# Sawmill: From Logs to Causal Diagnosis of Large Systems

Markos Markakis
MIT CSAIL
markakis@mit.edu

An Bo Chen
MIT CSAIL
anbochen@mit.edu

Brit Youngmann
Technion
brity@technion.ac.il

Trinity Gao
MIT CSAIL
trinityg@mit.edu

Ziyu Zhang
MIT CSAIL
sylziyuz@mit.edu

Rana Shahout
Harvard University
rana@seas.harvard.edu

Peter Baile Chen
MIT CSAIL
peterbc@mit.edu

Chunwei Liu
MIT CSAIL
chunwei@mit.edu

Ibrahim Sabek
University of Southern California
sabek@usc.edu

Michael Cafarella
MIT CSAIL
michjc@csail.mit.edu

## ABSTRACT

Causal analysis is an essential lens for understanding complex system dynamics in domains as varied as medicine, economics and law. Computer systems are often similarly complex, but much of the information about them is only available in long, messy, semi-structured log files. This demo presents Sawmill, an open-source system [3] that makes it possible to extract causal conclusions from log files. Sawmill employs methods drawn from the areas of data transformation, cleaning, and extraction in order to transform logs into a representation amenable to causal analysis. It 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 leverages original algorithms to efficiently use this representation for the novel process of *Exploration-based Causal Discovery* - the task of constructing a sufficient causal model of the system from available data. Users can engage with this process via an interactive interface, ultimately making causal inference possible using off-the-shelf tools. SIGMOD'24 participants will be able to use Sawmill to efficiently answer causal questions about logs. We will guide attendees through the process of quantifying the impact of parameter tuning on query latency using real-world PostgreSQL server logs, before letting them test Sawmill on additional logs with known causal effects but varying difficulty. A companion video for this submission is available online [4].

## CCS CONCEPTS

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

## KEYWORDS

Logs, Fault Diagnosis, Causality, LLMs, Causal Discovery

**ACM Reference Format:**
Markos Markakis, An Bo Chen, Brit Youngmann, Trinity Gao, Ziyu Zhang, Rana Shahout, Peter Baile Chen, Chunwei Liu, Ibrahim Sabek, and Michael Cafarella. 2024. Sawmill: From Logs to Causal Diagnosis of Large Systems. In *Proceedings of International Conference on Management of Data (SIGMOD '24).* ACM, New York, NY, USA, 4 pages. https://doi.org/XXXXXXX.XXXXXXX

## 1 INTRODUCTION

Failures are frequent in today's large, complex computer systems, and diagnosing them in production can be challenging [8]: there is usually not enough time (or often even the access permissions) for debugging techniques like testing [6], formal verification [7] and simulation [11]. Instead, operators have to work backward from failures using observational data collected from the system. Informally, we have heard of operations teams spending tens of hours with tools like Datadog [1], trying to diagnose a problem.

However, operations teams want to go beyond a diagnosis - they want to repair the system 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 [13], which has provided scientists with a common language to express and evaluate hypotheses across diverse domains.

We aim for a general-purpose causal diagnosis system that can address a wide range of software systems problems by helping engineers *pose and correctly answer Average Treatment Effect (ATE) queries* [13]. Intuitively, the ATE formalizes a "dose-response" model under Pearl's theory of causality [13] (which we adopt in this work): for example, per "dose" of paralellism (e.g., one more worker), by how much will query latency decrease? Crucially, calculating ATEs correctly from observational data involves adjusting for *confounders* - common causes of both variables involved in an ATE calculation that could bias the observed effect [13]. For example, the available memory could impact both the chosen degree of parallelization and the query latency directly. Calculating ATEs quantifies the trade-offs involved when several interventions could have *some* impact, helping engineers pick the most efficient one.

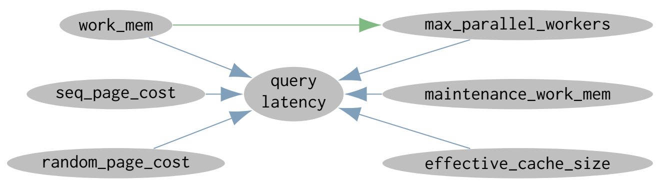
Unfortunately, directly calculating ATEs based on the raw data available to large systems operators is impossible. The raw data usually take the form of collections of logs: semi-structured chronological accounts in text form. Figure 1a presents a snippet of a real-world log from the PostgreSQL dataset that we will use in our demonstration. Past work on log management has led to significant infrastructure, including tools offering extensive coverage of software and hardware components [10]. However, causal inference toolkits require a different representation of the underlying

```
2023-11-06 16:34:40.810 EST [65495bf0.179a 3/2461] postgres@tpcds1
LOG: connection authenticated: method=scram-sha-256
2023-11-06 16:34:40.811 EST [65495bf0.179a 3/2462] postgres@tpcds1
LOG: statement: SET work_mem = 128;
2023-11-06 16:34:40.811 EST [65495bf0.179a 3/2462] postgres@tpcds1
LOG: duration: 0.065 ms
```

**(a) An example log snippet from PostgreSQL.**

| sessionID | auth_method | work_mem | ... | mean latency |
|---|---|---|---|---|
| 65495bf0.179a | scram-sha-256 | 128 | ... | 13483.59 |
| ... | ... | ... | ... | ... |

**(b) Causal inference requires tabular data instead.**



**(c) It also requires a causal graph to adjust for confounding.**

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

data [17], like that of Figure 1b – a table with *one row per data point*, including only *few, relevant variables* and *without missing data*. They also require a causal model [13], something non-trivial to recover from a log. A causal model captures domain knowledge about the relationships among variables and is often represented as a directed acyclic graph (DAG) [13], like that of Figure 1c: each node represents a variable and each directed edge encodes a direct influence of the source variable on the destination variable.

In summary, we aim to combine log management with the best theoretical machinery causality can offer, but face three challenges:

**Challenge A: Deriving the Schema.** Text logs are a far cry from the tabular dataset in Figure 1b. Log parsing algorithms can convert log data into *some* tabular representation, but can yield hundreds of variables without a human-friendly way to navigate them, like the interpretable column names in Figure 1b.

**Challenge B: Distilling the Data.** Causal inference requires the same data per causal unit - e.g., per session. Simply parsing the log falls short of this goal. First, there will be a lot of missing values because each log message only reports *some* variables. Second, logs record information at a very fine granularity. This information must be summarized, while preserving "causal usefulness".

**Challenge C: Obtaining the Causal Model.** For every possible pair of variables, an expert could easily decide the plausibility and direction of the causal relationship between them. However, fully specifying a model over all the log variables by hand is daunting given the problem size. Another option could be to derive the model from the data, a process called causal discovery [13]. However, existing algorithms would fall short due to the problem size and the functional dependencies among variables, while large language models would be tripped up by the context-specific variables.

We propose a framework to address these challenges and transform logs into high-grade causal resources to enable rapid diagnosis of system failures. We implemented this framework in Sawmill, a system that helps engineers managing complex systems formulate

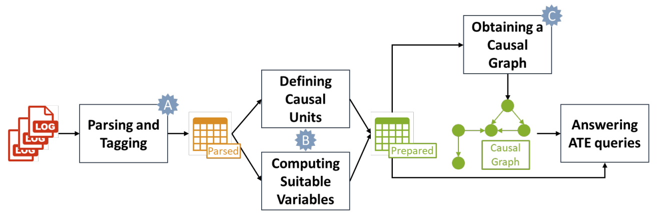


**Figure 2: Our framework, indicating the three challenges.**

a data-driven hypothesis about the cause of an observed failure and retrieve the correctly adjusted associated Average Treatment Effect (ATE). 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.

SIGMOD '24 participants will act as analysts using Sawmill to investigate the causes of system failures starting from system logs. This includes inspecting the raw log files, examining the generated dataset, and finding the primary causes of unexpected behavior. We will walk participants through an analysis of *real-world* PostgreSQL logs collected under different configurations, where `max_parallel_workers` and `work_mem` confound each other's impact on downstream latency on TPC-DS [15] (green edge in Figure 1c). We will then give them the chance to independently use Sawmill to analyze failures in more log datasets of varying difficulty.

## 2 SAWMILL ARCHITECTURE

Figure 2 summarizes our framework, which transforms logs into a table like Figure 1b and derives a causal graph like Figure 1c, enabling ATE calculations. It starts by **Parsing** logs and **Tagging** each variable with a human-understandable tag. The parsed information is then reorganized to satisfy the requirements for causal inference, by **Defining Causal Units** and **Computing Suitable Variables**, making **Obtaining a Causal Graph** and **Answering ATE Queries** possible. We will now dive deeper into each step.

### 2.1 From the Log to the Parsed Table

*2.1.1 Log Parsing.* Logs are first *parsed* into the *parsed table*, with one row per log message and one column per log-derived *parsed variable*. This is performed by calling the function Parse. Users can optionally pass into Parse regular expressions for variables like timestamps. Sawmill then uses Drain [9], an off-the-shelf algorithm, to identify the rest of the parsed variables. To drop uninteresting variables like unique identifiers, we discard any categorical parsed variable with over $0.15V$ distinct values across $V$ total occurrences, similar to past work [19]. If Drain has mapped semantically different variables to the same parsed variable, the user can identify it in the parsed table and correct it using Sawmill's Separate function.

*2.1.2 Variable Tagging.* For variables parsed using a regular expression, the user provides a tag together with the regular expression. Sawmill then automatically assigns a unique human-understandable tag to each of the remaining parsed variables. Sawmill consults three sources in sequence to generate a tag: the tokens preceding the variable in the corresponding *log template*; GPT-3.5-Turbo [2], given an example message including the variable and example values for the variable from other log messages; and GPT-4 [12], given the same information. If no source produces a tag, or if the produced tag is already assigned to a different variable, Sawmill assigns a unique string of symbols instead.

## 2.2 From the Parsed Table to the Prepared Table

*2.2.1 Defining Causal Units.* The parsed table includes one row per log message, but useful ATE queries for troubleshooting are usually about larger units, like sessions or machines. This requires grouping several log messages together as a *causal unit*, with each group being "one data point" for the ATE query at hand. To define causal units in Sawmill, a user calls SetCausalUnit on the appropriate parsed variable - e.g. `sessionID`. Not every choice of causal unit is permissible, because of the Stable Unit Treatment Value Assumption (SUTVA) [16]: the outcome of each unit should not depend on the treatments of *other units*. For example, users may be unsuitable causal units if they share hardware: user $A$'s work could impact user $B$'s latency. We defer to the user's knowledge of the particular system at hand to ensure that this condition is satisfied.

*2.2.2 Computing Suitable Variables.* There can be a varying number of values for each parsed variable in each causal unit. For example, each may have a different number of query latency readings. To make units comparable, Sawmill replaces such varying-size collections of values with *the same* aggregate(s) of the values per causal unit. This transformation, triggered using Prepare, yields the *prepared table*, with one row per causal unit and one column for each *prepared variable*. Each prepared variable is derived from some parsed variable (its *base variable*) by using some function (e.g. the mean) to reconcile the values within each causal unit. A number of prepared variables are generated for each parsed variable using different functions. Among prepared variables sharing a base variable, Sawmill then only keeps the one that maximizes empirical entropy, to limit the prepared table size and processing cost. We also let the user optionally impute missing values with a default based on domain knowledge, by passing the value to Prepare.

## 2.3 From the Prepared Table to ATEs

*2.3.1 Obtaining a Causal Graph.* To obtain a causal graph, Sawmill combines the data-driven and expert-driven approaches: it helps users incrementally build the relevant part of the causal graph, 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) and builds the causal graph around it in two phases:

- **Phase I - Finding a "root cause":** Sawmill helps identify a treatment prepared variable $T$, which *causally affects* $Y$ and is also "actionable". It does so by providing the function ExploreCandidateCauses($U$), which suggests likely causes for a prepared variable $U$. By iteratively leveraging it, the user can arrive at $T$, concluding Phase I. To efficiently leverage the user's attention, Sawmill only presents a *pruned* list of candidate causes, obtained using LASSO [18]. Because each variable is linearly related to its parents and the set of parents is often expected to be small [5], variables that get assigned a zero coefficient can be safely ignored. The user can inspect each candidate cause $U'$, ranked by increasing p-value, and decide whether to include $U' \to U$ in the causal graph (Accept($U'$,$U$)) or not (Reject($U'$,$U$)).
- **Phase II - Finding confounders:** The user continues constructing the causal graph to identify a sufficient set of confounders that affect $ATE(T, Y)$. We measure the user's progress in recovering the regions of the causal graph that *could* include confounders by calculating an *exploration score*: the fraction of accepted/rejected edges among edges that touch at least one node in the current graph. Phase II ends when the exploration score reaches 1. We provide SuggestNextExploration, which returns a prepared variable $U_s$ such that calling ExploreCandidateCauses($U_s$) will yield as many edges relevant to the exploration score calculation as possible, maximizing the user's decision-making efficiency.

*2.3.2 Answering ATE Queries.* Having transformed the log data into the prepared table and having obtained the part of the causal graph relevant for their analysis, the user is now ready to calculate the ATE of interest by calling the function GetATE($T$, $Y$).

## 3 PROTOTYPE AND DEMONSTRATION

## 3.1 Prototype System

Our prototype of Sawmill uses ~2800 lines of Python and is available online [3]. For log parsing, we used Drain [9]. For LASSO, we used scikit-learn [14]. We used "backdoor.linear_regression" from DoWhy [17] to implement GetATE.

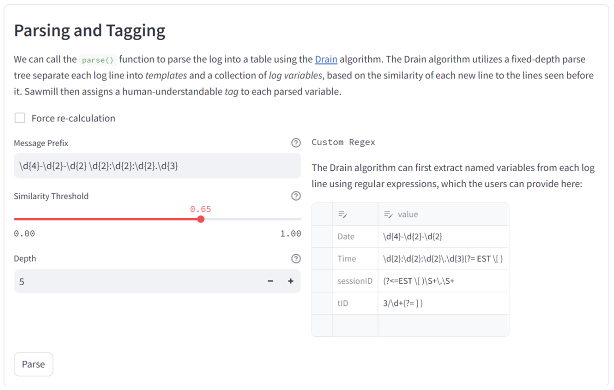
## 3.2 Guided Demonstration

In this part of our demo, SIGMOD attendees will act as operators at a database company. Customers are using the company's database under different configurations, with some experiencing higher mean latency for the same workload. Attendees are interested in correctly determining the impact of the `max_parallel_workers` parameter on mean latency, by following the four steps below. They have access to the **PostgreSQL** dataset, which includes logs collected on PostgreSQL 14, into which we loaded TPC-DS [15] for scale factor 1 and sequentially issued the TPC-DS queries, excluding the long-running queries 1, 4, 11 and 74. We ran this workload for different settings of the six key parameters from Figure 1c.

**Step 1: Obtaining the Parsed Table** Users inspect the log and invoke Parse, as shown in Figure 3a. Inspecting the parsed table reveals a parsing miss: distinct variables have been mapped to a single parsed variable. Users call Separate on the mis-parsed variable to instruct Sawmill to correct the parsing mistake.

**Step 2: Obtaining the Prepared Table** Users specify that each session is a causal unit using SetCausalUnit(`sessionID`). They then call Prepare to get the prepared table.

**Step 3: Calculating the ATE** Users ask Sawmill for candidate causes of mean query latency by invoking ExploreCandidateCauses(`duration:mean`), yielding 11 candidates as shown in Figure 3b. These candidates include 6 variables related to the modified



Parsing and Tagging

We can call the `parse()` function to parse the log into a table using the Drain algorithm. The Drain algorithm utilizes a fixed-depth parse tree separate each log line into *templates* and a collection of *log variables*, based on the similarity of each new line to the lines seen before it. Sawmill then assigns a human-understandable *tag* to each parsed variable.

☐ Force re-calculation

Message Prefix: \d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2}.\d{3}

Similarity Threshold: 0.65 (0.00 – 1.00)

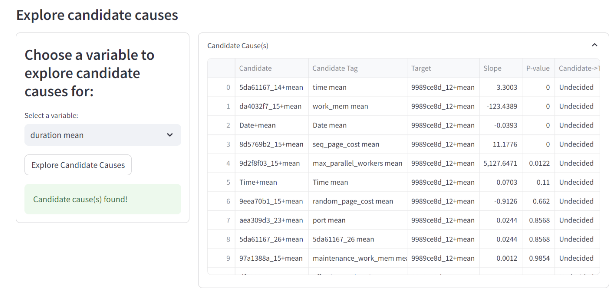
Depth: 5

Custom Regex

The Drain algorithm can first extract named variables from each log line using regular expressions, which the users can provide here:

| | value |
|---|---|
| Date | \d{4}-\d{2}-\d{2} |
| Time | \d{2}:\d{2}:\d{2}\.\d{3}(?= EST \[ ) |
| sessionID | (?<=EST \[ )\S+\.\S+ |
| tID | 3/\d+(?= ] ) |

(a) Users can edit the arguments to Parse, among other functions.



Explore candidate causes

Choose a variable to explore candidate causes for:

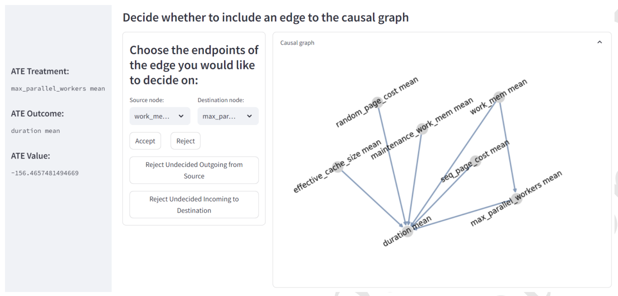
Select a variable: duration mean

Candidate cause(s) found!

Candidate Cause(s)

| | Candidate | Candidate Tag | Target | Slope | P-value | Candidate->T |
|---|---|---|---|---|---|---|
| 0 | 5da61167_14+mean | time mean | 9989ce8d_12+mean | 3.3003 | 0 | Undecided |
| 1 | da4032f7_15+mean | work_mem mean | 9989ce8d_12+mean | -123.4389 | 0 | Undecided |
| 2 | Date+mean | Date mean | 9989ce8d_12+mean | -0.0393 | 0 | Undecided |
| 3 | 8d5769b2_15+mean | seq_page_cost mean | 9989ce8d_12+mean | 11.1776 | 0 | Undecided |
| 4 | 9d2f8f03_15+mean | max_parallel_workers mean | 9989ce8d_12+mean | 5,127.6471 | 0.0122 | Undecided |
| 5 | Time+mean | Time mean | 9989ce8d_12+mean | 0.0703 | 0.11 | Undecided |
| 6 | 9eea70b1_15+mean | random_page_cost mean | 9989ce8d_12+mean | -0.9126 | 0.662 | Undecided |
| 7 | aea309d3_23+mean | port mean | 9989ce8d_12+mean | 0.0244 | 0.8568 | Undecided |
| 8 | 5da61167_26+mean | 5da61167_26 mean | 9989ce8d_12+mean | 0.0244 | 0.8568 | Undecided |
| 9 | 97a1388a_15+mean | maintenance_work_mem mea | 9989ce8d_12+mean | 0.0012 | 0.9854 | Undecided |

(b) An invocation of ExploreCandidateCauses.



ATE Treatment: max_parallel_workers mean

ATE Outcome: duration mean

ATE Value: -156.4657481494669

Decide whether to include an edge to the causal graph

Choose the endpoints of the edge you would like to decide on:

Source node: work_me... Destination node: max_par...

Accept
Reject
Reject Undecided Outgoing from Source
Reject Undecided Incoming to Destination
Causal graph
random_page_cost mean
maintenance_work_mem mean
work_mem mean
effective_cache_size mean
seq_page_cost mean
max_parallel_workers mean
duration mean

(c) Users can build a causal graph and monitor their ATE query.

**Figure 3: Indicative snapshots of using Sawmill.**

PostrgeSQL parameters, any of which could impact query latency, so the user includes the corresponding edges in the graph through calls to Accept, as shown in Figure 3c. Users then invoke the function GetATE(`max_parallel_workers:mean`, `duration:mean`) based on their current graph, but receive a perplexing result of 374.47 - **more parallelism means a longer duration**! Something is amiss.

**Step 4: Adjusting for Confounding** Users follow the output of Sawmill's SuggestNextExploration and explore candidate causes for `max_parallel_workers:mean`. They find that `work_mem:mean` is the top candidate - the amount of working memory confounds the effect of parallelism on latency! Customers only enable higher degrees of parallelism when they have also restricted the working memory of each worker. Users invoke Accept(`work_mem:mean`, `max_parallel_workers:mean`) to add the appropriate edge to the graph, recreating the graph of Figure 1c in Figure 3c. As shown on the left, this takes the ATE of interest to −156.47 - **more parallelism yields a lower query duration**, as expected.

## 3.3 Independent Exploration

SIGMOD attendees will use Sawmill on two more dataset families with known causal effects, under conditions of varying difficulty.

**Proprietary:** This dataset is based on a real log for an HTTP-based application from a large company. It covers a collection of users, a fraction $F$ of which is on a faulty OS version, failing HTTP requests with probability $p_f$. SIGMOD attendees will examine Sawmill's ability to accurately recover the ATE of the faulty OS version on HTTP failure responses. A lower $F$ and/or a lower $p_f$ reduce the "evidence of faultiness", making the causal effect harder to discern.

**XYZ:** This dataset logs the values of $V$ different synthetic variables for each of a collection of "machines". While the values of most variables are randomly chosen, those of $x$, $y$ and $z$ are not. We have that ATE($x$,$y$)=2, but this is distorted by confounding by $z$. Moreover, $x$ and $y$ include added Gaussian noise with $\sigma = R$ each time they are logged. SIGMOD attendees will examine Sawmill's ability to reliably detect and adjust for the confounding introduced by $z$. A higher $V$ and/or a higher $R$ increase the noisiness around the target ATE, making the causal effect harder to discern.

## REFERENCES

[1] 2023. Datadog. https://www.datadoghq.com/ [Accessed 15-Jun-2023].
[2] 2024. Models. https://platform.openai.com/docs/models/gpt-3-5
[3] 2024. Sawmill Code. https://anonymous.4open.science/r/sawmill
[4] 2024. Sawmill Companion Video. https://youtu.be/uCS3pJZ45DI
[5] Tom Claassen, Joris Mooij, and Tom Heskes. 2013. Learning sparse causal models is not NP-hard. *arXiv preprint arXiv:1309.6824* (2013).
[6] 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.
[7] 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.
[8] 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.
[9] 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
[10] Shilin He, Pinjia He, Zhuangbin Chen, Tianyi Yang, Yuxin Su, and Michael R Lyu. 2021. A survey on automated log analysis for reliability engineering. *ACM computing surveys (CSUR)* 54, 6 (2021), 1–37.
[11] 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.
[12] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]
[13] Judea Pearl. 2009. *Causality: Models, Reasoning and Inference*. Cambridge University Press.
[14] 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.
[15] 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.
[16] 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.
[17] Amit Sharma, Emre Kiciman, et al. 2019. DoWhy: A Python package for causal inference. https://github.com/microsoft/dowhy.
[18] Robert Tibshirani. 1996. Regression Shrinkage and Selection via the Lasso. *Journal of the Royal Statistical Society: Series B (Methodological)* 58, 1 (1996), 267–288.
[19] 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.