then one-hot encode any resulting categorical variables, yielding the set of prepared variables:

- A variable $a(W)$, $\forall\, a \in \mathcal{A}_W$, $\forall$ numerical $W \in \mathcal{W}$
- A variable $\mathbb{1}_{a(W)=v}$, $\forall\, v$, $\forall\, a \in \mathcal{A}_W$, $\forall$ categorical $W \in \mathcal{W}$

### 5.3 Aggregate Selection

We now present our response to Problem 2. In general, one cannot find the most useful aggregation function for each parsed variable a priori, since the downstream ATE question is not known. Even if it were, optimizing for it could lead to overfitting, generating variables that would be less informative for adjacent questions the user may later pose. Instead, we maximize the information captured by each prepared variable. For a parsed variable $W \in \mathcal{W}$, we let $\mathcal{A}_W$ be the set of candidate aggregation functions and $R(a)$ be the range of each $a \in \mathcal{A}_W$. Among the prepared variables with $W$ as their base variable, we compute the empirical entropy for each and then only keep the one that maximizes empirical entropy. That is, instead of Problem 2, we solve the related problem:

**PROBLEM** 2B (AGGREGATE SELECTION). *For each $W \in \mathcal{W}$, we only keep the prepared variable resulting from $a^*_W \in \mathcal{A}_W$, where*

$$a^*_W = \arg\max_{a \in \mathcal{A}_W} \sum_{x \in R(a)} -p(x) log_2(p(x))$$

### 5.4 Imputation

In each causal unit, beyond parsed variables with multiple observations, there may be parsed variables with none. For example, if a user never had a query time out, there are no observations of the corresponding log message. Disregarding causal units with missing values would result in *selection bias*, which may impact the accuracy of causal inference. Luckily, imputing missing values in the prepared table is often possible, since the missing values are interpretable: in the example above, the number of timeouts for users with no observed timeouts should clearly be set to zero.

We can impute a default based on domain knowledge, which is easy to tap now that log information is organized along causal units. Since the default may differ per prepared variable, we let the user pass it into PREPARE and perform no imputation by default.

Any causal units with missing values after the imputation step are disregarded from calculations involving the missing variables.

### 5.5 Computational Complexity

Aggregation takes $O(|C||\mathcal{W}|T_{max})$ time, where $C$ is the set of causal units, $\mathcal{W}$ is the set of parsed variables and $T_{max}$ is the dominant time complexity among the aggregation functions. Among the defaults, *mode* dominates with $O(|C|log|C|)$ for a causal unit with $|C|$ elements. The time complexity of one-hot encoding is $O(|C||\mathcal{W}_{cat}|k_{max})$, where $\mathcal{W}_{cat}$ is the set of categorical parsed variables and $k_{max}$ is the maximum number of categories for any of them. Imputation with fixed values takes $O((|C||\mathcal{W}|)$ time.

## 6 OBTAINING THE CAUSAL MODEL (CHALLENGE C)

Per Section 3.3, we want to combine domain expertise and available data to obtain a causal model. We will do this by incrementally building and refining the causal graph using user feedback efficiently, as required by Problem 3. Note that calculating ATE($T$, $Y$) does not involve the full causal graph, but only the paths between $T$ and $Y$. Therefore, we help a user incrementally discover only the part of the causal graph relevant to their analysis, by making data-driven suggestions that the user evaluates using expert knowledge.

We call this approach *Exploration-based Causal Discovery*. The user begins with an "outcome" variable $Y$ (e.g mean query latency for Alex) and constructs the causal graph around it in two phases:

- **Phase I - Finding a "root cause":** Our framework helps the user identify a treatment prepared variable $T$, which *causally affects* $Y$ and is also "actionable" by the user. At the end of this phase, the user has found the *question* - the $ATE(T, Y)$.
- **Phase II - Finding confounders:** The user continues constructing the causal graph to identify potential confounders that affect $ATE(T, Y)$. At the end of this phase, the user has ensured that the *answer* to the question is accurate.

### 6.1 Edge Pruning

Both phases above need user input, so we want to limit the context that the user has to act on. We achieve this through *edge pruning*. Defining a causal graph entails identifying the edges that represent parent-child relations. This task has a semantic component, which is why our overall approach involves the user - only the user can determine whether "CPU usage" and "network bandwidth" are causally related, and in which direction, given their knowledge of the system at hand. However, this task also has a quantitative component, which our framework can tackle alone.

In particular, note from Definition 3 that each variable is linearly related to its parents, up to its own error term. As such, if we run a multivariate regression of a prepared variable $U$ on the remaining prepared variables, and some prepared variable $U'$ gets assigned a zero coefficient, we can safely reject $U' \rightarrow U$: $U'$ does not impact $U$ with its fluctuations, given the other prepared variables.

If we further assume that the causal graph is sparse, a common assumption in causal discovery literature [19, 79], we can prune more edges by using a *sparse regression* algorithm. This will drive very small coefficients, likely to be an artifact of data availability rather than indicative of causality, to zero. This process requires no user input and typically prunes the vast majority of edges, since most prepared variables are unrelated.

**Algorithm 1** Pruning incoming edges to variable $U$.

```
1:  function Prune(prep_table, U)
2:      X ← prep_table \ U
3:      X, scale ← StandardScaleColumns(X)
4:      coefs ← LASSO(X, U)
5:      unscaled_coefs ← scale.Unscale(coefs)
6:      candidates ← []
7:      for U' in X do
8:          if unscaled_coefs(U') == 0 then
9:              ESM(U' → U) = −1
10:         else
11:             candidates.append(U')
12:     return candidates
```

**Algorithm 2** Exploring candidate causes for variable $U$.

```
1: function ExploreCandidateCauses(prep_table, U)
2:     candidates ← Prune(prep_table, U)
3:     res ← []
4:     for U' in candidates do
5:         slope, p-value ← OLSLinearRegression(U', U)
6:         res.append((U', slope, p-value))
7:     return res.sortby(p-value, ascending)
```

In particular, we use LASSO [81], as presented in Algorithm 1. First, we separate $U$ (the variable for which we want to prune incoming edges) from the rest of the prepared variables (line 2), which we then normalize by subtracting the mean and scaling to unit variance (line 3). We then apply LASSO (line 4) and undo our earlier scaling (line 5). For every variable $U'$ not selected by the sparse regression (lines 7-8), we mark the edge $U' \rightarrow U$ as rejected (line 9). At the same time, for every variable that *is* selected by LASSO, we keep track of it (lines 6, 11) and return it (line 14).

### 6.2 Phase I - Finding a "Root Cause"

We are now ready to involve the user. Recall that the user wants to find $ATE(T, Y)$ - but while $Y$ has been selected, $T$ will be pinpointed in this phase. The directed path from $T$ to $Y$ may be long. We help the user traverse it "backwards" through the function ExploreCandidateCauses($U$), which suggests likely causes for a prepared variable $U$. Algorithm 2 obtains candidates using Algorithm 1 (line 2) and applies an ordinary least-squares (OLS) linear regression to each candidate edge (line 5), recording the slope and p-value (line 6). The results are returned sorted by increasing p-value (line 8), placing the candidates with the most reliable relationships to $U$ near the top, as indicated by a low p-value.

The user can then inspect each returned candidate $U'$ and decide whether to include $U' \rightarrow U$ in the causal graph (Accept($U'$,$U$)) or not (Reject($U'$,$U$)). Accepting $U' \rightarrow U$ will automatically reject $U \rightarrow U'$. The user can also collectively reject all undecided edges to $U$, by calling RejectUndecidedIncoming($U$). By iteratively leveraging ExploreCandidateCauses($U$), the user can arrive at a satisfactory "root cause" variable $T$, concluding Phase I.

## 6.3 Phase II - Finding Confounders

Having found which ATE to calculate ($ATE(T, Y)$), the user must now focus on adjusting for a sufficient set of confounders. This set depends on the "correct" causal graph (i.e. the user's mental model of the system), so we cannot directly assess the user's progress. We can, however, measure the user's progress in recovering the regions of the causal graph that *could* include confounders. We define:

**Definition** 9 (Exploration Score). *For a causal graph $G$, the exploration score is the fraction of decided edges among edges that (1) are not self-edges, (2) touch at least one node in $G$; that is,*

$$ES = \frac{|\{U_i \to U_j \mid i \neq j \wedge (U_i \in G \vee U_j \in G) \wedge S(U_i \to U_j) \neq 0\}|}{|\{U_i \to U_j \mid i \neq j \wedge (U_i \in G \vee U_j \in G)\}|}$$

Phase II is concluded when the exploration score reaches 1: any possible edges involving variables outside the graph have been rejected, whereas all possible edges where both endpoints are in the graph have been decided. Therefore, the graph fully reflects the user's mental model of the system around the variables $T$ and $Y$, so we can use it to derive the set of confounders to adjust for and then calculate the ATE. This is achieved by calling GetATE($T$, $Y$).

To increase the exploration score efficiently, we introduce the function SuggestNextExploration, which returns the prepared variable $U_s$ in the current graph $G$ that has the most undecided incoming edges. If all incoming edges are decided for variables in $G$, it returns the prepared variable $U_s$ outside $G$ with the most undecided incoming edges from nodes in $G$. By then calling the function ExploreCandidateCauses($U_s$), the user will be presented with as many undecided edges relevant to the exploration score calculation as possible, maximizing their decision-making efficiency.

## 6.4 Computational Complexity

The computational complexity of Algorithm 1 is derived from that of LASSO [81] . If the threshold for convergence of the objective function is $\sigma$, a LASSO solution can be found in $O(|C||\mathcal{U}|\sigma^{-1/2})$ [14, 97], where $C$ is the set of causal units and $\mathcal{U}$ is the set of prepared variables. The computational complexity of Algorithm 2 is derived from those of LASSO [81] and ordinary least squares linear regression, so it will be $O(|C||\mathcal{U}|\sigma^{-1/2} + m|C||\mathcal{U}|^2 + m|\mathcal{U}|^3)$, where $m$ is the number of candidates returned by Algorithm 1.

# 7 PROTOTYPE SYSTEM

We implemented our framework from Figure 3 in Sawmill, using ~2800 lines of Python. For log parsing, we used Drain [37], based on an implementation by the authors of LogBERT [31]. Drain uses a fixed-depth parse tree in an online streaming manner that requires a single pass over the log. For variable pruning, we used LASSO from scikit-learn [62]. For estimating ATEs in GetATE we used DoWhy [73], an open-source Python library for causal reasoning. In particular, we used DoWhy's "backdoor.linear_regression".

# 8 EVALUATION

After presenting our experimental setting (Section 8.1), we evaluate four claims about causal inference on log data using Sawmill:

- **Accuracy**: Sawmill can find accurate causes for problems in real-life logs. This is true even when the causal effect is weak, highly confounded, or found in noisy data. (Section 8.2).
- **Computational Efficiency**: Sawmill is computationally efficient even for large and complicated logs (Section 8.3).
- **Human Efficiency**: Sawmill requires few additional human interactions to arrive at good answers (Section 8.4).
- **Completeness**: Every component of Sawmill is important for the final result and works well. (Section 8.5).

## 8.1 Experimental Setting

*8.1.1 Environment.* All experiments were run on a machine equipped with two 20-core 2.10 GHz Intel Xeon Gold 6230 CPUs [42] and 256 GiB of memory, running Linux 5.19.12.

*8.1.2 Metrics.* We use the following metrics for each claim:

- **Accuracy**: We measure accuracy in two ways. First, when ranking candidate causes for the user, we measure the quality of the ranking using the standard Mean Reciprocal Rank (MRR) measure (larger is better). Second, when we have ground-truth ATE values, we measure how close to them the ATE values returned by Sawmill are by calculating the ATE Error: $|\frac{ATE_{true} - ATE_{computed}}{ATE_{true}}|$ (lower is better).
- **Computational Efficiency**: We measure computational efficiency by simple execution time (lower is better).
- **Human Efficiency**: We measure human efficiency by counting how many function invocations are needed in order for each system to arrive at an answer (lower is better).
- **Completeness**: We introduce additional metrics as needed per experiment in Section 8.5.

*8.1.3 Baselines.* We argue that causality is indispensable for correctly estimating the effect of a treatment variable $T$ to an outcome variable $Y$, where both $T$ and $Y$ originate in logs. We will compare the results we derive from Sawmill's causal analysis to two alternative ways to estimate an effect from the prepared table:

- Regression: We call Regress, which runs a multivariate regression of $Y$ on the other prepared variables, without using causality. We ignore regressors with zero variance and normalize the rest to zero mean and unit variance. This is equivalent to considering all prepared variables beyond $T$ as confounders (see Definition 2). The ATE($T$, $Y$) is then the regression coefficient of $T$, divided by the original standard deviation $\sigma_T$. The MRR refers to the rank of $T$ among other regressors in order of decreasing regression coefficient, ignoring regressors with a p-value over 0.05.
- AskGPT: We call GPT-ExploreCandidateCauses, which provides GPT-4 Turbo (gpt-4-1106-preview) with the tags of all prepared variables and asks it to rank the (up to 25) most likely causes for $Y$. Recent work has shown that large language models can be effective causal advisors based on variable semantics [46], but log variables can be esoteric, so we also provide data for the first 10 causal units in our prompt, available with our code [4]. We treat these suggestions as a replacement for ExploreCandidateCauses and calculate the MRR based on this ranking. The ATE($T$, $Y$) is then calculated by calling GetATE on the graph obtained by the intersection of the graph created for Sawmill and the edges actually suggested by AskGPT.

| Dataset | | Size | | Parsed | Prepared |
|---|---|---|---|---|---|
| | | Lines | Bytes | Variables | Variables |
| PostgreSQL | | 507,648 | 20,587,286 | 71 | 172 |
| Proprietary | | 1,223,000 | 237,048,470 | 121 | 120 |
| XYZ | $V$=10 | 1,000,000 | 62,458,855 | 12 | 21 |
| | $V$=100 | 1,000,000 | 64,791,158 | 102 | 201 |
| | $V$=1000 | 1,000,000 | 65,943,179 | 1002 | 2001 |

**Table 1: Statistics about our main datasets.**

| Parameter | Values | Parameter | Values |
|---|---|---|---|
| work_mem | 128, 256 | max_parallel_workers | 1,2 |
| seq_page_cost | 1, 1000 | maintenance_work_mem | 32768, 65536 |
| random_page_cost | 4, 1000 | effective_cache_size | 262144, 524288 |

**Table 2: Parameter settings forPostgreSQL.**

*8.1.4 Datasets.* Open-source collections of logs exist [38], but we also need the *ground-truth causal effects* in the log-producing system, which are generally not available. Our three main datasets, summarized in Table 1, range from instrumenting a real system, to injecting a known causal effect into a real-world log, to generating synthetic log data for stress-testing. This strategy was also used by other causality-in-systems works, e.g., CausalSim [9]. We provide example snippets of each dataset online [4].

**PostgreSQL (Real-world):** This dataset emulates the situation of a database administrator who wants to determine which parameter is causing queries to be slow. We set up PostgreSQL 14 on an t2.2xlarge instance on AWS EC2 [5] and configured it to log the latency of each query. We then loaded the TPC-DS benchmark [63] for scale factor 1 and sequentially issued the TPC-DS queries, excluding 4 long-running ones (queries 1, 4, 11 and 74). We ran this workload 128 times, twice for each combination of the parameter settings in Table 2. These parameters both impact query latency and do not require a Postgres service restart to reset. We used a new connection per workload run, emulating different customers with different configurations. To create PostgreSQL, we then selected a subset of the runs as follows: for parameter combinations where `work_mem`=256 and `max_parallel_workers`=1, or `work_mem`=128 and `max_parallel_workers`=2, we added both workload runs in the dataset. For the rest of the parameter combinations, we only added a single run. This led to a dataset with confounding: `work_mem` affects both query latency and the value of `max_parallel_workers`. This is representative of a broader class of confounding related to resource sharing that can come up often in the real world. For this dataset, we can only measure accuracy in terms of MRR, since we know the identity of causes, but not their strength.

**Proprietary (Semi-synthetic):** We wanted to process a log from another complicated piece of software that was as realistic as possible, but for which we knew the ground truth. We thus created a "semi-synthetic" log. We obtained a real log for an HTTP-based client/server application from a large company, with different messages in the log generated by different parts of the software stack. Clients in this application periodically contact the server to perform operations. We kept the real-world syntax of this file but synthetically introduced a bug. We generated logs for 1,000 "users", to whom we assigned unique IDs, included in each of their log messages. We designated a fraction of the users $F$ as "faulty". For each non-faulty user, we generated a log identical to the original, except that HTTP responses have probability $p_n = 10\%$ of reporting error code 401 instead of success code 200. There are 36 such messages in the original 1223-line log. For each faulty user, we equivalently

| Dataset | | | Sawmill | Regression | AskGPT |
|---|---|---|---|---|---|
| | | | MRR | MRR | MRR |
| PostgreSQL | | | **0.5667** | 0.0476 | 0.4815 |
| Proprietary | $F$=0.5 | $p_f$=1.0 | **1.0000** | 1.0000 | 0.0000 |
| | | $p_f$=0.5 | **1.0000** | 1.0000 | 0.3333 |
| | | $p_f$=0.2 | **1.0000** | 1.0000 | 1.0000 |
| | $F$=0.1 | $p_f$=1.0 | **1.0000** | 1.0000 | 0.0000 |
| | | $p_f$=0.5 | **1.0000** | 1.0000 | 0.0000 |
| | | $p_f$=0.2 | **1.0000** | 1.0000 | 0.0000 |
| | $F$=0.01 | $p_f$=1.0 | **1.0000** | 1.0000 | 0.0714 |
| | | $p_f$=0.5 | **1.0000** | 0.0667 | 0.0000 |
| | | $p_f$=0.2 | **1.0000** | 0.0667 | 0.0000 |
| XYZ | $V$=10 | $R$=1 | **0.6667** | 0.6667 | 0.4007 |
| | | $R$=5 | 0.6111 | 0.5556 | **0.6667** |
| | | $R$=10 | **0.6667** | 0.5833 | 0.0664 |
| | $V$=100 | $R$=1 | **0.6667** | 0.5476 | 0.5000 |
| | | $R$=5 | **0.6667** | 0.5370 | 0.0000 |
| | | $R$=10 | 0.3889 | **0.6667** | 0.5000 |
| | $V$=1000 | $R$=1 | **0.6667** | 0.0000 | 0.1667 |
| | | $R$=5 | 0.6667 | 0.0000 | **0.8333** |
| | | $R$=10 | **0.6667** | 0.0000 | 0.1667 |
| **Mean on Proprietary** | | | **1.0000** | 0.7926 | 0.1561 |
| **Mean on XYZ** | | | **0.6296** | 0.3952 | 0.3667 |
| **Mean** | | | **0.8018** | 0.5651 | 0.2730 |

**Table 3: Sawmill maximizes mean MRR across our datasets.**

defined a probability $p_f > p_n$ and we indicated their OS as iOS 15.0 in the log message that reports it, compared to iOS 14.3 in the original log. We generated 9 experimental scenarios by setting $F$ to 50%, 10% or 1%, while $p_f$ is set to 100%, 50% or 20%. For this dataset, we can measure accuracy both in terms of ATE error, since we can calculate the true ATE based on $F$ and $p_f$, and in terms of MRR, since we know that the correct cause is the OS version.

**XYZ (Synthetic):** Finally, we created a family of synthetic logs to evaluate our framework's ability to recover and adjust for confounding. We wanted a synthetic log so we could test how well Sawmill can work even in the face of a harder-to-discern causal effect. We did that by carefully adjusting the level of noise and the number of irrelevant variables. Each XYZ log contains 1000 log messages for each of 1000 machines. Each message includes an artificial timestamp, a machine identifier and one of $V$ variables chosen uniformly at random. We ensure every variable is reported at least once. While the value of most variables is drawn uniformly at random between 0 and 100 each time the variable is reported, the values of 3 special variables ($x$, $y$ and $z$) are not:

- $z$ is drawn uniformly between 0 and 10 once per machine.
- $x$ is equal to $x_m + z$, with added Gaussian noise with $\sigma = R$. $x_m$ is drawn once per machine, uniformly at random.
- $y$ is equal to $2x + 3z$, with added Gaussian noise with $\sigma = R$.

As such, ATE($x$,$y$)=2 (due to $x_m$) after adjusting for confounding by $z$. We generate 9 scenarios by setting $V$ to 10, 100 or 1000, while $R$ is set to 1, 5 or 10. For this dataset, we can measure accuracy both in terms of ATE error, since we know the true ATE is 2, and in terms of MRR, since we know the correct causal relationships.

## 8.2 Accuracy

**Summary**

Sawmill's mean MRR is 41.89% higher than that of the next best baseline (Regression), while Sawmill's mean ATE Error is 10.99% lower than that of the next best baseline (Regression).

*8.2.1 Accuracy on PostgreSQL.* We begin with a case of confounding in real-world logs, using PostgreSQL. We executed the series of calls shown below. After calling Parse, Drain had incorrectly mapped the values of all 6 parameters we adjust in this dataset to a single variable `parameter_value`, making the call to Separate necessary. After it, Sawmill instead created a variable *P*`:mean` for each parameter *P*. Based on how the dataset was generated (see Section 8.1.4), we expected to see a **detrimental** impact of increasing `max_parallel_workers:mean` on `duration:mean` (i.e. a positive ATE). However, a system administrator suspicious of this paradoxical conclusion should be able to leverage Sawmill to unearth and adjust for the confounding by `work_mem:mean`.

**Sequence of Sawmill calls for PostgreSQL**

(1) Parse()
(2) Separate(parameter_value)
(3) SetCausalUnit(connectionID)
(4) Prepare()
(5) ExploreCandidateCauses(duration:mean)
(6) Accept(work_mem:mean → duration:mean)
(7) Accept(max_parallel_workers:mean → duration:mean)
(8) ExploreCandidateCauses(max_parallel_workers:mean)
(9) Accept(work_mem:mean → max_parallel_workers:mean)
(10) GetATE(max_parallel_workers:mean, duration:mean)

In the values returned by call (5), the variables `work_mem:mean` and `max_parallel_workers:mean` were listed second and fifth among the 11 returned candidate causes, respectively. In the values returned by call (8), `work_mem:mean` was returned as the top candidate cause. As shown in Table 3, these results gave rise to an MRR of 0.5667. Accepting the 3 edges mentioned above let Sawmill adjust for the confounding by `work_mem:mean`, taking the ATE of interest to −156.47. In other words, giving PostgreSQL a dose of `max_parallel_workers:mean` yields −156.47 units of `duration:mean` - **more parallelism yields a lower query duration**. However, had we not used causal reasoning to adjust for confounding, we would have obtained a confusing and incorrect value of 5127.65, as suspected: a dose of `max_parallel_workers:mean` would appear to yield 5127.65 units of `duration:mean` - **more parallelism would have meant a much longer duration**!

Sawmill therefore helped us detect the confounding relationship and adjust for it, without getting sidetracked by the remaining variables. As such, we successfully avoided the bias in the original data. Regression and AskGPT ranked the variables of interest much lower, leading to an MRR of 0.0476 and 0.4815, respectively.

*8.2.2 Accuracy on Proprietary.* With Proprietary, we tested whether Sawmill can handle measurement noise and many irrelevant variables in the log. For each combination of $F$ and $p_f$, we performed the sequence of calls shown below. After calling Prepare, the prepared table included a variable tagged as `version:mean` listing the (mean) iOS version per user. It also included several prepared variables, from different log templates, representing the *mean HTTP return code*, an indicator of the mix of 200 and 401 values (the only two return codes present). We focused on one such variable, tagged by the system as `code:mean`, aiming to recover `version:mean` as a high-ranked candidate cause for it.

| Dataset | | | True ATE | Sawmill ATE | Regression ATE | AskGPT ATE |
|---|---|---|---|---|---|---|
| Proprietary | $F$=0.5 | $p_f$=1.0 | 258.43 | **257.47** | 273.01 | 0.00 |
| | | $p_f$=0.5 | 114.86 | **112.66** | 118.04 | 112.66 |
| | | $p_f$=0.2 | 28.71 | **27.28** | 25.94 | 27.28 |
| | $F$=0.1 | $p_f$=1.0 | 258.43 | **258.64** | 256.01 | 0.00 |
| | | $p_f$=0.5 | 114.86 | 121.45 | **119.38** | 0.00 |
| | | $p_f$=0.2 | 28.71 | **33.98** | 35.30 | 0.00 |
| | $F$=0.01 | $p_f$=1.0 | 258.43 | **258.57** | 264.50 | 258.57 |
| | | $p_f$=0.5 | 114.86 | **85.66** | 84.79 | 0.0 |
| | | $p_f$=0.2 | 28.71 | **42.64** | 45.18 | 0.0 |
| XYZ | $V$=10 | $R$=1 | 2.00 | **2.00** | 2.00 | 2.00 |
| | | $R$=5 | 2.00 | 2.11 | **2.10** | 2.11 |
| | | $R$=10 | 2.00 | 1.97 | **1.98** | 1.97 |
| | $V$=100 | $R$=1 | 2.00 | **1.96** | 1.95 | 2.36 |
| | | $R$=5 | 2.00 | **1.60** | 1.58 | 0.00 |
| | | $R$=10 | 2.00 | 0.87 | 0.86 | **0.97** |
| | $V$=1000 | $R$=1 | 2.00 | **1.78** | 0.37 | 0.00 |
| | | $R$=5 | 2.00 | **0.62** | -1.61 | 0.62 |
| | | $R$=10 | 2.00 | 0.12 | **0.35** | 0.00 |
| **Mean % Error on Proprietary** | | | | **11.72%** | 14.64% | 67.44% |
| **Mean % Error on XYZ** | | | | **28.83%** | 47.88% | 49.50% |
| **Mean % Error** | | | | **20.27%** | 31.26% | 58.47% |

**Table 4: Sawmill minimizes ATE error across our datasets.**

**Sequence of Sawmill calls for Proprietary**

(1) Parse()
(2) SetCausalUnit(User)
(3) Prepare()
(4) ExploreCandidateCauses(code:mean)
(5) Accept(version:mean → code:mean)
(6) GetATE(version:mean, code:mean)

Per Table 3, Sawmill successfully identified `version:mean` as the *top-ranked* candidate cause of `code:mean` across difficulty settings, leading to an MRR of 1. Regression was similarly successful in many cases, but failed in some where the faulty users were only 1% of the total (i.e. $F = 0.01$), for a mean MRR of 0.7926. AskGPT's result varied wildly for each scenario, from ranking `version:mean` first to not ranking it at all, for a mean MRR of 0.1561. In terms of ATE accuracy, Sawmill was able to estimate a highly accurate ATE as shown in Table 4, with a mean error of only 11.72%. Regression followed closely behind Sawmill, achieving a mean ATE Error of 14.64%. AskGPT achieved the same ATE as Sawmill whenever it also suggested `version:mean` as a candidate cause, but that only happened for 3 scenarios, leading to a mean ATE Error of 67.44%.

*8.2.3 Accuracy on XYZ.* Finally, we move to detecting and adjusting for confounding in XYZ. For each scenario, we performed the sequence of calls shown below, using the convenient tags `x:mean`, `y:mean` and `z:mean` given to the variables of interest by Sawmill.

**Sequence of Sawmill calls for XYZ**

(1) Parse()
(2) SetCausalUnit(machineID)
(3) Prepare()
(4) ExploreCandidateCauses(y:mean)
(5) Accept(x:mean → y:mean)
(6) Accept(z:mean → y:mean)
(7) ExploreCandidateCauses(x:mean)
(8) Accept(z:mean → x:mean)
(9) GetATE(x:mean, y:mean)

In the values returned by call (4), `x:mean` and `z:mean` were consistently ranked as the top two candidate causes of `y:mean` in 8 out of 9 scenarios, even when $V$ and $R$ were high. In the values returned by call (7), `y:mean` consistently took the top spot, pushing `z:mean` (which interested us) to second in 8 out of 9 cases. As shown

| Dataset | | | Parse (s) | Prepare (s) | ExploreCandidateCauses (s) | **Total Time (s)** | Total Time over Log Size (s/MiB) |
|---|---|---|---|---|---|---|---|
| PostgreSQL | | | 37.65 | 4.41 | 4.85 | 46.91 | 2.39 |
| Proprietary | $F$=0.5 | $p_f$=1.0 | 189.19 | 48.58 | 2.62 | 240.39 | 1.06 |
| | | $p_f$=0.5 | 189.51 | 48.83 | 3.20 | 241.54 | 1.07 |
| | | $p_f$=0.2 | 195.58 | 49.32 | 3.18 | 248.08 | 1.10 |
| | $F$=0.1 | $p_f$=1.0 | 194.50 | 49.11 | 3.22 | 246.83 | 1.09 |
| | | $p_f$=0.5 | 189.53 | 49.18 | 3.24 | 241.95 | 1.07 |
| | | $p_f$=0.2 | 189.58 | 49.50 | 3.28 | 242.36 | 1.07 |
| | $F$=0.01 | $p_f$=1.0 | 190.70 | 49.99 | 3.31 | 244.00 | 1.08 |
| | | $p_f$=0.5 | 187.09 | 49.47 | 3.38 | 239.94 | 1.06 |
| | | $p_f$=0.2 | 191.18 | 49.30 | 3.43 | 243.91 | 1.08 |
| XYZ | $V$=10 | $R$=1 | 72.33 | 4.26 | 2.40 | 78.99 | 1.32 |
| | | $R$=5 | 80.04 | 5.27 | 2.98 | 88.29 | 1.48 |
| | | $R$=10 | 79.69 | 4.70 | 3.09 | 87.48 | 1.47 |
| | $V$=100 | $R$=1 | 145.67 | 33.24 | 3.59 | 182.50 | 2.95 |
| | | $R$=5 | 144.14 | 33.87 | 3.87 | 181.88 | 2.94 |
| | | $R$=10 | 144.67 | 33.79 | 3.97 | 182.43 | 2.95 |
| | $V$=1000 | $R$=1 | 853.14 | 326.83 | 8.38 | 1188.35 | 18.90 |
| | | $R$=5 | 822.43 | 334.47 | 8.91 | 1165.81 | 18.54 |
| | | $R$=10 | 849.08 | 335.59 | 15.09 | 1199.76 | 19.08 |
| **Mean** | | | 260.30 | 82.09 | 4.53 | 346.92 | 4.30 |
| **Breakdown (%)** | | | 75.03% | 23.66% | 1.30% | | |

**Table 5: Sawmill is computationally efficient.**

in Table 3, these rankings gave rise to an average MRR of 0.6296, compared to 0.3952 for Regression and 0.3667 for AskGPT.

Given the 3 detected edges, Sawmill adjusted for `z:mean`'s confounding, recovering the ATE of `x:mean` on `y:mean` with a mean error of 28.83%, compared to a mean ATE error of 47.88% for Regression. AskGPT achieved the same ATE as Sawmill whenever it identified the correct candidate causes, but it failed to do so in three scenarios, leading to a mean ATE error of 49.50%. As expected, Sawmill's ATE error varied with $V$ and $R$. Still, Sawmill recovered ATEs accurately even in the presence of large numbers of irrelevant variables, fewer observations of relevant ones, and large amounts of noise, leading to an effective consideration of confounding.

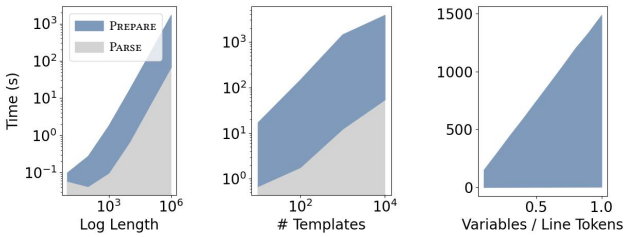
## 8.3 Computational Efficiency

**Summary**

Sawmill only requires an average of 346.92 s to go from a log to an ATE, 75.03% of which is required for Parse. Sawmill's performance scales linearly with log complexity.

*8.3.1 Computational Efficiency on Main Datasets.* Table 5 presents the computational efficiency of Sawmill on each of our datasets. Sawmill only required an average of 346.92 s to go from a log to an ATE, corresponding to 4.30 seconds per MiB of log data. Most of the processing time was taken up by parsing (75.03%), which is a standard step for log processing. Importantly, our use of exploration-based causal discovery to reap the accuracy gains of Section 8.2 only represented 1.30% of the required processing time on average. Note that the 9 scenarios for XYZ produced logs of roughly the same size (see Table 1), but with different complexity in terms of the number of variables $V$. This increasing complexity increased the runtime of Parse and Prepare linearly with $V$, explaining the significantly increased total time over log size when $V = 1000$.

*8.3.2 Computational Efficiency Scaling.* We now evaluate the scalability of our framework. We used synthetic datasets defined based on 4 parameters: $L$, $S$, $V$ and $C$. $L$ represented the length of the log. Each log message included $2 + V + C$ space-separated tokens. The first token was a line ID of the form `line_`$i$, where $i$ was the index of the message. The second token was a string token of the form `s_`$j$, where $j$ was a uniformly random integer between 1 and $S$, inclusive. The next $V$ tokens were numerical variables, each of which took the value $i$. The last $C$ tokens were constants equal to 0.




**(a) Log Length.** **(b) Log templates.** **(c) Log variables.**

**Figure 4: The performance scaling of our framework.**

**Log Length** We fixed $S = 10$, $V = 1$ and $C = 10$, and swept $L$ over powers of 10 between $10^1$ and $10^6$, inclusive. We parsed the log using a regular expression for `line_ID` and set the Drain similarity threshold so as to get 10 log templates, one per value of the string token. We then called Prepare(`line_ID`). Figure 4a shows the time for Parse and Prepare (note the logarithmic axes). Prepare dominated, but scaled linearly (i.e. optimally) with $L$.

**Number of distinct Templates** Next, we fixed $L = 10^4$, $V = 1$ and $C = 10$, and swept $S$ over powers of 10 between $10^1$ and $10^4$, inclusive. We again parsed the log using a similarity threshold that led to one log template per value of the string token - i.e. $S$ distinct log templates. We then called Prepare(`line_ID`). In Figure 4b, we again see that Prepare dominated but scaled linearly with the number of templates, confirming our framework's scalability.

**Fraction of tokens that are variables.** Finally, we fixed $L = 10^4$ and $S = 10$, and swept $V$ over multiples of 10 between 10 and 100 inclusive, while setting $C = 100 - V$. We adjusted the similarity threshold to still lead to one log template for each value of the string token for each of the experiment iterations - i.e. $S$ distinct log templates. We then called Prepare(`line_ID`). The fraction of parsed variables over line tokens varied from 11/102 when $V = 10$, to 101/102 when $V = 100$. In Figure 4c (note the linear scale), we see that Prepare dominated and grew linearly with the fraction of tokens that were variables, since additional variables in the parsed and prepared tables translated to linearly more work.

## 8.4 Human Efficiency

**Summary**

Sawmill only requires 6-10 user interactions to leverage causality, up to 5 more than the best baseline, Regression.

As shown in Table 6, all methods required at least 3 calls to get from any log to the prepared table: a call to parse, a call to SetCausalUnit, and a call to Prepare. For PostgreSQL, each method also required a call to Separate to specify that each tweaked parameter should be considered a separate parsed variable. Beyond these calls, Sawmill required the additional calls listed in Section 8.2 and AskGPT required the same number of calls as Sawmill.

| Dataset | System | Parse | Separate | SetCausalUnit | Prepare | Explore CandidateCauses | Accept | GetATE | Regress | GPT-Explore CandidateCauses | Total |
|---|---|---|---|---|---|---|---|---|---|---|---|
| PostgreSQL | Sawmill | 1 | 1 | 1 | 1 | 2 | 3 | 1 | 0 | 0 | 10 |
| | Regression | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | **5** |
| | AskGPT | 1 | 1 | 1 | 1 | 0 | 3 | 1 | 0 | 2 | 10 |
| Proprietary | Sawmill | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 6 |
| | Regression | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | **4** |
| | AskGPT | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 6 |
| XYZ | Sawmill | 1 | 0 | 1 | 1 | 2 | 3 | 1 | 0 | 0 | 9 |
| | Regression | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | **4** |
| | AskGPT | 1 | 0 | 1 | 1 | 0 | 3 | 1 | 0 | 2 | 9 |

**Table 6: Sawmill requires few extra function invocations.**

| Dataset | Variable Name | Variable Tag |
|---|---|---|
| PostgreSQL | 9989ce8d_12+mean | duration:mean |
| | da4032f7_15+mean | work_mem:mean |
| | 9d2f8f03_15+mean | max_parallel_workers:mean |
| Proprietary | 73b16c0a_196+mean | code:mean |
| | 30731d4c_11+mean | version:mean |
| XYZ | d1a33a13_4+mean | x:mean |
| | caa5dc5d_4+mean | y:mean |
| | de74037b_4+mean | z:mean |

**Table 7: Sawmill's tagging improves variable interpretability.**

Regression needed 2-5 fewer interactions than Sawmill's causal analysis. However, our goal is to help humans allocate actual engineering resources: an incorrect hypothesis about what caused a failure can mean many hours of wasted engineering time to consider and dismiss the hypothesis. We think a few extra steps — taking an average of just 4.53 total extra seconds to execute (see Table 5) and most of which involved a single click each — were worth the effort for an answer that yielded an MRR that is 41.89% higher, and cut the ATE error rate by more than a third.

## 8.5 Completeness

**Summary**

Our solutions to each of the 3 challenges add value to our framework and are effective.

We will next evaluate the individual components of our framework, in terms of their *value* as well as their *effectiveness*.

*8.5.1 Deriving the Schema (Challenge A).* We first turn to our tagging approach from Section 4.2. As shown in Table 7, tagging adds *value* to our framework by giving users more interpretable variable tags to reason about. In terms of *effectiveness*, Figure 5 presents the ability of our approach to generate tags for the parsed variables of our datasets. We have excluded XYZ from this experiment, since we have full control over its format and can always ensure the success of the "preceding 3 tokens" method. The sequential consultation of the 3 sources presented in Section 4.2 led to tags for 88.65% of the prepared variables, with GPT-3.5-Turbo tagging 58.68% of the prepared variables alone. The variables that no method could tag were assigned their system-generated variable names as tags.

*8.5.2 Distilling the Data (Challenge B).* We first show that selecting only one prepared variable per base variable provides *value*, by reducing the size of the prepared table and therefore the downstream cost of processing it. Indeed, as Table 8 shows, such selection reduced the size of the prepared table by an average of 51.27%.

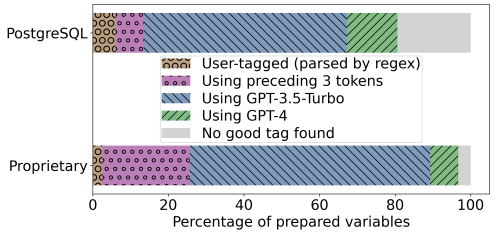



**Figure 5: Effectiveness of our tagging approach.**

| Dataset | | Prepared Table (bytes) Sawmill | Prepared Table (bytes) No selection | Reduction in Size (%) |
|---|---|---|---|---|
| PostgreSQL | | 139,325 | 236,278 | 41.03% |
| Proprietary | | 974,772 | 2,903,579 | 66.43% |
| XYZ | $V$=10 | 183,109 | 359,481 | 49.06% |
| | $V$=100 | 1,627,072 | 3,246,509 | 49.88% |
| | $V$=1000 | 16,066,682 | 32,116,727 | 49.97% |
| **Mean** | | | | **51.27%** |

**Table 8: Sawmill's aggregate selection saves memory.**

| Dataset | | | Lowest ATE Error (%) | Sawmill ATE Error (%) | Highest ATE Error (%) | % of gap closed |
|---|---|---|---|---|---|---|
| Proprietary | $F = 0.01$ | $p_f = 0.5$ | 5.81 | 25.42 | 100.00 | 79.18 |
| XYZ | $V = 100$ | $R = 5$ | 15.16 | 20.10 | 60.98 | 89.22 |
| | | $R = 10$ | 53.27 | 56.42 | 95.73 | 92.58 |

**Table 9: Sawmill's chosen aggregates do not minimize downstream ATE Error for only 3/18 scenarios.**

Next, we will show that Sawmill's approach to this selection is *effective*, justifying the claim in Section 5.3 that solving Problem 2b is an acceptable practical approximation to solving Problem 2.

For the datasets with known ATEs (Proprietary and XYZ), we used each combination of aggregation functions from Section 5.2 and calculated the downstream ATE error. That is, for Proprietary, we tried each combination of aggregating `code` and `version` using one of [*max*, *min*, *mean*] for each of them, and then calculated GetATE(`version`, `code`). For XYZ, we tried each combination of aggregating `x`, `y` and `z` using one of [*max*, *min*, *mean*] for each of them, and then calculated GetATE(`x`, `y`) adjusted for `z`.

For each dataset, for each difficulty scenario, some combination of aggregates minimized downstream ATE error. Among the 9 experimental scenarios in Proprietary, Sawmill's chosen combination of aggregates was the ATE Error-minimizing combination for 8 of them. For the final one, presented in Table 9, Sawmill's chosen aggregates led to an ATE Error of 25.42%, while the best combination achieved 5.81%. However, the chosen combination still closed 79.18% of the gap between the worst and best achievable ATE Error. For XYZ, Sawmill achieved the lowest ATE Error for 7 out of 9 scenarios, with the remaining two also presented in Table 9. While not optimal in these 2 cases, Sawmill closed on average over 90% of the gap between the worst and best possible choices. Over all 18 scenarios in the two datasets, the combination of aggregates chosen by Sawmill closed on average 97.83% of the gap between the two extremes. That is, choosing an aggregation function based on the empirical entropy of the data, with no knowledge of the downstream causal question, still led to an ATE almost as accurate as if the aggregates were chosen to minimize this specific ATE error.

| Dataset | | PC [79] | FCI [80] | LiNGAM [76] | GIN [89] | GRaSP [47] | GES [18] | Exact Search [78] | GPT-4 [46, 58] |
|---|---|---|---|---|---|---|---|---|---|
| PostgreSQL | | ✓ | ✓ | ✗ | ▲ | ✓ | ✓ | ✗ | ▲ |
| Proprietary | | ✓ | ✓ | ✗ | ▲ | ● | ● | ✗ | ▲ |
| XYZ | $V = 10$ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ▲ | ▲ |
| | $V = 100$ | ● | ● | ✓ | ▲ | ▲ | ▲ | ▲ | ▲ |
| | $V = 1000$ | ▲ | ▲ | ✗ | ▲ | ▲ | ▲ | ▲ | ▲ |

**Table 10: Existing causal discovery approaches. (✓ (●): non-empty (empty) graph; ▲: 30-minute timeout; ✗: error)**

*8.5.3 Obtaining the Causal Model (Challenge C).* We finally turn to exploration-based causal discovery. The *value* of our approach was already covered earlier, since it provided a better accuracy with minimal computational and human overhead. In terms of the *effectiveness* of our approach, it let us recover a useful causal graph, unlike existing causal discovery approaches. We tested the algorithms in causal-learn [98], a Python translation and extension of Tetrad [64]. We also tried probing GPT-4 [58] with the prompt developed in a previous work to determine causality between two variables [46], adding a third option: no causal relationship in either direction. For each pair of prepared variables, the prompt (available online [4]) included their tags and 3 example values for each.

In Table 10, we report whether a causal graph was produced within 30 minutes. For each algorithm, causal-learn [98] provides several choices of independence tests, scoring functions or search algorithms. We ran all of them on each dataset, but present in Table 10 only the results of the most effective choice for each algorithm: chisq for PC and FCI, kci for GIN, local_score_BDeu for GRaSP and GES, and dp for exact search. We otherwise used the default parameters of each algorithm.

For PostgreSQL, some algorithms produced a graph. However, PC and FCI produced graphs that lacked `duration:mean`, while the graphs of GRaSP and GES did not place `duration:mean`, `work_mem:mean` and `max_parallel_workers:mean` in the same connected component. For Proprietary, PC and FCI produced non-empty graphs - but PC got the edge between `Version:mean` and `Code:mean` backwards, while FCI did not include `Version:mean` in the causal graph at all. For XYZ, several algorithms produced a causal graph when $V = 10$, although none captured all the edges among `x:mean`, `y:mean` and `z:mean`. GES and GIN came closest, by defining latent factors that affected all 3 of the variables. When $V = 100$, only LiNGAM produced a non-empty graph, while for $V = 1000$, none of the algorithms was able to handle the complexity of the problem. Note that, across datasets, GPT-4 was not able to complete all the pairwise causality assessments in time.

# 9 RELATED WORK

## 9.1 Causality and Systems

Diagnosing and anticipating failures are crucial tasks for large system operators. Existing work has looked at formulations including failure classification [10], failing component detection [104], troubleshooting guide suggestion [43], job runtime impact analysis [103] and even risk simulation [87]. Some existing work has also leveraged causality [7, 9, 29, 55, 56], as presented in Section 1. Causal inference has also been used for other data management tasks, including query result and classifier explanation [8, 71], hypothetical reasoning [28], exploratory data analysis [50], data integration [75, 91] and model fairness [100]. However, logs have been under-explored as a source for system understanding, with most past work focusing on outlier detection [27, 31, 70], without working towards a deeper causal understanding of the system. One work does map outliers to their causes [16], but it requires a causal graph, which no work so far has derived from log data. Other works impose additional restrictions, like source code access [93], platform-specificity [99] or fine-grained hardware-supported logging [20]. On the commercial side, platforms like Splunk [1] and Datadog [2] provide log interfaces, but do not address the challenges we identified in Section 1. Datadog can identify some "root causes" for failures [23], but these verdicts appear driven just by temporal relationships.

## 9.2 Causal Discovery

Causal discovery has seen much prior research [30, 76, 80, 83, 86, 95, 102]. Some existing algorithms are "constraint-based", like PC [79], FCI [80], GES [80] and numerous variants of each. Others, like LiNGAM [76], try to estimate the parameters of Functional Causal Models (FCMs). Each algorithm is limited by its assumptions about the data [30], some of which are false for logs - e.g. functional dependencies yield singular correlation matrices, causing LiNGAM to fail. Other algorithms take a text-mining approach based on the variable *names* [33, 34, 39] - relying on such names being informative in the first place. A recent thrust in text-based methods has been the use of large language models like GPT-3 [15] or GPT-4 [58] to assist causal discovery [46, 92]. Despite often being impressive, LLMs still exhibit unpredictable failure modes that hinder their general adoption for causality yet [46]. Moreover, LLMs rely on publicly available textual information to derive their answers, which may be severely limited when dealing with log-derived variables particular to a specific architecture and deployment.

# 10 CONCLUSION AND LIMITATIONS

In this work, we presented a novel framework for processing system logs to make them amenable to causal inference. Our framework addresses the three challenges outlined in Section 1 by (1) leveraging LLMs to tag log-derived variables, (2) cleverly aggregating log contents around a user's chosen causal units, and (3) providing interactive algorithms for obtaining a causal model. Our prototype implementation, Sawmill, accurately and efficiently exposes causal relationships among log variables, in both real-world and synthetic logs, while scaling linearly with log complexity. This work can spur further exploration of applying causality in systems.

Although we addressed the challenges we presented, there are also limitations to our approach. First, the results we obtain are subject to both the *correctness* and the *completeness* of the logged values. Future work will focus on integrating influential missing variables from other sources, like code or system documentation. Second, crafting a realistic causal model requires conceptual inputs beyond just the data; this is why we only provide the user with *candidate* causes, which they need to assess. The level of the user's domain knowledge is therefore central to their ability to derive a high-quality causal model from our framework's suggestions.

# REFERENCES


[1] 2023. https://www.splunk.com/ [Accessed 13-Jun-2023].

[2] 2023. https://www.datadoghq.com/ [Accessed 15-Jun-2023].

[3] 2023. How Splunk Is Parsing Machine Logs With Machine Learning On NVIDIA's Triton and Morpheus. https://www.splunk.com/en_us/blog/it/how-splunk-is-parsing-machine-logs-with-machine-learning-on-nvidia-s-triton-and-morpheus.html [Accessed 15-Jun-2023].

[4] 2024. https://anonymous.4open.science/r/sawmill [Accessed 13-Jan-2024].

[5] 2024. Cloud Compute Capacity - Amazon EC2 - AWS. https://aws.amazon.com/ec2/ [Accessed 09-Jan-2024].

[6] 2024. Models. https://platform.openai.com/docs/models/gpt-3-5 [Accessed 17-Jan-2024].

[7] Pooja Aggarwal, Ajay Gupta, Prateeti Mohapatra, Seema Nagar, Atri Mandal, Qing Wang, and Amit Paradkar. 2020. Localization of operational faults in cloud applications by mining causal dependencies in logs using golden signals. In *International Conference on Service-Oriented Computing*. Springer, 137–149.

[8] Kamran Alipour, Aditya Lahiri, Ehsan Adeli, Babak Salimi, and Michael Pazzani. 2022. Explaining Image Classifiers Using Contrastive Counterfactuals in Generative Latent Spaces. arXiv:2206.05257 [cs.CV]

[9] Abdullah Alomar, Pouya Hamadanian, Arash Nasr-Esfahany, Anish Agarwal, Mohammad Alizadeh, and Devavrat Shah. 2023. {CausalSim}: A Causal Framework for Unbiased {Trace-Driven} Simulation. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 1115–1147.

[10] Behnaz Arzani, Selim Ciraci, Boon Thau Loo, Assaf Schuster, and Geoff Outhred. 2016. Taking the blame game out of data centers operations with netpoirot. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 440–453.

[11] Nicolas Aussel, Yohan Petetin, and Sophie Chabridon. 2018. Improving performances of log mining for anomaly prediction through nlp-based log parsing. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 237–243.

[12] Abhijit V Banerjee, Abhijit Banerjee, and Esther Duflo. 2011. *Poor economics: A radical rethinking of the way to fight global poverty*. Public Affairs.

[13] Elias Bareinboim and Judea Pearl. 2012. Controlling Selection Bias in Causal Inference. In *Proceedings of the Fifteenth International Conference on Artificial Intelligence and Statistics (Proceedings of Machine Learning Research, Vol. 22)*, Neil D. Lawrence and Mark Girolami (Eds.). PMLR, La Palma, Canary Islands, 100–108.

[14] Amir Beck and Marc Teboulle. 2009. A Fast Iterative Shrinkage-Thresholding Algorithm for Linear Inverse Problems. *SIAM Journal on Imaging Sciences* 2, 1 (2009), 183–202. https://doi.org/10.1137/080716542 arXiv:https://doi.org/10.1137/080716542

[15] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[16] Kailash Budhathoki, Lenon Minorics, Patrick Bloebaum, and Dominik Janzing. 2022. Causal structure-based root cause analysis of outliers. In *ICML 2022*. https://www.amazon.science/publications/causal-structure-based-root-cause-analysis-of-outliers

[17] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving fast with software verification. In *NASA Formal Methods Symposium*. Springer, 3–11.

[18] David Maxwell Chickering. 2002. Optimal structure identification with greedy search. *Journal of machine learning research* 3, Nov (2002), 507–554.

[19] Tom Claassen, Joris Mooij, and Tom Heskes. 2013. Learning sparse causal models is not NP-hard. *arXiv preprint arXiv:1309.6824* (2013).

[20] 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.

[21] Hetong Dai, Heng Li, Che-Shao Chen, Weiyi Shang, and Tse-Hsun Chen. 2020. Logram: Efficient Log Parsing Using *n* n-Gram Dictionaries. *IEEE Transactions on Software Engineering* 48, 3 (2020), 879–892.

[22] Ermira Daka and Gordon Fraser. 2014. A survey on unit testing practices and problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*. IEEE, 201–211.

[23] Datadog. 2022. Automated root cause analysis with Watchdog RCA. https://www.datadoghq.com/blog/datadog-watchdog-automated-root-cause-analysis/

[24] Biplob Debnath, Mohiuddin Solaimani, Muhammad Ali Gulzar Gulzar, Nipun Arora, Cristian Lumezanu, Jianwu Xu, Bo Zong, Hui Zhang, Guofei Jiang, and Latifur Khan. 2018. LogLens: A real-time log analysis system. In *2018 IEEE 38th international conference on distributed computing systems (ICDCS)*. IEEE, 1052–1062.

[25] Vijay D'silva, Daniel Kroening, and Georg Weissenbacher. 2008. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27, 7 (2008), 1165–1178.

[26] Min Du and Feifei Li. 2016. Spell: Streaming Parsing of System Event Logs. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*. IEEE, 859–864.

[27] Ilenia Fronza, Alberto Sillitti, Giancarlo Succi, Mikko Terho, and Jelena Vlasenko. 2013. Failure prediction based on log files using random indexing and support vector machines. *Journal of Systems and Software* 86, 1 (2013), 2–11.

[28] Sainyam Galhotra, Amir Gilad, Sudeepa Roy, and Babak Salimi. 2022. HypeR: Hypothetical Reasoning With What-If and How-To !eries Using a Probabilistic Causal Approach. In *Proceedings of the 2022 International Conference on Management of Data*. 1598–1611.

[29] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. 2021. Sage: Practical and Scalable ML-Driven Performance Debugging in Microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 135–151.

[30] Clark Glymour, Kun Zhang, and Peter Spirtes. 2019. Review of causal discovery methods based on graphical models. *Frontiers in genetics* 10 (2019), 524.

[31] Haixuan Guo, Shuhan Yuan, and Xintao Wu. 2021. LogBERT: Log Anomaly Detection via BERT. In *2021 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1–8.

[32] Saurabh Gupta, Tirthak Patel, Christian Engelmann, and Devesh Tiwari. 2017. Failures in large scale systems: long-term measurement, analysis, and implications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[33] Chikara Hashimoto. 2019. Weakly supervised multilingual causality extraction from Wikipedia. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. 2988–2999.

[34] Oktie Hassanzadeh, Debarun Bhattacharjya, Mark Feblowitz, Kavitha Srinivas, Michael Perrone, Shirin Sohrabi, and Michael Katz. 2019. Answering Binary Causal Questions Through Large-Scale Text Mining: An Evaluation Using Cause-Effect Pairs from Human Experts.. In *IJCAI*. 5003–5009.

[35] Pinjia He, Jieming Zhu, Shilin He, Jian Li, and Michael R Lyu. 2016. An evaluation study on log parsing and its use in log mining. In *2016 46th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*. IEEE, 654–661.

[36] Pinjia He, Jieming Zhu, Shilin He, Jian Li, and Michael R Lyu. 2017. Towards automated log parsing for large-scale log data analysis. *IEEE Transactions on Dependable and Secure Computing* 15, 6 (2017), 931–944.

[37] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R. Lyu. 2017. Drain: An Online Log Parsing Approach with Fixed Depth Tree. In *2017 IEEE International Conference on Web Services (ICWS)*. 33–40. https://doi.org/10.1109/ICWS.2017.13

[38] Shilin He, Jieming Zhu, Pinjia He, and Michael R Lyu. 2020. Loghub: A Large Collection of System Log Datasets towards Automated Log Analytics. *arXiv preprint arXiv:2008.06448* (2020).

[39] Stefan Heindorf, Yan Scholten, Henning Wachsmuth, Axel-Cyrille Ngonga Ngomo, and Martin Potthast. 2020. Causenet: Towards a causality graph extracted from the web. In *Proceedings of the 29th ACM international conference on information & knowledge management*. 3023–3030.

[40] Gerard J Holzmann and Doron Peled. 1995. An improvement in formal verification. In *Formal Description Techniques VII: Proceedings of the 7th IFIP WG 6.1 international conference on formal description techniques*. Springer, 197–211.

[41] Shaohan Huang, Yi Liu, Carol Fung, Rong He, Yining Zhao, Hailong Yang, and Zhongzhi Luan. 2020. Paddy: An event log parsing approach using dynamic dictionary. In *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 1–8.

[42] Intel Corporation. 2019. *Intel Xeon Gold 6230 CPU*. Retrieved June 27, 2023 from https://ark.intel.com/content/www/us/en/ark/products/192437/intel-xeon-gold-6230-processor-27-5m-cache-2-10-ghz.html

[43] Jiajun Jiang, Weihai Lu, Junjie Chen, Qingwei Lin, Pu Zhao, Yu Kang, Hongyu Zhang, Yingfei Xiong, Feng Gao, Zhangwei Xu, et al. 2020. How to mitigate the incident? an effective troubleshooting guide recommendation technique for online service systems. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1410–1420.

[44] James Kapinski, Jyotirmoy V Deshmukh, Xiaoqing Jin, Hisahiro Ito, and Ken Butts. 2016. Simulation-based approaches for verification of embedded control systems: An overview of traditional and advanced modeling, testing, and verification techniques. *IEEE Control Systems Magazine* 36, 6 (2016), 45–64.

[45] Baris Kasikci, Benjamin Schubert, Cristiano Pereira, Gilles Pokam, and George Candea. 2015. Failure sketching: A technique for automated root cause diagnosis of in-production failures. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 344–360.

[46] Emre Kıcıman, Robert Ness, Amit Sharma, and Chenhao Tan. 2023. Causal reasoning and large language models: Opening a new frontier for causality. *arXiv preprint arXiv:2305.00050* (2023).

[47] Wai-Yin Lam, Bryan Andrews, and Joseph Ramsey. 2022. Greedy relaxations of the sparsest permutation algorithm. In *Uncertainty in Artificial Intelligence*.

PMLR, 1052–1062.
[48] Lj Lazić and D Velašević. 2004. Applying simulation and design of experiments to the embedded software testing process. *Software Testing, Verification and Reliability* 14, 4 (2004), 257–282.
[49] Yudong Liu, Xu Zhang, Shilin He, Hongyu Zhang, Liqun Li, Yu Kang, Yong Xu, Minghua Ma, Qingwei Lin, Yingnong Dang, et al. 2022. Uniparser: A unified log parser for heterogeneous log data. In *Proceedings of the ACM Web Conference 2022*. 1893–1901.
[50] Pingchuan Ma, Rui Ding, Shuai Wang, Shi Han, and Dongmei Zhang. 2023. XInsight: eXplainable Data Analysis Through The Lens of Causality. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–27.
[51] Themis Melissaris, Markos Markakis, Kelly Shaw, and Margaret Martonosi. 2020. PerpLE: Improving the Speed and Effectiveness of Memory Consistency Testing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 329–341.
[52] Weibin Meng, Ying Liu, Federico Zaiter, Shenglin Zhang, Yihao Chen, Yuzhe Zhang, Yichen Zhu, En Wang, Ruizhi Zhang, Shimin Tao, et al. 2020. Logparse: Making log parsing adaptive through word classification. In *2020 29th International Conference on Computer Communications and Networks (ICCCN)*. IEEE, 1–9.
[53] Glenford J Myers, Tom Badgett, Todd M Thomas, and Corey Sandler. 2004. *The art of software testing*. Vol. 2. Wiley Online Library.
[54] Sasho Nedelkoski, Jasmin Bogatinovski, Alexander Acker, Jorge Cardoso, and Odej Kao. 2021. Self-supervised log parsing. In *Machine Learning and Knowledge Discovery in Databases: Applied Data Science Track: European Conference, ECML PKDD 2020, Ghent, Belgium, September 14–18, 2020, Proceedings, Part IV*. Springer, 122–138.
[55] Francisco Neves, Nuno Machado, et al. 2018. Falcon: A practical log-based analysis tool for distributed systems. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 534–541.
[56] Francisco Neves, Nuno Machado, Ricardo Vilaça, and José Pereira. 2021. Horus: Non-Intrusive Causal Analysis of Distributed Systems Logs. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 212–223.
[57] Michael Olan. 2003. Unit testing: test early, test often. *Journal of Computing Sciences in Colleges* 19, 2 (2003), 319–328.
[58] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]
[59] Judea Pearl. 1985. Bayesian networks: A model of self-activated memory for evidential reasoning. In *Proceedings of the 7th conference of the Cognitive Science Society*. 15–17.
[60] Judea Pearl. 2009. *Causality: Models, Reasoning and Inference*. Cambridge University Press.
[61] Judea Pearl and Dana Mackenzie. 2018. *The Book of Why: The New Science of Cause and Effect*. Basic books.
[62] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.
[63] Meikel Poess, Bryan Smith, Lubor Kollar, and Paul Larson. 2002. TPC-DS, Taking Decision Support Benchmarking to the Next Level. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. 582–587.
[64] Joseph D Ramsey, Kun Zhang, Madelyn Glymour, Ruben Sanchez Romero, Biwei Huang, Imme Ebert-Uphoff, Savini Samarasinghe, Elizabeth A Barnes, and Clark Glymour. 2018. TETRAD—A toolbox for causal discovery. In *8th international workshop on climate informatics*.
[65] James M Robins, Miguel Angel Hernan, and Babette Brumback. 2000. Marginal structural models and causal inference in epidemiology.
[66] Donald B Rubin. 1974. Estimating causal effects of treatments in randomized and nonrandomized studies. *Journal of educational Psychology* 66, 5 (1974), 688.
[67] Donald B. Rubin. 1980. Discussion of 'Randomization Analysis of Experimental Data in the Fisher Randomization Test'by Basu. *J. Amer. Statist. Assoc.* 75, 371 (1980), 591–93.
[68] Per Runeson. 2006. A survey of unit testing practices. *IEEE software* 23, 4 (2006), 22–29.
[69] Caitlin Sadowski and Jaeheon Yi. 2014. How developers use data race detection tools. In *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools*. 43–51.
[70] Felix Salfner and Steffen Tschirpke. 2008. Error Log Processing for Accurate Failure Prediction. *WASL* 8 (2008), 4.
[71] Babak Salimi, Johannes Gehrke, and Dan Suciu. 2018. Bias in OLAP Queries: Detection, Explanation, and Removal. In *Proceedings of the 2018 International Conference on Management of Data*. 1021–1035.
[72] Bianca Schroeder and Garth A Gibson. 2009. A large-scale study of failures in high-performance computing systems. *IEEE transactions on Dependable and Secure Computing* 7, 4 (2009), 337–350.
[73] Amit Sharma, Emre Kiciman, et al. 2019. DoWhy: A Python package for causal inference. https://github.com/microsoft/dowhy.
[74] Manish Shetty, Chetan Bansal, Sai Pramod Upadhyayula, Arjun Radhakrishna, and Anurag Gupta. 2022. AutoTSG: learning and synthesis for incident troubleshooting. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1477–1488.
[75] Xu Shi, Ziyang Pan, and Wang Miao. 2023. Data integration in causal inference. *Wiley Interdisciplinary Reviews: Computational Statistics* 15, 1 (2023), e1581.
[76] Shohei Shimizu, Patrik O Hoyer, Aapo Hyvärinen, Antti Kerminen, and Michael Jordan. 2006. A linear non-Gaussian acyclic model for causal discovery. *Journal of Machine Learning Research* 7, 10 (2006).
[77] Bill Shipley. 2016. *Cause and correlation in biology: a user's guide to path analysis, structural equations and causal inference with R*. Cambridge university press.
[78] Tomi Silander and Petri Myllymäki. 2006. A simple approach for finding the globally optimal Bayesian network structure. In *Conference on Uncertainty in Artificial Intelligence*. 445–452.
[79] Peter Spirtes and Clark Glymour. 1991. An algorithm for fast recovery of sparse causal graphs. *Social science computer review* 9, 1 (1991), 62–72.
[80] Peter Spirtes, Clark N Glymour, and Richard Scheines. 2000. *Causation, prediction, and search*. MIT press.
[81] Robert Tibshirani. 1996. Regression Shrinkage and Selection via the Lasso. *Journal of the Royal Statistical Society: Series B (Methodological)* 58, 1 (1996), 267–288.
[82] Joseph Tucek, Shan Lu, Chengdu Huang, Spiros Xanthos, and Yuanyuan Zhou. 2007. Triage: diagnosing production run failures at the user's site. *ACM SIGOPS Operating Systems Review* 41, 6 (2007), 131–144.
[83] Thomas Verma and Judea Pearl. 1990. Equivalence and Synthesis of Causal Models. In *Proceedings of the Sixth Annual Conference on Uncertainty in Artificial Intelligence (UAI '90)*. Elsevier Science Inc., USA, 255–270.
[84] Matthew J Vowels, Necati Cihan Camgoz, and Richard Bowden. 2022. D'ya like dags? a survey on structure learning and causal discovery. *Comput. Surveys* 55, 4 (2022), 1–36.
[85] Dolores R Wallace and Roger U Fujii. 1989. Software verification and validation: an overview. *Ieee Software* 6, 3 (1989), 10–17.
[86] Marco A Wiering et al. 2002. Evolving causal neural networks. In *Benelearn'02: Proceedings of the Twelfth Belgian-Dutch Conference on Machine Learning*. 103–108.
[87] Yiting Xia, Ying Zhang, Zhizhen Zhong, Guanqing Yan, Chiun Lin Lim, Satyajeet Singh Ahuja, Soshant Bali, Alexander Nikolaidis, Kimia Ghobadi, and Manya Ghobadi. 2021. A Social Network Under Social Distancing:{Risk-Driven} Backbone Management During {COVID-19} and Beyond. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 217–231.
[88] Tong Xiao, Zhe Quan, Zhi-Jie Wang, Kaiqi Zhao, and Xiangke Liao. 2020. Lpv: A log parser based on vectorization for offline and online log parsing. In *2020 IEEE International Conference on Data Mining (ICDM)*. IEEE, 1346–1351.
[89] Feng Xie, Ruichu Cai, Biwei Huang, Clark Glymour, Zhifeng Hao, and Kun Zhang. 2020. Generalized independent noise condition for estimating latent variable causal graphs. *Advances in neural information processing systems* 33 (2020), 14891–14902.
[90] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I Jordan. 2009. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 117–132.
[91] Brit Youngmann, Michael Cafarella, Babak Salimi, and Anna Zeng. 2023. Causal Data Integration. *Proceedings of the VLDB Endowment* 16, 10, 2659–2665.
[92] Brit Youngmann, Michael J. Cafarella, Babak Salimi, and Anna Zeng. 2023. Causal Data Integration. *Proc. VLDB Endow.* 16, 10 (2023), 2659–2665.
[93] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. 2010. Sherlog: error diagnosis by connecting clues from run-time logs. In *Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems*. 143–154.
[94] Yulai Yuan, Yongwei Wu, Qiuping Wang, Guangwen Yang, and Weimin Zheng. 2012. Job failures in high performance computing systems: A large-scale empirical study. *Computers & Mathematics with Applications* 63, 2 (2012), 365–377.
[95] Jiaming Zeng, Michael F Gensheimer, Daniel L Rubin, Susan Athey, and Ross D Shachter. 2022. Uncovering interpretable potential confounders in electronic medical records. *Nature Communications* 13, 1 (2022), 1014.
[96] Tianzhu Zhang, Han Qiu, Gabriele Castellano, Myriana Rifai, Chung Shue Chen, and Fabio Pianese. 2023. System Log Parsing: A Survey. *IEEE Transactions on Knowledge and Data Engineering* (2023).
[97] Yujie Zhao and Xiaoming Huo. 2023. A survey of numerical algorithms that can solve the Lasso problems. *WIREs Computational Statistics* 15, 4 (2023), e1602. https://doi.org/10.1002/wics.1602 arXiv:https://wires.onlinelibrary.wiley.com/doi/pdf/10.1002/wics.1602
[98] Yujia Zheng, Biwei Huang, Wei Chen, Joseph Ramsey, Mingming Gong, Ruichu Cai, Shohei Shimizu, Peter Spirtes, and Kun Zhang. 2023. Causal-learn: Causal Discovery in Python. *arXiv preprint arXiv:2307.16405* (2023).
[99] Ziming Zheng, Zhiling Lan, Byung H Park, and Al Geist. 2009. System log pre-processing to improve failure prediction. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*. IEEE, 572–577.

[100] Jiongli Zhu, Sainyam Galhotra, Nazanin Sabri, and Babak Salimi. 2023. Consistent Range Approximation for Fair Predictive Modeling. arXiv:2212.10839 [cs.LG]
[101] Jieming Zhu, Shilin He, Jinyang Liu, Pinjia He, Qi Xie, Zibin Zheng, and Michael R Lyu. 2019. Tools and benchmarks for automated log parsing. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 121–130.
[102] Shengyu Zhu, Ignavier Ng, and Zhitang Chen. 2019. Causal discovery with reinforcement learning. *arXiv preprint arXiv:1906.04477* (2019).
[103] Yiwen Zhu, Rathijit Sen, Robert Horton, and John Mark Agosta. 2023. Runtime Variation in Big Data Analytics. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–20.
[104] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Klaus-Tycho Förster, Arvind Krishnamurthy, and Thomas Anderson. 2017. Understanding and mitigating packet corruption in data center networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 362–375.

Received 17 January 2024; revised XX; accepted XX