Diagnosing and Managing Performance in Data Systems

by

Markos Markakis

B.S.E., Princeton University, 2020
S.M., Massachusetts Institute of Technology, 2022

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2026

© 2026 Markos Markakis. All rights reserved.

The author hereby grants to MIT a nonexclusive, worldwide, irrevocable, royalty-free license to exercise any and all rights under copyright, including to reproduce, preserve, distribute and publicly display copies of the thesis, or release the thesis under an open-access license.

Authored by: Markos Markakis
Department of Electrical Engineering and Computer Science
August 14, 2026

Certified by: Tim Kraska
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by: Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

THESIS COMMITTEE

Thesis Supervisor

Tim Kraska
Associate Professor of Electrical Engineering and Computer Science
Department of Electrical Engineering and Computer Science

Thesis Readers

Samuel R. Madden
MIT College of Computing Distinguished Professor
Department of Electrical Engineering and Computer Science

Michael J. Cafarella
Principal Research Scientist
MIT Computer Science and Artificial Intelligence Laboratory

Diagnosing and Managing Performance in Data Systems

by

Markos Markakis

Submitted to the Department of Electrical Engineering and Computer Science
on August 14, 2026 in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

ABSTRACT

Modern data systems are judged by their ability to meet the high-level performance objectives of the diverse applications they support. However, as growing system complexity makes performance sensitive to interactions among components, configurations, states, and workloads, relating these high-level objectives to the evidence and controls exposed by the underlying systems is becoming increasingly difficult.

This thesis traverses this **abstraction gap** in service of two complementary needs: interpreting operational evidence to obtain actionable diagnoses and translating operational objectives into system management decisions. In both cases, we are guided by three design principles: scoping system modeling based on the operator's intent; refining these system models through feedback; and prioritizing reasoning effort based on anticipated impact.

We first present LOGos, a human-in-the-loop framework for causal inference over textual system logs. Building on the insight that answering a particular causal question only requires specifying a portion of the causal graph, LOGos transforms raw logs into data suitable for causal inference, ranks plausible causes, and solicits targeted operator judgments only about causal relationships that can materially affect the effect of interest. It therefore interprets low-level log evidence to help answer high-level diagnostic questions. On real-world and synthetic logs, LOGos improves candidate-cause ranking by 1.08×–18×, requires 1.61×–16.83× fewer causal judgments, and scales linearly with multiple measures of log complexity.

We then present AutoSLO, a framework for managing multi-cluster cloud data warehouses to cost-effectively meet per-query latency service-level objectives (SLOs). AutoSLO operates across three timescales: a periodic Policy Tuner plans for recurring workload behavior, a reactive Autoscaler adjusts resources when observations deviate from the plan, and an online Query Router reacts to live concurrency. It therefore translates high-level latency SLOs into the low-level query routing and cluster management decisions needed to meet them. On realistic Redbench-generated workloads, AutoSLO meets latency objectives of varying strictness while reducing infrastructure cost by a mean of 26.4% relative to the per-scenario next-best baseline, with each component operating efficiently within its timescale.

Together, these systems traverse the abstraction gap in opposite directions, making complex data systems easier to understand and manage.

Thesis supervisor: Tim Kraska
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

I am immensely grateful to everyone who helped make this thesis a reality, whether by providing scientific advice, by accompanying me on my PhD journey, or by helping me become a person that could get to this point today.

First and foremost, I am thankful for the support and guidance of the members of my thesis committee: my advisor, Tim Kraska, who welcomed me to the Data Systems Group and whose eye for real-world high-impact problems has deeply inspired my own developing approach to selecting and tackling research directions; my mentor for the work in Chapter 2, Mike Cafarella, whose unmitigated fascination with new ideas and human-centric approach to advising made our collaboration one of the best periods of my graduate studies; and my close collaborator over the BRAD years, Sam Madden, who is simultaneously one of the smartest and one of the most approachable research mentors one could wish for.

Outside of my thesis committee, I have been lucky to collaborate with researchers in academia and industry who have added depth and nuance to my research. On that front, I would like to thank Umar Farooq Minhas, Per-Åke Larson, Nesime Tatbul, David Cohen, Babak Salimi, Batya Kenig, Oren Mishali, Lei Cao, Vikram Nathan, Davide Pagano, and Balakrishnan (Murali) Narayanaswamy. Of course, none of these collaborations would have been possible without Themis Melissaris, Margaret Martonosi, and Kelly Shaw, all of whom introduced me to research as an undergraduate student at Princeton.

Bridging the professional and the social, I have been lucky to find myself in one of the most collaborative and social research groups I am aware of, the MIT Data Systems Group. This journey would not have been possible without the suggestions, jokes, feedback, and constant positive presence of Geoffrey Yu, Ferdi Kossmann, Ziniu Wu, Matthew Russo, Tianyu Li, Sivaprasad Sudhir, Jialin Ding, Kapil Vaidya, Jason Mohoney, Eugenie Lai, Gerardo Vitagliano, Anna Zeng, Andreas Kipf, Xinjing Zhou, Amadou Ngom, Brit Youngmann, Oscar Moll, Matt Perron, Joana M.F. da Trindade, Sylvia Zhang, Dominik Horn, Pascal Pfeil, Fabian Wenz, Chunwei Liu, Trinity Gao, Wenjia He, Yuze Lou, Peter Baile Chen, Joshua Gu, Ibrahim Sabek, An Bo Chen, Rana Shahout, Darryl Ho, and Sushrut Borkar.

Outside the lab, staying happy and sane so far away from home was only possible through an irreplaceable community of friends assembled through the years. In Boston, I am thankful for Nikos Meimetis, George Margaritis, Pericles Petridis, Anastasia Repouliou, Vassilina Stoumpou, Angelos Koulouras, Chris Argyros, Luka Karginov, Krista Pullen, Korina Anagnostopoulou, Aristeidis Karalis, Christos Zarkos, Konstantinos Kanellis, Thodoris Koukouvinos, Giannis Spantidakis, Thymios Tzinis, Alkiviadis Mertzios, Nikola Samardzic, and Anastasia Tsilia. Back home, I have been lucky to have retained some extremely strong friendships with people who have been by my side for most of my life. A big thank you

goes to Thomas Trikoukis, Kostis Goulidis, Anastasis Gazis, Dimitris Tsilimigkakis, Lukas Mantzolis, and Solon Sefiha for all the fun and serious moments throughout the years.

Spending so much time thinking about DBMSs over the past several years would be much harder without another very important DB in my life: Defne Buyukyazgan. Her support and love have been invaluable throughout this journey and I am looking forward to continuing to build our individual and collective dreams together.

Last but definitely not least, all of this has only been possible thanks to my family, who have (literally) been there since day one and who have never stopped believing in me and my claim to a happy and fulfilled life. Thank you to my parents, grandparents, aunts, uncles, and cousins, for making me who I am today.

For the body of work presented in Chapter 2, we are grateful for the support from the DARPA ASKEM project (Award HR00112220042), the ARPA-H Biomedical Data Fabric project, and a grant from Liberty Mutual.

For the body of work presented in Chapter 3, we are grateful for the support from Amazon, Google, and Intel as part of the MIT Data Systems and AI Lab (DSAIL). This research was also sponsored by the Department of the Air Force Artificial Intelligence Accelerator and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Department of the Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

Contents

List of Figures

List of Tables

List of Algorithms

List of Formal Statements

Chapter 1

Introduction

Modern data systems are built to support applications and are judged by those applications' high-level outcomes. Examples include user-facing latency, availability, and reliability, as well as operating considerations like infrastructure cost and team budgets and timelines. Such outcomes can often be formalized as quantitative service-level objectives (SLOs) [30, 249], such as "95% of queries originating from this dashboard must complete within 500 ms".

While SLOs help define and identify problems, they do not offer deeper insight into their causes. An application-level symptom such as high latency may originate from insufficient resources, an inappropriate configuration, an unexpected workload spike, or a newly introduced software bug. Disambiguating these possibilities and taking corrective action has become increasingly difficult as modern data systems have grown more tailored [211] and complex, in response to forces as diverse as novel application requirements [25, 28, 117, 128, 169, 193], emerging hardware trends [36, 233, 244, 248], practical concerns about scalability, maintainability, and usability [63, 157, 177], and the commodification of infrastructure and data management services in the cloud [19, 52, 176, 224]. Moreover, performance is not a static property of the system design, but emerges from interactions among the system's components, its configuration, its current state, and its current workload. Decisions that led to SLO compliance under one set of conditions may therefore cease to do so as workloads evolve, data distributions shift, software is updated, and resource utilization varies.

Continuously interpreting system monitoring data, diagnosing problems, and issuing appropriate corrective actions can require significant technical expertise, beyond the business considerations driving the applications supported by the system. Rather than directly accepting a high-level SLO, the underlying system can often only be controlled by mechanisms such as memory allocations, scheduling policies, and resource provisioning, while it documents and exposes its past behavior through large volumes of logs, traces, and execution profiles. There is, therefore, often an **abstraction gap** between the level at which systems expose their behavior and the level at which system operators can productively reason about this behavior and manage it, as shown in Figure 1.1. As systems grow more complex, this gap widens further, since it becomes increasingly difficult for an operator to inspect all available evidence or evaluate every possible operational action.

The abstraction gap adds friction to the process of translating business needs into effective system decisions. Academia and industry have sought to narrow this gap in recent years by progressively raising the level of abstraction at which systems are observed, configured, and

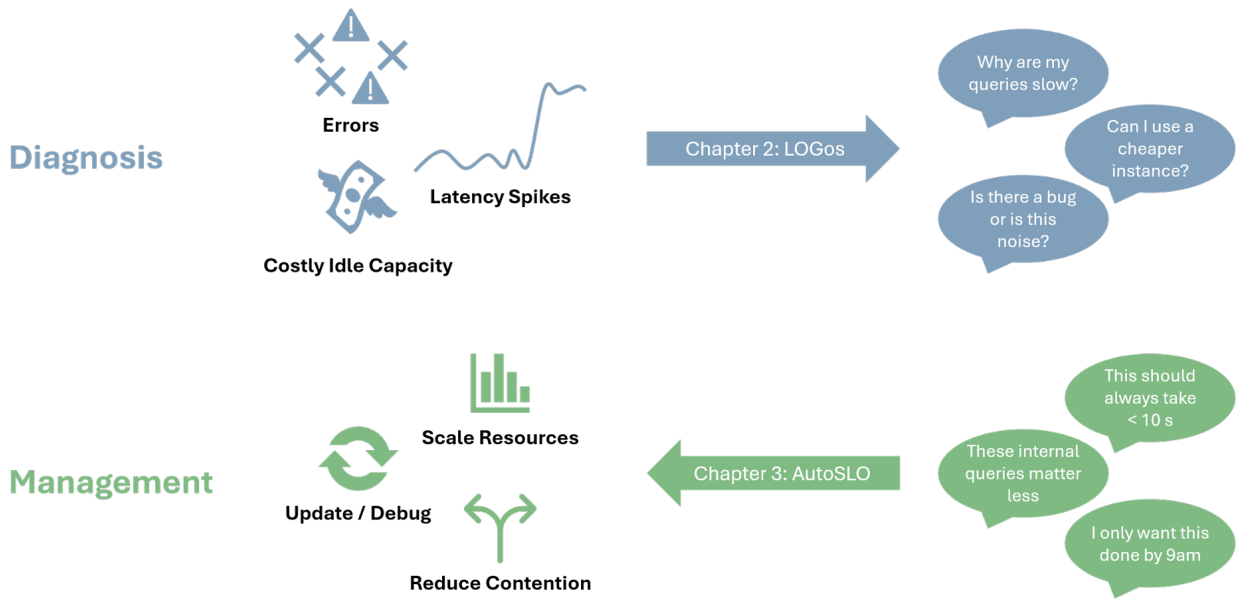


Figure 1.1: Managing complex systems is hampered by an abstraction gap between system inputs/outputs and operator questions and specifications. LOGos and AutoSLO narrow this gap, helping extract causal conclusions and enforce latency SLOs.

operated. Managed and serverless services hide many infrastructure decisions [204], while modern observability platforms [57, 209] and self-managing systems [157, 189] help operators automate some of the burden. The recent rise of large language models and AI agents has further streamlined the operator experience [40, 77, 123, 127, 243, 259, 260, 269, 270]. In this thesis, we contribute to this research direction in support of the following thesis statement:

Thesis Statement

Efficiently operating increasingly complex data systems requires narrowing the abstraction gap between their low-level monitoring and control mechanisms and operators' high-level questions and objectives.

In particular, we help address two complementary needs. First, when a system exhibits undesirable performance, operators need to understand the underlying causes and assess the expected impact of different possible corrective actions. For example, is it time to switch to a cluster with more available memory, or is performance degradation caused by a newly introduced software bug? The first part of this thesis investigates how textual system logs can be used to answer such questions in a principled manner while selectively tapping into the domain knowledge of operators. Second, as workloads and operating conditions evolve, operators need to ensure the system continues to achieve the desired SLOs. The second part of this thesis focuses on how query latency objectives can be translated into query routing and cluster management decisions across diverse workloads and timescales.

1.1 Design Principles

Although the two parts of this thesis traverse the abstraction gap in opposite directions, they both reflect three common design principles:

Principle 1: Scope modeling by intent. Fully modeling a complex modern system with all of its components, states, interactions, and possible behaviors can be extremely challenging, but it is often also unnecessary. An operator typically needs to answer a specific question or achieve a particular objective, rather than understand every aspect of the system. We can use this well-scoped intent to selectively expend modeling effort on understanding only the relevant pieces of evidence and the relationships among them. This reduces the overhead of constructing and maintaining such a model, while retaining the information needed to produce a useful answer or decision.

Principle 2: Refine models through feedback. System models constructed from operational evidence are inevitably imperfect. Even without deliberately limiting model scope, sources of imperfection abound: logs are only partial views of system behavior, historical observations may not represent current conditions, and complex interactions may be difficult to reliably infer. A practical tool should remain flexible, incorporating feedback as new or more detailed information becomes available, either in the form of direct operator knowledge or from observations collected during continued system operation.

Principle 3: Prioritize reasoning by impact. Resolving *every* model imperfection can become prohibitively expensive. This cost may take the form of computation, expert operator attention, or decision-making latency. However, not every imperfection is equally consequential: many additional refinements would leave the ultimate answer or action unchanged. We should therefore prioritize based on potential downstream impact, dedicating expensive analysis or human attention only when it could materially affect the final result.

Taken together, these principles help create practical and efficient management tools that can adapt over time. We next show how these principles inform each part of this thesis.

1.2 Diagnosing Performance with LOGos

The first direction across the abstraction gap is from low-level operational evidence to answers about undesirable performance. Observing an SLO violation or another application-level symptom identifies that a problem exists, but not what caused it or how an operator should respond. An actionable diagnosis must disambiguate among possible causes and help quantify how potential corrective interventions would change the observed outcome.

In a complex system where components may use diverse technologies, text logs remain a ubiquitous common denominator and an important source of evidence for obtaining a comprehensive view of the system state. Still, the same flexibility that makes logs easy to generate and merge implies widespread heterogeneity, for example in message formatting or in the frequency of logging, which makes logs hard to analyze. Existing log-analysis techniques largely sidestep this problem by taking a coarser-grained view and focusing on log anomalies or messages explicitly labeled as errors [56, 57, 70, 80, 187, 209]. However, detecting an anomaly is only a first step. It is a far cry from the actionable, verified diagnosis needed by an operator in order to determine what to do next and what results they can expect.

Reaching such a diagnosis calls for *causal inference*: the study of discovering and quantifying cause-effect relationships. However, applying causal inference to production textual logs is non-trivial. One must first determine and populate suitable variables using the log data, before positing their causal relationships and quantifying the implied causal effects. Manually performing this preprocessing is impractical given the scale and heterogeneity of production log data; at the same time, automating it fully is both computationally expensive and unreliable when dealing with the redundant and unevenly distributed log data.

Chapter 2 addresses this problem by presenting LOGos, a human-in-the-loop framework that efficiently empowers operators to extract causal conclusions from textual logs with minimal feedback. Its central insight reflects Principle 1: answering a particular causal question does not require constructing the system's complete causal graph. Instead, LOGos only constructs the portion of the causal graph needed to estimate the effect of interest. Following Principles 2 and 3, it constructs this partial graph by soliciting focused operator feedback, in the form of judgments over pairwise causal relationships between log variables. Concretely, the Log Converter transforms raw logs into a tabular representation suitable for causal inference; the Candidate Cause Ranker helps operators navigate this representation by ranking plausible causes of an observed outcome variable of interest; and the Interactive Causal Graph Refiner focuses operators' attention only on the unresolved causal relationships that could strongly affect the effect of interest.

Our evaluation of LOGos on real-world and synthetic logs shows that this human-in-the-loop approach improves candidate-cause ranking by 1.08×–18× over appropriate baselines, requires 1.61×–16.83× fewer operator judgments to refine the causal graph, and scales linearly with multiple measures of log complexity. As shown in Figure 1.1, LOGos demonstrates one way to bridge the abstraction gap in service of the performance diagnosis need: rather than reasoning over large volumes of low-level evidence, operators only need to use LOGos to posit a causal question and selectively contribute domain knowledge. This contribution is developed in Chapter 2 and draws on previously published work [145, 148, 149].

1.3 Managing Performance with AutoSLO

The second direction across the abstraction gap is from high-level performance objectives to the low-level actions needed to achieve them. An SLO specifies the desired outcome, but does not determine how resources should be provisioned, how work should be scheduled, or when those decisions should be revisited. Without appropriate management tools, operators must continually assess risks and manipulate these low-level mechanisms themselves.

This problem is particularly challenging in cloud data warehouses, which support heterogeneous workloads with substantially different performance expectations: interactive dashboards, ad hoc data-science queries, and batch-processing jobs may all operate on the same underlying data. The current architectural best practice is to take advantage of the compute-storage decoupling offered by modern cloud data warehouses, assigning different workloads to different clusters and managing them independently. However, assigning each workload to a dedicated cluster is only a coarse-grained attempt to safeguard its performance requirements; each dedicated cluster must still be continually scaled as its workload changes, with over-provisioning wasting resources and under-provisioning risking poor performance.

Automating workload management in cloud data warehouses has received significant attention in recent years from academia and industry alike, with production systems gradually raising the level of abstraction of their interfaces. For example, Amazon Redshift Serverless offers a qualitative slider interface through which users can express their price-performance sensitivity [157]. Existing systems, however, generally stop short of allowing application owners to express per-workload latency SLOs and then enforcing them automatically.

Chapter 3 presents AutoSLO, a workload management framework where such latency SLOs are treated as first-class inputs. The SLOs scope both the management objective and the modeling required to maintain it, reflecting Principle 1. Rather than constructing a detailed analytical model of the warehouse, AutoSLO is built around a learned contention-aware Latency Predictor, which is used to inform query-routing and cluster-management decisions.

Because workload forecasts and latency predictions are necessarily imperfect, AutoSLO receives and responds to feedback across three operating timescales, in accordance with Principle 2. A periodic Policy Tuner uses historical workloads to proactively plan workload execution by scheduling cluster scaling actions and configuring online policies. A reactive Autoscaler adjusts the active cluster set when recent workload behavior deviates from the plan. Finally, an online Query Router reacts to live concurrency when placing each arriving query, accounting for both its own SLO and its effect on already-running queries.

The separation across three timescales also reflects Principle 3. Expensive workload simulations used for policy exploration are performed by the Policy Tuner only periodically, which amortizes their cost by ensuring they can benefit a long enough future workload horizon. The Autoscaler considers reactive scaling only when recent observations indicate it may be consequential, and it uses a simple workload forecasting approach to remain efficient. Finally, the Query Router is designed to be efficient enough to operate on the critical path of every query. AutoSLO therefore matches the depth and latency of its reasoning to the scope, urgency, and anticipated impact of each decision.

Our evaluation on realistic Redbench-generated workloads shows that AutoSLO meets latency objectives of varying strictness while reducing infrastructure cost by a mean of 26.4% relative to the per-scenario next-best baseline. Component-level experiments further show that each component reduces SLO violations while remaining efficient for its intended operating timescale. As shown in Figure 1.1, AutoSLO bridges the abstraction gap for the performance management need in the opposite direction from LOGos: while LOGos translates low-level log-derived evidence into answers to high-level diagnostic questions, AutoSLO translates high-level performance objectives into the low-level query routing and cluster management actions needed to meet them cost-effectively. This contribution is developed in Chapter 3 and draws on previously published work [146, 147].

Chapter 2

From Logs to Causal Inference: Diagnosing Large Systems

2.1 Introduction

The scale and complexity of today's computer systems make failures frequent [82, 109, 190, 253] and their diagnosis challenging, especially in production [112, 185, 218]. Traditional systematic debugging techniques, like testing [54, 150, 153, 183], formal verification [35, 65, 95, 225], and simulation [111, 124, 236], often fall short for problems in a large, complex production system [185]. Operations teams do not have the time and expertise (or often even the access permissions) to fully ascertain the correctness of the codebase [109, 196]. Instead, they have to work backward from each failure towards its cause using observational data collected from the system, most notably large volumes of logs, usually in text format.

Episode 1 *Alex is an on-call engineer at a startup that offers a cloud-based software service. Several users are complaining that a certain feature, which triggers a query on a user-specific PostgreSQL back-end database, is very slow. Alex's manager has tasked Alex with discovering why the feature is so slow for these specific users and the best way to fix it quickly. But all Alex has to go on is text logs!*

User Goal. Informally, we have heard reports from operations teams spending tens of human-hours with tools such as Splunk [209] or Datadog [57] in order to diagnose a poorly understood problem. Such log analysis tools offer a good starting point, given their broad coverage of software and hardware components through existing log management infrastructure. However, if users could simply *ask* such a tool *why* an observed problem took place and *by how much* we expect each of a collection of remedies to help, then we might dramatically lower the Mean Time to Repair (MTTR).

To answer such questions in a principled manner, we turn to the growing field of causal reasoning [172], which aims to quantitatively describe cause-effect mechanisms. Causal reasoning has provided scientists with a common language to express and evaluate hypotheses about complex systems across diverse domains including medicine [231], economics [104], biology [47], and social sciences [205]. In recent years, causal reasoning has also piqued the interest of computer scientists. For example, machine learning researchers leverage it to provide

additional structure and trustworthiness to machine learning problems, leading to research areas like causal representation learning [191] and causal reinforcement learning [75, 256]. At the same time, computer systems researchers have used causal reasoning to better understand and debug large complicated systems [11, 73].

In this work, we adopt the standard theory of causality developed by Pearl [172], where the goal is to correctly calculate an *Average Treatment Effect (ATE)* corresponding to a hypothesis about the observed phenomenon – for example, in Alex's case, if one more worker is added to PostgreSQL, how much will mean query latency change? What if the working memory is increased by 1 MB?

To calculate such ATEs, Pearl's theory requires not only observations, but also another input called a *causal graph*. The causal graph represents each observed variable as a node and each potential direct causal relationship as a directed edge, and can be used to infer which variables to *adjust for* (the *adjustment set*). Without a proper adjustment set, an ATE calculation can be inaccurate due to *confounding bias*, where variables sharing a common cause erroneously appear to cause each other [172]. Therefore, the accuracy of the ATE directly depends on the quality of the causal graph. We aim for a causal diagnosis tool that can help address a wide range of software and data systems problems, ultimately enabling a user experience like the following:

Episode 2 *Alex discovers that our framework can create high-quality resources for causal inference out of log data. With a few interactions, Alex transforms the available log data and rapidly finds out that the maximum allowed number of parallel workers exerts a high absolute ATE on mean query latency. Alex forwards this to the database team to ensure that this setting is set appropriately, restoring user performance.*

Technical Problem. As described, correctly estimating an ATE from observational data requires a valid adjustment set, and finding a valid adjustment set is in turn made possible by an accurate causal graph. Obtaining such a causal graph is therefore our main concern.

However, high-quality causal graphs are difficult to obtain at scale. For problem instances with few variables, a causal graph is usually assumed to be manually curated by a domain expert [149, 222]. However, manual curation is a daunting and error-prone task over the possibly hundreds or thousands of log-derived variables [161], since the number of edges to consider for inclusion is quadratic in the number of variables. Instead, it is often possible to obtain a (partial) causal graph from the data itself, using one of several available *causal discovery* algorithms [78, 198, 208, 223, 255, 275]. Such algorithms progressively narrow the class of causal graphs that would be compatible with the available data, using various conditional independence calculations or information-theoretic scoring techniques.

Even though causal discovery algorithms are more scalable than manual construction, the graphs they generate are not perfect. Depending on the choice of algorithm, the generated graph may be *incomplete*, if the available data is not sufficient to determine the direction of all causal relationships [207, 208]. Even worse, depending on the requirements of the causal discovery algorithm (e.g. independent noise terms [198]) and possible biases in the available data (e.g. selection bias), the generated graph may be *incorrect* with respect to the actual data generation process [120]. Even small mistakes in a causal graph can have a large impact on downstream results, if they imply a different adjustment set.

In particular, as we will show in Section 2.2.2, causal discovery algorithms fall short for log datasets because of three **challenges of log data**:

1. **Functional Dependencies**: Logging aims to capture as much information about the system as possible, even if redundant, since reproducing a sequence of events may not be an option. This leads to widespread functional dependencies in the resulting datasets, which cause issues with the conditional independence tests often used for causal discovery.

2. **Large Number of Variables**: The sophistication and modularity of modern systems lead to logs capturing hundreds or thousands of variables. The exponential complexity of causal discovery algorithms can be prohibitive in this setting.

3. **Biased Data Collection**: Capturing a large number of variables at a high frequency can mean that an interesting log message is drowned out by strong "common case" message sequences, which will be more readily identified during causal discovery.

The key difficulty we tackle in this chapter is efficiently obtaining a causal graph from log data despite these challenges, without resorting to exhaustive manual causal graph construction.

Our Approach. To combat the three challenges above, we use a key insight: while overall graph fidelity is important, **correctly calculating an ATE only requires part of the causal graph**. This is because not every causal graph inaccuracy will alter the adjustment set, and it is the adjustment set that determines the ATE. As long as we focus on the impactful parts of the graph, we can solicit *some* human input without overwhelming users.

We therefore propose approaching causal inference over log data using a *human-in-the-loop* framework called LOGos, combining the intelligent use of available data and a judicious solicitation of user input. Through a series of *judgments*, the user can iteratively receive suggestions from LOGos and refine parts of their causal graph in order of decreasing importance. This leads to a causal graph sufficient for calculating the ATE of interest correctly, with significantly less user input than a naive approach. We achieve this through two human-in-the-loop modules: the Candidate Cause Ranker and the Interactive Causal Graph Refiner, together with a data preparation pipeline, the Log Converter.

The Candidate Cause Ranker helps users quickly discover causes of their outcome of interest. In each call, the user specifies a variable E and receives a ranked list of candidate causes $\{C_i\}$. The user can then tap their expertise to evaluate the plausibility of each candidate C_i and possibly add the edge $C_i \to E$ to their causal graph. Crucially, if some candidates are functionally dependent, the user has the freedom to include the most appropriate one for their analysis in the causal graph. Thus this module addresses **Challenge 1**. By iteratively applying this process, users can navigate log data efficiently to find a "root cause" T of their original outcome variable of interest O (e.g. mean latency, for Alex).

However, discovering *one* causal path from T to O is not enough to correctly calculate the ATE, since there may be additional variables confounding this effect. Such variables should be added to the graph and used to derive the adjustment set. For example, Alex may mis-estimate the impact of increased parallelism on mean query latency if the changes made to *other* settings (e.g. the working memory) in order to accommodate more parallelism are not considered. The Interactive Causal Graph Refiner addresses this problem through another human-in-the-loop process. In particular, given T, O, and the user's current causal graph, it

presents graph edits to the user that would *maximally affect* the ATE of T on O. The user can then consider the plausibility of such edits and decide whether to apply them, progressively increasing the accuracy of their graph and the robustness of their ATE calculation. This frees the user from having to consider the possible influence of *every* variable in the dataset on the ATE of interest, addressing **Challenge 2**.

Of course, even before a causal graph can be built, one must extract the problem variables from the textual log. Therefore, alongside our human-in-the-loop approach to causal graph construction, we present the Log Converter, a pipeline transforming raw log inputs into a tabular dataset appropriate for causal analysis. This dataset serves both as the input to our human-in-the-loop modules and as the second input to Pearl's model for ATE calculation (alongside the graph itself). After parsing the logs, our pipeline uses large language models to derive human-understandable variable tags, before distilling log information around the user's desired causal units and generating appropriate aggregated variables for every causal unit to maximize empirical entropy. This approach to aggregation allows interesting observations to cut through the noise of uninteresting log messages, combating **Challenge 3**.

Our framework can be an important tool for engineers managing complex systems. Our human-in-the-loop approach may also work well in other domains, where data exhibit challenges similar to those of logs. We will explore this direction in future work.

Overview of Related Work and Novelty. Large system debugging is an active research area with serious commercial implications. However, we are the first to simultaneously tap *principled Pearl-style causality* and the *wealth of information present in text logs*.

On the one hand, some works leverage causality but do not tap the ubiquitous textual log data. ExplainIt! [107] draws inspiration from causal techniques to build a root cause analysis engine over diverse time series, while Sage [73] uses modular causal graphs based on telemetry data to localize failures in microservice architectures. CausalSim [11] leverages causality to make the most out of existing tabular RCT data, avoiding further trials.

On the other hand, there are works that operate on logs but model causality informally at best. Aggarwal et al. only model causality among the *counts* of error-level log messages [4], while we consider the actual *values* embedded in them. Falcon [159] and Horus [160] use simple *temporal precedence* as a proxy for causality, whereas we model causality using the probabilistic Pearl model instead. Additional related work is discussed in Section 2.9.

Contributions. In summary, we make the following contributions:

- We propose a human-in-the-loop framework for efficiently applying causal inference to data derived from logs.
- We introduce the Candidate Cause Ranker, a module for uncovering candidate causes using curated data-driven suggestions.
- We present the Interactive Causal Graph Refiner, a module for refining causal graphs for accurate calculation of an ATE.
- We describe the Log Converter, a data transformation pipeline from logs into a tabular representation for causal inference.

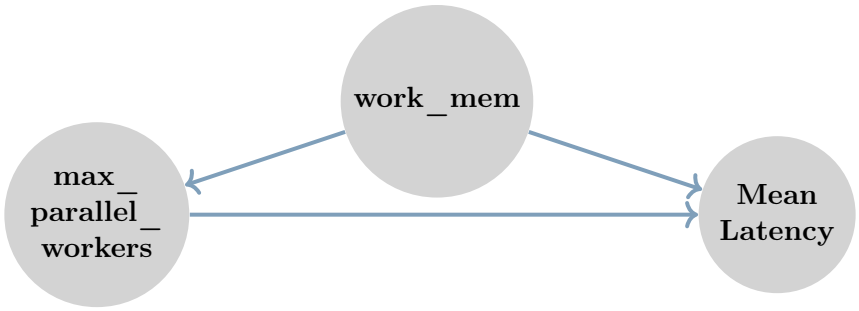


Figure 2.1: An example partial causal graph for Alex's problem.

- We evaluate LOGos, an open-source implementation of our proposed framework [144] and the first end-to-end system that can go from log files to ATEs, and find that:
  - The average precision achieved by the Candidate Cause Ranker is 1.08×–1.66× higher than Regression and 1.54×–18× higher than LangModel, while remaining interactive.
  - The number of judgments required by the Interactive Causal Graph Refiner is 1.61×–2.50× lower than Regression and 5.50×–16.83× lower than LangModel.
  - The strategy used inside the Interactive Causal Graph Refiner outperforms alternatives, tying the lowest ATE error after 10 of judgments while reducing latency by 9.28×.
  - The Log Converter scales linearly with each of three measures of the complexity of a log: length, distinct templates, and fraction of tokens that are variables.

In this chapter, we will provide some background and a motivating experiment (Section 2.2), followed by an overview of our framework (Section 2.3). We will next describe our data preparation pipeline (Section 2.4) and two human-in-the-loop modules (Sections 2.5–2.6), and present and evaluate a prototype of our framework (Sections 2.7–2.8). We then discuss related work (Section 2.9) and lessons learned (Section 2.10) before concluding (Section 2.11).

2.2 Problem Formulation

In this section, we will build towards our problem definitions more formally. We will introduce some useful notions from causal inference (Section 2.2.1), describe a motivating experiment (Section 2.2.2), and finally state the problems we target (Section 2.2.3).

2.2.1 Background on Causal Inference

We begin by introducing some notation and discussing elements of the conventional causal inference framework developed by Pearl [172, 173].

Notation. We use uppercase letters to represent variables (e.g. V) and lowercase letters to denote their values (e.g. $V = v$). $V \to W$ indicates a directed edge from V to W. Bold font denotes sets of variables, edges, or edge edits. Calligraphic font denotes causal graphs (e.g. $\mathcal{G}$). For directed graphs, we distinguish a *path* (where edges need not share the same orientation) from a *directed path*.

The ATE. Causal inference estimates the *Average Treatment Effect (ATE)* of a treatment T on an outcome O over a population of *causal units* (e.g. PostgreSQL sessions). The ATE may be distorted by *confounding variables* (or *confounders*) $\mathbf{Z}$ which influence both T and O [172, 173]. For example, as shown in Figure 2.1, if Alex estimates the ATE of `max_parallel_workers` on mean latency, a confounder may be `work_mem`: more working memory can accelerate operations and reduce latency, but it may also mean a lower allowable level of parallelism, in the interest of managing the total available memory in the system.

To avoid confounders, one can collect *interventional* data, where T no longer depends on $\mathbf{Z}$ because it is externally assigned – e.g. through a randomized controlled trial (RCT). However, RCTs can be expensive and time-consuming – for example, Alex would have to subject users to randomly selected settings. Depending on the domain, RCTs may also be unethical or completely infeasible. Thankfully, we can also compute the ATE over *observational* data, such as the log data we focus on, as long as we *adjust for the confounders*:

Definition 2.1: ATE from Observational Data [172]

Given a treatment T, an outcome O, and a valid adjustment set $\mathbf{Z}$,

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

For the ATE to be accurate, $\mathbf{Z}$ must be selected to correctly adjust for confounding bias. One method for achieving this is the back-door criterion [172], which we will now build towards.

Causal Model. A causal model captures causal relationships among the problem variables. These relationships can be arbitrary, but they are most often assumed to be linear [172, 208]:

Definition 2.2: Linear Causal Model

Given variables V_i for $i = 1, ..., n$, with associated sets of direct causes (*parents*) $\mathbf{P}_i$ and error terms E_i, a **linear causal model** is a set of equations:

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

Causal Graph. A causal model can be visualized as a *causal graph* [171], like Figure 2.1: a directed acyclic graph (DAG) with one node per variable, where a directed edge $V_i \to V_j$ implies $V_i \in \mathcal{P}_j$. In this context, we may use "variable" to refer to the corresponding node in the causal graph, and we can further define:

Definition 2.3: Ancestors and Descendants

Each *ancestor* W of V_i has a directed path *to* V_i ($W \in \mathbf{An}(V_i)$). Similarly, V_i has a directed path to each W among its *descendants* ($W \in \mathbf{De}(V_i)$).

Definition 2.4: T-Backdoor Path

A path p between T and O is a **T-backdoor path** if some directed edge on p has T as its destination.

Path Blocking and the Back-Door Criterion. A valid adjustment set is one that eliminates all sources of "indirect" causal influence, formalized using *path blocking*:

Definition 2.5: Path Blocking [172]

Path p is blocked by nodes $\mathbf{B}$ if and only if:

1. p has a chain $i \to m \to j$ or fork $i \leftarrow m \to j$ s.t. $m \in \mathbf{B}$; or
2. p has a collider $i \to m \leftarrow j$ s.t. $(\{m\} \cup \mathbf{De}(m)) \cap \mathbf{B} = \emptyset$.

Definition 2.6: Back-Door Criterion [172]

A set $\mathbf{Z}$ satisfies the **backdoor criterion** relative to variables (T, O) in a DAG $\mathcal{G}$ if:

(i) $\mathbf{Z} \cap \mathbf{De}(T) = \emptyset$; and

(ii) $\mathbf{Z}$ blocks every T-backdoor path between T and O.

In this case, $\mathbf{Z}$ is a sufficient adjustment set for $\text{ATE}(T, O)$.

2.2.2 How Existing Causal Discovery Approaches Fall Short

We will next show why existing approaches to causal discovery are ill-suited for our use case.

The PostgreSQL Dataset. This 20 MB dataset, presented fully in Section 2.8.1.4, mimics Alex's predicament. It includes collated logs from 96 workload runs of queries from the TPC-DS benchmark [179], issued against PostgreSQL. Before each run, we modified 6 PostgreSQL configuration knobs, each of which has a known impact on query latency. We selected the combinations of knob settings to introduce confounding between `work_mem` and `max_parallel_workers`. This is a small representative example of a broader class of confounding related to resource sharing that can come up often in the real world, albeit with much larger volumes of data. Using the Log Converter, we converted this log dataset into a tabular format with 172 variables. We then attempted to derive a causal graph over these variables. **A useful causal graph would, at a minimum, capture the hand-constructed structure shown in Figure 2.1: `work_mem` and `max_parallel_workers` each affect query latency and they confound each other's effect.**

Causal Discovery. We tested 29 combinations of causal discovery algorithms and configuration parameters from CausalLearn [266] on the PostgreSQL dataset. Per Table 2.1, 16 produced errors, 6 timed out, and 2 only produced an empty graph. Of the rest:

- PC-`chisq`/FCI-`chisq`: The variable for query latency was absent from the produced graph.
- GRaSP-`BDeu`/GES-`BDeu`: One 5-node connected component included the variable capturing query latency and no configuration-related variables. A different 4-node connected component included the variables for `work_mem` and `max_parallel_workers`.
- GRaSP-`CV_general`: The graph showed query latency as *causing* `work_mem`, while the variable for `max_parallel_workers` was absent.

Table 2.1: Performance of existing causal discovery algorithms on the PostgreSQL dataset. ✓ and ● indicate a non-empty and empty causal graph, respectively; ▲ indicates a 30-minute timeout; and ✗ indicates an error.

Alg.	Independence Test / Method	Result	Alg.	Scoring Function	Result
PC [207]	`fisherz`	✗	GIN [238]	`kci`	▲
	`mv_fisherz`	✗		`hsic`	▲
	`mc_fisherz`	✗	GRaSP [121]	`CV_general`	✓
	`kci`	▲		`marginal_general`	✗
	`chisq`	✓		`CV_multi`	✗
	`gsq`	●		`marginal_multi`	✗
	`d_separation`	✗		`BIC`	✗
FCI [208]	`fisherz`	✗		`BDeu`	✓
	`kci`	▲	GES [46]	`CV_general`	▲
	`chisq`	✓		`marginal_general`	✗
	`gsq`	●		`CV_multi`	▲
LiNGAM [198]		✗		`marginal_multi`	✗
				`BIC`	✗
Exact	`dp` [199]	✗		`BIC_from_cov`	✗
	`astar` [251]	✗		`BDeu`	✓

Analysis. The three challenges of log data from Section 2.1 directly lead to these failures:

1. **Functional Dependencies → Violated Assumptions**: Many of the 16 errors produced relate to violated assumptions, most notably due to a singular correlation matrix among the variables, which leads to errors for some conditional independence tests. This singularity is an expression of the functional dependencies in log data. While such dependencies could be automatically eliminated, this could unintentionally eliminate variables crucial to system understanding in favor of less interpretable alternatives. For example, in PostgreSQL, removing all the variables associated with the configuration knobs from the causal graph, because the session ID is sufficient to determine them, would clearly be unhelpful for Alex.

2. **Large Number of Variables → Prohibitive Running Time**: Even when the assumptions are met, deriving a full causal graph can be time-consuming; many of the algorithms offer exponential complexity, and log-derived datasets can contain hundreds to thousands of variables. When diagnosing a production problem, such scaling is unacceptable.

3. **Biased Data Collection → Non-Actionable Graphs**: Even when the algorithm finishes quickly, the resulting graph may not cover all variables of interest. As seen above, no algorithm places query latency, `work_mem`, and `max_parallel_workers` in the same connected component, with some not even including all of these variables in the graph. This is because causal discovery algorithms capture the causal relationships *most salient in the data*, but the biased logging may mean that these relationships are not the ones *most interesting to the user*, for which there may be only weaker evidence.

2.2.3 Problem Definitions

As seen in Section 2.2.2, fully automatic causal discovery algorithms fall short when applied to log-derived data. At the same time, although the user *could* fully specify the causal graph $\mathcal{G}_{real}$, this would be impractical for large $\mathbf{V}$ as it would require $O(|\mathbf{V}|^2)$ causal *judgments*:

Definition 2.7: Judgment

A user judgment for (A, B) outputs whether the causal graph should include $A \to B$, $A \leftarrow B$, or neither.

However, many of the judgments produced in such an exhaustive pass would neither help discover the root cause T of a variable of interest O, nor significantly impact the value of $ATE(T, O)$ once T is specified. The latter is true because many edges would be entirely irrelevant when assessing the backdoor criterion (Definition 2.6).

This observation reveals a path to reducing the work required by the user, bridging manual graph construction and automatic causal discovery: using the log data to *solicit user judgments in the most impactful order*. Ensuring a high-quality ordering directly translates to reduced user effort for a given target downstream quality.

Given this approach, we present the two problems targeted by our work, each addressed by one of our framework's human-in-the-loop modules:

Problem 2.8: Candidate Cause Ranking

Let O be a variable of interest. Rank highly relevant candidate causes for O based on the available observations and solicit user judgments in this order, aiming to maximize *average precision (AP)*, defined as:

$$AP = \frac{\sum_{i=1}^{ranking\_length} \textsc{PrecisionAmongTop}(i) \cdot \textsc{IsPositive}(i)}{\#\text{ Ground Truth Positive Elements}}$$

Problem 2.9: Interactive Causal Graph Refinement

From an initial causal graph $\mathcal{G}_{init}$, minimize the number of user judgment solicitations needed to converge to the ground truth value of an ATE of interest, $ATE(T, O)$.

2.3 Overview of LOGos

We now present our framework, illustrated in Figure 2.2. Input logs are transformed by the Log Converter into a tabular dataset, from which the Candidate Cause Ranker and the Interactive Causal Graph Refiner help derive a partial causal graph with a human in the loop. The tabular dataset and the causal graph can then be used to answer ATE queries.

Diving deeper, our human-in-the-loop modules both operate on a tabular dataset D. Although deriving *some* tabular dataset from the log can be easily achieved using a log parsing algorithm [89, 258, 274], there are two usability issues not covered by this approach:

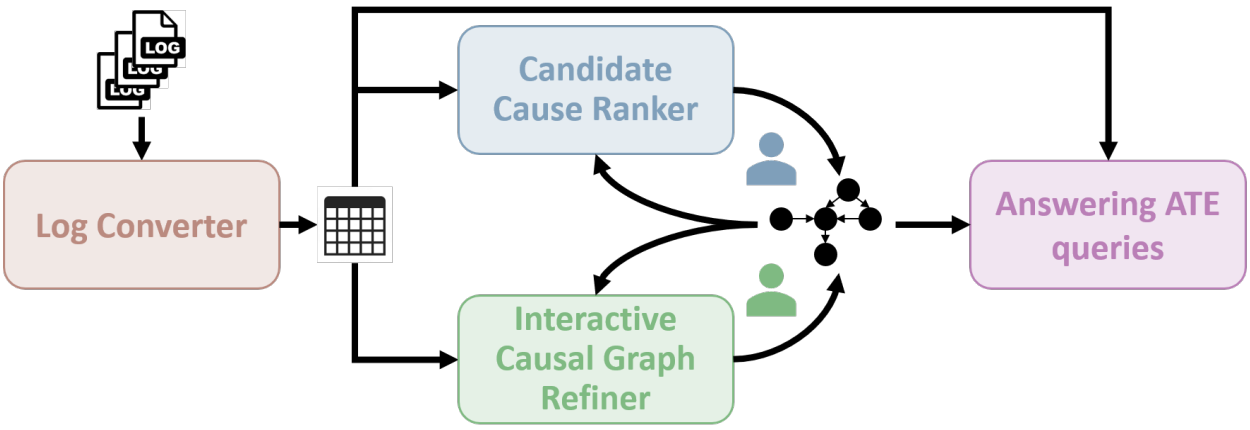


Figure 2.2: An architectural overview of our framework.

- **Opaque Variables**: Most log parsing algorithms *identify* variables but do not assign them meaningful names, which users need in order to reason about them interactively.
- **Per-Message Granularity**: Log parsing creates a record per log message, but users are interested in questions about larger units: sessions, machines, etc.

The Log Converter (Section 2.4) provides a data preparation pipeline that addresses these issues. It utilizes large language models to derive meaningful tags for each log-derived variable, provides the ability to aggregate log information over meaningful causal units, and computes entropy-maximizing aggregates of each variable. The resulting dataset D serves both as the input to our human-in-the-loop modules and as the second input to Pearl's model for ATE calculation alongside the causal graph.

The Candidate Cause Ranker (Section 2.5) then operates on the tabular dataset D to help address Problem 2.8. The user can invoke the Candidate Cause Ranker on a variable E, which we will refer to as the *effect variable*, yielding a ranked list of candidate causes. By producing a judgment for each of these causes C_i, the user can optionally add the corresponding edge $C_i \to E$ to the causal graph. The user can then apply this process recursively by first invoking Candidate Cause Ranker on O, then on one of the C_i for which $C_i \to O$ was added to the graph, and so forth, until a satisfactory root cause T is discovered and the user can pose their ATE query of interest: $ATE(T, O)$.

However, the user has so far only unveiled *one* directed path from T to O, which means that computing $ATE(T, O)$ may be erroneous. Per Section 2.2, correctly computing $ATE(T, O)$ from an observational dataset like D requires a valid adjustment set, which can be derived from a causal graph using the backdoor criterion. But to apply the backdoor criterion correctly, one must have access to a sufficiently refined graph, which includes all the T-backdoor paths between T and O. The Interactive Causal Graph Refiner (Section 2.6) helps construct this refined graph efficiently. It finds impactful adjustment set edits and transforms them to sets of causal graph edits, soliciting the corresponding judgments from the user. This helps the user progressively narrow the possible range of values for $ATE(T, O)$.

2.4 Log Converter

The two human-in-the-loop modules operate on a tabular dataset D. We will begin by describing the pipeline that transforms a log into such a dataset efficiently. This dataset also serves as the second input to Pearl's model for ATE calculation (alongside the causal graph built using the interactive modules of LOGos).

2.4.1 Log Parsing

Text logs must first be *parsed* into a table of variable observations:

Definition 2.10: Parsed Table

The parsed table has a row per log message and a column per log-derived parsed variable.

To generate the parsed table, we go through five steps:

1. Users can specify a log message prefix (e.g., a timestamp regular expression), to collate multi-line log messages, when present.

2. Users can pass zero or more regular expressions to parse formatted variables (e.g., IDs) from each log message, ensuring consistent parsing across different log templates.

3. An off-the-shelf parsing algorithm [89, 258, 274] extracts the rest of the parsed variables from their *log templates*. Any "classical" [66, 90, 91, 100] or learning-based [22, 53, 58, 132, 152, 158, 210, 237] log parsing algorithm is acceptable, as long as it leads to a parsed table that conforms with Definition 2.10. At times, log parsing algorithms may incorrectly map semantically different variables to the same parsed variable – our implementation provides functionality for the user to correct this, if needed.

4. We also add a binary indicator variable per log template, which takes the value 1 only for rows that match the template.

5. 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, similar to past work [241].

2.4.2 Variable Tagging

For users to reason about parsed variables, the variables must have *human-understandable names*. We will call these names *tags*. 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 to generate a tag:

1. The three tokens preceding the variable in the log template, if the variable is preceded by ':' or '=' (possibly with an interspersed opening quote).

2. GPT-3.5-Turbo (`gpt-3.5-turbo-1106`) [164], providing (a) an example message where the variable appears, (b) the 3 tokens that precede the variable, and (c) 4 additional example values for the variable from other log messages of the same template. While our prompt (available online [144]) delivers acceptable results, we acknowledge that the prompting technique and/or model can affect output quality [113, 114, 128, 129]. Fully exploring this problem is not a focus of our work.

3. GPT-4 (`gpt-4-0613`) [162], using the same prompt as above.

As soon as one source produces a tag, we move on to the next parsed variable. If no source produces a tag, we assign a unique string of symbols. Users can later edit these tags manually. 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.

2.4.3 Causal Unit Definition

The rows of the parsed table, one per log message, are not ideal for causal inference. We instead need a row per "data point" as defined for the question at hand (e.g. per session). We achieve this by bundling groups of log messages together into *causal units*.

Since the right choice of causal unit inherently depends on the question the user is interested in, we defer to them for a causal unit definition. Such a definition consists of a parsed variable W, on the value of which the units will be based, and optionally a discretization function (e.g., a binning function) if W is continuous.

However, not *every* choice of causal unit is permissible, because Definition 2.1 requires the Stable Unit Treatment Value Assumption (SUTVA) [182]: the outcome of each unit should not depend on the treatments of *other units*. For example, processes may be unsuitable causal units if they share hardware: process A's work could impact process B's latency. We defer to the user's knowledge of the system to ensure that this condition is satisfied.

2.4.4 Aggregation

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 session, the number of query latency readings for each session varies. To make causal units comparable, we must replace varying-size sets of values with *the same value(s) per causal unit*:

Definition 2.11: Prepared Table

The prepared table includes one row per causal unit and one column per prepared variable.

Each prepared variable is derived from some parsed variable, its *base variable*, by using an aggregation function (e.g. the mean) to reconcile the values within each causal unit. If the user is confident that a certain aggregation function is best for a particular parsed variable, they can explicitly specify it. Otherwise, one cannot automatically find the most useful aggregation function for each parsed variable a priori, since the downstream ATE question is

not yet 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. We therefore approximate this objective by trying to maximize the information captured by each prepared variable. We achieve this in two steps.

First, we compute multiple aggregates of each parsed variable $W \in \mathbf{W}$, based on its type. For numerical variables, we consider $\mathbf{A}_W = [max, min, mean]$. For categorical variables, we consider $\mathbf{A}_W = [first, last, mode]$, 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.

Then, for a parsed variable W, we let $R(W, a)$ be the range of $a(W)$, $a \in \mathbf{A}_W$. Among the prepared variables with W as their base variable, we keep the one that maximizes empirical entropy; that is, we keep the one resulting from $a^*_W(W)$, $a^*_W \in \mathbf{A}_W$, where:

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

We finally one-hot encode any categorical variables in this set.

2.4.5 Imputation

In each causal unit, there may be parsed variables with no observations. Ignoring causal units with missing values can result in *selection bias*. Luckily, imputing missing values in the prepared table is often possible, since the missing values are interpretable. 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 specify it if they so wish, performing no imputation by default.

2.4.6 Computational Complexity

Log Parsing. Depends on the chosen existing parsing algorithm.

Variable Tagging. For tagging, we require $O(M)$ time and $O(|\mathbf{W}|)$ space for the metadata fed into the GPT prompts, where M is the number of log messages and $\mathbf{W}$ 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 $O(|\mathbf{W}|)$ time and up to $2|\mathbf{W}|$ GPT calls to compute the tags, which take $O(|\mathbf{W}|)$ space to store. Since tagging is a per-variable operation, linear complexity is optimal for this problem.

Causal Unit Definition. Defining causal units takes $O(1)$ time.

Aggregation. Aggregation takes $O(|\mathbf{C}||\mathbf{W}|T_{max})$ time, where $\mathbf{C}$ is the set of causal units, $\mathbf{W}$ 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 $C \in \mathbf{C}$ with $|C|$ elements. The time complexity of one-hot encoding is $O(|\mathbf{C}||\mathbf{W}_{cat}|)$, where $\mathbf{W}_{cat}$ is the set of categorical parsed variables.

Imputation. Imputation with fixed values takes $O(|\mathbf{C}||\mathbf{W}|)$ time.

Algorithm 2.1 Rank candidate causes for an effect variable E.

Inputs: Dataset D, effect variable E
Outputs: A ranked list of candidate causes for E.

```
1: function ExploreCandidateCauses(D, E)
2:     X ← D \ E
3:     X, scale ← StandardScaleColumns(X)
4:     coefs ← LASSO(X, E)
5:     unscaled_coefs ← Unscale(coefs, scale)
6:     res = ∅
7:     for V ∈ X do
8:         if unscaled_coefs(V) ≠ 0 then
9:             slope, p_value ← OLSLinearRegression(V, E)
10:            res ← res ∪ {V, slope, p_value}
11:    return SortAscending(res, by=p_value)
```

2.5 Candidate Cause Ranker

The Candidate Cause Ranker helps discover a directed path from a sufficiently informative root cause T to the outcome variable of interest O "backwards", letting the user eventually specify the *causal question* they want to answer quantitatively – the $ATE(T, O)$.

2.5.1 Pruning and Ranking Candidate Causes

While using this module, the user will need to make judgments based on the returned ranking, so we need the ranking to be high-relevance. We achieve this through *edge pruning*. The guiding principle is as follows: the user's knowledge of the system is invaluable for distinguishing *which quantitative relationships* between pairs of variables are indeed manifestations of system-related causal mechanisms, as opposed to spurious correlations; however, the *absence of a quantitative relationship* can be a reliable indicator that a causal relationship does not exist, as long as the dataset is fairly representative of practical cases [107, 172].

In particular, note from Definition 2.2 that each variable is linearly related to its parents, up to its own error term. As such, if we run a multivariate regression of variable E on the remaining variables, and some variable V gets assigned a zero coefficient, we can disregard $V \to E$: V does not impact E with its fluctuations, given the other variables. If we further assume that the causal graph is sparse, a common assumption in causal discovery literature [49, 207], we can prune more edges by using a *sparse regression* algorithm. This will drive very small coefficients, likely artifacts of data availability rather than indicators of causality, to zero. This process requires no user input and typically prunes the vast majority of edges, since most variables are pairwise unrelated.

Concretely, we use LASSO [215], as presented in Algorithm 2.1. First, we normalize all variables except E by subtracting the mean and scaling by the standard deviation (lines 2–3) and apply LASSO to predict E (line 4) before undoing the normalization (line 5). We only perform further work for each variable V selected by LASSO (lines 7–8). In particular, we apply an ordinary least-squares (OLS) linear regression to each candidate causal relationship,

Algorithm 2.2 Compute allowed one-step causal graph edits.

Inputs: Causal graph $\mathcal{G}$, set of fixed edges $\mathbf{F}$, set of banned edges $\mathbf{B}$
Outputs: A set $\mathbf{S}$ of causal graph edits.

```
1:  function OneStepEdits(G, F, B)
2:      S ← ∅
3:      for (V, W) ∈ V × V do
4:          if (V → W) ∈ G and (V → W) ∉ F then
5:              S ← S ∪ {(V → W, Remove)}
6:          else if (V → W) ∉ G and (V → W) ∉ B then
7:              if (W → V) ∉ G then
8:                  S ← S ∪ {(V → W, Add)}
9:              else if (W → V) ∈ G and (W → V) ∉ F then
10:                 S ← S ∪ {(W → V, Flip)}
11:     return S
```

recording the slope and p-value (lines 6, 9–10). The results are sorted by increasing p-value (line 11), placing the candidates with the most reliable relationships to E (lowest p-value) at the top. The user can then inspect each returned candidate V and decide whether to add $V \to E$ to the causal graph. By iteratively calling ExploreCandidateCauses(E), the user can arrive at a satisfactory "root cause" variable T.

2.5.2 Computational Complexity

The complexity of Algorithm 2.1 is derived from those of LASSO [215] and ordinary least squares linear regression. If the threshold for convergence of the objective function is σ, a LASSO solution can be found in $O(|\mathrm{D}||\mathbf{V}|\sigma^{-1/2})$ [29, 265], where $|\mathrm{D}|$ is the number of data points and $\mathbf{V}$ is the set of variables. Each regression requires $O(|\mathrm{D}||\mathbf{V}|^2 + |\mathbf{V}|^3)$ time, so the overall complexity is $O(|\mathrm{D}||\mathbf{V}|\sigma^{-1/2} + m|\mathrm{D}||\mathbf{V}|^2 + m|\mathbf{V}|^3)$, if LASSO finds m candidates.

2.6 Interactive Causal Graph Refiner

Having found *which* ATE to calculate ($ATE(T, O)$), we next focus on calculating it *correctly* with respect to the underlying causal dynamics of the system at hand. This is the purpose of the Interactive Causal Graph Refiner, which solicits a sequence of additional causal judgments from the user. Doing so efficiently is important in order to minimize the required number of human interactions before the ATE can be accurately calculated.

2.6.1 Preliminary Strategies

To motivate the actual approach taken by the Interactive Causal Graph Refiner, we will briefly introduce two preliminary strategies and discuss their advantages and disadvantages. Section 2.8.4 also compares these strategies quantitatively. Both of these preliminary strategies make use of an auxiliary algorithm, OneStepEdits (Algorithm 2.2), which exhaustively

Algorithm 2.3 Solicit a judgment based on the most impactful single-edge graph edit.

Inputs: Causal graph $\mathcal{G}$, dataset D, treatment variable T, outcome variable O, set of fixed edges $\mathbf{F}$, set of banned edges $\mathbf{B}$

Outputs: A causal graph edit s about which to solicit a user judgment.

```
1: function SingleEdit(G, D, T, O, F, B)
2:     ate ← ATE(G, D, T, O)
3:     S ← OneStepEdits(G, F, B)
4:     M ← ∅
5:     for s ∈ S do
6:         G' ← ApplyEdits(G, {s})
7:         ate' ← ATE(G', D, T, O)
8:         M[s] ← |ate' − ate|
9:     return arg max_{s∈M} M[s]
```

collects the set of allowed single-edge causal graph edits. It operates on a causal graph $\mathcal{G}$, a set of *fixed* edges $\mathbf{F}$ (which should not be removed from the graph) and a set of *banned* edges $\mathbf{B}$ (which should not be added to the graph). For each node pair in $\mathcal{G}$, $(V, W) \in \mathbf{V} \times \mathbf{V}$:

1. If $V \to W$ is part of $\mathcal{G}$ and not fixed, collect an edge edit that removes it (lines 4–5).
2. If $V \to W$ is not part of $\mathcal{G}$ and not banned (line 6), only collect an edge edit that adds it if permitted by the status of its inverse edge $W \to V$ (lines 7–10).

2.6.1.1 Preliminary Strategy 1: Single Edge Edit

The first strategy, called SingleEdit, is listed in Algorithm 2.3. It is a simple greedy wrapper over OneStepEdits: it finds the single-edge causal graph edit which, when applied to the input graph $\mathcal{G}$, will result in the *maximum absolute change* in $ATE(T, O)$. In particular, it calculates the initial ATE of T on O using $\mathcal{G}$ (line 2) and then calculates the absolute ATE impact of each edit collected by OneStepEdits (lines 3–8). It then returns the edit with the maximum absolute impact (line 9), about which the user is asked to provide a judgment.

SingleEdit is straightforward, but it runs the risk of getting stuck in local minima. For example, there are cases where adding *two edges* would have affected the ATE of interest by creating a new path in the causal graph, but no single edge edit can affect the ATE. Since SingleEdit maximizes the ATE impact of *an individual edge edit* at a time, it will overlook such cases. This motivates our next preliminary strategy.

2.6.1.2 Preliminary Strategy 2: Exploring Edge Edits with the A* Algorithm

The next preliminary strategy, HeuristicEdit, changes the way suggested edits are scored, by relying on the A* heuristic search algorithm [85] to identify edges that may be "lower-impact" individually, but which lie along a promising "edit path". Heuristic searches have traditionally been a critical tool to tackle computationally hard problems; for example, simulated annealing for the traveling salesperson problem [1] or evolutionary algorithms for feature selection [2].

Algorithm 2.4 Solicit a judgment based on heuristic search scoring.

Inputs: Causal graph $\mathcal{G}$, dataset D, treatment variable T, outcome variable O, set of fixed edges $\mathbf{F}$, set of banned edges $\mathbf{B}$

Configuration: Computational budget b, number of top graphs to consider k, number of edges to return m

State: A cache of edits on which to solicit judgments next, *cache*.

Outputs: A causal graph edit s about which to solicit a user judgment.

function HEURISTICEDIT($\mathcal{G}$, D, T, O, $\mathbf{F}$, $\mathbf{B}$)
  if $cache = \emptyset$ **then**
    $frontier \leftarrow$ min-PRIORITYQUEUE
    $frontier$.PUSH($\mathcal{G}, 0$)
    $predecessor\_and\_edge \leftarrow \emptyset$
    $visited \leftarrow \emptyset$
    $seen \leftarrow \emptyset$
    while $frontier \neq \emptyset$ **and** $|visited| < b$ **do**
      $\mathcal{G}' \leftarrow frontier$.POP()
      if $\mathcal{G}' \in visited$ **then**
        continue
      $\mathbf{S} \leftarrow \{s \mid s \in \text{ONESTEPEDITS}(\mathcal{G}', \mathbf{F}, \mathbf{B}) \wedge (s.\text{type } \neq \text{FLIP})\}$
      for $s \in \mathbf{S}$ **do**
        $\mathcal{G}'' \leftarrow$ APPLYEDITS($\mathcal{G}'$, $\{s\}$)
        $score \leftarrow \Phi(\mathcal{G}) - \Phi(\mathcal{G}'')$
        $predecessor\_and\_edge[\mathcal{G}''] \leftarrow (\mathcal{G}', s)$
        $frontier$.PUSH($\mathcal{G}'', score + h(\mathcal{G}'')$)
        $seen \leftarrow seen \cup \{\mathcal{G}''\}$
      $visited \leftarrow visited \cup \{\mathcal{G}'\}$
      $seen \leftarrow seen \cup \{\mathcal{G}'\}$
    $top\_phi\_graphs \leftarrow \underset{\mathcal{H} \in seen}{\text{TOP}} \big(\Phi[\mathcal{H}], k\big)$
    $counts \leftarrow \emptyset$
    for $\mathcal{H} \in top\_phi\_graphs$ **do**
      $\mathcal{H}' \leftarrow \mathcal{H}$
      while $\mathcal{H}' \neq \mathcal{G}$ **do**
        $(\mathcal{P}, s) \leftarrow predecessor\_and\_edge[\mathcal{H}']$
        $counts[s] \leftarrow counts[s] + 1$ **if** $s \in counts$ **else** 1
        $\mathcal{H}' \leftarrow \mathcal{P}$
    $cache \leftarrow \underset{s \in \text{KEYS}(counts)}{\text{TOP}} \big(counts[s], m\big)$
  $s \leftarrow$ POPHEAD($cache$)
  return s

We begin by defining $\mathbf{\Gamma}$ as a *meta-graph of causal graphs*. Each node in $\mathbf{\Gamma}$ is a causal graph over the variables in $\mathbf{V}$ that respects the fixed list $\mathbf{F}$ and the banned list $\mathbf{B}$. Each pair of nodes $\mathcal{G}_m, \mathcal{G}_n \in \mathbf{\Gamma}$ is either nonadjacent or connected by two oppositely directed edges. If they exist, these meta-graph edges imply that there is a pair of variables $(V, W) \in \mathbf{V} \times \mathbf{V}$ such that $\mathcal{G}_m$ and $\mathcal{G}_n$ only differ by $V \to W$. Each directed edge in the meta-graph $\mathbf{\Gamma}$ has a *type*, consisting of a causal graph edge and an operation. For example, if $(V \to W) \notin \mathcal{G}_m$ and $(V \to W) \in \mathcal{G}_n$, then the meta-graph edge $\mathcal{G}_m \to \mathcal{G}_n$ has type $(V \to W, \textsc{Add})$, while $\mathcal{G}_m \leftarrow \mathcal{G}_n$ has type $(V \to W, \textsc{Remove})$.

Our algorithm starts from the input causal graph $\mathcal{G}$ and explores the meta-graph $\mathbf{\Gamma}$ towards the causal graph $\mathcal{G}^\star$ that maximizes the potential function:

$$\Phi(\mathcal{H}) = |\text{ATE}(\mathcal{H}, \text{D}, T, O) - \text{ATE}(\mathcal{G}, \text{D}, T, O)| + \gamma \cdot \textsc{Quality}(\mathcal{H})$$

The function Quality uses heuristic measures of the candidate causal graph, such as sparsity, to regularize the search. In our current implementation, we use the closeness of the sparsity to a tree. γ decides the strength of the regularization. In principle, the potential function can be further elaborated to take into account the quality of the model's fit to the data or other similar confidence measures. Let $\mathcal{G}^\star = \arg\max_{\mathcal{H} \in \mathbf{\Gamma}} \Phi(\mathcal{H})$, but note that $\mathcal{G}^\star$ is not known.

In $\mathbf{\Gamma}$, the weight of an edge $w(\mathcal{G}_m \to \mathcal{G}_n)$ is defined as $\Phi(\mathcal{G}_m) - \Phi(\mathcal{G}_n)$. This way, minimizing the cost of a path corresponds to maximizing the potential of the frontier node of the path (by telescoping). During a search starting from $\mathcal{G}$, when the frontier of the search path is $\mathcal{G}'$, A^* requires an estimate $h(\mathcal{G}'')$ of the cost of the *remaining path* from $\mathcal{G}''$ to $\mathcal{G}^\star$ for each neighbor $\mathcal{G}''$ of $\mathcal{G}'$, in order to decide which neighbor to explore next. To ensure that the heuristic strategy is *admissible* [85], let $h(\mathcal{G}'') = 2(\Phi(\mathcal{G}'') - \sum_{\mathcal{G} \in \Gamma} \Phi(\mathcal{G}))$. Since the remaining path cost is $\Phi(\mathcal{G}'') - \Phi(\mathcal{G}^\star)$, $h(\cdot)$ does not overestimate its value.

As Algorithm 2.4 shows, we set a computational budget b and run A* over $\mathbf{\Gamma}$ for b steps (lines 3–20); that is, we expand b nodes of the meta-graph. At the end, the algorithm takes the k explored graphs with the greatest potentials (line 21) and examines the paths from the initial graph $\mathcal{G}$ to these k candidates. We count the number of each type of search graph edge encountered (lines 22–28) and cache the m most common types to return as the suggested causal graph edits to the user, one at a time (lines 2, 29–31). Only after the user has issued the corresponding m judgments is HeuristicEdit invoked again to replenish the list of suggested causal graph edits.

The HeuristicEdit strategy showcases one way of avoiding local minima: exploring "farther away" from the starting graph in order to score the impact of each individual edge edit. However, the heuristic search in HeuristicEdit can incur a significant computational cost for large variable sets.

2.6.2 Adopted Strategy: Adjustment Set Edit

Our final adopted strategy instead works over the space of possible single-variable *adjustment set* edits, which is linear in $|\mathbf{V}|$. The intuition here is to take a similar approach as SingleEdit but apply it to the adjustment set instead of the graph edges. Namely, this strategy tries to identify the single most impactful *adjustment set edit* instead of the single most impactful edge edit, in terms of producing a large absolute ATE difference.

Algorithm 2.5 Return the next judgment to be solicited, based on the adjustment set edit that would impact $ATE(T, O)$ maximally.

Inputs: Causal graph $\mathcal{G}$, dataset D, treatment variable T, outcome variable O, set of fixed edges $\mathbf{F}$, set of banned edges $\mathbf{B}$

State: A *cache* of edits for which to solicit judgments next, a graph $\mathcal{G}_{next}$ that would result if the most recent user judgment agreed with the graph edit based on which it was solicited.

Outputs: A causal graph edit s about which to solicit a user judgment.

```
1:  function SolicitJudgment(G, D, T, O, F, B)
2:      if cache = ∅ or G ≠ G_next then
3:          ate ← ATE(G, D, T, O)
4:          base_adj_set ← AdjSet(G, T, O)
5:          best_edits ← ∅
6:          best_ate_impact ← 0
7:          for V ∈ (Nodes(G) \ {T, O}) do
8:              S ← ∅
9:              if V ∈ base_adj_set then
10:                 S ← MapRemoval(G, D, T, O, F, B, V)
11:             else
12:                 S ← MapAddition(G, D, T, O, F, B, V)
13:             G' ← ApplyEdits(G, S)
14:             ate' ← ATE(G', D, T, O)
15:             ate_impact ← |ate' − ate|
16:             if ate_impact > best_ate_impact then
17:                 best_edits ← S
18:                 best_ate_impact ← ate_impact
19:         cache ← best_edits
20:     s ← PopHead(cache)
21:     G_next ← ApplyEdits(G, {s})
22:     return s
```

The next judgment to be solicited from the user is provided by SolicitJudgment (Algorithm 2.5). It maintains two pieces of state: *cache* contains cached suggested edits from previous invocations, and $\mathcal{G}_{next}$ contains the causal graph that would result if the most recent user judgment agreed with the most recent suggested edit. Unlike the edits produced by HeuristicEdit, the set of edits produced by AdjSetEdit only makes sense *as a collection*; therefore, we only use the cache for as long as the user's judgments agree with the suggested edits (line 2). If the user disagrees (e.g. we suggest removing $V_i \to V_j$, but the user keeps this edge in the causal graph and fixes it), we discard the cache and re-run the algorithmic core.

In that case, the algorithm calculates the ATE(T, O) given $\mathcal{G}$ and D and finds a valid adjustment set using existing algorithms [214] (lines 3–4). It then considers each variable in D except T and O (line 7) and calls MapRemoval (Section 2.6.2.1) or MapAddition (Section 2.6.2.2) to find edge edits that would change that variable's adjustment set membership (lines 8–12). Next, it applies these edits to a copy of $\mathcal{G}$, and calculates the resulting ATE(T, O) and the associated ATE impact (lines 13–15). It keeps track of the edits that

Algorithm 2.6 Suggest causal graph edits to remove a variable from the adjustment set.

Inputs: Causal graph $\mathcal{G}$, dataset D, treatment variable T, outcome variable O, set of fixed edges $\mathbf{F}$, set of banned edges $\mathbf{B}$, variable to remove from the adjustment set V

Outputs: A set $\mathbf{S}$ of causal graph edits.

```
1:  function MapRemoval(𝒢, D, T, O, F, B, V)
2:      S ← ∅
3:      if V ∈ An(T) then
4:          S, success ← BreakPaths(𝒢, F, {V}, T )
5:          if not success then
6:              return ∅
7:      T ← De(T) ∪ {T}
8:      T ← SortAscending(T, by= DirPathLengthFrom(T))
9:      for W ∈ T do
10:         if (W → V) ∉ B then
11:             if (V → W) ∉ 𝒢 then
12:                 S ← S ∪ (W → V, Add)
13:             else
14:                 S ← S ∪ (V → W, Flip)
15:             return S
16:     return ∅
```

produce the maximum ATE impact and caches them (lines 5–6, 16–19). At this point a valid cache exists, so the algorithm removes its first element and computes the resulting graph, if the edit is applied (lines 20–21). The edit is then returned for the user to judge (line 22).

The algorithmically hardest part is deriving a set of causal graph edits that corresponds to the desired effect on the adjustment set – i.e. the functionality implemented by MapRemoval and MapAddition. In general, multiple such sets may exist that achieve the same effect – e.g. removing a certain variable from the adjustment set. We present one possible approach to finding such a set, based on Definition 2.6.

2.6.2.1 The MapRemoval Algorithm

MapRemoval (Algorithm 2.6) finds a set of edge edits that removes V from the adjustment set. It achieves this by making V a descendant of T, which means that V would violate the first condition of Definition 2.6 if it remained part of the adjustment set. There are two cases:

1. $V \notin \mathbf{An}(T)$**.** V can be easily made a descendant of T by adding a directed edge from T or one of its descendants (lines 7–15). We prioritize suggesting the option that minimizes the directed distance from T (lines 8–9) while not being banned (line 10).

2. $V \in \mathbf{An}(T)$**.** In this case, making V a descendant of T would create a cycle. To fix this, we must first remove V from $\mathbf{An}(T)$ by calling BreakPaths (explained in Section 2.6.2.3) to find edits breaking all directed paths from V to T (lines 3–4). If BreakPaths cannot find such edits, MapRemoval returns, since we cannot include V in $\mathbf{De}(T)$ without creating a cycle (lines 5–6).

Algorithm 2.7 Suggest causal graph edits to include a variable in the adjustment set.

Inputs: Causal graph $\mathcal{G}$, dataset D, treatment variable T, outcome variable O, set of fixed edges $\mathbf{F}$, set of banned edges $\mathbf{B}$, variable to add to the adjustment set V

Outputs: A set $\mathbf{S}$ of causal graph edits.

```
1: function MapAddition(𝒢, D, T, O, F, B, V)
2:     S ← ∅
3:     if V ∈ De(T) or V ∈ De(O) then
4:         S, success ← BreakPaths(𝒢, F, {T, O}, V)
5:         if not success then
6:             return ∅
7:     S ← S ∪ {(V → T, Add), (V → O, Add)}
8:     return S
```

2.6.2.2 The MapAddition Algorithm

MapAddition (Algorithm 2.7) finds a set of edge edits that ensures that V is included in the adjustment set. It achieves this by creating a new path $T \leftarrow V \rightarrow O$, which must be blocked in order to satisfy the second condition of Definition 2.6 and can only be blocked by including V in the adjustment set. There are, again, two cases:

1. $V \notin (\mathbf{De}(T) \cup \mathbf{De}(O))$**.** We can directly create the path described above by adding edges $V \rightarrow T$ and $V \rightarrow O$ (lines 7–8).

2. $V \in (\mathbf{De}(T) \cup \mathbf{De}(O))$**.** We first call BreakPaths to break these descendant relationships, if possible (lines 3–4), returning early if not (lines 5–6).

2.6.2.3 The BreakPaths Algorithm

Both MapRemoval and MapAddition may require identifying a set of graph edits that breaks each directed path in a set of directed paths. One such set is a minimum cut in the subgraph consisting only of these paths, discoverable in time $O(|\mathbf{V}|^2|\mathbf{E}|)$ using Dinitz's algorithm [64]. However, since we do not need the set to be minimal, we present a more efficient algorithm, shown in Algorithm 2.8.

In particular, we start by identifying the subgraph $\mathcal{G}_{filtered}$, consisting only of directed paths from any of the nodes in **sources** to *sink*. We detect all the nodes that are both reachable from the **sources** and can reach *sink*, and remove all other nodes (and their incident edges) from $\mathcal{G}$ (lines 2–5). We then apply a variant of breadth-first search: we initialize a queue and visited list based on **sources** (lines 7–8) and start processing elements from the queue until it is empty (lines 9–10). For each processed element, we attempt to remove all directed edges to its children, if they are not fixed (lines 6, 13–15). If some edge among them is indeed fixed, we defer the breaking of that path by adding its children to the queue (lines 16–18). If we reach the sink node, it means that some directed path only contained fixed edges, so breaking it is infeasible; we therefore return an empty set of edits and a boolean False value (lines 11–12). Otherwise, once the queue is empty, we return the identified edits and a True value (line 19).

Algorithm 2.8 Break each directed path from one of the **sources** to the *sink*.

Inputs: Causal graph $\mathcal{G}$, set of fixed edges $\mathbf{F}$, set of source nodes **sources**, sink node *sink*
Outputs: A set $\mathbf{S}$ of causal graph edits.

```
1: function BreakPaths(G, F, sources, sink)
2:     reachable_from_sources ← BFS(G, from=sources)
3:     sink_reachable_from ← BFS(ReverseGraph(G), from=sink)
4:     nodes_to_keep ← reachable_from_sources ∩ sink_reachable_from
5:     G_filtered ← RemoveNodes(G, Nodes(G) - nodes_to_keep)
6:     S ← ∅
7:     queue ← sources
8:     visited ← sources
9:     while queue ≠ ∅ do
10:        V ← Dequeue(queue)
11:        if V = sink then
12:            return ∅, False
13:        for W ∈ Children(G_filtered, V) do
14:            if (V → W) ∉ F then
15:                S ← S ∪ {V → W, Remove}
16:            else if W ∉ visited then
17:                visited ← visited ∪ {W}
18:                Enqueue(queue, W)
19:    return S, True
```

2.6.3 Computational Complexity

When not using the cache, SolicitJudgment loops over $O(|\mathbf{V}|)$ variables, calling MapRemoval or MapAddition in each iteration. Each of those algorithms calls BreakPaths, which involves three breadth-first searches (lines 2–3 and 7–18) of complexity $O(|\mathbf{V}| + |\mathbf{E}|) = O(|\mathbf{V}|^2)$. MapRemoval also includes a sort by distance, also implementable using a breadth-first search in $O(|\mathbf{V}|^2)$, and a loop of complexity $O(|\mathbf{V}|)$. Each of MapRemoval and MapAddition therefore has complexity $O(|\mathbf{V}|^2)$. SolicitJudgment then calls ATE, which involves a linear regression with $O(|\mathrm{D}||\mathbf{V}|^2 + |\mathbf{V}|^3)$. The full complexity of SolicitJudgment is then $O(|\mathrm{D}||\mathbf{V}|^3 + |\mathbf{V}|^4)$.

2.7 Prototype System: LOGos

We implemented our framework in LOGos, using $\sim$ 2,850 lines of Python [144]. An earlier prototype was presented as Sawmill [145], but we revised the name because of a name clash with a different product [105]. For log parsing, we used Drain [91], based on an implementation by the authors of LogBERT [80]. 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 [174]. For estimating ATEs in GetATE, we used "backdoor.linear_regression" from DoWhy [31, 195], an open-source Python library that simplifies the statistical tasks involved in estimating causal effects, given the data and an associated causal graph.

2.8 Evaluation

After presenting our evaluation setup (Section 2.8.1), we evaluate four claims about our framework as implemented in LOGos:

- LOGos's Candidate Cause Ranker ranks the ground truth root causes higher than the baselines while remaining interactive, addressing Problem 2.8 (Section 2.8.2).
- LOGos's Interactive Causal Graph Refiner quickly leads the user to a graph on which $ATE(T, O)$ matches the ground truth, addressing Problem 2.9 (Section 2.8.3).
- LOGos's adopted strategy for soliciting user judgments in the Interactive Causal Graph Refiner, as presented in Section 2.6.2, outperforms the preliminary strategies presented in Section 2.6.1 (Section 2.8.4).
- The Log Converter scales gracefully (Section 2.8.5).

2.8.1 Evaluation Setup

2.8.1.1 Environment

All experiments were run on a machine equipped with two 20-core 2.10 GHz Intel Xeon Gold 6230 CPUs [106] and 256 GiB of memory, running Linux 6.9.7-arch1-1.

2.8.1.2 Metrics

Our evaluation focuses on the following metrics:

Average Precision (Candidate Cause Ranker, higher is better). Since this module addresses Problem 2.8, we evaluate it using the average precision of its output ranking r, defined as:

$$AP(r) = \frac{\sum_{i=1}^{\textsc{Length}(r)} \textsc{PrecisionAmongTop}(i) \cdot \textsc{IsPositive}(i)}{\#\text{ Ground Truth Positive Elements}}$$

Number of Judgments (Interactive Causal Graph Refiner, lower is better). Leveraging the Absolute Relative Error [9] in ATE, we report the number n of judgments to reach $\mathcal{G}_n$ for which:

$$ARE\_ATE(n) = \left| \frac{ATE_{\mathcal{G}_n}(T, O) - ATE_{\mathcal{G}_{\text{Real}}}(T, O)}{ATE_{\mathcal{G}_{\text{Real}}}(T, O)} \right| \leq 10^{-5}$$

We selected this metric instead of measuring the structural accuracy of the causal graph because a fully correct graph is not necessary to estimate the ATE correctly, as explained in Section 2.1.

Latency (lower is better). We evaluate whether the latency of our two human-in-the-loop modules is indeed interactive, as well as how the Log Converter scales to more complex logs.

Table 2.2: Statistics about our evaluation datasets.

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
	$V = 1{,}000$	1,000,000	65,943,179	1002	2001

2.8.1.3 Baselines

We compare LOGos to two baselines:

Regression. Produce suggestions using linear regressions.

- **For Candidate Cause Ranking:** Drop prepared variables with zero variance and normalize the rest to zero mean and unit variance. Regress the effect variable E on the normalized data and rank the regressors by decreasing absolute regression coefficient.
- **For Interactive Causal Graph Refinement:** For each variable V in the current causal graph, for each of its non-neighbors W, regress V on W. Solicit a user judgment for the pair with the highest absolute regression coefficient. Before running the regressions, normalize as above. Cache regression results and continue soliciting judgments in ranked order while the user makes no changes to the graph.

LangModel. Produce suggestions using `gpt-4o-mini-2024-07-18` [163] (prompts online [144]).

- **For Candidate Cause Ranking:** Prompt the model with the tags of all prepared variables and up to 3 unique example values for each, and have it rank causes for E.
- **For Interactive Causal Graph Refinement:** Prompt the model with the tags of all prepared variables, the edges in the current causal graph, and the pair (T, O). Ask it to rank pairs of variables based on their relevance to $ATE(T, O)$. Solicit a user judgment for the pair ranked first and cache the rest. Continue soliciting judgments in ranked order as long as the user makes no changes to the graph.

2.8.1.4 Datasets

Open-source collections of logs exist [273], but to evaluate LOGos we also need access to the *ground-truth causal effects* in the log-producing system, which are generally unavailable. We therefore develop three new datasets, summarized in Table 2.2, ranging from instrumenting a real system, through augmenting a real-world log with a known causal relationship, to generating synthetic logs for stress-testing. This strategy was also used by other causality-in-systems works, e.g., CausalSim [11]. Where applicable, we provide these datasets with our code [144]. For the human-in-the-loop modules, we use as inputs the prepared variables for each dataset.

Table 2.3: Parameter settings for generating the PostgreSQL dataset.

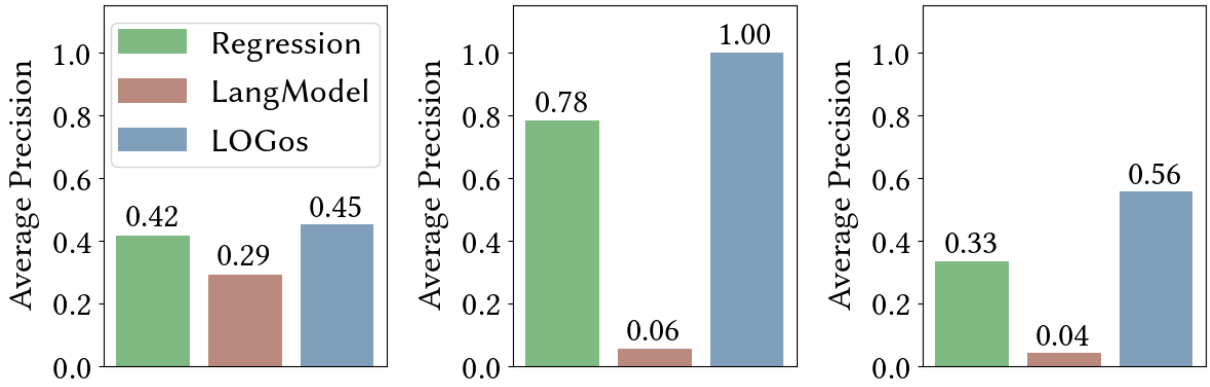
Parameter	Values
`work_mem`	128, 256
`seq_page_cost`	1, 1,000
`random_page_cost`	4, 1,000
`max_parallel_workers`	1, 2
`maintenance_work_mem`	32,768, 65,536
`effective_cache_size`	262,144, 524,288

PostgreSQL **(Real-world).** As mentioned in Section 2.2.2, 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 a t2.2xlarge instance on AWS EC2 [192] and configured it to log the latency of each query. We then loaded the TPC-DS benchmark [179] for scale factor 1 and executed a workload consisting of sequentially issuing one query per TPC-DS template, excluding 4 long-running templates (1, 4, 11, and 74).

We ran this workload 64 times, once per combination of the parameter settings in Table 2.3, each in a separate session. These parameters both impact query latency and do not require a Postgres service restart to reset. For each of the 32 parameter combinations where the product of `work_mem` and `max_parallel_workers` is 256, we then ran the workload once more, leading to 96 total runs. For a randomly selected run, the aforementioned product is therefore more likely to be 256, so we have confounding: `work_mem` affects both query latency and the value of `max_parallel_workers`.

Proprietary **(Real-world with post-processing).** 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 post-processed a real-world 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 had probability $p_n = 10\%$ of reporting error code 401 instead of success code 200. There are 36 such messages in the original 1,223-line log. For each faulty user, we equivalently defined a probability $p_f > p_n$ and we indicated their OS as iOS 15.0 in the log message that reported 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 was set to 100%, 50%, or 20%.

XYZ **(Synthetic).** Finally, we created a family of synthetic logs to evaluate the limits of our framework's ability to recover and adjust for confounding. We wanted a synthetic log so we could test how well LOGos 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 1,000 log messages for each of 1,000 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).



(a) PostgreSQL (b) Proprietary (c) XYZ

Figure 2.3: Average precision for Candidate Cause Ranking (higher is better).

While the value of most variables is drawn uniformly at random from $[0, 100]$ 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 $\sigma = R$, with X_m drawn once per machine uniformly at random. Y is equal to $2X + 3Z$, with added Gaussian noise with $\sigma = R$. As such, $ATE(X, Y) = 2$ after adjusting for Z. We generated 9 scenarios by setting V to 10, 100, or 1,000, and R to 1, 5, or 10.

2.8.2 Candidate Cause Ranking

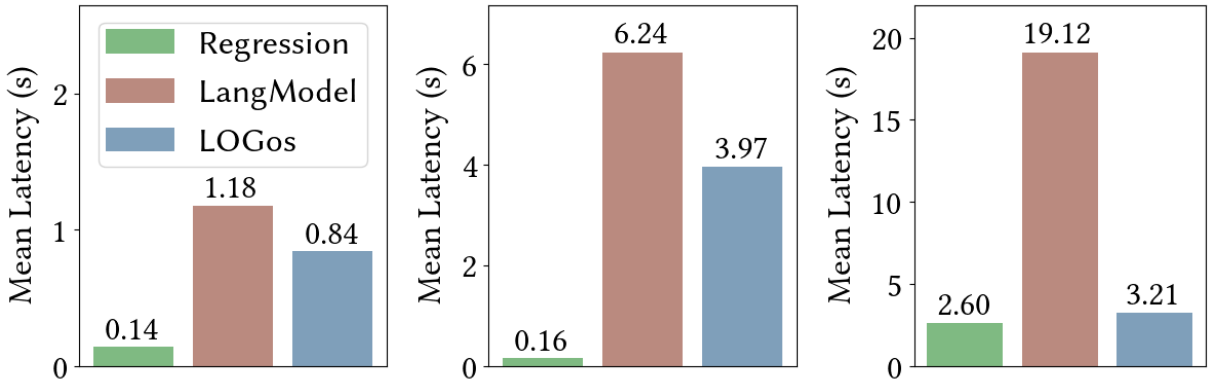
We begin our evaluation by assessing the ability of the Candidate Cause Ranker to produce high-quality rankings of candidate causes, therefore addressing Problem 2.8.

2.8.2.1 PostgreSQL

For this dataset, we explored candidate causes of query latency, represented by the prepared variable `duration mean`, and we aimed to recover the prepared variables associated with the two parameters involved in confounding: `work_mem mean` and `max_parallel_workers mean`. As shown in Figure 2.3a, the candidate cause ranking produced by LOGos achieved superior average precision in this task, beating Regression by 1.08× and LangModel by 1.54×. From Figure 2.4a we see that Regression offered a much lower latency, but the sub-second latency offered by LOGos was still firmly within the interactive range we are targeting.

2.8.2.2 Proprietary

For this dataset, we explored candidate causes of failure codes in HTTP responses, represented by the prepared variable `code mean`, and we aimed to recover the prepared variable for the OS version, tagged `version mean`. We present mean results over the 9 experimental scenarios. As shown in Figure 2.3b, LOGos outperformed Regression by 1.28× and LangModel by 18× in terms of average precision. The latency insight from Figure 2.4b is again that Regression offered very low latency, but LOGos remained interactive.



(a) PostgreSQL (b) Proprietary (c) XYZ

Figure 2.4: Candidate Cause Ranking latency (lower is better).

2.8.2.3 XYZ

Finally, for XYZ, we explored candidate causes of `Y mean`, and we aimed to recover `X mean` as the candidate cause. We present mean results over the 9 experimental scenarios. As shown in Figure 2.3c, LOGos once again led in average precision, beating Regression by 1.66× and LangModel by 13.47×. In terms of latency, Figure 2.4c once more shows that Regression produced results the fastest, but LOGos remained interactive.

Finding from 2.8.2: Candidate Cause Ranking

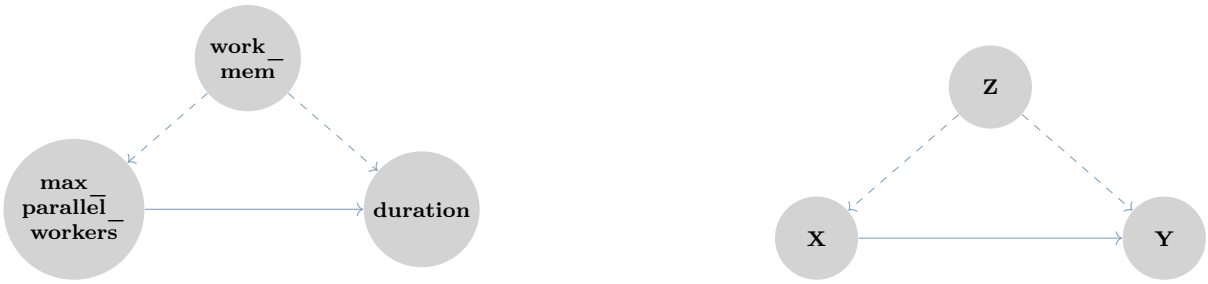
The average precision of the Candidate Cause Ranker is 1.08×–1.66× higher than Regression and 1.54×–18× higher than LangModel. A call to the Candidate Cause Ranker requires on average under 4 s, which remains interactive, although slower than Regression. LangModel is at least 40% slower than LOGos.

2.8.3 Interactive Causal Graph Refinement

2.8.3.1 PostgreSQL

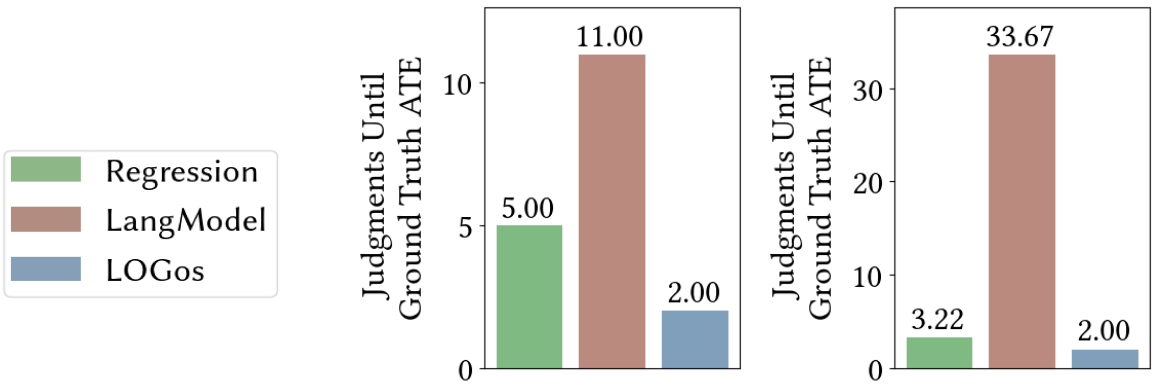
We calculated the ground truth value of the $ATE($`max_parallel_workers`, `duration`$)$ on the graph shown in Figure 2.5a. Then, we removed the dashed edges to derive a starting graph. For each available method, we responded to each judgment it solicited with the appropriate edge status based on the ground truth causal graph, and measured how many judgments were solicited before the ATE converged to its ground truth value. As seen in Figure 2.6a, LOGos was able to immediately zero in on the two dashed edges and solicit judgments for them, reaching the ground truth graph (and thus the ground truth ATE) in only 2 judgments. Regression instead required 5 judgments (2.50× more) to achieve the same goal, while LangModel required 11 judgments (5.50× more).

Moreover, as shown in Figure 2.7a, LOGos required only 1.90 s of total processing time for the 2 judgment solicitations, firmly within the interactive realm. While Regression was also interactive, LangModel required 5.76 s in total, 3.04× more than LOGos.



(a) PostgreSQL (b) XYZ

Figure 2.5: Ground truth causal graphs for Section 2.8.3.



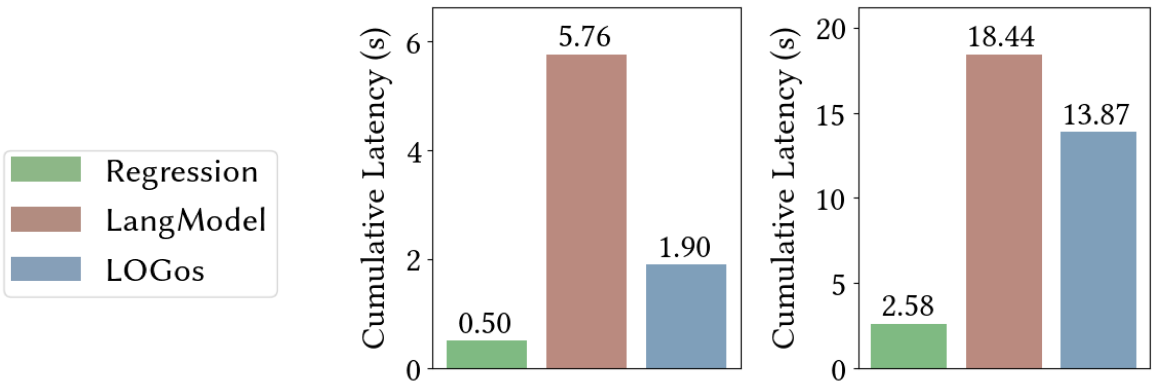
(a) PostgreSQL (b) XYZ

Figure 2.6: Number of judgments required to converge to ground truth ATE for Interactive Causal Graph Refinement (lower is better).

2.8.3.2 XYZ

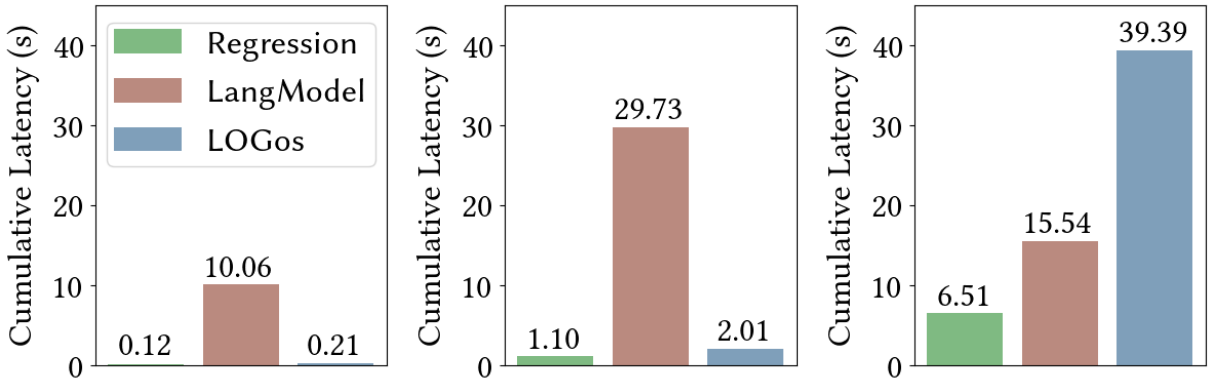
We calculated the ground truth $ATE($`X mean`, `Y mean`$)$ on the graph of Figure 2.5b. Then, we removed the dashed edges to derive a starting graph and followed the same strategy as in Section 2.8.3.1. We present mean results over the 9 experimental scenarios of this dataset. As seen in Figure 2.6b, LOGos again outperformed the competition, requiring exactly 2 judgment solicitations to identify the 2 dashed edges and help the user reconstruct the ground truth graph. Regression instead required 3.22 judgments on average (1.61× more) to achieve the same goal, while LangModel required 33.67 judgments (16.83× more).

As shown in Figure 2.7b, LOGos required on average 13.87 seconds of total processing time across the 2 judgment solicitations. However, as the breakdown by value of V in Figure 2.8 shows, this was mainly caused by the deliberately computationally intensive experimental scenarios with $V = 1{,}000$, which contain over 10× as many prepared variables as PostgreSQL and Proprietary (see Table 2.2). LOGos offered interactive latency otherwise. While Regression was interactive across experimental settings, LangModel required 18.44 seconds in total on average, 1.33× more than LOGos.



(a) PostgreSQL (b) XYZ

Figure 2.7: Cumulative latency for Interactive Causal Graph Refinement (lower is better).



(a) $V = 10$ (b) $V = 100$ (c) $V = 1{,}000$

Figure 2.8: Figure 2.7b split by value of V (lower is better).

Finding from 2.8.3: Interactive Causal Graph Refinement

The Interactive Causal Graph Refiner requires 1.61×–2.50× fewer judgments than Regression and 5.50×–16.83× fewer than LangModel. A call to the Interactive Causal Graph Refiner is interactive for most datasets, although slower than Regression. LangModel is at least 33% slower than LOGos.

2.8.4 Deep Dive into Interactive Causal Graph Refiner Strategy

We now dive deeper into the effectiveness of the strategy used by the Interactive Causal Graph Refiner, comparing it to the preliminary strategies presented in Section 2.6.1.

Degree of Freedom	Generation Method	Cardinality
Ground Truth Graphs	GenDAG(10, 0.5, −10, 10, 0.001, 2)	10
Datasets	GenData($\mathcal{G}$, 1,000, −10, 10)	3
Starting Graphs	GenDAG(10, 0.5, −10, 10, 0.001, 2)	10
Choices of T/O	Combinations($\lvert\mathcal{G}.nodes\rvert$, 2)	$\binom{10}{2} = 45$
Strategies	RandomEdit	3 runs
	SingleEdit	1 run
	HeuristicEdit	1 run
	AdjSetEdit	1 run
Total	$\mathbf{10 \times 3 \times 10 \times 45 \times (1 + 1 + 1 + 3) = 81{,}000}$	

Table 2.4: Breakdown of our experiment instances.

2.8.4.1 Additional Setup Details

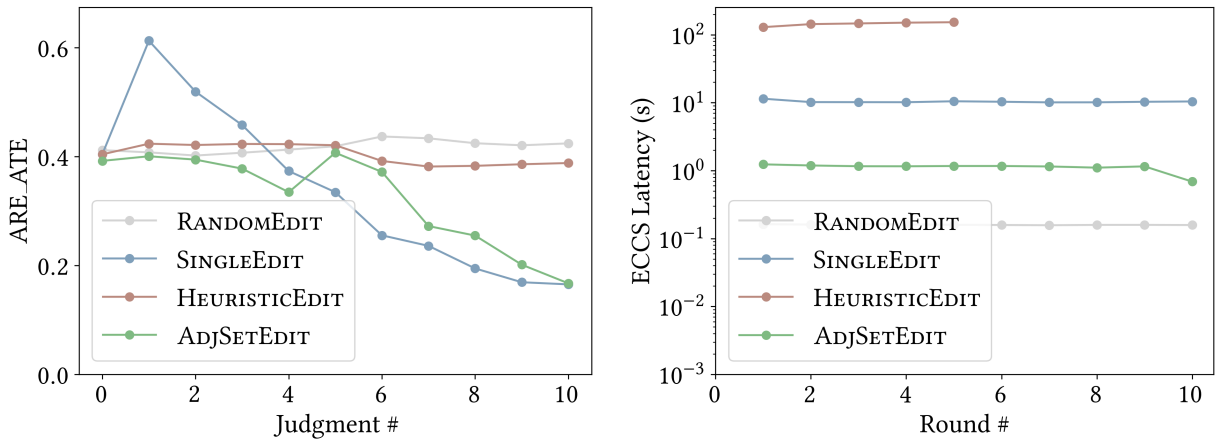
Datasets. For this experiment, we used synthetic causal graphs generated in two steps. First, we defined an algorithm GenDAG(N, p, w_{min}, w_{max}, n_{min}, n_{max}), which creates a graph $\mathcal{G}$ by instantiating N nodes in topological order and adding each edge from a node to each successor of it with probability p. Each edge has an associated *weight* drawn uniformly from $[w_{min}, w_{max}]$ and an associated *noise standard deviation*, drawn uniformly from $[n_{min}, n_{max}]$.

Then, GenData($\mathcal{G}$, L, v_{min}, v_{max}) creates a dataset of L points. For each data point, each variable associated with a source node in $\mathcal{G}$ is drawn uniformly from $[v_{min}, v_{max}]$. The rest are assigned values based on their incoming edges. In particular, for each incoming edge e, we multiply the value of the source of e by the weight of e and add Gaussian noise with zero mean and the noise standard deviation of e. The value of a non-source variable is the sum of all the values computed this way for each of its incoming edges.

User Model. We simulated a user who began with a causal graph $\mathcal{G}_{init}$ obtained by calling GenDAG and empty sets of fixed and banned edges, $\mathbf{F}_0 = \emptyset$ and $\mathbf{B}_0 = \emptyset$. The user also specified a dataset D, a treatment variable T and an outcome variable O. Like a human expert, such a user has knowledge of the ground truth graph $\mathcal{G}_{real}$ used to generate D, but interacts with LOGos to recover the relevant part of $\mathcal{G}_{real}$, rather than specifying it fully. In each interaction with the Interactive Causal Graph Refiner, the user sequentially produced judgments in the order solicited by Interactive Causal Graph Refiner (using their knowledge of the ground truth graph), marking any added/rejected edges as fixed/banned, respectively.

Baseline. As a baseline, we used RandomEdit, which solicits judgments based on random single-edge edits. The strategy samples elements of OneStepEdits($\mathcal{G}$, $\mathbf{F}$, $\mathbf{B}$) (Algorithm 2.2) uniformly at random without replacement. A user invoking this strategy is guaranteed to arrive at the correct causal graph, and therefore also at the correct ATE, in at most $\binom{|\mathbf{V}|}{2}$ calls, corresponding to the unordered pairs of distinct elements of $\mathbf{V}$.

Experiment instances. We created a total of 81,000 experiment instances, as indicated in Table 2.4. We created 10 "ground truth" causal graphs with 10 nodes and 3 associated 1,000-point datasets for each. We then created 10 more 10-node graphs from which the user started their interaction with LOGos. For each combination of a ground truth graph, a



(a) ARE_ATE (b) Latency

Figure 2.9: The evolution of our metrics of interest for each strategy over 10 user judgments.

corresponding dataset and a starting graph, we set the user's choice of treatment variable T and outcome variable O to all possible options. Because of the topological order-based approach of GenDAG, we know that a directed path from V_i to V_j can only exist if $i < j$, so we get 45 valid pairs. Some experiment instances were deemed invalid because there was no directed path from the selected treatment to the selected outcome in both the ground truth DAG and in the starting graph. The results presented below are aggregated over *valid* experiment instances. Finally, for each combination of (ground truth graph, starting graph, T, O), we simulated each of our four strategies over a horizon of 10 judgments. For HeuristicEdit, we set $b = 1{,}000$, $m = 2, k = 100$. For RandomEdit, we repeated each 10-judgment interaction 3 times and averaged the performance. Note that certain strategies may stop producing graph edit suggestions before the 10^{th} judgment is made (for example, if no suggestion they could produce complies with $\mathbf{F}/\mathbf{B}$). In such cases, we assume that the metrics of interest remain constant going forward.

2.8.4.2 RandomEdit

We first briefly discuss the performance of RandomEdit. On the ARE_ATE, Figure 2.9a reveals that the impact of this strategy is negligible and non-monotonic, as there is no connection between the suggested edits and the ATE of interest. After 10 judgments, this strategy achieves an average ARE_ATE of 0.42. However, as evident from Figure 2.9b, this approach is fast, with each call into LOGos taking an average of 0.16 seconds.

2.8.4.3 SingleEdit

Moving on to SingleEdit, per Figure 2.9a, the impact on the primary metric of ARE_ATE is dramatic, reaching 0.17 after 10 judgments – a reduction of 61.06% compared to RandomEdit. As shown in Figure 2.9b, the price for the ARE_ATE improvement is a latency increase of 2 orders of magnitude, with each call to LOGos requiring on average 10.39 seconds when using this strategy.

2.8.4.4 HeuristicEdit

Despite its increased sophistication, HeuristicEdit does not improve the metrics of interest. Figure 2.9a shows that it achieves an average ARE_ATE of 0.39 after 10 judgments without a clear decreasing trend, an improvement of only 8.51% over RandomEdit. With an average latency of 144.90 seconds per interaction round, further work would be required to make the ideas of HeuristicEdit useful. The main drawback of this method (beyond the long latency imposed by the heuristic search) appears to be its sequential identification of *different* suggested edge edits that attempt to achieve the inclusion or omission of the *same* impactful adjustment variable, even when the user keeps rejecting them.

2.8.4.5 AdjSetEdit

Finally, turning to AdjSetEdit, we observe from Figure 2.9a that performance on the primary metric of interest (ARE_ATE) is comparable to SingleEdit: after 10 judgments, it achieves a value of 0.17, a reduction of 60.60% compared to the baseline. However, this strategy offers a 9.28× improvement in latency compared to SingleEdit, with Figure 2.9b showing that each call to LOGos only takes 1.12 seconds on average. This latency improvement is substantial because it brings the system within an "interactive" operating domain, something that SingleEdit and HeuristicEdit do not offer.

Finding from 2.8.4: Deep Dive into Interactive Causal Graph Refiner Strategy

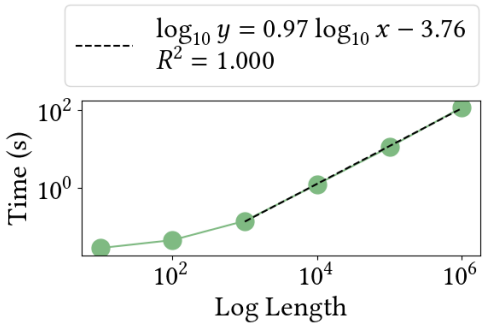
AdjSetEdit outperforms other strategies for the Interactive Causal Graph Refiner, tying SingleEdit for the joint-lowest ARE_ATE after 10 judgments with 9.28× lower latency.

2.8.5 Log Converter Scaling

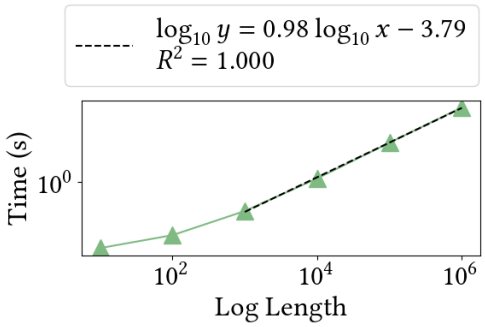
We close with an evaluation of the prototype Log Converter. In particular, we evaluated the scalability of Log Parsing (Section 2.4.1) and Aggregation (Section 2.4.4), which present the greatest algorithmic interest. Recall that Log Parsing relies heavily on Drain [91] (Section 2.7).

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. The second token was a string token of the form `s_`j, where $j \in \mathbb{Z}$ was uniformly random from $[1, S]$. The next V tokens were numerical variables, each of which took the value i. The last C tokens were constants equal to 0. To sweep log lengths, we fixed $S = 10$, $V = 1$, and $C = 10$, and swept L over powers of 10 from 10^1 to 10^6. To sweep the number of templates, we fixed $L = 10^4$, $V = 1$, and $C = 10$, and swept S over powers of 10 from 10^1 to 10^4. To sweep the fraction of variables, we fixed $L = 10^4$, $S = 10$, and $C = 100 - V$, and swept V over multiples of 10 from 10 to 100.

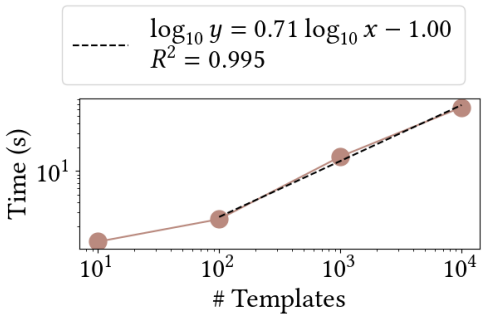
We parsed each log using a regular expression for `line_ID` and set the Drain similarity threshold to yield one log template per value of the string token in each experiment. As Figures 2.10a, 2.10c, and 2.10e show, the time taken by log parsing scaled linearly with each measure of the complexity of a log. We then aggregated each log into causal units determined by `line_ID`. Figures 2.10b, 2.10d, and 2.10f show the results, which again scale linearly.



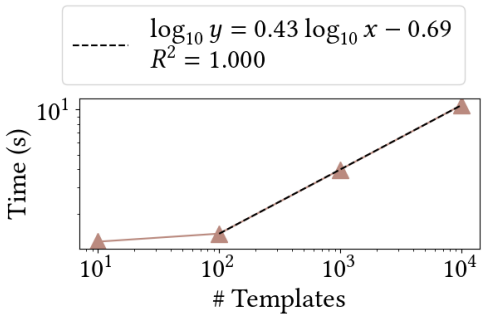
(a) Parsing vs. Log Length



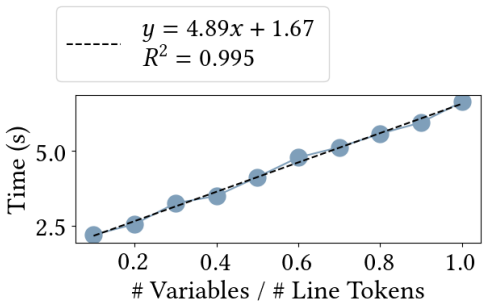
(b) Aggregation vs. Log Length



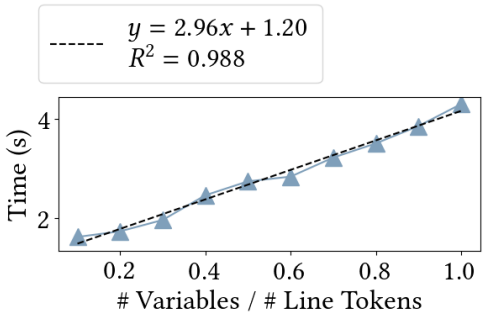
(c) Parsing vs. # Templates



(d) Aggregation vs. # Templates



(e) Parsing vs. % Variables



(f) Aggregation vs. % Variables

Figure 2.10: Log Converter scaling.

Finding from 2.8.5: Log Converter Scaling

Log Parsing and Aggregation in the Log Converter scale linearly with three measures of log complexity: length, number of templates, and fraction of tokens that are variables.

2.9 Related Work

2.9.1 Causality and Systems

Managing failures is crucial for large system operators. Past work has used formulations including failure classification [21], failing component detection [277], troubleshooting guide suggestion [109], job runtime impact analysis [276], and risk simulation [236]. Some existing work has also leveraged causality [4, 11, 73, 79, 84, 107, 159, 160]. Causal inference has also been used for other data management tasks more broadly, including query result and classification explanation [10, 188], hypothetical reasoning [72], exploratory data analysis [138], fault localization [4, 73, 145], data integration [197, 246], model fairness [272], and maximizing RCT data usefulness [11]. Any such system can benefit from the ability of LOGos to improve the quality of the causal graph.

However, logs have been under-explored as a data source, with most past work simply focusing on outlier detection [70, 80, 187]. One work does map outliers to their causes [33], but it requires a fully externally provided causal graph. The outlier detection framing also fails to flag longer-term problems (e.g. some users being consistently slower). Other works impose additional restrictions, like source code access [252], platform-specificity [267], or fine-grained hardware-supported logging [50]. On the commercial side, platforms like Splunk [209] and Datadog [57] provide log access interfaces, but no tailored support for causal inference. Datadog can identify some "root causes" for failures [56], but these verdicts appear driven just by temporal relationships, similar to Falcon [159] and Horus [160].

2.9.2 Causal Discovery

Causal discovery has seen much prior research [78, 136, 198, 207, 208, 223, 230, 255, 275]. Some existing algorithms are constraint-based, like PC [207] and FCI [208], while others are score-based, like GES [46]. Others, like LiNGAM [198], estimate the parameters of Functional Causal Models (FCMs). A subset of existing algorithms focuses specifically on *local* causal discovery around the treatment variable to reduce computational cost, and characterizes the range of possible causal effect sizes based on observational data [81, 139, 226, 245]. Each algorithm makes assumptions about the data [78], some of which are false for logs – e.g. functional dependencies yield singular correlation matrices. The number of variables involved is also a challenge for existing algorithms, as evidenced by the timeouts in Section 2.2.2. A text-mining approach based on the variable *names* may also be possible if such names are informative enough [86, 87, 92], most recently by leveraging large language models (LLMs) [18, 23, 24, 120, 131, 135, 154, 217, 222]. Despite often being impressive, LLMs still exhibit unpredictable failure modes that hinder their general adoption for causality [120]. 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. LOGos aims to efficiently use expert user feedback to tackle such failure modes.

2.9.3 Data-Driven Causal Graph Refinement

One approach to causal graph refinement has been targeted data collection through judicious interventions [48, 98, 103, 115, 175, 194]. This approach may be necessary when expert verification is unsuited for determining the direction of causality. In contrast, LOGos targets domains where such verification is possible, so that intervening is not necessary. Moreover, these works often build on a computed skeleton or Markov Equivalence Class [17, 88, 206] and focus on determining the direction of undirected edges in the skeleton, which limits the possibility of refining over the causal structure itself, unlike LOGos. Another line of work combats the possibility of spurious correlations being identified as causal relationships, by searching for necessary and sufficient causal graphs over only a relevant subset of variables [34]. While this approach can lead to a better-quality causal graph after the initial automatic causal discovery step, it is complementary to invoking the interactive components of LOGos on the resulting graph to also incorporate expert knowledge about the domain.

2.9.4 Adjustment Sets and ATE Calculation

Under the DAG model, we use Pearl's backdoor criterion [172], but there also exist variants of it [220]. Another line of research on identifying causal models and computing ATEs operates in the domain of Markov Equivalence Classes and completed partially directed acyclic graphs (CPDAGs) [17, 88, 254]. A notable recent work [81] uses the observation that each possible valid value of the ATE corresponds to a valid set of parents of the treatment to adjust for in the CPDAG. It uses local constraint-based causal discovery to efficiently identify the Markov Blanket of the treatment, all possible valid parent sets to adjust for, and therefore all possible ATE values. This algorithm still has exponential complexity in bad cases. However, unlike our work, this work does not aim to find ATE values that are close to ground truth. Previous work has also examined this problem in ancestral graphs [219].

2.10 Pitfalls and Lessons Learned

We will now discuss three pitfalls faced during the conception and development of LOGos and the corresponding lessons learned, which can inform the design of future systems. For each lesson, we will give appropriate context, discuss our initial approach, present our observations about its behavior, and conclude with our revised approach.

2.10.1 Treating Logs as Text

2.10.1.1 Context

In the initial exploration phase of this project, we were interested in the general domain of mining system text logs for performance-related insights. Our initial problem definition had a predictive angle: could we use the current log entries to predict upcoming ones, identify performance problems (as evidenced by the predicted appearance of error messages), and notify system administrators ahead of time? At this point in time, we had not yet definitively decided to use causal inference, but were instead exploring all available techniques.

2.10.1.2 Initial Approach

Based on the textual nature of the system logs at hand, we initially experimented with some of the large language models (LLMs) available at the time (Q3 of 2022) like GPT-3. We constructed a naive proof-of-concept approach to decide whether the capabilities of then-current general-purpose LLMs were appropriate for further use in our system. At a high level, this initial approach treated recent log messages as text, which the LLM was tasked with continuing. If concerning log messages appeared in the log continuation, one could then notify the system administrator. We did not intend for this approach to constitute a complete solution, but we wanted to evaluate its viability as a building block of a larger system.

2.10.1.3 Observations

The performance of this initial approach led us to some interesting insights. On the one hand, we observed that LLMs of that time were capable of generating a syntactically correct continuation for the provided log fragment, following the format of prior log messages. The models were even somewhat capable of substituting plausible values for some of the log variables, for example by advancing the timestamp with which log messages were prefixed.

On the other hand, this approach exhibited a set of problems, which can be summarized by the observation that, from an information-density standpoint, *logs are not really text*. The success of the next-token prediction approach, around which LLMs are designed, is predicated on the assumption that a piece of text can be accurately inferred from the text appearing before/around it, without extra-textual influences. However, this assumption is not true of log messages, where each "piece of text" (log message) is generated by the evolution of an underlying system, rather than as an autoregressive continuation of previous messages.

One way to understand this difference is by describing most text generation (e.g. prose) as "pull"-based: text is written to convey a particular argument, with earlier sentences setting up the stage for later ones and "pulling" them into existence. Logs, on the other hand, are generated based on a "push" model, whenever a particular event occurs in the underlying system, regardless of the presence of previous log messages.

Many of the log messages therefore reflect "normal operation", adding little additional information and drowning out the key log messages that may be describing a failure. Moreover, when a failure *does* occur, it may have been caused by any number of real-world events that are not reflected in earlier log messages. The predictive power of previous log messages is therefore often limited in terms of determining the next ones.

2.10.1.4 Revised Approach

Based on the observations above, we decided to move away both from LLMs as the insight-generating component of our system and from text as the dominant representation of information within our pipeline. As part of this shift, we developed the Log Converter to derive a structured, tabular representation of log contents that could be analyzed with a wider array of tools. Among these tools, we later arrived at causal inference as the most principled approach.

Lesson 2.1

Despite being textual, log messages are generated by extra-textual processes rather than following from previous messages, making next-token-prediction ineffective.

2.10.2 Using Time-Based Causal Units

2.10.2.1 Context

Based on Lesson 2.1, we next moved to conceptualize logs as partial, imperfect observations of a latent system state. Analyzing the logs could give insights into this latent state and its evolution over time, and causal inference quickly emerged as a promising analysis angle. However, we noticed that each log message only reports certain aspects of the latent state (i.e. certain variables or a certain event); therefore, in order to "fill in the gaps", it would be necessary to look at *sets of log messages* at a time, which hopefully collectively painted a more complete picture of the underlying system. The question of how to form such sets of messages was essentially equivalent to defining our causal units, a necessary prerequisite for any downstream ATE calculation.

2.10.2.2 Initial Approach

Viewing log messages as imperfect observations of a latent hidden state, our initial approach was to define each causal unit as consisting of log messages with *nearby timestamps*. In practice, the system may be in a unique latent state at the generation time of each message; however, we posited that log messages close in time may reflect similar enough latent states, so that we could analyze them jointly.

We therefore initially designed the Log Converter around time-based causal units, with the user only affecting the time resolution of the grouping (e.g. every 1 second or every 1 minute). In a follow-up formulation during the same period, we also considered separating the log messages into distinct time series based on the frequency of certain log message templates, allowing for a hierarchy of time-based causal units.

2.10.2.3 Observations

We quickly discovered that time-based causal units were inappropriate, because of the Stable Unit Treatment Value Assumption (SUTVA). SUTVA indicates that the "treatment" assigned to one causal unit cannot affect the "outcome" observed by a different unit, in order to correctly calculate average treatment effects.

When using causal inference in other scientific domains like medicine or economics, this assumption is often trivially satisfied, because the treatments are assigned to different physical entities or groups, like patients. However, if each of our causal units is, for example, a 5-minute window of system operation, then the assumption would be violated; an event that occurred some time ago and happened to fall in the previous window may nevertheless affect system state and influence what happens in the current window (e.g. a memory leak). Put differently, an apparent problematic value of an outcome variable at a particular point in time may be attributable to the value of a treatment variable far earlier in time, in a different causal unit.

2.10.2.4 Revised Approach

Based on the observations above, we reconsidered our approach to causal unit selection, explicitly moving away from time-based causal units. While analyzing the PROPRIETARY dataset (see Section 2.8.1.4), we found that real-world logs include a wide array of identifiers and tags that could be used to identify subparts of the system or track the progress of particular entities throughout the system. While such causal units may still interact in certain ways (e.g. different users sharing a cluster, or different clusters sharing the same network infrastructure), inter-unit causal relationships are far less direct than those induced by lumping together heterogeneous log messages that just happened to have arrived close in time. We therefore made causal unit selection a human-in-the-loop step in our pipeline, prioritizing causal units with real-life significance like machines or users.

Lesson 2.2

Temporal proximity is an inappropriate basis for designing causal units for log messages, because the underlying system state introduces "leaks" among causal units.

2.10.3 Reasoning About Confounders Directly

2.10.3.1 Context

Delegating causal unit selection to expert users moved our approach towards human-in-the-loop territory. This inspired us to explore other ways to tap the domain knowledge of such users, for example by having them assess possible causal relationships among log variables. Such an approach could help refine or supplement the outputs of classical causal discovery algorithms, which were facing challenges when operating on log data (see Section 2.2.2).

2.10.3.2 Initial Approach

Our initial conception of a human-in-the-loop causal exploration focused on a given causal question as a first-class citizen, with the user interacting with our system to calculate the correct ATE. The interface exposed by our system centered on finding the most potent confounders for the question at hand and exposing them to the user, for their approval or rejection. In particular, the causal graph was not a focal point of the interaction, but rather a secondary supporting artifact that was produced incidentally; the user directly made judgments about the existence of confounding relationships.

2.10.3.3 Observations

We observed that it was difficult and unintuitive for system experts to directly opine on whether something was or was not a confounder, since they may have been unfamiliar with the technical definition. The simple initial interface also gave no way for users to specify causal relationships *among* such discovered confounders, which could directly influence whether or not they need to all be included in the adjustment set.

2.10.3.4 Revised Approach

Based on the above observations, we redesigned our interface so that the *causal graph* became the primary artifact, into which user feedback is incorporated and maintained over time. We found that it could be easier for users to reason about the existence of causal edges, rather than the confounder status of some variable. However, to quantify and prioritize graph edits, our system still had to operate in "adjustment set space". This required the novel algorithms of Section 2.6.2 in order to map the system-derived adjustment variables back to human-understandable causal graph edits, which the user could accept or reject.

Lesson 2.3

Reasoning about the causal graph is more maintainable and intuitive for users, even if the system performs part of the processing by reasoning about adjustment sets.

2.11 Conclusion and Limitations

In this work, we presented a novel human-in-the-loop framework for applying causal inference to log data. This endeavor is challenging because log data suffer from three problems: Functional Dependencies, Large Number of Variables, and Biased Data Collection. Our framework combines novel algorithmic approaches with judicious use of human input to mitigate these problems. Our prototype, LOGos, accurately and efficiently calculates Average Treatment Effects among log variables, in both real-world and synthetic logs, while scaling linearly with the complexity of a log. This work can spur further exploration of applying causality in systems.

Although powerful, our approach has two limitations. First, we currently rely on both the *correctness* and the *coverage* of the input log data. Still, our approach is flexible enough to integrate more data sources, like code or documentation, using preprocessing modules analogous to the Log Converter. Second, since we include a human in the loop, some user domain knowledge is necessary to derive a high-quality causal model from our framework's suggestions. We believe the required level of expertise is appropriate for our intended audience (engineers administering large systems).

Chapter 3

AutoSLO: Practical Latency SLOs on Cloud Data Warehouses

3.1 Introduction

The Problem. Modern cloud data warehouses like Amazon Redshift Serverless [13] and Snowflake [203] decouple compute from storage, letting multiple compute clusters access the same underlying data. For example, multiple Redshift Serverless workgroups can access the same data through Datashares [15], while Snowflake supports multi-cluster warehouses [201].

Customers have embraced this feature [221] because it enables coarse-grained performance isolation by allowing the separation of workloads with different latency service-level objectives (SLOs). For example, interactive dashboards may need to respond within a few seconds, nightly ETL workloads may need to finish before business hours, and data-science workloads may tolerate some additional latency as long as costs are kept bounded.

However, even if workloads are each assigned to a separate cluster, current systems do not allow SLOs to be specified directly. This means that administrators have to experiment with cluster settings (e.g. number of nodes or price-performance hint [157]) until the latency SLO is met. Even after such tuning, there is no control over what happens as queries *within* each workload may interfere with one another. As a recent work by the Amazon Redshift team notes, *"large ad-hoc queries can have a severe negative impact on the overall performance of the cluster, since they can occupy compute resources and cause cache thrashing"* [157].

In this work, we take a different approach: we treat latency SLOs as first-class inputs and use the mechanisms exposed by cloud data warehouses to meet them cost-efficiently. The goal is not cross-workload performance isolation for its own sake, but rather meeting the SLO *of each individual query*, while minimizing infrastructure cost. This requires deciding which clusters to use, when to scale them, and where to route each incoming query.

What Success Looks Like. If each cluster could be spun up instantaneously and immediately offer its full performance, the task at hand would be simple: the system could wait until demand appears, create exactly the resources needed, and route queries accordingly. However, it takes time to provision new resources, and additional time for caches and other internal state to warm up. As a result, to cost-efficiently meet latency SLOs, we must manage cluster configuration across multiple timescales, as summarized in Table 3.1:

Table 3.1: No prior work exhibits all key desired behaviors.

Key Desired Behavior	**Plan** resources based on history	**Adjust** resources based on demand	**React** to concurrent live load
Timescale	≈ 24 h	≈ 5 min	≈ 1 s
ResTune [261]	✓ *Replay-based knob tuning on replica*	✗	✗
WiseDB [140]	✓ *Tree model for fixed per-template routing*	✗	✗
Das et al. [55]	✗	✓ *Container sizing w/ token bucket*	✗
SLAOrchestrator [165, 166, 167]	✗	✓ *Utilization- or simulation-based pool resizing*	✗
BRAD [118, 249]	✗	✓ *Blueprint beam search*	✗
Auto-WLM [189]	✗	✓ *Queueing-based horizontal scaling*	✗
RAIS [157]	✓ *Multi-forecast parameter tuning*	✓ *Queueing-based horizontal scaling*	✗
AutoSLO	✓ *Multi-forecast spinup planning & tuning*	✓ *Simulation-based best-cluster spinup*	✓ *Concurrency-aware routing*
Component	Policy Tuner	Autoscaler	Query Router
Details in...	Section 3.7	Section 3.6	Section 3.5

- **Plan (≈ 24 h):** Enterprise workloads often exhibit recurring patterns [221]. By extracting and anticipating these patterns, the system can make long-term cluster-management and configuration decisions proactively, such as when to spin up or tear down clusters and how to tune the policies used during execution. Because these decisions are made offline and amortized over longer periods, one can afford to evaluate a richer set of alternatives.
- **Adjust (≈ 5 min):** Forecasts are imperfect, and workloads can evolve. The system must be able to respond by spinning up or tearing down clusters to compensate for unexpected bursts, lulls, or shifts in workload mix. Unlike planning, adjustment operates under tighter time constraints and therefore makes more localized decisions within the overall policy already chosen by the planner.
- **React (≈ 1 s):** Finally, each arriving query must be assigned to one of the currently active clusters. Since the goal is no longer workload performance isolation, but rather meeting query-level latency SLOs cost-efficiently, this decision must account for the query's own latency SLO, its impact on already-running queries, and the infrastructure cost implied by the routing decision.

These behaviors are complementary. Planning reduces dependence on slow reactive scaling by preparing resources before expected demand arrives. Adjustment handles forecast error and unexpected workload changes. Reaction handles the per-query effects of concurrency and contention, which remain present even under a well-chosen cluster configuration. Together, they allow a system to leverage multiple clusters to meet query-level latency SLOs without requiring rigid workload-to-cluster assignments.

How Prior Work Falls Short. As shown in Table 3.1, several automatic workload management systems have been proposed, but none exhibits all three **key desired behaviors**.

Some systems can only **plan** for a known or forecasted workload, using replays [261] or modeling [140]. However, planning alone is insufficient: forecasts are imperfect, workload mixes can shift, and live query-level contention can affect SLO adherence and must be managed. Other systems can only **adjust** resources online in response to demand, using various forms of short-term performance modeling/assumptions under different available resources [55, 118, 165, 166, 167, 189, 249]. However, spinning up new clusters can require non-negligible time, so purely reactive approaches will transiently leave the system under-provisioned, risking SLO violations.

Finally, existing systems route queries using static workload classes, query templates, or estimated resource requirements [140, 157, 167, 189, 249]. They do not **react** to live concurrent query load by explicitly considering how placing a new query on a particular cluster affects SLO adherence (for both the new query and the already-running ones). This is how RAIS [157] still falls short of addressing the full problem: since it only exposes a *qualitative* price-performance slider, rather than a way to specify latency SLOs, it cannot provide SLO-aware query placement.

Our Approach. AutoSLO addresses the gaps discussed above by jointly providing all three key desired behaviors: planning from historical workload patterns, adjusting resources when reality deviates from the plan, and reacting to live concurrent load when routing each query. It does so through three components, each corresponding to one key behavior: the Policy Tuner, the Autoscaler, and the Query Router.

The Policy Tuner can **plan** using the historical workload. It generates and simulates forecasted workloads to determine proactive cluster-management actions and configure reactive scaling parameters. By ensuring that the right resources are available at the right time (shortly before demand materializes), it reduces dependence on reactive scaling and mitigates the impact of cluster spinup delays on live query traffic.

The Autoscaler can **adjust** the active cluster set when the observed workload behavior deviates from the forecast for which the system was tuned. Its important contributions include cluster spinup/teardown triggers, which identify opportune moments for resource adjustments, and a cluster spinup size selector, which uses short-term simulations to estimate the SLO and cost implications of different scaling actions.

Finally, the Query Router can **react** to live load. When a query q arrives, it selects which active cluster should execute it by managing a multi-way tradeoff among (i) the risk that q will violate its SLO; (ii) the risk q poses to the SLO adherence of already-running queries; and (iii) the resource cost implied by each routing decision.

Contributions. In summary, we make the following contributions:

- We formulate the problem of SLO-aware multi-cluster workload management for cloud data warehouses, where the goal is to bound the latency SLO violation rate at minimal infrastructure cost.
- We present AutoSLO, a multi-timescale framework that provides the three key desired behaviors needed for this problem: planning from historical workload patterns, adjusting resources when workloads deviate from the forecast, and reacting to live concurrent load when routing individual queries.
- We develop the components of AutoSLO: a forecast-driven Policy Tuner for proactive cluster management and tuning, a simulation-based Autoscaler for cost- and SLO-aware scaling, and a concurrency-aware Query Router for per-query placement.
- We evaluate AutoSLO on realistic Redbench-generated workloads and show that it can successfully meet latency SLOs of varying strictness, reducing cost by a mean of 26.4% compared to the per-scenario next-best baseline.
- We also evaluate per-component performance and find that the Query Router and Autoscaler reduce the SLO violation rate by a mean of 47.8% and 93.7%, respectively, compared to corresponding alternatives on TPC-DS-derived workloads, while the Policy Tuner can reduce the SLO violation rate by a mean of 44.6% using a single day of workload history. Each component is also shown to be efficient enough for its operating timescale.

3.2 Problem Formulation

In this section, we will build towards our problem definition more formally. We will introduce some useful notions related to cloud data warehouses (Section 3.2.1) before stating the problem we target (Section 3.2.2).

3.2.1 Background on Cloud Data Warehouses

We begin by defining the workload and system setting we target more precisely.

Online, Diverse Query Arrivals. Each query q is issued online at time $q.t_{\text{arrival}}$. At each point in time, no information about future query arrivals is known exactly. Each query q has varying resource demands (e.g. CPU, memory, I/O), which may interact with those of concurrently executing queries, impacting their latency.

Latency SLOs. Each query q has a latency service-level objective $SLO(q)$, treated as an input provided by the application or administrator. For example, the SLO may be inferred from the query's source application, user-defined priority, or other submission-time metadata. A query q *meets* its SLO whenever its *client-side* latency $L(q) \leq SLO(q)$. Otherwise, q *violates* its SLO.

Clusters. Multiple clusters can query the same data, with the vendor managing concurrency control and consistency. Each cluster c has a *size* $S(c)$ describing its computational capacity. The queries running on c are denoted Q_c. Instead of statically assigning each workload to a cluster, we allow routing queries independently:

Definition 3.1: Routing Policy

Let $\mathcal{C}_{q.t_a}$ be the active cluster set when query q arrives, and $\mathcal{H}_{q.t_a}$ be the historical workload observed until then, including prior query arrivals, routing decisions, completions, and SLOs. A routing policy $\mathcal{R}$ determines which active cluster to execute q on:

$$\mathcal{R}(q, \mathcal{C}_{q.t_a}, \mathcal{H}_{q.t_a}) \mapsto c \in \mathcal{C}_{q.t_a}$$

Adjustable Cluster Pool. New clusters can be requested from the vendor, and existing clusters can be released. Spinning up a new cluster c takes time δ; tearing down an existing cluster takes time ζ. We present δ and ζ as constants, but our techniques also support sampling them. This leads to another decision point:

Definition 3.2: Scaling Policy

At time t, given the active cluster set $\mathcal{C}_t$ and the historical workload $\mathcal{H}_t$, a scaling policy $\mathcal{S}$ outputs a new active cluster set $\mathcal{C}_{new}$:

$$\mathcal{S}(t, \mathcal{C}_t, \mathcal{H}_t) \mapsto \mathcal{C}_{new}$$

Usage-Based Billing. Let $\mathcal{P} = (\mathcal{R}, \mathcal{S})$ denote a pair of policies. The cost $k^{\mathcal{P}}(c, \mathcal{Q})$ of a cluster c over workload $\mathcal{Q}$ with these policies is the product of (i) a constant *price* p; (ii) c's size $S(c)$; and (iii) c's *active time* while $\mathcal{Q}$ is executed.

The active time of a cluster c is conceptually measured by a timer as follows: the timer is initially paused; if paused, the timer starts running whenever a query arrives at c; if running, the timer pauses whenever two conditions are both met. The two conditions are: (i) there are no running queries on c; and (ii) at least m seconds (e.g. $m = 60$ [14, 202]) have elapsed since the timer most recently started running. Billing interacts with routing: using an idle cluster restarts its timer, while consolidating reduces cost but can cause interference.

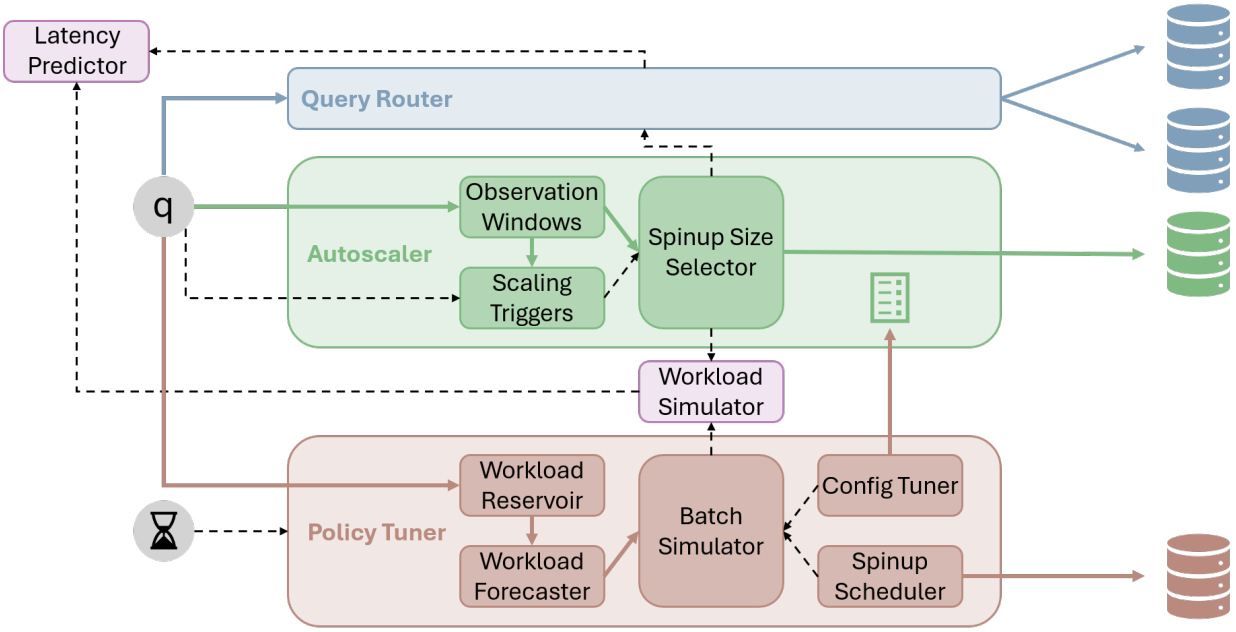


Figure 3.1: The architecture of AutoSLO. Solid lines indicate input/output flow, while dashed lines indicate control flow.

3.2.2 Problem Definition

For a workload $\mathcal{Q}$ under policies $\mathcal{P}$, let the *SLO violation rate* be $v^{\mathcal{P}}(\mathcal{Q}) = \frac{1}{|\mathcal{Q}|}\sum_{q\in\mathcal{Q}} \mathbf{1}[L^{\mathcal{P}}(q) > SLO(q)]$ and the *total cost* be $k^{\mathcal{P}}(\mathcal{Q}) = \sum_c k^{\mathcal{P}}(c, \mathcal{Q})$. We also allow specifying an SLO violation rate *target* τ. Not treating τ as a hard constraint lets us compare policy pairs even when their SLO violation rate is above τ. In particular, given a set $\mathcal{M}$ containing the SLO violation rate and cost of each policy pair, $\mathcal{M} = \{(v^{\mathcal{P}_1}, k^{\mathcal{P}_1}), \ldots, (v^{\mathcal{P}_n}, k^{\mathcal{P}_n})\}$, we use:

$$\textsc{BestWithTarget}(\mathcal{M}, \tau) = \arg\min^{\text{lex}}_{(v,k)\in\mathcal{M}} (\max(0,\ v-\tau),\ k,\ v)$$

We seek the best policy under the lexicographic target-aware objective above. Concretely:

Problem 3.3: SLO-aware Multi-Cluster Workload Management

For an SLO violation rate target $\tau \geq 0$, select a policy pair $\mathcal{P}^\star \in \Pi$ that prioritizes meeting τ and only then reducing cost:

$$\mathcal{P}^\star \in \left\{ \mathcal{P} \in \Pi \;\middle|\; (v^{\mathcal{P}}, k^{\mathcal{P}}) = \textsc{BestWithTarget}\left(\{(v^{\mathcal{P}'}, k^{\mathcal{P}'}) \mid \mathcal{P}' \in \Pi\}, \tau\right) \right\}.$$

3.3 Overview of AutoSLO

To address the problem above, we design AutoSLO to demonstrate the three **key desired behaviors** from Section 3.1. For our technical exposition in the following sections, unlike our top-down introduction, we follow a query's path through the system bottom-up.

As shown in Figure 3.1, an incoming query q first encounters the Query Router, which can **react** to the current workload and decide which active cluster to forward q to. This decision is powered by the Latency Predictor, which estimates how different placements would affect both q and the already-running queries. Sections 3.4–3.5 describe these components.

Each query is also added to the Autoscaler's sliding Observation Windows and leads to a recalculation of the Scaling Triggers, which decide whether to **adjust** the active cluster set. If a spinup is triggered, the Spinup Size Selector simulates a short-term workload forecast under different hypothetical cluster additions to balance SLO adherence and cost. It uses the Workload Simulator, a simple event-driven simulator that can replay query arrivals and uses the Latency Predictor to determine query completions. Section 3.6 expands on this process.

Finally, each query is added to the Workload Reservoir within the Policy Tuner, which is triggered periodically (e.g. every 24 hours). The Workload Forecaster uses the reservoir to generate forecasted workloads, which are efficiently simulated using the Batch Simulator. Based on these simulations, the Spinup Scheduler optimizes cluster spinup timing for reliably recurring load, while the Configuration Tuner then tunes Autoscaler parameters governing on-demand autoscaling. Section 3.7 covers this functionality.

3.4 Latency Predictor

We start with Iconq [235], a recent LSTM-based concurrent query latency estimation model, but modify it to address the needs of AutoSLO. In particular, we need support for clusters of different sizes (in the Query Router), while we also need to use the model to determine query completion times during simulations (in the Workload Simulator). Before we discuss how we address these requirements in our Latency Predictor, called Iconq+, we present Iconq.

3.4.1 Background: Iconq

Iconq was developed for single-cluster query scheduling, where queries can be queued and/or reordered to reduce mean latency. At a high level, Iconq predicts the latency of a query q by ingesting a sequence of *interaction feature vectors*, derived from q and its neighbors, with an LSTM. We define q_n as a neighbor of q if their executions overlap in time. Each interaction feature vector concerns q and a particular neighbor q_n and includes: query-plan-derived features for q and q_n; relative timing information about their arrivals; and concurrency-unaware latency estimates for each query, $\hat{\ell}_q$ and $\hat{\ell}_{q_n}$, derived from a fast sub-model (Stage [234]). Iconq ingests neighbors arriving before q and after q separately, enabling latency prediction *updates* for already-running queries when new ones arrive.

3.4.2 Query Featurization

Iconq assumes all queries run on the same cluster. In AutoSLO, however, we must predict latencies across clusters of different sizes. We therefore extend the *interaction feature vector* with features related to cluster size, including: (i) the raw cluster size; (ii) $\log_2(\text{size})$ to capture non-linear scaling; (iii) $1/\text{size}$ to capture inverse-capacity effects; (iv) $\hat{\ell}_q \cdot \text{size}$ and $\hat{\ell}_{q_n} \cdot \text{size}$ as proxies for the compute work implied by each query's isolated latency; and (v)

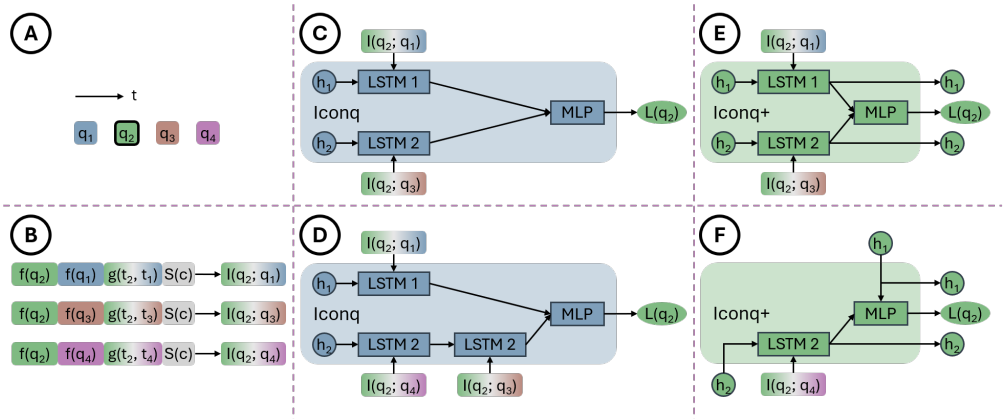


Figure 3.2: When predicting the latency of q_2, Iconq+ ingests the interaction feature vectors of later neighbors (q_3 and q_4) in chronological order, enabling incremental inference.

$(\hat{\ell}_q + \hat{\ell}_{q_n})$/size as a capacity-normalized pairwise contention signal. In Figure 3.2**B**, we see the interaction feature vectors for the queries in Figure 3.2**A**, when predicting the latency of q_2. Each includes query plan features and Stage latency predictions per query $f(q)$, relative timing features $g(t; t_{neighbor})$ and cluster size features $S(c)$ as above.

3.4.3 Censored Observations

To predict the latency of q, Iconq uses features derived from q and its neighbors. When scheduling online, precise neighbor sets are known at decision time. In AutoSLO, however, we also use the Latency Predictor in the Workload Simulator. There, whether two queries overlap can depend on their simulated (predicted) latencies, which require overlap knowledge. Hard inclusion of neighbors therefore creates a feedback loop, where early prediction errors alter downstream neighbor sets and lead to compounding errors.

To mitigate this, we also train Iconq+ on censored observations using sampled partial neighbor sets, where labels are treated as *lower bounds* (i.e. loss is zero if $pred \geq label$). This lets us extract more from the training data. For example, if q ran from $t = 0$ to $t = 10$ and q' arrived at $t = 7$, then we know that $L(q) = 10$ in the presence of q'; however, we also know that $L(q) \geq 7$ without q', since q was still running when q' arrived.

Censoring also lets us train on queries aborted due to timeouts or errors, with time-to-failure as their censored label. This is especially important for small clusters, where timeouts can be more common; discarding timed-out queries would over-represent successful executions and bias the model toward latency underprediction.

3.4.4 Incremental Predictions

Iconq uses a bidirectional LSTM to ingest a sequence of interaction feature vectors. Per Figure 3.2**C**/**D**, when predicting the latency of q_2, neighbors arriving before q_2 (i.e. q_1) are ingested in chronological order, while neighbors arriving after q_2 (i.e. q_3 and q_4) are ingested in reverse chronological order. This makes both sequences end near q_2's arrival, emphasizing

Algorithm 3.1 Overall workflow of the Query Router.

Inputs: Incoming query q, active cluster set $\mathcal{C} = \{c\}$ with latency predictions $\hat{L}_c^{\text{before}}$ and latency predictor states M_c^{before} for running queries.

Outputs: Selected cluster $c^\star$, updated latency predictions $\hat{L}_{c^\star}$ and updated latency predictor states $M_{c^\star}$

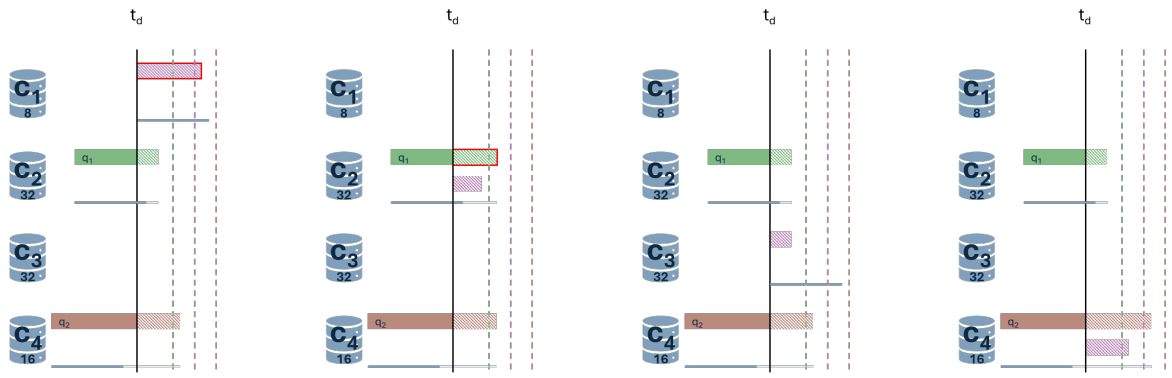
1: **function** ROUTE(q, $\mathcal{C}$)
2: **for** $c \in \mathcal{C}$ **do**
3: $v_c^{\text{before}} \leftarrow \sum_{q_i \in Q_c} \mathbf{1}[\hat{L}_c^{\text{before}}(q_i) > \text{SLO}(q_i)]$
4: $k_c^{\text{before}} \leftarrow$ CLUSTERCOSTUNTILDRAINED($c, Q_c, \hat{L}_c^{\text{before}}$)
5: $\mathcal{N}_c \leftarrow$ PERQUERYNEIGHBORSIFWEROUTE(c, q)
6: $\{L_c^{\text{after}}, M_c^{\text{after}}\}_{c \in \mathcal{C}} \leftarrow$ PREDICTWITHICONQ+($\{\mathcal{N}_c, M_c^{\text{before}}\}_{c \in \mathcal{C}}$)
7: **for** $c \in \mathcal{C}$ **do**
8: **for** $q_i \in Q_c$ **do**
9: $\hat{L}_c^{\text{after}}(q_i) \leftarrow \max(\hat{L}_c^{\text{after}}(q_i),\ \hat{L}_c^{\text{before}}(q_i), q.t_{\text{arrival}} - q_i.t_{\text{arrival}})$
10: **for** $c \in \mathcal{C}$ **do**
11: $Q_c^{\text{after}} \leftarrow Q_c \cup \{q\}$
12: $v_c^{\text{after}} \leftarrow \sum_{q_i \in Q_c^{\text{after}}} \mathbf{1}[\hat{L}_c^{\text{after}}(q_i) > \text{SLO}(q_i)]$
13: $k_c^{\text{after}} \leftarrow$ CLUSTERCOSTUNTILDRAINED($c, Q_c^{\text{after}}, \hat{L}_c^{\text{after}}$)
14: $\Delta v_c \leftarrow v_c^{\text{after}} - v_c^{\text{before}}$
15: $\Delta k_c \leftarrow k_c^{\text{after}} - k_c^{\text{before}}$
16: $c^\star \leftarrow \text{lexicographic-argmin}_{c \in \mathcal{C}} \left(\Delta v_c,\ \Delta k_c\right)$
17: **return** $c^\star$, $\hat{L}_{c^\star}^{\text{after}}$, $M_{c^\star}^{\text{after}}$

nearby interactions in the LSTM state. However, it also means that whenever a new neighbor arrives (e.g. q_4 in panel **D**), every neighbor after q_2 must be re-ingested.

Although the re-ingestion overhead per query can be small, it can add up during simulations, slowing down the Autoscaler and the Policy Tuner. This is especially true because simulations interleave predictions and CPU-heavy state bookkeeping, effectively mandating CPU inference. To avoid repeated re-ingestion, we modify Iconq+ to also process after-neighbors in chronological order, and add support for incremental inference from cached model state (Figure 3.2**E**/**F**).

3.5 Query Router

The Query Router routes each incoming query q to an active cluster, leveraging the Latency Predictor to balance three concerns: (1) the ability of q to meet its SLO, (2) the impact of q on the SLO adherence of currently running queries, and (3) the cost implications of the routing decision. The Query Router implements ROUTE (Algorithm 3.1). It first inspects each cluster c, gathering the SLO-violating queries (lines 2–3) and the total projected cost, from c's spinup until the running queries are predicted to drain (line 4). In the same pass, it builds each cluster's counterfactual query neighbor map, as if the incoming query q were placed there (line 5).



(a) q SLO miss (b) q_1 SLO miss (c) Expensive (d) Balanced

Figure 3.3: The Query Router balances SLOs and cost. The x-axis is time. Each query is a rectangle and same-color dashed lines show when it will miss its SLO. Hatching indicates predicted latencies; a red outline marks an SLO miss. Thin blue lines show billed time; dark blue indicates the minimum billed duration.

It then runs Iconq+ inference on these counterfactual neighbor maps in a single batch (line 6), yielding updated model predictions $\hat{L}_c^{\text{after}}$ and model states M_c^{after}. As explained in Section 3.4.4, Iconq+ updates the latency predictions and model states for already-running queries *incrementally* within this step. ROUTE then applies a monotonicity guard for stability (lines 7–9): the latency prediction for each running query is never updated downwards, reflecting the fact that later queries generally cannot make an earlier query faster, and cannot be smaller than its observed latency so far.

For each candidate cluster c, the algorithm then counts the SLO-violating queries and implied cost in a manner similar to the baseline state (lines 10–13), ultimately calculating the marginal impact of the routing decision (lines 14–15). It then identifies the cluster $c^\star$ that minimizes marginal SLO violations and cost (in this order) and returns it, together with the corresponding updated latency predictions and latency predictor states (lines 16–17).

Figure 3.3 shows an example, where query q (purple) is routed among 4 clusters, with 2 active queries q_1 (green) and $q2$ (brown). Routing q to c_1 would make it miss its SLO, while routing it to c_2 would make q_1 miss its own SLO. Routing it to c_3 would imply no SLO violations, but it would start a new billing window, leading to high cost. The best option is therefore c_4; there are no SLO violations, while the marginal cost impact is minimized.

By explicitly reasoning about query interference, SLO violation risk, and resource cost, the Query Router can make decisions that balance performance and efficiency in real time.

Algorithm 3.1 Computational Time Complexity. Before and after model inference, ROUTE does a constant amount of work per running query, implying $O(N)$, where $N = \sum_c |Q_c|$. For line 6, each LSTM step is $O(dh + Lh^2)$ (to ingest the input and update the hidden state per layer), assuming input dimension d, hidden size h, and L layers. We will treat this as constant, since we use a fixed model. We perform two steps per running query q_{running}: one to update q_{running}'s latency prediction if q is added to its neighbors (see Section 3.4.4), and another while ingesting q_{running} as a neighbor of q while predicting the latency of q. Line 6 is then also $O(N)$, meaning that ROUTE's overall time complexity is $O(N)$.

3.6 Autoscaler

The Query Router routes each query *among the active cluster set* $\mathcal{C}$; however, it may be that $\mathcal{C}$ is no longer appropriate for the current workload, which may instead be better served by a different set of clusters. The Autoscaler addresses such cases through reactive autoscaling, using three sub-components (see Figure 3.1): the Observation Windows, the Scaling Triggers, and the Spinup Size Selector.

3.6.1 Observation Windows

The Autoscaler runs in a background thread and maintains two trailing Observation Windows over the last w seconds. It appends each incoming query to the *arrival window* $\mathcal{W}^a$ while it appends each completed query (alongside its arrival time and latency) to the *completion window* $\mathcal{W}^c$, truncating each as needed to keep its length below w. Each query arrival also trips two Scaling Triggers, explained next.

3.6.2 Scaling Triggers

3.6.2.1 Teardown Trigger

The teardown trigger determines whether one of the active clusters should be torn down. For an active cluster c, it fires when three conditions are simultaneously met: (i) no queries are currently running on c; (ii) no query has been routed to c for the last T_{idle} seconds; and (iii) c was spun up at least $T_{\text{min\_lifetime}}$ seconds earlier. If all conditions are met, the Autoscaler initiates the teardown of c. T_{idle} and $T_{\text{min\_lifetime}}$ can be periodically optimized by the Policy Tuner, as we will examine in Section 3.7.5.

3.6.2.2 Spinup Trigger

The spinup trigger determines whether a new cluster should be spun up, based on whether recent SLO adherence is unsatisfactory. As described in Algorithm 3.2, it first checks whether the active cluster set has changed since the last invocation; if so, it saves it, records the current arrival time as t_{change} and clears the in-flight flag f (lines 2–5). The algorithm then assembles R (line 6), the set of running queries that arrived after t_{change}, and D (line 7), the set of queries from $\mathcal{W}^c$ that arrived after t_{change}, alongside their predicted and realized latencies, respectively. If the combined size of these two sets is at least θ and the in-flight flag is not set (line 8), the algorithm computes the SLO violation rate among $R \cup D$ (line 9) and compares it to τ_{trigger} (line 10). The trigger fires only if this threshold is exceeded, at which point the in-flight flag is also set (lines 11–13). The parameters τ_{trigger} and θ can be periodically optimized by the Policy Tuner, as we will examine in Section 3.7.5. If the spinup trigger fires, the Autoscaler invokes the Spinup Size Selector, explained next in Section 3.6.3.

Algorithm 3.2 Computational Time Complexity. The comparison on line 2 takes time $O(\min(|\mathcal{K}|, |\mathcal{C}|))$, while constructing R and D on lines 6–7 requires time $O(N + |\mathcal{W}_c|)$, where $N = \sum_c |Q_c|$. Line 9 (if reached) operates on a subset of these query sets, so the overall complexity is $O(\min(|\mathcal{K}|, |\mathcal{C}|) + N + |\mathcal{W}_c|)$.

Algorithm 3.2 The spinup trigger of the Autoscaler.

Inputs: Completion window $\mathcal{W}^c$, active cluster set $\mathcal{C} = \{c\}$ with predictions $\hat{L}_c$, query q_{new}
Configuration: Trigger SLO threshold τ_{trigger}, minimum observations θ
State: Known active cluster set $\mathcal{K}$, change bound t_{change} (shared with **Algorithm 3.3**), in-flight flag f (shared with **Algorithm 3.3**)
Outputs: Whether to invoke the Spinup Size Selector.

1: **function** MAYBESPINUP($\mathcal{W}^c$, $\mathcal{C}$, q_{new})
2: **if** $\mathcal{K} \neq \mathcal{C}$ **then**
3: $\mathcal{K} \leftarrow \mathcal{C}$
4: $t_{\text{change}} \leftarrow \max(t_{\text{change}}, q_{\text{new}}.t_{\text{arrival}})$
5: $f \leftarrow$ FALSE
6: $R \leftarrow \{(q, \hat{L}_c(q)) \mid (q \in Q_c, c \in \mathcal{C}) \wedge (q.t_{\text{arrival}} \geq t_{\text{change}})\}$
7: $D \leftarrow \{(q, L(q)) \mid (q \in \mathcal{W}^c) \wedge (q.t_{\text{arrival}} \geq t_{\text{change}})\}$
8: **if** $\neg f$ **and** $|R| + |D| \geq \theta$ **then**
9: $v_{\text{cur}} \leftarrow \frac{1}{|R|+|D|} \sum_{(q,\ell) \in \{R \cup D\}} \mathbf{1}[\ell > \text{SLO}(q)]$
10: **if** $v_{\text{cur}} > \tau_{\text{trigger}}$ **then**
11: $f \leftarrow$ TRUE
12: **return** TRUE
13: **return** FALSE

3.6.3 Spinup Size Selector

Once the spinup trigger fires, the Spinup Size Selector must determine the size of the cluster to spin up, if any. At a high level, for each eligible cluster size, it hypothesizes a cluster spinup of that size and compares the downstream SLO violation rate and cost. It does this by simulating a short-term workload forecast, where queries arrive according to copies of the arrival window $\mathcal{W}^a$, are routed as usual by the Query Router, and complete based on their predicted latencies. Crucially, it also evaluates a *do-nothing baseline* (no new cluster added), and only actually recommends a cluster spinup if the best candidate strictly improves upon not changing the active cluster set at all.

Remember that a new cluster will not be available instantly, but only after a spinup delay δ. It is vital to capture this *preparation period* in the simulation, because it may generate a workload backlog that the new cluster will have to help relieve, once available. However, the events of this preparation period are independent of the size of the requested cluster, so that it is sufficient to compute the simulation state at the end of the preparation period once.

More precisely, the Spinup Size Selector operates according to Algorithm 3.3. Lines 2–12 describe the preparation period, during which the new cluster has been requested but it is not yet available from the vendor: copies of the queries in the arrival window are issued (lines 4–5 and 11) and routed (lines 9–10), while queries that are predicted to have completed are removed from tracking (line 8). Once the new cluster is available (lines 2–3 and 6–7), replay stops and we snapshot the cluster set state (line 12). We extend the candidate set with a do-nothing baseline (line 13) and, for each possible action (i.e. spinning up a cluster of each of the eligible sizes, or doing nothing), we initialize the cluster set from the snapshot captured at the end of the preparation period (lines 14–18).

Algorithm 3.3 The Spinup Size Selector.

Inputs: Arrival window $\mathcal{W}^a$, active cluster set $\mathcal{C} = \{c\}$ with latency predictions $\hat{L}_c$, decision time t_d, target SLO violation rate τ_{target}

Configuration: Eligible cluster sizes $\mathcal{S}$, spinup delay δ, minimum completions μ, observation window width w

State: Change bound t_{change}, in-flight flag f (both shared with **Algorithm 3.2**)

Outputs: Cluster size s to spin up, or $\emptyset$ if none advisable.

function FindBestSpinupSize($\mathcal{W}^a$, $\mathcal{C}$, t_d, τ_{target})
  $t_{\text{available}} \leftarrow t_d + \delta$; $i \leftarrow 0$
  while True **do**
    $q \leftarrow \mathcal{W}^a[i \mod |\mathcal{W}^a|]$
    $q.t_{\text{arrival}} \leftarrow q.t_{\text{arrival}} + w \cdot \lfloor i/|\mathcal{W}^a| \rfloor$
    if $q.t_{\text{arrival}} \geq t_{\text{available}}$ **then**
      break
    CollectFinishedUntil($\mathcal{C}$, $q.t_{\text{arrival}}$)
    $(c^\star, \hat{L}_{c^\star}, M_{c^\star}) \leftarrow$ Route(q, $\mathcal{C}$)
    $c^\star$.AddQuery(q, $\hat{L}_{c^\star}$, $M_{c^\star}$)
    $i \leftarrow i + 1$
  $(\mathcal{C}_{\text{snap}}, i_{\text{snap}}) \leftarrow (\mathcal{C}, i)$
  $\mathcal{S}' \leftarrow \mathcal{S} \cup \{\emptyset\}$
  for $s \in \mathcal{S}'$ **do**
    $\mathcal{C}_s \leftarrow \text{copy}(\mathcal{C}_{\text{snap}})$
    if $s \neq \emptyset$ **then**
      $\mathcal{C}_s \leftarrow \mathcal{C}_s \cup \{$NewCluster(size $= s$)$\}$
    $i \leftarrow i_{\text{snap}}$
    $D \leftarrow \emptyset$
    while $|D| < \mu$ **do**
      $q \leftarrow \mathcal{W}^a[i \mod |\mathcal{W}^a|]$
      $q.t_{\text{arrival}} \leftarrow q.t_{\text{arrival}} + w \cdot \lfloor i/|\mathcal{W}^a| \rfloor$
      $F \leftarrow$ CollectFinishedUntil($\mathcal{C}_s$, $q.t_{\text{arrival}}$)
      $D \leftarrow D \cup \{(q', L(q')) \mid (q' \in F) \wedge (q'.t_{\text{arrival}} \geq t_{\text{available}})\}$
      $(c^\star, \hat{L}_{c^\star}, M_{c^\star}) \leftarrow$ Route($q, \mathcal{C}_s$)
      $c^\star$.AddQuery(q, $\hat{L}_{c^\star}$, $M_{c^\star}$)
      $i \leftarrow i + 1$
    $D \leftarrow D \cup_{c \in \mathcal{C}_s} \{(q', \hat{L}_c(q')) \mid q' \in Q_c\}$
    $v^s \leftarrow \frac{1}{|D|} \sum_{(q,\ell) \in D} \mathbf{1}[\ell > \text{SLO}(q)]$
    $k^s \leftarrow \sum_{c \in \mathcal{C}_s}$ ClusterCostUntilDrained(c, Q_c, $\hat{L}_c$)
  $(v^{\text{best}}, k^{\text{best}}) \leftarrow$ BestWithTarget($\{(v^s, k^s)\}_{s \in \mathcal{S}'}$, τ_{target})
  if $(v^{\text{best}}, k^{\text{best}}) = (v^\emptyset, k^\emptyset)$ **then**
    $t_{\text{change}} \leftarrow t_d + \delta$
    $f \leftarrow$ False
    return $\emptyset$
  return $\max\{s \in \mathcal{S} \mid (v^{\text{best}}, k^{\text{best}}) = (v^s, k^s)\}$

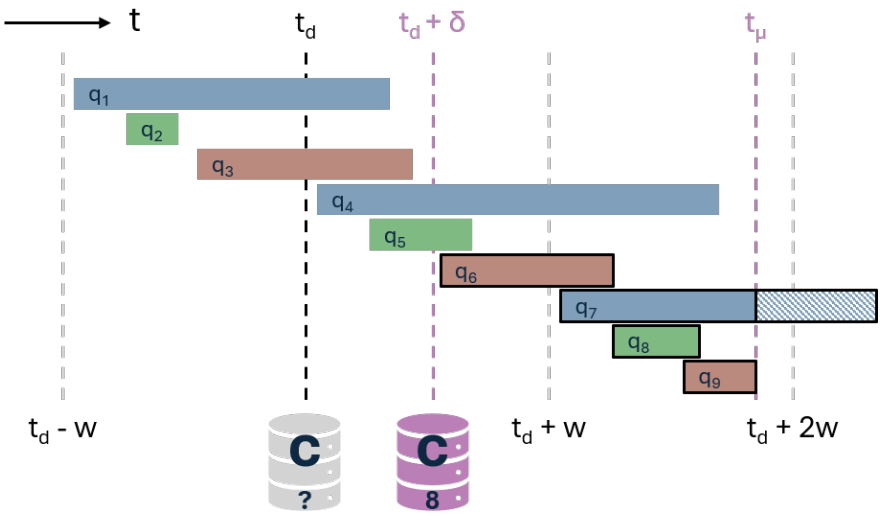


Figure 3.4: The Spinup Size Selector evaluates each scaling action at t_d by simulating copies of the arrival window $\mathcal{W}^a$. It stops once μ queries (here $\mu = 3$) issued after the new cluster is available ($t_d + \delta$; queries outlined in black) finish executing.

We then continue until we have simulated μ query completions (lines 19–20). The core simulation loop is mostly the same (lines 21–27), with the exception that we record the latency of completing queries that arrived after $t_{\text{available}}$ (lines 23–24). After exiting the simulation, we augment this set of μ completed queries with the predicted latencies of outstanding queries (line 28) and compute the SLO violation rate and cost (lines 29–30). We then find the best pair of SLO violation rate and cost according to an SLO violation rate target τ_{target} (line 31, see the end of Section 3.2). If $\emptyset$ (i.e. the do-nothing baseline) can achieve these values, we return it, suppressing trigger re-evaluation for δ seconds to prevent immediate re-firing (lines 32–35, see Algorithm 3.2). Otherwise, we return the largest cluster tied for the best outcome (line 36), erring on the side of caution when multiple sizes perform equally well. The parameters μ and w can be periodically optimized by the Policy Tuner, as we will examine in Section 3.7.5.

Figure 3.4 shows an example for $\mu = 3$, where the x-axis is time and each rectangle is a query. The query arrivals q_1–q_3 in the arrival window $\mathcal{W}^a$ are replayed twice until q_9's completion at t_μ marks the third completion after $t_d + \delta$, the time the new (purple) cluster of size 8 became available. The simulated latencies of q_6, q_8, and q_9, and the predicted latency of q_7 as of t_μ, contribute to v_8 and k_8.

Algorithm 3.3 Computational Time Complexity. In the preparation phase (lines 3–11), the number of query arrivals will be $O(|\mathcal{W}^a| \cdot \frac{\delta}{w})$. Let the number of query arrivals required on lines 21–27 until μ queries complete be F_s when the new cluster has size s (or $s = \emptyset$), and let $F^{\max} = \max_{s \in \mathcal{S}'} F_s$. Let each arriving query q encounter R_q running queries, among which any finished queries must be identified ($O(R_q)$) before q is routed ($O(R_q)$), and let $R^{\max} = \max_q R_q$. For the drain phase and SLO/cost calculations (lines 28–30) we similarly incur $O(R_q)$. Therefore, the overall complexity of FindBestSpinupSize is $O((|\mathcal{W}^a| \cdot \frac{\delta}{w} + F^{\max}) \cdot R^{\max})$.

3.7 Policy Tuner

As described in Section 3.6, the Autoscaler can use the recent workload to adjust the active cluster. However, over longer time horizons, additional optimization opportunities open up. The Policy Tuner, invoked at the granularity of several hours[1], captures such opportunities, as follows. Each incoming query is added to the Workload Reservoir (Section 3.7.1), which can be used by the Workload Forecaster (Section 3.7.2) to draw forecasted workloads. The Batch Simulator (Section 3.7.3) can efficiently simulate these workloads under different configurations, empowering the Spinup Scheduler (Section 3.7.4) to schedule proactive cluster spinups and the Configuration Tuner (Section 3.7.5) to optimize Autoscaler knobs.

3.7.1 Workload Reservoir

Each query that reaches AutoSLO is also added to the Workload Reservoir. It consists of two tables indexed by calendar date and hour of day. The *count table* records, for each (*date*, *hour*, *query_template*) triple, how many times that query template was issued. The *arrivals table* records the within-hour second offset of every query arrival in each (*date*, *hour*) bin. Both tables have configurable sampling policies to keep their footprint bounded, but they do not need to be kept in memory, since they are only used infrequently by the Policy Tuner.

3.7.2 Workload Forecaster

The Workload Forecaster generates forecasted workloads for the day ahead and partitions them into training and validation splits. In our current implementation, it iterates over hours of the day and samples query texts from all available same-day-and-hour-of-week bins. It applies a decay to their sampling probability, so that samples from k weeks ago are weighted by λ^{k-1} (with $\lambda \in (0, 1)$). To assign arrival times to each forecasted query, it calculates interarrival deciles from the *arrivals table* (over the same-day-and-hour-of-week bins) and samples uniformly within each decile. For sparse bins, it falls back to uniform sampling.

Note that the forecasted workloads only need to approximate the load contours of the real workload, so that coarse scaling decisions can be pre-planned. Other forecasting and sampling policies are also supported (e.g., a single prior day, or an equally weighted 7-day window), but the above policy best balanced simplicity and predictive performance.

3.7.3 Batch Simulator

SimulateBatch is the shared evaluation primitive invoked by every phase of the Policy Tuner. It accepts a set of *execution configurations* $\{E_i\}$ and a set of workloads W, and returns a result matrix where entry $\mathcal{R}[i][j]$ holds the simulation outcome for configuration E_i on workload $W[j]$. All $|\{E_i\}| \times |W|$ simulations are dispatched to a process pool and run in parallel in an optimized fashion. Each *execution configuration* specifies any scheduled cluster spinups for the workload ahead (cluster size and spinup time), as well as values for the Autoscaler configuration parameters: T_{idle} and $T_{\text{min\_lifetime}}$ (Section 3.6.2.1); τ_{trigger} and θ (Section 3.6.2.2); and μ and w (Section 3.6.3).

[1]Below we assume that the Policy Tuner is invoked daily, but this is not strictly required.

3.7.4 Spinup Scheduler

3.7.4.1 High-Level Workflow

The Spinup Scheduler augments execution configurations with scheduled cluster spinups for periods of predictable congestion. Conceptually, it greedily evolves an initial execution configuration by adding spinups over at most η rounds. In each round it simulates the current configuration on the training workloads, identifies promising times at which additional capacity may help reduce SLO violations, and accepts the best spinup candidate only if it improves upon the existing configuration.

Algorithm 3.4 implements this search for a set of initial configurations $\mathcal{E} = \{E_i^0\}_{i\in\mathcal{I}}$. It maintains the best configuration found so far for each candidate, E_i^{best}, and an index set of candidates that remain eligible for further spinups, alive_at_all (lines 2–3). The outer loop performs up to η greedy rounds (line 4). At the beginning of each round, all still-active candidates are simulated on the training workloads in a single batch (line 5). For each candidate, the scheduler aggregates its current SLO violation rate and cost (across the training workloads), then calls FindGoodSpinupTime (Algorithm 3.5) to obtain a ranked list Γ_i of promising spinup times (lines 6–8). Candidates with no promising times are removed from alive_at_all (line 9).

For the remaining candidates, the scheduler searches for the best-performing spinup to add in the current round. It initializes each candidate's attempt counter and places all active candidates into a set alive_this_round (lines 10–11). The inner loop then evaluates candidates in attempt waves (lines 12–26). In each wave, every still-alive candidate i is paired with every eligible cluster size $s \in \mathcal{S}$, producing a trial configuration that adds a spinup at the candidate's currently attempted time $\Gamma_i(\text{attempt}_i)$ with size s (lines 13–14). These trial configurations are simulated in parallel on the training workloads, before we aggregate their violation rates and costs (lines 15–17).

The scheduler then calculates, for each candidate, the best performance (v_i^b, k_i^b) among all new configurations and the current baseline using BestWithTarget (line 18). If this new best performance differs from the previous best performance, a configuration that achieves the new best performance is promoted to E_i^{best} and the candidate leaves the current round because it has accepted a spinup (lines 19–21). If no cluster size improves the candidate at the current time, the scheduler advances to the next promising time in Γ_i, up to the per-round attempt limit ϕ (lines 22–23). If the candidate exhausts all allowed attempts without finding an improvement, it is removed both from the current round and from future rounds (lines 24–26). Thus, each outer round adds at most one spinup per candidate, and candidates stop being considered once no useful spinup can be found.

After the greedy training phase, we evaluate the best configuration for each initial candidate on the validation workloads (lines 27–28). We then select and return a configuration $E^\star$ according to the target-aware lexicographic objective (lines 29–31). The batching in Algorithm 3.4 does not change the greedy search logic that underpins our overall approach; it simply improves simulation throughput.

Algorithm 3.4 Computational Time Complexity. Let B be the maximum parallel simulation batch size and s the maximum simulation duration. Then the overall computational time complexity of ScheduleSpinups is $O\big(|\mathcal{I}| \cdot \frac{s}{B} \cdot \big(\eta \cdot |\mathcal{S}| \cdot \phi \cdot |W^{\text{tr}}| + |W^{\text{val}}|\big)\big)$.

Algorithm 3.4 The Spinup Scheduler.

Inputs: Initial execution configurations $\mathcal{E} = \{E_i^0\}_{i \in \mathcal{I}}$, training workloads W^{tr}, validation workloads W^{val}, target SLO violation rate τ_{target}

Configuration: Maximum spinups η, aggregation function α, eligible cluster sizes $\mathcal{S}$, maximum attempts per round ϕ

Outputs: Optimized config $E^\star$ with scheduled spinups

function ScheduleSpinups($\mathcal{E}$, W^{tr}, W^{val}, τ_{target})
  $E_i^{\text{best}} \leftarrow E_i^0$ for all $i \in \mathcal{I}$
  alive_at_all $\leftarrow \mathcal{I}$
  for $r \in [0, \ldots, \eta)$ **do**
    $\{\Lambda_i\}_{i \in \text{alive\_at\_all}} \leftarrow$ SimulateBatch(W^{tr}, $\{E_i^{\text{best}}\}_{i \in \text{alive\_at\_all}}$)
    for $i \in$ alive_at_all **do**
      $v_i^{\text{base}}, k_i^{\text{base}} \leftarrow$ AggPerf(Λ_i, α)
      $\Gamma_i \leftarrow$ FindGoodSpinupTime(Λ_i, τ_{target})
      if $\Gamma_i = \emptyset$ **then** alive_at_all $\leftarrow$ alive_at_all $\setminus \{i\}$
    attempt$_i \leftarrow 0$ for all $i \in$ alive_at_all
    alive_this_round $\leftarrow$ alive_at_all
    while alive_this_round $\neq \emptyset$ **do**
      for $(i, s) \in$ alive_this_round $\times \mathcal{S}$ **do**
        $E_{i,s} \leftarrow$ AddSpinup(E_i^{best}, time $= \Gamma_i$(attempt$_i$), size $= s$)
      $\{\Lambda_{i,s}\} \leftarrow$ SimulateBatch(W^{tr}, $\{E_{i,s}\}$)
      for $i \in$ alive_this_round **do**
        $v_{i,s}, k_{i,s} \leftarrow$ AggPerf($\Lambda_{i,s}, \alpha$) for all $s \in \mathcal{S}$
        $(v_i^{\text{b}}, k_i^{\text{b}}) \leftarrow$ BestWithTarget($\{(v_{i,s}, k_{i,s})\}_s \cup \{(v_i^{\text{base}}, k_i^{\text{base}})\}$, τ_{target})
        if $(v_i^{\text{b}}, k_i^{\text{b}}) \neq (v_i^{\text{base}}, k_i^{\text{base}})$ **then**
          $E_i^{\text{best}} \leftarrow E_{i,s^\star}$ for any $s^\star$ such that $(v_{i,s^\star}, k_{i,s^\star}) = (v_i^b, k_i^b)$
          alive_this_round $\leftarrow$ alive_this_round $\setminus \{i\}$
        else if attempt$_i + 1 <$ min($|\Gamma_i|$, ϕ) **then**
          attempt$_i \leftarrow$ attempt$_i + 1$
        else
          alive_this_round $\leftarrow$ alive_this_round $\setminus \{i\}$
          alive_at_all $\leftarrow$ alive_at_all $\setminus \{i\}$
    $\{\Lambda_i^{\text{val}}\} \leftarrow$ SimulateBatch(W^{val}, $\{E_i^{\text{best}}\}_{i \in \mathcal{I}}$)
    $v_i^{\text{val}}, k_i^{\text{val}} \leftarrow$ AggPerf(Λ_i^{val}, α) for all $i \in \mathcal{I}$
    $(v^\star, k^\star) \leftarrow$ BestWithTarget($\{(v_i^{\text{val}}, k_i^{\text{val}})\}_{i \in \mathcal{I}}$, τ_{target})
    $E^\star \leftarrow E_{i^\star}$ for any $i^\star$ such that $(v_{i^\star}^{\text{val}}, k_{i^\star}^{\text{val}}) = (v^\star, k^\star)$
  return $E^\star$

3.7.4.2 Finding Good Scheduled Spinup Times

Algorithm 3.5 Congestion point detection.

Inputs: Per-workload simulation logs $\{\Lambda_i\}$, target SLO violation rate τ_{target}
Configuration: Delinquency threshold k, spinup delay δ, spacing $\sigma_{\min}$
Outputs: List Γ of promising spinup placement times.

```
1:  function FindGoodSpinupTime({Λ_i}, τ_target)
2:      events ← All query arrival/completion events from simulation logs {Λ_i}
3:      events ← Sort(events, {time, Asc})
4:      V_i ← 0, N_i ← 0, D_i ← False for all i
5:      n_D ← 0;   inEpoch ← False;   Γ ← [ ]
6:      for j = 0 to |events| − 2 do
7:          e ← events[j]
8:          if e.type = Start then
9:              V_e.i += e.v;   N_e.i += 1
10:         else
11:             V_e.i −= e.v;   N_e.i −= 1
12:         d′ ← (V_e.i / N_e.i ≥ τ_target if N_e.i > 0 else False)
13:         if d′ ≠ D_e.i then
14:             n_D += (d′ ? +1 : −1);   D_e.i ← d′
15:         if events[j+1].time = e.time: continue
16:         if n_D ≥ k and not inEpoch then
17:             inEpoch ← True
18:             t_cand ← e.time − δ
19:         else if n_D < k and inEpoch then
20:             inEpoch ← False
21:             if {t ∈ Γ | |t − (t_cand)| < σ_min} = ∅ then
22:                 Γ.Append(t_cand)
23:     if inEpoch and {t ∈ Γ | |t − (t_cand)| < σ_min} = ∅ then
24:         Γ.Append(t_cand)
25:     return Γ
```

Per Section 3.7.4.1, the Spinup Scheduler needs to determine promising times for scheduled spinups. Algorithm 3.5 achieves this in a single linear scan. It begins by building a unified sorted timeline of query arrival and completion events across all per-training-workload simulation logs for the same execution configuration, pre-gathering per-query SLO violation information (lines 2–3). It then initializes per-workload running-sum accumulators V_i, N_i, delinquency flags D_i, the global delinquent-workload count n_D, and the epoch-tracking state (lines 4–5). For each event e, the running sums for the affected workload are updated (lines 7–11), and the workload's delinquency flag and n_D are refreshed (lines 12–14). Using running sums means that SLO adherence is evaluated in $O(1)$ per event. Zero-length intervals between simultaneous events are skipped (line 15). When n_D first reaches the threshold k, a *congestion epoch* begins and a candidate spinup time is recorded δ seconds earlier (lines 16–18); when n_D drops back below k, the epoch ends and the candidate spinup time is appended to Γ only

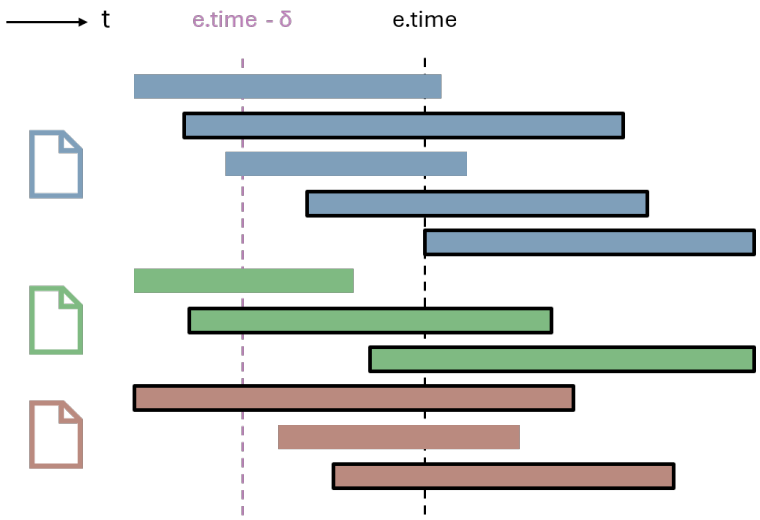


Figure 3.5: Candidate spinup time determination. Here, queries with bold outlines violated their SLOs. The indicated time is the first candidate spinup time for $\tau_{\text{target}} = 0.6$ and $k = 3$.

if it is at least $\sigma_{\min}$ ahead of the previous candidate (lines 19–22). Any open epoch at the end of the timeline is closed after the loop (lines 23–24). The returned list Γ (line 25) is implicitly sorted in ascending order by placement time.

Figure 3.5 shows an example with 3 simulation logs, with queries colored blue, green, and red, respectively. Bold outlines indicate SLO violations. For $\tau_{\text{target}} = 0.6$ and $k = 3$, all 3 workloads have more than 60% of their running queries violating the SLO once the fifth blue query arrives, suggesting a candidate spinup time δ s earlier.

Algorithm 3.5 Computational Time Complexity. Let $Q = \sum_i |\Lambda_i|$, so that $|\text{events}| = 2Q$. The event timeline is sorted and then processed once, so that FindGoodSpinupTime is $O(Q \log Q)$.

3.7.5 Configuration Tuner

After we determine the scheduled spinups, the Configuration Tuner optimizes the parameters of the Autoscaler. It first generates candidate execution configurations using a configurable strategy; options include `grid` (exhaustive cross-product of specified values), `random` (budget-limited random sample from the grid), `coordinate_descent` (iterative per-parameter optimization), and `adaptive_batch` (progressive widening with early stopping). It then evaluates all candidate configurations on the training workloads in parallel via SimulateBatch and retains only the top-k with respect to a configurable aggregate metric over the training workloads (e.g. mean). These are re-evaluated (using SimulateBatch) on the validation workloads, and the configuration with the best aggregate validation performance is returned.

3.8 Evaluation

We implemented AutoSLO in around 26,000 lines of Python [143]. After explaining our evaluation setup (Section 3.8.1), we present end-to-end experiments (Section 3.8.2) and assess the effectiveness (Sections 3.8.3–3.8.7) and efficiency (Section 3.8.8) of each component. In Tables 3.2–3.5, VR denotes the SLO violation rate and bold/underline indicates the best/second-best VR per row.

3.8.1 Evaluation Setup

3.8.1.1 Configuration

We used a machine with two 20-core 2.10 GHz Intel Xeon Gold 6230 CPUs [106] and 256 GiB of memory, running Linux 6.9.7-arch1-1. From this machine, we used `psycopg2` and `boto3` to interact with Amazon Redshift Serverless. For continuity, we refer to workgroups as "clusters" below. We used the current Amazon Redshift Serverless price for `us-east-1` ($0.375/RPU/$h$), where an RPU is the unit of cluster size.

3.8.1.2 Workloads

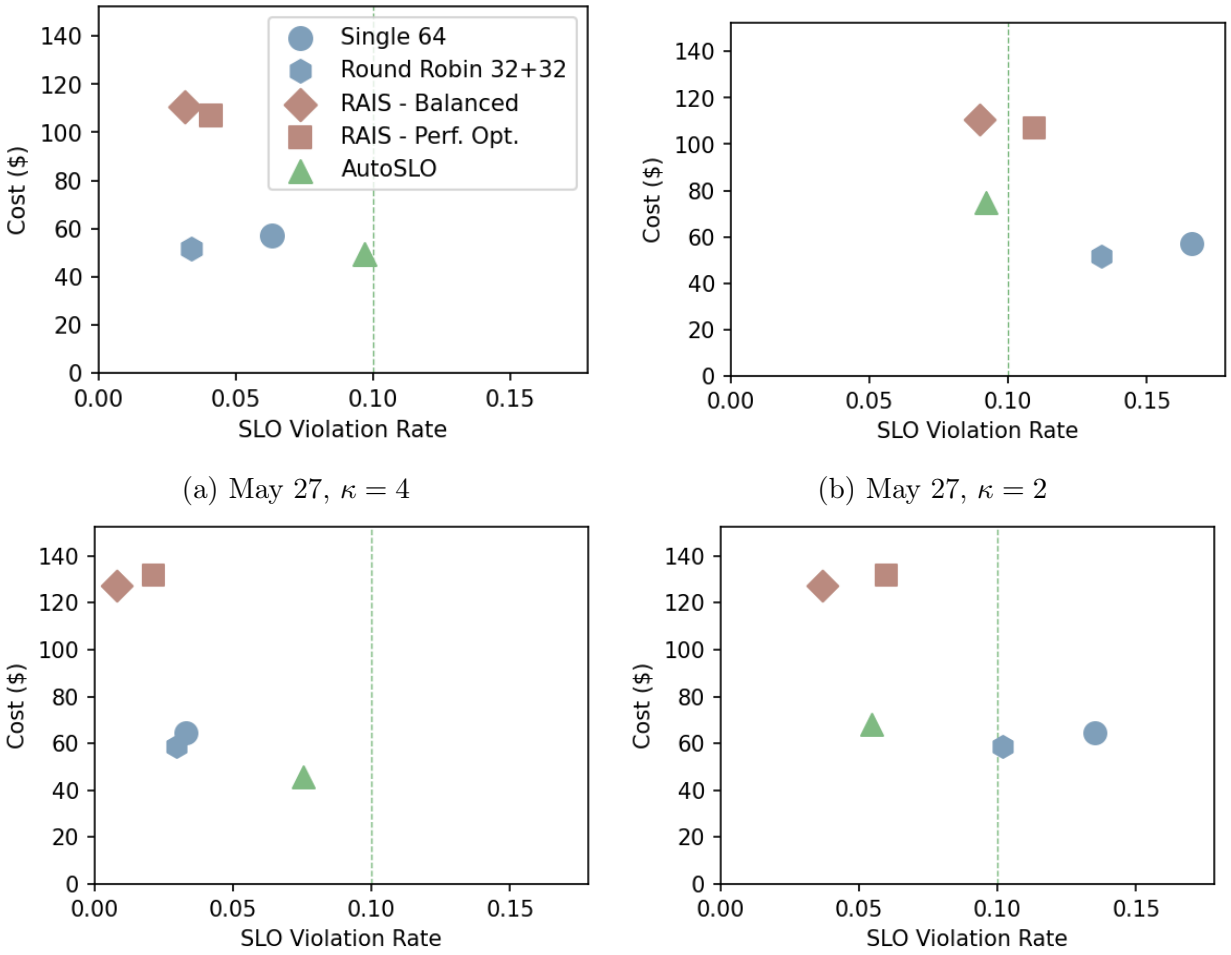
All experiments were executed against TPC-DS SF 1000 (1 TB). Several of our experiments used a workload consisting of 3 copies of a single query text per benchmark query template, for a total of 297 queries, shuffled and issued according to a uniform Poisson process with rate λ. We call such workloads BaseWorkload(λ). Other experiments each describe the workload(s) they use, as needed.

3.8.1.3 SLOs

We ran BaseWorkload in a closed loop against a 16-RPU Amazon Redshift Serverless cluster and recorded the maximum latency $\ell_{\text{baseline}}(t)$ per query template t. Below, we let the SLO of each query q of template t be $\kappa \cdot \ell_{\text{baseline}}(t)$, with higher κ indicating easier-to-satisfy SLOs.

3.8.2 End-to-End Effectiveness

We begin by evaluating AutoSLO end-to-end using Redbench [229], which creates realistic workloads based on the real production traces in Redset [221]. Redset includes query timing and complexity information for real Amazon Redshift customer workloads, but lacks actual query texts and data. Redbench bridges this gap by mapping each Redset query to appropriate query texts from TPC-DS [179]. We obtained an executable version of the SELECT workload faced by a particular Redset cluster (provisioned cluster 157) using Redbench's join matching algorithm. For practical purposes, we compressed the interarrival times in the workload by a factor of 6 during simulation/execution, so that each day's workload took 4 hours to execute. In the Policy Tuner, this compression was only applied in the Batch Simulator; that is, the Workload Reservoir was maintained using real (not scaled) arrival times, based on which the Workload Forecaster also operates.



(a) May 27, $\kappa = 4$

(b) May 27, $\kappa = 2$

(c) April 15, $\kappa = 4$

(d) April 15, $\kappa = 2$

Figure 3.6: End-to-end performance of AutoSLO.

Based on this workload, we derived 4 experimental scenarios as the cross product of two variables, as shown in Figure 3.6. First, we varied the workload day: we used the last Monday (May 27, 2024) and middle Monday (April 15, 2024) of the workload, to ensure non-overlapping recent workload histories in the Policy Tuner. Then, we examined two different levels of SLO difficulty, for $\kappa \in \{2, 4\}$. Across scenarios, our SLO violation rate target per Problem 3.3 was $\tau_{\text{target}} = 0.1$, shown as a dashed green line.

We compare the performance of AutoSLO (green triangle), after using the Policy Tuner over 30 days of past data, against two families of baselines. In blue, we have a pair of naive baselines using a total of 64 RPU each, either in a single cluster (circle) or in two clusters with 32 RPU each, with round-robin query routing (hexagon). In red, we have a single cluster using Redshift Serverless AI-driven Scaling [157], with the price-performance slider set to either "balanced" (position 50, diamond) or "high performance" (position 100, square), and a specified maximum of 128 RPU[2].

As evident, AutoSLO provides the best performance across scenarios, meeting τ_{target} for all 4 scenarios and reducing cost by a mean of 26.4% compared to the next-best baseline per scenario. Compared to the only other baseline that meets τ_{target} in all 4 scenarios (RAIS-Balanced), AutoSLO reduces cost by a mean of 49.6%.

Per Section 3.1, neither family of baselines allows explicitly specifying SLOs or τ_{target}, which translates to ineffective cost-performance tradeoffs. While the SLOs are loose ($\kappa = 4$, Figures 3.6a and 3.6c), all baselines meet τ_{target} but remain unaware of it, spending additional resources to reduce latency in a regime where it no longer matters. When the SLOs tighten ($\kappa = 2$, Figures 3.6b and 3.6d), no baseline can be informed, since they are SLO-unaware; they each act the same as before, leading to higher SLO violation rates, with the naive baselines all missing τ_{target}. AutoSLO instead adapts, increasing spending just enough to meet τ_{target}.

Finding from 3.8.2: End-to-end SLO Adherence

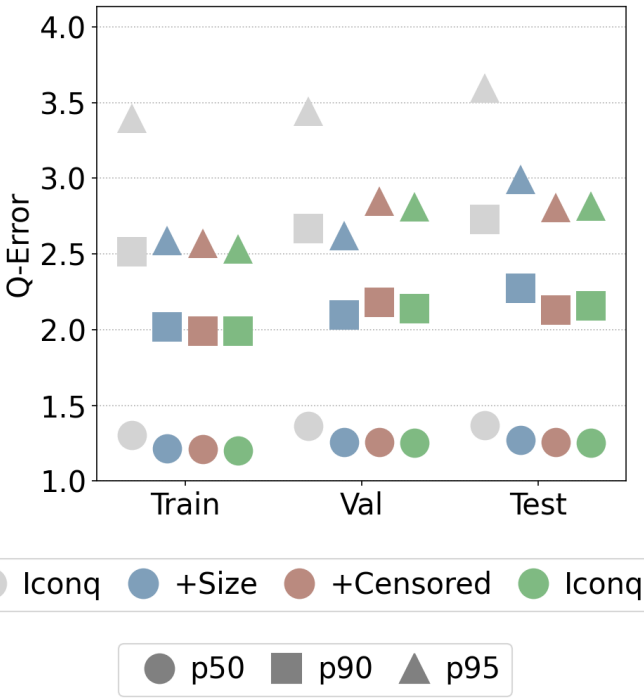
AutoSLO can cost-effectively maintain latency SLOs on realistic workloads at the specified target level, reducing cost by a mean of 49.6% against the cheapest RAIS baseline per scenario.

3.8.3 Latency Predictor (Iconq+)

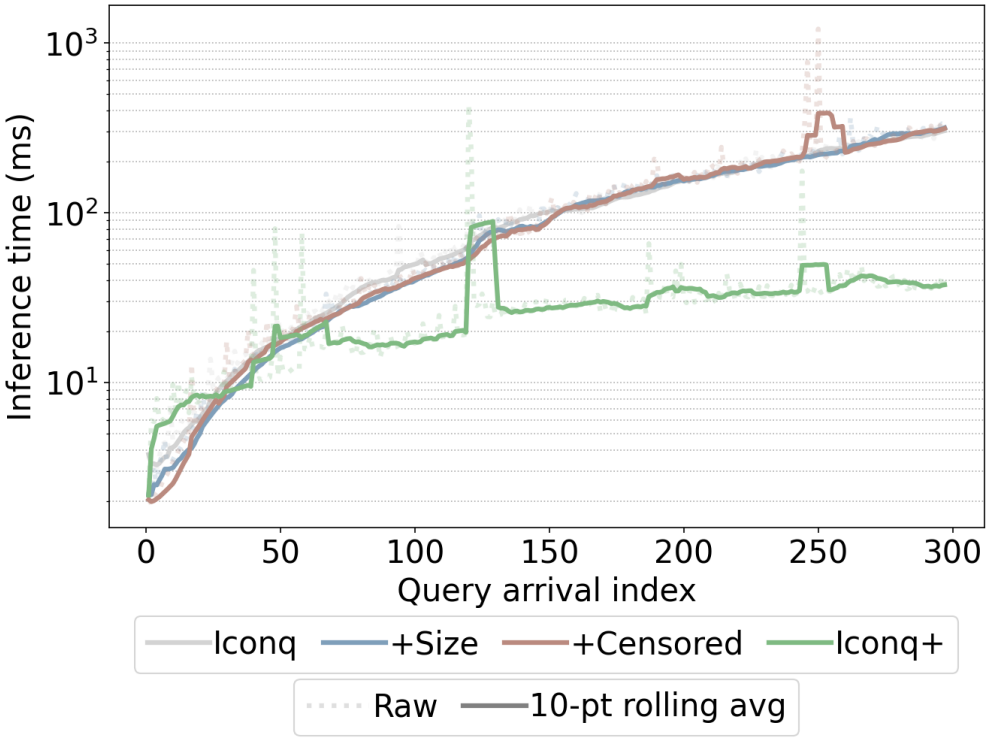
3.8.3.1 Variants

To assess our Latency Predictor, we compare four model variants in an ablation study: Iconq includes no changes [235]; +Size enhances the interaction feature vectors with cluster size information (Section 3.4.2); +Censored additionally trains the model on censored observations sampled with probability 0.5% (Section 3.4.3); finally, Iconq+ also supports incremental predictions (Section 3.4.4).

[2]We observed that throughout the execution of our workloads, RAIS billed for the full 128 RPU (according to the billing timer of Section 3.2) regardless of the slider setting.



(a) Accuracy



(b) Efficiency

Figure 3.7: Iconq+ delivers superior accuracy and efficiency.

3.8.3.2 Training Details

We trained each variant on data from executing BaseWorkload for $\lambda \in [0.05, 0.1, 0.2, 0.5, 1.0]$ on each of 4, 8, 16, and 32 RPU. On each cluster size, we also executed four "phased" versions of BaseWorkload, alternating between x queries at $\lambda = 0.05$ and 10 queries at λ_{high}, for $x \in \{40, 15\}$ and $\lambda_{high} \in \{0.2, 0.5\}$. For practicality, we imposed a one-hour query timeout.

We also performed two training stability modifications compared to the original Iconq paper [235]. First, we only trained the underlying Stage [234] model (used to derive neighbor-unaware latency predictions included in interaction feature vectors as query complexity proxies) on isolated query executions, i.e., executions with no neighbors. This made the resulting predictions better reflect intrinsic query complexity rather than interference from concurrent queries. Second, we interpreted data labels and model latency predictions in *logarithmic space*, guaranteeing positivity after re-exponentiation and aligning naturally with multiplicative error. This meant we no longer needed to heavily penalize negative/tiny predictions in the loss function, so we used a simpler Q-error-based objective instead.

3.8.3.3 Results

Figure 3.7a evaluates the accuracy of each model. It shows the 50th, 90th, and 95th percentiles of the Q-error achieved by each variant on each of its folds. As evident, adding the cluster-size-related features in +Size offers noticeable benefits on the test set, in addition to being necessary for cross-cluster-size comparisons when routing. +Censored extends these benefits further, especially at p90 and p95, with Iconq+ achieving similar accuracy.

Figure 3.7b concerns the efficiency of the different variants on BaseWorkload with $\lambda = 0.1$. For each query arrival, we predicted its own latency and updated the latency predictions of previous queries, assuming no query had completed yet. This is not reflective of a real execution of this particular workload, but this experiment is only stress-testing total inference latency per arrival under increasing load. As evident, inference time grows with increasing congestion, but Iconq+ can use incremental inference to sustain sub-40 ms inference latency. In contrast, every other variant requires around 300 ms by the end of the workload.

Finding from 3.8.3: Latency Predictor (Iconq+)

Iconq+ is effective: adding cluster size features and censored observations noticeably improves prediction accuracy, while incremental inference keeps inference time manageable.

3.8.4 Query Router

We continue by assessing the performance of the Query Router. We compare three query routing approaches. Round-Robin cycles among the active clusters, as a naive baseline. Stage + Cost Opt. uses the underlying Stage [234] model of our trained Iconq+ instance to derive concurrency-unaware latency predictions on each active cluster and then routes each query to a cluster where its SLO will be met, breaking ties by cost. Such concurrency-agnostic routing mirrors the approach of published descriptions from cloud vendors [157, 189]. Ours (Iconq+ + Cost Opt.) represents our approach as described in Section 3.5, where the concurrency-aware Iconq+ model is used and the risk to the SLOs of running queries is also assessed.

Table 3.2: Query Router evaluation.

λ	κ	C	Round Robin		Stage + Cost.Opt.		Iconq+ + Cost.Opt.	
			VR	Cost	VR	Cost	VR	Cost
0.1	4	[32, 16]	0.0269	$15.78	**0.0135**	$11.01	0.0168	$11.17
0.1	4	[16, 16]	0.0976	$10.40	0.0370	$7.99	**0.0303**	$8.91
0.1	2	[32, 16]	0.0842	$15.78	0.0808	$10.14	**0.0505**	$12.28
0.1	2	[16, 16]	0.2088	$10.57	0.1414	$7.43	**0.1246**	$8.77
0.2	4	[32, 16]	0.1414	$7.92	0.0707	$7.56	**0.0202**	$7.60
0.2	4	[16, 16]	0.3064	$5.41	0.2761	$4.81	**0.1919**	$5.29
0.2	2	[32, 16]	0.1919	$8.13	0.1919	$6.92	**0.1044**	$7.71
0.2	2	[16, 16]	0.4512	$5.48	0.5118	$5.20	**0.3300**	$4.98
Mean Δ			—	—	-24.3%	-19.3%	**-47.8%**	-12.9%

We evaluated each approach in 8 scenarios, derived as follows: (i) we ran BaseWorkload with either $\lambda = 0.1$ or $\lambda = 0.2$; (ii) we defined SLOs using either $\kappa = 4$ or $\kappa = 2$; and (iii) we either routed between two clusters with 16 RPU each ($\mathcal{C} = [16, 16]$), or between one 32-RPU cluster and one 16-RPU cluster ($\mathcal{C} = [32, 16]$).

Table 3.2 shows the results. Query latency prediction with Stage already outperforms Round-Robin, reducing the SLO violation rate by a mean of 24.3%. It also reduces cost by a mean of 19.3%, since the Query Router also considers cost. However, using Iconq+ can improve even further over Round-Robin, achieving a mean SLO violation rate reduction of 47.8% while still lowering cost.

Finding from 3.8.4: Query Router

Our Query Router outperforms the alternatives, reducing SLO violation rate by a mean of 47.8% and cost by a mean of 12.9%.

3.8.5 Autoscaler – Spinup Size Selector

We next focus on the cluster sizes selected by the Spinup Size Selector. We compare four approaches to modifying the active cluster set. No-Op makes no changes to the active clusters. Horizontal spins up an additional cluster of the same size as the existing one, using our Query Router to route between the two. Add Cluster w/ Ours represents our approach described in Section 3.6. Vertical w/ Ours uses the simulation-based decision-making of our method, but instead of considering what single cluster to *add*, it considers what single cluster to *have* (i.e. it stops routing to the existing cluster once the new one becomes available).

We evaluated each approach in 8 scenarios, derived as in Section 3.8.4, but we started with only a single cluster of size either 8 or 16 RPU. We executed 20% of the workload queries, at which point we *forcibly triggered* autoscaling. We then let the following 60% of the workload elapse, for scaling to take effect and any congestion-affected queries around the autoscaling point to drain. We report the SLO violation rate and cost over the last 20% of the workload, once a steady state has been reached.

Table 3.3: Spinup Size Selector steady-state evaluation.

λ	κ	C	No-Op		Horizontal		Vertical w/Ours		Add Cluster w/Ours	
			VR	Cost	VR	Cost	VR	Cost	VR	Cost
0.1	4	[8]	0.9833	$2.81	0.8167	$1.99	0.2000	$1.79	**0.0167**	$2.16
0.1	4	[16]	0.1000	$1.65	**0.0000**	$2.67	0.0833	$1.59	**0.0000**	$2.35
0.1	2	[8]	0.9833	$2.86	1.0000	$2.04	**0.0167**	$3.40	0.1000	$2.09
0.1	2	[16]	0.3667	$1.74	0.0667	$3.14	**0.0167**	$3.40	0.0833	$2.81
0.2	4	[8]	1.0000	$3.27	1.0000	$3.55	**0.0333**	$3.11	**0.0333**	$2.88
0.2	4	[16]	0.8333	$2.21	0.1167	$1.83	0.0333	$2.14	**0.0000**	$2.71
0.2	2	[8]	1.0000	$3.31	0.9500	$3.34	**0.0667**	$3.12	**0.0667**	$3.15
0.2	2	[16]	0.9000	$2.05	0.4167	$2.26	0.0667	$2.29	**0.0500**	$2.75
Mean Δ			—	—	-42.7%	+11.0%	-83.6%	+9.1%	**-93.7%**	+11.8%

The results are presented in Table 3.3. The cluster size selected by the Spinup Size Selector outperforms horizontal scaling: Add Cluster w/ Ours achieves a mean SLO violation rate reduction of 93.7%, compared to 42.7% for Horizontal, while the two methods impact cost similarly. Relying on our algorithm for vertical scaling is also promising, with Vertical w/ Ours achieving a mean SLO violation rate reduction of 83.6%. However, Add Cluster w/ Ours performs best on average, making effective use of the pre-existing cluster.

Finding from 3.8.5: Autoscaler – Spinup Size Selector

The Spinup Size Selector recommends appropriate cluster sizes: using the new cluster alongside the existing one achieves a mean SLO violation rate reduction of 93.7%.

3.8.6 Autoscaler – Spinup Trigger

We previously focused on *which* cluster the Autoscaler spins up, once triggered; we now also consider *when* it is triggered. We compare four trigger approaches, all of which then use the standard Spinup Size Selector. No-Op is never triggered. Queue@32 is triggered whenever there are ≥ 32 outstanding queries, mirroring the queuing-based triggers of Auto-WLM [189] and RAIS [157]. Observed is similar to Algorithm 3.2 but without R (line 6); it only acts on *observed* latencies. Finally, Ours represents our approach described in Section 3.6.2.2, which augments Observed with the projected SLO violation status of currently running queries.

We used the same 8 experimental scenarios as in Section 3.8.5. Since each approach could decide when and what to spin up, we compare SLO violation rate and cost throughout the whole workload. As shown in Table 3.4, Ours clearly outperforms the alternatives on average, reducing the SLO violation rate by a mean of 54.6%.

Finding from 3.8.6: Autoscaler – Spinup Trigger

Our Autoscaler spinup trigger, combining observed and projected SLO violations, outperforms the alternatives and delivers a mean SLO violation rate reduction of 54.6%.

Table 3.4: Autoscaler Spinup Trigger evaluation.

λ	κ	C	No-Op		Queue@32		Observed		Ours	
			VR	Cost	VR	Cost	VR	Cost	VR	Cost
0.1	4	[8]	0.9697	$4.56	0.4276	$6.10	0.4007	$5.88	**0.3199**	$7.31
0.1	4	[16]	0.2896	$5.31	0.3064	$5.09	**0.2222**	$7.18	0.2626	$7.83
0.1	2	[8]	0.9865	$4.71	0.4242	$8.67	0.8013	$6.38	**0.2458**	$11.10
0.1	2	[16]	0.4882	$5.24	0.4916	$5.25	0.2727	$8.68	**0.2660**	$8.74
0.2	4	[8]	0.9899	$4.13	**0.2896**	$5.54	0.9327	$5.04	0.3098	$5.19
0.2	4	[16]	0.8114	$3.71	0.4141	$5.72	**0.3434**	$5.91	0.4007	$4.95
0.2	2	[8]	0.9966	$4.16	0.4680	$5.37	0.5791	$5.91	**0.4040**	$6.02
0.2	2	[16]	0.9495	$3.65	0.4882	$5.45	0.4680	$5.77	**0.3670**	$7.27
Mean Δ			—	—	-41.0%	+35.1%	-37.6%	+43.3%	**-54.6%**	+64.1%

Table 3.5: Policy Tuner evaluation.

Day	κ	No Past Data		Past 1 Day		Past 7 Days		Past 30 Days	
		VR	Cost	VR	Cost	VR	Cost	VR	Cost
5/27	4	0.1997	$41.47	0.1149	$46.14	**0.0639**	$51.75	0.0970	$49.25
5/27	2	0.2162	$61.72	0.1092	$71.39	0.1293	$68.38	**0.0920**	$74.89
4/15	4	0.0833	$42.64	**0.0664**	$44.15	0.0848	$45.65	0.0752	$45.82
4/15	2	0.1600	$59.80	**0.0546**	$69.78	0.0575	$68.11	**0.0546**	$68.01
Mean Δ		—	—	-44.6%	+11.8%	-42.6%	+14.1%	**-46.1%**	+15.3%

3.8.7 Policy Tuner

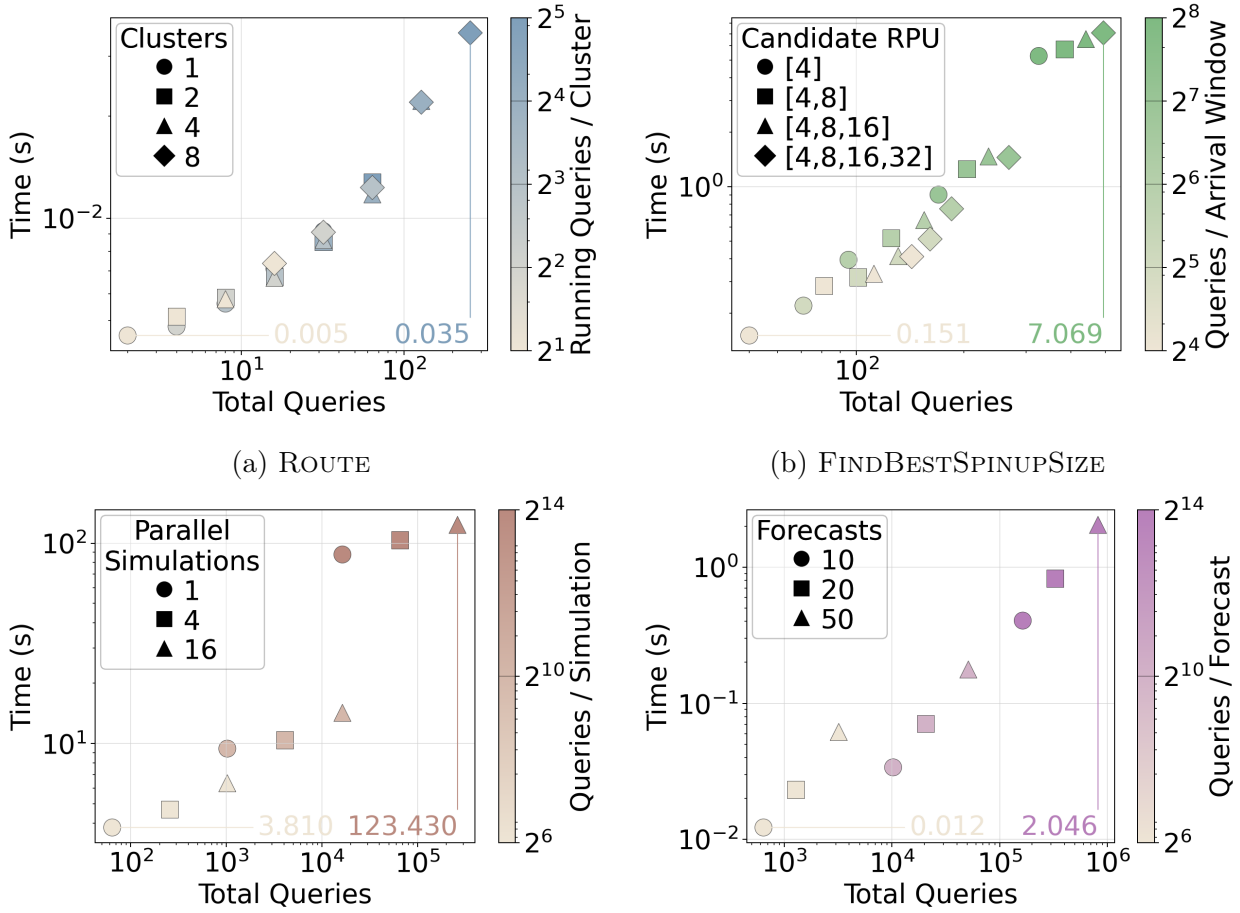
We next evaluate the impact of the Policy Tuner on the downstream performance of the execution configuration it produces. In particular, for the day/κ scenarios used in Section 3.8.2, we used the Policy Tuner to optimize a configuration on 0, 1, 7, or 30 days of past data. We then used each optimized configuration to execute the target workload. As seen in Table 3.5, the Policy Tuner can leverage even a single day of workload history to reduce the SLO violation rate by a mean of 44.6%, increasing cost by a mean of just 11.8%. With access to 30 days of history, the mean SLO violation rate reduction further improves to 46.1%, but most of the tuning benefit can already be reaped with minimal history.

Finding from 3.8.7: Policy Tuner

The Policy Tuner can efficiently produce better execution configurations, reducing the SLO violation rate by a mean of 44.6% with access to a single day of workload history.

3.8.8 AutoSLO Efficiency

We close with efficiency micro-experiments. In Figure 3.8, each point represents the median execution time across 10 repetitions of the corresponding combination of parameters.



(a) Route

(b) FindBestSpinupSize

(c) SimulateBatch

(d) FindGoodSpinupTime

Figure 3.8: Efficiency evaluation of key AutoSLO algorithms.

3.8.8.1 Query Router Efficiency

The Query Router is invoked in the critical path of each query, so it needs to be highly efficient. We assessed this by sweeping two variables: the number of active clusters and the number of running queries per cluster. As evident from Figure 3.8a, the Query Router achieves a decision-making time under 10 ms in most scenarios, with a maximum of only 35 ms.

3.8.8.2 Autoscaler Efficiency

The Autoscaler runs in a background thread, as described in Section 3.6. Still, it must be reasonably efficient for the scaling decision to remain timely. We assessed its efficiency by sweeping two variables: the number of candidate cluster sizes and the arrival rate of queries in the Arrival Window. As seen in Figure 3.8b, its decision-making time is around 7 s even in the most challenging case, providing fast responsiveness for reactive scaling.

3.8.8.3 Policy Tuner Efficiency

The Policy Tuner is invoked periodically, as described in Section 3.7, so its overhead can be somewhat higher. Still, it must not consume excessive resources that could otherwise be used for workload execution. Aside from bookkeeping, two expensive functions are used by the Spinup Scheduler and the Configuration Tuner: SimulateBatch and FindGoodSpinupTime.

For SimulateBatch, we swept two variables: the number of queries per simulated workload and the number of workloads simulated in parallel. As seen in Figure 3.8c, batch simulation parallelizes well and finishes in around 2 min even with over 15,000 queries, well within the requirements of an infrequent, background Policy Tuner invocation. For FindGoodSpinupTime, we swept the number of queries in each forecasted workload and the number of forecasts considered. As Figure 3.8d shows, the algorithm completes in slightly over 2 s in the worst case.

Finding from 3.8.8: AutoSLO Efficiency

Each critical algorithm of AutoSLO is efficient, meeting the latency requirements of its operating timescale.

3.9 Related Work

We will now dive deeper into related work on maintaining SLOs, guided by Table 3.6.

3.9.1 SLOs for Cloud Data Systems

As shown in the top part of Table 3.6, no prior work on cloud data systems offers SLOs and exhibits all three **key desired behaviors**.

Only offline planning. Some works **plan** for the future workload without adjusting resources online or reacting at routing time. ResTune [261] reduces resource utilization by replaying the workload on a replica and tuning knobs using constrained optimization. WiseDB [140] trains a decision model on pre-selected query templates to derive fixed template-to-cluster routing decisions. These works expend significant effort to derive optimized configurations, but cannot adapt if the observed workload deviates from what they optimized for.

Only online adjustment. On the flip side, some works only **adjust** resources based on online load, neither reacting to this load at routing time nor planning future resource allocation. Das et al. [55] use an adapted version of the token bucket algorithm [213] to balance the estimated resource demand of each customer. SLAOrchestrator [165, 166, 167] suggests and enforces per-query latency SLOs over workloads of ad hoc SELECT-PROJECT-JOIN (SPJ) queries, by having users select their desired cost/performance tier among provided options. Most recently, BRAD [249] focuses on how to select, configure, and route among *different engines*. A number of additional works focus on *vendor-side* optimization, packing customer workloads onto machines [20, 51, 122, 133, 155, 156, 212, 239], unlike our focus on optimizing the requirements of each and every workload. Earlier works have also evaluated alternatives like having the user fully define the autoscaling triggers and exact corrective actions [186, 263] or using reinforcement learning [141].

Table 3.6: Coverage of related work.

Section 3.9.1: SLOs for Cloud Data Systems

Category	Has SLOs	Plans offline	Adjusts online	Reacts when routing
Only offline planning [140, 261]	✓	✓	✗	✗
Only online adjustment [20, 51, 55, 118, 122, 133] [141, 155, 156, 165, 166, 167] [186, 212, 239, 249, 263]	✓	✗	✓	✗
No explicit SLOs [157, 189, 234]	✗	✓	✓	✗
Fixed, no interference [43, 44, 45, 125, 240]	✓	✗	✗	✗
Fixed, no SLOs [7, 101, 180, 184, 235]	✗	✗	✗	✓

Section 3.9.2: SLOs for Other Cloud Systems

Domain	Unlike queries the scheduled tasks have...
Serverless Computing [3, 71, 110]	Opaque, function-level behavior
Application Placement [39, 59, 60, 134, 170, 181]	Longer lifetimes, SLOs over inbound requests
ML Model Serving [41, 116, 151]	Similar computational demands
LLM Serving [37, 96, 97, 119, 126, 247, 268]	Regular structure, only prefix-based caching

No explicit SLOs. The currently published mechanism of Amazon Redshift, as introduced in Auto-WLM [189] and refined in RAIS [157], balances **adjusting** and **planning** but does not optimize for explicit SLOs, nor does it **react** to observed load through concurrency-aware predictions. Instead, autoscaling is triggered by query queue length, with a qualitative price-performance sensitivity slider controlling the aggressiveness. Query routing relies on concurrency-unaware latency predictions, as described in Stage [234].

Fixed resources. A final class of systems assumes a fixed amount of compute per customer. Some support SLOs but do not **react** to query interactions when routing [43, 44, 45, 125, 240]; others **react** to query-level interactions but do not optimize for SLOs [7, 101, 180, 184, 235].

3.9.2 SLOs for Other Cloud Systems

Per the bottom part of Table 3.6, related work on SLOs for other cloud systems can also be instructive, although not directly applicable.

Serverless Computing. One area of focus is scheduling tasks in serverless function-as-a-service offerings (e.g. AWS Lambda [16]). In this domain, re-executing the *same* function on warm resources can be faster, but there is no notion of a cross-function "cache". This is unlike databases, where the same cached data can be useful for many different queries. Nevertheless, some concerns are similar. Hermod [110] balances performance and resource efficiency through an execution-time-agnostic but cost- and load-aware hybrid policy, based on simulation-derived insights. FaaSCache [71] casts function keep-alive as a caching problem, reducing cold-start overhead using a variant of the Greedy-Dual [42] policy. Palette [3] tags each function invocation with a user-provided locality hint (*color*); scheduling then maintains a map from colors to instances. Such works prize generality and treat the computational demands of each function as opaque. However, when dealing with database queries, latency prediction is more tractable and a critical source of scheduling-relevant information.

Application Placement. Works like Paragon [59], Quasar [60], and Mage [181] optimize application placement (called *scheduling* in this literature) in datacenters with heterogeneous platforms and resources. They treat resource allocation and assignment as a recommendation problem, using limited profiling to balance hardware utilization and the desired performance. Works like Heracles [134], PARTIES [39], and CLITE [170] explore co-locating such latency-sensitive applications on the same resources as "background", throughput-focused work. In this context, SLOs are defined for *user requests*, while the scheduled entity is the *long-running application*. This is different from routing each individual SLO-bound database query.

ML Model Serving. Another line of work studies resource allocation for cloud-deployed ML models subject to inference latency SLOs. For example, Mendoza et al. [151] discuss an interference-aware model co-location policy, based on a random forest regressor trained on offline model/hardware profiling. Mudi [41] further attempts to co-locate inference tasks with model training while minimizing interference, by estimating inference latency as a piecewise linear function of allocated resources. CascadeServe [116] uses model output quality as an additional optimization knob, optimizing the design and placement of *model cascades* that invoke higher-latency inference for "harder" requests only. These systems exploit the relative uniformity of inference requests to a given model, whereas analytical queries can differ substantially in operators, data access patterns, and resource demands.

LLM Serving. LLM serving introduces more per-request variability and state management (e.g. KV-cache entries) compared to traditional model serving. Several works deal with these issues for individual requests: Orca [247] proposes dealing with variable-length decode phases using iteration-level scheduling and selective batching. PagedAttention [119] uses OS-inspired techniques to reduce KV-cache fragmentation and improve throughput. DistServe [268] and TetriInfer [97] explore prefill-decode disaggregation, so that the resource allocation for each phase can be separately optimized. SOLA [96] seeks to balance the different SLOs of prefill and decode through state-aware scheduling. HotPrefix [126] addresses the increasing prevalence of requests with the same prefix (e.g. system prompt) by tracking and leveraging the hotness of each prefix while managing the KV-cache. More recent works look at optimizing

compound AI systems built around LLMs. Murakkab [37] proposes a declarative abstraction that enables under-the-hood efficiency-driven optimizations. However, LLM requests still have a comparatively regular structure (centered on prefill/decode) and only prefix-based caching. Database queries have more diverse plans and richer forms of data reuse, leading to a much larger decision space.

3.10 Pitfalls and Lessons Learned

As in Section 2.10, we will now discuss four pitfalls faced during the conception and development of AutoSLO and the corresponding lessons learned, which can inform the design of future systems. For each lesson, we will give appropriate context, discuss our initial approach, present our observations about its behavior, and conclude with our revised approach.

3.10.1 Using Poisson Arrivals in the Workload Forecaster

3.10.1.1 Context

The Spinup Scheduler aims to schedule promising cluster spinups, which can reduce AutoSLO's online reaction time to workload peaks in the following period (day) and ensure uninterrupted SLO adherence. This requires *some* forecast of what the workload will look like in the day ahead, which is the responsibility of the Workload Forecaster. In particular, to schedule cluster spinups in a robust manner, we would ideally need *multiple* such forecasts, derived in a diverse manner; this would reduce overfitting to the specific details of each forecasted workload. Generating accurate database workload forecasts can be very challenging, as evidenced by the existence of an entire subfield of study [61, 74, 83, 93, 99, 137].

However, unlike other applications of workload forecasting, our use of forecasts has a much higher error tolerance. All the Spinup Scheduler does is accelerate responsiveness by making additional capacity available ahead of time, a very coarse-grained decision; the Query Router and Autoscaler remain capable of adjusting resources and routing to meet the SLO based on the actual observed workload. Therefore, even if the workload forecast is somewhat inaccurate (e.g. in terms of the exact timing of some queries), the online components are capable of handling the real load as it appears. All the forecasts need to do is capture coarse-grained workload trends to drive pre-provisioning decisions.

3.10.1.2 Initial Approach

Throughout this project, we assume that the only information available to the Workload Forecaster is the past workload faced by AutoSLO. In practice, additional information may be readily available and highly useful (e.g. a list of pre-scheduled recurring ETL jobs), but we make this looser assumption to broaden the applicability of AutoSLO. Therefore, a tradeoff immediately emerges: how much of the past workload history, and in what form, should AutoSLO retain (in the Workload Reservoir) to support workload forecasting? A more detailed history could support more elaborate forecasting techniques, but it would raise the storage footprint and the risk of overfitting.

Given the relative error tolerance of this domain, our initial approach focused on just storing and sampling query texts. Compared to the description in Sections 3.7.1–3.7.2, this initial approach lacked the *arrivals table*, retaining no intra-hour timing information about query arrivals. Instead, the downstream Workload Forecaster sampled an appropriate *total number* of queries for each hour (based on the day of the week), using recency weighting like the one described in Section 3.7.2. It then simply issued the queries within each hour according to a Poisson process with the specified total number of arrivals.

3.10.1.3 Observations

Even though our initial approach generally matched the coarse-grained intensity of the workload, as designed, we found that it was not sufficient to support the Spinup Scheduler in its decision-making. This became apparent by comparing the scheduled spinups determined using forecasted workloads, to the scheduled spinups that the Spinup Scheduler would have chosen if given oracle advance access to the true workload for a target day: oracle access to the true workload led to more aggressive scheduled spinups, with more clusters of larger sizes.

By diving deeper into the choices of the Spinup Scheduler, we quickly discovered the cause of this discrepancy: our Poisson-process-based arrivals model did not capture bursty load *spikes* in the workload, which were responsible for most of the SLO violations. Only capturing the overall workload pressure within each hour was not sufficient to recreate the high-contention regimes of the real workload that posed the highest SLO violation risk.

3.10.1.4 Revised Approach

In response to our observations, we revised our workload forecasting approach to the version currently described in Sections 3.7.1–3.7.2. In particular, introducing the *arrivals table* as a source of representative interarrival times ensured that our forecasted workloads included realistic bursts, matching the real workload patterns. This selective increase in the fidelity of the forecast helped the Spinup Scheduler make better decisions, contributing to the overall high performance of the Policy Tuner, as presented in Section 3.8.7.

Lesson 3.1

Even though our domain tolerates coarser workload forecasts, preserving workload intensity spikes is necessary to correctly assess SLO risk.

3.10.2 Overpredicting Latency to Conservatively Meet SLOs

3.10.2.1 Context

Database query latency prediction is a highly challenging task, as evidenced by the wealth of related work [6, 8, 67, 94, 142, 232, 234, 235, 249, 264, 271]. For SLO-aware Multi-Cluster Workload Management (Problem 3.3) in particular, we are highly sensitive to the *absolute* quality of predictions, because we plan to compare them to the values of the SLOs.

This is not always the case. For example, consider the domain in which Iconq [235] was proposed: single-cluster query scheduling. There, latency predictions were used to help

determine whether it is better to submit a query q for execution *now*, or to queue it and wait. To support this decision, the authors compared the predicted latency of q in each of these two scenarios. In this domain, all that matters is getting the *directionality* of the comparison right, so that you are led to the right decision. It does not matter if both predictions are off by a factor of 10, as long as their total order is the same. The same is true in every domain where query latency predictions are only compared to each other; for example, when comparing the predicted performance of two possible plans during query optimization. That is why query optimizer "costs" can be in arbitrary units.

In our domain, however, we would like to use latency predictions to help the Query Router decide whether submitting query q to cluster c will meet q's SLO. This determination is sensitive to the absolute error in each latency prediction, since *there is now a concrete threshold to compare latency to*; it is not enough to get the directionality of inter-prediction comparisons correctly. However, tweaking the model to "focus on reducing absolute error" is non-trivial. Query latencies span several orders of magnitude, so penalizing absolute error during training can easily yield a model that just focuses on predicting the long-running queries correctly, in order to shave off the largest amount of absolute error seconds.

3.10.2.2 Initial Approach

Focusing on properly supporting SLO-aware query routing, our initial approach to dealing with the "threshold comparison" problem was based on the following observation: *the two directions of prediction error are not made equal*. In particular, according to Problem 3.3, meeting the SLO is our primary consideration for each query, with cost concerns only coming into the picture later. This means that overpredicting the latency implied by a particular cluster/neighbor environment is preferable to underpredicting it. Overprediction will simply force us to prefer a less loaded/larger cluster, increasing cost but improving SLO adherence. However, underprediction could lead to overly optimistic query routing and SLO violations.

Based on this reasoning, our initial approach modified the training of the Latency Predictor to target a high percentile (e.g. p90) of the latency distribution. That is, instead of rewarding predictions that were correct *on average*, we rewarded predictions that were *higher than the ground truth most of the time*. We hypothesized that such a model would lead to more conservative routing decisions than a perfect model, but that it would avoid SLO violations.

3.10.2.3 Observations

Indeed, for the purposes of query routing, a model trained with a quantile-based objective performed as expected, producing conservative routing decisions with low SLO violation rates. However, the picture soon became more complicated, when attempting to use the same model in other components of AutoSLO, most importantly the Workload Simulator.

The key obstacle has to do with the input featurization of Iconq (Section 3.4.1): an interaction feature vector for each of the *neighbors* of target query q. In an online routing setting, Algorithm 3.1 can exactly observe the neighbor sets of each query it needs to predict the latency of: for the incoming query, each running query is a neighbor; for each running query, we know its past neighbors and we add the incoming query. However, when using the Latency Predictor to drive the Workload Simulator by treating predicted latencies as true, we

introduce a feedback loop: mispredicting the latency of q_1 affects whether it overlaps with q_2, altering the neighbor set of q_2; now the inputs to the latency prediction of q_2 are different, changing the prediction itself; this in turn affects whether q_2 overlaps with q_3; and so on.

Clearly, this problem can get much worse if we introduce a systematic latency overprediction bias. Now, every query seems to run for longer, collecting more neighbors (with later arrival times) and appearing to be itself a neighbor of more future queries. This feedback loop can completely derail the fidelity of the simulation and the decision-making of the components that use it (the Spinup Size Selector and the Batch Simulator).

3.10.2.4 Revised Approach

Based on the observations above, we concluded that a quantile-based objective was unsuitable for training our Latency Predictor. Instead, we would have to find alternative ways to reduce the absolute error of our predictions, while keeping *all* of the use cases of the Latency Predictor in mind. These considerations were important factors in introducing the censored observation handling detailed in Section 3.4.3, so as to increase the signal obtained from our training data and improve model predictions in environments of partial neighbor observability.

Lesson 3.2

When reusing models across the system, we need to carefully consider how the failure modes of each component are different.

3.10.3 Basing the Spinup Size Selector on Counterfactual Simulation

3.10.3.1 Context

The Spinup Size Selector helps AutoSLO **adjust** the active cluster set in response to recent workload performance, as described in Section 3.3. At a high level, in order to recommend a size for the future cluster, the Spinup Size Selector needs to compare the predicted performance of different cluster size options. When collecting data for this comparison, there is a tradeoff to manage: on the one hand, using the Workload Simulator to assess the impact of the new cluster over a longer period can provide a more accurate picture; on the other hand, it can slow down decision-making and allow forecasting and latency prediction errors to compound.

3.10.3.2 Initial Approach

We initially designed the Spinup Size Selector to invoke the Workload Simulator in a *counterfactual* manner: it virtually "rewound" workload execution to the start of the current Arrival Window (i.e. w seconds ago, see Section 3.6.1), posited the existence of an additional cluster of a given size at that point in time, and then simulated forward from that point until the present. After performing this simulation for each candidate size for the new cluster, it compared their achieved SLO violation rates and costs and selected the best one according to BestWithTarget. The operation of this design roughly corresponded to the following principle: *we are in a bad position now; what would have been the best action w seconds ago to avoid reaching this point?*

This approach provided two important benefits: (i) different scaling actions were compared using actual recent query arrivals, accurately reflecting recent workload trends; and (ii) the length of the simulation for each cluster size could be precisely controlled through w and was the same across candidate sizes, offering stable simulation performance.

3.10.3.3 Observations

When evaluating the initial design of the Spinup Size Selector in earlier versions of the experiment of Section 3.8.5, we observed a concerning outcome: once new clusters were spun up, they gave rise to an oscillatory effect that prevented AutoSLO from converging to a steady SLO-compliant state.

In particular, when the Spinup Size Selector is invoked, the Spinup Trigger must have fired, which implies that AutoSLO is already observing an elevated SLO violation rate. Since any new cluster requires time δ to spin up and be added to the active cluster set (see Section 3.2.1), additional load is likely to have accumulated in the system by that time. The new cluster will then be called upon to help relieve this backlog. More precisely, once the new cluster joins the active cluster set, it will initially appear as a very appealing routing target for the Query Router, since existing clusters will have been burdened by dealing with the accumulating query backlog. The new cluster is therefore likely to receive a significant fraction of arriving queries right after it becomes available.

However, when selecting the size of the new cluster through counterfactual simulation, there was no accounting for such an accumulated query backlog; instead, the size was chosen based on the *preventative* effect it would have had if available in the past. Because of this, we observed that the chosen cluster sizes were generally too small to successfully absorb the load placed on them right after they became available. Instead, this load soon overwhelmed the new cluster, making the pre-existing clusters (partially caught up with their own load in the meantime) attractive routing targets in turn. This cycle continued while the system traffic remained elevated, causing a high SLO violation rate in the process.

3.10.3.4 Revised Approach

Despite the benefits of our initial approach, the oscillatory effect described above led us to reconsider the Spinup Size Selector design and eventually arrive at the current approach, as described in Section 3.6.3. Instead of taking a counterfactual view, this approach explicitly simulates the future development of the query load in the system, capturing the presence and effect of a possible query backlog. In essence, it asks the relevant question directly: *we are in a bad position now; how do we best get out of it?*

This new design has two drawbacks, each the flip side of one of the benefits described in Section 3.10.3.2. First, assuming that the Arrival Window will keep repeating identically is unlikely to be a completely accurate short-term forecast. However, in practice it serves as a satisfactory approximation and encourages the Spinup Size Selector to pick more appropriate cluster sizes. Second, the length of the simulation is now undefined; how far into the future should we simulate? This conundrum is addressed by introducing the parameter μ, as discussed in Section 3.6.3, leading to a practical revised design.

Lesson 3.3

Reactive decisions need to account for their own trigger, not just steady state.

3.10.4 Collecting Training Data Using Uniform Workloads

3.10.4.1 Context

In order to train our Latency Predictor, we needed to collect training data that provides sufficient coverage across the different domains the model may encounter in practice. In particular, it was important to provide the model with observations of different cluster sizes, different query features, and different mixes of concurrent neighboring queries.

3.10.4.2 Initial Approach

For training data collection purposes, we constructed the BaseWorkload(λ) described in Section 3.8.1.2 by simply shuffling the TPC-DS queries and issuing them according to a Poisson arrival process with a variable rate λ (in queries per second). This approach provides a family of workloads that is well-aligned with existing benchmarks familiar to our community, while providing enough flexibility to enable diverse workloads for model training.

Alongside the ability of λ to control the arrival rate of workload queries, we were also interested in generating workloads with varying query mixes. We achieved this by constraining the variety of TPC-DS query templates, as well as the variety of unique query texts per template, present in each workload.

Based on the considerations above, we generated eight training workloads and executed each of them on Amazon Redshift Serverless workgroups of four sizes: 4 RPU, 8 RPU, 16 RPU, and 32 RPU. The eight workloads were derived as a cross product of three parameters, each with two options: (i) the number of TPC-DS templates $N_{\text{templates}}$ in the workload (66 or 99); (ii) the number of distinct query texts N_{texts} per TPC-DS template (2 or 3); and (iii) the arrival rate parameter λ (0.1 or 0.05). We then derived training, validation, and testing splits from these 32 workload runs, on which to train and evaluate our Latency Predictor.

3.10.4.3 Observations

Even though our initial workloads nominally provided good coverage of the design space, we observed certain regimes of low performance for the resulting Latency Predictor, which we eventually traced back to the patterns of query load observed during training.

First, we observed a correlation between the cluster size and the distribution of the number of neighbors of each query. Smaller-sized clusters were generally underpowered for the training workloads, leading to significant contention during query execution and ultimately yielding training data points where each query was more likely to have a large number of neighbors. For large clusters, we observed the opposite extreme: queries generally completed faster, so that most training data points reflected queries with only a handful of neighbors. This correlation meant that the Latency Predictor was under-trained on instances of low contention on small clusters and high contention on large clusters, both of which are valuable regimes during its regular operation as part of the Query Router and Workload Simulator.

Second, due to the regular pattern of arrivals in each of our training workloads, there were limited opportunities to observe cluster behavior under sudden workload peaks or "drain" regimes, where backlogged queries gradually complete and free up resources during a period of lower arrival intensity. Since arrivals in each workload were only governed by a single Poisson process, there were two predominant behaviors: either a given cluster size was "large enough" for a given workload, whereby queries completed in a timely fashion and the system saw no real instances of high contention; or it was not large enough, which led to the cluster being increasingly overwhelmed as more queries started piling up.

3.10.4.4 Revised Approach

In response to these observations, we extended our training set in two ways. First, we collected training runs using larger values of λ, namely $\lambda \in \{0.2, 0.5, 1.0\}$, albeit for a fixed combination of $N_{\text{templates}} = 99$ and $N_{\text{texts}} = 3$. These workloads were successful in providing training examples with higher neighbor counts for large cluster sizes, allowing the Latency Predictor to learn the behavior of large clusters in this regime. For smaller cluster sizes, these runs led to even worse contention and overall slowdown; however, thanks to our support for censored observations (see Section 3.4.3), these training runs were still able to provide a wealth of training examples.

Second, we collected data using "phased" versions of the BaseWorkload, with the parameters discussed in Section 3.8.3.2. Near phase boundaries, these workloads were able to provide training examples that show system behavior under unexpected peaks and lulls, improving the accuracy of the Latency Predictor as shown in Figure 3.7a.

Lesson 3.4

Training data collection can introduce confounding or selection biases; it is important to mitigate them to ensure high accuracy across the board.

3.11 Conclusion

We presented AutoSLO, a framework for cost-efficiently meeting latency SLOs on a multi-cluster cloud data warehouse, consisting of a proactive Policy Tuner that **plans** infrastructure scaling, a reactive Autoscaler that **adjusts** resources based on workload fluctuations, and an online Query Router that **reacts** to concurrent load. We showed that these components can work together to successfully meet latency SLOs of varying strictness, reducing cost by a mean of 26.4% compared to the per-scenario next-best baseline on realistic Redbench workloads. Component-level evaluations showed that the Query Router and Autoscaler respectively reduce SLO violation rates by a mean of 47.8% and 93.7%, relative to their corresponding alternatives. Finally, we showed that the Policy Tuner can reduce the SLO violation rate by a mean of 44.6% using a single day of workload history, and that each component is efficient given its intended operating timescale.

Chapter 4

Synthesis and Research Outlook

Chapter 1 introduced the **abstraction gap** and three principles for traversing it. In Chapters 2 and 3, we leveraged these principles to design and implement systems bridging this gap in opposite directions. In this chapter, we will first demonstrate how diagnosis and management can interact through a case study connecting LOGos and AutoSLO. We will then situate this thesis within concurrent trends and propose future research directions.

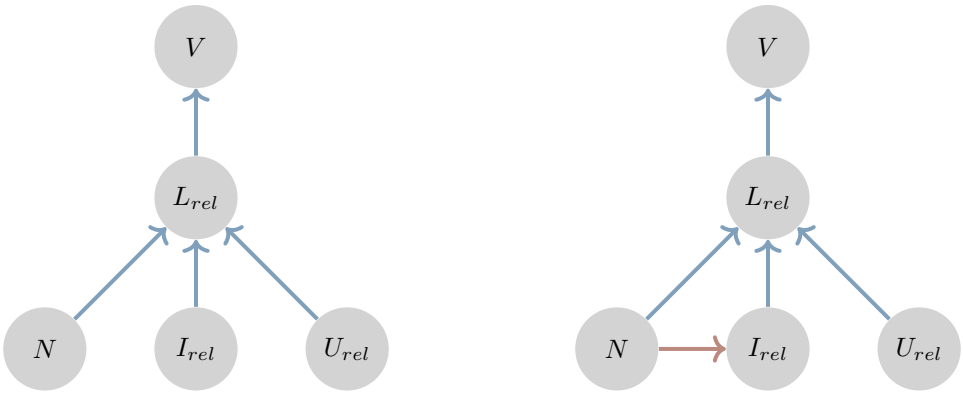
4.1 Case Study: Diagnosing AutoSLO with LOGos

We begin by exploring one way of bridging LOGos and AutoSLO by using the former to diagnose the behavior of the latter, examining whether operational evidence produced during performance management can support a useful causal investigation. In particular, we apply LOGos to the logs generated by AutoSLO during the April 15, $\kappa = 4$ experiment from Figure 3.6c, with the goal of investigating the remaining observed SLO violations.

The resulting log contained 17,399 records spanning 1,356 completed queries. The recorded events included query arrivals and completions; query-routing decisions, including the selected cluster size and the number of queries running at routing time; and latency predictions generated when a query was routed or when a new neighbor joined its cluster during execution. We augmented these records with each query's observed latency, applicable SLO, and query-plan features, and treated each query as one causal unit. We then applied the interactive components of LOGos to investigate the causes of the observed SLO violations.

4.1.1 Causal Investigation

What causes SLO violations? We began from the high-level outcome variable experienced by an AutoSLO user: whether each query q violated its SLO. We call this variable V. The Candidate Cause Ranker suggested two candidate causes of V based on log data: a query's absolute observed latency (L), and its observed latency relative to its SLO (L_{rel}). Both of these are correctly identified as plausible causes: L may cause V since, all other things being equal, a longer query is probably more likely to violate its SLO; and L_{rel} causes V by definition, since a query violates its SLO if and only if $L_{rel} > 1$.



(a) Causal graph before refinement. (b) Causal graph after refinement.

Figure 4.1: When applied to logs from AutoSLO, the Interactive Causal Graph Refiner solicits a judgment that exposes the dual influence of N on the observed SLO-normalized latency L_{rel}.

Faced with this ranking, we initialized our causal graph with the edge $L_{rel} \to V$. While this is an unsurprising causal relationship, it allows our investigation to move from a binary symptom to the continuous performance quantity that underlies it. This continuous quantity is likely to be better quantifiable based on other observed variables.

What causes high SLO-normalized latency? We then continued our investigation by invoking the Candidate Cause Ranker on L_{rel} in turn. This time, LOGos was able to surface a more interesting set of candidate causes. Among the top candidates it exposed three variables that we know are indeed causes of L_{rel} based on our knowledge of the architecture of AutoSLO: (i) N, the number of queries that were already running on the cluster c that q was routed to when q arrived; (ii) I_{rel}, the initial prediction of q's latency at the time of its routing, relative to q's SLO; and (iii) U_{rel}, the latest update to q's predicted latency relative to its SLO, produced when q' was routed to c, where q' is the neighbor of q that arrived last.

Understanding how N may influence L_{rel} is easy: already-running queries will contend with q for cluster resources, so they can clearly affect L_{rel}. However, assessing I_{rel} and U_{rel} as candidate causes requires careful interpretation. At first glance, latency predictions (whether initial or updated) do not physically influence the execution of a query q, so we may be tempted to discredit them as causes of its latency. What latency predictions do influence, though, are *routing decisions*: the initial prediction I_{rel} was used by the Query Router to select c as the cluster to execute q on, while the updated prediction U_{rel} was used by the Query Router to select c as the cluster to execute q' on, as part of assessing the SLO-violation risk posed to q by q'. Both of these decisions affect the amount of resources available for q's execution and its ultimate observed latency.

Based on this reasoning, as shown in Figure 4.1a, we added three new edges to our emerging causal graph: $N \to L_{rel}$, $I_{rel} \to L_{rel}$, and $U_{rel} \to L_{rel}$. Note that the edges $I_{rel} \to L_{rel}$ and $U_{rel} \to L_{rel}$ are meant as compact representations of the routing-policy-mediated relationships described above. A more detailed causal graph could explicitly represent the features used by the Latency Predictor, the routing decision, and the selected execution environment as intermediate variables. However, since such variables are not explicitly logged, LOGos successfully identified the latency predictions associated with q as proxies for the routing

decisions that involved q directly or indirectly. This reflects a common situation in real-world logs, where the quantity of interest may be missing or hard to explicitly include in a log, but identifying informative proxies can still be valuable for moving an investigation forward.

Initial ATE estimate. Under the initial graph of Figure 4.1a, LOGos estimated that a unit increase in I_{rel} implies an increase in L_{rel} of approximately 0.65 units. While the exact value of this estimate may only offer limited actionable paths forward, it shows three things:

1. **The value is positive**, meaning that a higher initial SLO-normalized latency prediction corresponds on average to a higher observed SLO-normalized latency; this confirms that our initial latency predictions are directionally consistent with observed latencies.

2. **The value is below 1**, meaning that there are mitigating factors at play influencing the final observed latency. Based on our knowledge of the architecture of AutoSLO, this makes sense: a higher initial latency prediction for q may discourage future queries from being routed to the same cluster c, which can ultimately translate to more resource availability for q and a more timely completion.

3. The value itself **serves as a baseline** for future comparisons, as we will see below.

Refining the causal graph. Having obtained reasonable candidate causes using the Candidate Cause Ranker, we next moved to the Interactive Causal Graph Refiner. In particular, we invoked it to determine causal graph refinements that could have a large impact on $ATE(I_{rel}, L_{rel})$, which we discussed above.

In response to our request, the Interactive Causal Graph Refiner solicited our judgment with respect to the following causal graph edge: $N \rightarrow I_{rel}$. Such an edge, if present, would imply that the number of queries that are active on c at the time of q's routing influences the initial prediction of q's latency. Based on our knowledge of the Latency Predictor, this edge should indeed be present: Iconq+ bases its latency predictions in part on the neighbors of the target query, which are used to form the interaction feature vectors. As such, we accepted the addition of this edge, leading to the graph in Figure 4.1b.

Note that adding $N \rightarrow I_{rel}$ has now revealed N as a common cause of both I_{rel} and L_{rel}, confounding the effect of the former on the latter. This is indeed a relationship that makes sense within AutoSLO: more neighbors at routing time not only imply direct competition for resources affecting observed latency, but also influence the latency *prediction*, which informs subsequent routing decisions and the additional resource contention they may introduce.

Updated ATE estimate. Indeed, after moving to the graph in Figure 4.1b, LOGos updated its estimate of $ATE(I_{rel}, L_{rel})$ from 0.65 to 2.44. That is, if we adjust for the number of queries running on c when q arrives, the initial SLO-normalized latency prediction for q affects its observed latency much more, indicating that routing-time concurrency was suppressing this relationship in the original graph. This is in line with our earlier discussion of the influence of the initial prediction on subsequent routing decisions. Higher initial contention may have been causing the Query Router to avoid c, partially offsetting the risk to q's SLO compliance; once we adjust for initial contention, we remove some of this "protective" effect, unearthing a stronger relationship between predicted and observed SLO-normalized latency.

4.1.2 Discussion

This case study provides a qualitative proof of concept for connecting the two systems. Starting from an application-level outcome of interest (i.e. SLO violations), LOGos correctly surfaced relationships involving the Latency Predictor, the Query Router, and the concurrency experienced by incoming queries. In particular, it exposed how routing-time load can affect observed latency both directly, through resource contention, and indirectly, by influencing latency predictions and subsequent routing decisions. Accounting for this dual role materially changed the estimated effect of the initial SLO-normalized latency prediction (I_{rel}) on the observed SLO-normalized latency (L_{rel}).

This case study also illustrates how the three design principles of Chapter 1 can apply across the boundary between performance management and diagnosis, rather than only within each project. First, according to Principle 1, our high-level diagnostic intent scoped the model: focusing on SLO violations allowed us to construct only a small relevant part of the overall causal graph, covering only the relevant mechanics of AutoSLO. Second, according to Principle 2, our feedback was solicited and integrated to refine this partial model; we used our architectural knowledge of AutoSLO to confirm the relationship between the number of running queries at routing time and the initial latency prediction, adding an edge to the causal graph. Third, according to Principle 3, the Interactive Causal Graph Refiner prioritized this particular judgment because it was expected to materially affect the ATE, reflecting the key fact that AutoSLO's Latency Predictor is *concurrency-aware*.

This limited investigation does not constitute a definitive account of all remaining SLO violations in AutoSLO, nor does it necessarily establish that initial prediction error is their sole or primary cause. However, it does motivate further investigation into the sensitivity of the Latency Predictor's performance to concurrent load. Since the Query Router relies on these predictions to make routing decisions, systematic errors under particular concurrency regimes could help explain why some violations remain, as we also saw in Section 3.10.4.

4.2 Concurrent Developments and Trends

Section 4.1 outlined one concrete path forward based on the contributions of this thesis: a closed diagnostic and management loop, where the management system records its observations, predictions, and decisions in logs; the diagnostic system uses this evidence to identify relationships that may explain undesirable high-level outcomes; and the resulting insights guide the development of the next generation of the management system. Realizing this loop requires answering broader questions about what evidence the management system must preserve, how its models should be refined as system state evolves, and how the impact of possible refinements and actions can be estimated reliably.

Before we explore such questions in more detail in Section 4.3, we now briefly discuss relevant trends from academia and industry that were observed concurrently with the development of this thesis: expanding the purview of performance diagnosis, raising the abstraction level of performance management, and increasingly tapping AI-driven reasoning for diagnosis and management alike. Understanding these trends is crucial for situating the contributions of this thesis within the broader literature and building on them effectively.

4.2.1 Expanding the Purview of Performance Diagnosis

An extensive line of work has shared the goal of our work on LOGos, focusing on automatically extracting insights from the telemetry of complex systems. Recent advances in this domain have generally focused on one of two things: expanding the *coverage* of diagnostic systems by allowing them to reason over more information, or enhancing the *robustness* of these systems so that they can still extract insights despite this expanded input space.

In the former category, TracEx [68] supports analyzing and comparing detailed database execution traces through common abstractions and a corresponding user interface, unlocking cross-domain workload understanding and comparison. RCRank [168] instead explicitly focuses on supporting multimodal operational evidence (e.g. CPU readings and query plans) to rank candidate causes for slow queries. Finally, DBDoctor [228] expands *what* gets monitored in the first place, leveraging eBPF (Extended Berkeley Packet Filter) technology to collect event-driven rather than sampling-driven metric observations.

In the latter category, BALANCE [38] focuses on root cause analysis from operational time-series traces, using a Bayesian multicollinear feature selection (BMFS) model to promote sparsity in candidate cause selection (similar to how the Candidate Cause Ranker in LOGos uses LASSO to also promote sparsity). Yan et al. [242] propose a Siamese Discrepancy Network (SDN) that can more accurately identify anomalous metrics, even if anomalous events are rare in training data, precluding the training of other model classes. RT-Log [108] focuses on improving the robustness and transferability of log anomaly detection by learning environment-invariant representations.

Our diagnostic system, LOGos, also reflects these two concurrent trends. On the *coverage* side, we explicitly focus on the values of log-derived *variables* and their aggregates, rather than the coarser focus of earlier work on log *templates*. On the *robustness* side, we build our framework around causal inference techniques, improving quantitative diagnosis by enabling principled reasoning about the complex causal relationships present in a modern system. At the same time, the continued expansion of diagnostic evidence makes the scoping problem underlying Principle 1 increasingly important: broader coverage can improve diagnosis, but attending to every available signal can also introduce noise and end up exposing the complexity that a higher-level diagnostic interface is intended to hide.

4.2.2 Abstracting Away Performance Management

A parallel thread of concurrent work has been underway on the performance management side. Again, we can broadly understand these advances through two (non-exclusive) trends: an *abstraction* trend where an increasing share of configuration and management responsibility shifts to vendors, and a coupled *usability* trend that aims to align system interfaces and control mechanisms with the infrastructure realities and performance needs of real users. Put another way, the former trend makes it possible for vendors to offer automated objective-oriented resource management; the latter makes it possible for users to specify these objectives.

As part of the abstraction trend, cloud database products have continued to move towards a "serverless" paradigm, unlocking new opportunities for efficient resource sharing. As an illustrative example, Amazon Aurora redesigned managed PostgreSQL/MySQL for the cloud in 2015 [27] and had already introduced autoscaling as "Aurora Serverless" in 2018–

2019 [26, 102, 178]. However, in 2025, Amazon moved even further towards automated resource management by releasing the more flexible resource-disaggregated Aurora DSQL [12, 32]. Academic work over the same period has contributed even higher-level abstractions that could enable more under-the-hood optimization, such as Virtual Database Engines (VDBEs) [250].

As part of the usability trend, the focus has been on specifying, modeling, and enacting the users' desired cost-performance tradeoffs. Zhang et al. [257] argued for the importance of so-called "cost intelligence", identifying the challenges for its realization. Some works focus on coarse-grained SLO maintenance, such as BRAD [249], which optimizes for workload-level SLOs while cost-efficiently juggling multiple database engines. Other works take a more invasive approach, modifying the query engine internals. Tao [130] proposed decomposing SQL operators into tasklets (coroutine-based execution units of the physical plan), which are then scheduled in an SLO-compliant manner based on a token bucket model. Accordion [262] exposed intra-query parallelism as an adjustable knob *during* query execution, enabling real-time cost-performance decisions.

The development of AutoSLO to address the SLO-aware Multi-Cluster Workload Management problem (Problem 3.3) is integrally related to these two trends. This problem is only solvable because the *abstraction* trend has made resource disaggregation the norm, contributing the "multi-cluster" piece. It is also timely because the *usability* trend has highlighted the important role of SLOs, contributing the "SLO-aware" part. More generally, these trends show that narrowing the abstraction gap requires work on both sides of the interface: systems must expose a sufficiently rich space of actions, while users must be able to express their objectives at an appropriate level of abstraction, so that an automated management system can navigate that action space effectively.

4.2.3 Integrating AI-Driven Reasoning

System diagnosis and performance management have each given rise to rich bodies of literature. The rapid advent of large language models (LLMs) and the software layers built upon them, such as AI agents, has accelerated each of these research directions, blurring the distinction between diagnosis and management. Rather than dealing with targeted diagnostic or management tasks, these LLM-based systems contain a more general reasoning layer that can (to varying extents) consume heterogeneous operational information, invoke specialized tools, draw on internal or external knowledge, and propose subsequent actions.

The potential of LLMs for system diagnosis became evident very quickly. Large language models can not only provide a natural-language interface that shrinks the abstraction gap, but also act as powerful reasoning orchestrators that can quickly prune and combine large amounts of evidence. In this vein, Panda [200], D-Bot [270], and Andromeda [40] each proposed architectures for integrating publicly available system-related knowledge (e.g. from manuals or documentation) with scenario-specific signals (e.g. performance telemetry), in order to compose in-context root cause diagnoses for observed database system problems. More recent works have continued to refine this paradigm, optimizing knowledge retrieval and representation to improve accuracy and operational efficiency. For example, DBAIOps [269] structured operational knowledge as a knowledge graph, combining this representation with anomaly detection models and reasoning LLMs to construct diagnostic paths and generate root-cause analyses for observed issues.

As the reliability of such LLM-driven diagnostic tools improved, a parallel class of system management tools also emerged. While their architectural patterns are similar to those of diagnostic tools, these management tools do not help with a particular failure; instead, they generate recommendations from "steady-state" inputs such as workload and system descriptions. For example, GPTuner [123] used LLMs to construct a documentation-based search space for configuration tuning, developing a more conventional Bayesian optimization framework to prune and explore this space. Rather than tuning individual knobs, λ-Tune [77] instead cast the problem as efficiently generating a bounded set of complete configurations, which were then evaluated and compared on the actual workload. With the rise of agentic architectures, systems like AgentTune [127] experimented with decomposing the configuration task into multiple smaller tasks, such as workload analysis or search-space pruning, each assigned to a separate agent. Other works tackled the partially orthogonal problem of structured reasoning over even more diverse inputs; for example, SysInsight [260] argued for combining LLM reasoning with static analysis of the DBMS source code.

While all of these advances are undeniably exciting, they all underscore the same challenge: LLMs (and agents built upon them) provide a very powerful reasoning core, but bridging this core with the real world requires significant system building. The work of this thesis supplies elements that can be integrated into this emerging peri-LLM fabric. On the diagnostic side, LOGos offers an interactive mechanism for drawing causal conclusions from log data; while developed for expert human users, agents could also leverage the same mechanism to extract grounded signals from logs. On the performance management side, AutoSLO offers an efficient control loop for maintaining performance objectives; the same loop can become part of an AI-designed or AI-operated system, with agents selecting the performance objectives based on higher-level reasoning and communication with the end (human) users.

4.3 Future Research Directions

Informed by the work of this thesis, as well as the concurrent trends we presented in Section 4.2, we will now discuss four research questions that can help narrow the abstraction gap further. Each question can be explored through concrete future work building on this thesis.

4.3.1 Maintaining Representations for Online Diagnosis

As we saw in Section 4.2, automated system observability and fault diagnosis platforms are generally converging to the following philosophy: collect as much data as possible, and let AI-assisted reasoning prune and integrate it. Our first research question therefore emerges: *how can we best distill this continuous stream of incoming data into task-specific, evolving system representations that human or AI operators can use efficiently?* In order to leverage LOGos to answer this question, there are at least three possible research directions to explore.

Dealing with Selection Bias in Log Generation. As discussed in Section 2.10.1, log messages are generated using a "push" model based on exercising particular code paths or triggering certain events. This means that the presence or absence of particular log messages is itself a signal about the operational health of the system, which could inform a diagnosis. For example, under heavy load, the logging infrastructure may also suffer and provide degraded

observability into what is happening in the system, as evidenced by less frequent or delayed log messages. Going one step further, deciding whether and how to introduce additional logging infrastructure could help detect previously undocumented problems. In a world of "more is more" when it comes to operational data collection, harnessing the selection bias implied by *what* gets logged could be a valuable additional source of evidence.

Updating the Causal Graph Online. In Chapter 2, we focused on answering a specific causal question. However, for a production system, it is more realistic to assume a *stream* of questions as the system evolves. This requires two forms of online maintenance. First, for a given partial causal graph, one can develop techniques for efficiently ingesting new log data online. This can keep the expected quantitative effect of different interventions in line with the latest observations. Second, the *structure* of the causal graph may itself need to be updated based on the evolution of the system. For example, a new update may introduce or break causal relationships. New techniques could detect drift in estimates of key causal effects and notify the user (human or AI), so that they can potentially revise the causal graph.

Applying Causal Inference Over Multimodal Signals/Metrics. Finally, in the current iteration of LOGos, we based diagnoses solely on text logs and user judgments. However, there may be other information that could help paint a more accurate picture of both the slowly evolving system architecture and the instantaneous condition of a deployment. For example, one could rely on structured observability metrics, bug reports and associated tickets, internal documentation about the system, or even user discussion forums, as recent LLM-driven systems have done [40, 123, 270]. Integrating these capabilities with the causal machinery of LOGos could lead to better-informed decisions across the system.

4.3.2 Meeting Diverse Objectives

As discussed in Section 4.2.2, system builders have been raising the level of abstraction of their interfaces, allowing users to specify objectives in formats more closely aligned with their true operational goals, while reserving enough flexibility under the hood to manage resources in accordance with these objectives. AutoSLO demonstrated an effective system of this kind for a particular class of objectives, namely latency SLOs in cloud data warehouses. A second research question therefore naturally emerges: *how can we generalize the lessons of* AutoSLO *to help meet diverse user objectives?* Concretely, there are at least three paths forward.

Supporting Different SLO Types. In Chapter 3, we focused on per-query latency SLOs, possibly derived from coarser query collections (e.g. a particular workload). However, major organizations may also use different types of SLOs. Some workloads may consist of recurring queries that need to be completed by a certain deadline, like nightly ETL jobs. Another class of workloads may instead be cost- rather than latency-sensitive, like a data science team needing to operate within a specific budget, even if performance suffers slightly. Extending AutoSLO to support these alternative SLO types would require new forecasting and modeling techniques in order to determine the best submission times for deadline-driven queries and the per-period budget allocation under a longer-term budget ceiling.

Suggesting and Refining SLOs. No matter the exact type of SLO, it could be that a slight reconsideration could lead to non-negligible benefits. For example, relaxing the SLO for queries in a dashboarding workload from 20 ms to 23 ms could mean that a smaller cluster

would be sufficient to handle it. Put another way, the current formulation in Problem 3.3 concerns *whether* each query meets its SLO, without quantifying *how close* the observed latency is to the SLO value (in either direction) and the opportunities such closeness may imply. In this vein, one could extend AutoSLO so that it also helps *refine* the user-provided SLOs. By using the workload history to analyze latency trends across different contexts, the system could suggest revised SLOs when attainable at a significantly lower cost. Previous work has studied part of this problem in the context of allocating a number of containers to each query [165, 167] and could serve as valuable inspiration.

Expanding the Action Space. AutoSLO currently focuses on two types of actions: routing queries and spinning up/tearing down clusters. However, in order to meet arbitrary performance objectives, one could consider a broader arsenal of actions. For example, one could implement vertical resizing of existing clusters, internal queuing/reordering, cache-aware query routing and cache pre-warming, or query preemption. Supporting additional action types could lead to significant improvements in the capabilities of the performance management system. However, new techniques would be needed to accurately and efficiently assess the tradeoffs that each new action type implies.

4.3.3 Operating in Uncertain Environments

The complexity of modern systems has motivated both this thesis and the three principles introduced in Section 1.1. Such systems give rise to similarly complex internal states: for example, query execution conditions may never be exactly replicable when resources are virtualized; different software components may be updated and patched at different times and frequencies; and many relationships may be non-linear or non-monotonic to begin with. When building models of such systems, for diagnostic or management purposes, we must contend with a third research question: *how can diagnosis and management remain reliable and system models remain accurate when relationships can genuinely vary across operating conditions?* Two extensions of this thesis can pave the way to addressing this question.

Calculating Conditional Average Treatment Effects. Our current formulation in Chapter 2 focuses on computing *average* treatment effects. However, some effects in real systems may depend on the values of additional variables, which define different operational regimes. For example, increasing the memory allocated to a query is not beneficial indefinitely; once every intermediate and end result can fit in memory, additional allocated memory may no longer affect query performance. In light of this observation, one could extend the causal machinery in LOGos to support *conditional* average treatment effects [62], where the causal influence among variables can be modulated based on the values of other variables. This extension would require new interface support to avoid overburdening the user by requiring an overly detailed system specification, but could be an important step towards ensuring the generalizability of the constructed causal model.

Quantifying Latency Prediction Uncertainty. As discussed in Section 3.10.2, accurately predicting query latency under contention is a challenging problem, especially when one needs to compare it to fixed thresholds like SLOs. Even though our current Latency Predictor is highly accurate, it only outputs a single value as the latency prediction, which is then compared to the SLO. As such, any prediction error is directly collapsed into a binary yes/no

decision. Instead, one could redesign the Latency Predictor to output uncertainty-aware predictions, for example by predicting both the mean and standard deviation of the latency for each input query. The main challenge with this approach is that uncertainty estimates are often derived from repeated predictions under sampled model parameters, which can directly increase inference latency. Additional work would be required to mitigate this problem, such as selectively producing uncertainty estimates only when necessary.

4.3.4 Integrating Causal Diagnosis and Management

Finally, as discussed in Section 4.2.3, there is significant interest in closing the control loop between performance diagnosis and management and moving towards a fully automated system life cycle. As our case study in Section 4.1 showed, LOGos and AutoSLO can already be combined to yield useful insights. Our last research question is therefore: *how can we integrate causal system modeling into a continuous system management and improvement loop?* One can pursue this direction in at least three ways.

Making Workload Management Causal. Going beyond merely using LOGos to *interpret* the behavior of AutoSLO, as we did in Section 4.1, one could "close the loop" and use the causal model maintained by LOGos to *drive* workload management decisions in the first place. This can increase the robustness of decision-making when past observations are limited. For example, we could make principled predictions about the performance impact of increasing available memory from 32 GB to 64 GB based on the causal model, even if observations using 64 GB are rare. One challenge with this approach is constructing and maintaining a sufficiently detailed causal model across the boundaries of heterogeneous subsystems. The key to making LOGos work was focusing on a particular causal question in the context of a particular collection of logs, which allowed for the *partial* determination of the overall causal graph. However, to support causal reasoning in the context of end-to-end workload management, one may need to collect and maintain a significant amount of additional detail.

Improving the Design of AutoSLO. Alongside an "online" integration of the two systems, an "offline" integration is also possible. In such an integration, one can use LOGos to improve the *design* of AutoSLO over time, with or without directly leveraging causal reasoning for SLO-related decision-making. In particular, one would use LOGos for periodic troubleshooting over time, keeping track of the variables or events that tend to emerge as common culprits or promising treatments during causal inference. One could then use these lessons to design more effective components for AutoSLO. For example, the case study in Section 4.1 could be used to motivate additional work on the Latency Predictor; after that work is completed, repeating the same analysis may point us to the next bottleneck. Realizing this direction would require care to avoid performance regressions and overfitting as both the underlying data warehouse and AutoSLO evolve over time.

Refining the Causal Model in an SLO-aware Way. Finally, there are ways that the performance management capability unlocked by AutoSLO could help collect data to improve the causal model itself. In some of the causal inference literature [5, 69, 76, 216, 227], controlled experimentation is used to selectively collect interventional data that maximally reduce uncertainty about the causal model. For example, recent work proposes a Bayesian factor-based intervention strategy to quantify the evidence strength in favor of different

hypotheses regarding the presence of causal graph edges [227]. Normally, such approaches are challenging to implement in a live production system, because the interventional experiments can transiently hurt performance. However, one could use the SLO maintenance capability of AutoSLO to intelligently select *when* to experiment, while also quickly reacting to any adverse effects of this experimentation. Over time, this approach would help construct a more complete causal model of the underlying data warehouse, which could support performance diagnosis and counterfactual reasoning.

4.4 Closing Remarks

In this thesis, we began by observing that increasingly complex data systems contribute to an abstraction gap. Although complexity provides new capabilities and greater flexibility, it implies larger volumes of low-level operational evidence and similarly fine-grained control mechanisms. Meanwhile, operators reason in terms of higher-level questions and objectives. Therefore, fully reaping the performance benefits of a complex system in a cost-efficient manner requires interfaces that can translate between these two levels of abstraction.

We presented two such complementary interfaces. With LOGos (Chapter 2), we investigated how textual system logs can be used to answer causal questions. By selectively soliciting human judgments only about the most impactful causal relationships, LOGos empowers operators to investigate undesirable behavior and assess possible interventions without requiring them to construct or verify a complete causal graph of the system.

With AutoSLO (Chapter 3), we investigated the inverse problem of translating operational objectives into system actions. AutoSLO treats latency SLOs as first-class inputs and meets them by coordinating query routing and cluster management decisions across multiple timescales. In doing so, it preserves acceptable application-level performance while minimizing cost, without requiring operators to manually translate their business-driven latency constraints into system-related resource configuration specifications.

Although these two systems traverse the abstraction gap in opposite directions, they both reflect the three common design principles introduced in Section 1.1: they scope modeling based on operator intent, refine their models of the underlying system through feedback, and prioritize reasoning effort based on its anticipated impact. As data systems continue to grow in capability and complexity, continuing to pursue research directions like the ones presented in Section 4.3 will be important for ensuring that the performance benefits of these new systems do not come at the expense of our ability to understand and manage them.

Bibliography

[1] Emile HL Aarts, Jan HM Korst, and Peter JM van Laarhoven. 1988. A Quantitative Analysis of the Simulated Annealing Algorithm: A Case Study for the Traveling Salesman Problem. *Journal of Statistical Physics* 50 (1988), 187–206.

[2] Nadia Abd-Alsabour. 2014. A Review on Evolutionary Feature Selection. In *2014 European Modelling Symposium*. IEEE Computer Society CPS, Los Alamitos, CA, USA, 20–26. https://doi.org/10.1109/EMS.2014.28

[3] Mania Abdi, Samuel Ginzburg, Xiayue Charles Lin, Jose Faleiro, Gohar Irfan Chaudhry, Inigo Goiri, Ricardo Bianchini, Daniel S Berger, and Rodrigo Fonseca. 2023. Palette Load Balancing: Locality Hints for Serverless Functions. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 365–380. https://doi.org/10.1145/3552326.3567496

[4] Pooja Aggarwal, Ajay Gupta, Prateeti Mohapatra, Seema Nagar, Atri Mandal, Qing Wang, and Amit Paradkar. 2021. Localization of Operational Faults in Cloud Applications by Mining Causal Dependencies in Logs Using Golden Signals. In *Service-Oriented Computing - ICSOC 2020 Workshops*. Springer International, Cham, 137–149. https://doi.org/10.1007/978-3-030-76352-7_17

[5] Raj Agrawal, Chandler Squires, Karren Yang, Karthik Shanmugam, and Caroline Uhler. 2019. ABCD-Strategy: Budgeted Experimental Design for Targeted Causal Structure Discovery. arXiv:1902.10347 [stat.ME] https://arxiv.org/abs/1902.10347

[6] Mumtaz Ahmad, Ashraf Aboulnaga, and Shivnath Babu. 2009. Query interactions in database workloads. In *Proceedings of the Second International Workshop on Testing Database Systems* (Providence, Rhode Island) *(DBTest '09)*. Association for Computing Machinery, New York, NY, USA, Article 11, 6 pages. https://doi.org/10.1145/1594156.1594170

[7] Mumtaz Ahmad, Ashraf Aboulnaga, Shivnath Babu, and Kamesh Munagala. 2008. QShuffler: Getting the Query Mix Right. In *2008 IEEE 24th International Conference on Data Engineering*. IEEE, Piscataway, NJ, USA, 1415–1417. https://doi.org/10.1109/ICDE.2008.4497574

[8] Mumtaz Ahmad, Songyun Duan, Ashraf Aboulnaga, and Shivnath Babu. 2011. Predicting completion times of batch query workloads using interaction-aware models and

simulation. In *Proceedings of the 14th International Conference on Extending Database Technology* (Uppsala, Sweden) *(EDBT/ICDT '11)*. Association for Computing Machinery, New York, NY, USA, 449–460. https://doi.org/10.1145/1951365.1951419

[9] OECD AI. 2026. Absolute Relative Error (ARE). Retrieved July 13, 2026 from https://oecd.ai/en/catalogue/metrics/absolute-relative-error-are

[10] 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] https://arxiv.org/abs/2206.05257

[11] 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)*. USENIX Association, Boston, MA, 1115–1147. https://www.usenix.org/conference/nsdi23/presentation/alomar

[12] Amazon Web Services. 2025. Amazon Aurora DSQL is now generally available. AWS What's New. https://aws.amazon.com/about-aws/whats-new/2025/05/amazon-aurora-dsql-generally-available/ Accessed: 2026-08-08.

[13] Amazon Web Services. 2026. Amazon Redshift. https://aws.amazon.com/redshift/. Retrieved June 1, 2026.

[14] Amazon Web Services. 2026. Amazon Redshift Pricing. https://aws.amazon.com/redshift/pricing/. Retrieved March 17, 2026.

[15] Amazon Web Services. 2026. Datashares - Amazon Redshift. https://docs.aws.amazon.com/redshift/latest/mgmt/query-editor-v2-datashare-using.html. Retrieved March 19, 2026.

[16] Amazon Web Services. 2026. Serverless Computing - AWS Lambda. https://aws.amazon.com/lambda/. Retrieved March 20, 2026.

[17] Steen A. Andersson, David Madigan, and Michael D. Perlman. 1997. A Characterization of Markov Equivalence Classes for Acyclic Digraphs. *The Annals of Statistics* 25, 2 (1997), 505 – 541. https://doi.org/10.1214/aos/1031833662

[18] Alessandro Antonucci, Gregorio Piqué, and Marco Zaffalon. 2023. Zero-shot Causal Graph Extrapolation from Text via LLMs. arXiv:2312.14670 [cs.AI] https://arxiv.org/abs/2312.14670

[19] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. 2022. Amazon Redshift Re-invented. In *Proceedings of the 2022 International Conference*

on Management of Data (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 2205–2217. https://doi.org/10.1145/3514221.3526045

[20] Pankaj Arora, Surajit Chaudhuri, Sudipto Das, Junfeng Dong, Cyril George, Ajay Kalhan, Arnd Christian König, Willis Lang, Changsong Li, Feng Li, et al. 2023. Flexible Resource Allocation for Relational Database-as-a-Service. *Proceedings of the VLDB Endowment* 16, 13 (2023), 4202–4215.

[21] 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 (SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 440–453. https://doi.org/10.1145/2934872.2934884

[22] 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, Piscataway, NJ, USA, 237–243. https://doi.org/10.1109/MASCOTS.2018.00031

[23] Taiyu Ban, Lyuzhou Chen, Derui Lyu, Xiangyu Wang, and Huanhuan Chen. 2023. Causal Structure Learning Supervised by Large Language Model. arXiv:2311.11689 [cs.AI] https://arxiv.org/abs/2311.11689

[24] Taiyu Ban, Lyvzhou Chen, Xiangyu Wang, and Huanhuan Chen. 2023. From Query Tools to Causal Architects: Harnessing Large Language Models for Advanced Causal Discovery from Data. arXiv:2306.16902 [cs.AI] https://arxiv.org/abs/2306.16902

[25] Jaeho Bang, Gaurav Tarlok Kakkar, Pramod Chunduri, Subrata Mitra, and Joy Arulraj. 2023. Seiden: Revisiting Query Processing in Video Database Systems. *Proc. VLDB Endow.* 16, 9 (May 2023), 2289–2301. https://doi.org/10.14778/3598581.3598599

[26] Bradley Barnhart, Marc Brooker, Daniil Chinenkov, Tony Hooper, Jihoun Im, Prakash Chandra Jha, Tim Kraska, Ashok Kurakula, Alexey Kuznetsov, Grant McAlister, Arjun Muthukrishnan, Aravinthan Narayanan, Douglas Terry, Bhuvan Urgaonkar, and Jiaming Yan. 2024. Resource Management in Aurora Serverless. *Proc. VLDB Endow.* 17, 12 (Aug. 2024), 4038–4050. https://doi.org/10.14778/3685800.3685825

[27] Jeff Barr. 2015. Now Available – Amazon Aurora. AWS News Blog. https://aws.amazon.com/blogs/aws/now-available-amazon-aurora/ Accessed: 2026-08-08.

[28] Favyen Bastani, Oscar Moll, and Sam Madden. 2020. Vaas: video analytics at scale. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 2877–2880. https://doi.org/10.14778/3415478.3415498

[29] 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

[30] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. 2016. *Site Reliability Engineering: How Google Runs Production Systems* (1st ed.). O'Reilly Media, Inc., Santa Rosa, CA, USA.

[31] Patrick Blöbaum, Peter Götz, Kailash Budhathoki, Atalanti A. Mastakouri, and Dominik Janzing. 2024. DoWhy-GCM: An Extension of DoWhy for Causal Inference in Graphical Causal Models. *Journal of Machine Learning Research* 25, 147 (2024), 1–7. http://jmlr.org/papers/v25/22-1258.html

[32] Marc Brooker, Marc Bowes, Mike Hershey, Zak van der Merwe, James Morle, and Matthys Strydom. 2026. Aurora DSQL: Scalable, Multi-Region OLTP. arXiv:2607.13276 [cs.DB] https://arxiv.org/abs/2607.13276

[33] Kailash Budhathoki, Lenon Minorics, Patrick Bloebaum, and Dominik Janzing. 2022. Causal structure-based root cause analysis of outliers. In *Proceedings of the 39th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato (Eds.), Vol. 162. PMLR, Cambridge, MA, USA, 2357–2369. https://proceedings.mlr.press/v162/budhathoki22a.html

[34] Hengrui Cai, Yixin Wang, Michael Jordan, and Rui Song. 2023. On Learning Necessary and Sufficient Causal Graphs. In *Advances in Neural Information Processing Systems*, Vol. 36. Curran Associates, Inc., Red Hook, NY, USA, 42148–42160. https://proceedings.neurips.cc/paper_files/paper/2023/file/837b396039248acb08c385bebb6291b4-Paper-Conference.pdf

[35] Cristiano Calcagno, Dino Distefano, Jeremy 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*, Klaus Havelund, Gerard Holzmann, and Rajeev Joshi (Eds.). Springer International, Cham, 3–11.

[36] Jiashen Cao, Rathijit Sen, Matteo Interlandi, Joy Arulraj, and Hyesoon Kim. 2023. GPU Database Systems Characterization and Optimization. *Proc. VLDB Endow.* 17, 3 (Nov. 2023), 441–454. https://doi.org/10.14778/3632093.3632107

[37] Gohar Irfan Chaudhry, Esha Choukse, Íñigo Goiri, Rodrigo Fonseca, Adam Belay, and Ricardo Bianchini. 2025. Towards Resource-Efficient Compound AI Systems. In *Proceedings of the 2025 Workshop on Hot Topics in Operating Systems* (Banff, AB, Canada) *(HotOS '25)*. Association for Computing Machinery, New York, NY, USA, 218–224. https://doi.org/10.1145/3713082.3730377

[38] Chaoyu Chen, Hang Yu, Zhichao Lei, Jianguo Li, Shaokang Ren, Tingkai Zhang, Silin Hu, Jianchao Wang, and Wenhui Shi. 2023. BALANCE: Bayesian Linear Attribution for Root Cause Localization. *Proc. ACM Manag. Data* 1, 1, Article 95 (May 2023), 26 pages. https://doi.org/10.1145/3588949

[39] Shuang Chen, Christina Delimitrou, and José F. Martínez. 2019. PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 107–120. https://doi.org/10.1145/3297858.3304005

[40] Sibei Chen, Ju Fan, Bin Wu, Nan Tang, Chao Deng, Pengyi Wang, Ye Li, Jian Tan, Feifei Li, Jingren Zhou, and Xiaoyong Du. 2025. Automatic Database Configuration Debugging using Retrieval-Augmented Language Models. *Proc. ACM Manag. Data* 3, 1, Article 13 (Feb. 2025), 27 pages. https://doi.org/10.1145/3709663

[41] Wenyan Chen, Chengzhi Lu, Huanle Xu, Kejiang Ye, and Chengzhong Xu. 2025. Multiplexing Dynamic Deep Learning Workloads with SLO-awareness in GPU Clusters. In *Proceedings of the Twentieth European Conference on Computer Systems* (Rotterdam, Netherlands) *(EuroSys '25)*. Association for Computing Machinery, New York, NY, USA, 589–604. https://doi.org/10.1145/3689031.3696074

[42] Ludmila Cherkasova. 1998. *Improving WWW proxies performance with greedy-dual-size-frequency caching policy.* Hewlett-Packard Laboratories, Palo Alto, CA, USA.

[43] Yun Chi, Hakan Hacıgümüş, Wang-Pin Hsiung, and Jeffrey F. Naughton. 2013. Distribution-based Query Scheduling. *Proc. VLDB Endow.* 6, 9 (jul 2013), 673–684. https://doi.org/10.14778/2536360.2536367

[44] Yun Chi, Hyun Jin Moon, and Hakan Hacıgümüş. 2011. iCBS: incremental cost-based scheduling under piecewise linear SLAs. *Proceedings of the VLDB Endowment* 4, 9 (2011), 563–574.

[45] Yun Chi, Hyun Jin Moon, Hakan Hacıgümüş, and Junichi Tatemura. 2011. SLA-tree: a framework for efficiently supporting SLA-based decisions in cloud computing. In *Proceedings of the 14th International Conference on Extending Database Technology* (Uppsala, Sweden) *(EDBT/ICDT '11)*. Association for Computing Machinery, New York, NY, USA, 129–140. https://doi.org/10.1145/1951365.1951383

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

[47] Leonid Chindelevitch, Daniel Ziemek, Ahmed Enayetallah, Ranjit Randhawa, Ben Sidders, Christoph Brockel, and Enoch S. Huang. 2012. Causal reasoning on biological networks: interpreting transcriptional changes. *Bioinformatics* 28, 8 (02 2012), 1114–1121. https://doi.org/10.1093/bioinformatics/bts090 arXiv:https://academic.oup.com/bioinformatics/article-pdf/28/8/1114/48930575/bioinformatics_28_8_1114.pdf

[48] Davin Choo, Kirankumar Shiragur, and Arnab Bhattacharyya. 2022. Verification and search algorithms for causal DAGs. In *Advances in Neural Information Processing Systems*, Vol. 35. Curran Associates, Inc., Red Hook, NY,

USA, 12787–12799. https://proceedings.neurips.cc/paper_files/paper/2022/file/5340b0c0b76dc0115f5cc91c20c1251d-Paper-Conference.pdf

[49] Tom Claassen, Joris Mooij, and Tom Heskes. 2013. Learning Sparse Causal Models is not NP-hard. arXiv:1309.6824 [cs.AI] https://arxiv.org/abs/1309.6824

[50] 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)*. USENIX Association, Carlsbad, CA, 17–32. https://www.usenix.org/conference/osdi18/presentation/weidong

[51] Carlo Curino, Evan P.C. Jones, Samuel Madden, and Hari Balakrishnan. 2011. Workload-aware database monitoring and consolidation. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data* (Athens, Greece) *(SIGMOD '11)*. Association for Computing Machinery, New York, NY, USA, 313–324. https://doi.org/10.1145/1989323.1989357

[52] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 215–226. https://doi.org/10.1145/2882903.2903741

[53] Hetong Dai, Heng Li, Che-Shao Chen, Weiyi Shang, and Tse-Hsun Chen. 2022. Logram: Efficient Log Parsing Using n-Gram Dictionaries. *IEEE Transactions on Software Engineering* 48, 3 (2022), 879–892. https://doi.org/10.1109/TSE.2020.3007554

[54] Ermira Daka and Gordon Fraser. 2014. A Survey on Unit Testing Practices and Problems. In *IEEE 25th International Symposium on Software Reliability Engineering*. IEEE, Piscataway, NJ, USA, 201–211. https://doi.org/10.1109/ISSRE.2014.11

[55] Sudipto Das, Feng Li, Vivek R. Narasayya, and Arnd Christian König. 2016. Automated Demand-driven Resource Scaling in Relational Database-as-a-Service. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1923–1934. https://doi.org/10.1145/2882903.2903733

[56] Datadog. 2022. Automated root cause analysis with Watchdog RCA. Retrieved October 21, 2024 from https://www.datadoghq.com/blog/datadog-watchdog-automated-root-cause-analysis/

[57] Datadog. 2024. Retrieved October 21, 2024 from https://www.datadoghq.com/

[58] Biplob Debnath, Mohiuddin Solaimani, Muhammad Ali 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, Piscataway, NJ, USA, 1052–1062. https://doi.org/10.1109/ICDCS.2018.00105

[59] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware scheduling for heterogeneous datacenters. *SIGPLAN Not.* 48, 4 (March 2013), 77–88. https://doi.org/10.1145/2499368.2451125

[60] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: resource-efficient and QoS-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Salt Lake City, Utah, USA) *(ASPLOS '14)*. Association for Computing Machinery, New York, NY, USA, 127–144. https://doi.org/10.1145/2541940.2541941

[61] Yanlei Diao, Dominik Horn, Andreas Kipf, Oleksandr Shchur, Ines Benito, Wenjian Dong, Davide Pagano, Pascal Pfeil, Vikram Nathan, Balakrishnan Narayanaswamy, and Tim Kraska. 2024. Forecasting Algorithms for Intelligent Resource Scaling: An Experimental Analysis. In *Proceedings of the 2024 ACM Symposium on Cloud Computing* (Redmond, WA, USA) *(SoCC '24)*. Association for Computing Machinery, New York, NY, USA, 126–143. https://doi.org/10.1145/3698038.3698564

[62] Vanessa Didelez and Robin J. Evans. 2025. Conditional Effects. *Causal Inference* lecture notes. https://www.stats.ox.ac.uk/~evans/APTS/ce.html#conditional-effects Accessed July 25, 2026.

[63] Jialin Ding, Matt Abrams, Sanghita Bandyopadhyay, Luciano Di Palma, Yanzhu Ji, Davide Pagano, Gopal Paliwal, Panos Parchas, Pascal Pfeil, Orestis Polychroniou, Gaurav Saxena, Aamer Shah, Amina Voloder, Sherry Xiao, Davis Zhang, and Tim Kraska. 2024. Automated Multidimensional Data Layouts in Amazon Redshift. In *Companion of the 2024 International Conference on Management of Data* (Santiago, AA, Chile) *(SIGMOD '24)*. Association for Computing Machinery, New York, NY, USA, 55–67. https://doi.org/10.1145/3626246.3653379

[64] Efim A Dinic. 1970. Algorithm for solution of a problem of maximum flow in networks with power estimation. In *Soviet Math. Doklady*, Vol. 11. American Mathematical Society, Providence, RI, USA, 1277–1280.

[65] 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.

[66] Min Du and Feifei Li. 2016. Spell: Streaming Parsing of System Event Logs. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*. IEEE, Piscataway, NJ, USA, 859–864. https://doi.org/10.1109/ICDM.2016.0103

[67] Jennie Duggan, Ugur Cetintemel, Olga Papaemmanouil, and Eli Upfal. 2011. Performance prediction for concurrent database workloads. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data* (Athens, Greece)

(SIGMOD '11). Association for Computing Machinery, New York, NY, USA, 337–348. https://doi.org/10.1145/1989323.1989359

[68] Dominik Durner, Lennart Espe, Jana Giceva, and Anja Gruenheid. 2024. TracEx: Understanding and Analyzing Database Traces. In *Conference on Innovative Data Systems Research (CIDR)*. Very Large Data Base Endowment Inc, USA, 7.

[69] Muhammad Qasim Elahi, Lai Wei, Murat Kocaoglu, and Mahsa Ghasemi. 2024. Adaptive online experimental design for causal discovery. In *Proceedings of the 41st International Conference on Machine Learning* (Vienna, Austria) *(ICML'24)*. JMLR.org, Article 493, 24 pages.

[70] 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.

[71] Alexander Fuerst and Prateek Sharma. 2021. FaasCache: keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 386–400. https://doi.org/10.1145/3445814.3446757

[72] Sainyam Galhotra, Amir Gilad, Sudeepa Roy, and Babak Salimi. 2022. HypeR: Hypothetical Reasoning With What-If and How-To Queries Using a Probabilistic Causal Approach. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1598–1611. https://doi.org/10.1145/3514221.3526149

[73] 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* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 135–151. https://doi.org/10.1145/3445814.3446700

[74] Yuanning Gao, Xiuqi Huang, Xuanhe Zhou, Xiaofeng Gao, Guoliang Li, and Guihai Chen. 2023. DBAugur: An Adversarial-based Trend Forecasting System for Diversified Workloads. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, Piscataway, NJ, USA, 27–39. https://doi.org/10.1109/ICDE55515.2023.00385

[75] Maxime Gasse, Damien Grasset, Guillaume Gaudron, and Pierre-Yves Oudeyer. 2021. Causal Reinforcement Learning using Observational and Interventional Data. arXiv:2106.14421 [cs.LG]

[76] AmirEmad Ghassami, Saber Salehkaleybar, Negar Kiyavash, and Elias Bareinboim. 2018. Budgeted experiment design for causal structure learning. In *International Conference on Machine Learning*. PMLR, 1724–1733.

[77] Victor Giannakouris and Immanuel Trummer. 2025. λ-Tune: Harnessing Large Language Models for Automated Database System Tuning. *Proc. ACM Manag. Data* 3, 1, Article 2 (Feb. 2025), 26 pages. https://doi.org/10.1145/3709652

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

[79] Helga Gudmundsdottir, Babak Salimi, Magdalena Balazinska, Dan R.K. Ports, and Dan Suciu. 2017. A Demonstration of Interactive Analysis of Performance Measurements with Viska. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1707–1710. https://doi.org/10.1145/3035918.3056448

[80] Haixuan Guo, Shuhan Yuan, and Xintao Wu. 2021. LogBERT: Log Anomaly Detection via BERT. In *2021 International Joint Conference on Neural Networks (IJCNN)*. IEEE, Piscataway, NJ, USA, 1–8. https://doi.org/10.1109/IJCNN52387.2021.9534113

[81] Shantanu Gupta, David Childers, and Zachary Chase Lipton. 2023. Local Causal Discovery for Estimating Causal Effects. In *Proceedings of the Second Conference on Causal Learning and Reasoning (Proceedings of Machine Learning Research)*, Vol. 213. PMLR, Cambridge, MA, USA, 408–447. https://proceedings.mlr.press/v213/gupta23b.html

[82] 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* (Denver, Colorado) *(SC '17)*. Association for Computing Machinery, New York, NY, USA, Article 44, 12 pages. https://doi.org/10.1145/3126908.3126937

[83] Haitian Hang, Xiu Tang, Jianling Sun, Lingfeng Bao, David Lo, and Haoye Wang. 2024. Robust Auto-Scaling with Probabilistic Workload Forecasting for Cloud Databases. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, Piscataway, NJ, USA, 4016–4029. https://doi.org/10.1109/ICDE60146.2024.00308

[84] Vipul Harsh, Wenxuan Zhou, Sachin Ashok, Radhika Niranjan Mysore, Brighten Godfrey, and Sujata Banerjee. 2023. Murphy: Performance Diagnosis of Distributed Cloud Applications. In *Proceedings of the ACM SIGCOMM 2023 Conference* (New York, NY, USA) *(ACM SIGCOMM '23)*. Association for Computing Machinery, New York, NY, USA, 438–451. https://doi.org/10.1145/3603269.3604877

[85] Peter E Hart, Nils J Nilsson, and Bertram Raphael. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE transactions on Systems Science and Cybernetics* 4, 2 (1968), 100–107.

[86] 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)*, Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun

Wan (Eds.). Association for Computational Linguistics, Hong Kong, China, 2988–2999. https://doi.org/10.18653/v1/D19-1296

[87] 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 *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*. International Joint Conferences on Artificial Intelligence Organization, California, USA, 5003–5009. https://doi.org/10.24963/ijcai.2019/695

[88] Alain Hauser and Peter Bühlmann. 2012. Characterization and Greedy Learning of Interventional Markov Equivalence Classes of Directed Acyclic Graphs. *Journal of Machine Learning Research* 13, 1 (Aug 2012), 2409–2464.

[89] 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, Piscataway, NJ, USA, 654–661. https://doi.org/10.1109/DSN.2016.66

[90] 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.

[91] 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)*. IEEE, Piscataway, NJ, USA, 33–40. https://doi.org/10.1109/ICWS.2017.13

[92] 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* (Virtual Event, Ireland) *(CIKM '20)*. Association for Computing Machinery, New York, NY, USA, 3023–3030. https://doi.org/10.1145/3340531.3412763

[93] Antony S. Higginson, Mihaela Dediu, Octavian Arsene, Norman W. Paton, and Suzanne M. Embury. 2020. Database Workload Capacity Planning using Time Series Analysis and Machine Learning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 769–783. https://doi.org/10.1145/3318464.3386140

[94] Benjamin Hilprecht and Carsten Binnig. 2022. Zero-shot cost models for out-of-the-box learned cost prediction. *Proc. VLDB Endow.* 15, 11 (July 2022), 2361–2374. https://doi.org/10.14778/3551793.3551799

[95] Gerard J. Holzmann and Doron Peled. 1995. *An Improvement in Formal Verification*. Springer US, Boston, MA, 197–211. https://doi.org/10.1007/978-0-387-34878-0_13

[96] Ke Hong, Xiuhong Li, Lufang Chen, Qiuli Mao, Guohao Dai, Xuefei Ning, Shengen Yan, Yun Liang, and Yu Wang. 2025. SOLA: Optimizing SLO Attainment for Large Language Model Serving with State-Aware Scheduling. In *Eighth Conference on Machine Learning and Systems*. 13. https://openreview.net/forum?id=ubIvpetAd6

[97] Cunchen Hu, Heyang Huang, Liangliang Xu, Xusheng Chen, Jiang Xu, Shuang Chen, Hao Feng, Chenxi Wang, Sa Wang, Yungang Bao, Ninghui Sun, and Yizhou Shan. 2024. Inference without Interference: Disaggregate LLM Inference for Mixed Downstream Workloads. arXiv:2401.11181 [cs.DC] https://arxiv.org/abs/2401.11181

[98] Huining Hu, Zhentao Li, and Adrian R Vetta. 2014. Randomized Experimental Design for Causal Graph Discovery. In *Advances in Neural Information Processing Systems*, Vol. 27. Curran Associates, Inc., Red Hook, NY, USA. https://proceedings.neurips.cc/paper_files/paper/2014/file/e53a0a2978c28872a4505bdb51db06dc-Paper.pdf

[99] Hanxian Huang, Tarique Siddiqui, Rana Alotaibi, Carlo Curino, Jyoti Leeka, Alekh Jindal, Jishen Zhao, Jesús Camacho-Rodríguez, and Yuanyuan Tian. 2024. Sibyl: Forecasting Time-Evolving Query Workloads. *Proc. ACM Manag. Data* 2, 1, Article 53 (March 2024), 27 pages. https://doi.org/10.1145/3639308

[100] 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, Piscataway, NJ, USA, 1–8. https://doi.org/10.1109/NOMS47738.2020.9110435

[101] Yuwei Huang and Guoliang Li. 2024. Laser: Buffer-Aware Learned Query Scheduling in Master-Standby Databases. *Proc. VLDB Endow.* 18, 3 (Nov. 2024), 743–755. https://doi.org/10.14778/3712221.3712239

[102] Randall Hunt. 2018. Aurora Serverless MySQL Generally Available. AWS News Blog. https://aws.amazon.com/blogs/aws/aurora-serverless-ga/ Accessed: 2026-08-08.

[103] Antti Hyttinen, Frederick Eberhardt, and Patrik O Hoyer. 2013. Experiment Selection for Causal Discovery. *Journal of Machine Learning Research* 14, 1 (2013), 3041–3071.

[104] Guido W. Imbens. 2020. Potential Outcome and Directed Acyclic Graph Approaches to Causality: Relevance for Empirical Practice in Economics. *Journal of Economic Literature* 58, 4 (December 2020), 1129–79. https://doi.org/10.1257/jel.20191597

[105] Flowerfire Inc. 2024. Sawmill: Universal Log File Analysis and Reporting. Retrieved October 21, 2024 from http://www.sawmill.net/

[106] Intel Corporation. 2019. *Intel Xeon Gold 6230 CPU*. Intel Corporation. Retrieved October 21, 2024 from https://ark.intel.com/content/www/us/en/ark/products/192437/intel-xeon-gold-6230-processor-27-5m-cache-2-10-ghz.html

[107] Vimalkumar Jeyakumar, Omid Madani, Ali Parandeh, Ashutosh Kulshreshtha, Weifei Zeng, and Navindra Yadav. 2019. ExplainIt! – A Declarative Root-cause Analysis

Engine for Time Series Data. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 333–348. https://doi.org/10.1145/3299869.3314048

[108] Peng Jia, Shaofeng Cai, Beng Chin Ooi, Pinghui Wang, and Yiyuan Xiong. 2023. Robust and Transferable Log-based Anomaly Detection. *Proc. ACM Manag. Data* 1, 1, Article 64 (May 2023), 26 pages. https://doi.org/10.1145/3588918

[109] Jiajun Jiang, Weihai Lu, Junjie Chen, Qingwei Lin, Pu Zhao, Yu Kang, Hongyu Zhang, Yingfei Xiong, Feng Gao, Zhangwei Xu, Yingnong Dang, and Dongmei Zhang. 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* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1410–1420. https://doi.org/10.1145/3368089.3417054

[110] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2022. Hermod: principled and practical scheduling for serverless functions. In *Proceedings of the 13th Symposium on Cloud Computing* (San Francisco, California) *(SoCC '22)*. Association for Computing Machinery, New York, NY, USA, 289–305. https://doi.org/10.1145/3542929.3563468

[111] 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.

[112] 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 (SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 344–360. https://doi.org/10.1145/2815400.2815412

[113] Omar Khattab, Keshav Santhanam, Xiang Lisa Li, David Hall, Percy Liang, Christopher Potts, and Matei Zaharia. 2023. Demonstrate-Search-Predict: Composing retrieval and language models for knowledge-intensive NLP. arXiv:2212.14024 [cs.CL] https://arxiv.org/abs/2212.14024

[114] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. 2023. DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines. arXiv:2310.03714 [cs.CL] https://arxiv.org/abs/2310.03714

[115] Murat Kocaoglu, Alex Dimakis, and Sriram Vishwanath. 2017. Cost-Optimal Learning of Causal Graphs. In *Proceedings of the 34th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Vol. 70. PMLR, Cambridge, MA, USA, 1875–1884. https://proceedings.mlr.press/v70/kocaoglu17a.html

[116] Ferdi Kossmann, Ziniu Wu, Alex Turk, Nesime Tatbul, Lei Cao, and Samuel Madden. 2024. CascadeServe: Unlocking Model Cascades for Inference Serving. arXiv:2406.14424 [cs.DC] https://arxiv.org/abs/2406.14424

[117] Ferdi Kossmann, Ziniu Wu, Alex Turk, Nesime Tatbul, Lei Cao, and Samuel Madden. 2026. KEN: An Execution Engine for Unstructured Database Systems. *Proc. VLDB Endow.* 19, 5 (May 2026), 902–916. https://doi.org/10.14778/3796195.3796204

[118] Tim Kraska, Tianyu Li, Samuel Madden, Markos Markakis, Amadou Ngom, Ziniu Wu, and Geoffrey X. Yu. 2023. Check Out the Big Brain on BRAD: Simplifying Cloud Data Processing with Learned Automated Data Meshes. *Proc. VLDB Endow.* 16, 11 (jul 2023), 3293–3301. https://doi.org/10.14778/3611479.3611526

[119] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 611–626. https://doi.org/10.1145/3600006.3613165

[120] Emre Kıcıman, Robert Ness, Amit Sharma, and Chenhao Tan. 2023. Causal Reasoning and Large Language Models: Opening a New Frontier for Causality. arXiv:2305.00050 [cs.AI] https://arxiv.org/abs/2305.00050

[121] Wai-Yin Lam, Bryan Andrews, and Joseph Ramsey. 2022. Greedy relaxations of the sparsest permutation algorithm. In *Proceedings of the Thirty-Eighth Conference on Uncertainty in Artificial Intelligence (Proceedings of Machine Learning Research)*, James Cussens and Kun Zhang (Eds.), Vol. 180. PMLR, Cambridge, MA, USA, 1052–1062. https://proceedings.mlr.press/v180/lam22a.html

[122] Willis Lang, Srinath Shankar, Jignesh M. Patel, and Ajay Kalhan. 2014. Towards Multi-Tenant Performance SLOs. *IEEE Transactions on Knowledge and Data Engineering* 26, 6 (2014), 1447–1463. https://doi.org/10.1109/TKDE.2013.74

[123] Jiale Lao, Yibo Wang, Yufei Li, Jianping Wang, Yunjia Zhang, Zhiyuan Cheng, Wanghu Chen, Mingjie Tang, and Jianguo Wang. 2024. GPTuner: A Manual-Reading Database Tuning System via GPT-Guided Bayesian Optimization. *Proc. VLDB Endow.* 17, 8 (April 2024), 1939–1952. https://doi.org/10.14778/3659437.3659449

[124] 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.

[125] Philipp Leitner, Waldemar Hummer, Benjamin Satzger, Christian Inzinger, and Schahram Dustdar. 2012. Cost-efficient and application SLA-aware client side request scheduling in an infrastructure-as-a-service cloud. In *2012 IEEE Fifth International Conference on Cloud Computing*. IEEE, IEEE, Piscataway, NJ, USA, 213–220.

[126] Yuhang Li, Rong Gu, Chengying Huan, Zhibin Wang, Renjie Yao, Chen Tian, and Guihai Chen. 2025. HotPrefix: Hotness-Aware KV Cache Scheduling for Efficient Prefix Sharing in LLM Inference Systems. *Proc. ACM Manag. Data* 3, 4, Article 250 (Sept. 2025), 27 pages. https://doi.org/10.1145/3749168

[127] Yiyan Li, Haoyang Li, Jing Zhang, Renata Borovica-Gajic, Shuai Wang, Tieying Zhang, Jianjun Chen, Rui Shi, Cuiping Li, and Hong Chen. 2025. AgentTune: An Agent-Based Large Language Model Framework for Database Knob Tuning. *Proc. ACM Manag. Data* 3, 6, Article 293 (Dec. 2025), 29 pages. https://doi.org/10.1145/3769758

[128] Chunwei Liu, Matthew Russo, Michael Cafarella, Lei Cao, Peter Baile Chen, Zui Chen, Michael Franklin, Tim Kraska, Samuel Madden, Rana Shahout, and Gerardo Vitagliano. 2025. Palimpzest: Optimizing AI-Powered Analytics with Declarative Query Processing. In *Conference on Innovative Data Systems Research (CIDR)*. Very Large Data Base Endowment Inc, USA, 7.

[129] Chunwei Liu, Matthew Russo, Michael Cafarella, Lei Cao, Peter Baille Chen, Zui Chen, Michael Franklin, Tim Kraska, Samuel Madden, and Gerardo Vitagliano. 2024. A Declarative System for Optimizing AI Workloads. arXiv:2405.14696 [cs.CL] https://arxiv.org/abs/2405.14696

[130] Haotian Liu, Runzhong Li, Ziyang Zhang, and Bo Tang. 2024. Tao: Improving Resource Utilization while Guaranteeing SLO in Multi-tenant Relational Database-as-a-Service. *Proc. ACM Manag. Data* 2, 4, Article 205 (Sept. 2024), 26 pages. https://doi.org/10.1145/3677141

[131] Xiaoyu Liu, Paiheng Xu, Junda Wu, Jiaxin Yuan, Yifan Yang, Yuhang Zhou, Fuxiao Liu, Tianrui Guan, Haoliang Wang, Tong Yu, Julian McAuley, Wei Ai, and Furong Huang. 2024. Large Language Models and Causal Inference in Collaboration: A Comprehensive Survey. arXiv:2403.09606 [cs.CL] https://arxiv.org/abs/2403.09606

[132] Yudong Liu, Xu Zhang, Shilin He, Hongyu Zhang, Liqun Li, Yu Kang, Yong Xu, Minghua Ma, Qingwei Lin, Yingnong Dang, Saravan Rajmohan, and Dongmei Zhang. 2022. UniParser: A Unified Log Parser for Heterogeneous Log Data. In *Proceedings of the ACM Web Conference 2022* (Virtual Event, Lyon, France) *(WWW '22)*. Association for Computing Machinery, New York, NY, USA, 1893–1901. https://doi.org/10.1145/3485447.3511993

[133] Ziyang Liu, Hakan Hacıgümüş, Hyun Jin Moon, Yun Chi, and Wang-Pin Hsiung. 2013. PMAX: tenant placement in multitenant databases for profit maximization. In *Proceedings of the 16th International Conference on Extending Database Technology* (Genoa, Italy) *(EDBT '13)*. Association for Computing Machinery, New York, NY, USA, 442–453. https://doi.org/10.1145/2452376.2452428

[134] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: improving resource efficiency at scale. *SIGARCH Comput. Archit. News* 43, 3S (June 2015), 450–462. https://doi.org/10.1145/2872887.2749475

[135] Stephanie Long, Tibor Schuster, and Alexandre Piché. 2024. Can large language models build causal graphs? arXiv:2303.05279 [cs.CL] https://arxiv.org/abs/2303.05279

[136] Sindy Löwe, David Madras, Richard Zemel, and Max Welling. 2022. Amortized Causal Discovery: Learning to Infer Causal Graphs from Time-Series Data. In *Proceedings of the First Conference on Causal Learning and Reasoning (Proceedings of Machine Learning Research)*, Vol. 177. PMLR, Cambridge, MA, USA, 509–525. https://proceedings.mlr.press/v177/lowe22a.html

[137] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J. Gordon. 2018. Query-based Workload Forecasting for Self-Driving Database Management Systems. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 631–645. https://doi.org/10.1145/3183713.3196908

[138] 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.

[139] Marloes H. Maathuis, Markus Kalisch, and Peter Bühlmann. 2009. Estimating High-dimensional Intervention Effects from Observational Data. *The Annals of Statistics* 37, 6A (2009), 3133 – 3164. https://doi.org/10.1214/09-AOS685

[140] Ryan Marcus and Olga Papaemmanouil. 2016. WiSeDB: a learning-based workload management advisor for cloud databases. *Proc. VLDB Endow.* 9, 10 (jun 2016), 780–791. https://doi.org/10.14778/2977797.2977804

[141] Ryan Marcus and Olga Papaemmanouil. 2017. Releasing Cloud Databases for the Chains of Performance Prediction Models. In *Conference on Innovative Data Systems Research (CIDR)*. Very Large Data Base Endowment Inc, USA, 11.

[142] Ryan Marcus and Olga Papaemmanouil. 2019. Plan-structured deep neural network models for query performance prediction. *Proc. VLDB Endow.* 12, 11 (July 2019), 1733–1746. https://doi.org/10.14778/3342263.3342646

[143] Markos Markakis. 2026. AutoSLO Code. Retrieved Aug 14, 2026 from https://github.com/mmarkakis/autoslo

[144] Markos Markakis, An Bo Chen, Trinity Gao, and Peter Baile Chen. 2024. LOGos Code. Retrieved Aug 14, 2026 from https://github.com/mitdbg/logos

[145] 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 *Companion of the 2024 International Conference on Management of Data* (Santiago, AA, Chile) *(SIGMOD/PODS '24)*. Association for Computing Machinery, New York, NY, USA, 444–447. https://doi.org/10.1145/3626246.3654731

[146] Markos Markakis and Tim Kraska. 2026. AutoSLO: Practical Latency SLOs on Cloud Data Warehouses – Extended Version. arXiv:2607.11770 [cs.DB] https://arxiv.org/abs/2607.11770

[147] Markos Markakis and Tim Kraska. 2026. Towards Practical Latency SLOs on Cloud Data Warehouses. In *Proceedings of the 2026 Applied AI for Database Systems and Applications (AIDB) Workshop*. Association for Computing Machinery, New York, NY, USA, 5.

[148] Markos Markakis, Brit Youngmann, Trinity Gao, Ziyu Zhang, Rana Shahout, Peter Baile Chen, Chunwei Liu, Ibrahim Sabek, and Michael Cafarella. 2024. From Logs to Causal Inference: Diagnosing Large Systems. *Proc. VLDB Endow.* 18, 2 (Oct. 2024), 158–172. https://doi.org/10.14778/3705829.3705836

[149] Markos Markakis, Ziyu Zhang, Rana Shahout, Trinity Gao, Chunwei Liu, Ibrahim Sabek, and Michael Cafarella. 2024. Press ECCS to Doubt (Your Causal Graph). In *Proceedings of the Conference on Governance, Understanding and Integration of Data for Effective and Responsible AI* (Santiago, AA, Chile) *(GUIDE-AI '24)*. Association for Computing Machinery, New York, NY, USA, 6–15. https://doi.org/10.1145/3665601.3669842

[150] Themis Melissaris, Markos Markakis, Kelly Shaw, and Margaret Martonosi. 2020. PerpLE: Improving the Speed and Effectiveness of Memory Consistency Testing. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, Piscataway, NJ, USA, 329–341. https://doi.org/10.1109/MICRO50266.2020.00037

[151] Daniel Mendoza, Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2021. Interference-Aware Scheduling for Inference Serving. In *Proceedings of the 1st Workshop on Machine Learning and Systems* (Online, United Kingdom) *(EuroMLSys '21)*. Association for Computing Machinery, New York, NY, USA, 80–88. https://doi.org/10.1145/3437984.3458837

[152] Weibin Meng, Ying Liu, Federico Zaiter, Shenglin Zhang, Yihao Chen, Yuzhe Zhang, Yichen Zhu, En Wang, Ruizhi Zhang, Shimin Tao, Dian Yang, Rong Zhou, and Dan Pei. 2020. LogParse: Making Log Parsing Adaptive through Word Classification. In *2020 29th International Conference on Computer Communications and Networks (ICCCN)*. IEEE, Piscataway, NJ, USA, 1–9. https://doi.org/10.1109/ICCCN49398.2020.9209681

[153] Glenford J Myers, Corey Sandler, and Tom Badgett. 2011. *The Art of Software Testing*. Wiley, Hoboken, NJ, USA.

[154] Narmada Naik, Ayush Khandelwal, Mohit Joshi, Madhusudan Atre, Hollis Wright, Kavya Kannan, Scott Hill, Giridhar Mamidipudi, Ganapati Srinivasa, Carlo Bifulco, Brian Piening, and Kevin Matlock. 2023. Applying Large Language Models for Causal Structure Learning in Non Small Cell Lung Cancer. arXiv:2311.07191 [cs.AI] https://arxiv.org/abs/2311.07191

[155] Vivek Narasayya, Sudipto Das, Manoj Syamala, Badrish Chandramuli, and Surajit Chaudhuri. 2013. SQLVM: Performance Isolation in Multi-Tenant Relational Database-as-a-Service. In *Conference on Innovative Data Systems Research (CIDR)*. Very Large Data Base Endowment Inc, USA, 9.

[156] Vivek Narasayya, Sudipto Das, Manoj Syamala, Surajit Chaudhuri, Feng Li, and Hyunjung Park. 2013. A demonstration of SQLVM: performance isolation in multi-tenant relational database-as-a-service. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 1077–1080. https://doi.org/10.1145/2463676.2463686

[157] Vikram Nathan, Vikramank Singh, Zhengchun Liu, Mohammad Rahman, Andreas Kipf, Dominik Horn, Davide Pagano, Gaurav Saxena, Balakrishnan Narayanaswamy, and Tim Kraska. 2024. Intelligent Scaling in Amazon Redshift. In *Companion of the 2024 International Conference on Management of Data* (Santiago, AA, Chile) *(SIGMOD '24)*. Association for Computing Machinery, New York, NY, USA, 269–279. https://doi.org/10.1145/3626246.3653394

[158] 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*, Yuxiao Dong, Dunja Mladenić, and Craig Saunders (Eds.). Springer International, Cham, 122–138.

[159] Francisco Neves, Nuno Machado, and José Pereira. 2018. Falcon: A Practical Log-Based Analysis Tool for Distributed Systems. In *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, Piscataway, NJ, USA, 534–541. https://doi.org/10.1109/DSN.2018.00061

[160] Francisco Neves, Nuno Machado, Ricardo Vilaça, and José Pereira. 2021. Horus: Non-Intrusive Causal Analysis of Distributed Systems Logs. In *51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, Piscataway, NJ, USA, 212–223. https://doi.org/10.1109/DSN48987.2021.00035

[161] Chris J Oates, Jessica Kasza, Julie A Simpson, and Andrew B Forbes. 2017. Repair of partly misspecified causal diagrams. *Epidemiology* 28, 4 (2017), 548–552.

[162] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL] https://arxiv.org/abs/2303.08774

[163] OpenAI. 2024. GPT-4o mini: advancing cost-efficient intelligence. Retrieved October 21, 2024 from https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/

[164] OpenAI. 2024. Models. Retrieved October 21, 2024 from https://platform.openai.com/docs/models/gpt-3-5-turbo

[165] Jennifer Ortiz, Victor Teixeira De Almeida, and Magdalena Balazinska. 2015. Changing the Face of Database Cloud Services with Personalized Service Level Agreements.

In *Conference on Innovative Data Systems Research (CIDR)*. Very Large Data Base Endowment Inc, USA, 13.

[166] Jennifer Ortiz, Brendan Lee, and Magdalena Balazinska. 2016. PerfEnforce Demonstration: Data Analytics with Performance Guarantees. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 2141–2144. https://doi.org/10.1145/2882903.2899402

[167] Jennifer Ortiz, Brendan Lee, Magdalena Balazinska, Johannes Gehrke, and Joseph L Hellerstein. 2018. SLAOrchestrator: Reducing the Cost of Performance SLAs for Cloud Data Analytics. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Berkeley, CA, USA, 547–560.

[168] Biao Ouyang, Yingying Zhang, Hanyin Cheng, Yang Shu, Chenjuan Guo, Bin Yang, Qingsong Wen, Lunting Fan, and Christian S. Jensen. 2024. RCRank: Multimodal Ranking of Root Causes of Slow Queries in Cloud Database Systems. *Proc. VLDB Endow.* 18, 4 (Dec. 2024), 1169–1182. https://doi.org/10.14778/3717755.3717774

[169] James Jie Pan, Jianguo Wang, and Guoliang Li. 2024. Vector Database Management Techniques and Systems. In *Companion of the 2024 International Conference on Management of Data* (Santiago, AA, Chile) *(SIGMOD '24)*. Association for Computing Machinery, New York, NY, USA, 597–604. https://doi.org/10.1145/3626246.3654691

[170] Tirthak Patel and Devesh Tiwari. 2020. CLITE: Efficient and QoS-Aware Co-Location of Multiple Latency-Critical Jobs for Warehouse Scale Computers. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, Piscataway, NJ, USA, 193–206. https://doi.org/10.1109/HPCA47549.2020.00025

[171] 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*. Cognitive Science Society, Seattle, WA, USA, 15–17.

[172] Judea Pearl. 2009. *Causality: Models, Reasoning and Inference*. Cambridge University Press, Cambridge, UK.

[173] Judea Pearl and Dana Mackenzie. 2018. *The Book of Why: The New Science of Cause and Effect*. Basic Books, New York, NY, USA.

[174] 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.

[175] Chen Peng, Di Zhang, and Urbashi Mitra. 2024. Graph Identification and Upper Confidence Evaluation for Causal Bandits with Linear Models. In *ICASSP 2024 - 2024 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, Piscataway, NJ, USA, 7165–7169. https://doi.org/10.1109/ICASSP48485.2024.10445823

[176] Matthew Perron, Raul Castro Fernandez, David DeWitt, Michael Cafarella, and Samuel Madden. 2023. Cackle: Analytical Workload Cost and Performance Stability With Elastic Pools. *Proc. ACM Manag. Data* 1, 4, Article 233 (Dec. 2023), 25 pages. https://doi.org/10.1145/3626720

[177] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. 2020. Starling: A Scalable Query Engine on Cloud Functions. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 131–141. https://doi.org/10.1145/3318464.3380609

[178] Danilo Poccia. 2019. Amazon Aurora PostgreSQL Serverless – Now Generally Available. AWS News Blog. https://aws.amazon.com/blogs/aws/amazon-aurora-postgresql-serverless-now-generally-available/ Accessed: 2026-08-08.

[179] 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* (Madison, Wisconsin) *(SIGMOD '02)*. Association for Computing Machinery, New York, NY, USA, 582–587. https://doi.org/10.1145/564691.564759

[180] Uwe Rohm, Klemens Bohm, and H-J Schek. 2001. Cache-aware query routing in a cluster of databases. In *Proceedings 17th International Conference on Data Engineering*. IEEE, IEEE, Piscataway, NJ, USA, 641–650.

[181] Francisco Romero and Christina Delimitrou. 2018. Mage: online and interference-aware scheduling for multi-scale heterogeneous systems. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques* (Limassol, Cyprus) *(PACT '18)*. Association for Computing Machinery, New York, NY, USA, Article 19, 13 pages. https://doi.org/10.1145/3243176.3243183

[182] 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.

[183] Per Runeson. 2006. A survey of unit testing practices. *IEEE software* 23, 4 (2006), 22–29.

[184] Ibrahim Sabek, Tenzin Samten Ukyab, and Tim Kraska. 2022. LSched: A Workload-Aware Learned Query Scheduler for Analytical Database Systems. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1228–1242. https://doi.org/10.1145/3514221.3526158

[185] 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* (Portland, Oregon, USA) *(PLATEAU '14)*. Association for

Computing Machinery, New York, NY, USA, 43–51. https://doi.org/10.1145/2688204.2688205

[186] Sherif Sakr and Anna Liu. 2012. SLA-based and consumer-centric dynamic provisioning for cloud databases. In *2012 IEEE Fifth International Conference on Cloud Computing*. IEEE, IEEE, Piscataway, NJ, USA, 360–367.

[187] Felix Salfner and Steffen Tschirpke. 2008. Error Log Processing for Accurate Failure Prediction. *WASL* 8 (2008), 4.

[188] 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* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1021–1035. https://doi.org/10.1145/3183713.3196914

[189] Gaurav Saxena, Mohammad Rahman, Naresh Chainani, Chunbin Lin, George Caragea, Fahim Chowdhury, Ryan Marcus, Tim Kraska, Ippokratis Pandis, and Balakrishnan (Murali) Narayanaswamy. 2023. Auto-WLM: Machine Learning Enhanced Workload Management in Amazon Redshift. In *Companion of the 2023 International Conference on Management of Data* (Seattle, WA, USA) *(SIGMOD '23)*. Association for Computing Machinery, New York, NY, USA, 225–237. https://doi.org/10.1145/3555041.3589677

[190] 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.

[191] Bernhard Schölkopf, Francesco Locatello, Stefan Bauer, Nan Rosemary Ke, Nal Kalchbrenner, Anirudh Goyal, and Yoshua Bengio. 2021. Towards Causal Representation Learning. arXiv:2102.11107 [cs.LG]

[192] Amazon Web Services. 2026. Secure and resizable cloud compute - Amazon EC2 - AWS. Retrieved July 13, 2026 from https://aws.amazon.com/ec2/

[193] Shreya Shankar, Tristan Chambers, Tarak Shah, Aditya G. Parameswaran, and Eugene Wu. 2025. DocETL: Agentic Query Rewriting and Evaluation for Complex Document Processing. *Proc. VLDB Endow.* 18, 9 (May 2025), 3035–3048. https://doi.org/10.14778/3746405.3746426

[194] Karthikeyan Shanmugam, Murat Kocaoglu, Alexandros G Dimakis, and Sriram Vishwanath. 2015. Learning Causal Graphs with Small Interventions. In *Advances in Neural Information Processing Systems*, Vol. 28. Curran Associates, Inc., Red Hook, NY, USA. https://proceedings.neurips.cc/paper_files/paper/2015/file/b865367fc4c0845c0682bd466e6ebf4c-Paper.pdf

[195] Amit Sharma and Emre Kiciman. 2020. DoWhy: An End-to-End Library for Causal Inference. arXiv:2011.04216 [stat.ME] https://arxiv.org/abs/2011.04216

[196] 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* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 1477–1488. https://doi.org/10.1145/3540250.3558958

[197] Xu Shi, Ziyang Pan, and Wang Miao. 2023. Data integration in causal inference. *Wiley Interdisciplinary Reviews: Computational Statistics* 15, 1 (2023), e1581.

[198] 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, 72 (2006), 2003–2030. http://jmlr.org/papers/v7/shimizu06a.html

[199] Tomi Silander and Petri Myllymäki. 2006. A simple approach for finding the globally optimal Bayesian network structure. In *Proceedings of the Twenty-Second Conference on Uncertainty in Artificial Intelligence* (Cambridge, MA, USA) *(UAI'06)*. AUAI Press, Arlington, Virginia, USA, 445–452.

[200] Vikramank Singh, Kapil Eknath Vaidya, Vinayshekhar Bannihatti Kumar, Sopan Khosla, Murali Narayanaswamy, Rashmi Gangadharaiah, and Tim Kraska. 2024. Panda: Performance debugging for databases using LLM agents. In *Conference on Innovative Data Systems Research (CIDR)*. Very Large Data Base Endowment Inc, USA, 6.

[201] Snowflake. 2026. Multi-cluster warehouses | Snowflake Documentation. https://docs.snowflake.com/en/user-guide/warehouses-multicluster. Retrieved March 19, 2026.

[202] Snowflake. 2026. Pricing Options. https://www.snowflake.com/en/pricing-options/. Retrieved March 17, 2026.

[203] Snowflake. 2026. Snowflake AI Data Cloud. https://www.snowflake.com/en/. Retrieved June 1, 2026.

[204] Snowflake. 2026. Snowflake key concepts and architecture | Snowflake Documentation. https://docs.snowflake.com/en/user-guide/intro-key-concepts. Retrieved March 19, 2026.

[205] Michael E. Sobel. 2000. Causal Inference in the Social Sciences. *J. Amer. Statist. Assoc.* 95, 450 (01 Jun 2000), 647–651. https://doi.org/10.1080/01621459.2000.10474243

[206] Arjun Sondhi and Ali Shojaie. 2019. The Reduced PC-Algorithm: Improved Causal Structure Learning in Large Random Networks. *Journal of Machine Learning Research* 20, 164 (2019), 1–31. http://jmlr.org/papers/v20/17-601.html

[207] Peter Spirtes and Clark Glymour. 1991. An algorithm for fast recovery of sparse causal graphs. *Social science computer review* 9, 1 (1991), 62–72.

[208] Peter Spirtes, Clark N Glymour, and Richard Scheines. 2000. *Causation, prediction, and search*. MIT press, Cambridge, MA, USA.

[209] Splunk. 2024. Retrieved October 21, 2024 from https://www.splunk.com/

[210] Abraham Starosta. 2021. How Splunk Is Parsing Machine Logs With Machine Learning On NVIDIA's Triton and Morpheus. Retrieved October 21, 2024 from https://www.splunk.com/en_us/blog/it/how-splunk-is-parsing-machine-logs-with-machine-learning-on-nvidia-s-triton-and-morpheus.html

[211] Michael Stonebraker and Ugur Cetintemel. 2005. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *Proceedings of the 21st International Conference on Data Engineering (ICDE '05)*. IEEE Computer Society, USA, 2–11. https://doi.org/10.1109/ICDE.2005.1

[212] Rebecca Taft, Willis Lang, Jennie Duggan, Aaron J. Elmore, Michael Stonebraker, and David DeWitt. 2016. STeP: Scalable Tenant Placement for Managing Database-as-a-Service Deployments. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (Santa Clara, CA, USA) *(SoCC '16)*. Association for Computing Machinery, New York, NY, USA, 388–400. https://doi.org/10.1145/2987550.2987575

[213] A.S. Tanenbaum and D. Wetherall. 2011. *Computer Networks*. Pearson Prentice Hall. https://books.google.com/books?id=T7lGswEACAAJ

[214] Jin Tian, Azaria Paz, and Judea Pearl. 1998. *Finding Minimal D-separators*. Citeseer, University Park, PA, USA.

[215] Robert Tibshirani. 1996. Regression Shrinkage and Selection via the Lasso. *Journal of the Royal Statistical Society: Series B (Methodological)* 58, 1 (1996), 267–288.

[216] Christian Toth, Lars Lorch, Christian Knoll, Andreas Krause, Franz Pernkopf, Robert Peharz, and Julius Von Kügelgen. 2022. Active bayesian causal inference. *Advances in Neural Information Processing Systems* 35 (2022), 16261–16275.

[217] Ruibo Tu, Chao Ma, and Cheng Zhang. 2023. Causal-Discovery Performance of ChatGPT in the context of Neuropathic Pain Diagnosis. arXiv:2301.13819 [cs.CL] https://arxiv.org/abs/2301.13819

[218] 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.

[219] Benito van der Zander, Maciej Liśkiewicz, and Johannes Textor. 2014. Constructing Separators and Adjustment Sets in Ancestral Graphs. In *Proceedings of the UAI 2014 Conference on Causal Inference: Learning and Prediction - Volume 1274* (Quebec City, Canada) *(CI'14)*. CEUR-WS.org, Aachen, DEU, 11–24.

[220] Benito van der Zander, Maciej Liśkiewicz, and Johannes Textor. 2019. Separators and adjustment sets in causal graphs: Complete criteria and an algorithmic framework. *Artificial Intelligence* 270 (2019), 1–40. https://doi.org/10.1016/j.artint.2018.12.006

[221] Alexander van Renen, Dominik Horn, Pascal Pfeil, Kapil Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. 2024. Why TPC is Not Enough: An Analysis of the Amazon Redshift Fleet. *Proc. VLDB Endow.* 17, 11 (July 2024), 3694–3706. https://doi.org/10.14778/3681954.3682031

[222] Aniket Vashishtha, Abbavaram Gowtham Reddy, Abhinav Kumar, Saketh Bachu, Vineeth N Balasubramanian, and Amit Sharma. 2023. Causal Inference Using LLM-Guided Discovery. arXiv:2310.15117 [cs.AI] https://arxiv.org/abs/2310.15117

[223] 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.

[224] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building an elastic query engine on disaggregated storage. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation* (Santa Clara, CA, USA) *(NSDI'20)*. USENIX Association, USA, 449–462.

[225] Dolores R Wallace and Roger U Fujii. 1989. Software verification and validation: an overview. *Ieee Software* 6, 3 (1989), 10–17.

[226] Changzhang Wang, You Zhou, Qiang Zhao, and Zhi Geng. 2014. Discovering and orienting the edges connected to a target variable in a DAG via a sequential local learning approach. *Computational statistics & data analysis* 77 (2014), 252–266.

[227] Yuxuan Wang, Mingzhou Liu, Xinwei Sun, Wei Wang, and Yizhou Wang. 2025. Bayesian active learning for bivariate causal discovery. In *Proceedings of the 42nd International Conference on Machine Learning* (Vancouver, Canada) *(ICML'25)*. JMLR.org, Article 2529, 21 pages.

[228] Zhengjin Wang, Xiaofeng Hu, Haoqiong Bian, Yunda Guo, Jikuan Zhang, and Xiang Zheng. 2026. Demonstrating DBdoctor: A Fine-grained and Non-intrusive Performance Diagnosis Platform for Databases. In *Companion of the International Conference on Management of Data* (India) *(SIGMOD Companion '26)*. Association for Computing Machinery, New York, NY, USA, 130–133. https://doi.org/10.1145/3788853.3801614

[229] Johannes Wehrstein, Roman Heinrich, Mihail Stoian, Skander Krid, Martin Stemmer, Andreas Kipf, Carsten Binnig, and Muhammad El-Hindi. 2025. Redbench: Workload Synthesis From Cloud Traces. arXiv:2511.13059 [cs.DB] https://arxiv.org/abs/2511.13059

[230] Marco A Wiering et al. 2002. Evolving causal neural networks. In *Proceedings of the Twelfth Belgian-Dutch Conference on Machine Learning*. 103–108.

[231] Thomas C. Williams, Cathrine C. Bach, Niels B. Matthiesen, Tine B. Henriksen, and Luigi Gagliardi. 2018. Directed acyclic graphs: a tool for causal studies in paediatrics. *Pediatric Research* 84, 4 (01 Oct 2018), 487–493. https://doi.org/10.1038/s41390-018-0071-3

[232] Wentao Wu, Yun Chi, Hakan Hacıgümüş, and Jeffrey F. Naughton. 2013. Towards predicting query execution time for concurrent and dynamic database workloads. *Proc. VLDB Endow.* 6, 10 (Aug. 2013), 925–936. https://doi.org/10.14778/2536206.2536219

[233] Yinjun Wu, Kwanghyun Park, Rathijit Sen, Brian Kroth, and Jaeyoung Do. 2020. Lessons learned from the early performance evaluation of Intel optane DC persistent memory in DBMS. In *Proceedings of the 16th International Workshop on Data Management on New Hardware* (Portland, Oregon) *(DaMoN '20)*. Association for Computing Machinery, New York, NY, USA, Article 14, 3 pages. https://doi.org/10.1145/3399666.3399898

[234] Ziniu Wu, Ryan Marcus, Zhengchun Liu, Parimarjan Negi, Vikram Nathan, Pascal Pfeil, Gaurav Saxena, Mohammad Rahman, Balakrishnan Narayanaswamy, and Tim Kraska. 2024. Stage: Query Execution Time Prediction in Amazon Redshift. In *Companion of the 2024 International Conference on Management of Data* (Santiago, AA, Chile) *(SIGMOD/PODS '24)*. Association for Computing Machinery, New York, NY, USA, 280–294. https://doi.org/10.1145/3626246.3653391

[235] Ziniu Wu, Markos Markakis, Chunwei Liu, Peter Baile Chen, Balakrishnan Narayanaswamy, Tim Kraska, and Samuel Madden. 2025. Improving DBMS Scheduling Decisions with Fine-grained Performance Prediction on Concurrent Queries – Extended. arXiv:2501.16256 [cs.DB] https://arxiv.org/abs/2501.16256

[236] 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)*. USENIX Association, Berkeley, CA, USA, 217–231.

[237] 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, Piscataway, NJ, USA, 1346–1351. https://doi.org/10.1109/ICDM50108.2020.00175

[238] 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.

[239] Pengcheng Xiong, Yun Chi, Shenghuo Zhu, Hyun Jin Moon, Calton Pu, and Hakan Hacıgümüş. 2011. Intelligent management of virtualized resources for database systems in cloud environment. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering (ICDE '11)*. IEEE Computer Society, USA, 87–98. https://doi.org/10.1109/ICDE.2011.5767928

[240] Pengcheng Xiong, Yun Chi, Shenghuo Zhu, Junichi Tatemura, Calton Pu, and Hakan Hacıgümüş. 2011. ActiveSLA: a profit-oriented admission control framework for database-as-a-service providers. In *Proceedings of the 2nd ACM Symposium on Cloud*

Computing (Cascais, Portugal) *(SOCC '11)*. Association for Computing Machinery, New York, NY, USA, Article 15, 14 pages. https://doi.org/10.1145/2038916.2038931

[241] 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* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 117–132. https://doi.org/10.1145/1629575.1629587

[242] Lingsen Yan, Bolong Zheng, Junjie Qing, Wenlong You, Tingyang Chen, Zhi Xu, Shuncheng Liu, Kai Zeng, Tao Ye, and Xiaofang Zhou. 2025. Anomaly Diagnosis with Siamese Discrepancy Networks in Distributed Cloud Databases. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. IEEE, Piscataway, NJ, USA, 4053–4065. https://doi.org/10.1109/ICDE65448.2025.00302

[243] Zihan Yan, Rui Xi, and Mengshu Hou. 2025. MCTuner: Spatial Decomposition-Enhanced Database Tuning via LLM-Guided Exploration. *Proc. ACM Manag. Data* 3, 6, Article 342 (Dec. 2025), 25 pages. https://doi.org/10.1145/3769807

[244] Xinjun Yang, Yingqiang Zhang, Hao Chen, Feifei Li, Gerry Fan, Yang Kong, Bo Wang, Jing Fang, Yuhui Wang, Tao Huang, Wenpu Hu, Jim Kao, and Jianping Jiang. 2025. Unlocking the Potential of CXL for Disaggregated Memory in Cloud-Native Databases. In *Companion of the 2025 International Conference on Management of Data* (Berlin, Germany) *(SIGMOD/PODS '25)*. Association for Computing Machinery, New York, NY, USA, 689–702. https://doi.org/10.1145/3722212.3724460

[245] Jianxin Yin, You Zhou, Changzhang Wang, Ping He, Cheng Zheng, and Zhi Geng. 2008. Partial orientation and local structural learning of causal networks for prediction. In *Proceedings of the Workshop on the Causation and Prediction Challenge at WCCI 2008 (Proceedings of Machine Learning Research)*, Vol. 3. PMLR, Cambridge, MA, USA, 93–105. http://proceedings.mlr.press/v3/yin08a.html

[246] Brit Youngmann, Michael Cafarella, Babak Salimi, and Anna Zeng. 2023. Causal Data Integration. *Proc. VLDB Endow.* 16, 10 (jun 2023), 2659–2665. https://doi.org/10.14778/3603581.3603602

[247] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX symposium on operating systems design and implementation (OSDI 22)*. USENIX Association, Berkeley, CA, USA, 521–538.

[248] Geoffrey X. Yu, Markos Markakis, Andreas Kipf, Per-Åke Larson, Umar Farooq Minhas, and Tim Kraska. 2022. TreeLine: an update-in-place key-value store for modern storage. *Proc. VLDB Endow.* 16, 1 (Sept. 2022), 99–112. https://doi.org/10.14778/3561261.3561270

[249] Geoffrey X. Yu, Ziniu Wu, Ferdi Kossmann, Tianyu Li, Markos Markakis, Amadou Ngom, Samuel Madden, and Tim Kraska. 2024. Blueprinting the Cloud: Unifying

and Automatically Optimizing Cloud Data Infrastructures with BRAD. *Proc. VLDB Endow.* 17, 11 (Aug 2024), 3629–3643. https://doi.org/10.14778/3681954.3682026

[250] Geoffrey X. Yu, Ziniu Wu, Ferdi Kossmann, Tianyu Li, Markos Markakis, Amadou Ngom, Sophie Zhang, Tim Kraska, and Samuel Madden. 2025. Virtualizing Cloud Data Infrastructures with BRAD. In *Companion of the 2025 International Conference on Management of Data* (Berlin, Germany) *(SIGMOD/PODS '25)*. Association for Computing Machinery, New York, NY, USA, 271–274. https://doi.org/10.1145/3722212.3725141

[251] Changhe Yuan and Brandon Malone. 2013. Learning optimal Bayesian networks: A shortest path perspective. *Journal of Artificial Intelligence Research* 48 (2013), 23–65.

[252] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. 2010. SherLog: error diagnosis by connecting clues from run-time logs. *SIGARCH Comput. Archit. News* 38, 1 (mar 2010), 143–154. https://doi.org/10.1145/1735970.1736038

[253] 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.

[254] Benito van der Zander and Maciej Liśkiewicz. 2016. Separators and adjustment sets in Markov equivalent DAGs. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence* (Phoenix, Arizona) *(AAAI'16)*. AAAI Press, 3315–3321.

[255] 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.

[256] Yan Zeng, Ruichu Cai, Fuchun Sun, Libo Huang, and Zhifeng Hao. 2023. A Survey on Causal Reinforcement Learning. arXiv:2302.05209 [cs.AI]

[257] Huanchen Zhang, Yihao Liu, and Jiaqi Yan. 2023. Cost-Intelligent Data Analytics in the Cloud. arXiv:2308.09569 [cs.DB] https://arxiv.org/abs/2308.09569

[258] 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* 35, 8 (2023), 8596–8614. https://doi.org/10.1109/TKDE.2022.3222417

[259] William Zhang, Wan Shen Lim, and Andrew Pavlo. 2026. This is Going to Sound Crazy, But What If We Used Large Language Models to Boost Automatic Database Tuning Algorithms By Leveraging Prior History? We Will Find Better Configurations More Quickly Than Retraining From Scratch! *Proc. ACM Manag. Data* 4, 1, Article 90 (April 2026), 29 pages. https://doi.org/10.1145/3786704

[260] Xinyi Zhang, Tiantian Chen, Zhentao Han, Zhaoyan Hong, Wei Lu, Sheng Wang, Mo Sha, Anni Wang, Shuang Liu, Yakun Zhang, Feifei Li, and Xiaoyong Du. 2026. Why Database Manuals are Not Enough: Efficient and Reliable Configuration Tuning for DBMSs via Code-Driven LLM Agents. *Proc. VLDB Endow.* 19, 6 (May 2026), 1358–1371. https://doi.org/10.14778/3797919.3797940

[261] Xinyi Zhang, Hong Wu, Zhuo Chang, Shuowei Jin, Jian Tan, Feifei Li, Tieying Zhang, and Bin Cui. 2021. ResTune: Resource Oriented Tuning Boosted by Meta-Learning for Cloud Databases. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2102–2114. https://doi.org/10.1145/3448016.3457291

[262] Xukang Zhang, Huanchen Zhang, and Xiaofeng Meng. 2025. Intra-Query Runtime Elasticity for Cloud-Native Data Analysis. *Proc. ACM Manag. Data* 3, 3, Article 178 (June 2025), 28 pages. https://doi.org/10.1145/3725315

[263] Liang Zhao, Sherif Sakr, and Anna Liu. 2013. A framework for consumer-centric SLA management of cloud-hosted databases. *IEEE Transactions on Services Computing* 8, 4 (2013), 534–549.

[264] Yue Zhao, Gao Cong, Jiachen Shi, and Chunyan Miao. 2022. QueryFormer: a tree transformer model for query plan representation. *Proc. VLDB Endow.* 15, 8 (April 2022), 1658–1670. https://doi.org/10.14778/3529337.3529349

[265] 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

[266] Yujia Zheng, Biwei Huang, Wei Chen, Joseph Ramsey, Mingming Gong, Ruichu Cai, Shohei Shimizu, Peter Spirtes, and Kun Zhang. 2024. Causal-learn: Causal Discovery in Python. *Journal of Machine Learning Research* 25, 60 (2024), 1–8. http://jmlr.org/papers/v25/23-0970.html

[267] 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, Piscataway, NJ, USA, 572–577. https://doi.org/10.1109/DSN.2009.5270289

[268] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. DistServe: disaggregating prefill and decoding for goodput-optimized large language model serving. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation* (Santa Clara, CA, USA) *(OSDI'24)*. USENIX Association, USA, Article 11, 18 pages.

[269] Wei Zhou, Peng Sun, Xuanhe Zhou, Qianglei Zang, Ji Xu, Tieying Zhang, Guoliang Li, and Fan Wu. 2026. DBAIOps: A Reasoning LLM-Enhanced Database Operation and Maintenance System using Knowledge Graphs. *Proc. VLDB Endow.* 19, 6 (May 2026), 1319–1331. https://doi.org/10.14778/3797919.3797937

[270] Xuanhe Zhou, Guoliang Li, Zhaoyan Sun, Zhiyuan Liu, Weize Chen, Jianming Wu, Jiesi Liu, Ruohang Feng, and Guoyang Zeng. 2024. D-Bot: Database Diagnosis System using Large Language Models. *Proc. VLDB Endow.* 17, 10 (June 2024), 2514–2527. https://doi.org/10.14778/3675034.3675043

[271] Xuanhe Zhou, Ji Sun, Guoliang Li, and Jianhua Feng. 2020. Query performance prediction for concurrent queries using graph embedding. *Proc. VLDB Endow.* 13, 9 (May 2020), 1416–1428. https://doi.org/10.14778/3397230.3397238

[272] Jiongli Zhu, Sainyam Galhotra, Nazanin Sabri, and Babak Salimi. 2023. Consistent Range Approximation for Fair Predictive Modeling. arXiv:2212.10839 [cs.LG] https://arxiv.org/abs/2212.10839

[273] Jieming Zhu, Shilin He, Pinjia He, Jinyang Liu, and Michael R. Lyu. 2023. Loghub: A Large Collection of System Log Datasets for AI-driven Log Analytics. arXiv:2008.06448 [cs.SE] https://arxiv.org/abs/2008.06448

[274] 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 *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice* (Montreal, Quebec, Canada) *(ICSE-SEIP '19)*. IEEE, Piscataway, NJ, USA, 121–130. https://doi.org/10.1109/ICSE-SEIP.2019.00021

[275] Shengyu Zhu, Ignavier Ng, and Zhitang Chen. 2020. Causal Discovery with Reinforcement Learning. arXiv:1906.04477 [cs.LG] https://arxiv.org/abs/1906.04477

[276] 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.

[277] 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* (Los Angeles, CA, USA) *(SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 362–375. https://doi.org/10.1145/3098822.3098849