

UCLA

UCLA Electronic Theses and Dissertations

Title

Towards Cloud-Scale Debugging

Permalink

<https://escholarship.org/uc/item/3783653w>

Author

Dogga, Pradeep

Publication Date

2024

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA
Los Angeles

Towards Cloud-Scale Debugging

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Pradeep Dogga

2024

© Copyright by

Pradeep Dogga

2024

ABSTRACT OF THE DISSERTATION

Towards Cloud-Scale Debugging

by

Pradeep Dogga

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2024

Professor Ravi Arun Netravali, Co-Chair

Professor George Varghese, Co-Chair

Cloud computing is an integral part of today's world: it primarily enables individuals and enterprises to provision and manage resources such as compute, storage, etc., for their needs with the click of a button. Modular approach to software development enabled cloud providers to rapidly evolve and deliver increasing number of services to users rendering clouds mission-critical. To insure prompt serviceability of this Achilles' Heel from facing incidents, cloud providers employ significant human resources. However, with the ever increasing number of services offered by clouds and growing types of workloads such as the proliferation of Machine Learning workloads in recent times, it is no longer viable for cloud providers to scale their human resources at this pace to insure prompt serviceability of their clouds.

In this dissertation, I present my work towards improving the serviceability of clouds by leveraging insights from my experience with real debugging workflows employed at the three largest clouds today. I present techniques from Machine Learning and Natural Language Processing to leverage the vast amount of historical debugging data in clouds to develop tools that provide assistance to their engineers. I present a 'Coarsening' framework that enables

transition towards a centralized debugging plane and discuss practical evaluations of tools built using this framework.

I present REVELIO, a tool that can generate debugging queries for engineers to execute over system-wide logged data, whose results can likely hint them of the root cause of an incident. To enable benchmarking many techniques, I also built a distributed systems debugging testbed that can inject faults into services, interface with human users and collect execution logs across the system. I present AUTOARTS, a tool that can tag a lengthy postmortem report of an incident in the cloud with all root causes from an extensive taxonomy and can also highlight key pieces of information from a postmortem for ease of analysis. I present PERFRCA, a tool that can scale causal discovery to production-scale telemetry to reason performance degradations. I conclude with my vision for a centralized approach to automatically extract generalizable debugging assistance to engineers across a cloud.

The dissertation of Pradeep Dogga is approved.

Omid Salehi-Abari

Harry Guoqing Xu

Suman Nath

Ravi Arun Netravali, Committee Co-Chair

George Varghese, Committee Co-Chair

University of California, Los Angeles

2024

To my grandfather, B. Simhachalam

TABLE OF CONTENTS

1	Introduction	1
1.1	Thesis: Enabling Cloud-Scale Centralized Debugging via Coarsening	5
1.2	Debugging Lifecycle	6
1.3	Limitations of Team Level Debugging	9
1.4	Thesis Contributions	10
2	Towards Cloud-Scale Debugging via Coarsening	13
2.1	Coarsening	14
2.1.1	Coarsening Formulation - Incident \rightarrow Alarms	20
2.1.2	Coarsening Formulation - Alarms \rightarrow Team	21
2.1.3	Coarsening Formulation - Team \rightarrow Query	22
2.1.4	Coarsening Formulation - Query \rightarrow Events	23
2.1.5	Coarsening Formulation - Document \rightarrow Label(s)	24
2.2	Retrospective Analysis	25
2.2.1	Quick Fixes	26
2.2.2	Consistent Root-Cause Labelling	27
2.3	Diagnosability	28
2.3.1	Coarse Dependency Graphs	30
2.3.2	Incident Routing	30
2.3.3	Root-Cause Analysis	31
2.3.4	Preliminary Evaluation - Incident Routing	32
2.3.5	Preliminary Evaluation - Root Cause Analysis	32

2.3.6	Coarse Log Summarization using Chains	33
2.3.7	Preliminary Evaluation - Log Summarization	35
2.4	Observability	36
2.4.1	Monitoring	36
2.4.2	Data Retention	36
2.5	Centralized Debugging Plane	37
2.5.1	Global data lake	38
2.5.2	Incident data store	38
2.5.3	AIOps engine	39
2.5.4	Incremental deployment of CDP	39
3	NLP-Powered Debugging Assistance	40
3.1	Auxiliary Data	40
3.2	NLP Powered System-Wide Debugging Assistant	41
3.2.1	Opportunities for Automation	43
3.3	Preliminary Experiments	44
3.3.1	Label Prediction for GitHub Issues	45
3.3.2	Source Code Folder Prediction for GitHub Fixes	46
3.3.3	Debugging Query Generation	47
3.3.4	Results	48
3.4	Related Work	49
3.4.1	Program analysis and synthesis	50
3.4.2	Program debugging	50
3.4.3	Big Code	51

4	Generating Debugging Queries	52
4.1	Debugging Queries	53
4.2	Overview - REVELIO	54
4.2.1	Challenges	54
4.2.2	Solutions	55
4.3	REVELIO's ML Model	58
4.3.1	Predicting Probabilities for Query Templates	59
4.3.2	Predicting Values to Fill Query Templates	61
4.3.3	Choosing the Final Queries	62
4.3.4	Diagrams Illustrating Model Operation/Insights	63
4.4	Study of Production Incident Debugging at <i>Anon1</i>	65
4.4.1	Insights from Debugging Workflows	65
4.4.2	Insights from Production Incidents	66
4.4.3	Literature Survey of Incidents	66
4.5	Distributed Systems Debugging Testbed	69
4.5.1	Single-machine Emulation of Distibuted Applications	70
4.5.2	Overview of Applications	72
4.5.3	Overview of Debugging Tools	73
4.5.4	Fault Injection Service	76
4.5.5	Dataset Collection using AWS MTurk	76
4.6	Evaluation of REVELIO	80
4.6.1	REVELIO's Performance on Repeat Faults	80
4.6.2	REVELIO's Performance on New Faults	80

4.6.3	Understanding REVELIO’s Performance	81
4.6.4	Developer Study	85
4.7	Related Work	87
4.7.1	Debugging Tools for Distributed Systems	87
4.7.2	Leveraging Natural Language Data Sources	87
5	Automated Root-Cause Labelling	89
5.1	Incident Postmortem Reports	90
5.2	Overview - Root Cause Labelling	91
5.2.1	Challenges	92
5.2.2	Solutions	94
5.3	Analysis of Production Incidents at Microsoft Azure	95
5.3.1	Manual Analysis of High-Impact Incidents	96
5.3.2	Findings from Empirical Analysis of Incidents	97
5.4	ARTS Root Cause Taxonomy	102
5.5	AUTOARTS’s ML Models	104
5.5.1	Identifying Root Cause Labels from the ARTS Taxonomy	105
5.5.2	Extracting Root Cause Context from Postmortems	107
5.6	Evaluation of AUTOARTS	109
5.6.1	Methodology	109
5.6.2	Featurization	110
5.6.3	AUTOARTS’s Performance on Root Cause Labelling	111
5.6.4	AUTOARTS’s Performance on Context Extraction	112
5.6.5	User Study	113

5.7	Related Work	115
5.7.1	Root-cause analysis of past incidents	115
5.7.2	Text summarization & root-cause classification	115
6	Coarse Causal Reasoning in Telemetry	117
6.1	Causal Discovery for Root Cause Analysis	117
6.2	Overview - Reasoning Slow Queries in <i>Anon2</i>	118
6.2.1	Challenges	118
6.3	Limitations of Causal Discovery	120
6.4	PERFRCA	121
6.4.1	Feature Computation	122
6.4.2	Anomaly Extraction	125
6.4.3	Causal Discovery	125
6.5	Preliminary Results	126
6.5.1	Case Studies	126
7	Conclusion	128
7.1	Future Work and Open Problems	129
7.1.1	Evaluation on Production Systems	130
7.1.2	Impact of LLMs on Debugging	131
	References	132

LIST OF FIGURES

1.1	The debugging lifecycle of incidents in production distributed systems.	7
2.1	Learned coarsening function that maps a concrete element (root cause of an incident) to an abstract element (e.g., root cause label)	16
2.2	Implicit concretization of concrete element through assistance from abstract element (debugging tool’s output) forming a galois connection	17
2.3	Coarsening in stages to iteratively scale down manual work and assist engineers in debugging tasks. Dashed arrows indicate hard manual labor that engineers would need to do without any assistance and solid arrows indicate automated completion of or assistance for one debugging task, progressing engineers to the next task. The size of each oval blob indicates the amount of work to be done and it’s operational scale is indicated below.	18
2.4	CDF of mitigation times of over 2000 incidents from our manual study at <i>Anon</i> (TTM over 24 hrs depicted as 24 hrs).	26
2.5	Health checks for reachability in a cloud network. Arrows indicate reachability probes within clusters, fabrics, and across WANs.	30
2.6	Coarse dependency graph for Reddit in our distributed systems testbed.	31
2.7	Overview of our approach to detect anomalous execution during runtime for root cause analysis.	34

3.1	Overview of how our proposed debugging assistant improves different steps in a developer’s end-to-end debugging workflow. Developers begin by submitting system-wide bugs or performance concerns to the NL debugging assistant. The assistant generates hints (e.g., files to investigate) or actions (e.g., debugging queries to issue) based on analyzing past bug report data, design documents, tracing information collected throughout the system, and developer input. The process is iteratively followed until a bug is resolved.	42
3.2	Examples of label & folder predictions for two repos: hashicorp/terraform and angular/material2.	49
3.3	Example query predictions for debugging a given issue.	49
4.1	Example of a debugging query in it’s syntax tree representation issued over Marple’s switch queue depth counters from two switches ‘b1’ and ‘b2’.	53
4.2	REVELIO as an implementation of <i>Coarsening</i> through Debugging Queries.	55
4.3	Overview of REVELIO’s factorized, 2-phase approach to generating debugging queries for root cause diagnosis.	56
4.4	Example showing how rank-ordering helps to generalize to faults of the same type at different locations. After ordering (right), despite the fault location being different, the queue depth order statistics in testing are correlated with those in training. In contrast, without ordering (left), the unseen fault location results in queue depth values that are dissimilar from training data.	57
4.5	Example illustrating the generation of system log vector L ; for simplicity, the example considers only network logs. Values for each feature (across switches) are first rank ordered, and then the resulting lists are concatenated to form L	63

4.6	Example inputs for each input variable in REVELIO’s model. This example is for a network (Marple) query. For the query template (T), the entire tree represents the template, while the parameters to be filled in are shaded in grey.	64
4.7	A slice (2/14 microservices) of our testbed for Sock Shop [Wea17]. Debugging tools and fault injection are omitted.	69
4.8	The topology of our distributed systems testbed for Reddit [red22]. Each P4 switch has a congestion traffic sender/receiver to emulate different network conditions, and the testbed incorporates four recent debugging tools and a fault injection service. We illustrate the Sock Shop [Wea17] topology in Figure 4.7, and note that Online Boutique [Goo19] follows the same architectural patterns.	71
4.9	User Interface of our MTurk experiment presented to users for data collection. .	78
4.10	Cumulative distribution (per app) of the rank of the correct query over our test set of <i>repeat</i> faults.	81
4.11	Cumulative distribution (per app) of the rank of the correct query over our test set of <i>previously unseen</i> faults.	81
4.12	Comparing the average rank for single-tool and multi-tool versions of REVELIO. Results are for Reddit.	83
4.13	Top-5 query accuracy when training REVELIO on random subsets of the data. Results are for Reddit.	84
4.14	Summary of time saved in debugging each fault in our developer study. Bars represent average time spent across all developers who correctly identified the root cause.	86
5.1	Redacted example of an inident postmortem report from a service in Microsoft Azure.	90

5.2	Labelling incident postmortem reports documented by several engineers in a cloud enables owners to extract insights with meaningful analyses.	91
5.3	AUTOARTS as an implementation of <i>Coarsening</i> through root cause labels from ARTS taxonomy.	92
5.4	Overview of our Context Extraction and Hierarchical Root-cause Classification using Post-Incident reports.	95
5.5	Distribution of incidents across number of distinct contributing factors (shown until 10 factors).	98
5.6	Average number of incidents successfully tagged until a new root cause tag is introduced across quarters.	99
5.7	ARTS taxonomy visualized with partially expanded root cause categories. Left end of an edge indicates the parent root cause category and the right end indicates finer subcategory within the parent.	103
5.8	Context extraction from a redacted PIR. Green sentences are extracted by both our model and human expert, red are extracted by model only and blue are extracted by human only.	108
6.1	Overview of PERFRCA as dynamic causal discovery over anomalous features from a large set of telemetry of slow queries.	122
6.2	Output of PERFRCA for a WLM misclassification.	127
6.3	Output of PERFRCA for a overload in system due to high number of prefetches from S3.	127

LIST OF TABLES

2.1	Top-5 contributing factors across 450+ cloud services at Microsoft Azure and the corresponding quick fixes.	25
2.2	Comparing SDN and CDP architectures	38
3.1	Results for label and folder prediction, as well as template-based query generation tasks.	48
4.1	Variables in Revelio’s ML model. Figure 4.6 lists example input values for each.	58
4.2	Summary of closed debugging tickets at <i>Anon1</i> over a 4-month period. Examples have been partially Anonymized and summarize the root causes listed in representative tickets. Debugging times are in minutes.	67
4.3	Overview of faults injected into our distributed systems testbed. Numbers listed are for Sock Shop[Wea17].	77
4.4	Summary of debugging queries and user reports collected through AWS MTurk experiment.	77
4.5	Metrics in system logs. Marple, Jaeger, and cAdvisor metrics are recorded per-switch, per-function, and per-container; tcpdump is omitted for space.	79
4.6	Examples of text in user reports collected from Mechanical Turk participants.	79
4.7	Impact of different input sources on REVELIO’s performance. Results list avg rank (% in top-5) and are for Reddit.	82
4.8	Comparison with simpler ML approaches. Results list avg rank (% in top-5) for Reddit.	82

4.9	REVELIO’s performance when metrics from system logs are selectively removed. Removed Marple, Jaeger, and cAdvisor features are shown in blue, red, and grey, respectively. Results list avg rank (% in top-5) and are for Reddit.	83
5.1	High-level root cause categories from ARTS taxonomy with their descriptions, frequency of occurrence in our analysis and mean Time-To-Mitigate (TTM) for incidents caused by their sub-categories.	97
5.2	Distribution of top 10 most frequent contributing factors in our analysis from the ARTS taxonomy.	101
5.3	Study on the utility of different PIR sections in top-level root cause classification using Random Forests.	110
5.4	Performance of HiAGM compared to using flattened root cause taxonomy (HiAGM_Flat) and a finetuned-BERT based multilabel classifier (BERT_MLC).	111
5.5	Performance of Pegasus and T5 models. (P) indicates Pre-trained versions and (F) indicates fine-tuned versions. We also present performance of unsupervised clustering based approach for extractive summarization using BERT.	112
5.6	Quantitative user feedback from an expert over the effectiveness of AUTOARTS across context generation and root cause classification tasks over a randomly chosen set of 10 incidents.	113
6.1	Time-To-Finish for Causal Discovery on a controlled synthetic dataset using the PC algorithm on a 32 core machine with a 24hr timeout.	121

ACKNOWLEDGMENTS

As a Ph.D. student, I was very fortunate to work with two outstanding advisors, Prof. George Varghese and Prof. Ravi Netravali. I joined UCLA solely because of an offer from George to be my advisor and I only met Ravi, who had just started his tenure at UCLA, at a water filling station near our offices for the first time. Based on our mutual research interests, Ravi offered to co-advise me with George, who graciously agreed to it. Looking back, this decision was super critical for shaping my perspectives as both George and Ravi had their own unique shares in them.

No words can do justice to express my gratitude towards George. George is a master of conducting inter-disciplinary research with unique insights that enriches both disciplines, a rare feat I have seen researchers achieve. He taught me to see the big picture while handling the minute details and greatly helped me shape ideas. George is a true ‘enabler’ by all means and he showed me the power of deep meaningful collaborations that have greatly benefited me. Through his lectures at UCLA, George showed me the positive impact a teacher can have on students and as a result contributed to my passion towards teaching. He taught me to be brave in questioning the obvious, to be patient in meticulous work and most importantly, to be empathetic. He encouraged me to work with cloud providers to gain practical experience at a time when I didn’t know it’s value and it significantly benefited my thesis. Despite circumstances, George truly stood by me in times of need and for everything he has done for my growth, I am forever indebted to him.

Ravi and I spent countless hours together at UCLA working on research papers. Ravi showed me the impact of time management by being a true example (not sure if can ever match it), he was in his office before anyone everyday. He was always available to discuss anything I wanted to with the same enthusiasm. Ravi is master of asking the right questions with an impeccable critical analysis and suggesting the necessary details which specially helped me with my naive optimism and with concretizing system-building ideas. He taught me

all the work it takes to deliver a system which has shaped all of my work. Most importantly, Ravi taught me how to present my work to an audience with great patience and solely shaped my perspective towards presentations and communication in general. I am immensely grateful to Ravi for always being there and everything he contributed for my growth.

I am deeply grateful for my thesis committee members Harry Xu and Omid Abari for their feedback and support. This thesis is a result of joint work with many collaborators: George Varghese, Ravi Netravali, Suman Nath, Anirudh Sivaraman, Karthik Narasimhan, Chetan Bansal, Xuchao Zhang, Shiv Saini, Jens Palsberg, Subrata Mitra, Zhengchun Liu, Ayush Chauhan, Mayur Patel, Nathaniel Flath. Suman was my manager at Microsoft during my internship. Despite his workload, Suman quickly understood my thesis focus and always made sure the work I do is both impactful and beneficial for my Ph.D. thesis study. Suman taught me a great deal in understanding the practical challenges faced in industry which was priceless. Anirudh helped a great deal in shaping my initial vision for my thesis work and taught me how to conduct experiments and evaluate the practicality of approaches to be deployed in the real world. He also gave me much needed guidance on navigating through career decisions and supported me when I needed it. Karthik offered great support in learning advanced techniques in the Machine Learning, Natural Language Processing domains and was a very important part in shaping my vision for debugging powered by Machine Learning.

Shiv was my first mentor who taught me how to search for and read research papers, how to brainstorm systematically, how to present my work and conduct experiments which fundamentally helped me develop a passion for research. I am greatly indebted to Shiv for being my first mentor. Subrata was instrumental in forming my perspectives towards impactful research work in systems through his mentorship prior to my graduate study.

My UCLA colleagues past and present, Siva, Arjun, Aishwarya, Murali, Akshay, Zeina, Shuyang, Ana, Micky, Proova, Neil, Shagha, Lana, Tianyi, Muhammad, Navjot, John Bender, Christian, Arthi, John Thorpe, Jon, Yifan, Jean, Angelica, Alan gave me good reason to come to work, to bounce off ideas, learn new concepts from different disciplines, get feedback

on papers and talks, chat about things going on in our lives, organize potlucks, get ramen from Killer Noodles and play an afternoon game of foosball. Siva has been a great friend and mentor since my days at IIT, his guidance was very helpful for navigating through graduate school and beyond. Arjun has been a wonderful roommate and I am very grateful for his advice, movie and game nights during the pandemic. Aishwarya and Murali were gracious to offer me housing during the pandemic when moving was difficult. Siva, Arjun and Aish cooked me many yummy meals over the years which I will cherish. Akshay was always there for game nights every week to get through school. John organized many squash game nights at UCLA which were fun. Arthi got all our group together by hosting friendsgiving and potlucks. Ana, Poorva, Micky, Zeina, Shuyang and Akshay provided me with many fun breaks. Poorva specially, helped me with my moves between apartments. Trips with my friends Sohan and Ritam were much needed and are so memorable.

The students in my discussion section of CS 118 in Fall of 2020 were incredible and helped me learn a lot about teaching that I will forever cherish. Joseph Brown was a phenomenal student affairs officer who gave me every possible way out of deadlines, penalties and making sure I get things done. Edna Todd was a great administrative assistant and was super patient in handling every purchase, reimbursement and helped me greatly in following guidelines. Curtis offered me a great deal of help with finding housing by going above and beyond his duty.

I am grateful for my life in LA during graduate school. I never thought about the impact of locale when I joined UCLA. Driving through the winding Pacific Coast Highway, mesmerized by the magnificent giant sequoia groves, getting lost in the Sierra Nevadas, star gazing in the deserts and rides on my motorcycle to the Hindu temple in the canyons of Malibu for prayers, all left a profound appreciation in me for life, nature and freedom.

My parents Jayalakshmi and Lakshun Naidu, are my constant support. They check in on me everyday and cheered on for my highs and put up with my lows. I am beyond grateful for their care for my well-being without which I would be hopeless. My brother Sandeep is

pivotal in managing things for me that I would not be able to do so otherwise. My cousins Chaitu, Vinnu, Ramya and Sujith kept me going with much needed weekly group calls. My grandmother who is so curious about my life kept me appreciative of the things I would be doing and her child-like innocence is so refreshing and much needed. Swetha, the love of my life, was the greatest finding during graduate school and she has been through everything along with me. I am grateful for her for proofreading my paper drafts, organizing food for my thesis defense, giving me useful feedback for my pitches, helping me take care of my health during submission deadlines, believing in me for some reason and giving me strength. I am also grateful for her family for accepting me as one of their own and giving me a sense of purpose and belonging far away from my family.

My grandfather, B. Simhachalam, is the sole reason I strive to grow every day. He taught me to brave, to stand up for the things I believe in and to be righteous. The examples he set are inspiring and noble, be it his commitment to delivering quality education to underprivileged students as a teacher, be it his relentless discipline, or the many things he has done for me and our family. His persistence for my growth and well-being is infectious and unbelievable. Hearing his life experiences gave me a rooted perspective that guides me in various aspects of life. For being my inspiration, for directly and indirectly eliminating obstacles and paving the way forward for me, I dedicate this thesis to him.

VITA

- 2014 Inspire Fellowship, from Govt of India.
- 2016 Best Intern, Smartron, Hyderabad, India.
- 2017 Best Overall Project Award, Adobe Research, Bengaluru, India.
- 2018 B. Tech. in Computer Science and Engineering, IIT Kharagpur.
- 2018 - 2019 Graduate Division Fellowship, UCLA.
- 2019 Member of the System Demonstrations Program Committee, NAACL-HLT.
- 2020 Teaching Assistant (“Computer Network Fundamentals” course), Computer Science Department, UCLA.
- 2021 Research Intern, Google, California.
- 2021 Student Researcher, Google, California.
- 2022 M. S. in Computer Science, UCLA.
- 2022 Research Intern, Microsoft Research, Redmond.
- 2023 Applied Scientist Intern, Amazon Web Services (AWS), California.
- 2024 Teaching Assistant (“Computer Network Fundamentals” course), Computer Science Department, UCLA.
- 2024 Teaching Assistant (“Software Construction” course), Computer Science Department, UCLA.

PUBLICATIONS

Pradeep Dogga, Karthik Narasimhan, Anirudh Sivaraman and Ravi Netravali. “A System-Wide Debugging Assistant Powered by Natural Language Processing.” In Proceedings of the 10th ACM Symposium on Cloud Computing (SoCC), pp. 171-177. 2019.

Pradeep Dogga, Karthik Narasimhan, Anirudh Sivaraman, Shiv Saini, George Varghese, and Ravi Netravali. “REVELIO: ML-Generated Debugging Queries for Finding Root Causes in Distributed Systems.” In Proceedings of the 4th Machine Learning and Systems Conference (MLSys), pp. 601-622. 2022.

Pradeep Dogga, Chetan Bansal, Richard Costleigh, Gopinath Jayagopal, Suman Nath and Xuchao Zhang. “AUTOARTS: Taxonomy, Insights and Tools for Root Cause Labelling of Incidents in Microsoft Azure.” In Proceedings of the 30th USENIX Annual Technical Conference (USENIX ATC), pp. 359-372. 2023.

CHAPTER 1

Introduction

Dubbed as the modern day equivalent of Robin Hood [Far12], cloud computing has become an integral part of our lives by democratizing computing for users. With cloud computing, users can provision resources (e.g., servers, storage, network) per their demand at the click of a button. This eliminates the need to purchase resources, configure, maintain and dispose of these resources. This empowers low-income individuals entering into global markets with the necessary infrastructure, thereby playing an important role in bridging socio-economic inequalities [CK12]. It also offers several other benefits such as significantly reducing energy consumption, waste and greenhouse gas emissions [Wal21] as compared to users using physical computers, and has helped accelerate vital vaccine research and vaccination management [Dcr22, Bri21, Lan21] particularly during the Covid-19 pandemic, among other grand societal challenges.

Despite being the enabler of myriad applications, clouds are also the *Achilles' Heel* of many services that rely on clouds delivering mission-critical performance, high availability and reliability. This is because clouds suffer from several major incidents [Sec22, Goo23, LLM19b, Mac22] that cause gaps in their availability and performance. Today cloud providers rely on human resources to insure prompt serviceability of their services. Unfortunately, this means that cloud providers have to continue scaling human resources to keep up with the pace of increasing and every changing workloads, like the proliferation of Machine Learning workloads in the recent times. It is no longer viable for cloud providers to rely on human resources to insure clouds deliver mission-critical performance, availability and reliability.

In this thesis, I posit that high impact incidents still plague clouds today because engineers lack a holistic understanding of all the components that comprise a large system, which is a requirement for successfully debugging incidents. Engineers today use their hard-won intuitions to debug production incidents manually: this is error-prone and time-consuming [DNS22, LLM19c, GYM20]. Curiously, this is due to the classic modular approach to developing services which enables cloud providers to rapidly and continuously offer new product features. Such modular approaches allow thousands of engineers to work concurrently on pieces of the system and speeds development; unfortunately, at the same time, as I will argue later, this modular approach causes engineers to develop tunnel-vision.

But can classic reliability techniques – testing, formal verification, tracing, and learning from past incidents – either avoid incidents in clouds or help debug them easily? I argue that they cannot be used conventionally.

Testing: The competitive rat-race between cloud providers (e.g., Microsoft Azure, Google Cloud) forces engineers to rapidly evolve software features; this creates a combinatorial explosion in the number of scenarios to test before production deployment. Foolproof testing of software services is impossible due to manifestation of multiple unique contributing factors that can result in an incident. For example, a VM provisioning service can read a corrupt configuration file, propagate incorrect data across other components and accidentally destroy provisioned VMs, resulting in an incident.

Formal Verification: On the other hand, formal verification methods do not scale either to entire clouds. The increasing diversity in the services offered by clouds such as storage, ML pipelines, analytics, etc., makes formally specifying software behavior practically impossible. The exponentially large domain spaces of inputs, effectively renders formal verification techniques inapplicable to prevent incidents. Each service comprises hundreds of components that interact with each other through loosely-coupled APIs, developed by hundreds of engineers with varying degrees of expertise.

Distributed Tracing and Logging: As services evolve incrementally, different generations of software interact with each other, some of which might not be actively maintained (legacy code) by the original developers, instrumented with different tracing frameworks or completely lacking tracing instrumentation. Re-designing these components to employ a uniform tracing framework is expensive, time-consuming and hence distributed tracing simply doesn't work at cloud scale (e.g., missing correlation Global UIDs in legacy code), resulting in error-prone debugging efforts from engineers during incidents. Unfortunately, there is no effective policy on what information needs to be logged, where in the software to log from and when they should be logged/used. Hence, engineers might log information that is not relevant for debugging which causes performance overheads; worse, they may miss logging vital information that delays debugging incidents.

Learning from the Past: To learn from past debugging experiences, engineers commonly document debugging steps taken to resolve an incident in lengthy natural language postmortem reports. While these are treasure troves of rich insights, they are often not exploited due to time required to manually digest these insights; thus, engineers often resort to hard-won intuitions which are undocumented. When such failure "postmortem" reports are documented, analyzing these reports can inform robust system design principles that can reduce the impact of incidents. Unfortunately, no existing tools leverage postmortems and manually analyzing them doesn't scale at cloud-level. It is important to note that postmortems are documented by thousands of engineers across the cloud, all with varying degrees of expertise, approach to debugging and usage of language. Further, many incidents can go undocumented due to lack of strict policies creating a loss of rich debugging insights to improve cloud serviceability.

Manual Cloud Debugging Today: To overcome these issues, engineers operationalize their undocumented hard-won intuitions in ad hoc ways by organizing conference calls with engineers across different teams to investigate different logs during debugging an incident together, manually defining rules on metrics to alert them of impending incidents, making mental notes of log distributions to attribute to a specific root-cause, etc. The manual and

ad hoc nature of this approach is not viable to improve the serviceability of clouds as their complexity only increases with time, as engineers move across different services with their experience and restarting this painful process elsewhere.

Research in Debugging: Over the past several years, major cloud providers, startups and researchers have investigated a plethora of techniques to improve cloud reliability through formal verification methods [LNF12, PKL09, ADS91], tracing frameworks [MRF15, SBB10, FLM20, FPK07, Sym14], testing strategies [CDE08, MSP18] and post-hoc log analyses [New22, TL18, Twi15, SBN21]. These techniques are commonly inspired by taking a distributed approach, which enables their functionality at small-scale but defeats them at cloud-scale due to lack of effective coordination, making them inefficient and inaccurate.

While distribution can scale, centralization of debugging efforts – as I will argue for in this thesis – does not necessarily mean merely replicating the debugging process at small-scale. Rather, I argue for a shift in perspective which can also enable building effective tools. Further, the classic distributed approach found in the literature also results in building tools limited to specific types of bugs (e.g., performance, configurations, etc.). Unfortunately, this causes significant overhead in deploying hundreds of specific tools to assist engineers in debugging for *rare* wins (as we will see below from our data), making them impractical. Finally, the distributed approach also results in newly emerging services not leveraging the collective debugging wisdom accumulated by past services and different teams, resulting in repeated incidents with similar causes and redundant effort.

1.1 Thesis: Enabling Cloud-Scale Centralized Debugging via Coarsening

To scale, debugging tools and practices today mimic the modular approach used in developing services operating at team-level (where individual teams are responsible for debugging incidents). Doing so results in time-consuming manual, inaccurate and inefficient debugging of production incidents in clouds due to the complexity of interactions across teams and limits reliability improvements due to learning across teams that are inherently diverse.

My central thesis is that these problems will not disappear until we transition from team level debugging to cloud scale debugging. Further, this can be done in scalable fashion even for very large clouds without dismantling the successful modular structure used for team development by abstraction, changing perspective in terms of problems to solve, and defining appropriate interfaces and structures to a centralized debugging plane (CDP in what follows).

I propose the following desirable properties for solutions that contribute to cloud-scale debugging, which I call CASH:

Centralized: Operate across the system to avoid redundant efforts and store the results in a Centralized Data Plane

Abstract: Pass abstract team level information across team level APIs to the CDP to help scaling and to allow better generalization and uniform learning

Scalable: Operate across different services, incidents and data types at cloud scale

Hierarchical: Interface at varying granularity for ease of use from cloud level to team level and possibly at finer granularity.

I propose *Coarsening* as a theoretical framework to achieve CASH properties in solutions. Inspired by the theory of Abstract Interpretation, I define *Coarsening* as an abstraction of concrete poset of raw logs that capture various information related to debugging tasks (root

causes, repairs, etc.) to an element in an abstract poset of data structures. The resulting assistance to the engineer towards the debugging task serves as the concretization function forming a galois connection measured by empirical and user studies. This is to accommodate for the lack of well-defined semantics of logs and to improve learnability of the abstraction function to develop tools.

The rest of this chapter is organized as follows. In § 1.2, I provide more context by describing the debugging life cycle in clouds today. Next, in § 1.3, I present the limitations of team-level distributed approaches to different stages in the debugging lifecycle. This sets the stage to summarize my thesis contributions in § 1.4.

1.2 Debugging Lifecycle

Delivering high availability has become a predominant concern of nearly all global cloud services. As a result, global cloud services today routinely target upwards of “five nines” (or 99.999%) availability. For context, this translates to tolerable downtimes of only 6 minutes per year. Figure 1.1 presents the different tasks across which debugging an incident is distributed to meet these targets.

Incident Detection: Meeting these stringent availability targets requires accurately and quickly *detecting* incidents. To do this, engineers today deploy a slew of commercially available monitoring tools – e.g., Cadvisor [Goo21], DataDog [Dat22b] – to log various metrics that characterize system operation such as resource utilization (e.g., CPU, network, memory) and application-level performance (e.g., response times, number of dropped requests). These logs are then continuously analyzed in different ways to flag potential anomalies during system operation [Twi15, TL18, New22, AWS23]. The corresponding *alerts* trigger a human operator to further investigate the raised alert and captured logs to determine if it pertains to an actual incident. These alerts are then *triaged* to compose an incident report with observed symptoms, impact severity, etc. In the worst case scenario, an incident can go undetected by

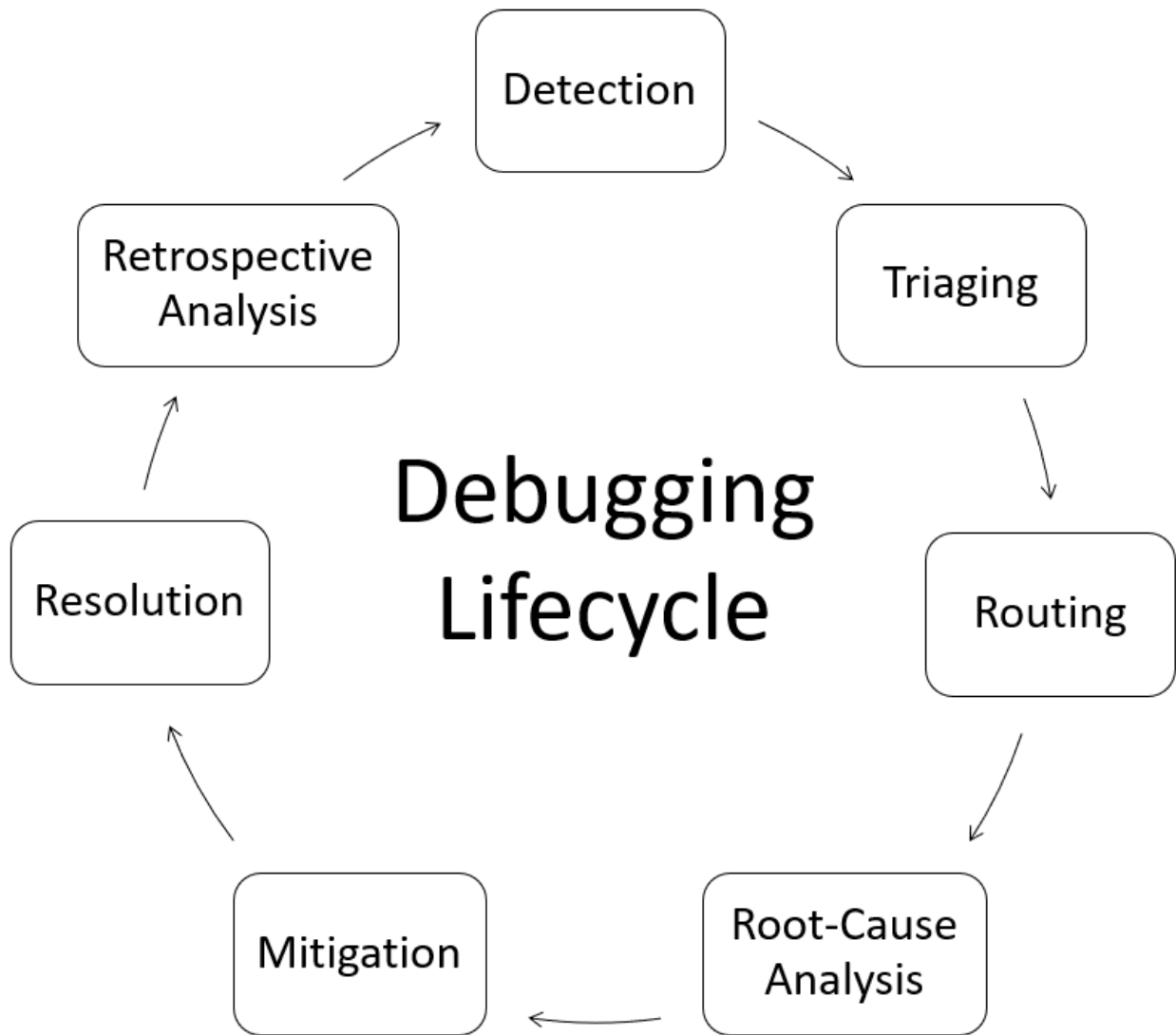


Figure 1.1: The debugging lifecycle of incidents in production distributed systems.

alerts and will be reported by end-users (customers) to trigger a human operator to initiate the debugging process.

Incident Routing: After an incident is detected, due to the complex interactions between service components, the underlying root-cause of the incident might not lie in the component managed by an operator. This requires them to manually analyze the captured

logs, determine that their component is behaving as expected, guess the plausible root-cause, possibly document the evidence in the incident report (to avoid finger-pointing) and *route* the incident to an appropriate operator for further investigation. In some instances, representatives from several components investigate logs simultaneously on a conference call, effectively rendering the incident routed to all of them.

Root-Cause Analysis: When an incident report reaches an operator, they investigate the captured logs to diagnose the *root-cause* of the incident. They often rely on execution traces to reason event triggers across different components and the corresponding outcomes. Because of the complexity of interactions across components within a service, they first hypothesize a root-cause from the observed symptoms, as there can be many such hypotheses. To evaluate their hypothesis, they execute queries (which I call debugging queries) over the datastores(s) collecting these logs. When their observation aligns this hypothesis, root-cause analysis is concluded and if their observation does not align with the hypothesis, they repeat this process with additional information they gathered from the debugging query and repeat the process until a root-cause hypothesis is established with evidence. The incident report is then tagged with this root-cause.

Mitigation: Engineers often attempt quick fixes to *mitigate* the incident symptoms while root-cause analysis can still be in progress. For example, deleting temporary files in a machine to free up disk space and restarting a crashed serviced component. While the root-cause of the incident could be one of many reasons, this action lets the service function normally while engineers continue to investigate logs to diagnose the root-cause. Engineers also devise scripts that can check for certain symptoms and automatically deploy a mitigative quick fix using their domain-expertise.

Resolution: A permanent fix that addresses the underlying root-cause of an incident (after conclusion of root-cause analysis) is then deployed to prevent recurring incidents with the same root-cause. This can take anywhere from days to weeks depending on the complexity

of the fix. For example, code can be changed to prevent a race condition and tested against to deploy but a flaw in the design of components will require major changes.

Retrospective Analysis: After an incident is resolved, engineers document the debugging process in a detailed postmortem report for management to review debugging practices and for peers to learn from their debugging experience. These reports are treasure troves of rich debugging insights that can be leveraged to inform resilient system design [GMK16] and impact all the other stages of the debugging lifecycle.

1.3 Limitations of Team Level Debugging

We have witnessed a flurry of recent advances (both tooling and methodology) for each of these tasks. Several core challenges still persist with the existing practice of *team-level debugging* despite a flurry of recent advances (both tooling and methodology) [GYM20, BRA20, LCL21, SBB10, MRF15, MBK20, DNS22, LLM19a, AWS23, GMK16, JLC20, SH22, DNS19a, ZYJ21a, Tor22, CKL20].

First, a team-level view can hurt *accuracy* of incident detection, routing, and root-cause analysis. It is common for an incident to be caused by multiple factors spread across (and its symptoms visible to) multiple services/teams. An individual team may not identify the incident (e.g., when the symptoms are insignificant within a team but significant globally), may assign it to a wrong person who spends significant resources before passing it to another team, and may fail to identify all factors across teams that contribute to the incident.

Team-level debugging can also be *inefficient*: for example, in the absence of distributed tracing across teams or automated correlation of cross-team telemetry, teams often communicate in ad hoc and *time consuming* fashion. Engineers often use diverse data sources (e.g., bug reports, source code, comments, documentation, and execution traces) to make sense of system-wide semantics, bridging together outputs and features from existing debugging tools for root-cause analysis, which is not addressed by existing research.

Finally, team-level debugging can *limit reliability improvements* because learning from retrospective analysis cannot be shared across teams. Postmortems are labelled today with a single root-cause based on an ad hoc taxonomy of root-cause tags. However, this manual process is error-prone and a single root-cause is inadequate to capture all contributing factors behind an incident, and ad hoc taxonomies fail to capture the diverse categories of root-causes, wasting rich treasure troves of debugging insights.

1.4 Thesis Contributions

Figure 1.1 highlights my completed studies next to different stages in debugging lifecycle and I present an overview of them below.

Chapter 2} *Towards Cloud-Scale Debugging via Coarsening*

In this joint work with Suman Nath from Microsoft Research and professors Ravi Netravali and George Varghese, we present that most debugging research and practice focuses on specific hard problems, shifting focus to debugging systematically at cloud-scale and across the entire life cycle suggests new research problems and a potential architecture. We structure our discussion around three axes: observability, diagnosability, and retrospective analysis. These axes suggest new problems which we explore: a) Automatically generating quick fixes (e.g., mitigation) based on failure cues; b) Consistently labelling failures across teams for retrospective analysis, c) Scalably maintaining coarse dependency graphs among components; d) Scalably maintaining telemetry and failure data for long periods. We suggest initial approaches to these questions, provide early results highlighting the potential benefits of this vision, and outline a future agenda to realize it.

Chapter 3} *A System-Wide Debugging Assistant Powered by Natural Language Processing* [DNS19b]

In this joint work with professors Karthik Narasimhan, Anirudh Sivaraman and Ravi

Netravali, we present using statistical natural language processing (NLP) techniques to help automatically analyze often ignored diverse data sources (e.g., bug reports, source code, comments, documentation, and execution traces) that engineers *collectively* use to make sense of system-wide semantics, bridging together outputs and features from existing debugging tools to improve their debugging experience. We show this by automatically identifying the location of root-cause in a code repository using issue reports from popular large open-source code repositories on Github. We further identify the many systems and learning challenges that must be overcome to realize this vision.

Chapter 4 } *REVELIO: ML-Generated Debugging Queries for Finding Root Causes in Distributed Systems* [DNS22]

In this joint work with Shiv Saini from Adobe Research and professors Karthik Narasimhan, Anirudh Sivaraman, George Varghese and Ravi Netravali, we present REVELIO, a debugging assistant which takes user reports and system logs together as input, and outputs debugging queries that developers can use to find a bug’s root cause. It exploits observations from production systems to factorize query generation into two computationally and statistically simpler learning tasks and exploits NLP and ML models to uniformly represent diverse types of data. We built a testbed with multiple distributed applications and debugging tools. By injecting faults and training on logs and reports from 800 Mechanical Turkers, we show that REVELIO generates the most helpful query in its top-3 outputs 96% of the time and we conducted a developer study which confirmed the utility of REVELIO.

Chapter 5 } *AUTOARTS: Insights and Tools for Rootcausing Incidents in Microsoft Azure* [DBJ23]

In this joint work with Chetan Bansal, Gopinath Jayagopal, Richie Costleigh, Suman Nath and Xuchao Zhang, we present that the existing manual root-cause labelling process in clouds is error-prone and a single root-cause is inadequate to capture all contributing factors behind an incident, and ad-hoc taxonomies fail to capture the diverse categories

of rootcauses. To address this, we present a three-pronged approach. First, we conduct the largest and most comprehensive study of production incident postmortem reports to understand *all* contributing factors behind incidents. Second, based on the empirical study, we propose a novel hierarchical and comprehensive taxonomy of contributing factors. Lastly, we develop an automated tool that can assist humans in the labelling process. To evaluate the tool, we present empirical evaluation as well as a user study.

Chapter 6 Scalable Causal Discovery for Root Cause Analysis of Performance Degradation in Distributed Systems

In this joint work with Vikramank Singh, Zhengchun Liu, Murali Narayanaswamy and Tim Kraska, we present that performance degradation over request response times are challenging and time-consuming for engineers to reason. Engineers' domain expertise over large-scale services is narrow due to modularity and existing causal discovery techniques cannot scale to production-scale telemetry due to computational complexity and to the large and dynamic execution state of distributed systems due to missing confounders. To address this, we present domain-agnostic featurization technique that enables leveraging wide range of telemetry. We then use scalable anomaly detection over the telemetry as a proxy to the execution states of the distributed system to enable causal discovery across the telemetry to assist engineers in root cause analysis. To evaluate the tool, we present a user study and an empirical evaluation.

CHAPTER 2

Towards Cloud-Scale Debugging via Coarsening

This chapter provides an overview of a joint work with Suman Nath from Microsoft Research and professors Ravi Netravali and George Varghese. While most debugging research focuses on specific hard problems, shifting focus to debugging systematically at cloud-scale and across the entire life cycle suggests new research problems and a potential architecture.

For scalability, debugging effort is usually distributed among loosely-coupled service teams [CC20]: an outage in a service is debugged by the corresponding team primarily with the help of its own domain knowledge, telemetry, and logs. Only if this is found inadequate, the team may seek help from other teams, often in ad hoc ways [GYM20]. Based on our experience at several large cloud platforms, we find this *team-level* debugging sub-optimal, often resulting in delayed, insufficient, or even incorrect mitigation despite a flurry of recent advances for different tasks in the debugging lifecycle [GYM20, BRA20, LCL21, SBB10, MRF15, MBK20, DNS22, LLM19a, AWS23, GMK16, JLC20, SH22, DNS19a, ZYJ21a, Tor22, CKL20].

We argue that *cloud-scale* debugging — where debugging is done using centralized domain knowledge and selected logs/telemetry from all service teams — can address many of these limitations. Cloud-scale optimal debugging decisions are also team-level optimal. For example, accurate cloud-scale incident routing means minimal involvement in other team incidents, and shared learning allows teams to prioritize solutions to quickly mitigate incidents. While the scale of clouds prompted team-level debugging, we argue that our vision for cloud-scale debugging is indeed enabled by it.

We envision a centralized debugging plane (CDP) (details in § 2.5 with a *large* central store of catalogued key information that allows teams to learn where, when and how to instrument and monitor information to reduce alert false positives, coarse dependency graphs that capture *global* dependencies to automate rapid incident routing to the right team, automated learning of quick mitigation suggestions and key failure patterns from the *many* incidents across teams to make process and investment changes that increase overall cloud reliability. To provide a theoretical framework towards building components of CDP, we propose *Coarsening*, explained in detail in § 2.1.

We present the unique advantages and challenges (beyond scale) that cloud-scale debugging introduces. In particular, we focus on three related debugging tasks: *retrospective analysis* of past incidents to identify and prevent common failure patterns to improve long term reliability, *diagnosability* of an incident to find its root causes, and *observability* of deployed systems to help diagnosis and retrospective analysis. While the tasks are not exhaustive, we hope that they motivate our key thesis and inspire future work on effectively debugging large cloud systems.

2.1 Coarsening

The theory of Abstract Interpretation [Cou05] is used to make a sound approximation of the semantics of a computer program. The elements in a concrete poset (e.g., inputs to a program) are mapped to elements in an abstract poset (e.g., sign of an arithmetic expression) such that computations over the abstract poset are computationally simpler. The resulting abstract computation can then be concretized forming a galois connection, which loses some information (e.g., output of arithmetic expression) but preserves the property (e.g., potential sign of the output of arithmetic expression).

Abstraction function and concretization function should be well-defined to form a galois connection. This is achieved by exhaustive enumeration of semantics (e.g., addition of

two positive values results in a positive value). Unfortunately, data logged from service components is largely heterogeneous and has little to no semantics due to the complexity of services. Inspired by Abstraction Interpretation and humbled by the nuances of incidents in production systems, I define *Coarsening* to be composed of:

- Two partially-ordered sets:
 - C (concrete): A poset of real incidents from across the clouds stored across various raw logs. Note that depending on the use case an incident can be a postmortem report, it's logs, combination of different subsets of data, etc.
 - A (abstract): A poset of data structures that are relevant for a debugging task and have well-defined syntax and semantics. For example, debugging queries are relevant for root cause analysis and have well-defined syntax and semantics.
- A learned (ML-based) coarsening function α that maps $C \rightarrow A$ as shown in Figure 2.1. The ground-truth for this mapping is ideally recorded by experts in historical debugging data (e.g., the severity level of an incident). The goodness of α is evaluated through empirical studies using metrics such as precision, recall, F1-scores, etc., over different cuts of available incident data.
- The debugging assistance γ provided by the abstract data structure to an engineer to perform the debugging task, as the implicit concretization function γ that maps $A \rightarrow C$. The goodness of γ is estimated through user studies.
- A galois connection between the learned Coarsening function α and the debugging assistance offered γ :
 - $\forall a \in A, c \in C : \alpha(c) \leq a \iff c \subseteq \gamma(a)$
 - $\forall a \in A : \alpha(\gamma(a)) \leq a$
 - where \subseteq, \leq are ordering of posets C, A respectively.

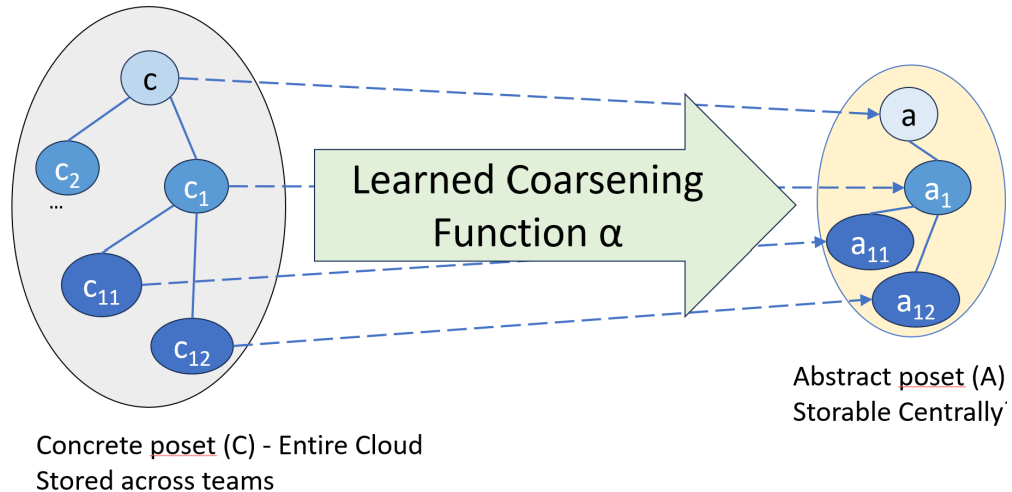


Figure 2.1: Learned coarsening function that maps a concrete element (root cause of an incident) to an abstract element (e.g., root cause label)

- This galois connection indicates that an engineer can arrive at a ‘coarser’ completion of the debugging task, ideally completing the task with little to no extra work.

The following characteristics of *Coarsening*, make it amenable for transitioning towards cloud-scale centralized debugging:

- Enables centralized storage of incident debugging data through compact abstractions and sequentially scaling down on relevant logs over space/time at each stage of debugging. (e.g., a debugging query localizes spatially over a subset of log, a repair set localizes on a component of service/infrastructure).
- Accommodates for the ill-defined semantics of root causes of incidents with task-specific abstract representations (e.g., debugging queries, labels, repair sets, causal graphs) that have well-defined semantics that can be consumed through APIs.

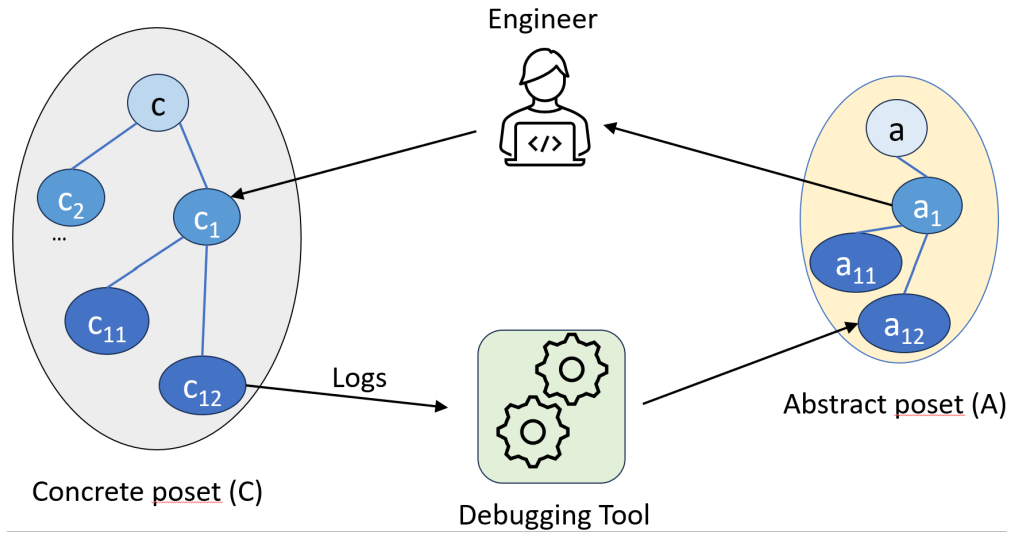


Figure 2.2: Implicit concretization of concrete element through assistance from abstract element (debugging tool’s output) forming a galois connection

- Improves learnability of ML-based tools by increasing the density of abstract elements, i.e., ML tools can better learn the semantics of a coarser abstract element through increased number of training samples from concrete elements.

Figure 2.3 shows *Coarsening* in iterations across different stages of the debugging lifecycle to assist engineers. The size of the oval blobs indicate the amount of work involved to resolve an incident and extract insights from it. The dashed arrows indicate manual labor engineers have to put in from the starting point (the task at the bottom of an oval blob) to resolution. The solid arrows indicate automated completion of (or) assistance for one debugging task, enabling engineers to progress to the next task. Each oval blob also is annotated by its operational scale, for example, the left-most blob (the biggest) indicates that the operational scale is the entire cloud, as there is no available information about whether or not there is an incident and its symptoms. At cloud-scale, the desired end outcome of an incident are insights to improve the reliability of cloud services against future incidents. Unfortunately,

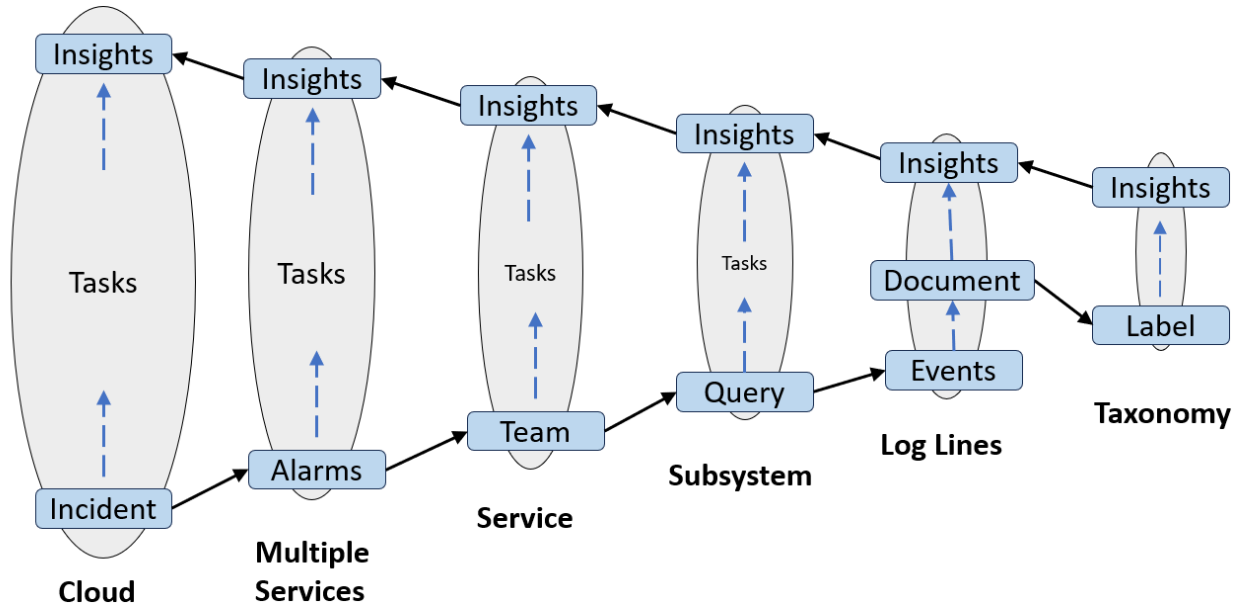


Figure 2.3: Coarsening in stages to iteratively scale down manual work and assist engineers in debugging tasks. Dashed arrows indicate hard manual labor that engineers would need to do without any assistance and solid arrows indicate automated completion of or assistance for one debugging task, progressing engineers to the next task. The size of each oval blob indicates the amount of work to be done and its operational scale is indicated below.

today engineers have to manually learn insights from incidents by painstakingly working through all the stages of the debugging lifecycle.

Our proposal is that this enables a central cloud-scale authority to conduct debugging through a relatively much smaller storage overhead (for each incident, the owning service name, the debugging query, the root cause labels, the repair set) and computational overheads in building ML tools at each stage by scaling down necessary logs. From left to right in Figure 2.3, ML-based debugging tools can be built that can automatically assist engineers in achieving a ‘coarse’ completion of the overall debugging tasks *and* as a result narrow-down the scope of available data:

- Coarsening begins with analyzing all the available logs and user reports cloud-wide to evaluate whether an incident is present/absent, if present, the impacted regions and duration based on SLAs. As a result, the operating set is now scaled down to a subset of ‘alarms’ possibly across multiple services and regions.
- In the next stage, by leveraging probes, coarse dependencies and other KPIs (Key Performance Indicators), an ‘owning’ team is identified to whom the incident is routed to for further debugging. As a result, the operating set is now scaled down to one service at a region.
- In the next stage, by leveraging the logs in the ‘owning’ team’s service, a debugging query is generated which identifies the key culprit metrics within the service log and the infrastructure components that validates a root cause hypothesis. As a result, the operating set is now scale down to a handful of metrics and infrastructure.
- In the next stage, by leveraging the log lines captured in that service, a sequence of log lines with an unexpected pattern is identified for more granular root cause diagnosis and mitigation. As a result, the operating set is now a few log lines at a small number of infrastructure components and service code.
- In the next stage, by leveraging the natural language postmortem report, root cause labels are identified, which can ultimately be used to extract insights to improve the overall reliability of the cloud.

In what follows, we explain the ‘Coarsening’ formulations that achieve iterative scaling down of scope and assistance to debugging tasks in Figure 2.3 one stage at a time. Note that, detailed explanations of the techniques used in building the debugging tools are available in their respective chapters.

2.1.1 Coarsening Formulation - Incident \rightarrow Alarms

Due to the lack of *accurate* and *interpretable* alerting tools, engineers today use their domain knowledge to define rules on system metrics to raise alerts. This approach gives them the benefit of defining post alert rulebooks to investigate the incident, but unfortunately the false positive rate is still so high to cause "alert-fatigue" and obsolescence of rules added in the past. Drawing from the importance of interpretable alert rules in detecting and handling incidents, we present a coarsening formulation as follows:

- $Incidents = \{i_1, i_2, \dots, i_m | i_x = \text{Unique Incident IDs}\}$
- $Alarms = \{a_1, a_2, \dots, a_n | a_i = \text{unique alarm ID cloud - level}\}$
- $Fired(alarm) = 1 \iff \text{alarm is triggered \& alarm} \in Alarms$
- $Ground_Truth(incident) = alarms \subseteq Alarms$
- $Concrete\ Poset\ (C, \subseteq) : c_x \subseteq c_y \iff g_y \subseteq g_x$
 $Fired(alarm) = 1 \forall alarm \in alarms_x \cup alarms_y$
 $g_i = Ground_Truth(c_i)$
 $c_i = incidents \subseteq Incidents | Ground_Truth(incident \in incidents) = alarms_i$
 $\{c_i, c_j\} \in C$
- $Abstract\ Poset\ (A, \leq) : a_x \leq a_y \iff alarms_y \subseteq alarms_x$
 $Fired(alarm) = 1 \forall alarm \in alarms_x \cup alarms_y$
 $a_i = alarms_i \subseteq Alarms$
 $\{a_i, a_j\} \in A$
- Input for the debugging tool: $Alarms, Fired$
- Output of the debugging tool: $alarms \subseteq Alarms, Fired(alarm) = 1 \forall alarm \in alarms$

2.1.2 Coarsening Formulation - Alarms \rightarrow Team

In this stage of debugging, we already have a list of alarms that are relevant for an incident. The debugging task is to identify the culprit team from these alarms, by using dependency graphs, probes, etc. The Coarsening formulation is as follows that ideally enables identifying the right team and if not, a ‘coarser’(parent) team that can route the incident to the right engineers within their team:

- $Incidents = \{i_1, i_2, \dots, i_m | i_x = \text{Unique Incident IDs}\}$
- $Dependency\ Graph, DG = (T, E)$
 $T = \{Developer\ Teams\}, E = \{(T_i, T_j) | Code(T_i) \text{ calls } Code(T_j) \forall T_i, T_j \in T\}$
- $Code(team) = \text{Code developed by team}$
- $Ground_Truth(incident) = team \in T$
- $Concrete\ Poset (C, \subseteq) : c_x \subseteq c_y \iff Code(g_x) \subseteq Code(g_y)$
 $g_i = Ground_Truth(c_i)$
 $c_i = incidents \subseteq Incidents | Ground_Truth(incident) = team_i$
 $\forall incident \in incidents$
 $\{c_i, c_j\} \in C$
- $Abstract\ Poset (A, \leq) : a_x \leq a_y \iff Code(a_x) \in Code(a_y)$
 $a_i = team_i \in T$
 $\{a_i, a_j\} \in A$
- Input for the debugging tool: Logs, Probes collected from services with the relevant alarms from previous stage.
- Output of the debugging tool: $team \in T, Fired(alarm) = 1 \exists alarm \in Code(team)$

2.1.3 Coarsening Formulation - Team \rightarrow Query

In this stage of debugging, we already have root cause analysis routed to the right team. The debugging task is to identify the culprit component within the components managed by this team, by using debugging queries, system logs, etc. The Coarsening formulation is as follows that ideally enables identifying the right component and if not, a ‘coarser’ template that can help engineers diagnose the root cause of the incident:

- $Incidents = \{i_1, i_2, \dots, i_m | i_x = \text{Unique Incident IDs}\}$
- $Debugging\ Queries : Q = \{Q_1, Q_2, \dots, Q_n | Q_i = \text{valid query over logs}\}$
- $Result_Set(query) = \text{Result - set from issuing query}$
- $Ground_Truth(incident) = query \in Q$
- $Concrete\ Poset (C, \subseteq) : c_x \subseteq c_y \iff Result_Set(g_x) \subseteq Result_Set(g_y)$
 $g_i = Ground_Truth(c_i)$
 $c_i = incidents \subseteq Incidents | Ground_Truth(incident) = Q_i \forall incident \in incidents$
 $\{c_i, c_j\} \in C$
- $Abstract\ Poset (A, \leq) : a_x \leq a_y \iff Result_Set(a_x) \in Result_Set(a_y)$
 $a_i = Q_i \in Q$
 $\{a_i, a_j\} \in A$
- Input for the debugging tool: Logs, User Reports collected from components within the service owned by the team identified from previous stage.
- Output of the debugging tool: $query \in Q$

2.1.4 Coarsening Formulation - Query \rightarrow Events

In this stage of debugging, we already have identified a culprit sub-component/sub-system within a service. The debugging task is to identify a summary of log that captures the sequence of events that led to the incident by using system logs, events, etc. The Coarsening formulation is as follows that ideally enables identifying the right sequence of events to help engineers diagnose the root cause of the incident:

- $Incidents = \{i_1, i_2, \dots, i_m | i_x = \text{Unique Incident IDs}\}$
- $Events : E = \{E_1, E_2, \dots, E_n | E_i = \text{events captured in logs}\}$
- $Follows(E_i, E_j) = True \iff E_j \text{ should follow } E_i \text{ per design}$
- $Ground_Truth(incident) = [E_a, E_b | Follows(E_a, E_b) = 0] \subseteq E$
- $Concrete\ Poset (C, \subseteq) : c_x \subseteq c_y \iff Follows(g_x[1], g_y[1]) = 1$
 $g_i = Ground_Truth(c_i)$
 $c_i = incidents \subseteq Incidents | Ground_Truth(incident) = [E_a, E_b]$
 $\forall incident \in incidents$
 $\{c_i, c_j\} \in C$
- $Abstract\ Poset (A, \leq) : a_x \leq a_y \iff Follows(a_y, a_x) = 1$
 $a_i = E_i \in E$
 $\{a_i, a_j\} \in A$
- Input for the debugging tool: Logs collected from the culprit component identified from previous stage.
- Output of the debugging tool: $[E_x, E_y | Follows(E_x, E_y) = 0]$

2.1.5 Coarsening Formulation - Document \rightarrow Label(s)

In this stage of debugging, the incident has been resolved and a postmortem report has been document. The debugging task is to label the postmortem report with all the contributing factors to enable extracting meaningful insights to improve the overall reliability of the cloud using a well-defined taxonomy and the content of the postmortem report. The Coarsening formulation is as follows:

- $Incidents = \{i_1, i_2, \dots, i_m | i_x = \text{Unique Incident IDs}\}$
- $Taxonomy : (L) = \{L_1, L_2, \dots, L_n | L_i = \text{Root Cause Label}\}$
- $Parent_Of(L_1, L_2) = 1 \iff L_2 \text{ is a sub - category of } L_1$
- $Ground_Truth(incident) = [L_x, L_y, \dots, L_z] \subseteq L$
- $Concrete\ Poset\ (C, \subseteq) :$
 - $c_x \subseteq c_y \iff Parent_Of(g_y, g_x) = 1 \ \& \ Parent_Of(g_x, g_y) = 0$
 - $g_i = Ground_Truth(c_i)$
 - $c_i = incidents \subseteq Incidents \mid Ground_Truth(incident) = [L_i, L_j, \dots]$
 - $\forall incident \in incidents$
 - $\{c_i, c_j\} \in C$
- $Abstract\ Poset\ (A, \leq) : a_x \leq a_y \iff Parent_Of(a_y, a_x) = 1$
 - $a_i = L_i \in L$
 - $\{a_i, a_j\} \in A$
- Input for the debugging tool: Postmortem Report of incident in natural language.
- Output of the debugging tool: $[L_x, L_y, \dots, L_z] \subseteq L$

2.2 Retrospective Analysis

A cloud-scale retrospective analysis of incidents and their fixes uncovers common problem areas and patterns to mitigate future failures [GMK16]. These insights guide cloud providers to prioritize their reliability investments. We demonstrate this with a multiple person-year effort that analyzed 2000+ incidents from 450+ teams within Microsoft Azure. Table 2.1 shows the top 5 contributing factors of severe incidents and Figure 2.4 shows the distribution of time taken to mitigate (TTM) these incidents with a median TTM of 4 hours. We also list the steps taken to mitigate these incidents (besides root causing), that we refer to from now on as *Quick Fixes*. Observe that quick fixes come in two categories. The first is immediate actions like rolling back recently deployed changes (Row 2), and procedural changes (Rows 1, 3, 4, and 5) such as requiring better tracing. Procedural changes will not fix an ongoing incident quickly but will likely prevent future problems or help diagnose them faster.

Contributing factor	Corrective Steps
Missing alerts	Monitor failures, exceptions, QoS, etc.
Buggy code change	Roll-back recently deployed changes.
Insufficient telemetry	Trace QoS related metrics, resources.
Latent bugs	Improve test coverage & define alerts.
Poor test coverage	Multi-dimensional test-cases (hardware, dependency, stress.)

Table 2.1: Top-5 contributing factors across 450+ cloud services at Microsoft Azure and the corresponding quick fixes.

This study shows that identifying and addressing common flaws can significantly decrease incidents; simple logistical wins can be had by leveraging past debugging experience. These insights were generated by *manual* cloud-scale analysis of a *subset* of incidents. Though quick fixes are easily actionable (e.g., rebooting a server, scaling-up resources) once found, identifying which of the many ad-hoc quick fixes are worth a team’s time to deploy is non-trivial and is sub-optimal (e.g., a quick fix already identified by one of their many

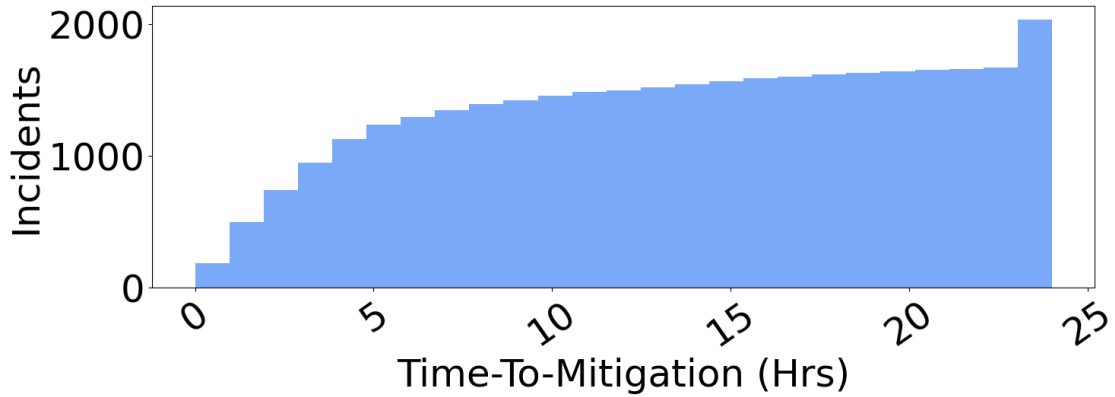


Figure 2.4: CDF of mitigation times of over 2000 incidents from our manual study at *Anon* (TTM over 24 hrs depicted as 24 hrs).

counterparts) at team-level. The rat-race between different cloud competitors and the scale of cloud services prevents engineers from identifying and deploying all useful quick fixes team-locally. This motivates cloud-scale retrospective analysis wherein the increased scale of information can generate global insights which can then be deployed team-locally.

2.2.1 Quick Fixes

Much existing research focuses on building tools to identify and fix specific failures (e.g., performance [GLD21], concurrency bugs [LLM19a], configuration changes [MBK20]). At *Anon*, these failures only constituted a very small fraction of incidents. For example in our study, configuration changes (broadly) contributed to only 4.51% and capacity/load issues contributed to only 5.1% incidents. It is impractical to run *each* specific tool *continuously* in *each* production service to achieve a *rare* win in diagnosing the root cause of an ongoing incident.

A global, retrospective analysis can identify relatively simpler and more practical logistical wins to quickly mitigate production incidents. Examples of simple logistical wins based on incidents at *Anon* are:

(Q1) After finding that latent bugs frequently cause incidents, a policy can be enforced to deploy code only if its test coverage exceeds say 80%, instead of zeroing-in on the buggy code, which is time-consuming and challenging [LLM19b, DNS22].

(Q2) After seeing that authentication related failures cause many production incidents, one can invest in building (fault injection) tools that can uncover authentication issues during testing, automatically rotate certificates, and employ monitors to detect soon-to-be expired certificates in production.

These efforts at *Anon* are proving useful by reducing incidents. These examples suggest that cloud operators should shift the arc of debugging research from automation for finding deep causes (whose coverage is necessarily small) to automation for finding quick fixes with broader coverage.

The main challenge in identifying quick fixes is learning how to associate a quick fix with features of a failure. While this could be done within teams, the same patterns can recur across teams though the specific implementations differ. Aggregating across teams in cloud-scale debugging allows larger datasets to better train ML models on unstructured failure reports. Other research questions include: extracting meaningful information from lengthy natural language documents to aid policy-makers in simple fix generation; identifying metrics to monitor, determining which test-cases are needed to increase test coverage; estimating the risks of new deployments based on past incidents; and automating insight generation from postmortem reports.

2.2.2 Consistent Root-Cause Labelling

Table 2.1 hides hundreds more such rows in the real table due to space constraints. The number of rows (and fixes) can significantly increase when analysis scales to hundreds of thousands of past incidents. These past fixes are valuable for engineers to fix ongoing incidents. For example, while fighting an ongoing incident caused by a specific root cause (e.g., a faulty

network driver), an engineer may find it useful to consult past fixes for incidents caused by the same root cause.

At cloud-scale, large amounts of training data exist, but off-the-shelf techniques can be inaccurate due to heterogeneity in domain-specific language use. A more practical solution is to label each incident and postmortem report with tags describing their root causes and fixes; then one can easily search over these tags without expensive NLP. A key requirement, however, is that the labelling must be consistent— e.g., all network driver related incidents/postmortems need to be labelled with the same tag. At Microsoft Azure, we found that team-level labelling is unworkable —different teams label the same root cause with different labels due to differing expertise and interpretation. We describe our attempt at solving this in [Chapter 5](#)

While we made significant progress towards bootstrapping consistent root-cause labelling, there are several problems that need addressing. Examples include (but are not limited to): Unsupervised clustering of incidents where each cluster represents semantically singular root-cause (Our tool was trained on small subset of *labelled* incidents); Highlighting key pieces of information that can enable engineers in appending it to the taxonomy (Our taxonomy was *manually* grown); Extracting patterns in logs/telemetry to label root-causes for incidents that do not have or have partial postmortem analysis (Our tool *only* uses postmortem reports).

2.3 Diagnosability

We focus on incident routing to show the advantages of cloud-scale debugging. We have seen at least three common practices today, all of which work at team-scale. First, all affected teams meet and coordinate, which doesn't scale. Second, manual rule-based routing is used, which causes incidents to be routed back and forth across teams. Finally, automated tools like Scouts [[GYM20](#)] are trained by individual teams, where each team can only debug based

on internal metrics. Once assigned, engineers bear the burden of gathering enough evidence to route to another team.

Globally orchestrating incident routing by leveraging and correlating telemetry from all teams can greatly increase its accuracy. For example, at *Anon*, an outage at service X was caused because of a failure at another service Z that X depends on via service Y . By knowing that X experienced the incident at the same time Z 's telemetry indicated failure, a global incident routing algorithm can route the incident to Z who can investigate the root cause behind Z 's failure. X 's team-level approach does not consider Z 's telemetry and will fail to do so.

However, at cloud-scale, it is non-trivial to distinguish concurrent failures from common failure symptoms across services. Earlier work has already shown methods to construct and use *dependency graphs*. However, there are two key challenges at cloud-scale. First, instrumentation-based techniques [SBB10] are hard to consistently deploy across teams for logistical reasons (e.g., legacy code). At *Anon*, many teams do not implement distributed tracing or ignore crucial correlation IDs, making stitching together traces across services infeasible. Second, fine-grained dependency graphs are noisy and vary over request types and over time as service implementations evolve.

We present our attempt with Coarse Dependency Graphs for centralized incident routing. Key research questions include: (1) how to construct CDGs despite the absence of distributed tracing in some teams, (2) how to efficiently maintain CDGs as services evolve, (3) how to leverage CDGs to scale automated correlation of telemetry for diagnosing cross-team incidents and distinguishing concurrent failures from symptoms of a single failure. We are investigating (1) and (2) with automated instrumentation and offline analysis of auxiliary information e.g., configuration files and DNS requests for dependencies.

2.3.1 Coarse Dependency Graphs

A dependency graph contains edges $x \rightarrow y$ if component x depends on component y at runtime. At the very least, a coarse-grained dependency graph shows dependencies of various services and teams (Figure 2.5 and Figure 2.6). By contrast, a fine-grained dependency graph shows dependencies between components within a service and can be useful for root causing. To allow team sovereignty, the CDP may let individual teams maintain their own fine-grained dependency graphs. Dependency graphs can be constructed manually with domain knowledge, automatically by data-driven techniques [CMF14], or via automatic instrumentation.

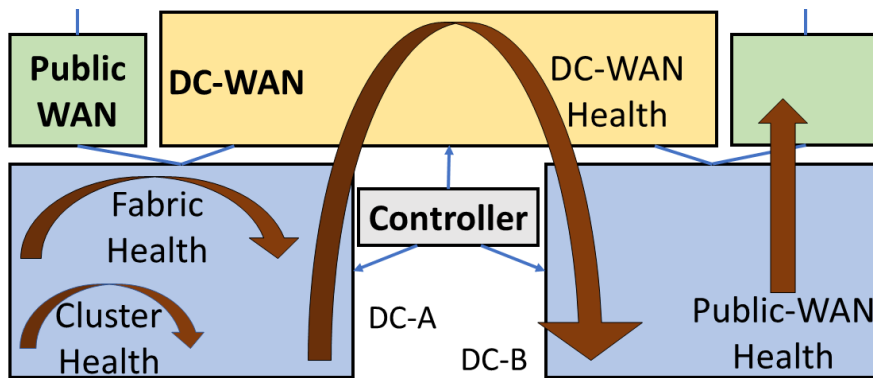


Figure 2.5: Health checks for reachability in a cloud network. Arrows indicate reachability probes within clusters, fabrics, and across WANs.

2.3.2 Incident Routing

Symptom Explainability: We define symptom explainability of team T as the fraction of symptoms explained assuming team T is the only one to fail. Unfortunately, this does not generalize when there is a vector of symptoms of different types (e.g., infrastructure health metrics exceeding thresholds, network probes failing). Instead, we define the vector of symptoms as an *incident syndrome*. We then define *symptom explainability* for team T as the cosine similarity of the incident syndrome to the syndrome if *only* team T failed. This allows

for noise and normalizes each team’s explainability feature to lie between 0 and 1. Symptom explainability assumes only a single failure. However in systems with many microservices, multiple concurrent failures are common and we present a technique for root-cause analysis.

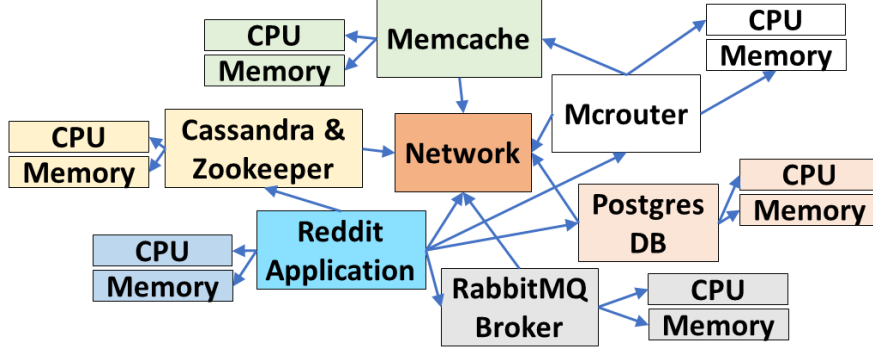


Figure 2.6: Coarse dependency graph for Reddit in our distributed systems testbed.

2.3.3 Root-Cause Analysis

Dependency Graph Traversal: Given an incident at a component S and a dependency graph D for S , we aim to identify a small set X of nodes in D whose failures have likely propagated to S to cause the incident. We initialize $X = \{S\}$, and traverse D breadth-first starting from S . At each step, we select a node $x \in X$ and find all its child nodes $y \notin S$ such that (1) there is an edge $x \rightarrow y$ in D and (2) y have failed, i.e., reported bad health telemetry, around the time of the incident. If any such child node is found, it is added to X and x is removed from X . Intuitively, if x depends on y (i.e., $x \rightarrow y$) and both x and y has failed, the traversal keeps y and discards x since y is the root-cause of x ’s failure.

To reduce computational complexity, symptom explainability over the coarse dependency graph can be used to rank culprits, and graph traversal can be done on the finer-grained dependency graph for those culprits.

2.3.4 Preliminary Evaluation - Incident Routing

We construct a coarse dependency graph (shown in Figure 2.6) for our testbed (details in Chapter 4). We identify 8 “teams”: Network, Application and Infrastructure (per-microservice in the topology — 6 teams) to route incidents to. On 560 incidents collected in the Revelio incident dataset [DNS22] with the open-source Reddit [red22] application, CDP’s incident routing achieves an accuracy of 78.57% indicating the benefit of symptom explainability as compared to using only internal metrics (45.24%). We also conduct an experiment to evaluate a distributed Scouts-like approach to incident routing (using only 3 teams to allow for sufficient training data for each team’s scout) achieving an accuracy of 21.43%. Observe the reduced performance caused by lack of visibility into the global structure.

2.3.5 Preliminary Evaluation - Root Cause Analysis

In our second experiment, we use an Azure Stack[Mic22] private cloud deployment to evaluate the dependency graph traversal technique. To construct a fine-grained dependency graph, we instrument Azure Stack binaries to track interactions between various micro-services and their API calls. We then deploy the instrumented version of Azure Stack on a 4-machine cluster and run a workload including a VM creation operation. Our instrumentation produces fine-grained dependency graphs for the VM creation operation. The graph consists of 29 nodes and 38 edges.

For each of the 10 incidents, dependency traversal narrowed down the set of root-causes to fewer than 3 nodes (out of 29 nodes), which included the true root-cause identified by engineers. This shows that the traversal algorithm can significantly reduce the set of micro-services and their API calls that engineers need to investigate.

2.3.6 Coarse Log Summarization using Chains

We analyzed a private cloud system at *Anon* and found that only 0.87% of logged events from its 343 services carried a correlation ID required for distributed tracing. To enable accurate and complete distributed tracing, engineers across all services (including legacy components) need to instrument the code with a standard distributed tracing framework which is expensive and time-consuming. Further, distributed tracing alone does not help localizing faults. We solve both problems by a form of coarsening that we call chain summarization.

Each log entry (event) is timestamped, and identified by the service name that emits the entry and an opaque event ID which corresponds to the line number in the source code of the service that emits this entry. Each entry also contains a parametric description (in natural language as written by the developer) and a list of parameters that are instantiated in the description. Our experience is that such a stylized format is also common to other cloud providers and lends itself to a simple templating strategy (without sophisticated NLP) to extract parameters by aligning several occurrences of log entries for the same event ID, and extracting parameters as the variable portions.

We adopt a 3-phased approach (see Figure 2.7) to generate execution traces: (1) we extract start and end points of an execution trace by tracking the occurrence of an event B after the occurrence of an event A within a sliding window of duration $10minutes$. We then add a directed edge from event A to event B if and only if event B appears within the 10 minute window after all occurrences of event A at least 90% of the number of times. We use this heuristic (B often follows A within a short time window) to surmise that A causes B . (2) We then construct a directed graph of events whose edges are drawn from step-1 and prune the edges of this graph by only preserving edges between events which share common parameters in their respective properties and direct edges such as $A \rightarrow C$ if $A \rightarrow B$ and $B \rightarrow C$. (3) With the resulting graph, we search for all events between the occurrences of events A and B of an edge ($A \rightarrow B$) which share common parameters with A or B , which

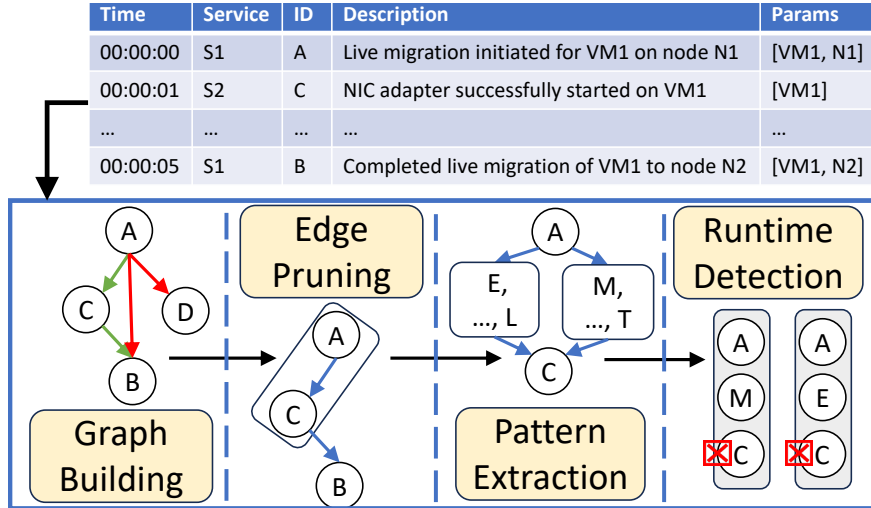


Figure 2.7: Overview of our approach to detect anomalous execution during runtime for root cause analysis.

are not captured in the graph (due to parallelism and branching). We then pick the patterns (third frame in Figure 2.7) that cover the 90th percentile of executions that start with event A and end in event B . Note that the events between A and B in a given pattern are unordered to build robust patterns and are only ordered by timestamp during runtime. We refer to such sequence of events as *chains*.

During runtime, we can efficiently find all occurrences of A that are not followed by a subsequent expected B event by initializing a queue of all chains that start with A and simply removing the expected events in the chain as they occur. If after a period, expected events remain in the queue (e.g., M & E in Frame 4 of Figure 2.7 due to missing C), we output the failed chain. and highlight which events are expected to occur after A based on step-3 to localize to specific components/services: this is helpful for engineers in root cause analysis.

Note that we choose a large enough window to not miss true $A \rightarrow B$ relationships and we choose a high threshold to prevent false positive edges. We further refine the graph based on

common shared parameters in the event properties to get rid of cycles in the graph due to the large window size (e.g., background worker events making it past threshold).

Engineers can investigate these incomplete chains and provide feedback as to which of failures are expected by design (e.g., failed live migration of a VM to a node because it doesn't have sufficient resources). This can then be incorporated to our runtime checks to avoid future false positives.

2.3.7 Preliminary Evaluation - Log Summarization

As an early experiment, we consider millions of events from live clusters in the cloud with 3089 distinct event IDs from 289 services. Steps 1 and 2 resulted in a directed acyclic chain graph with 375 nodes and 389 edges. We then focused on one edge $A \rightarrow B$ in the graph (which was validated by engineers to be correct) where A is the 'initiation of a VM {param1} live migration from source node {param2}' and B is 'successfully completed the live migration of VM {param1} to destination node {param2}'. Using this edge, we were able to identify 2500 live migrations that successfully completed and 163 live migrations which failed (resulting in a 94% reduction in events to investigate). Further, from step-3 of our approach, we identified 5 execution patterns (reduction of 99.8% data for investigation) that cover 100 failed live migrations through which engineers easily identified 2 of the patterns as expected behavior and the remaining 3 patterns to be investigated for root cause analysis.

While these results are very preliminary, they suggest that the coarsening provided by chain summarization of service logs can be succinct (500X data reduction, which enables storage at a central repository), extremely fast to process using the queue algorithm, and yet provides strong localization (94% less events to examine compared to sifting the entire log as engineers do today). Note that even in a world with perfect tracing where Step 1 can be provided by traces, Steps 2 and 3 are still useful for localization.

2.4 Observability

2.4.1 Monitoring

Developers log information needed to analyze performance, incidents, etc. Each team (responsible for a component in the cloud) exercises its sovereignty to determine what, where, and when to log leveraging their expertise and overhead measurements. Existing logging frameworks like AWS’s CloudWatch, Google’s Lighthouse and Datadog [AWS23, Lig22, Dat22b] operate at the scale of an individual service. The following challenges remain: (1) No policies exist to say where, when, what and how to log and monitor information — communication across teams can inform strategic logging and alerting, (2) It is non-trivial to catalog and analyze the data — clouds need standards and mechanisms to handle schema changes (3) There is no uniform framework to capture other data such as debugging queries issued on logs, mitigation commands, etc., issued via CLI or other interfaces in ad hoc ways. Cloud-scale observability can also enable spatio-temporal log and alert refinement (which metrics at components to log and when to increase logging granularity) by leveraging CDGs.

2.4.2 Data Retention

It is impossible to retain logs permanently due to their volume. Often, retention policies retain incoming stream of logs within a buffer duration and discard older data. ML based approaches exist to debug production incidents [GYM20, BRA20, DNS22, GLD21, SBN21]. However, they require past historical incident data. Postmortem reports and incident tickets are permanently retained at large clouds today, but many tools rely on logs and telemetry data for root-cause diagnosis.

We instead propose retention policies over logs and telemetry that are collected from production services *during incidents* to enable supervised learning strategies. The rarity of incidents enables retaining logs from the components involved to be feasible with minimal

overhead. Besides debugging, clouds can benefit from such retention policies in multiple ways. For example, clouds engage significant resources to ensure impacted customers are compensated according to the SLAs; third-party (e.g., judiciary, customers) audits can also be uniformly performed to investigate potential privacy/security threats resulting from certain incidents as cloud usage increases.

Several challenging research questions can be answered with intelligent retention policies over logs such as tuning alert thresholds to improve incident detection accuracy, these include extracting root-cause signatures from logs; generating mitigation scripts for past incidents; quantitatively measuring the effectiveness of policies in reducing mitigation times; and developing standards for storing, querying and instrumenting logs.

2.5 Centralized Debugging Plane

The earlier questions and concepts suggest that the current distributed approach to debugging is time-consuming and inefficient, and should be augmented by a centralized, cloud-scale component that we call a Centralized Debugging Plane (CDP). Importantly, we are not proposing dismantling the highly successful modular, team-oriented approach to developing clouds. Rather, we advocate augmenting this approach with a *narrow* API to a CDP to enable a cloud-scale view of incidents. Our vision is inspired by the Software Defined Network (SDN) movement where a centralized controller orchestrates individual switches; in our case, the CDP can be implemented using software deployed on a set of servers.

The CDP collects coarse-grained, minimal information from individual teams, and uses this information to adjudicate when there is contention between teams (e.g., to orchestrate incident routing). The CDP also provides a long-term store for debugging data (for long term insights or hints to guide per-team debugging) and training examples for data-driven debugging. Like SDN, CDP is more than mere centralization (Table 2.2). We envision the following components to CDP.

Dimension	SDN	CDP
Functionality	Maximize resource utilization, Flexible access control	Speed up mitigation, Consumable root-cause analysis
Scale	Network-wide centralized observation, route computation	Cloud-wide centralized observation, labelling and recommendations
APIs	OpenFlow	Open Telemetry
Structure	Controller, Network, Virtualization	CDGs, Extensive root-cause taxonomy, Quick fixes
Enabling Technologies	NoSQL databases, Compilers, Optimization algorithms	ML, NLP, Data warehouses, Coarsening algorithms

Table 2.2: Comparing SDN and CDP architectures

2.5.1 Global data lake

At the core of the CDP is a real-time data lake that provides a global view of alerts, incidents, telemetry, and logs from service teams. We aim for a fine balance between effective coordination and team sovereignty. In our vision, data is provided and managed by individual teams. However, to enable one team to easily discover and consume data from others, the data lake provides mechanisms such as: (1) A queryable global catalog describing data sets and metadata, (2) a uniform schema, (3) access control policies, (4) automation that continuously processes real-time telemetry and logs, and (5) policies to retain data.

We can leverage recent advances in building large-scale data lakes [Ama22b, Dat22a] and instrumentation frameworks such as Open Telemetry [Ope22a] to achieve these goals.

2.5.2 Incident data store

The global view provided by CDP’s data lake can help identifying all data (across teams) relevant to an incident. This enables CDP to use sophisticated retention policies: e.g., it can retain all data that are related to incidents for a long period of time and a small sample of failure-free data. Since the number of major faults is fairly small, the amount of data

retained for machine learning is manageable. A more speculative idea is to keep ML models (presumably smaller than logs) over very long periods to concisely capture how failure patterns evolve with time.

2.5.3 AIOps engine

CDP’s data lake provides a natural place for developing AIOps¹ solutions. For example, one can build AIOps solutions to (1) denoise telemetry and logs when they are injected into the data lake, (2) enrich incidents with additional metadata such as similar incidents, potential root causes, and related fixes learnt from retrospective analysis, (3) learn rules to prioritize and route incidents, (4) take automatic mitigation steps such as rebooting an unhealthy micro-service, and so on. Even though there are existing AIOps efforts[CKL20], CDP’s data lake makes it easier to experiment with new techniques, especially those that involve data from multiple teams.

2.5.4 Incremental deployment of CDP

CDP is incrementally deployable. First, CDP enhances, rather than replace, existing debugging efforts. Second, many CDP components can be incrementally built; e.g., the global data lake can be started with data from a small number of teams. Third, some CDP components can be constructed at a smaller scale. For example, the dependency graph of a service can be learned from its test deployment. Moreover, many cloud providers offer private clouds that mimic the architecture of their public clouds and can be deployed in a local cluster (e.g, Azure Stack[Mic22] or AWS Outposts[Ama22a]). We hypothesize that it is possible to construct various CDP components for a private cloud first and gradually transfer them to the public cloud.

¹ Artificial Intelligence for IT operations

CHAPTER 3

NLP-Powered Debugging Assistance

This chapter provides an overview of a joint work [DNS19b] with professors Karthik Narasimhan, Anirudh Sivaraman and Ravi Netravali. As discussed in § 1.3, despite the proliferation of debugging techniques and tools [NSN17, MRF15, KMB17, TAL15, GNU22, VNN13, MF18, MEH10], diagnosing and fixing bugs in systems remains challenging. Existing debugging tools ignore diverse types of data (e.g., bug reports, source code, comments, documentation, and execution traces) that engineers *collectively* use to make sense of system-wide semantics, bridging together outputs and features from existing debugging tools to debug production systems. Intrigued by the variety of abstractions provided by unstructured natural language data, we paint our vision for a debugging assistant that is powered by NLP techniques.

3.1 Auxiliary Data

The debugging process for developers typically entails reading a bug report, understanding its context in the system, reproducing the bug, iteratively asking and answering questions to identify its root cause, and then developing and testing potential fixes. Much of this process is manual, has many false starts, and requires significant developer familiarity with the system at hand—an increasingly elusive requirement as systems get more complex.

A key challenge for developers is identifying how to leverage the fine-grained operations they support (e.g., queries) when presented with a high-level bug or issue. For example, what debugging tool should the developer use when a service is deemed unreachable? What query

should they issue? At the same time, the amount of *auxiliary data* associated with software systems has rapidly grown, e.g., monitoring logs, execution traces, bug reports, patches, and code documentation. These data sources embed helpful debugging information [GMK16], but existing debugging tools fail to fully leverage them, and instead place the burden of extracting insights from this data on developers.

There are two reasons why this auxiliary data isn't fully exploited in today's debugging workflows. First, the auxiliary data sources, in contrast to source code, are highly unstructured and varied, ranging from custom logging formats (e.g., from debugging tools) to natural language bug reports. Techniques for automatic test generation [CDE08, MSP18] and program repair [LNF12, PKL09] rely on precise definitions of correctness and cannot fully leverage such unstructured data.

Second, in large systems, problems occur at the intersection of different system components. The above approaches, as well as existing monitoring and debugging tools, only instrument specific subsystems (e.g., network switches [HHJ14, NAR16, NSN17], end-host network stacks and operating systems [GNU22, MYG16, TAL16, TAL15], or applications [NGM16, VNN13]), but still place the onus of correlating information across subsystems on developers. Thus, existing tools have limited utility in debugging issues that arise due to the interaction between subsystems. For example, they are unable to automatically determine that an application timeout was the result of a routing black hole, since they fail to link together data from multiple disparate sources (in this case, network and application-level traces from an entire cluster).

3.2 NLP Powered System-Wide Debugging Assistant

Our thesis is that natural language processing (NLP) techniques are well-positioned to analyze the large amount of auxiliary data across multiple subsystems and extract insights that significantly enhance the current debugging experience. The natural language nature of

auxiliary data and the inexact nature of the debugging process require models that go beyond existing techniques that need exact and structured inputs. NLP can provide these models.

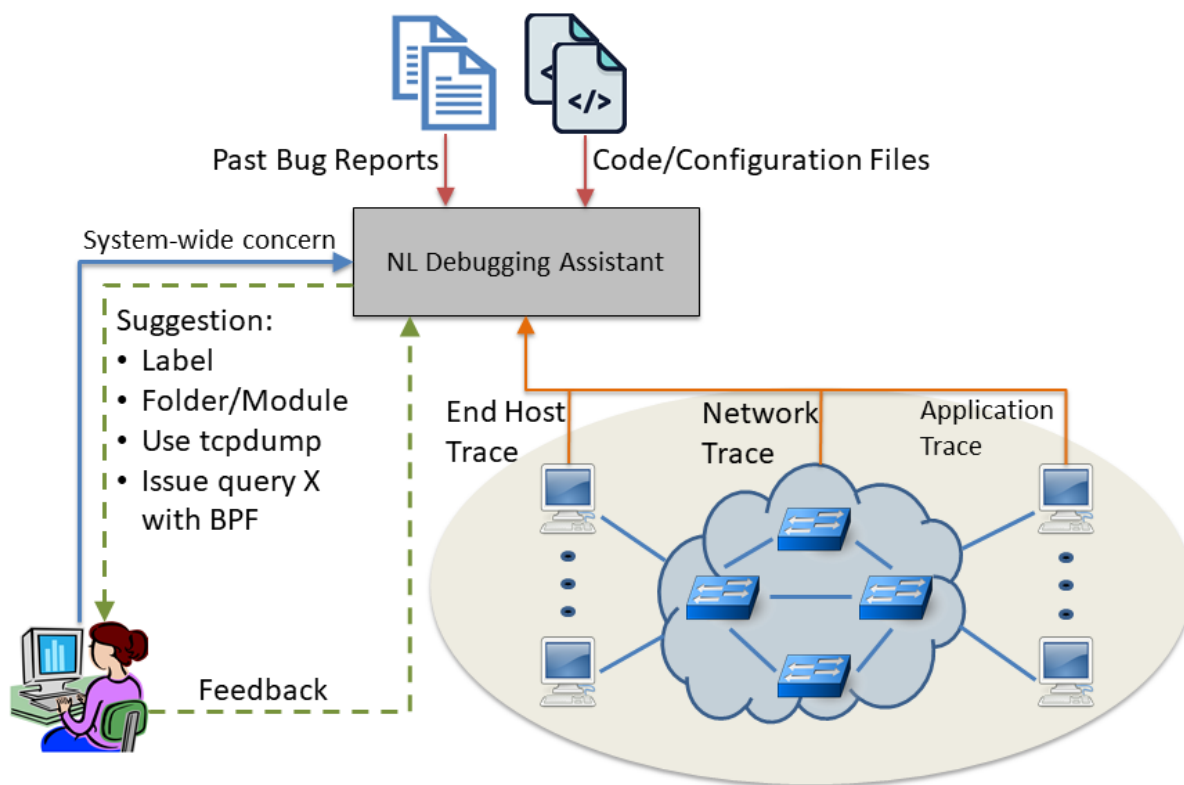


Figure 3.1: Overview of how our proposed debugging assistant improves different steps in a developer’s end-to-end debugging workflow. Developers begin by submitting system-wide bugs or performance concerns to the NL debugging assistant. The assistant generates hints (e.g., files to investigate) or actions (e.g., debugging queries to issue) based on analyzing past bug report data, design documents, tracing information collected throughout the system, and developer input. The process is iteratively followed until a bug is resolved.

Prior work has focused on specific sub-problems (e.g., generating Bash scripts from natural language [LWZ18] or extracting keywords from bug reports [PJN13]). Though promising, we believe there is a significant untapped opportunity for systems research that integrates these NLP techniques—and develops new ones—into a system-wide interactive debugging assistant

that facilitates and accelerates each step in a developer’s end-to-end debugging process (Figure 3.1). We identify three features that are crucial to such a system: a) preliminary diagnosis of incoming bug reports, b) automatic generation of debugging queries (for existing debugging tools) to monitor different subsystems, and c) taking multiple diagnostic or corrective decisions towards fixing the bug.

3.2.1 Opportunities for Automation

Developer intuition is invaluable to the debugging process and complementary to the automation afforded by techniques from NLP. Conceptually, imagine an NLP-powered debugging assistant running in the background to continuously ingest text from various sources: the bug report’s text, the bug report’s comments as they come in, the source code of the repository, and different traces. It then produces recommendations, e.g., assign a particular label to the bug report, look at this folder to diagnose the bug. These recommendations are shown to the developer, who can act on them and fix the bug, or provide further input to the debugging assistant based on their domain knowledge. This assistant can be applied to several different parts of the workflow:

1. *Preliminary diagnosis*: At the beginning of the workflow, the assistant can be used to perform diagnostics on incoming issues such as assigning labels or localizing the relevant subsystems in the project at various granularities, from top-level directories to individual lines of code. The assistant can also assign each ticket to the most relevant developer in order to streamline reviews. These tasks broadly fall into the framework of text classification and document retrieval. However, our setup presents a unique challenge of learning joint representations of the data that capture useful information from both unstructured text and structured source code.

2. *Generating debugging queries*: Another part of the workflow involves the assistant generating domain-specific queries to monitor different subsystems (e.g., BPF code for monitoring the kernel). This falls under the umbrella of language generation which plays a

key role in problems like machine translation or text summarization. The added challenge here lies in learning a model that can effectively use a diverse set of sources like issue text, system status information, and source code semantics to generate useful and syntactically correct debugging queries (e.g., queries in BPF [Wik22] or AppDynamics [App21]).

3. *Active (interactive) debugging*: Finally, the debugging assistant could take multiple diagnostic or corrective actions, each building upon previous actions and their results. For this, we will draw upon techniques for sequential decision making like reinforcement learning, with the goal being to perform the optimal sequence of actions to fix the problem, within constraints on the latency of performing these actions or the compute resources expended in the process. In addition, we can keep the developer in the loop to supervise the entire process, and simultaneously help fine-tune the assistant’s decision making. We imagine that developer and assistant actions will be interleaved to debug the issue efficiently and with minimal human effort.

Ultimately, our vision is an assistant that captures the hard-won debugging wisdom of the expert programmer in different parts of the workflow by exploiting the abundant data available within source code repositories and their associated issue trackers. Of course, across all of these use cases, we must provide system support to ensure a developer-assistant interface that seamlessly handles expressive input options from the developer and provides timely responses from the assistant. In particular, developers must be able to (but not required to) input case-specific debugging information (e.g., time limits, expertise levels), and receive responses in a way that does not add undue latency to any debugging stage.

3.3 Preliminary Experiments

We perform empirical evaluation of our models on real-world data sourced from publicly available code repositories, specifically 98 repositories hosted on GitHub. We collect text from closed issues (~240k) along with associated labels and pull requests containing details

on modified folders in the source code. This gives us supervised data for the following classification problems. Our results highlight the ability of NLP techniques to relate varied data sources (i.e., by learning a model across repos that are quite different in terms of topic, code, and structure).

3.3.1 Label Prediction for GitHub Issues

We formulate the label prediction task as a standard text classification problem. As an initial foray, we use a bag-of-words representation to convert the text in each issue into a suitable vector. Formally, consider a dataset $\mathcal{D} = (x, y)$ containing pairs of issue text (x) and their corresponding labels (y). Since each issue may have more than one valid label, our task is a multi-label classification problem [TK07]. Therefore, we consider each y to be a one-hot vector of size $|L|$, where L is the set of all possible labels, with each entry in the vector being 1 or 0 depending on whether a particular label applies to the issue or not. Our goal is to train a model to accurately predict as many labels as possible.

The key challenge lies in learning an appropriate representation for the bug reports available in textual form. This representation should be able to capture the semantics of the issue sufficiently to predict accurate labels. There are several techniques and models that have shown considerable promise in NLP such as word embeddings [MSC13, PSM14] or LSTM recurrent neural networks [HS97], which convert discrete textual symbols into a real-valued vector representation. As an initial foray, we use a bag-of-words representation to convert the text in each issue into a suitable vector: $\phi(x)$. Each entry in this vector corresponds to the number of times a particular word appears in the text (most entries will be 0). We train a linear classifier f to predict probabilities for each label from this representation:

$$f_{\theta}(\phi(x)) = W \cdot \phi(x) + b; \quad \hat{y} = \sigma(f_{\theta}(\phi(x)))$$

where W is a matrix of weights, b is a bias vector, σ is the ReLU function and \hat{y} is a vector of predictions over all labels. We train our model by minimizing the binary cross-entropy loss (over all the labels) with respect to θ using stochastic gradient descent [Bot10]:

$$\mathcal{L}(\theta) = - \sum_i [y_i \cdot \log \hat{y}_i + (1 - y_i) \cdot \log(1 - \hat{y}_i)]$$

3.3.2 Source Code Folder Prediction for GitHub Fixes

Moving down a level of granularity, we also consider the task of predicting files/folders that might be relevant to a particular issue, and would be useful for a developer to look at. This is a challenging task to automate since it requires a semantic understanding of both natural language text as well as the semantics of each folder (and its contents, e.g., source code).

As a first step, we focus on predicting folders that appear in change-lists linked to issues. This is inherently similar to the problem of information retrieval, where the goal is to return relevant documents given a natural language query. In our case, the the issue is a query and our goal is to return a (ranked) list of relevant folders in the project. Our key requirement is to learn good representations and a similarity metric between issues and folders.

Assume each instance in our dataset is a pair (x, z) , where each x is an issue text and each z is the text corresponding to a single folder (e.g. folder name). Our goal is to learn a similarity function $\psi(\phi_1(x), \phi_2(z); \theta)$ which can be used to predict relevant folders given a new issue x' . Here, the ϕ s are again suitable representations for converting text into a real-valued vector; we use the same bag of words (BOW) representation as previously described. For the similarity function ψ , we train a 2-layer neural network that operates on the concatenation of both BOW vectors $[\phi_1(x); \phi_2(z)]$ to predict the probability of a folder being relevant: $\hat{P}(y = 1|x, z)$.

We treat the pairs in the dataset as positive examples of matches, and generate pairs of negative examples ($\mathcal{D}' = (x', z')$), by randomly matching issues to a folder in the code that is

not relevant. With this, we can train our model by minimizing the following loss function with respect to θ using stochastic gradient descent.

$$\mathcal{L}(\theta) = -\left[\sum_{(x,z) \in D} \log \hat{P}(y = 1|x, z) + \sum_{(x',z') \in D'} \log \hat{P}(y = 0|x', z') \right]$$

3.3.3 Debugging Query Generation

We envision our assistant to automatically generate syntactically correct queries for systems and network debugging tools (e.g., Marple [NSN17], GDB [GNU22]) to aid the human debugging process. We formulate this as a contextual language generation problem, where the system takes user-written issue text x as input¹ and generates a structured debugging query q . As an initial foray, we perform template-based generation [RD00], where we train a classifier (f) to predict the most relevant template \mathcal{T} for an issue and then predict values for the slots in the template to generate q :

$$\hat{\mathcal{T}} = f(x); \hat{q} = g(\hat{\mathcal{T}}, x)$$

The classifiers are trained using ground truth data collected using our setup (described in greater detail in Chapter 4), using cross entropy as the loss function during training. We stress that template-based generation is a first step. In the future, we plan to investigate more sophisticated generation models such as recurrent neural networks [HS97] and transformers [VSP17], which have shown considerable promise in tasks like language modeling and machine translation.

¹ The input could also consist of other signals like system status.

3.3.4 Results

We use standard classification metrics of precision, recall and F-1 scores. For the label prediction task, we also report accuracy, considering a case to be correct if the set of predicted labels exactly matches the set of true labels. For folder prediction, we also report scores for mean average precision (MAP), which is an aggregate metric over precision at various levels of recall. We achieve 77.8% accuracy and 0.77 F-1 on label prediction (Table 3.1), which is quite promising for a *multi-label* classification problem over 5 classes.

Task	Precision	Recall	F-1
<i>Label Prediction</i>	0.76	0.78	0.77
<i>Folder Prediction</i>	0.74	0.76	0.75
<i>Query Generation</i>	0.82	0.67	0.76

Table 3.1: Results for label and folder prediction, as well as template-based query generation tasks.

For comparison, a random baseline would achieve 20% on a simpler *single-label* classification problem. On folder prediction, we achieve an F-1 score of 0.75 and a MAP score of 0.72 for predicting relevant folders from more than 160 folders in the entire repository. This is significantly higher than a random baseline which would get a MAP score of 0.0062.

Figure 3.2 lists qualitative examples of our model predictions. These results highlight the ability of NLP techniques to relate varied data sources (i.e., by learning a model across repos that are quite different in terms of topic, code, and structure).

Our results also show early promise for template-based query generation. We achieve a MAP score of 0.82 and an F-1 score of 0.76 for generating relevant Marple queries (Table 3.1). Figure 3.3 shows an example query output for a scenario where a system component has failed. As shown, the model correctly predicts both the appropriate query template to use and the switch to issue it on.

- **Issue 1:** *I want to be able to access a specific resource variable within that resource. For example : run a provisioner for an instance and supply it with the instance private ip (or id or anything else).*
True Labels: CORE, ENHANCEMENT
Predicted Labels: CORE, ENHANCEMENT
- **Issue 2:** *The menu panel not being closed when its ‘overlayref‘ is detached externally using ‘detach‘ for example when using the ‘closestrategy‘. ****note:**** this is a re-submit of #8654 due to some sync issues.*
True Folder: `src/lib/menu` **Model score:** 0.99
False Folder: `src/tools/dashboard` **Model score:** 0.0

Figure 3.2: Examples of label & folder predictions for two repos: hashicorp/terraform and angular/material2.

- **Issue:** *Took a while and the webpage says ‘You broke reddit’ and ‘Funny 500 page message 3’. Upon refreshing the page it says ‘Funny 500 page message 6’. Upon further loads, the browser is stuck on ‘waiting for 10.0.0.2’*
- **Actual Fault:** mcrouter instance down.
Relevant Query: `stream = filter(T, switch==1); result = groupby(stream, [srcip, dstip, srcport, dstport, proto], count);` **Model Score:** 0.94
Irrelevant Query: `stream = filter(T, switch==5); result = groupby(stream, [srcip, dstip, srcport, dstport, proto], count);` **Model Score:** 0.01

Figure 3.3: Example query predictions for debugging a given issue.

3.4 Related Work

The following related work offers a glimpse into the ability to extract meaning from natural-language auxiliary data present in software projects. They also illustrate the benefits of combining information from natural language and source code [Ern17]. However, these tools fall short of realizing our vision. First, these approaches are limited to ingesting data from a single subsystem. However, our target distributed systems scenarios require extracting

and relating diverse data (e.g., bug reports, source code, code comments, execution traces) from multiple subsystems (e.g., the network and applications on end hosts). Second, all of these approaches assume a single-step process, where the NLP system has to perform a single prediction. In contrast, we focus on the end-to-end system debugging process that is iterative by nature.

3.4.1 Program analysis and synthesis

NLP techniques have been utilized in multiple aspects of software development [Ern17]. Examples include detecting operations with incompatible variable types [HCE15] and converting natural language comments into assertions [GGE16]. More recently, NLP has also been used in code generation by allowing developers to specify requirements in high-level natural language in the forms of regular expressions [LND16], Bash programs [LWZ18], API sequences [GZZ16], and queries in domain specific languages [DGH16].

3.4.2 Program debugging

NetSieve [PJJ13] used NLP to parse network trouble tickets by generating a list of keywords and using a domain-specific ontology model to extract ticket summaries from those keywords. While NetSieve automates parsing, significant manual effort is still required in (1) offline construction of an ontology model and (2) determining what constitutes a keyword. In contrast, we seek to build models that learn automatically from data, with minimal manual effort. Net2Text [BDV18] translates English queries into SQL queries, issues those queries, summarizes the results, and translates them back into natural language for easy interpretation. We aim to go further and automatically determine which queries to issue based on bug reports, debugging traces, and source code. Recent bug localization work uses information retrieval techniques [ZZL12, NPK13, KTK13], but requires manual feature engineering.

3.4.3 Big Code

Recent efforts such as the Big Code initiative [WST16] perform statistical program analysis to take advantage of the large amount of code in existence today, with the goal of extracting insights to aid code generation, refinement, and debugging. Learning techniques have been used to identify comments that are largely redundant with source code [LDB18] or generate natural language summaries of source code [PWY18, YWC18]. We view work in the Big Code initiative as contributing to some of the individual building blocks of our proposed debugging assistant. However, significant effort is required to integrate these building blocks.

CHAPTER 4

Generating Debugging Queries

This chapter provides an overview of a joint work [DNS22] with Shiv Saini from Adobe Research and professors Karthik Narasimhan, Anirudh Sivaraman, George Varghese and Ravi Netravali. As discussed in §1.3, existing root-cause analysis techniques focus on specific types of issues and only interface with specific types of logs. A major difficulty in debugging lies in *manually* determining which of the many available tools to use and how to query its logs. Engineers use debugging queries to interface with these tools and validate their root-cause hypotheses. We investigate the problem of automatically suggesting debugging queries that engineers can use for root-cause analysis.

Specifically, we explore the idea of *centralized* generation of debugging queries which can be issued over logs across different subsystems, tools and suggest the underlying root-cause to a developer who is familiar with the semantics of a component. To do this, we *abstract* the symptoms of an incident *hierarchically* through: incident user reports (service-level symptoms), ordered log features (subsystem-level symptoms) and ranking (component-level symptoms). To *scale* down the complexity of query generation, we factorize the task of debugging query generation into: (1) Template Prediction (indicates likely root-cause hypothesis) and (2) Parameter Filling (indicates likely components where root-cause originates). We also conduct a study at a large service provider to understand the types of root-cause that occur in production systems. We use insights from the study to motivate our solution and the design of a realistic testbed to evaluate our technique.

4.1 Debugging Queries

Root-causes of incidents or their semantics cannot be expressed structurally across different systems, due to varying interpretations and data types. However, we observe that engineers often translate informal reports about problems provided by a user into actionable information that identifies the root cause of a bug. To do this, engineers execute structured queries over logs to reason and validate their hypothesis of a root-cause. We call these *Debugging Queries* (example query in Figure 4.1) and the results of their execution can sufficiently validate a developer’s specific root-cause hypothesis, to conclude root-cause analysis.

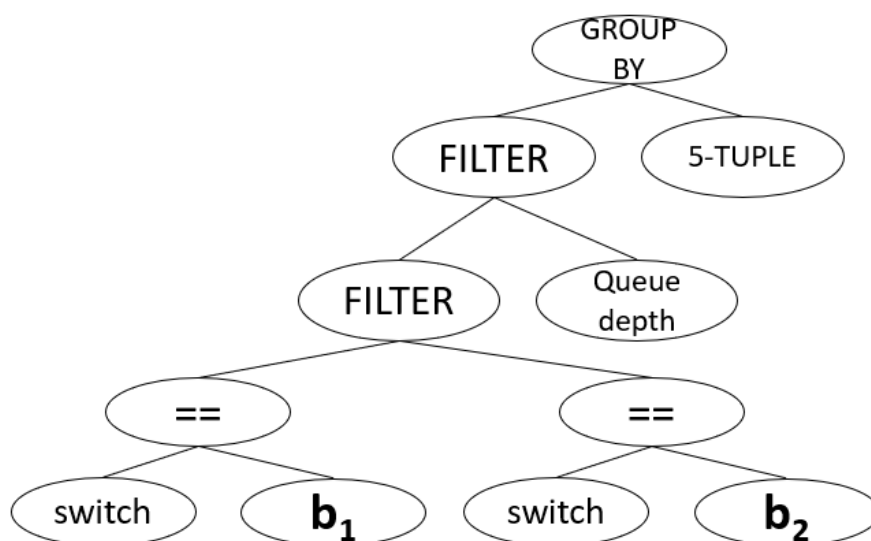


Figure 4.1: Example of a debugging query in its syntax tree representation issued over Marple’s switch queue depth counters from two switches ‘b1’ and ‘b2’.

The above figure shows the syntax tree representation of the following debugging query over Marple [NSN17] counters from P4 [BDG14b, BDG14a] programmable switches in a network. An engineer can issue this debugging query if they hypothesize that traffic originating from a faulty server is causing high queue depths in switches ‘b1’ and ‘b2’. The result set produced

by this query is sufficient for an engineer to mitigate an incident by shutting down the faulty server or invalidate their hypothesis.

4.2 Overview - REVELIO

We propose REVELIO, a tool to automatically generate a ranked list of debugging queries that developers can execute over logs to identify the root-cause of an incident. At a high level, REVELIO takes two inputs: (1) a *user report* filed by a system user, and (2) the *system logs* collected during the user’s interactions with the system. The two sources provide distinct perspectives into the state of the system when a fault occurs—the former from an external and the latter from an internal viewpoint. Further, the two data sources differ fundamentally: system logs are highly structured, accurate, and contextually close to a developer’s debugging options; user inputs are often noisy, unstructured (e.g., raw text), and abstract with respect to low-level execution (e.g., a user may report that the system is slow to respond with no further information). As the output, REVELIO generates a ranked list of top-k debugging queries that are directly executable on the target debugging framework(s) and highlight the root cause of the fault.

Figure 4.2 shows *Coarsening* through mapping root causes to debugging queries generated by REVELIO. Element c in the concrete poset indicates all the incidents in the cloud caused by a network-related root cause (very coarse). c_1 indicates all the incidents in the cloud caused by a packet loss related issue in the network (coarse). c_{12} indicates all the incidents in the cloud caused by a packet loss related issue at a specific link in the network (very fine).

4.2.1 Challenges

REVELIO must overcome four key challenges to generate debugging queries. First, it has to combine and relate diverse and seemingly disparate data inputs. Second, the output space of queries is highly structured, making it harder than standard multi-label classification where

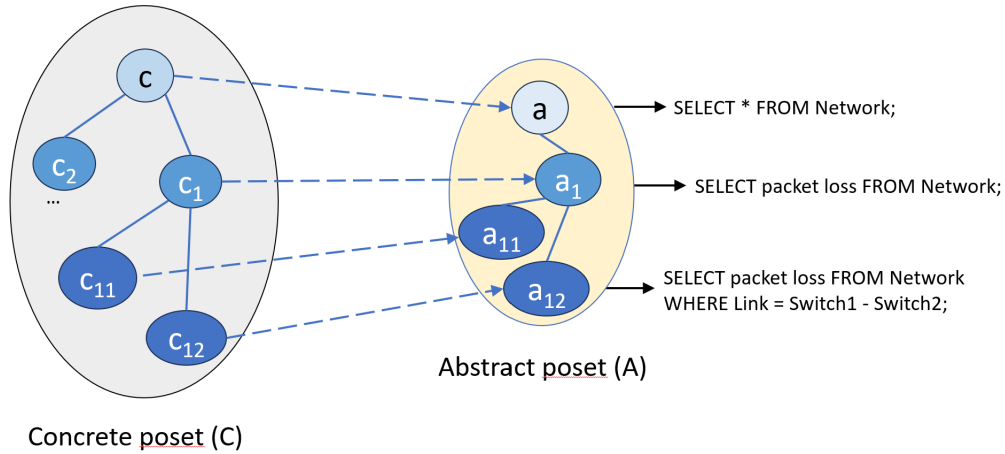


Figure 4.2: REVELIO as an implementation of *Coarsening* through Debugging Queries.

each label is independent [BHS07]. This is because all debugging queries for a tool are drawn from the same language grammar, unlike opaque and independent labels. Third, the space of potential queries for a given input is large, requiring new techniques to scale to large distributed systems. Fourth, the model must generalize in a specific sense: if a fault occurs at one location during training and is debugged with a specific query, then, during testing, the model must predict the same query with a different parameter if the same fault occurs at a different location.

4.2.2 Solutions

Challenge 1: Diverse data. We handle diverse data sources by converting each into a vector and concatenating all vectors to form the system state vector (Figure 4.5). This has two benefits. First, each data source is normalized for downstream operations in the ML model. Second, the architecture is extensible: a new data source (e.g., crash reports) can be added by converting it into a vector (either learned or manually) that is then concatenated with the existing system state vector.

Challenge 2: Predicting queries. To generate queries, which can be represented as abstract syntax trees (ASTs) in the grammar of a tool’s query language, we employ a Graph Convolutional Network to convert the AST into a query vector. A vector-based representation is easier to use with the rest of the ML model relative to richer representations such as trees. During training, given pairs of query and system vectors, we find model parameters that maximize the probability that these query vectors were predicted from these system vectors. During inference, given the ML model’s parameters, we find the query that maximizes the probability of a query vector given the system vector.

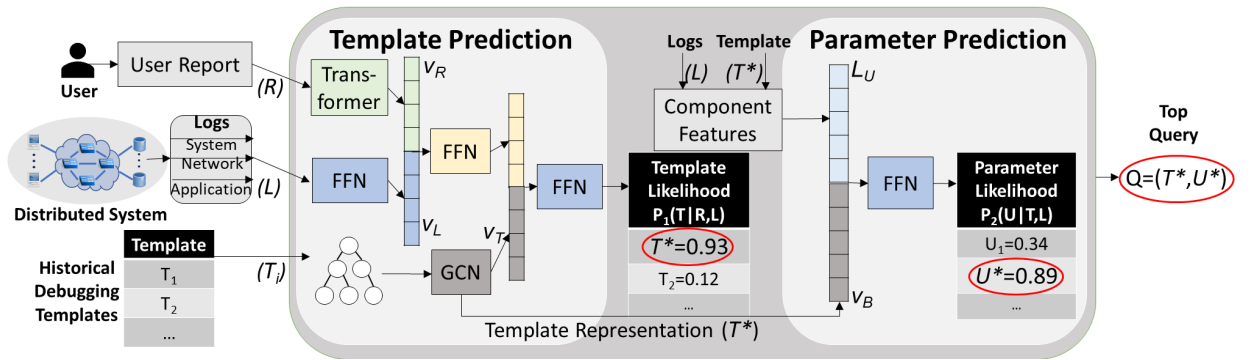


Figure 4.3: Overview of REVELIO’s factorized, 2-phase approach to generating debugging queries for root cause diagnosis.

Challenge 3: Scaling to large systems. REVELIO has to search over a large space of queries to output the best query in response to a given input. This search space scales with the size of the distributed system. To handle this, we exploit modularity and factorize our ML model into two cascaded components (Figure 4.3). The first uses user reports and system logs to generate query templates, which are skeleton queries for a particular subsystem with all numeric parameters left unspecified (e.g., `SELECT _ FROM _`). The second component then predicts the corresponding parameters using only the predicted template and system logs. This approach is motivated by two ideas. First, production faults typically involve recurring types (based on our study), and can thus be debugged using a small number of

templates (one per fault type). Second, we assume that system logs sufficiently highlight the set of potential parameter values and the relative importance of each; user reports are often abstract and rarely list parameter values (e.g., switch IDs). Modularization thus shrinks the output space of the first model, simplifying training computationally, regardless of system scale. It also shrinks the input space of the second model, making it less likely to overfit to spurious inputs, which in turn improves accuracy and generalizability.

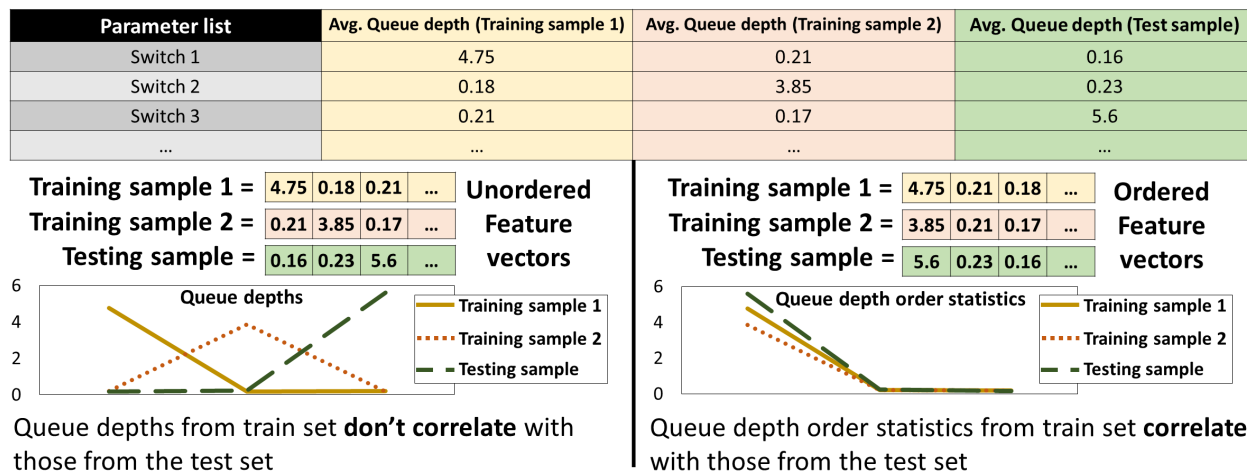


Figure 4.4: Example showing how rank-ordering helps to generalize to faults of the same type at different locations. After ordering (right), despite the fault location being different, the queue depth order statistics in testing are correlated with those in training. In contrast, without ordering (left), the unseen fault location results in queue depth values that are dissimilar from training data.

Challenge 4: Generalizing to new fault locations. Given the scale of production systems, it is infeasible to rely on training data that captures all possible locations of a given fault category. Thus, our model should generalize to *different locations* for fault types seen during training. To aid with such generalization, we convert concrete switch/function ids in the system logs into abstract ids based on the rank order per feature (e.g., queue depth). This allows our models to learn the relevance of a given template or the importance of a particular subsystem based on a stable property like the subsystem’s rank on a feature rather than a

volatile property (Figure 4.4), e.g., switch ID. For example, during template prediction, the model is able to learn about the applicability of a template to the order statistics [Wik21] of feature values across the system, rather than to the numerical or ordinal values of these features at specific subsystems. This is important because, if a given fault occurs at two different locations (both of which warrant the same template), the order statistics of feature values may be correlated, whereas the specific value assignments definitively will not. Similarly, for parameter prediction, ordering information is more robust to the addition, deletion, or restructuring of subsystems.

4.3 REVELIO’s ML Model

To enable Revelio’s prediction capabilities, we need to induce a distribution $\mathbf{P}(Q|R, L)$ where Q is a debugging query, R is a user report, and L refers to the system logs (Table 4.1 lists the model’s variables).

Name	Description	Example
T	Query template	SELECT QUEUE_SIZE FROM T WHERE SWITCH_ID = _
B	Blanks in template $\{b_1, b_2, \dots, b_z\}$	$b_i = _$ in the above example
U	Query parameters $\{u_1, u_2, \dots, u_z\}$	$u_i =$ switch ID
R	User report	“Page is loading slowly”
L	System logs	OpenTracing and Marple logs

Table 4.1: Variables in Revelio’s ML model. Figure 4.6 lists example input values for each.

Once the parameters of this distribution have been learned by maximum likelihood, the distribution allows us to predict the query Q that maximizes $\mathbf{P}(Q|R, L)$. The required data is a set of triples $\langle R, L, Q \rangle$. While the above formulation seems straightforward, it involves learning a probability distribution over all possible queries and across all tools, which is

challenging and requires a substantial amount of data. Thus, we instead split up each query Q into a query template T (e.g., `SELECT _ FROM _`) and a set of values U (to fill in the blanks). This lets us factorize the prior distribution as:

$$\mathbf{P}(Q|R, L) = \mathbf{P}(T, U|R, L) = \mathbf{P}_1(T|R, L)\mathbf{P}_2(U|T, R, L) \quad (4.1)$$

To simplify our training further, we make an independence assumption on P_2 by assuming that R is not likely to help predict U (as described in §4.2.2). Thus, we have:

$$\mathbf{P}_2(U|T, R, L) = \mathbf{P}_2(U|T, L) \quad (4.2)$$

We can further factorize this into a product of distributions over values u_i for each blank b_i in the template T :

$$\mathbf{P}_2(U = \{u_1, u_2, \dots, u_z\}|T, L) = \prod_{i \in [1, z]} \mathbf{P}_2(u_i|b_i, T, L) \quad (4.3)$$

where z is the total number of blanks in the template.

From an inference standpoint, this means we have a 2-phase query generation process: we first generate a query template and then fill in the blanks with appropriate values using the system logs (Figure 4.3). We next detail how we model each of the distributions (\mathbf{P}_1 and \mathbf{P}_2), as well as our learning and inference procedures for each.

4.3.1 Predicting Probabilities for Query Templates

Assume the user report R to be in the form of raw text and L to be a vector obtained by concatenating ordered vectors for each feature (Figure 4.5) extracted from the system logs (e.g., time-windowed average, min queueing delay). Recall from §4.2.2 that rank ordering per feature in L enables our model to learn about the order statistics of feature values across

subsystems, rather than about numerical or ordinal values at specific subsystems (Figure 4.4). From here, a straightforward way of modeling $\mathbf{P}_1(T|R, L)$ would be to use a multi-label classifier with each template T being a different label. However, as discussed in § 4.2.2, query templates are structured and made up of smaller atomic components (e.g., IF, MAX statements). In other words, the ASTs of many query templates share common subtrees. Thus, *simply treating each template as an independent output label is wasteful in terms of not sharing statistical strength.*

Therefore, we adopt a different approach to modeling the output templates. In order to preserve the structural aspects in queries, we represent each template T in the form of an abstract syntax tree (AST). Each node in the tree is an operator (e.g., SELECT) and the edges represent how the operators are composed together to form larger trees.

We use a Graph Convolutional Network (GCN) [KW16] to construct a vector representation v_T for each query template’s abstract syntax tree. The GCN updates each node’s vector representation in the AST by pooling information from all its neighbors and performs this process multiple times, allowing it to combine information from all nodes in the tree. The GCN outputs a vector for each node in the tree – we take the vector of the root node v_T to represent the tree’s information. In parallel, we use a contextual text encoder (BERT) [DCL18] to convert the issue report R into a vector v_R and pass the log L through a linear neural network layer to get a vector v_L . v_R and v_L are concatenated and fed through a non-linear layer followed by a linear layer to get a single vector v_S representing the system state from both internal and external viewpoints. Finally, we use both v_S and v_T to obtain a measure for how likely the template T is applicable to the debugging scenario $\langle R, L \rangle$ (i.e., the probability of T given R and L). We then search for a set of neural network parameters that maximize this score (S) or likelihood. The sequence of operations are summarized as:

$$\begin{aligned}
v_T &= \text{GCN}(T)[\text{root}] \\
v_R &= \text{BERT}(R) \\
v_L &= \text{LINEAR}(L) \\
v_S &= \text{LINEAR}(\text{RELU}([v_R; v_L; v_T])) \\
S(T, R, L) &= \text{LINEAR}(\text{RELU}([v_S; v_T])) \\
\mathbf{P}_1(T|R, L) &= \text{SOFTMAX}(S(T, R, L)) = \frac{S(T, R, L)}{\sum_{T'} S(T', R, L)}
\end{aligned}$$

where $[\cdot]$ represents a concatenation of two or more vectors and $\text{GCN}(T)[\text{root}]$ represents indexing the output of the GCN to get the vector of the root node.

The above operations represent a continuous flow of information through a *single* DNN whose parameters θ can be trained through back-propagation and stochastic gradient descent [GBC16]. We use the following maximization objective to learn the parameters:

$$\max_{\theta} \mathcal{L}(\theta) = \sum_{(T, R, L) \sim \mathcal{D}} \mathbf{P}_1(T|R, L) \tag{4.4}$$

Enumerating all trees T' is intractable, so we employ Noise Contrastive Estimation (NCE) [GH10] and draw $m = 2$ negative samples to form each T' to approximate the objective.

4.3.2 Predicting Values to Fill Query Templates

Now that we have a method to pick a template T^* , we must fill in the values for each blank b in T^* . Each template implicitly specifies the type of subsystem that is relevant for the fault at hand. Thus, using the template, we first extract a list of all relevant subsystems from the system logs L . For each subsystem u in this list, we have a feature vector L_u which

summarizes all of its logs (Table 4.5). We also include ranking information $rank_u$ for each feature in L_u (e.g., u 's rank in queue depth across all switches). Note that ranks embed the same information as ordering from § 4.3.1; ordering is not possible here because each L_u pertains to only one subsystem. We use these features, along with a vector representation of the blank in the template (described below), to pick the most likely subsystem for the blank.

We feed the template (AST) T^* through the same GCN module as in § 4.3.1 and choose the vector representation for blank b to be the output vector of its corresponding node in the tree. This allows us to represent the requirements of b using the properties of its neighboring nodes in the AST. Our goal is to then pick the *most suitable subsystem* u for the blank, and return the corresponding system identifier (e.g., IP address or port number). We use similar operations to those in § 4.3.1 to pick the most likely u to fill b :

$$\begin{aligned} v_b &= \text{GCN}(T)[b] \\ S(u, b, T, L) &= \text{LINEAR}(\text{RELU}(\text{LINEAR}(v_b; L_u; rank_u))) \\ \mathbf{P}_2(u|b, T, L) &= \text{SOFTMAX}(S(u, b, T, L)) \end{aligned}$$

where $rank_u$ indicates the rank of subsystem u in its subsystem's logs L , based on the feature of interest (e.g., rank of a switch, across all switches, on mean queue depth). We then use an objective similar to Eq. 4.4 to maximize \mathbf{P}_2 over ground truth data and learn the model parameters ϕ .

4.3.3 Choosing the Final Queries

Once each of the two models above have been trained, during inference, we find the combination of query template and query parameters that maximizes the probability that the resulting

query would result from the given system state vector. This probability in turn is the product of the two probabilities predicted by each of our models \mathbf{P}_1 and \mathbf{P}_2 .

$$Q^* = (T^*, U^*) = \arg \max_{T, U} \mathbf{P}_1(T|R, L) \mathbf{P}_2(U|T, L) \quad (4.5)$$

We can also pick the top k most relevant queries, rather than just the single most relevant one, using the ranking produced by the probabilities above. If $|T| \times |U|$ proves to be very large, we can approximate the above by considering only the top few templates according to $\mathbf{P}_1(T|R, L)$.

For all FFN layers in our ranking model, we use two linear layers, each with hidden size 300, along with ReLU non-linearity. The GCN also uses a hidden vector size of 300. We use the Adam optimizer [KB14] with a learning rate of 0.0001.

4.3.4 Diagrams Illustrating Model Operation/Insights

Figure 4.5 shows an example of generating feature vector from system logs (counter logs from programmable switches) to input to REVELIO. In this example, each P4 programmable switch in the network logs queue depth counters at each interface’s ingress port and periodically streams these values to a log collection database.

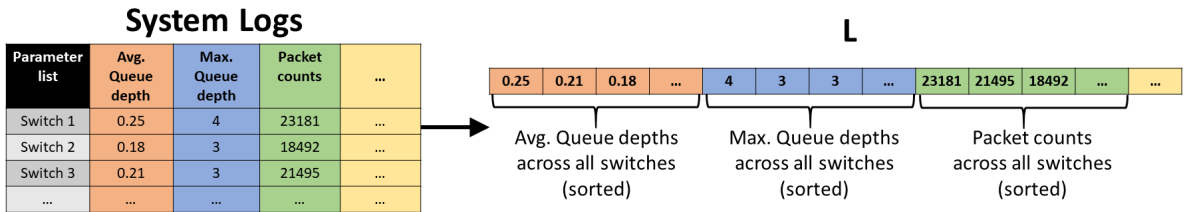


Figure 4.5: Example illustrating the generation of system log vector L ; for simplicity, the example considers only network logs. Values for each feature (across switches) are first rank ordered, and then the resulting lists are concatenated to form L .

We first aggregate these queue depths at each switch by using standard statistical measures (mean, std. dev., max, quartiles, 90th, 95th, 99th percentiles) to obtain a table as shown in Figure 4.5. For each of these statistical measures, we append the values to a feature vector in decreasing order across all switches and note the rank of each switch's value for that statistical measure. We then concatenate feature vectors of all statistical measures to form a feature vector which is used for predicting probabilities of query templates for an incident. The rankings of each switch for each statistical measure is used later to predict parameters to fill in the chosen query template. Figure 4.6 shows example values for all variables used in REVELIO.

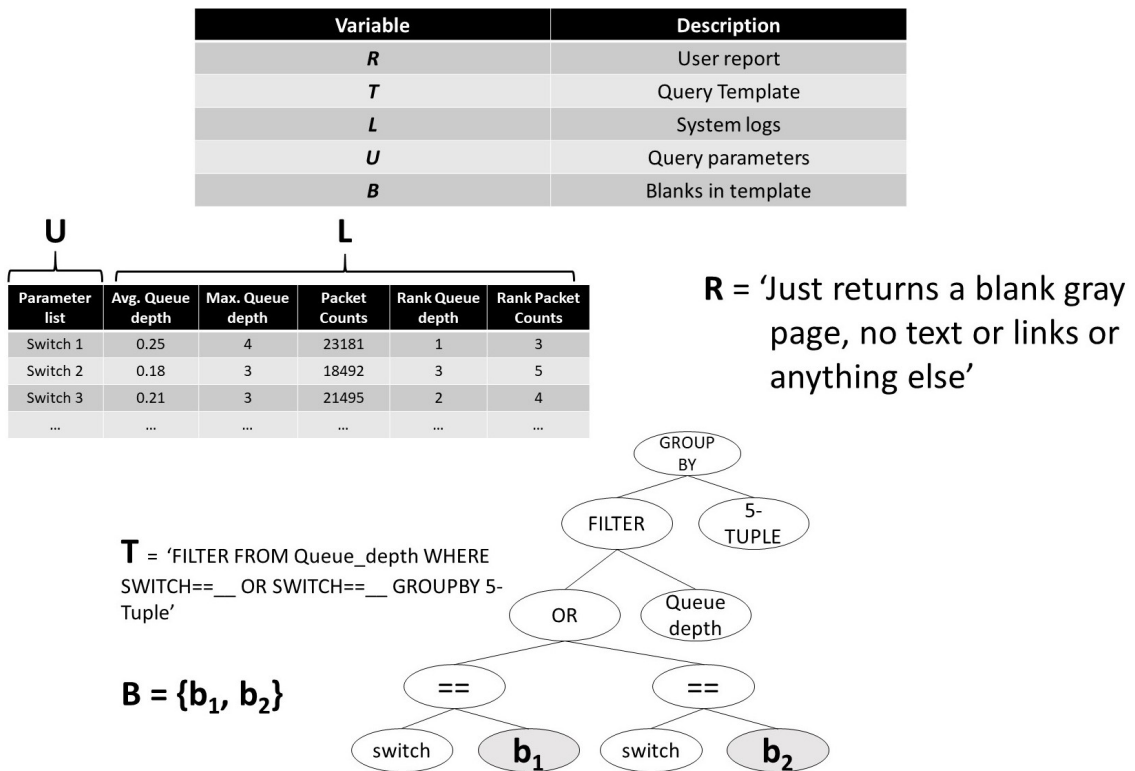


Figure 4.6: Example inputs for each input variable in REVELIO's model. This example is for a network (Marple) query. For the query template (T), the entire tree represents the template, while the parameters to be filled in are shaded in grey.

4.4 Study of Production Incident Debugging at *Anon1*

To understand the operation and limitations of debugging tools and workflows in production distributed systems, we conducted a study at a major SaaS company (*Anon1*). Our analysis involved 7 services at *Anon1* that collectively handle 83 million user requests per day. Across these services, we examined the debugging process through a developer survey and a manual analysis of completed debugging tickets over a 4-month time period.

4.4.1 Insights from Debugging Workflows

Developers at *Anon1* use a variety of state-of-the-art monitoring tools (e.g., Splunk [Spl21], Datadog [Dat22b], others [Lig21, New22, Pin21, Ici21]) that continuously analyze system logs, visualize that data with dashboards, and raise alerts when anomalous or potentially buggy behavior is detected. Alerts are raised on a given time-series based on either manually-specified heuristics and thresholds, or standard statistical analysis techniques that compare recent data to historical baselines [Lig21, TL18, XCZ18, Twi15]. As user or internal reports are filed, the burden of debugging falls largely to developers. For each report, developers must (1) filter through the raised alerts (across subsystems) to determine which are worth investigating and pertain to actual bugs and the issue at hand (vs. false positives), and (2) for bugs, find the root cause by analyzing the system as a whole. Both steps involve iteratively analyzing low-level system logs, inspecting prior debugging tickets and the current report (both written in natural language), and issuing debugging queries (using interfaces that run atop the same logs used to raise alerts [Gra22, NSN17]). Once a root cause is identified, a summary of the issue, bug, and debugging process (e.g., investigated subsystems, issued queries) is documented as a completed ticket.

4.4.2 Insights from Production Incidents

We manually analyzed all 176 debugging tickets that were created for the aforementioned services between November 2019 and February 2020. Our analysis involved manually clustering the tickets according to their root causes (as documented by *Anon1*'s developers). We make three primary observations from this analysis:

1. A few recurring categories of root causes collectively represent the vast majority (94%) of bugs.
2. The faults in a given category often manifest at different locations in the distributed system. For example, numerous “Resource underprovisioning” tickets involve high CPU loads but pertain to different servers, e.g., gateway vs. storage servers.
3. Identifying the root cause for a fault is time consuming, taking an average of 8.5 hours (min: 14 min, max: 2.9 days). We found that these lengthy durations are largely a result of the error-prone nature of root cause analysis: developers at *Anon1* must explore multiple subsystems (5 on avg.) and issue many debugging queries (8 on avg.) to find the root cause of a problem.

4.4.3 Literature Survey of Incidents

Our analysis and results (as shown in Table 4.2) follow a general taxonomy based on a literature survey we conducted of publicly reported bugs in production distributed systems. Our survey includes major outages in large-scale services (e.g., Dropbox [Dro19], Kubernetes [Sal17]), bugs in cloud services (e.g., Google [SBB10], Facebook [KMB17], Azure [LLM19c]), and experiences with open source systems (e.g., Cassandra, HDFS [YLZ14a]). Our survey revealed the following bug categories:

1. **System software and configuration faults.**

Root Cause Category	# of Tickets	# Unique Locations	Example Root Cause	Average Diagnosis Time
Resource under-provisioning	17	11	Load balancer is consuming all available memory and starving other co-located services	293
Component failures	58	29	3 nodes for a service were down, leading to queued 400 ERRORS	176
Subsystem mis-configurations	11	7	Incorrect host mapping configuration in Zookeeper caused failure, and prevented cluster from servicing any events	276
Network congestion	5	4	A spike in wide-area traffic caused unusually low data transfer rates between two cities	725
Network-level misconfigs	18	10	Instances in a region are pointing to a NAT instance with incorrectly configured security groups, leading to dropped traffic	92
Subsystem/Source code bugs	31	22	Service returning 5xx errors due to a code change that added a condition on the availability of a parent asset ID	1607
Incorrect data exchange	26	16	4xx errors were being raised because the noise classifier service is sending additional data with each stock request	417
One-off or unknown	10	8	278 customer accounts were inadvertently canceled for unknown reason	464

Table 4.2: Summary of closed debugging tickets at *Anon1* over a 4-month period. Examples have been partially Anon1ymized and summarize the root causes listed in representative tickets. Debugging times are in minutes.

- **Resource underprovisioning** [KMB17, Sal17]: In such bugs (e.g., at Facebook [KMB17]), the containers or VMs running parts of a distributed system are allocated insufficient CPU, memory, disk, or network bandwidth.

- **Component failures** [FPK07, YLZ14a, DHH18, VM19, Dro19]: Failures are common at scale, and can result from a faulty physical machine, a bug in the machine’s hypervisor, or an unduly small amount of memory being allocated to a particular component.
- **Subsystem misconfigurations** [YLZ14a, VM19, LLM19c]: Errors in the internal configuration files for a given subsystem are common, especially given complex interoperation with other subsystems. Examples include incorrect hostname mappings that result in improper traffic routing and poorly configured values for timeouts or maximum connection limits [LLM19c].

2. Network faults.

- **Network congestion** [KMB17]: Within data centers [KMB17], queues build up at various network locations (e.g., virtual and physical switches) that connect subsystems, either due to temporarily increased application traffic (e.g., TCP incast [CGL09]) or cross traffic.
- **Incorrect network configuration** [KMB17, YLZ14a, VM19]: Network devices (e.g., firewalls, NATs, switches) between subsystems that communicate via RPCs may be incorrectly configured with forward/drop rules. This could cause unintended forwarding of packets to a destination or incorrect packet dropping.

3. Application logic faults.

- **Bugs within subsystems** [SBB10, QTS05, LPS08, ABI18, LLQ05, Don17, Sal17]: Bugs in application logic are prevalent in practice [NM19, AKS18], and can result in a wide range of system effects. For example, certain bugs arise from (accidentally) inverted branch conditions that trigger seemingly inconsistent behavior: an application may traverse an incorrect branch and display incorrect content or result in a program error. In contrast, certain code changes can trigger performance degradations, e.g., if unnecessary RPC calls are generated between microservices.

- **Incorrect data exchange formats and values [LLM19c]:** Particularly in microservice settings as in Azure services [LLM19c], bugs can arise if the RPC formats of the sender and receiver do not match. For instance, a change in the API exposed by one microservice could result in a bug if its callers are unaware of this change. Also included in this category are certificate or credential updates that have only been partially distributed (resulting in access control errors).

4.5 Distributed Systems Debugging Testbed

Developing and testing REVELIO requires access to a distributed systems environment with debugging data and system logs. Industrial systems satisfy these requirements internally [Jac21, Ora21] but, to our knowledge, no such environment exists for public use. We instead opt for an extensible in-house testbed (Figure 4.7) that incorporates state-of-the-art distributed apps, debugging tools, and fault injection. Our testbed is open-sourced and can be found at <https://github.com/debugging-assistant>.

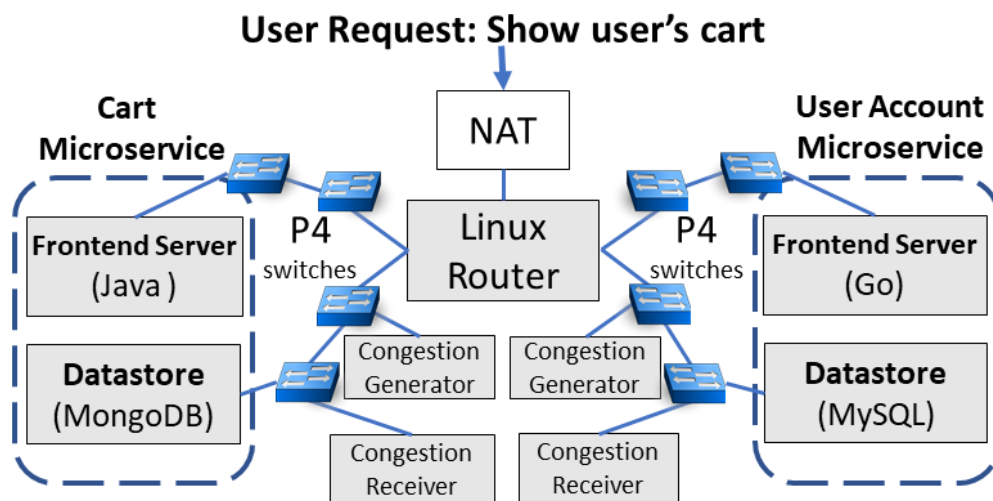


Figure 4.7: A slice (2/14 microservices) of our testbed for Sock Shop [Wea17]. Debugging tools and fault injection are omitted.

Our testbed embodies the takeaways from our study of *Anon1*'s production incidents, debugging workflows and literature survey.

4.5.1 Single-machine Emulation of Distributed Applications

Our testbed considers three open-source distributed web apps: Reddit [red22] (monolithic), Sock Shop [Wea17] (microservice-based), and Online Boutique [Goo19] (microservice-based). For each, we use the publicly available source code that was provided by the corresponding organization and is intended to capture the technologies and architectures they employ in production. We focus on distributed web applications because they are typically deployed in end user-facing scenarios. Consequently, we expect that the Mechanical Turk users in our data collection phase (§ 4.5.5) will be able to more naturally interact with these applications, leading to more realistic user reports. § 4.5.2 provides additional details for the composition of each application.

Our goal is to run each application in a distributed and controlled manner, in order to scale to large workloads and deployments, consider broad sets of realistic distributed debugging faults, and ultimately generate complete debugging datasets for REVELIO. In addition, we aim for our testbed to be extensible with respect to new applications and debugging tools. One approach would be to run each application service on VM instances in the public cloud. However, public cloud offerings typically hide inter-instance network components (e.g., switches) from users, precluding the use of in-network debugging tools (§ 4.7).

Instead, we opt for local emulation whereby we run each subsystem (or service) in a different container on the same machine, and specify the network infrastructure and connectivity between them. We use Containernet [PKR16], an extension of Mininet [LHM10] that can coordinate Docker [Doc19] containers, each running on a dedicated core; we assign a separate core for network operation (i.e., P4 switch simulation). Testbed throughput can be scaled up by using more physical machines through distributed emulation [Git21].

We note that a single, well-provisioned (4 cores in our setup) machine is sufficient for the distributed web applications and workloads that we consider in our evaluation, which do not stress the network significantly. However, our testbed can be scaled to support applications with higher throughput demands by making use of additional physical machines; added resources can be incorporated either with distributed emulation platforms [Git21] or through the use of faster hardware switches instead of Mininet’s software switches.

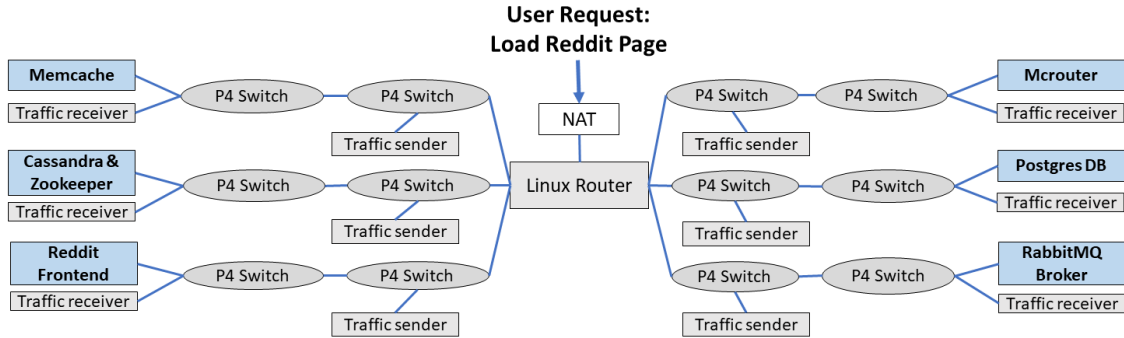


Figure 4.8: The topology of our distributed systems testbed for Reddit [red22]. Each P4 switch has a congestion traffic sender/receiver to emulate different network conditions, and the testbed incorporates four recent debugging tools and a fault injection service. We illustrate the Sock Shop [Wea17] topology in Figure 4.7, and note that Online Boutique [Goo19] follows the same architectural patterns.

For each application, we configure its subsystem/service containers into a star topology. This structure is highly amenable to the broad range of faults we aim to inject (§ 4.5.4). At the center is a router which is responsible for layer 3 faults (e.g., firewall configuration errors). Each subsystem is connected to this router via two P4 programmable switches [BDG14a]. We set routing rules to ensure that all subsystems are appropriately reachable. Finally, a NAT connects the central router to the host machine’s Internet-reachable interface.

Once this configuration is established, we alter the internal app configuration files to point to the appropriate containers for each subsystem. We use two switches to simplify network congestion-induced faults for a given subsystem: rather than sending fake traffic over

the ingress link to the subsystem, we run a dedicated client and server for traffic generation across the two switches and flood the link one hop away from the subsystem. Figure 4.8 shows the topology of our testbed for Reddit application.

In order to connect our virtualized environment to the outside world, we incorporate a NAT box which connects the central router to the physical host machine’s Internet-reachable interface. Finally, to ensure realistic operation, we alter each application’s default data population script to function in our setting. For example, in the context of Reddit, the data population script scrapes subreddits, popularity information, and comment chains from the real Reddit website, and generates fake user account information to mimic that seen in practice.

Note there are a bunch of options, each with tradeoffs: for instance, could run each app component on public cloud but this is tricky because we cannot control network between components. alternatively, we could set up an on-site cluster where each service runs on an individual machine (or set of machines); this is not portable (we want to release this so others can use our data but also generate their own data). The public Reddit codebase defaults to running all subsystems (e.g., application tier, Cassandra, PostgreSQL) on a single machine. However, in order to scale to larger workloads and deployments, and consider broader sets of distributed debugging issues and faults, we modified the code base to run in a distributed manner.

4.5.2 Overview of Applications

Reddit. Reddit [red22] is a popular discussion website whose three-tier backend architecture is representative of many distributed applications that utilize the monolithic architectural paradigm. In the front-end tier, HAProxy [hap19] load balances traffic across web servers. The application tier, implemented using the Pylons framework for Python [pyl19], embeds the core application program logic and accesses data objects from the storage tier. The storage

tier consists of three data stores: PostgreSQL [pos19] is mainly used as a key/value store for objects such as accounts and comments; Cassandra [cas16] is used as a key/value store for precomputed objects such as comment trees; and Memcache [Dor15] is used for caching throughout the system. Reddit also uses the RabbitMQ message broker [rab19] to manage asynchronous writes to the storage layer.

Sock Shop. Developed by Weaveworks, Sock Shop [Wea17] is an e-commerce application that employs a microservice-based backend architecture. Sock Shop incorporates 14 different microservices, including a user-facing Node.js frontend microservice, a shopping cart management microservice, a catalog microservice, and so on. Each microservice includes an application server whose logic is implemented in one of a variety of programming languages (e.g., Java, Go), and select microservices additionally operate an individually-managed datastore. For instance, separate MongoDB database instances [Inc19] are used for cart information, processed order transactions, and user profiles, while catalog information is stored in a MySQL database [Mys21]. As with Reddit, RabbitMQ manages inter-microservice communication.

Online Boutique. Online Boutique [Goo19] is another microservice-based e-commerce platform from Google that includes 10 distinct microservices that are implemented in Python, Go, C#, Java, and JavaScript. Microservices include a frontend HTTP server (implemented in Go), a payment microservice, a cart microservice, and an ad microservice. Each microservice operates its own datastore, e.g., the cart microservice stores a user's to-be-purchased items in Redis [Red21]. Microservices communicate using the gRPC framework [Env21].

4.5.3 Overview of Debugging Tools

Marple. Marple [NSN17] is a performance query language for network monitoring that uses SQL-like constructs (e.g., groupby, filter). To operate, Marple assigns each network switch and packet a unique ID, and supports queries that track 1) per-packet and per-

switch queuing delays, and 2) user-defined aggregation functions across packets. In our implementation, switches log queue depths that each arriving packet encounters, and the packet’s 5-tuple (src/dest ip addresses, src/dest ports, and protocol). This is sufficient to track queuing information and high-level statistics such as packet counts. We write queries in Marple to capture and track these values, and then use Marple’s compiler to generate P4 programs [BDG14a] that can run directly on our emulated switches. Switches stream query results to a data collection server running on the same host machine for further analysis.

tcpdump. tcpdump [Tcp22] is an end-host network stack inspector which analyzes all incoming and outgoing packets across all of the host’s network interfaces. tcpdump’s command line interface supports querying in the form of packet content filtering (e.g., by hostname, packet type, checksum, etc.), which can be applied at runtime or offline. In our implementation, tcpdump is configured to collect all network packet information in a pcap file, and filters are applied offline.

Jaeger. Uber’s Jaeger framework [Jae21] is an end-to-end distributed systems tracing system which, like its predecessors Dapper [SBB10] and Zipkin [Zip22], implements distributed tracing according to the OpenTracing specification [Ope22b]. With Jaeger, developers embed tracepoints directly into their system source code (or RPC monitoring proxies [Grp21]) and specify custom state (e.g., variable values) to log at each one. By aggregating tracepoint and timing information, Jaeger provides distributed context propagation so developers can understand how data values and control state flows across time and subsystems. We modified each application’s source code (application tier for Reddit, and each microservice frontend for Sock Shop and Online Boutique) to include tracepoints for each function accessed during HTTP response generation. As per the examples provided by OpenTracing [Ope22b], at each tracepoint, we log the accessed variables, function execution duration, and any thrown exceptions. During execution, all tracepoint information is sent to a Jaeger aggregation server running on the same host machine for subsequent querying.

cAdvisor. Google’s cAdvisor framework [Goo21] profiles the resource utilization of individual containers. To do so, cAdvisor runs in a dedicated container, which coordinates with a Docker daemon running on the same machine to get a listing of all active containers to profile (and the process ids that each owns). With this information, caAdvisor uses the Linux cgroups kernel feature to extract resource utilization information for each container. We use cAdvisor’s default configuration, in which the following values are reported every 1 second: instantaneous CPU usage, memory usage, and disk read/write throughput, and cumulative number of page faults. Resource usage information collected by cAdvisor is dynamically sent to a custom logging server running on the same host machine for subsequent querying.

We integrate the above debugging tools into our testbed the following way:

1. **Marple** [NSN17] is a performance query language for network monitoring that uses SQL-like constructs (e.g., groupby, filter) to support queries that track 1) per-packet and per-switch queuing delays, and 2) user-defined aggregation functions across packets.
2. **tcpdump** [Tcp22] is an end-host network stack inspector which analyzes all packets flowing through the host’s network interfaces, and supports querying via packet content filtering (e.g., by hostname).
3. **Uber’s Jaeger** [Jae21] is a distributed systems tracer which follows the OpenTracing specification [Ope22b]. Developers embed tracepoints into system source code to log custom state, and then aggregate tracepoint and timing information to understand data/control flows across time and subsystems.
4. **Google’s cAdvisor** [Goo21] profiles the resource utilization of individual containers, logging the following per second: instantaneous CPU usage, memory usage, disk throughput, and total page faults.

Note that these tools represent only part of the state-of-the-art for network and distributed systems monitoring; our setup is amenable to others [KMB17, FPK07, App21].

4.5.4 Fault Injection Service

To create data from realistic debugging scenarios, we created an automatic fault injection service. We note that our goal is not necessarily to match the system scale at which production faults were reported, but instead to evoke the user reports, system log patterns, and queries that correspond to the reported fault categories. Further, while our survey and study at *Anon1* revealed a relatively small set of common fault categories, others can arise; our fault injection service can be easily extended to incorporate new bugs. Our service is guided by our literature survey of production faults and our findings at *Anon1* (§4.4.2). Specifically, we incorporate faults that cover all of the observed categories, and match the ratios across categories with the data from *Anon1* (Table 4.2). These categories cover the observable performance (i.e., increased response times) and functionality (i.e., missing or inconsistent content, crashes) issues for the apps we consider. Table 4.3 lists the faults we inject.

Our fault injector operates differently per fault class. For network or system configuration faults, we use Mininet and Docker commands to bring down a subsystem, start a congestion generator, change a service’s provisioned resource values, or inject a firewall/routing rule at the router. Application logic faults require modified source code. In each container pertaining to an application’s service logic, we include a script that takes in a fault instruction and replaces the appropriate source code with a version embedding the fault, and restarts the application.

4.5.5 Dataset Collection using AWS MTurk

To extract system logs, user reports, and debugging queries from our testbed, we conducted a large-scale data collection experiment on Amazon Mechanical Turk. For each application, we set up an EC2 instance per fault that we consider (Table 4.4). Each instance runs the entire testbed for that application, with the associated fault injected into it. Application data is collected using scripts included in each application repository. All instances for an app were

Fault Type	Number of Faults	Example
Resource underprovisioning	15	Reducing the CPU quota for the docker container running PostgreSQL
Component failures	15	Take down container for a given microservice
Subsystem misconfigurations	12	Incorrectly configure hostname of a database
Network congestion	13	Generate significant network cross-traffic between hosts for different microservice
Network-level misconfigurations	16	Incorrect firewall rules at routers to drop or forward packets on an incorrect interface
Subsystem/Source-code bugs	16	Negated if condition resulting in different execution path
Incorrect data exchange	15	Alter function signature within a microservice, triggering argument violations

Table 4.3: Overview of faults injected into our distributed systems testbed. Numbers listed are for Sock Shop[Wea17].

populated with the same content, generated using an app-provided script, e.g., content for Reddit was scraped from the real Reddit website, and includes multiple subreddits, each with roughly 10 posts and user profiles.

Metric	Reddit	Sock Shop	Online Boutique
# of Unique Faults	76	102	80
# of Unique Queries	118	320	269
Query Vocabulary Size	60	136	122
Report Vocabulary Size	1040	1327	1258

Table 4.4: Summary of debugging queries and user reports collected through AWS MTurk experiment.

Our experiment supported only “master” Turk users, and each was only allowed to participate once per fault+application pair. Each user was assigned to a specific instance/fault at random, and was presented with a UI that pointed to the corresponding instance’s frontend web server. Users were asked to perform multiple tasks within each app, including loading the homepage, clicking on item pages or user profiles, adding comments, and adding items to their carts. Prior to the experiment, users were shown example page loads (to ensure familiarity).

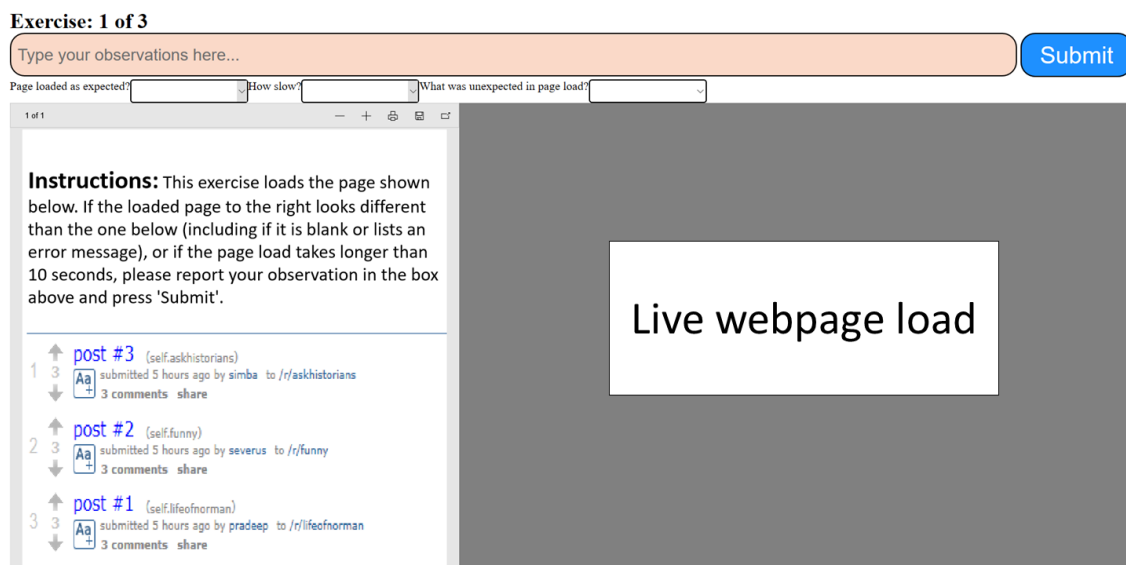


Figure 4.9: User Interface of our MTurk experiment presented to users for data collection.

Figure 4.9 shows the user interface of our MTurk experiment that users interact with. The left half of the UI has instructions for users on what to do in the experiment and an image of what a successful webpage load of an application looks like. The right half of the UI contains the UI for the applications hosted our testbed (e.g., a reddit user page). By using the left half of the UI as reference, users can decide if the actual application load looks any different and they report their observations in the textbox provided on top of the UI and also select some drop-down choices that indicate objective feedback in addition to the natural language observations.

Marple	Jaeger	cAdvisor
Packet count	# of accessed variables	CPU utilization
Queue depth	Duration of execution	Memory utilization
N/A	# of exceptions thrown	Disk throughput

Table 4.5: Metrics in system logs. Marple, Jaeger, and cAdvisor metrics are recorded per-switch, per-function, and per-container; tcpdump is omitted for space.

During each experiment, the standard system logs for each testbed tool were collected on the instance (complete list shown in Table 4.5). We condense and featurize the time-series data for each metric using standard statistics, e.g., min, max, avg. User reports were paired with the associated system logs. We allowed up to 5 concurrent users per instance, and system logs reflect the interactions of all concurrent users. To complete our dataset, for each fault, we generate a debugging query with the appropriate tool that sufficiently highlights the root cause. This query is intended to represent the result of a past (successful) debugging experience. Table 4.4 summarizes our dataset, and Table 4.6 lists examples of user reports.

User report text
there is nothing on the page, it is empty, nothing to click on
'you broke reddit' with the cartoon showed up
Page took forever to load. Sat at the gray screen for almost a minute, entirely too long..
I clicked to expand for comments, and page went away and defaulted to a grey screen.. No Page
'Funny 500 Page Message 8' message below that. Blank otherwise. . Page does not include any usernames

Table 4.6: Examples of text in user reports collected from Mechanical Turk participants.

4.6 Evaluation of REVELIO

Methodology. We divided the dataset collected from our MTurk experiment for each application into: 53% for training revelio’s ML models, 13% for validation to pick the best performing ML models, and 34% for testing the chosen models. We further divided our testing data into two test sets, *test_generalize* and *test_repeat*. *test_generalize* evaluates REVELIO’s ability to generalize to new locations for previously seen fault types, and includes only data for faults that have matching query templates in the training data, but different parameters. *test_repeat* evaluates REVELIO’s ability to suggest relevant queries for repeat faults, and includes only data for faults that have matching query templates *and* parameters in the training data. All results test the best observed model from the validation set on the test sets. We evaluate REVELIO primarily using two metrics: 1) *rank* of the correct query (i.e., the query which most directly highlights the root cause) among the ordered list of the model’s predicted queries, and 2) *top-k accuracy*, defined as the presence of the correct query in the top-k predictions.

4.6.1 REVELIO’s Performance on Repeat Faults

For each fault in *test_repeat* set, we measured the rank of the correct query in REVELIO’s predictions. As shown in Figure 4.10, for 80% of the Reddit test samples, REVELIO assigns a rank of 1 to the correct query. Further, for 96% of the Reddit test cases, the correct query is in the top 3 predicted queries. Performance is similar for Sock Shop, with the correct query being in the top 3 100% of the time.

4.6.2 REVELIO’s Performance on New Faults

Figure 4.11 shows results for the more challenging scenario of new fault locations for repeat fault types (i.e., *test_generalize*). As shown, REVELIO consistently predicts the correct query:

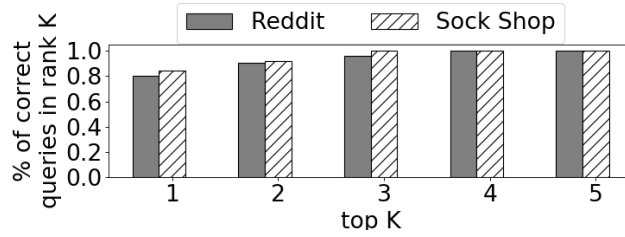


Figure 4.10: Cumulative distribution (per app) of the rank of the correct query over our test set of *repeat* faults.

REVELIO’s model assigns a rank of 1 to the correct query 60% and 85% of the time for Reddit and Sock Shop. For both apps, the correct query was *always* in the top-5 predictions.

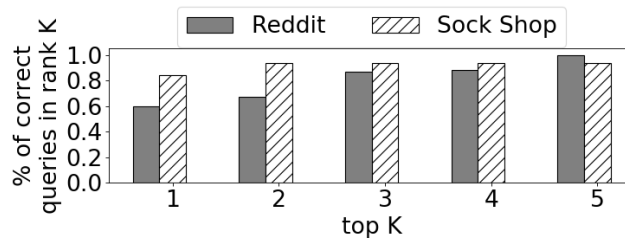


Figure 4.11: Cumulative distribution (per app) of the rank of the correct query over our test set of *previously unseen* faults.

4.6.3 Understanding REVELIO’s Performance

Importance of user reports.. By default, REVELIO’s model accepts both natural language user reports and quantitative system logs. To understand the importance of considering user reports in query generation, we evaluated a version of REVELIO that excludes user reports from its input set; note that system logs cannot be excluded as they are required for parameter prediction. As shown in Table 4.7, REVELIO significantly benefits from having access to both inputs. For example, on `test_generalize`, the average rank of the correct query is 1.97 and 2.29 with and without user reports.

Scenario	test_repeat	test_generalize
User report+system logs	1.33 (100%)	1.97 (100%)
Only user report	4.70 (<1%)	75 (<1%)
Only system logs	1.86 (100%)	2.29 (90.2%)

Table 4.7: Impact of different input sources on REVELIO’s performance. Results list avg rank (% in top-5) and are for Reddit.

Model	test_repeat	test_generalize
REVELIO	1.33 (100%)	1.97 (100%)
REVELIO_monolithic	17.5 (15.1%)	22.4 (18.5%)
REVELIO_no_rank_order	1.29 (100%)	N/A
REVELIO_classifier	2.41 (88.7%)	2.69 (86.9%)

Table 4.8: Comparison with simpler ML approaches. Results list avg rank (% in top-5) for Reddit.

Simpler ML approaches.. To understand the importance of REVELIO’s design (§ 4.3), we compared it with the following variants: 1) *Revelio_monolithic* uses a single model to output a fully-formed query, 2) *Revelio_no_rank_order* eliminates the rank ordering of features in REVELIO’s models, and 3) *REVELIO_classifier* uses a multi-label classifier to select query templates rather than employing a GCN to construct a vector representation of each template’s AST. Table 4.8 lists our results which highlight three points. First, REVELIO outperforms *Revelio_monolithic* on both test sets, highlighting the importance of factorization in terms of simplifying (both computationally and statistically) query prediction, particularly for generalization. Second, by rank ordering feature values, REVELIO achieves an average rank of 1.97 for test_generalize; in contrast, *Revelio_no_rank_order* is fundamentally unable to predict templates and parameters (and thus, queries) for repeat fault types in new locations. Third, REVELIO’s improved performance over *REVELIO_classifier* illustrates the importance of extracting semantic information about query structure (which a classifier cannot).

Removed Feature	test_repeat	test_generalize
Packet count	2.14 (100%)	7.93 (80.4%)
Queueing delay	2.27 (96.1%)	2.51 (92.4%)
Variable count	1.67 (96.1%)	3.54 (71.7%)
Duration of execution	7.29 (88.2%)	4.11 (89.1%)
CPU utilization	1.65 (96.1%)	4.37 (83.7%)
Memory utilization	1.75 (100%)	2.50 (88.0%)

Table 4.9: REVELIO’s performance when metrics from system logs are selectively removed. Removed Marple, Jaeger, and cAdvisor features are shown in blue, red, and grey, respectively. Results list avg rank (% in top-5) and are for Reddit.

System log analysis. To understand the relative importance of each metric in the system logs, we evaluated a variety of REVELIO models that were trained with each log feature removed, in turn (Table 4.9). As shown, removing the per-switch *packet counts* from the network logs led to the largest accuracy degradation, with a drop in average rank from 1.97 to 7.93 (for test_generalize). Importantly, removing each considered feature led to marginal degradations in REVELIO’s performance, highlighting their utility.

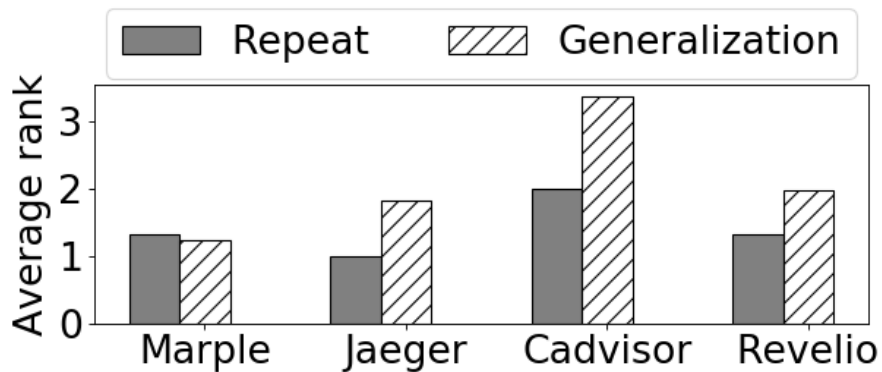


Figure 4.12: Comparing the average rank for single-tool and multi-tool versions of REVELIO. Results are for Reddit.

Multi-tool vs. single-tool models. We performed another ablation study where we compare REVELIO when training and testing on logs from each tool together (multi-tool model), and in isolation (single-tool model). For each isolated tool, we prune the training, validation, `test_repeat`, and `test_generalize` sets to include only faults pertaining to that tool. Figure 4.12 shows that the per-tool models achieve better average ranks than the combined (default) model. The reason is that focusing on one tool allows REVELIO to predict templates and parameters from a smaller space. However, REVELIO pays only a small cost for operating across debugging tools: the average rank in the combined model is only 33% higher than the best per-tool model. This is key to REVELIO’s ability to alleviate the burden of determining which tool to use for a particular scenario.

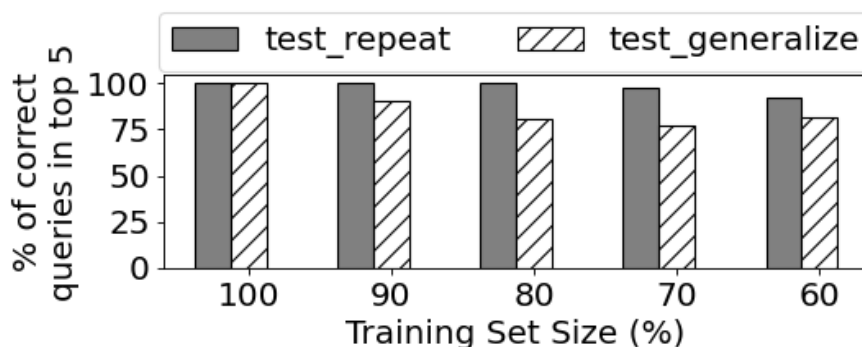


Figure 4.13: Top-5 query accuracy when training REVELIO on random subsets of the data. Results are for Reddit.

Data and training costs. Training REVELIO (using per-sample loss propagation) took 55 minutes in our experiments, roughly evenly split across the template and parameter prediction models. However, due to computational or security restrictions, organizations may be unable to train on all of the available debugging data, e.g., *Anon1* archives years of the debugging data that REVELIO requires. Figure 4.13 shows that REVELIO’s accuracy remains relatively stable as the training dataset shrinks: with only 60% of training data, top-5 accuracy drops to 92% and 82% for the `test_repeat` and (harder) `test_generalize` test sets, respectively.

4.6.4 Developer Study

To evaluate REVELIO’s ability to accelerate end-to-end root cause diagnosis, we used our testbed (§ 4.5) to conduct a developer study. Developers were presented with the testbed’s tools and logs, both with and without REVELIO, and were tasked with diagnosing the root cause of multiple high-level user reports. In summary, developers with access to REVELIO were able to correctly identify 90% of the root causes (compared to 60% without REVELIO), and did so 72% faster.

Setup and methodology. Our study involved 20 PhD students and postdoctoral researchers in systems and networking. All participants brought their own laptops, but debugging tasks were performed inside a provided VM for uniformity. Prior to the study, the authors delivered a 5-hour tutorial explaining the testbed and Sock Shop UI/code base; the study only involved Sock Shop to ease the developers’ ability to become intimately familiar with the application to debug. For each tool (Marple, Jaeger, cAdvisor, tcpdump, REVELIO), we described its logs, query language, and interface. Developers were given 1 hour to experiment with the testbed and resolve any questions.

During the study, developers were presented with a series of six debugging scenarios: 2 in-network faults for routing errors and congestion (targeting Marple), 2 system configuration faults for resource underprovisioning and component failures (targeting cAdvisor), and 2 application logic faults for branch condition and RPC errors (targeting Jaeger); we exclude end-host network faults due to time constraints. For each fault type, developers were randomly assigned to debug one fault using only the testbed’s tools, and one also using REVELIO. Ordering of the faults and tool assignments was randomized across participants to ensure a fair comparison.

For each fault, developers were presented with 1) a user report, 2) system logs for all testbed tools collected during the faulty run, and 3) the faulty testbed code. Developers were given 30 mins to diagnose each fault and provide a short qualitative description of the

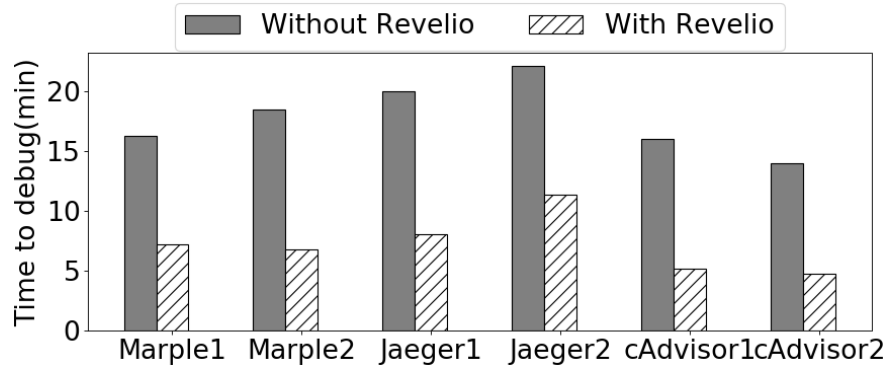


Figure 4.14: Summary of time saved in debugging each fault in our developer study. Bars represent average time spent across all developers who correctly identified the root cause.

root cause. For example, a routing configuration error that disconnected Cassandra could be successfully reported as “Cassandra could not receive any network packets, leading to missing page content.” When a developer believed she had found the root cause, she informed the paper authors who verified its correctness. If incorrect, the developer was told to keep debugging until a correct diagnosis was generated, or 30 mins elapsed. Developers were unrestricted in their debugging methodologies, e.g., they were not required to use queries, though most did. Without REVELIO, developers had to generate any query they wished to issue on their own; with REVELIO, developers could generate queries or use the 5 suggested by REVELIO.

REVELIO’s impact on root cause diagnosis. The results of our developer study were promising and suggest that REVELIO can be an effective addition to state-of-the-art debugging frameworks in terms of accelerating root cause diagnosis. Across all of the faults, REVELIO increased the fraction of developers who could correctly diagnose the faults within the given time frame from 60% to 90%. Further, as shown in Figure 4.14, REVELIO sped up the average root cause diagnosis time by 72% (~14 minutes) in cases where the developers were able to report the correct root cause.

After the study, we asked each developer qualitative questions about their experience with REVELIO. The most commonly reported benefit of REVELIO was in shrinking the set of tools and queries that a developer had to consider. The primary gripe was with respect to REVELIO’s UI, which is admittedly unpolished. Most importantly, the response to "Would you prefer to use existing systems and networking debugging tools with REVELIO?", was “yes” for all 20 participants.

4.7 Related Work

4.7.1 Debugging Tools for Distributed Systems

There exist dozens of powerful logging and querying tools for distributed systems [MRF15, SPB16, SBB10, MF18, FPK07, KMB17], networks [HHJ14, NAR16, NSN17, MYG16, TAL15, TAL16, Tcp22], and end-host stacks [GNU22, ANC00, NM19, FB88, KM08, LGN08, VNN13, GBF99, KL88]. However, two limitations exist. First, these tools are not coordinated and lack context about system-wide debugging. Thus, the cognitive burden of deciding which tools to use, when, and how falls on developers. REVELIO interoperates with these tools and alleviates this burden by automatically predicting helpful debugging queries. Second, these tools ignore natural language inputs, despite them containing debugging insights [PJN13, GMK16].

4.7.2 Leveraging Natural Language Data Sources

Program debugging: NetSieve [PJN13] uses NLP to parse network tickets by generating a list of keywords and using a domain-specific ontology model to extract ticket summaries from those keywords; summaries highlight potential problems and fixes. While NetSieve automates parsing, much manual effort is still required in (1) offline construction of an ontology model, and (2) determining what constitutes a keyword. In contrast, REVELIO’s models learn automatically from data, with minimal manual effort, and generate queries for root cause

diagnosis rather than potential fixes from a restricted set of actions. Net2Text [BDV18] translates English queries into SQL queries, issues those queries, summarizes the results, and translates them back into natural language for easy interpretation. REVELIO, instead, ingests high-level user issues and system logs; the unstructured and abstract nature of this input makes REVELIO’s problem harder than Net2Text’s.

Program analysis and synthesis: NLP techniques have been utilized in multiple aspects of software development [Ern17]. Examples include detecting operations with incompatible variable types [HCE15] and converting natural language comments into assertions [GGE16]. More recently, NLP has also been used in code generation by converting developer-specified requirements in natural language to structured output in the forms of regular expressions [LND16], Bash programs [LWZ18], API sequences [GZZ16], and queries in DSLs [DGH16]. Though these projects show the potential to extract meaning from natural language debugging data, they are limited to ingesting a single stream of data from a single subsystem. In contrast, REVELIO combines and extracts meaning from varied input forms to construct structured queries.

CHAPTER 5

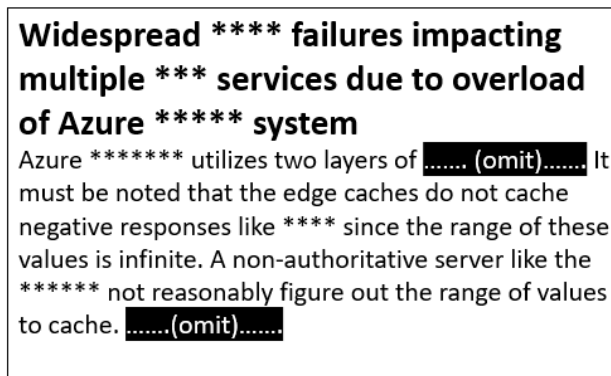
Automated Root-Cause Labelling

This chapter provides an overview of a joint work [DBJ23] with Chetan Bansal, Gopinath Jayagopal, Richie Costleigh, Suman Nath and Xuchao Zhang. As mentioned in §1.3, existing practice is to manually label postmortems with a single root-cause based on an ad hoc taxonomy of root-cause tags. However, the manual process is error-prone and a single root-cause is inadequate to capture all contributing factors behind an incident, and ad hoc taxonomies fail to capture the diverse categories of root-causes, wasting rich treasure troves of debugging insights. We investigate the problem of automatically labelling *all* the contributing factors behind incident.

Specifically, we address this problem with a three-pronged approach. First, we conduct a large *scale* multi-year analysis of 2000+ incidents from 450+ services in Microsoft Azure to understand *all* contributing factors which caused the incidents. Second, based on the empirical study, we propose a novel *hierarchical* and comprehensive taxonomy of potential contributing factors which cause the production incidents. Lastly, we develop a *centralized* automated tool that can assist humans in the labelling process to *abstract* the details of root-causes into tags that are consumable. To the best of our knowledge, this is the largest and most comprehensive study of production incident postmortem reports yet.

5.1 Incident Postmortem Reports

One effective approach to discover and resolve existing reliability risks [GMK16] is to aggregate root-causes of past incidents based on their postmortem reports to uncover common problem areas, trends, patterns, and risks (e.g., identify the most common root-causes in the last year). Similarly, bucketing past incidents based on their root-causes can enable on-call engineers (OCEs) to quickly retrieve and learn common mitigation strategies for a given root-cause category. However, postmortems are commonly written in natural language with little structure. This makes the tasks of aggregating or bucketing past reports challenging, especially at a large-scale.



Widespread ** failures impacting multiple *** services due to overload of Azure ***** system**
Azure ***** utilizes two layers of (omit)..... It must be noted that the edge caches do not cache negative responses like **** since the range of these values is infinite. A non-authoritative server like the ***** not reasonably figure out the range of values to cache.(omit).....

Figure 5.1: Redacted example of an incident postmortem report from a service in Microsoft Azure.

Figure 5.1 shows an example of a redacted postmortem report of a production incident in one of many Microsoft Azure services. Postmortem reports are treasure troves of rich debugging insights. They have detailed descriptions of incidents like symptoms, what root causes were found, steps taken to mitigate the incident, etc. In Microsoft Azure, specifically, engineers are required to prepare and answer at least 5 questions that start with a ‘why’ to better understand the incident and its debugging workflow. We will use Post Incident Reports (PIRs) interchangeably with incident postmortem reports from here on.

5.2 Overview - Root Cause Labelling

Root cause labelling of incident postmortem reports enables quick and accurate aggregation of and retrieval from a large collection of postmortems solely based on their root-cause tags. Figure 5.2 shows postmortems written by engineers from across a cloud’s services, which can be analyzed by owners/managers with the help of root cause labelling. Such analysis can help identify top root causes causing high impact incidents, the trends of root causes over time to identify potential vulnerabilities and to inform strategic investments into improving the reliability of a cloud.

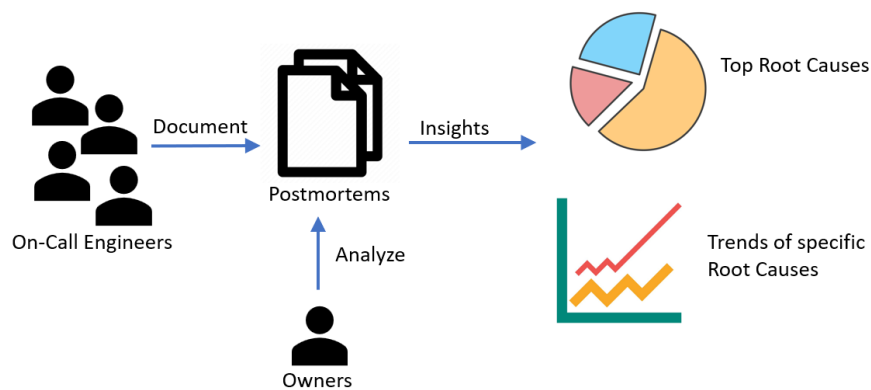


Figure 5.2: Labelling incident postmortem reports documented by several engineers in a cloud enables owners to extract insights with meaningful analyses.

Figure 5.3 shows *Coarsening* through mapping contributing factors of incidents to root cause labels from ARTS taxonomy assigned by AUTOARTS. Element c in the concrete poset indicates all the incidents in the cloud caused by a authentication-related contributing factor (very coarse). c_1 indicates all the incidents in the cloud caused by a certificate related authentication issue in the cloud (coarse). c_{12} indicates all the incidents in the cloud caused by an expired certificate (very fine).

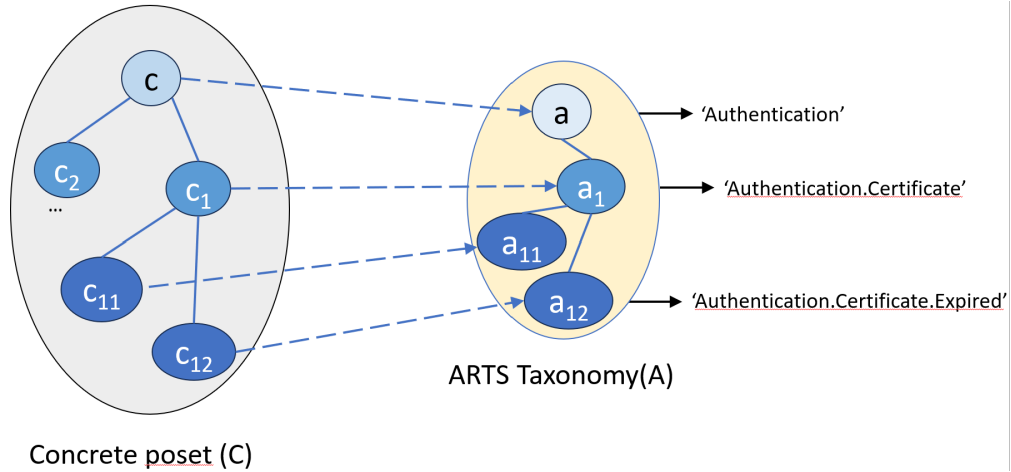


Figure 5.3: AUTOARTS as an implementation of *Coarsening* through root cause labels from ARTS taxonomy.

5.2.1 Challenges

We analyze a sample of $\approx 1.7M$ root cause analyses in Microsoft Azure, across all its services, to understand root cause labelling of PIRs. We found the following challenges:

Existing taxonomies, although designed by domain experts, are not comprehensive enough. This is due to the lack of a comprehensive study of root causes, many potential root cause categories are missed or not anticipated when a taxonomy is designed. As an implication of this, a PIR author may not find a suitable predefined root cause tag to describe the current incident. In our sample of root cause analyses, $\approx 20\%$ incidents are labelled as ‘Other’ and $\approx 58\%$ are labelled with categories containing ‘Other’ (e.g., ‘Network - Other’), implying that their root causes are not covered or only partially covered by the existing taxonomies. Such ‘Other’ tags are not useful in the aforementioned root cause aggregation and retrieval tasks.

Existing manual root cause labelling process is expensive and error-prone. Root cause label of an incident is often determined based on its PIR and incident report. These

documents are usually *long* (4542 words per incident in our sample) and *complex* (on average, ≈ 9 engineers engaged in discussion exchanging 20 comments). Thoroughly understanding these long documents to identify all contributing factors behind an incident, and then selecting from predefined root cause labels that represent the factors, is a nontrivial task.

Even when the root cause is understood, PIR authors may make mistakes in choosing the correct tag. This can happen due to multiple factors. Existing taxonomies at Microsoft are flat long lists, making it difficult to navigate through them and to pick the right tags. Moreover, many individuals are involved in the root cause analysis efforts. For example, we observe 34K distinct individuals involved in a sample of 600K PIRs in Microsoft. This large number of individuals are likely to have varying degrees of expertise and different interpretations of root cause tags. This is further exacerbated by ambiguous or confusing tags in the taxonomy (e.g., ‘Network’ and ‘Datacenter - Network’). All these factors can contribute to inconsistent and/or inaccurate labels. We manually examined a small sample of 1241 PIRs and found that 29% of the assigned tags are incorrect.

Compact, comprehensive, granular taxonomy design. Postmortems need to be labelled with a well-defined taxonomy of root cause tags; otherwise, the same root-cause may be tagged differently in different postmortems authored by different teams, hindering aggregation analysis across different teams. A well-designed taxonomy for a large-scale cloud system such as Microsoft Azure should achieve a sweet spot between two competing objectives: it should be *comprehensive* enough to cover the myriad of potential root-causes, yet *compact* enough for the OCEs to navigate and use easily. Moreover, it should be *fine-grained enough* to be useful for surfacing actionable insights across many services.

Designing such a taxonomy is non-trivial. Several recent works analyzed production incidents and proposed taxonomies to capture their root-causes. However, most of these efforts focus on specific root-cause categories such as software bugs[LLL16, GDQ18, CDJ19, ZYJ21a, CLP14, LLM19d] and hence their taxonomies are not comprehensive enough (e.g., to capture hardware failures). Several other efforts consider multiple types of root-causes,

but instead of a large-scale cloud system, they consider specific services or systems such as big-data systems[YLZ14b], business data processing platform[DKI14] and a specific cloud service[GSB22, GMK16]. Moreover, existing taxonomies are not fine-grained enough to represent all root-causes we observe in Azure incidents.

Labelling postmortems at-scale.. The current practice is to do this manually—an individual (OCE) carefully studies *lengthy* incident and post-incident reports, identifies the root-cause, and selects a representative tag from a taxonomy that is usually a long flat list of tags. Like all manual efforts, the process is error-prone. Moreover, tags can be inconsistent across incidents—at Microsoft, root-causing is conducted by tens of thousands of OCEs with varying degrees of expertise and different interpretations of root-cause tags.

5.2.2 Solutions

First, we manually analyze 2000+ high impact production incidents in 468 services in Microsoft Azure. Unlike previous empirical analysis of production incidents[GDQ18, CDJ19, ZYJ21a, CLP14, DNS22, LLM19d, DKI14, GMK16, YLZ14b], our analysis aims to identify not only a single root-cause, but *all* factors contributing to the incidents. The analysis took more than four person-years and it has so far identified 346 distinct root-cause categories spanning all aspects of a production service including hardware and software, infrastructure and application, software code and configuration, and so on. To the best of our knowledge, this is the most comprehensive empirical analysis of production incidents in cloud systems, in terms of the scale of incidents and affected services, the depth of analysis, and the diversity of root-causes.

Second, we propose a comprehensive taxonomy called Azure Reliability Tagging System (ARTS), that organizes the root-cause categories identified in the above analysis. For ease-of-use, the taxonomy is hierarchically organized and each leaf node represents a *root-cause tag* describing a factor contributing to an incident. We believe that the root-causes will apply to

other cloud services as well and hence we make the taxonomy available for other researchers and practitioners at <https://AutoARTS-rca-taxonomy.github.io/>.

Finally, to avoid manual errors and inconsistencies while assigning a tag to a postmortem, we develop a tool called AUTOARTS. At the core of AUTOARTS are ML-based algorithms that can assist a human in the labelling process with two important tasks (Figure 5.4). First, it uses a multi-label classification technique to automatically analyze an incident’s postmortem (written in natural language) and to identify multiple contributing factors and their representative tags from our proposed taxonomy. Second, it can produce a short text snippet (from the postmortem) that captures important context explaining the factors. The snippet enables a human to easily review the selected tags without reading lengthy incident reports or postmortems. We present how we adapt existing ML techniques for this purpose.

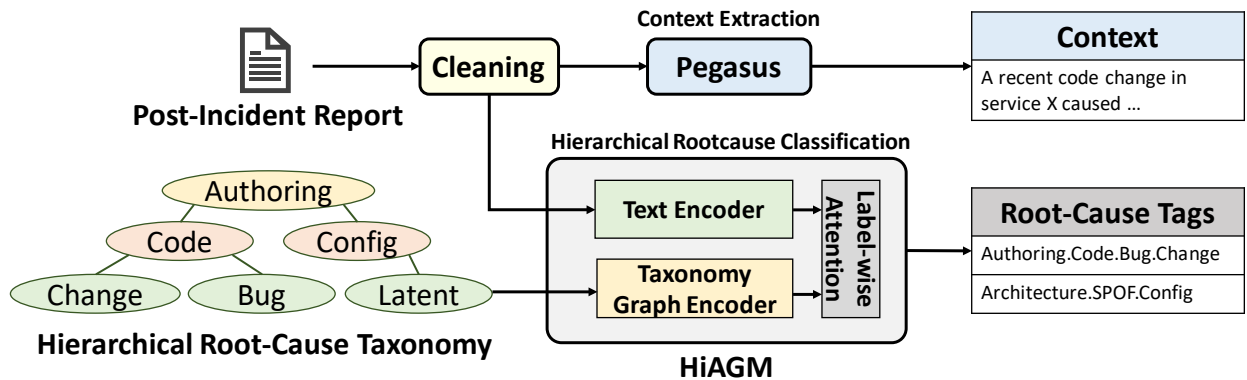


Figure 5.4: Overview of our Context Extraction and Hierarchical Root-cause Classification using Post-Incident reports.

5.3 Analysis of Production Incidents at Microsoft Azure

There exists several empirical studies of production incidents in large-scale cloud systems [GDQ18, CDJ19, ZYJ21b, DNS22, CLP14, LLM19d, DKI14, GMK16, YLZ14b]. We have two goals that differentiate our study from them. First, we do not restrict our analysis to

a limited set of root cause categories (e.g., software bugs [LLL16, GDQ18, CDJ19, ZYJ21b, CLP14, LLM19d]) or specific services/platforms (e.g., big-data systems [YLZ14b]). Second, for each incident, we try to identify not a single root cause, but *all* factors contributing to the incidents. These two goals enable us to identify a wide-range of contributing factors behind incidents happening in large number of services/platforms.

5.3.1 Manual Analysis of High-Impact Incidents

We analyze 2051 high-impact incidents in 468 Microsoft Azure services. We carefully analyze each incident by carefully reading and understanding its incident report and PIR, the discussion comments, and even the work items (e.g., bug fix, system upgrade) that are created due to the incident. When something is not clear, we reach out to the incident owners to clarify. As a part of the analysis, we not only identify the *contributing factors* causing the incident but also extract text snippets or *context* from the incident and PIR which helps explain and justify the identified root cause tag for future reference and validation. Every week, we peer review a randomly selected subset of incidents to help us refine our collective understanding of tag usage, promote learning and improve accuracy.

If we identify a new category of root causes, which is not covered by existing tags, we then propose new tags which are internally reviewed before getting introduced to the taxonomy. For any tag in the taxonomy, we also provide it's description in natural language for future reference. This data lives in an internal database which can be easily joined with incident databases and visualization reports are created for easy data analysis based on various pivots such as contributing factors, services, incident impact, etc. We also meet with the engineering teams on a weekly basis, and review our data both for accuracy and to share insights that result in reliability improvements.

The above process needs significant manual effort. Since 2020, we have analyzed 2051 incidents in 468 Azure services with a team of 2-4 members.

Category	Description	Frequency	TTM (Hrs)
Detection	Issues related to detecting problems before they affect production	61%	50
Authoring	Issues in authoring artifacts like code, config, troubleshooting guides, etc.	50%	58
Dependency	Issues in a dependency the service has, most typically another service but can also be some things where a boundary between teams is present	37%	16
Architecture	Issues in how the service is architected and likely where any work to fix would require changes to the architecture of the service	20%	33
Deployment	Issues related to deployment of code or config	20%	27
Process	Any issue caused by human errors, a flawed process or the lack of a process	18%	123
Load	Any issue caused by the service not being able to handle changes in load	14%	13
Auth	An authentication or authorization related issue	7%	21
Performance	An issue that caused excess latency	6%	16
Datacenter	Events (hardware, installations, power interruption, etc.) in the datacenter	4%	70

Table 5.1: High-level root cause categories from ARTS taxonomy with their descriptions, frequency of occurrence in our analysis and mean Time-To-Mitigate (TTM) for incidents caused by their sub-categories.

5.3.2 Findings from Empirical Analysis of Incidents

Finding 1. *Incidents are often caused by multiple contributing factors working together instead of an isolated root cause.*

This is contrary to prior work [GSB22, LLM19d, GHL14, GHS16] that focus on identifying a single root cause per incident. Consider, for example, a real incident where a service became unavailable after a single customer continuously pushed a load that was 60x greater than what the service was scaled to handle. The original PIR author chose the root cause label “*Service*

– *Load Threshold.*” This itself is not an inaccurate root cause when forced to pick only one cause. However, there are many more factors involved in this incident: (1) there was an inrush of load from a single customer, (2) there was a lack of throttling on the customer end as well as the service end, (3) increased load significantly increased CPU and heap usage and thread count at the server, which lead to failed requests with exceptions, (4) the exception handling didn’t release some resources that were allocated by the failed requests, leading to resource leaks, (5) there were no automated watchdogs to detect early symptoms of the outage (or resource leaks), and (6) the team was unable to access their own metrics during the outage since the metrics were collocated with the service. In contrast, our analysis of the incident identifies all these factors and the corresponding tags in our taxonomy.

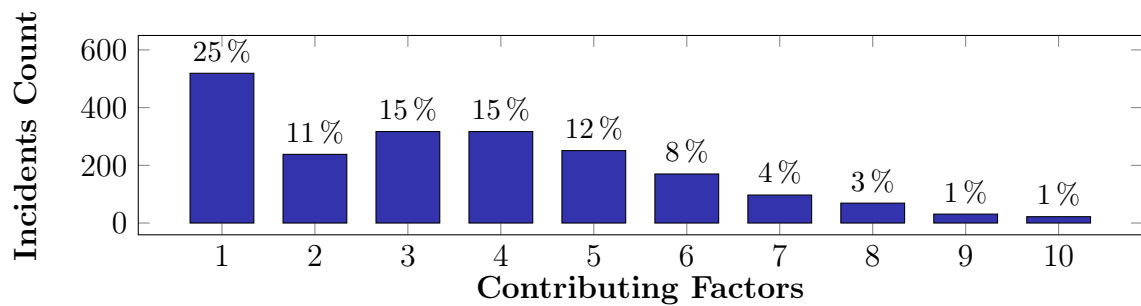


Figure 5.5: Distribution of incidents across number of distinct contributing factors (shown until 10 factors).

Figure 5.5 shows the distribution of the number of contributing factors behind each incident. As shown, over 75% of incidents have been caused by more than one contributing factors. And, more than 50% of the incidents have 4 or more contributing factors. On average, each incident has ≈ 3.6 factors. This reaffirms the need for holistically analyzing the incidents to understand all the contributing factors.

The presence of multiple contributing factors per incident has important implications. On one hand, identifying the possibility of such incidents before deployment to production with integration and end-to-end tests is challenging since testing needs to be performed in the

presence of multiple potential contributing factors (e.g., high load *and* no throttling *and* no monitoring of early symptoms). On the other hand, *preventing such an incident does not always require addressing all the causal factors, but only one (or a small subset) of them*. For example, the aforementioned incident could have been prevented by using proper throttling mechanism, *or* by fixing the resource leak bug, *or* by having monitors that can restart the service on early symptoms of resource leaks. This insight presents a unique opportunity to fix the incidents (by addressing the easiest causal factor); but it requires identification of all the causal factors (as we do) instead of identifying a single root cause.

Finding 2. *A wide-variety of factors contribute to production incidents.*

Our analysis identified a wide range of factors, including hardware, software, code bugs, underlying infrastructure to external dependency issues, configuration errors, deployment issues, and so on. Specifically, we have identified 346 root cause categories (i.e., contributing factors) for the 2051 incidents we analyzed. Table 5.1 shows the high-level root cause categories, each of which contains many finer-grained subcategories. The full list of categories and their respective frequencies observed in our analysis can be found in <https://AutoARTS-rca-taxonomy.github.io>. This contrasts our study with prior works that focus on a small set of root causes such as code bugs [LLL16, GDQ18, CDJ19, ZYJ21b, CLP14, LLM19d].

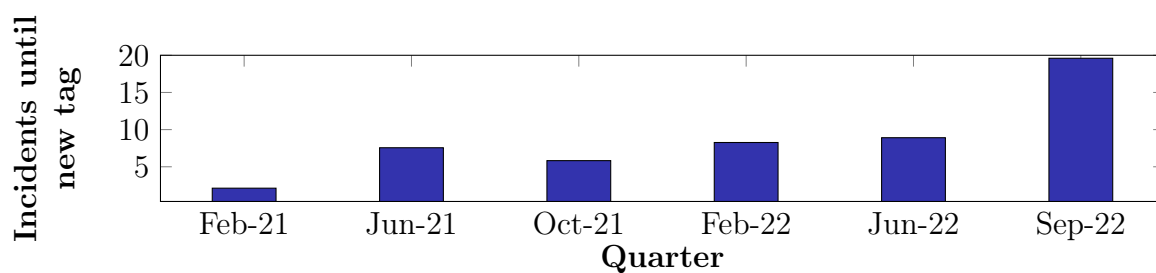


Figure 5.6: Average number of incidents successfully tagged until a new root cause tag is introduced across quarters.

Finding 3. *New root-cause categories keep appearing over time.*

As software and hardware systems evolve, novel root causes appear to contribute to their incidents. For example, when a service migrates to a containerized environment, its incidents may be caused by container-related factors. Similarly, when a service takes a new external dependency, it may start experiencing incidents caused by factors related to the failures of the new dependency. We analyze incidents in the same timeline as they appear and we create root cause categories incrementally—we create a new category only if none of the existing ones can precisely represent a new root cause. We observe that even though many common root cause categories (e.g., code bugs) appear in early incidents that we analyze, a few categories appear only in much later incidents (e.g., those happening two years after the first incident we analyzed). Figure 5.6 shows how often such new categories appear in our analysis. As shown, even after 1.5 years, new root cause categories appear, albeit with a smaller rate (i.e., we can tag higher number of incidents successfully before we need to introduce a new root cause category). The fact that novel root cause categories keep appearing implies that root cause labelling needs to be a continuous process to identify (and take actions on) emerging root cause categories. This calls for an automated solution.

Finding 4. *Lack of monitoring (i.e., observability) is the most common factor behind incidents.*

Table 5.1 shows the distribution of various contributing factors behind the incidents we analyzed (only high-level factors are shown). As shown, Detection is the most common contributing factor leading to outages. Detection related issues represent missing observability signals that prevent us from detecting early symptoms of problems, many of which could have been avoided, e.g., by rebooting the service, if their early symptoms were detected. We also analyzed finer-grained contributing factors from ARTS taxonomy. Table 5.2 shows distributions of the top ten contributing factors (a contributing factor X.Y.Z means factor Z is a specific case of factor Y, which is a specific case of factor X). As shown, Missing Alerts, which is a specific case of Monitoring, which is a specific case of Detection, is the most common contributing factor. Insufficient telemetry captured from services is also a major

Contributing Factor	Frequency
Detection.Monitoring.MissingAlert	34%
Authoring.Code.Bug.Change	25%
Detection.Monitoring.InsufficientTelemetry	18%
Detection.Validation.MissingTestCoverage	17%
Detection.Monitoring.CodeDeployment. InsufficientHealthSignal	9%
Authoring.Documentation. NoOrInsufficientTSG	9%
Architecture.SinglePointOfFailure	8%
Authoring.Code.Bug.Latent	7%
Detection.Monitoring.Synthetic	6%
Deployment.Mitigation.ManualTouch	5%

Table 5.2: Distribution of top 10 most frequent contributing factors in our analysis from the ARTS taxonomy.

contributing factor which also prevents from deploying automated alerts. An organizational policy on collecting key telemetry and defining automated watchdogs informed by this aggregate analysis can mitigate incidents (or severity) in the future.

We also analyze the most frequently *co-occurring* root causes to identify the pairs that jointly cause incidents. The two most frequent pairs are “*Authoring.Code.Bug.Change*” & “*Detection.Monitoring.MissingAlert*” (15%) and “*Authoring.Code.Bug.Change*” & “*Detection.Validation.MissingTestCoverage*” (11%). This aligns with our experience that many production incidents are caused by buggy code changes that are deployed without proper monitoring and testing.

Finding 5. *Incidents caused by deployment and datacenter related issues are the most time consuming to mitigate.*

In incident management, TTM is defined as the time elapsed between the start of the incident and when its customer impact was completely resolved. The higher the TTM, the more the customer impact and dissatisfaction. From Table 5.1, we can see that incidents

caused by Process and Datacenter related root causes have the highest mean TTM. Process related incidents have a high TTM because these incidents are caused by human errors and lack of standard operating procedures which result in non-trivial hard-to-resolve issues (e.g., accidental deletion of a database). Datacenter related incidents are caused primarily due to hardware failures which are quite complex given that there are multiple layers of capacity buffers all of which need to fail before an incident is caused by hardware issues.

5.4 ARTS Root Cause Taxonomy

We organize the root cause categories identified in our empirical study as a taxonomy of reliability tags that can be used to label PIRs of incidents.

Design goals. We have the following design goals in designing the taxonomy. First, the taxonomy should be *comprehensive* enough to capture not only the primary root causes of past incidents in Azure, but also other (secondary) contributing factors. Second, in order to avoid having a taxonomy too large to be easily used in practice, the taxonomy should be *sufficient* and it should include only the root causes found in past incidents. This implies that the taxonomy is continuously and organically grown to include new categories as they are discovered. Third, the tags should be *unambiguous*, to enable high-quality annotations. Finally, the taxonomy is organized *hierarchically*, for ease of labelling and updates.

We achieve the goals with a novel taxonomy called ARTS (Azure Reliability Tagging System) taxonomy. This taxonomy is built on top of the root cause categories identified by our empirical analysis described before. Figure 5.7 shows a portion of the hierarchical ARTS taxonomy expanded due to space constraints. We open source the taxonomy, with all tags and their descriptions, at <https://AutoARTS-rca-taxonomy.github.io>. We believe our open-source effort will foster future research and allow practitioners use our taxonomy. Even though the ARTS taxonomy is developed based on incidents in Azure, we believe that its categories are general enough to be used in any large-scale cloud system.

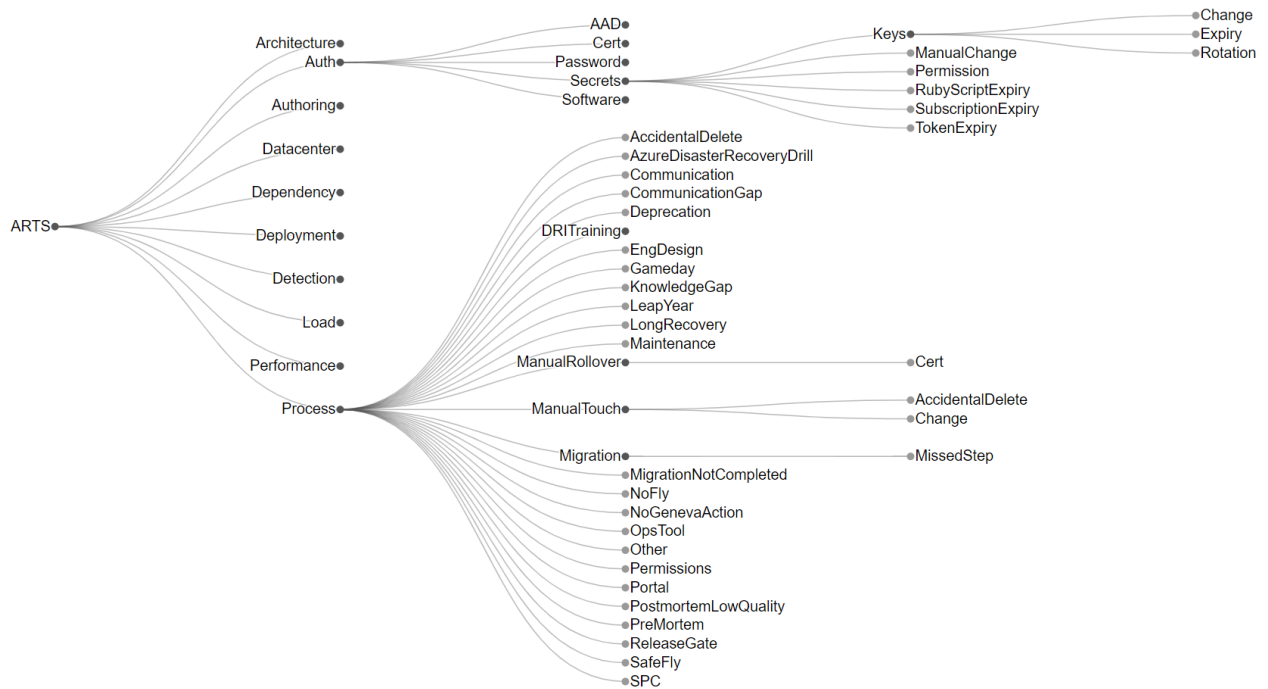


Figure 5.7: ARTS taxonomy visualized with partially expanded root cause categories. Left end of an edge indicates the parent root cause category and the right end indicates finer subcategory within the parent.

We start with a small number of tags representing orthogonal categories of themes (such as datacenter issues and authentication issues) and grow it horizontally to include new themes and vertically to include more specific sub-themes as new incidents are analyzed and existing themes/sub-themes deem inadequate. We have established a continuous feedback loop based process for building the ARTS taxonomy and tagging of new incidents on an ongoing basis.

For ease-of-use, we organize the ARTS taxonomy hierarchically, by grouping related sub-themes under one common theme. Currently it consists of four levels and contains 346 root cause categories identified from our empirical analysis. The top level consists of ten broad themes (shown in Table 5.1), each of which consists of multiple sub-themes. There are 346 leaf nodes, each representing a root cause tag with the name obtained by concatenating

the names of the path from the root to the leaf node. For example, the root cause of “a gap in pre-production detection due to missing integration tests” is represented with the tag “*Detection.IntegrationTest.Missing*” in which “*Missing*” is the most precise leaf-level tag. The hierarchical taxonomy naturally distinguishes between problem spaces at different granularities. In this example, if the root cause is that the integration tests existed but were skipped somehow, that representative tag would have the leaf-level tag “*NotRun*” instead of “*Missing*”.

As mentioned, the taxonomy is grown as new root causes are identified in newly analyzed incidents. Figure 5.6 shows how the taxonomy has been growing over time, with the y-axis showing the average number of incidents analyzed until a new tag needed to be introduced in ARTS. A larger value indicates better stability of the taxonomy: many incidents can be analyzed with existing tags. As shown, over time, the taxonomy can be seen becoming stable. Specifically, in the most recent quarter, only one new tag needed to be introduced after analyzing ≈ 20 incidents (i.e., after using ≈ 70 existing tags) on average. We hope to see significantly more stability in coming months.

5.5 AUTOARTS’s ML Models

We developed an automated tool called AUTOARTS that can assist a human in the labelling process of a PIR with two important tasks. First, AUTOARTS uses a multi-label classification technique to automatically analyze an incident’s PIR (written in natural language) and to identify *all* contributing factors and their representative ARTS tags. Second, AUTOARTS can produce a short text snippet (from the PIR) that captures important context explaining the factors. The snippet enables a human to easily review the selected tags without reading lengthy incident reports or PIRs. We now describe how AUTOARTS uses ML techniques and the ARTS taxonomy to achieve the above. Figure 5.4 shows the architecture and components of AUTOARTS.

5.5.1 Identifying Root Cause Labels from the ARTS Taxonomy

AUTOARTS uses multi-label text classification to classify a PIR into a set of ARTS labels. One key challenge we face is that conventional multi-label text classification algorithms that treat each class as opaque and independent, require sufficient labelled data for each class to achieve good accuracy. However, even though we have a collection of labelled data, many of the fine-grained labels (i.e., ARTS contributing factors) contain very few labelled samples (i.e., PIRs). Specifically, 68% of the labels from ARTS taxonomy have fewer than 10 labelled samples from our dataset, which is adversarial to classification accuracy.

To address this, we leverage the hierarchical relationships between root cause labels in the taxonomy using Graph Convolutional Networks [KW16]. This enables transfer of knowledge from the categories with adequate labels to categories with few labels (§5.6.3). In particular, we apply a hierarchical text classification model called HiAGM [ZML20]. Contrary to the conventional multi-label text classification methods that disregard the holistic label structure for correlational features, we attempt to fully utilize the relationships between the text features and labels, as well as inter-label dependencies. As an example, the “*Authoring.Code.Change*” label only has 13 samples, making training difficult. However, by modeling the root cause taxonomy as a graph, we can transfer knowledge from the “*Authoring.Code.Bug*” label, which has 733 samples, to “*Authoring.Code.Change*” since they share the features of their same parent root cause label, “*Authoring.Code*”.

Given a Post-Incident report $x = (w_1, w_2, \dots, w_n)$ with n tokens, the sequence of token embedding is initially fed into a bidirectional GRU neural network [CGC14] to extract text contextual features. Following the GRU model, multiple CNN layers are employed to generate the n-gram features. The top-k max pooling layer is then applied to obtain the overall text representation $S \in \mathbb{R}^{n \times d_c}$ that highlights the key information, where n is the top-k output dimension of CNN layers and d_c represents the embedding dimension.

To model the ARTS taxonomy, we formulate the taxonomic hierarchy as a directed acyclic graph $G = \{V, E_t, E_b\}$, where V refers to the set of label nodes. E_t and E_b represent the top-down and bottom-up hierarchy paths respectively. To encode the hierarchy graph, a GCN-based hierarchy encoder [KW16], Hierarchy-GCN, is used to aggregate data flows within the top-down, bottom-up and self-loop edges based on the associated neighborhood of each node. The GCN-based graph encoder adapts the convolution concept from images to graphs, in which the graph convolutional operator can effectively convolve the multi-order neighborhood information by forming multiple propagation steps during the forward pass. For each node, the feature information is aggregated by the node feature from all the neighbors, including the node itself, to leverage the graph structure of label taxonomy.

Next, we aggregate the features of texts and labels together using a label-wise attention mechanism [YYD16]. Specifically, the attention α_{kj} , which indicates how informative the j -th text feature is for the k -th label, is calculated as follows:

$$\alpha_{kj} = \frac{e^{\mathbf{s}_j \mathbf{h}_k^T}}{\sum_{j=1}^n e^{\mathbf{s}_j \mathbf{h}_k^T}}, \quad (5.1)$$

where \mathbf{s}_j is the j -th text feature of the root cause input and \mathbf{h}_k represents the k -th node in the label hierarchy. The label-aligned text feature $\mathbf{v}_k = \sum_{i=1}^n \alpha_{ki} \mathbf{s}_i$ for the k -th label is then obtained and fed into a classifier for hierarchical label prediction.

Finally, we flatten the label hierarchy by treating all nodes as leaf nodes for multi-label classification, regardless of whether a node is a leaf or an internal node. A binary cross-entropy loss function is employed to train the model using the ground truth and predicted sigmoid score for each label. In addition, a recursive regularization for the parameters of the final fully connected layer is used to encourage classes nearby in the hierarchy to share similar model parameters.

$$L_r = \sum_{i \in C} \sum_{j \in \text{child}(i)} \frac{1}{2} \|\mathbf{w}_i - \mathbf{w}_j\|^2, \quad (5.2)$$

where the node j is the child of node i in the label hierarchy.

5.5.2 Extracting Root Cause Context from Postmortems

The objective of context extraction is to extract key text snippets from a given Post-Incident Report (PIR) for on-call engineers to reason the contributing factors of an incident without reading the complete lengthy report. Many text summarization techniques exist. *Abstractive summarization*, where summaries may contain generated sentences, is not a good fit for us since our goal is to select and highlight existing texts in the PIR, as is done by *extractive summarization*. However, existing language models such as BERT [DCL18] and XLNet [YDY19] are trained on large corpuses such as Wikipedia articles, etc., where the syntax and semantics of the language used is quite different from what is observed in PIRs due to domain-specific usage of words (e.g., ‘Fabric’ in networking terminology vs clothing) and different vocabulary. Moreover, existing extractive summarization models are trained on and their traditional usage in *summarizing* text documents, which is different from context extraction from PIRs. We therefore finetune an existing model called Pegasus [ZZS20] with our labelled data (from § 5.3.1).

In Pegasus, important sentences are removed or masked from an input text and are generated together as one output sequence from the other remaining sentences, similar to an extractive summary. Hence, Pegasus is amenable for context extraction from PIRs, because we can mask the key sentences identified in our analysis § 5.3.1 to finetune the model parameters. Using the standard Transformer encoder-decoder, Pegasus model is pre-trained on two enormous text corpora: 1) Colossal and Cleaned version of Common Crawl (C4) [RSR20], which comprises of text from 350 million web pages with a size of 750 gigabytes; 2) HugeNews [ZZS20], a dataset of 1.5 billion news articles gathered from 2013 to 2019. Similar to MLM tasks for predicting masked tokens, a new pre-training task called Gap Sentences Generation (GSG), is applied to fill the masked sentences. Three different strategies are applied for selecting m gap sentences without replacement from a document. The first method is to uniformly select m sentences at random, whereas the second strategy is to simply select the first m sentences. The aforementioned two strategies are combined with

the Principal strategy, in which top- m scored sentences are chosen based on their significance as measured by rouge score [Lin04] without the selected sentence. Formally, the score s_i of the i -th sentence x_i can be expressed as follows:

$$s_i = \text{rouge}(x_i, D \setminus \{x_i\}), \quad (5.3)$$

where the D is the set of all the sentences in the document and rouge function is a commonly employed metric for evaluating how good an automatically produced summary against a reference summary.

Even though Pegasus has been pre-trained on massive datasets, it is not trained to generate context from root cause descriptions in software engineering domain. To completely comprehend the context extraction task in our domain, we utilize the human-labeled context to further fine-tune the Pegasus model as a sequence-to-sequence task.

The root cause of this monitor alert was that a lot of subscriptions could not deploy VMs on indiacentral region.

... .. (omit 132 words)

This problem occurred because the traffic that was re routed to AZSM could not be handled. This problem occurred on indiacentral prod b and indiacentral prod b. As part of increasing inventory we have introduced news sets of AMD clusters. The AZSM services on these clusters still needed some configuration and build out related processed to be completed. Hence these clusters stamps could not handled the traffic re routed to them. The traffic was routed as part of default behavior. We are going to change this. The Fabricator clusters started taking tenant traffic even though their corresponding Az SM clusters weren't ready. This was done as part of flighting on Broad clusters. India Central was one of the region for this flighting. We did not anticipate a case where new build out clusters would not be able to take the new traffic. This was detected as part of the API failure monitor. We will be working on adding more robust feature specific monitoring and more strict rollout to not encounter this failure again.

Figure 5.8: Context extraction from a redacted PIR. Green sentences are extracted by both our model and human expert, red are extracted by model only and blue are extracted by human only.

Figure 5.8 illustrates an example of context extraction from a PIR report consisting of 328 words. The human-labeled context is shown in blue and green, whereas the context extracted

from Pegasus is shown in the green and red. This example shows that $\approx 50\%$ tokens can be filtered, which can considerably enhance the efficiency with which on-call engineers read the PIR report.

5.6 Evaluation of AUTOARTS

To empirically, evaluate the performance of AUTOARTS, we use the labelled dataset from our manual analysis (§ 5.3.1).

5.6.1 Methodology

Dataset. Our dataset consists of 1120 PIRs that are expert-annotated with ARTS root cause tags and contextual sentences to justify them. We use stratified sampling to divide this dataset into train(72%) and validation(8%) splits to train and tune the hyperparameters of different models and test(20%) split to report the results with the trained models.

Data Pre-processing. We found that engineers often included various types of data such as debugging queries issued on logs, error messages, stack traces, screenshots, etc., in PIRs (also identified in [SBK21]). These add significant noise to the vocabulary of the language processed by NLP models, without contributing to performance. We carefully remove such noise with regular-expression based filters and only select alphabetic text for our evaluation. For experiments in § 5.6.2, we also use the NLTK [Bir06] library to remove stop-words and extract stems of words to construct vocabulary.

Evaluation Metrics. For root cause classification, we use micro-F1 score to analyze performance across different incidents with multiple labels. We also use weighted-F1 score to analyze performance across different classes since our dataset is imbalanced as shown in Table 5.1.

For context extraction, we use ROUGE (Recall-Oriented Understudy of Gisting Evaluation) [Lin04] and BLEU (Bi-Lingual Evaluation Understudy) [PRW02] scores to evaluate the similarity of extracted context against the ground truth. Rouge-N score is based on the percentage (higher the better) of N-grams from the ground truth that are present in the extracted context. BLEU-N score indicates the percentage of N-grams from the extracted context that are present in the ground truth. Rouge-L F1-score is based on the longest common subsequence (not necessarily consecutive) between the extracted context and target context.

5.6.2 Featurization

Sophisticated DL language models impose constraints on input sequence length (e.g., 512 tokens for BERT [DCL18]). The limit is much smaller than our preprocessed PIRs (avg. length of ≈ 1900 words) and hence we cannot train DL models with the entire PIRs. However, a PIR is organized into multiple sections and not all sections are equally important for root cause classification. We therefore conduct an ablation study by featurizing each section in the PIR into Bag-of-Words encodings and classify them to top-level root cause categories using a Random Forest classifier.

Section	Micro-F1	Weighted-F1
Whole PIR	0.55	0.40
Title	0.53	0.45
Summary	0.47	0.46
RC-Details	0.52	0.45
5-Whys	0.54	0.40
Discussion	0.53	0.40
Mitigation	0.47	0.40
RC-Details + 5-Whys	0.56	0.42

Table 5.3: Study on the utility of different PIR sections in top-level root cause classification using Random Forests.

Table 5.3 highlights that “root cause details” and “5-Whys” sections achieve better micro-F1(1.8% higher) and weighted-F1 (5% higher) scores compared to using the whole PIR. These sections capture information relevant to root cause classification and by only using them, we minimize sequence length to meet constraints imposed by DL models.

5.6.3 AUTOARTS’s Performance on Root Cause Labelling

Table 5.4 compares the level-3 root cause classification performance of our trained HiAGM model against a flattened version of our hierarchical taxonomy (HiAGM_Flat), where we remove parent-child relationships between different root causes in the taxonomy and consider all the root causes tags to be opaque and independent of each other.

Model	Micro-F1	Weighted-F1	Precision	Recall
HiAGM	83.16	89.63	71.17	100.00
HiAGM_Flat	45.40	68.66	65.61	74.11
BERT_MLC	42.29	46.85	43.63	53.26

Table 5.4: Performance of HiAGM compared to using flattened root cause taxonomy (HiAGM_Flat) and a finetuned-BERT based multilabel classifier (BERT_MLC).

We also compare HiAGM against a multi-label classifier (BERT_MLC) with the flattened version of the taxonomy using finetuned BERT [DCL18] model to encode the PIR text. We observe that HiAGM performs significantly better (31% higher weighted-F1 measure) than HiAGM_Flat indicating the utility of GCN to leverage neighboring relationships between root causes and the need for root cause taxonomies to be hierarchical. HiAGM performs significantly better (91% higher weighted-F1 measure) than BERT_MLC along with HiAGM_Flat (47% higher weighted-F1 measure), demonstrating no utility in finetuning existing language models on PIRs.

5.6.4 AUTOARTS’s Performance on Context Extraction

Using the train and validation splits of the dataset, we finetune Pegasus for 15 epochs and T5 (3 Billion parameters) [RSR20] for 7 epochs and report the results on the test set.

Model	ROUGE			BLEU			
	Rouge-1	Rouge-2	Rouge-L	BLEU	BLEU-1	BLEU-2	BLEU-3
Pegasus - (P)	32.55	18.72	24.30	9.61	18.03	10.31	8.93
Pegasus - (F)	45.46	35.65	38.43	24.60	32.19	24.98	23.41
T5 - (P)	34.38	23.31	28.03	10.06	15.68	10.83	9.43
T5 - (F)	41.63	33.86	35.76	23.81	29.81	24.10	22.70
BERT-cased - (P)	40.05	27.03	31.01	18.62	28.43	18.95	16.83
BERT-cased - (F)	40.08	27.35	31.20	18.80	28.32	19.03	16.95
BERT-uncased - (P)	39.52	26.58	30.74	17.63	27.47	17.98	15.89
BERT-uncased - (F)	39.92	27.44	31.57	18.64	28.08	18.91	16.90

Table 5.5: Performance of Pegasus and T5 models. (P) indicates Pre-trained versions and (F) indicates fine-tuned versions. We also present performance of unsupervised clustering based approach for extractive summarization using BERT.

Table 5.5 compares the performance of finetuned Pegasus model against baseline approaches using T5 and clustering-based extractive summarization [Mil19] using BERT. We can clearly see that our finetuned Pegasus model achieves the highest performance across various ROUGE and BLEU metrics. We observe a significant (58.15%) improvement in Rouge-L score as a result of finetuning Pegasus, because pre-trained version of Pegasus is trained on significantly different domains of language such as news articles, etc., and is trained to summarize them, which is different from context extraction. We also observe a 7.6% increase in ROUGE-L score compared to the finetuned-T5 model, because Pegasus extracts sequences of text from the PIR as opposed to T5 which generates new sequences of words, which might not represent the content present in the PIR which is where engineers derive their context from. Finetuned-Pegasus performs significantly higher (21.73%) than unsupervised clustering based summarization approaches using finetuned-BERT, because summary of the PIR doesn’t represent the context.

5.6.5 User Study

To evaluate the utility of AUTOARTS, we randomly sampled 10 example incidents and the tool’s generated contexts, the corresponding root cause categories and presented them to one of the leads that developed the ARTS taxonomy (by studying PIRs). These were examples that were tagged by them in the past that are not used for training any of our models. The goal of this study is to understand, for each example: (Q1) How useful the generated context is in identifying all the contributing factors that they identified, (Q2) If the generated context has extra details that are not useful for identifying contributing factors (to evaluate the succinctness of our generated contexts), (Q3) Whether the generated context can help them identify any new contributing factor that they have not identified previously (to evaluate the generalization of the model’s outputs beyond accidental False Negatives in ground-truth) and (Q4) Whether the tool missed the most important contributing factors (to evaluate the importance of False Negatives).

Metric	Response	Description
(Q1) Usefulness of generated context to identify contributing factors	4.6/5	1 - Not useful at all, 5 - Very useful
(Q2) # Contexts generated with unnecessary details	0/10	No unnecessary details in generated contexts
(Q3) # New Rootcauses from generated contexts	2	False negatives identified by AUTOARTS
(Q4) # Examples with a crucial Rootcause tag missing in classification	7/10	Crucial contributing factor missing from predictions

Table 5.6: Quantitative user feedback from an expert over the effectiveness of AUTOARTS across context generation and root cause classification tasks over a randomly chosen set of 10 incidents.

Although we quantitatively evaluated the syntactic similarity of generated contexts to the ground truth, the developer study helps us understand if they are semantically similar and ultimately usable by a human (OCE). Similarly for Root cause classification task, the

relative severity of each individual contributing factor is not identifiable from ground truth (no ranking). Q4 helps us understand if the predicted contributing factors miss any crucial tag from the ARTS taxonomy.

We quantify response to Q1 on a Likert scale of 1 to 5, where 1 meant 'not useful at all' and 5 meant 'Very useful'. Q2-Q4 were answered as a binary Yes/No, by providing clarifying responses wherever necessary for sanity check. Table 5.6 shows the utility of the tool based on the subject's responses to Q1-Q4. We find from the study that the contexts generated by the tool are extremely useful in identifying all the contributing factors and they are succinct enough without presenting additional information that is not useful in identifying contributing factors. In addition to this, we also found that our tool helped the subject find 2 new root cause tags that should have been assigned to these incidents in the past, highlighting the difficulty in manually sifting through postmortem reports to identify contributing factors.

At the end of the experiment, the subject was asked to answer on a scale of 1-5 (5 being very useful, 1 being not useful at all), indicating the overall usefulness of our tool to assist them in their task based on the 10 examples. The subject rated our tool 'above 4.5'. In addition to this, the subject's verbatim feedback on our tool — **'This tool is very useful from context generator perspective for the root cause classification task. From the Tags perspective, if we had 4th level for just code change related tags this is very useful for change management standards team. Need to fix the dependency tags related logic as it's defaulting to "Data Bricks". Over all I am very happy with this tool'** — highlights the utility of our context generation and the lapses in automated root cause classification. The imbalance in tag distribution over our training set resulted in misclassifying incidents with tags that do not have sufficient supporting training samples. Overall, the feedback indicates the promise for deploying the tool for practical use in assisting engineers by providing enough context from PIRs to assign root cause tags from ARTS taxonomy.

5.7 Related Work

5.7.1 Root-cause analysis of past incidents

Root-cause analysis of incidents and outages and defining taxonomies to capture their root-causes has been a popular topic of study in the software engineering and systems community. We find that prior work can be categorized into two major buckets. The first category of prior work focuses on specific type of production issues such as software bugs[LLL16, GDQ18, CDJ19, ZYJ21a, CLP14, LLM19d] or network issues [GMK16]. The other category focuses on specific services or systems such as big-data systems[YLZ14b], business data processing platform[DKI14] and a specific cloud service[GSB22]. In contrast, we consider a large-scale cloud system consisting of many hundreds of services and *all* types of failures including hardware and software, infrastructure and application, software code and configuration, and so on. Prior work that propose solutions to simplify the task of identifying root-cause of a failure [SBN21, DNS22] are orthogonal to our focus.

5.7.2 Text summarization & root-cause classification

Text summarization [SKR21] is the task of rewriting a long document into a condensed form while retaining its essential meaning. The most prevalent paradigms for summarization are extractive and abstractive based approaches. When generating summaries, abstractive summarization approaches [XZW22] are typically considered as a sequence-to-sequence learning problem [NZG16, SLM17, PXS17], whereas extractive summarization methods [GDR18, ZLC20] extract key sentences as summary directly from the text. In our context extraction task, we not only to condense the postmortem context, but also to extract essential descriptions covering different root-cause factors.

Prior work focused on diagnosing different kinds of incidents such as, Rex [MBK20] diagnoses misconfigurations using syntax trees, DeepAnalyze [SBN21] identifies culprit frame

in WER crash stack traces. Orca [BKM18] identifies buggy commits using differential code analysis and provenance tracking. SoftNER [SBK21] analyzes postmortem reports to extract entities. To the best of our knowledge we are the first to classify incident postmortems into an extensive high-granularity taxonomy.

CHAPTER 6

Coarse Causal Reasoning in Telemetry

This chapter provides an overview of a joint work with Vikramank Singh, Zhengchun Liu, Murali Narayanaswamy and Tim Kraska. As mentioned in § 1.3, existing practice is to manually reason telemetry across different teams to diagnose the root cause of a performance degradation in large scale distributed systems. However, engineers’ domain expertise over large-scale services is narrow due to modularity and existing causal discovery techniques cannot scale to production-scale telemetry due to computational complexity and to the large and dynamic execution state of distributed systems due to missing confounders.

Specifically, we address this problem by presenting a domain-agnostic featurization technique that enables leveraging wide range of telemetry. We then use scalable anomaly detection over the telemetry as a proxy to the execution states of the distributed system to enable causal discovery across the telemetry to assist engineers in root cause analysis. Contrary to traditional method of defining a causal model and running causal inference atop the model, we dynamically discover causal models from anomalous runs due to the aforementioned challenges. To the best of our knowledge, our approach is able to identify the largest number of cause-effect relationships from production telemetry.

6.1 Causal Discovery for Root Cause Analysis

One effective approach to reason performance degradations in a distributed system is to identify cause-effect relationships [ICM22, GLD21] for an engineer to diagnose the root cause

of an incident. A causal model is a directed graph, where each edge indicates a cause-effect relationship between the source and the destination respectively. This can enable on-call engineers (OCEs) to quickly identify the root cause and take mitigative actions to restore normal performance in the service. Note that, today engineers have an approximation of bits and piece of this causal model based on their scope of operation in a large-scale services distributed as microservices.

6.2 Overview - Reasoning Slow Queries in *Anon2*

To reason events in a system when an incident is detected, engineers instrument their service code with logging frameworks. In production services like *Anon2* hundreds of metrics capture different aspects of a query execution during run time. These, when tied together with an engineer’s hard-won experience can help diagnose the root cause of an incident (e.g., why a query is slow). When analyzing an incident for it’s root cause, engineers manually sift through the plethora of logs to identify “suspicious” metrics based on their own narrow and unstructured domain-expertise over the database service (e.g., CPU usage is typically 30% but it is 90% during the incident). They hypothesize a possible root cause that led to these symptoms and validate it. If it is valid, the investigation is concluded, the findings are reported and any useful mitigation/resolution steps are taken to prevent it from recurring. On the other hand, if their hypothesis is invalid, they use this information to rule out certain root causes and iterate this process until they find a valid root cause hypothesis.

6.2.1 Challenges

Limited Domain-Expertise. Engineers are limited by their narrow domain-expertise over system behavior due to modularity and hence, define simplistic alert rules (e.g., $Avg_{5mins}(CPU) > 60\%$) over a small set of telemetry which without additional context cannot help in root cause diagnosis and result in false alerts which are cognitively burdening

to investigate. This also results in stale rules due to high-risk in potentially missing an incident due to removal of an alert rule by another engineer.

Telemetry Scale and Granularity. Production telemetry captures hundreds of metrics and the size of outputs from anomaly detection and clustering techniques quickly exceed a handful to meaningfully assist engineers in root cause analysis. Telemetry is also reported at various granularity (e.g., Disk usage at machine-level, CPU usage at segment-level, etc.) whose cardinality varies across queries due to execution of varying number of segments across varying number of machines.

Dynamic Cause-Effects. Causal inference techniques rely on a graphical model that capture cause-effect relationships between various telemetry. This graphical model is large (in width and depth) and existing causal discovery methods are prohibitively expensive to discover it. Moreover, this graph is highly dynamic due to missing confounders from telemetry due to control flow branching (results of 'If' conditions in code), system-level events, etc. For these reasons, causal discovery methods cannot identify graphical models that capture cause-effect relationships between telemetry in large-scale production services.

To better understand these woes, consider an example from *Anon2*. An ML-based predictor determines whether an incoming query is 'short' or 'long' for the workload manager to enqueue the query into an appropriate queue for execution with a set time-out duration. Due to a mis-prediction from this model, a 'long' query can be put into a 'short' query acceleration queue and suffer from queueing delays. The telemetry captures the queueing time, ML model prediction but whether the prediction is correct/wrong is *not* captured. There can be a variety of other reasons that cause high queueing time (e.g., workload size, long queries). For a causal discovery method to identify that the ML model can cause high queueing delays *only* during a misprediction is non-trivial as this information is missing. On the other hand, telemetry consists of several other instances of query execution where (a) there is no cause-effect relationship (queueing affected due to other reasons), (b) there is a positive cause-effect relationship (correct prediction), (c) there is a negative cause-effect

relationship (misprediction) and (d) there is a flipped cause-effect relationship between the two variables (if the ML model relies on current queuing delays for a prediction). This example also highlights the need to leverage available telemetry as no resource (e.g., CPU, I/O) is overloaded and even if queuing delay is used in existing tools, it is unclear why the queuing delay is high for root cause analysis. When a set of engineers manually built a graph that captures cause-effect relationships between telemetry, the ML prediction is not included in the graph demonstrating the limitations of a narrow domain-expertise.

6.3 Limitations of Causal Discovery

To understand the resource and time usage of causal discovery methods, we created synthetic datasets across varying number of normally distributed random variables and samples. For each random variable in the synthetic datasets, we randomly choose a distinct mean (μ) from a large range (10x the number of variables) and a standard deviation (σ) to sample values from a normal distribution with (μ, σ) . For each such dataset with V random variables and S samples, our goal is to measure the execution time of a causal discovery algorithm to identify all the causal dependencies between the random variables.

We chose PC algorithm with Fisher’s z-test as the conditional independence test implemented in an open-source python library. We use a machine with 32 physical CPU cores and 64GB RAM running a Linux operating system for the experiment. We set a 24 hour duration as the time out for experiments. Table 6.1 shows execution times of causal discovery on datasets with V ranging from [10, 750] and S ranging from [1k, 50M]. Though these datasets are randomly generated, they can potentially be correlated and the causal discovery algorithms can discovery correlations or potentially causal dependencies between variables. As shown in Table 6.1, causal discovery methods cannot scale to the scale of production telemetry (that is beyond 750 features) from millions of queries.

Number of Queries	Size of Telemetry					
	10	50	100	250	500	750
1000	.03s	.44s	5.88s	5.85m	1.34hr	T/O
10000	.02s	.57s	9.77s	37m	4.67hr	T/O
100000	.02s	.77s	10.82s	2.5hr	T/O	T/O
1000000	.13s	1.6s	24.01s	15.78hr	T/O	T/O
10000000	1.26s	7.7s	74.76s	T/O	T/O	T/O
25000000	3.12s	17.32s	60.23s	T/O	T/O	T/O
50000000	6.31s	39.42s	T/O	T/O	T/O	T/O

Table 6.1: Time-To-Finish for Causal Discovery on a controlled synthetic dataset using the PC algorithm on a 32 core machine with a 24hr timeout.

6.4 PERFRCA

We design PERFRCA, as shown in Figure 6.1 to enable causal discovery between telemetry to assist engineers in root cause analysis of slow queries in *Anon2*. Our principal hypothesis is that anomaly detection or clustering based approaches for root cause analysis of performance degradation in large-scale production systems are cognitively burdening on engineers due to the volume of telemetry. For example, for an engineer to reason 30 anomalous metrics out of 900 metrics is non-trivial and sometimes prohibitive due to the modular nature of service development where only a handful engineers can reason a certain metric from components developed by them. To address this issue and other practical challenges that arise in large-scale systems, PERFRCA outputs causal relationships between production-scale telemetry relevant for diagnosing the root cause of a slow query.

To do this, PERFRCA employs offline learning to learn a knowledge base of causal graphs between various subsets of production telemetry and leverages this knowledge base to present the engineer with the appropriate causal graph in an online fashion. As discussed in the challenges, existing causal discovery methods neither scale to the volume of production

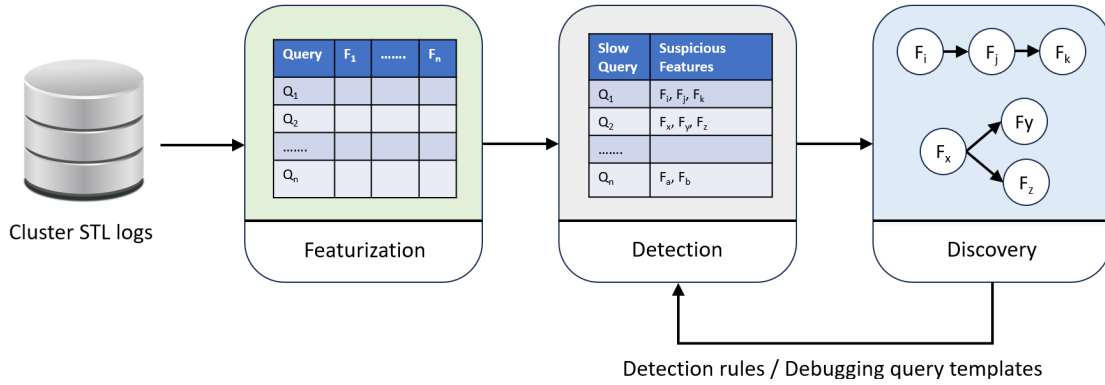


Figure 6.1: Overview of PERFRCA as dynamic causal discovery over anomalous features from a large set of telemetry of slow queries.

telemetry nor identify causal relationships between telemetry due to the dynamic relationships resulting from the complexity of modern large-scale database systems.

6.4.1 Feature Computation

Anon2 captures diverse telemetry across various aspects of query execution and infrastructure usage on production clusters. *Anon2* is a columnar database and to optimize the execution of a query for its latency, each query is compiled by a compiler to generate an execution plan. A given query can run simultaneously across multiple machines as segments which are further divided into sequential steps. To optimize for resource utilization in production clusters and throughput of query workloads, *Anon2* uses a workload manager that categorizes queries into multiple service classes. The workload manager estimates execution times and resource usages of queries based on recent executions to schedule queries in a given production cluster.

Due to this nature of query execution, telemetry captured in *Anon2* is heterogeneous in their reporting granularity. For example, resource utilization metrics are reported at machine level, latency is reported at segment and query levels. To add to this problem, each query can have different number of segments dependent on the tables, query operators, etc.,

which means heterogeneity in the number of metrics collected for each query. Further, we have observed that engineers use several notations to indicate the granularity of a metric in ad hoc ways. For example, in a telemetry table with columns ('Query ID', 'Segment ID', 'Latency'), each row in the table indicates the latency of a segment within a query's execution but when the value of 'Segment ID' is -1, the latency value indicates the latency of the overall query execution. Telemetry captured is also heterogeneous in its data types. For example, some metrics like latency are real valued (ordinal) where as service class identifiers are symbols (nominal). Given the heterogeneity in telemetry that is the result of diverse types of metrics captured and the execution. We also found that multiple metrics captured in different telemetry tables can capture the same metrics denoted by a different metric name for various reasons. Many existing approaches employ domain-expertise to process telemetry and cherry pick a limited number of metrics. Often largely composed of generic metrics such as CPU utilization which are not very meaningful for an engineer without additional context specific to the query/service.

PERFRCA employs a feature computation technique that can scale to increasing telemetry sizes and deal with heterogeneity in data types, reporting granularity and sizes of query execution plans. We separate telemetry tables to: 'query-level' tables whose rows have a query identifier and 'system-level' tables whose rows do not have a query identifier indicating that these tables capture metrics independent of a specific query. To tie metrics captured across these 2 categories of tables automatically, we leverage timestamps of queries and automatically detect identifiers for 'system-level' tables in 'query-level' tables. For example, 'machine ID' is an identifier for 'system-level' table and a 'query-level' table contains mapping between query ID and the machine ID to attach metrics from that particular machine to the query's metrics.

We facilitate engineers to specify through a configuration file: (1) omitting certain metrics that they know are irrelevant for debugging (such as customer account id, etc.), (2) for precise data type inference, specify the data type of a column in telemetry as one of ('nominal',

‘ordinal’ or ‘timestamp’) to process it appropriately, (3) the ‘golden’ metric based on which we identify a ‘slow’ query and (4) to specify naming conventions in telemetry such as all timestamps with column name prefix ‘start_’ indicating the start timestamp of an event. We argue that the above steps are relatively much stable to be only updated when adding a new metric to the telemetry and therefore a one-time specification, for example the data type of a metric does not change.

With the resulting metrics for each query (at various granularity), we aggregate metrics in each column in to a list of values. We automatically identify metrics that are at query-level by confirming singleton list of values across all queries. For all the other metrics, we compute the following statistical aggregates for ‘ordinal’ data type metrics per query: (‘mean’, ‘standard deviation’, ‘max’, ‘min’, ‘q25’, ‘q50’, ‘q75’, ‘q90’, ‘q95’, ‘q99’), such as mean CPU utilization across all segments of a query. Note that we eliminate any negative values (which are largely used as symbols) such as -1 when computing these features. These features address the issue of different queries having different number of segments while simultaneously providing rich features that can expose and distinguish between spikes, level shifts, jitters accurately. These features also address the issue of ad hoc conditional values such as value 0 indicates no timeout duration and non-zero values indicate the timeout duration for a query. For ‘nominal’ data type metrics, we compute the frequency of each symbol such as the number of segments with service class ‘high’.

For ‘ordinal’ metric features, we remove duplicate features by computing cosine similarity between 2 features across all queries and with a high threshold (0.99). This addresses the issue of same metric being reported with two different names in different tables. This also the issue of heterogeneous granularity where certain metrics are only reported twice for example per query where it is redundant to maintain multiple statistical features such as quartiles, etc.

6.4.2 Anomaly Extraction

It is hard to determine if a high response time of a query is expected or not because of the complexity of *Anon2* service and the diversity in the query workloads. For example, cache misses, increase in number of rows in a table due to recent data ingestion, insufficient resources, etc., in isolation or in various combinations are service-level causes for high response times of a query. Query-level differences such as queries with JOINS can implicitly demand more compute resources and therefore higher response times as compared to those of queries with COUNTs which can be optimized.

To identify slow queries, we isolate query-level differences by clustering query telemetry across ‘same’ queries i.e., queries that have the exact same compiled execution plan over the same set of tables. From the resulting cluster of queries, we sort them by their response times and identify the slowest 10% of queries as ‘anomalous’ queries and the fastest 80% of the queries as the ‘baseline’ set of queries. For each of the features computed over query telemetry, we compute the mean and standard deviation of observed values over the baseline set. For each of the queries in the anomalous set, we identify the subset of features whose z-score is atleast 3 as the relevant features to reason the performance degradation.

For each of the resulting subsets of relevant features, We identify their corresponding values across slow queries from all query clusters with the same relevant feature subsets.

6.4.3 Causal Discovery

We hypothesize that the set of slow queries with the same relevant feature set share a common cause of performance degradation, which is a common assumption in other related works such as DBSherlock [YNM16] and iSQUAD [MYZ20]. The number of relevant features identified from anomaly extraction in a production-scale telemetry makes root cause analysis a cognitively burdening task on engineers. Through anomaly extraction, we only identified relevant features correlated with the increase in response times of slow queries. To further

assist engineers in root cause analysis of slow queries, we learn cause-effect relationships between relevant features as a causal graph using causal discovery algorithms.

6.5 Preliminary Results

To evaluate PERFRCA, we considered 668 production clusters from two zones of *Anon2* within the US. Over 70 million queries were ran over one week and of these queries we try to identify 2M queries that are significantly slow compared to their baseline. PERFRCA was able to identify 1000s of edges in causal models across these queries within a few minutes, where otherwise we would not be able to.

6.5.1 Case Studies

We present 2 case studies to showcase the effectiveness of PERFRCA through a user study. In the first example, WLM (Workload Manager) misclassifies query as ‘short’ query and assigns timeout value. Hence, it enqueues the query into a Short Query Acceleration queue and the query execution exceeds timeout duration. WLM then has to move the query to another queue causing behind other queries resulting high queueing time. Figure 6.2 shows PERFRCA’s output which clearly indicates that when the total queue time is high for these slow queries, the ‘is_short_query’ value was also high (0 indicates long query, real value indicates short query) clearly indicating a misclassification from WLM. It further shows that the actual execution time of the query was also high in those cases. We confirmed from engineers that this is indeed a major root cause of slow queries and the causal graph is an accurate representation.

In the second example, missing data on disk (due to large number of queries working on different data) causes high number of prefetches from AWS S3. These prefetches cause an increase in memory and disk usage. Leads to high RA (Read-Ahead) points, which is *Anon2*’s way of tracking resource usage in clusters. This ultimately causes high queueing

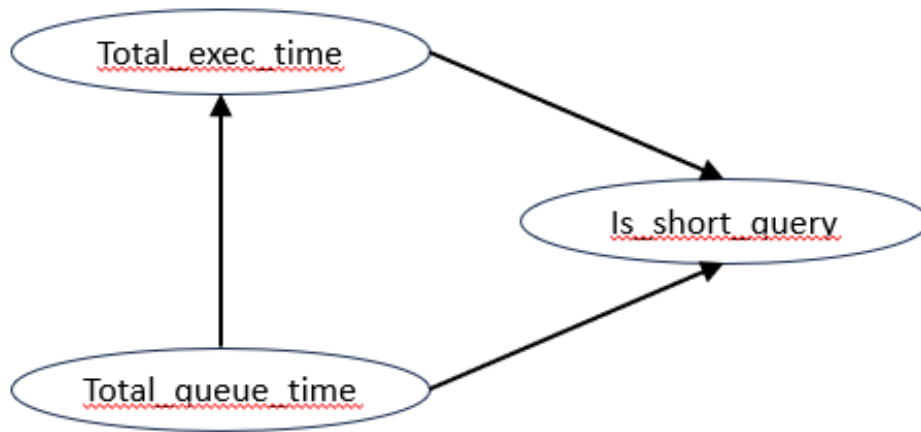


Figure 6.2: Output of PERFRCA for a WLM misclassification.

delays due to time taken to fetch data from S3. Figure 6.3 shows PERFRCA’s output which clearly indicates that high number of prefetches cause RA points and high queueing delays (of individual segments) ultimately leading to high queueing delays (of the entire query). We confirmed from engineers that this is accurate and it was interesting to find that even senior engineers could not interpret the meaning of RA points without bringing in an expert who develops code around RA points, showing the limitations of domain expertise of engineers over large scale systems.

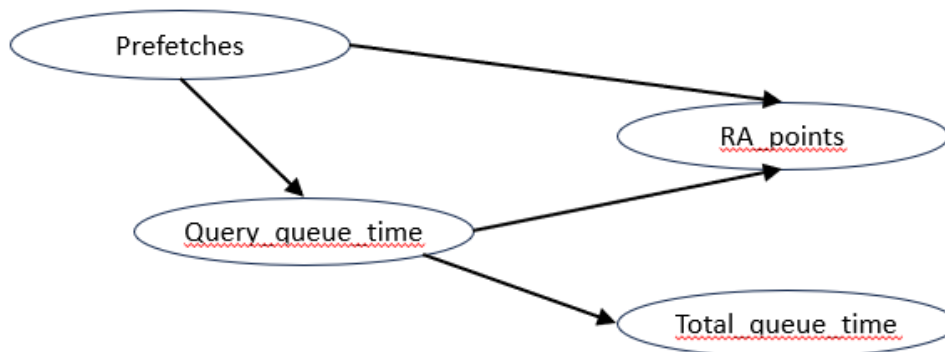


Figure 6.3: Output of PERFRCA for an overload in system due to high number of prefetches from S3.

CHAPTER 7

Conclusion

Cloud computing services have become mission-critical to many aspects of our life from health-care to travel to governance and so on. Keeping these services reliable is non-trivial and is no longer viable to be maintained by human resources at their pace of growth. In this thesis, I propose ML and NLP based techniques combined with rich debugging insights from my experience working with planetary-scale cloud services to leverage the vast amounts of available historical debugging data to help engineers in debugging production incidents. Further, I propose a ‘Coarsening’ framework that enables cloud providers to take scalable centralized debugging initiatives to get rid of the problems caused by inefficient, redundant and time-consuming distributed debugging workflows.

I discuss tools that I built as a part of this framework motivated by the learnings from real debugging workflows at major cloud providers. REVELIO produces debugging queries for engineers whose results can aid in validating a root cause hypothesis. REVELIO leverages ML models to deal with heterogeneity in data (e.g., unstructured user reports, quantitative system logs) and uses stable rank ordering based features to generalize to new and unseen fault locations better than other approaches. I developed a distributed systems debugging testbed for evaluating REVELIO and the fault injector in the testbed is motivated from a study of incidents and debugging workflows at *Anon1*. AUTOARTS labels an incident postmortem report with all contributing factors from a hierarchical and comprehensive ARTS root cause taxonomy. AUTOARTS also extracts key context for each contributing factor from a lengthy postmortem report to assist engineers in reasoning the choices for a root

cause label. The manual and empirical analyses conducted at Microsoft Azure resulted in many interesting findings about production incidents. PERFRCA enables causal discovery at-scale between telemetry in large-scale distributed systems to assist engineers in reasoning the cause of performance degradations. My work on coarse log summarization over real events captured in log from the datacenter operating system of a large cloud *Anon* achieved promising preliminary results to identify the chains of events that explain the root cause of an incident.

I envision that today's clouds will be ever more reliable by adopting a centralized cloud-scale debugging plane that enables: transfer of rich debugging insights across teams in a cloud, standardization of logging, tracing and telemetry collection frameworks across services to ease reasoning of and correlating logs, devise strategic retention plans over logs to build useful ML tools and make learning more tractable with the 'Coarsening' framework as a bootstrapping strategy. In this thesis, I presented evaluations conducted over real production incidents to show the preliminary proof-of-concept in using the 'Coarsening' framework and the presented ML, NLP techniques used in building tools. Realizing this vision at-scale takes a long time and support from cloud administrations to facilitate cross-team efforts. But, it is absolutely crucial to maintaining the reliability and availability targets of tomorrow's clouds.

7.1 Future Work and Open Problems

I am excited by the recent advancements in the field of Large Language Models (LLMs) that can increasingly make sense of unstructured natural language data, code scripts and synthesize both unstructured (e.g., hold conversations [TAB24, WHL23]) and structured outputs (e.g., code suggestions [WHL23, NN22]). There are however open problems that still need solutions as I mention below to enable functional training/finetuning of these models which can only be enabled by a cloud-scale centralized debugging effort. Significant

cross-team efforts are needed to bootstrap this transition to deploy debugging tools that can function at-scale.

7.1.1 Evaluation on Production Systems

My experience with data-driven debugging in the three largest clouds has been very frustrating. We found that incident routing was mostly done manually with long meetings between teams, root cause analysis was ad hoc, and retrospective analysis was primitive. When we attempted data driven debugging, we found data was retained for only a few months (or weeks for different types of data) and the true positives were small (high sparsity). Further data across teams was unavailable (e.g., different retention policies, gaps in tracing correlation IDs), making it hard to debug incidents that spanned teams. Which is what led us to a Centralized Debugging Plane that balances modularity needed for rapid development with the holistic view needed for debugging via ‘Coarsening’. While we have proposed ‘Coarsening’ as a framework and have preliminary results, a large-scale evaluation of these approaches on real systems is a mammoth-undertaking.

It requires coordination between teams to get access to necessary data, process data and potentially waiting for months to collect sufficient amount of data in the format that is needed. While the techniques proposed in this thesis generalize well to different debugging tools and systems, implementing them to work with different services is non-trivial and hence a major roadblock to deploying and evaluating the performance of debugging tools across clouds and services. We hope that a CDP lens can enable cloud providers to design consumable, general APIs that are designed with cloud-scale debugging as a key consumer can help ease this process. Such initiatives can enable engineers to identify alerting techniques, identify key vulnerabilities across the cloud and experiment with different logging and tracing frameworks. Simply put, a centralized cloud-scale lens to debugging is necessary to get rid of blind spots due to distributed debugging workflows and scalably improve reliability.

7.1.2 Impact of LLMs on Debugging

Designing a good developer-assistant interface that seamlessly handles expressive input options from the developer and provides timely responses is crucial to debugging assistance. Keeping the developer in the loop is critical because the developer represents years of domain expertise that is complementary to the data-driven NLP approach. However, determining the right interface is challenging as developer time and inputs are scarce resources. Repeatedly asking the developer for inputs negates automation benefits, but judicious developer inputs (e.g., a hint that the bug might reside in a particular subsystem) could significantly improve the assistant's output. It is also important to determine what developer inputs to request. Further, since developer time is precious, the developer should be able to smoothly trade-off prediction accuracy or granularity for lower prediction time.

All these issues present an opportunity for using LLMs based on recent advances in the ML and NLP domains. LLMs like ChatGPT [WHL23] from OpenAI, Gemini [TAB24] from Google are capable of human-like interaction with users. A major challenge in this thesis was to deal with heterogeneity and unstructured data sources in providing debugging assistance. We took the approach of designing data structures and frameworks that can enable and leverage ML and NLP techniques, but it's not always trivial to deploy such techniques in production systems because of lack of the exact type of data needed for a specific tool. LLMs on the other hand are easy to deploy because of lack of constraints on the structure of the data they can ingest. The principal drawback of using LLMs, however, is that the data sources they are trained on might not be very amenable to be used on debugging related information, atleast not with a high quality. We attempted this at Microsoft Azure and found that it is difficult to finetune LLMs to a functional-level beyond simple tasks such as summarization, search, etc. Cloud providers can invest resources into finetuning LLMs on debugging related tasks to better leverage their historical debugging data and improve interactive debugging experience with developers.

REFERENCES

- [ABI18] Nipun Arora, Jonathan Bell, Franjo Ivančić, Gail Kaiser, and Baishakhi Ray. “Replay without Recording of Production Bugs for Service Oriented Applications.” In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018*, p. 452–463, New York, NY, USA, 2018. Association for Computing Machinery.
- [ADS91] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. “Dynamic Slicing in the Presence of Unconstrained Pointers.” In *Proceedings of the Symposium on Testing, Analysis, and Verification, TAV4*. ACM, 1991.
- [AKS18] Firas Abuzaid, Peter Kraft, Sahaana Suri, Edward Gan, Eric Xu, Atul Shenoy, Asvin Ananthanarayan, John Sheu, Erik Meijer, Xi Wu, and et al. “DIFF: A Relational Interface for Large-Scale Data Explanation.” *Proc. VLDB Endow.*, **12**(4):419–432, December 2018.
- [Ama22a] Amazon.com. “AWS Outposts Family.” <https://aws.amazon.com/outposts/>, 2022.
- [Ama22b] Amazon.com. “What is a datalake?” <https://aws.amazon.com/big-data/datalakes-and-analytics/what-is-a-data-lake/>, 2022.
- [ANC00] Bowen Alpern, Ton Ngo, Jong-Deok Choi, and Manu Sridharan. “DejaVu: Deterministic Java Replay Debugger for Jalapeño Java Virtual Machine.” In *Proceedings of OOPSLA*. ACM, 2000.
- [App21] Appdynamics.com. “ADQL Reference.” <https://learn.appdynamics.com/courses/appdynamics-query-language-aly310>, 2021.
- [AWS23] AWS. “Application and Infrastructure Monitoring - AWS CloudWatch.” <https://aws.amazon.com/cloudwatch/>, 2023.
- [BDG14a] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. “P4: Programming protocol-independent packet processors.” *ACM SIGCOMM Computer Communication Review*, **44**(3):87–95, 2014.
- [BDG14b] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. “P4: Programming Protocol-Independent Packet Processors.” *SIGCOMM CCR*, July 2014.
- [BDV18] Rüdiger Birkner, Dana Drachler-Cohen, Laurent Vanbever, and Martin T. Vechev. “Net2Text: Query-Guided Summarization of Network Forwarding Behaviors.” In

- 15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018*, pp. 609–623, 2018.
- [BHS07] Gökhan Bakır, Thomas Hofmann, Bernhard Schölkopf, Alexander J Smola, and Ben Taskar. *Predicting structured data*. MIT press, 2007.
- [Bir06] Steven Bird. “NLTK: The Natural Language Toolkit.” In *Proceedings of the COLING/ACL on Interactive Presentation Sessions*, COLING-ACL ’06, p. 69–72, USA, 2006. Association for Computational Linguistics.
- [BKM18] Ranjita Bhagwan, Rahul Kumar, Chandra Maddila, and Adithya Abraham Philip. “Orca: Differential Bug Localization in Large-Scale Services.” In *13th USENIX Symposium on Operating Systems Design and Implementation*. USENIX, October 2018. Won the Jay Lepreau Best Paper Award.
- [Bot10] Léon Bottou. “Large-scale machine learning with stochastic gradient descent.” In *Proceedings of COMPSTAT’2010*, pp. 177–186. Springer, 2010.
- [BRA20] Chetan Bansal, Sundararajan Renganathan, Ashima Asudani, Olivier Midy, and Mathru Janakiraman. “DeCaf: Diagnosing and Triaging Performance Issues in Large-Scale Cloud Services.” In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP ’20, p. 201–210, New York, NY, USA, 2020. Association for Computing Machinery.
- [Bri21] University of British Columbia. “Cloud computing support accelerates COVID-19 vaccine improvements.” <https://www.asbmb.org/asbmb-today/science/051521/cloud-computing-support-accelerates-covid-19-vaccci>, 2021.
- [cas16] “Apache Cassandra.” <http://cassandra.apache.org/>, 2016.
- [CC20] Charisma Chan and Beth Cooper. “Debugging Incidents in Google’s Distributed Systems.” *Commun. ACM*, **63**(10):40–46, sep 2020.
- [CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler. “KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs.” In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI’08, pp. 209–224, Berkeley, CA, USA, 2008. USENIX Association.
- [CDJ19] Haicheng Chen, Wensheng Dou, Yanyan Jiang, and Feng Qin. “Understanding exception-related bugs in large-scale cloud systems.” In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 339–351. IEEE, 2019.

- [CGC14] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. “Empirical evaluation of gated recurrent neural networks on sequence modeling.” *arXiv preprint arXiv:1412.3555*, 2014.
- [CGL09] Yanpei Chen, Rean Griffith, Junda Liu, Randy H. Katz, and Anthony D. Joseph. “Understanding TCP Incast Throughput Collapse in Datacenter Networks.” In *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking*, WREN ’09, pp. 73–82. ACM, 2009.
- [CK12] Peter Cowhey and Michael Kleeman. *Unlocking the Benefits of Cloud Computing for Emerging Economies: A Policy Overview*. University of California San Diego, 2012.
- [CKL20] Zhuangbin Chen, Yu Kang, Liqun Li, Xu Zhang, Hongyu Zhang, Hui Xu, Yangfan Zhou, Li Yang, Jeffrey Sun, Zhangwei Xu, et al. “Towards intelligent incident management: why we need it and how we make it.” In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1487–1497, 2020.
- [CLP14] Xin Chen, Charng-Da Lu, and Karthik Pattabiraman. “Failure analysis of jobs in compute clouds: A google cluster case study.” In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pp. 167–177. IEEE, 2014.
- [CMF14] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F. Wenisch. “The Mystery Machine: End-to-End Performance Analysis of Large-Scale Internet Services.” In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI’14, p. 217–231, USA, 2014. USENIX Association.
- [Cou05] Patrick Cousot. “MIT Course 6.339: Abstract Interpretation.” <https://web.mit.edu/afs/athena.mit.edu/course/6/6.339/www/>, 2005.
- [Dat22a] Databricks.com. “Introduction to Datalakes.” <https://databricks.com/discover/data-lakes/introduction>, 2022.
- [Dat22b] Datadoghq.com. “Cloud Monitoring as a Service | Datadog.” <https://www.datadoghq.com/>, 2022.
- [DBJ23] Pradeep Dogga, Chetan Bansal, Gopinath Jayagopal, Richie Costleigh, Suman Nath, and Xuchao Zhang. “AutoARTS: Insights and Tools for Rootcausing Incidents in Microsoft Azure.” *In submission for review at USENIX ATC’23*, 2023.
- [DCL18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.”, 2018.

- [Dcr22] Neil Dcruz. “How Cloud Computing helped accelerate the COVID-19 vaccination.” <https://www.mygreatlearning.com/blog/how-cloud-computing-helped-accelerate-the-covid-19-vaccination/>, 2022.
- [DGH16] Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Sailesh R, and Subhajit Roy. “Program Synthesis Using Natural Language.” In *Proceedings of the 38th International Conference on Software Engineering*, ICSE. ACM, 2016.
- [DHH18] DHH. “Postmortem on the read-only outage of Basecamp on November 9th, 2018.” <https://bit.ly/2S9pq0t>, 2018.
- [DKI14] Catello Di Martino, Zbigniew Kalbarczyk, Ravishankar K Iyer, Geetika Goel, Santonu Sarkar, and Rajeshwari Ganesan. “Characterization of operational failures from a business data processing saas platform.” In *Companion Proceedings of the 36th International Conference on Software Engineering*, pp. 195–204, 2014.
- [DNS19a] Pradeep Dogga, Karthik Narasimhan, Anirudh Sivaraman, and Ravi Netravali. “A System-Wide Debugging Assistant Powered by Natural Language Processing.” In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC ’19, p. 171–177, New York, NY, USA, 2019. Association for Computing Machinery.
- [DNS19b] Pradeep Dogga, Karthik Narasimhan, Anirudh Sivaraman, and Ravi Netravali. “A system-wide debugging assistant powered by natural language processing.” In *Proceedings of the ACM Symposium on Cloud Computing*, pp. 171–177, 2019.
- [DNS22] Pradeep Dogga, Karthik Narasimhan, Anirudh Sivaraman, Shiv Saini, George Varghese, and Ravi Netravali. “Revelio: ML-Generated Debugging Queries for Finding Root Causes in Distributed Systems.” *Proceedings of Machine Learning and Systems*, 4:601–622, 2022.
- [Doc19] Docker.com. “Enterprise Container Platform | Docker.” <https://docker.com/>, 2019.
- [Don17] Brian Donohue. “Instapaper Outage Cause & Recovery.” <https://medium.com/making-instapaper/instapaper-outage-cause-recovery-3c32a7e9cc5f>, 2017.
- [Dor15] Dormando. “Memcached-a distributed memory object caching system.” <https://memcached.org/>, 2015.
- [Dro19] Dropbox. “Kelsey Fix shares the story behind Dropbox’s largest outage ever.” <https://bit.ly/2S6p8az>, 2019.
- [Env21] Envoyproxy.io. “envoy: an open source edge and service proxy, designed for cloud-native applications.” <https://www.envoyproxy.io/>, 2021.

- [Ern17] Michael D. Ernst. “Natural Language is a Programming Language: Applying Natural Language Processing to Software Development.” In *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA*, pp. 4:1–4:14, 2017.
- [Far12] Christina Farr. “The Cloud is Robin Hood: it is bridging the gap between rich and poor.” <https://venturebeat.com/business/the-cloud-is-robin-hood-it-is-bridging-the-gap-between-rich-and-poor/>, 2012.
- [FB88] Stuart I. Feldman and Channing B. Brown. “IGOR: A System for Program Debugging via Reversible Execution.” In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, PADD. ACM, 1988.
- [FLM20] Chongrong Fang, Haoyu Liu, Mao Miao, Jie Ye, Lei Wang, Wansheng Zhang, Daxiang Kang, Biao Lyv, Peng Cheng, and Jiming Chen. “VTrace: Automatic Diagnostic System for Persistent Packet Loss in Cloud-Scale Overlay Network.” In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM ’20, p. 31–43, New York, NY, USA, 2020. Association for Computing Machinery.
- [FPK07] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. “X-trace: A Pervasive Network Tracing Framework.” In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, NSDI. USENIX Association, 2007.
- [GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.
- [GBF99] Tibor Gyimóthy, Árpád Beszédés, and Istán Forgács. “An Efficient Relevant Slicing Method for Debugging.” In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-7, London, UK, UK, 1999. Springer-Verlag.
- [GDQ18] Yu Gao, Wensheng Dou, Feng Qin, Chushu Gao, Dong Wang, Jun Wei, Ruirui Huang, Li Zhou, and Yongming Wu. “An empirical study on crash recovery bugs in large-scale distributed systems.” In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 539–550, 2018.
- [GDR18] Sebastian Gehrmann, Yuntian Deng, and Alexander M Rush. “Bottom-up abstractive summarization.” *arXiv preprint arXiv:1808.10792*, 2018.

- [GGE16] Alberto Goffi, Alessandra Gorla, Michael D. Ernst, and Mauro Pezzè. “Automatic Generation of Oracles for Exceptional Behaviors.” In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016*. ACM, 2016.
- [GH10] Michael Gutmann and Aapo Hyvärinen. “Noise-contrastive estimation: A new estimation principle for unnormalized statistical models.” In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pp. 297–304, 2010.
- [GHL14] Haryadi S Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patananake, Thanh Do, Jeffrey Adityatama, Kurnia J Eliazar, Agung Laksono, Jeffrey F Lukman, Vincentius Martin, et al. “What bugs live in the cloud? a study of 3000+ issues in cloud systems.” In *Proceedings of the ACM symposium on cloud computing*, pp. 1–14, 2014.
- [GHS16] Haryadi S Gunawi, Mingzhe Hao, Riza O Suminto, Agung Laksono, Anang D Satria, Jeffrey Adityatama, and Kurnia J Eliazar. “Why does the cloud stop computing? lessons from hundreds of service outages.” In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pp. 1–16, 2016.
- [Git21] Github.io. “MaxiNet: Distributed Emulation of Software-Defined Networks.” <https://maxinet.github.io/>, 2021.
- [GLD21] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. “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, ASPLOS ’21*, p. 135–151, New York, NY, USA, 2021. Association for Computing Machinery.
- [GMK16] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. “Evolve or Die: High-Availability Design Principles Drawn from Googles Network Infrastructure.” In *Proceedings of the 2016 ACM SIGCOMM Conference, SIGCOMM*. ACM, 2016.
- [GNU22] GNU.org. “GDB: The GNU Project Debugger.” <https://www.gnu.org/software/gdb/>, 2022.
- [Goo19] Google. “Online Boutique: Sample cloud-native application with 10 microservices showcasing Kubernetes, Istio, gRPC and OpenCensus.” <https://github.com/GoogleCloudPlatform/microservices-demo>, 2019.
- [Goo21] Google. “cAdvisor.” <https://github.com/google/cadvisor>, 2021.
- [Goo23] Google. “Google Cloud Service Health.” <https://status.cloud.google.com/summary>, 2023.

- [Gra22] Grafana.com. “Grafana Features | Grafana Labs.” <https://grafana.com/grafana/>, 2022.
- [Grp21] Grpc.io. “gRPC: A high performance, open-source universal RPC framework.” <https://grpc.io/>, 2021.
- [GSB22] Supriyo GHOSH, Manish Shetty, Chetan Bansal, and Suman Nath. “How to Fight Production Incidents? An Empirical Study on a Large-scale Cloud Service.” In *SoCC 2022*. ACM, November 2022.
- [GYM20] Jiaqi Gao, Nofel Yaseen, Robert MacDavid, Felipe Vieira Frujeri, Vincent Liu, Ricardo Bianchini, Ramaswamy Aditya, Xiaohang Wang, Henry Lee , Dave Maltz, Minlan Yu , and Behnaz Arzani. “Scouts: Improving the Diagnosis Process Through Domain-customized Incident Routing.” In *SIGCOMM*. ACM, August 2020.
- [GZZ16] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. “Deep API learning.” In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 631–642. ACM, 2016.
- [hap19] “HAProxy: The Reliable, High Performance TCP/HTTP Load Balancer.” <http://www.haproxy.org/>, 2019.
- [HCE15] Irfan Ul Haq, Juan Caballero, and Michael D. Ernst. “Ayudante: Identifying Undesired Variable Interactions.” In *Proceedings of the 13th International Workshop on Dynamic Analysis*, WODA 2015. ACM, 2015.
- [HHJ14] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. “I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks.” In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI. USENIX Association, 2014.
- [HS97] Sepp Hochreiter and Jürgen Schmidhuber. “Long short-term memory.” *Neural computation*, **9**(8):1735–1780, 1997.
- [Ici21] Icinga.com. “Inspect your Entire Infrastructure.” <https://icinga.com/>, 2021.
- [ICM22] Azam Ikram, Sarthak Chakraborty, Subrata Mitra, Shiv Saini, Saurabh Bagchi, and Murat Kocaoglu. “Root Cause Analysis of Failures in Microservices through Causal Discovery.” In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pp. 31158–31170. Curran Associates, Inc., 2022.
- [Inc19] MongoDB Inc. “mongoDB: The database for modern applications.” <https://www.mongodb.com/>, 2019.

- [Jac21] Joab Jackson. “Debugging Microservices: Lessons from Google, Facebook, Lyft.” <https://bit.ly/2tBS9By>, 2021.
- [Jae21] Jaegertracing.io. “Jaeger.” <https://www.jaegertracing.io/docs/1.26/>, 2021.
- [JLC20] 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. *How to Mitigate the Incident? An Effective Troubleshooting Guide Recommendation Technique for Online Service Systems*, p. 1410–1420. Association for Computing Machinery, New York, NY, USA, 2020.
- [KB14] Diederik P Kingma and Jimmy Ba. “Adam: A method for stochastic optimization.” *arXiv preprint arXiv:1412.6980*, 2014.
- [KL88] Bogdan Korel and Janusz Laski. “Dynamic program slicing.” *Information processing letters*, **29**(3):155–163, 1988.
- [KM08] Andrew J. Ko and Brad A. Myers. “Debugging Reinvented: Asking and Answering Why and Why Not Questions About Program Behavior.” In *Proceedings of the 30th International Conference on Software Engineering, ICSE*. ACM, 2008.
- [KMB17] Jonathan Kaldor, Jonathan Mace, Michal Bejda, Edison Gao, Wiktor Kuropatwa, Joe O’Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Venkataraman, Kaushik Veeraraghavan, and Yee Jiun Song. “Canopy: An End-to-End Performance Tracing And Analysis System.” In *Proceedings of the 26th Symposium on Operating Systems Principles, SOSP*. ACM, 2017.
- [KTK13] Dongsun Kim, Yida Tao, Sunghun Kim, and Andreas Zeller. “Where should we fix this bug? a two-phase recommendation model.” *IEEE transactions on software Engineering*, **39**(11):1597–1610, 2013.
- [KW16] Thomas N Kipf and Max Welling. “Semi-supervised classification with graph convolutional networks.” *arXiv preprint arXiv:1609.02907*, 2016.
- [Lan21] Heather Landi. “Google Cloud launches vaccine management tools as tech giants jump into distribution efforts.” <https://www.fiercehealthcare.com/tech/google-cloud-rolls-out-tools-for-vaccine-logistics-as-tech-giants-jump-into-distribution>, 2021.
- [LCL21] Guangpu Li, Dongjie Chen, Shan Lu, Madanlal Musuvathi, and Suman Nath. “SherLock: Unsupervised Synchronization-Operation Inference.” In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2021*, p. 314–328, New York, NY, USA, 2021. Association for Computing Machinery.

- [LDB18] Annie Louis, Santanu Kumar Dash, Earl T. Barr, and Charles A. Sutton. “Deep Learning to Detect Redundant Method Comments.” *CoRR*, **abs/1806.04616**, 2018.
- [LGN08] Adrian Lienhard, Tudor Gîrba, and Oscar Nierstrasz. “Practical Object-Oriented Back-in-Time Debugging.” In *Proceedings of the 22Nd European Conference on Object-Oriented Programming*, ECOOP, Berlin, Heidelberg, 2008.
- [LHM10] Bob Lantz, Brandon Heller, and Nick McKeown. “A Network in a Laptop: Rapid Prototyping for Software-defined Networks.” In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX. ACM, 2010.
- [Lig21] Lightstep.com. “Simple Observability for Deep Systems.” <https://lightstep.com/>, 2021.
- [Lig22] Lighthouse. “Lighthouse | Tools for Web Developers | Google Developers.” <https://developers.google.com/web/tools/lighthouse>, 2022.
- [Lin04] Chin-Yew Lin. “Rouge: A package for automatic evaluation of summaries.” In *Text summarization branches out*, pp. 74–81, 2004.
- [LLL16] Tanakorn Leesatapornwongsa, Jeffrey F Lukman, Shan Lu, and Haryadi S Gunawi. “TaxDC: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems.” In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 517–530, 2016.
- [LLM19a] Guangpu Li, Shan Lu, Madanlal Musuvathi, Suman Nath, and Rohan Padhye. “Efficient Scalable Thread-Safety-Violation Detection: Finding Thousands of Concurrency Bugs during Testing.” In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP ’19, p. 162–180, New York, NY, USA, 2019. Association for Computing Machinery.
- [LLM19b] H. Liu, S. Lu, M. Musuvathi, and S. Nath. “What bugs cause production cloud incidents?” In *Proceedings of the Workshop on Hot Topics in Operating Systems*, May 2019.
- [LLM19c] Haopeng Liu, Shan Lu, Madan Musuvathi, and Suman Nath. “What Bugs Cause Production Cloud Incidents?” In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS ’19, pp. 155–162, New York, NY, USA, 2019. Association for Computing Machinery.
- [LLM19d] Haopeng Liu, Shan Lu, Madan Musuvathi, and Suman Nath. “What bugs cause production cloud incidents?” In *Workshop on Hot Topics in Operating Systems (HotOS)*, May 2019.

- [LLQ05] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. “Bugbench: Benchmarks for evaluating bug detection tools.” In *In Workshop on the Evaluation of Software Defect Detection Tools*, 2005.
- [LND16] Nicholas Locascio, Karthik Narasimhan, Eduardo DeLeon, Nate Kushman, and Regina Barzilay. “Neural Generation of Regular Expressions from Natural Language with Minimal Domain Knowledge.” In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, EMNLP 2016, Austin, Texas, USA, November 1-4, 2016*, 2016.
- [LNF12] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. “GenProg: A Generic Method for Automatic Software Repair.” *IEEE Trans. Softw. Eng.*, **38**(1):54–72, January 2012.
- [LPS08] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. “Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics.” *SIGOPS Oper. Syst. Rev.*, **42**(2):329–339, March 2008.
- [LWZ18] Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D. Ernst. “NL2Bash: A Corpus and Semantic Parser for Natural Language Interface to the Linux Operating System.” In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation, LREC 2018, Miyazaki, Japan, May 7-12, 2018.*, 2018.
- [Mac22] Duncan MacRae. “81% of companies had a cloud security incident in the last year.” <https://www.cloudcomputing-news.net/news/2022/oct/03/81-of-companies-had-a-cloud-security-incident-in-the-last-year/>, 2022.
- [MBK20] Sonu Mehta, Ranjita Bhagwan, Rahul Kumar, Chetan Bansal, Chandra Maddila, B. Ashok, Sumit Asthana, Christian Bird, and Aditya Kumar. “Rex: Preventing Bugs and Misconfiguration in Large Services Using Correlated Change Analysis.” In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pp. 435–448, Santa Clara, CA, February 2020. USENIX Association.
- [MEH10] James Mickens, Jeremy Elson, and Jon Howell. “Mugshot: Deterministic Capture and Replay for Javascript Applications.” In *Proceedings of NSDI*, 2010.
- [MF18] Jonathan Mace and Rodrigo Fonseca. “Universal Context Propagation for Distributed System Instrumentation.” In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys. ACM, 2018.
- [Mic22] Microsoft.com. “Azure Stack.” <https://azure.microsoft.com/en-us/overview/azure-stack/>, 2022.
- [Mil19] Derek Miller. “Leveraging BERT for Extractive Text Summarization on Lectures.” *CoRR*, **abs/1906.04165**, 2019.

- [MRF15] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. “Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems.” In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP*. ACM, 2015.
- [MSC13] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. “Distributed representations of words and phrases and their compositionality.” In *Advances in neural information processing systems*, pp. 3111–3119, 2013.
- [MSP18] Mateusz Machalica, Alex Samykin, Meredith Porth, and Satish Chandra. “Predictive Test Selection.” *CoRR*, **abs/1810.05286**, 2018.
- [MYG16] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. “Trumpet: Timely and Precise Triggers in Data Centers.” In *Proceedings of the 2016 ACM SIGCOMM Conference, SIGCOMM ’16*, p. 129–143, New York, NY, USA, 2016. Association for Computing Machinery.
- [Mys21] Mysql.com. “MySQL.” <https://www.mysql.com/>, 2021.
- [MYZ20] Minghua Ma, Zheng Yin, Shenglin Zhang, Sheng Wang, Christopher Zheng, Xinhao Jiang, Hanwen Hu, Cheng Luo, Yilin Li, Nengjun Qiu, Feifei Li, Changcheng Chen, and Dan Pei. “Diagnosing root causes of intermittent slow queries in cloud databases.” *Proc. VLDB Endow.*, **13**(8):1176–1189, apr 2020.
- [NAR16] Srinivas Narayana, Mina Tashmasbi Arashloo, Jennifer Rexford, and David Walker. “Compiling Path Queries.” In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation, NSDI*. USENIX Association, 2016.
- [New22] Newrelic.com. “New Relic | Deliver more perfect software.” <https://newrelic.com/>, 2022.
- [NGM16] Ravi Netravali, Ameesh Goyal, James Mickens, and Hari Balakrishnan. “Polaris: Faster Page Loads Using Fine-grained Dependency Tracking.” In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation, NSDI*. USENIX Association, 2016.
- [NM19] Ravi Netravali and James Mickens. “Reverb: Speculative Debugging for Web Applications.” In *Proceedings of the ACM Symposium on Cloud Computing, SoCC ’19*. ACM, 2019.
- [NN22] Nhan Nguyen and Sarah Nadi. “An empirical evaluation of GitHub copilot’s code suggestions.” In *Proceedings of the 19th International Conference on Mining Software Repositories, MSR ’22*, p. 1–5, New York, NY, USA, 2022. Association for Computing Machinery.

- [NPK13] Jaechang Nam, Sinno Jialin Pan, and Sunghun Kim. “Transfer defect learning.” In *2013 35th International Conference on Software Engineering (ICSE)*, pp. 382–391. IEEE, 2013.
- [NSN17] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. “Language-Directed Hardware Design for Network Performance Monitoring.” In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM*. ACM, 2017.
- [NZG16] Ramesh Nallapati, Bowen Zhou, Caglar Gulcehre, Bing Xiang, et al. “Abstractive text summarization using sequence-to-sequence rnns and beyond.” *arXiv preprint arXiv:1602.06023*, 2016.
- [Ope22a] Opentelemetry.io. “Automatic Instrumentation | OpenTelemetry.” <https://opentelemetry.io/docs/instrumentation/java/automatic/>, 2022.
- [Ope22b] Opentracing.io. “The OpenTracing project.” <https://opentracing.io/>, 2022.
- [Ora21] Orate. “Ticketmaster Traces 100 Million Transactions per Day with Jaeger.” <https://bit.ly/39rTn1N>, 2021.
- [Pin21] Pingdom.com. “Website Performance and Availability Monitoring | Pingdom.” <https://www.pingdom.com/>, 2021.
- [PJN13] Rahul Potharaju, Navendu Jain, and Cristina Nita-Rotaru. “Juggling the Jigsaw: Towards Automated Problem Inference from Network Trouble Tickets.” In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation, NSDI*. USENIX Association, 2013.
- [PKL09] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. “Automatically Patching Errors in Deployed Software.” In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles, SOSP*. ACM, 2009.
- [PKR16] M. Peuster, H. Karl, and S. van Rossem. “MeDICINE: Rapid prototyping of production-ready network services in multi-PoP environments.” In *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pp. 148–153, Nov 2016.
- [pos19] “PostgreSQL.” <https://www.postgresql.org/>, 2019.
- [PRW02] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. “Bleu: a Method for Automatic Evaluation of Machine Translation.” In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pp. 311–318,

Philadelphia, Pennsylvania, USA, July 2002. Association for Computational Linguistics.

- [PSM14] Jeffrey Pennington, Richard Socher, and Christopher Manning. “Glove: Global vectors for word representation.” In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pp. 1532–1543, 2014.
- [PWY18] Jayavardhan Peddamail, Zhen Wang, Ziyu Yao, and Huan Sun. “A Comprehensive Study of StaQC for Deep Code Summarization.” In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Lond, UK, August 2018*, 2018.
- [PXS17] Romain Paulus, Caiming Xiong, and Richard Socher. “A deep reinforced model for abstractive summarization.” *arXiv preprint arXiv:1705.04304*, 2017.
- [pyl19] “Pylons.” <http://www.pylonsproject.org/>, 2019.
- [QTS05] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. “Rx: Treating Bugs as Allergies—a Safe Method to Survive Software Failures.” In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles, SOSP '05*, pp. 235–248, New York, NY, USA, 2005. Association for Computing Machinery.
- [rab19] “Messaging that just works – RabbitMQ.” <https://www.rabbitmq.com/>, 2019.
- [RD00] Ehud Reiter and Robert Dale. *Building natural language generation systems*. Cambridge university press, 2000.
- [Red21] Redis.io. “Redis.” <https://redis.io/>, 2021.
- [red22] reddit.com. “reddit: the front page of the internet.” <https://reddit.com/>, 2022.
- [RSR20] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. “Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer.” *Journal of Machine Learning Research*, **21**(140):1–67, 2020.
- [Sal17] Saltside Engineering. “Our First Kubernetes Outage | Saltside Engineering.” <https://engineering.saltside.se/our-first-kubernetes-outage-c6b9249cfd3a>, 2017.
- [SBB10] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspán, and Chandan Shanbhag. “Dapper, a large-scale distributed systems tracing infrastructure.” Technical report, Technical report, Google, 2010.

- [SBK21] Manish Shetty, Chetan Bansal, Sumit Kumar, Nikitha Rao, Nachiappan Nagappan, and Thomas Zimmermann. “Neural knowledge extraction from cloud service incidents.” In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 218–227. IEEE, 2021.
- [SBN21] Manish Shetty, Chetan Bansal, Suman Nath, Sean Bowles, Henry Wang, Ozgur Arman, and Siamak Ahari. “Large-scale Crash Localization using Multi-Task Learning.” *arXiv preprint arXiv:2109.14326*, 2021.
- [Sec22] SecurityShelf. “Cloud Misconfig Exposes 3TB of Sensitive Airport Data in Amazon S3 Bucket: ‘Lives at Stake’.” <https://securityshelf.com/2022/07/06/cloud-misconfig-exposes-3tb-of-sensitive-airport-data-in-amazon-s3-bucket-lives-at-stake/>, 2022.
- [SH22] Amrita Saha and Steven CH Hoi. “Mining Root Cause Knowledge from Cloud Service Incident Investigations for AIOps.” *arXiv preprint arXiv:2204.11598*, 2022.
- [SKR21] Tian Shi, Yaser Keneshloo, Naren Ramakrishnan, and Chandan K Reddy. “Neural abstractive text summarization with sequence-to-sequence models.” *ACM Transactions on Data Science*, **2**(1):1–37, 2021.
- [SLM17] Abigail See, Peter J Liu, and Christopher D Manning. “Get to the point: Summarization with pointer-generator networks.” *arXiv preprint arXiv:1704.04368*, 2017.
- [SPB16] Colin Scott, Aurojit Panda, Vjekoslav Brajkovic, George Necula, Arvind Krishnamurthy, and Scott Shenker. “Minimizing Faulty Executions of Distributed Systems.” In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI. USENIX Association, 2016.
- [Spl21] Splunk.com. “SIEM, AIOps, Application Management, Log Management, Machine Learning, and Compliance | Splunk.” <https://www.splunk.com/>, 2021.
- [Sym14] Symantec. “Tracking Cookie.” https://www.symantec.com/security_response/writeup.jsp?docid=2006-080217-3524-99, July 22, 2014.
- [TAB24] Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M. Dai, Anja Hauth, Katie Millican, David Silver, Melvin Johnson, Ioannis Antonoglou, Julian Schrittwieser, Amelia Glaese, Jilin Chen, Emily Pitler, Timothy Lillicrap, Angeliki Lazaridou, Orhan Firat, James Molloy, Michael Isard, Paul R. Barham, Tom Hennigan, Benjamin Lee, Fabio Viola, Malcolm Reynolds, Yuanzhong Xu, Ryan Doherty, Eli Collins, Clemens Meyer, Eliza Rutherford, Erica Moreira, Kareem Ayoub, Megha Goel,

Jack Krawczyk, Cosmo Du, Ed Chi, Heng-Tze Cheng, Eric Ni, Purvi Shah, Patrick Kane, Betty Chan, Manaal Faruqui, Aliaksei Severyn, Hanzhao Lin, YaGuang Li, Yong Cheng, Abe Ittycheriah, Mahdis Mahdieh, Mia Chen, Pei Sun, Dustin Tran, Sumit Bagri, Balaji Lakshminarayanan, Jeremiah Liu, Andras Orban, Fabian Güra, Hao Zhou, Xinying Song, Aurelien Boffy, Harish Ganapathy, Steven Zheng, HyunJeong Choe, Ágoston Weisz, Tao Zhu, Yifeng Lu, Siddharth Gopal, Jarrod Kahn, Maciej Kula, Jeff Pitman, Rushin Shah, Emanuel Taropa, Majd Al Merey, Martin Baeuml, Zhifeng Chen, Laurent El Shafey, Yujing Zhang, Olcan Sercinoglu, George Tucker, Enrique Piqueras, Maxim Krikun, Iain Barr, Nikolay Savinov, Ivo Danihelka, Becca Roelofs, Anaïs White, Anders Andreassen, Tamara von Glehn, Lakshman Yagati, Mehran Kazemi, Lucas Gonzalez, Misha Khalman, Jakub Synchronowski, Alexandre Frechette, Charlotte Smith, Laura Culp, Lev Proleev, Yi Luan, Xi Chen, James Lottes, Nathan Schucher, Federico Lebron, Alban Rrustemi, Natalie Clay, Phil Crone, Tomas Kocisky, Jeffrey Zhao, Bartek Perz, Dian Yu, Heidi Howard, Adam Bloniarz, Jack W. Rae, Han Lu, Laurent Sifre, Marcello Maggioni, Fred Alcober, Dan Garrette, Megan Barnes, Shantanu Thakoor, Jacob Austin, Gabriel Barth-Maron, William Wong, Rishabh Joshi, Rahma Chaabouni, Deeni Fatiha, Arun Ahuja, Gaurav Singh Tomar, Evan Senter, Martin Chadwick, Ilya Kornakov, Nithya Attaluri, Iñaki Iturrate, Ruibo Liu, Yunxuan Li, Sarah Cogan, Jeremy Chen, Chao Jia, Chenjie Gu, Qiao Zhang, Jordan Grimstad, Ale Jakse Hartman, Xavier Garcia, Thanumalayan Sankaranarayana Pillai, Jacob Devlin, Michael Laskin, Diego de Las Casas, Dasha Valter, Connie Tao, Lorenzo Blanco, Adrià Puigdomènech Badia, David Reitter, Mianna Chen, Jenny Brennan, Clara Rivera, Sergey Brin, Shariq Iqbal, Gabriela Surita, Jane Labanowski, Abhi Rao, Stephanie Winkler, Emilio Parisotto, Yiming Gu, Kate Olszewska, Ravi Addanki, Antoine Miech, Annie Louis, Denis Teplyashin, Geoff Brown, Elliot Catt, Jan Balaguer, Jackie Xiang, Pidong Wang, Zoe Ashwood, Anton Briukhov, Albert Webson, Sanjay Ganapathy, Smit Sanghavi, Ajay Kannan, Ming-Wei Chang, Axel Stjerngren, Josip Djolonga, Yuting Sun, Ankur Bapna, Matthew Aitchison, Pedram Pejman, Henryk Michalewski, Tianhe Yu, Cindy Wang, Juliette Love, Junwhan Ahn, Dawn Bloxwich, Kehang Han, Peter Humphreys, Thibault Sellam, James Bradbury, Varun Godbole, Sina Samangooei, Bogdan Damoc, Alex Kaskasoli, Sébastien M. R. Arnold, Vijay Vasudevan, Shubham Agrawal, Jason Riesa, Dmitry Lepikhin, Richard Tanburn, Srivatsan Srinivasan, Hyeontaek Lim, Sarah Hodgkinson, Pranav Shyam, Johan Ferret, Steven Hand, Ankush Garg, Tom Le Paine, Jian Li, Yujia Li, Minh Giang, Alexander Neitz, Zaheer Abbas, Sarah York, Machel Reid, Elizabeth Cole, Aakanksha Chowdhery, Dipanjan Das, Dominika Rogozińska, Vitaliy Nikolaev, Pablo Sprechmann, Zachary Nado, Lukas Zilka, Flavien Prost, Luheng He, Marianne Monteiro, Gaurav Mishra, Chris Welty, Josh Newlan, Dawei Jia, Miltiadis Allamanis, Clara Huiyi Hu, Raoul de Liedekerke, Justin Gilmer, Carl Saroufim, Shruti Rijhwani, Shaobo Hou, Disha Shrivastava, Anirudh Baddepudi, Alex Goldin, Adnan Ozturel, Albin Cassirer, Yunhan Xu,

Daniel Sohn, Devendra Sachan, Reinald Kim Amplayo, Craig Swanson, Dessie Petrova, Shashi Narayan, Arthur Guez, Siddhartha Brahma, Jessica Landon, Miteyan Patel, Ruizhe Zhao, Kevin Villela, Luyu Wang, Wenhao Jia, Matthew Rahtz, Mai Giménez, Legg Yeung, James Keeling, Petko Georgiev, Diana Mincu, Boxi Wu, Salem Haykal, Rachel Saputro, Kiran Vodrahalli, James Qin, Zeynep Cankara, Abhanshu Sharma, Nick Fernando, Will Hawkins, Behnam Neyshabur, Solomon Kim, Adrian Hutter, Priyanka Agrawal, Alex Castro-Ros, George van den Driessche, Tao Wang, Fan Yang, Shuo yiin Chang, Paul Komarek, Ross McIlroy, Mario Lučić, Guodong Zhang, Wael Farhan, Michael Sharman, Paul Natsev, Paul Michel, Yamini Bansal, Siyuan Qiao, Kris Cao, Siamak Shakeri, Christina Butterfield, Justin Chung, Paul Kishan Rubenstein, Shivani Agrawal, Arthur Mensch, Kedar Soparkar, Karel Lenc, Timothy Chung, Aedan Pope, Loren Maggiore, Jackie Kay, Priya Jhakra, Shibo Wang, Joshua Maynez, Mary Phuong, Taylor Tobin, Andrea Tacchetti, Maja Trebacz, Kevin Robinson, Yash Katariya, Sebastian Riedel, Paige Bailey, Kefan Xiao, Nimesh Ghelani, Lora Aroyo, Ambrose Slone, Neil Houlsby, Xuehan Xiong, Zhen Yang, Elena Gribovskaya, Jonas Adler, Mateo Wirth, Lisa Lee, Music Li, Thais Kagohara, Jay Pavagadhi, Sophie Bridgers, Anna Bortsova, Sanjay Ghemawat, Zafarali Ahmed, Tianqi Liu, Richard Powell, Vijay Bolina, Mariko Iinuma, Polina Zablotskaia, James Besley, Da-Woon Chung, Timothy Dozat, Ramona Comanescu, Xiance Si, Jeremy Greer, Guolong Su, Martin Polacek, Raphaël Lopez Kaufman, Simon Tokumine, Hexiang Hu, Elena Buchatskaya, Yingjie Miao, Mohamed Elhawaty, Aditya Siddhant, Nenad Tomasev, Jinwei Xing, Christina Greer, Helen Miller, Shereen Ashraf, Aurko Roy, Zizhao Zhang, Ada Ma, Angelos Filos, Milos Besta, Rory Blevins, Ted Klimenko, Chih-Kuan Yeh, Soravit Changpinyo, Jiaqi Mu, Oscar Chang, Mantas Pajarskas, Carrie Muir, Vered Cohen, Charline Le Lan, Krishna Haridasan, Amit Marathe, Steven Hansen, Sholto Douglas, Rajkumar Samuel, Mingqiu Wang, Sophia Austin, Chang Lan, Jiepu Jiang, Justin Chiu, Jaime Alonso Lorenzo, Lars Lowe Sjö-sund, Sébastien Cevey, Zach Gleicher, Thi Avrahami, Anudhyan Boral, Hansa Srinivasan, Vittorio Selo, Rhys May, Konstantinos Aisopos, Léonard Hussenot, Livio Baldini Soares, Kate Baumli, Michael B. Chang, Adrià Recasens, Ben Caine, Alexander Pritzel, Filip Pavetic, Fabio Pardo, Anita Gergely, Justin Frye, Vinay Ramasesh, Dan Horgan, Kartikeya Badola, Nora Kassner, Subhrajit Roy, Ethan Dyer, Víctor Campos Campos, Alex Tomala, Yunhao Tang, Dalia El Badawy, El-speth White, Basil Mustafa, Oran Lang, Abhishek Jindal, Sharad Vikram, Zhitao Gong, Sergi Caelles, Ross Hemsley, Gregory Thornton, Fangxiaoyu Feng, Wojciech Stokowiec, Ce Zheng, Phoebe Thacker, Çağlar Ünlü, Zhishuai Zhang, Mohammad Saleh, James Svensson, Max Bileschi, Piyush Patil, Ankesh Anand, Roman Ring, Katerina Tsihlas, Arpi Vezer, Marco Selvi, Toby Shevlane, Mikel Rodriguez, Tom Kwiatkowski, Samira Daruki, Keran Rong, Allan Dafoe, Nicholas FitzGerald, Keren Gu-Lemberg, Mina Khan, Lisa Anne Hendricks, Marie Pellat, Vladimir Feinberg, James Cobon-Kerr, Tara Sainath, Maribeth Rauh, Sayed Hadi Hashemi,

Richard Ives, Yana Hasson, Eric Noland, Yuan Cao, Nathan Byrd, Le Hou, Qingze Wang, Thibault Sottiaux, Michela Paganini, Jean-Baptiste Lespiau, Alexandre Moufarek, Samer Hassan, Kaushik Shivakumar, Joost van Amersfoort, Amol Mandhane, Pratik Joshi, Anirudh Goyal, Matthew Tung, Andrew Brock, Hannah Sheahan, Vedant Misra, Cheng Li, Nemanja Rakićević, Mostafa Dehghani, Fangyu Liu, Sid Mittal, Junhyuk Oh, Seb Noury, Eren Sezener, Fantine Huot, Matthew Lamm, Nicola De Cao, Charlie Chen, Sidharth Mudgal, Romina Stella, Kevin Brooks, Gautam Vasudevan, Chenxi Liu, Mainak Chain, Nivedita Melinkeri, Aaron Cohen, Venus Wang, Kristie Seymore, Sergey Zubkov, Rahul Goel, Summer Yue, Sai Krishnakumaran, Brian Albert, Nate Hurley, Motoki Sano, Anhad Mohananey, Jonah Joughin, Egor Filonov, Tomasz Kępa, Yomna Eldawy, Jiawern Lim, Rahul Rishi, Shirin Badiehzadegan, Taylor Bos, Jerry Chang, Sanil Jain, Sri Gayatri Sundara Padmanabhan, Subha Puttagunta, Kalpesh Krishna, Leslie Baker, Norbert Kalb, Vamsi Bedapudi, Adam Kurzrok, Shuntong Lei, Anthony Yu, Oren Litvin, Xiang Zhou, Zhichun Wu, Sam Sobell, Andrea Siciliano, Alan Papir, Robby Neale, Jonas Bragagnolo, Tej Toor, Tina Chen, Valentin Anklin, Feiran Wang, Richie Feng, Milad Gholami, Kevin Ling, Lijuan Liu, Jules Walter, Hamid Moghaddam, Arun Kishore, Jakub Adamek, Tyler Mercado, Jonathan Mallinson, Siddhinita Wandekar, Stephen Cagle, Eran Ofek, Guillermo Garrido, Clemens Lombriser, Maksim Mukha, Botu Sun, Hafeezul Rahman Mohammad, Josip Matak, Yadi Qian, Vikas Peswani, Pawel Janus, Quan Yuan, Leif Schelin, Oana David, Ankur Garg, Yifan He, Oleksii Duzhyi, Anton Älgmyr, Timothée Lottaz, Qi Li, Vikas Yadav, Luyao Xu, Alex Chinien, Rakesh Shivanna, Aleksandr Chuklin, Josie Li, Carrie Spadine, Travis Wolfe, Kareem Mohamed, Subhabrata Das, Zihang Dai, Kyle He, Daniel von Dincklage, Shyam Upadhyay, Akanksha Maurya, Luyan Chi, Sebastian Krause, Khalid Salama, Pam G Rabinovitch, Pavan Kumar Reddy M, Aarush Selvan, Mikhail Dektiarev, Golnaz Ghiasi, Erdem Guven, Himanshu Gupta, Boyi Liu, Deepak Sharma, Idan Heimlich Shtacher, Shachi Paul, Oscar Akerlund, François-Xavier Aubet, Terry Huang, Chen Zhu, Eric Zhu, Elico Teixeira, Matthew Fritze, Francesco Bertolini, Liana-Eleonora Marinescu, Martin Bülle, Dominik Paulus, Khyatti Gupta, Tejasi Latkar, Max Chang, Jason Sanders, Roopa Wilson, Xuwei Wu, Yi-Xuan Tan, Lam Nguyen Thiet, Tulsee Doshi, Sid Lall, Swaroop Mishra, Wanming Chen, Thang Luong, Seth Benjamin, Jasmine Lee, Ewa Andrejczuk, Dominik Rabiej, Vipul Ranjan, Krzysztof Styrc, Pengcheng Yin, Jon Simon, Malcolm Rose Harriott, Mudit Bansal, Alexei Robsky, Geoff Bacon, David Greene, Daniil Mirylenka, Chen Zhou, Obaid Sarvana, Abhimanyu Goyal, Samuel Andermatt, Patrick Siegler, Ben Horn, Assaf Israel, Francesco Pongetti, Chih-Wei "Louis" Chen, Marco Selvatici, Pedro Silva, Kathie Wang, Jackson Tolins, Kelvin Guu, Roey Yoge, Xiaochen Cai, Alessandro Agostini, Maulik Shah, Hung Nguyen, Noah Ó Donnaile, Sébastien Pereira, Linda Friso, Adam Stambler, Adam Kurzrok, Chenkai Kuang, Yan Romanikhin, Mark Geller, ZJ Yan, Kane Jang, Cheng-Chun Lee, Wojciech Fica, Eric Malmi, Qijun Tan, Dan Banica, Daniel

Balle, Ryan Pham, Yanping Huang, Diana Avram, Hongzhi Shi, Jasjot Singh, Chris Hidey, Niharika Ahuja, Pranab Saxena, Dan Dooley, Srividya Pranavi Potharaju, Eileen O'Neill, Anand Gokulchandran, Ryan Foley, Kai Zhao, Mike Dusenberry, Yuan Liu, Pulkit Mehta, Ragha Kotikalapudi, Chalence Safranek-Shrader, Andrew Goodman, Joshua Kessinger, Eran Globen, Prateek Kolhar, Chris Gorgolewski, Ali Ibrahim, Yang Song, Ali Eichenbaum, Thomas Brovelli, Sahitya Potluri, Preethi Lahoti, Cip Baetu, Ali Ghorbani, Charles Chen, Andy Crawford, Shalini Pal, Mukund Sridhar, Petru Gurita, Asier Mujika, Igor Petrovski, Pierre-Louis Cedoz, Chenmei Li, Shiyuan Chen, Niccolò Dal Santo, Siddharth Goyal, Jitesh Punjabi, Karthik Kappaganthu, Chester Kwak, Pallavi LV, Sarmishta Velury, Himadri Choudhury, Jamie Hall, Premal Shah, Ricardo Figueira, Matt Thomas, Minjie Lu, Ting Zhou, Chintu Kumar, Thomas Jurdi, Sharat Chikkerur, Yenai Ma, Adams Yu, Soo Kwak, Victor Ähdel, Sujeevan Rajayogam, Travis Choma, Fei Liu, Aditya Barua, Colin Ji, Ji Ho Park, Vincent Hellendoorn, Alex Bailey, Taylan Bilal, Huanjie Zhou, Mehrdad Khatir, Charles Sutton, Wojciech Rządowski, Fiona Macintosh, Konstantin Shagin, Paul Medina, Chen Liang, Jinjing Zhou, Pararth Shah, Yingying Bi, Attila Dankovics, Shipra Banga, Sabine Lehmann, Marissa Bredesen, Zifan Lin, John Eric Hoffmann, Jonathan Lai, Raynald Chung, Kai Yang, Nihal Balani, Arthur Bražinskas, Andrei Sozanschi, Matthew Hayes, Héctor Fernández Alcalde, Peter Makarov, Will Chen, Antonio Stella, Liselotte Snijders, Michael Mandl, Ante Kärroman, Paweł Nowak, Xinyi Wu, Alex Dyck, Krishnan Vaidyanathan, Raghavender R, Jessica Mallet, Mitch Rudominer, Eric Johnston, Sushil Mittal, Akhil Udathu, Janara Christensen, Vishal Verma, Zach Irving, Andreas Santucci, Gamaleldin Elsayed, Elnaz Davoodi, Marin Georgiev, Ian Tenney, Nan Hua, Geoffrey Cideron, Edouard Leurent, Mahmoud Alnahlawi, Ionut Georgescu, Nan Wei, Ivy Zheng, Dylan Scandinaro, Heinrich Jiang, Jasper Snoek, Mukund Sundararajan, Xuezhi Wang, Zack Ontiveros, Itay Karo, Jeremy Cole, Vinu Rajashekhar, Lara Tumeh, Eyal Ben-David, Rishub Jain, Jonathan Uesato, Romina Datta, Oskar Bunyan, Shimu Wu, John Zhang, Piotr Stanczyk, Ye Zhang, David Steiner, Subhajit Naskar, Michael Azzam, Matthew Johnson, Adam Paszke, Chung-Cheng Chiu, Jaume Sanchez Elias, Afroz Mohiuddin, Faizan Muhammad, Jin Miao, Andrew Lee, Nino Vieillard, Jane Park, Jiageng Zhang, Jeff Stanway, Drew Garmon, Abhijit Karmarkar, Zhe Dong, Jong Lee, Aviral Kumar, Luowei Zhou, Jonathan Evens, William Isaac, Geoffrey Irving, Edward Loper, Michael Fink, Isha Arkatkar, Nanxin Chen, Izhak Shafran, Ivan Petrychenko, Zhe Chen, Johnson Jia, Anselm Levskaya, Zhenkai Zhu, Peter Grabowski, Yu Mao, Alberto Magni, Kaisheng Yao, Javier Snaider, Norman Casagrande, Evan Palmer, Paul Suganthan, Alfonso Castaño, Irene Giannoumis, Wooyeol Kim, Mikołaj Rybiński, Ashwin Sreevatsa, Jennifer Prendki, David Soergel, Adrian Goedeckemeyer, Willi Gierke, Mohsen Jafari, Meenu Gaba, Jeremy Wiesner, Diana Gage Wright, Yawen Wei, Harsha Vashisht, Yana Kulizhskaya, Jay Hoover, Maigo Le, Lu Li, Chimezie Iwuanyanwu, Lu Liu, Kevin Ramirez, Andrey Khorlin, Albert Cui, Tian

LIN, Marcus Wu, Ricardo Aguilar, Keith Pallo, Abhishek Chakladar, Ginger Perng, Elena Allica Abellan, Mingyang Zhang, Ishita Dasgupta, Nate Kushman, Ivo Penchev, Alena Repina, Xihui Wu, Tom van der Weide, Priya Ponnappalli, Caroline Kaplan, Jiri Simsa, Shuangfeng Li, Olivier Dousse, Fan Yang, Jeff Piper, Nathan Ie, Rama Pasumarthi, Nathan Lintz, Anitha Vijayakumar, Daniel Andor, Pedro Valenzuela, Minnie Lui, Cosmin Padurararu, Daiyi Peng, Katherine Lee, Shuyuan Zhang, Somer Greene, Duc Dung Nguyen, Paula Kurylowicz, Cassidy Hardin, Lucas Dixon, Lili Janzer, Kiam Choo, Ziqiang Feng, Biao Zhang, Achintya Singhal, Dayou Du, Dan McKinnon, Natasha Antropova, Tolga Bolukbasi, Orgad Keller, David Reid, Daniel Finchelstein, Maria Abi Raad, Remi Crocker, Peter Hawkins, Robert Dadashi, Colin Gaffney, Ken Franko, Anna Bulanova, Rémi Leblond, Shirley Chung, Harry Askham, Luis C. Cobo, Kelvin Xu, Felix Fischer, Jun Xu, Christina Sorokin, Chris Alberti, Chu-Cheng Lin, Colin Evans, Alek Dimitriev, Hannah Forbes, Dylan Banarse, Zora Tung, Mark Omernick, Colton Bishop, Rachel Sterneck, Rohan Jain, Jiawei Xia, Ehsan Amid, Francesco Piccinno, Xingyu Wang, Praseem Banzal, Daniel J. Mankowitz, Alex Polozov, Victoria Krakovna, Sasha Brown, MohammadHossein Bateni, Dennis Duan, Vlad Firoiu, Meghana Thotakuri, Tom Natan, Matthieu Geist, Ser tan Girgin, Hui Li, Jiayu Ye, Ofir Roval, Reiko Tojo, Michael Kwong, James Lee-Thorp, Christopher Yew, Danila Sinopalnikov, Sabela Ramos, John Mellor, Abhishek Sharma, Kathy Wu, David Miller, Nicolas Sonnerat, Denis Vnukov, Rory Greig, Jennifer Beattie, Emily Caveness, Libin Bai, Julian Eisenschlos, Alex Korchemniy, Tomy Tsai, Mimi Jasarevic, Weize Kong, Phuong Dao, Zeyu Zheng, Frederick Liu, Fan Yang, Rui Zhu, Tian Huey Teh, Jason Sanmiya, Evgeny Gladchenko, Nejc Trdin, Daniel Toyama, Evan Rosen, Sasan Tavakkol, Linting Xue, Chen Elkind, Oliver Woodman, John Carpenter, George Papamakarios, Rupert Kemp, Sushant Kalle, Tanya Grunina, Rishika Sinha, Alice Talbert, Diane Wu, Denese Owusu-Afriyie, Cosmo Du, Chloe Thornton, Jordi Pont-Tuset, Pradyumna Narayana, Jing Li, Saaber Fatehi, John Wieting, Omar Ajmeri, Benigno Uria, Yeongil Ko, Laura Knight, Amélie Héliou, Ning Niu, Shane Gu, Chenxi Pang, Yeqing Li, Nir Levine, Ariel Stolovich, Rebeca Santamaria-Fernandez, Sonam Goenka, Wenny Yustalim, Robin Strudel, Ali Elqursh, Charlie Deck, Hyo Lee, Zonglin Li, Kyle Levin, Raphael Hoffmann, Dan Holtmann-Rice, Olivier Bachem, Sho Arora, Christy Koh, Soheil Hassas Yeganeh, Siim Pöder, Mukarram Tariq, Yanhua Sun, Lucian Ionita, Mojtaba Seyedhosseini, Pouya Tafti, Zhiyu Liu, Anmol Gulati, Jasmine Liu, Xinyu Ye, Bart Chrzaszcz, Lily Wang, Nikhil Sethi, Tianrun Li, Ben Brown, Shreya Singh, Wei Fan, Aaron Parisi, Joe Stanton, Vinod Koverkathu, Christopher A. Choquette-Choo, Yunjie Li, TJ Lu, Abe Ittycheriah, Prakash Shroff, Mani Varadarajan, Sanaz Bahargam, Rob Willoughby, David Gaddy, Guillaume Desjardins, Marco Cornero, Brona Robenek, Bhavishya Mittal, Ben Albrecht, Ashish Shenoy, Fedor Moiseev, Henrik Jacobsson, Alireza Ghaffarkhah, Morgane Rivière, Alanna Walton, Clément Crepy, Alicia Parrish, Zongwei Zhou, Clement

Farabet, Carey Radebaugh, Praveen Srinivasan, Claudia van der Salm, Andreas Fidjeland, Salvatore Scellato, Eri Latorre-Chimoto, Hanna Klimczak-Plucińska, David Bridson, Dario de Cesare, Tom Hudson, Piermaria Mendolicchio, Lexi Walker, Alex Morris, Matthew Mauger, Alexey Guseynov, Alison Reid, Seth Odoom, Lucia Loher, Victor Cotruta, Madhavi Yenugula, Dominik Grewe, Anastasia Petrushkina, Tom Duerig, Antonio Sanchez, Steve Yadlowsky, Amy Shen, Amir Globerson, Lynette Webb, Sahil Dua, Dong Li, Surya Bhupatiraju, Dan Hurt, Haroon Qureshi, Ananth Agarwal, Tomer Shani, Matan Eyal, Anuj Khare, Shreyas Rammohan Belle, Lei Wang, Chetan Tekur, Mihir Sanjay Kale, Jinliang Wei, Ruoxin Sang, Brennan Saeta, Tyler Liechty, Yi Sun, Yao Zhao, Stephan Lee, Pandu Nayak, Doug Fritz, Manish Reddy Vuyyuru, John Aslanides, Nidhi Vyas, Martin Wicke, Xiao Ma, Evgenii Eltyshev, Nina Martin, Hardie Cate, James Manyika, Keyvan Amiri, Yelin Kim, Xi Xiong, Kai Kang, Florian Luisier, Nilesh Tripuraneni, David Madras, Mandy Guo, Austin Waters, Oliver Wang, Joshua Ainslie, Jason Baldridge, Han Zhang, Garima Pruthi, Jakob Bauer, Feng Yang, Riham Mansour, Jason Gelman, Yang Xu, George Polovets, Ji Liu, Honglong Cai, Warren Chen, XiangHai Sheng, Emily Xue, Sherjil Ozair, Christof Angermueller, Xiaowei Li, Anoop Sinha, Weiren Wang, Julia Wiesinger, Emmanouil Koukoumidis, Yuan Tian, Anand Iyer, Madhu Gurumurthy, Mark Goldenson, Parashar Shah, MK Blake, Hongkun Yu, Anthony Urbanowicz, Jennimaria Palomaki, Chrisantha Fernando, Ken Durden, Harsh Mehta, Nikola Momchev, Elahe Rahimtoroghi, Maria Georgaki, Amit Raul, Sebastian Ruder, Morgan Redshaw, Jinhyuk Lee, Denny Zhou, Komal Jalan, Dinghua Li, Blake Hechtman, Parker Schuh, Milad Nasr, Kieran Milan, Vladimir Mikulik, Juliana Franco, Tim Green, Nam Nguyen, Joe Kelley, Aroma Mahendru, Andrea Hu, Joshua Howland, Ben Vargas, Jeffrey Hui, Kshitij Bansal, Vikram Rao, Rakesh Ghiya, Emma Wang, Ke Ye, Jean Michel Sarr, Melanie Moranski Preston, Madeleine Elish, Steve Li, Aakash Kaku, Jigar Gupta, Ice Pasupat, Da-Cheng Juan, Milan Someswar, Tejvi M., Xinyun Chen, Aida Amini, Alex Fabrikant, Eric Chu, Xuanyi Dong, Amruta Muthal, Senaka Buttpitiya, Sarthak Jauhari, Nan Hua, Urvashi Khandelwal, Ayal Hitron, Jie Ren, Larissa Rinaldi, Shahar Drath, Avigail Dabush, Nan-Jiang Jiang, Harshal Godhia, Uli Sachs, Anthony Chen, Yicheng Fan, Hagai Taitelbaum, Hila Noga, Zhuyun Dai, James Wang, Chen Liang, Jenny Hamer, Chun-Sung Ferng, Chenel Elkind, Aviel Atias, Paulina Lee, Vít Listík, Mathias Carlen, Jan van de Kerkhof, Marcin Pikus, Krunoslav Zaher, Paul Müller, Sasha Zykova, Richard Stefanec, Vitaly Gatsko, Christoph Hirschall, Ashwin Sethi, Xingyu Federico Xu, Chetan Ahuja, Beth Tsai, Anca Stefanoiu, Bo Feng, Keshav Dhandhania, Manish Katyal, Akshay Gupta, Atharva Parulekar, Divya Pitta, Jing Zhao, Vivaan Bhatia, Yashodha Bhavnani, Omar Alhadlaq, Xiaolin Li, Peter Danenberg, Dennis Tu, Alex Pine, Vera Filippova, Abhipso Ghosh, Ben Limonchik, Bhargava Urala, Chaitanya Krishna Lanka, Derik Clive, Yi Sun, Edward Li, Hao Wu, Kevin Hongtongsak, Ianna Li, Kalind Thakkar, Kuanysh Omarov, Kushal Majmundar, Michael Alverson,

Michael Kucharski, Mohak Patel, Mudit Jain, Maksim Zabelin, Paolo Pelagatti, Rohan Kohli, Saurabh Kumar, Joseph Kim, Swetha Sankar, Vineet Shah, Lakshmi Ramachandruni, Xiangkai Zeng, Ben Bariach, Laura Weidinger, Tu Vu, Amar Subramanya, Sissie Hsiao, Demis Hassabis, Koray Kavukcuoglu, Adam Sadovsky, Quoc Le, Trevor Strohman, Yonghui Wu, Slav Petrov, Jeffrey Dean, and Oriol Vinyals. “Gemini: A Family of Highly Capable Multimodal Models.”, 2024.

- [TAL15] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. “CherryPick: Tracing Packet Trajectory in Software-defined Datacenter Networks.” In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR. ACM, 2015.
- [TAL16] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. “Simplifying Datacenter Network Debugging with Pathdump.” In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI. USENIX Association, 2016.
- [Tcp22] Tcpcdump.org. “TCPDUMP/LIBPCAP public repository.” <https://www.tcpcdump.org/>, 2022.
- [TK07] Grigorios Tsoumakas and Ioannis Katakis. “Multi-label classification: An overview.” *International Journal of Data Warehousing and Mining (IJDWM)*, **3**(3):1–13, 2007.
- [TL18] Sean J Taylor and Benjamin Letham. “Forecasting at scale.” *The American Statistician*, **72**(1):37–45, 2018.
- [Tor22] Adam Tornhill. “Microservice Dependencies - Visualization.” <https://codescene.com/blog/visualize-microservice-dependencies-in-team-context/>, 2022.
- [Twi15] Twitter Engineering. “Introducing practical and robust anomaly detection in a time series.” <https://bit.ly/3oS2Ry9>, 2015.
- [VM19] Kurt Vagner and Rani Molla. “After almost 24 hours of technical difficulties, Facebook is back - Vox.” <https://www.vox.com/2019/3/14/18265793/facebook-app-down-outage-resolved-fixed>, 2019.
- [VNN13] Nicolas Viennot, Siddharth Nair, and Jason Nieh. “Transparent Mutable Replay for Multicore Debugging and Patch Validation.” In *Proceedings of ASPLOS*, 2013.
- [VSP17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. “Attention is all you need.” In *Advances in Neural Information Processing Systems*, pp. 5998–6008, 2017.

- [Wal21] Sina Walleit. “Cloud Computing Environmental Benefits: Be Part of The Solution.” <https://www.parallels.com/blogs/ras/cloud-computing-environmental-benefits/>, 2021.
- [Wea17] Weaveworks. “Sock Shop: A Microservices Demo Application.” <https://microservices-demo.github.io/>, 2017.
- [WHL23] Tianyu Wu, Shizhu He, Jingping Liu, Siqi Sun, Kang Liu, Qing-Long Han, and Yang Tang. “A Brief Overview of ChatGPT: The History, Status Quo and Potential Future Development.” *IEEE/CAA Journal of Automatica Sinica*, **10**(5):1122–1136, 2023.
- [Wik21] Wikipedia.org. “Order Statistic - Wikipedia.” https://en.wikipedia.org/wiki/Order_statistic, 2021.
- [Wik22] Wikipedia.org. “Berkeley Packet Filter - Wikipedia.” https://en.wikipedia.org/wiki/Berkeley_Packet_Filter, 2022.
- [WST16] William W. Cohen, Charles Sutton, and Martin T. Vechev. “Programming with "Big Code" (Dagstuhl Seminar 15472).” 01 2016.
- [XCZ18] Haowen Xu, Wenxiao Chen, Nengwen Zhao, Zeyan Li, Jiahao Bu, Zhihan Li, Ying Liu, Youjian Zhao, Dan Pei, Yang Feng, et al. “Unsupervised anomaly detection via variational auto-encoder for seasonal kpis in web applications.” In *Proceedings of the 2018 World Wide Web Conference*, pp. 187–196, 2018.
- [XZW22] Shusheng Xu, Xingxing Zhang, Yi Wu, and Furu Wei. “Sequence level contrastive learning for text summarization.” In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pp. 11556–11565, 2022.
- [YDY19] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime G. Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. “XLNet: Generalized Autoregressive Pretraining for Language Understanding.” *CoRR*, **abs/1906.08237**, 2019.
- [YLZ14a] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. “Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems.” In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI’14, pp. 249–265, USA, 2014. USENIX Association.
- [YLZ14b] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U Jain, and Michael Stumm. “Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed {Data-Intensive} Systems.” In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pp. 249–265, 2014.

- [YNM16] Dong Young Yoon, Ning Niu, and Barzan Mozafari. “DBSherlock: A Performance Diagnostic Tool for Transactional Databases.” In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD ’16*, p. 1599–1614, New York, NY, USA, 2016. Association for Computing Machinery.
- [YWC18] Ziyu Yao, Daniel S. Weld, Wei-Peng Chen, and Huan Sun. “StaQC: A Systematically Mined Question-Code Dataset from Stack Overflow.” In *Proceedings of the 2018 World Wide Web Conference, WWW ’18*, 2018.
- [YYD16] Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alex Smola, and Eduard Hovy. “Hierarchical attention networks for document classification.” In *Proceedings of the 2016 conference of the North American chapter of the association for computational linguistics: human language technologies*, pp. 1480–1489, 2016.
- [Zip22] Zipkin.io. “OpenZipkin A distributed tracing system.” <https://zipkin.io/>, 2022.
- [ZLC20] Ming Zhong, Pengfei Liu, Yiran Chen, Danqing Wang, Xipeng Qiu, and Xuanjing Huang. “Extractive summarization as text matching.” *arXiv preprint arXiv:2004.08795*, 2020.
- [ZML20] Jie Zhou, Chunping Ma, Dingkun Long, Guangwei Xu, Ning Ding, Haoyu Zhang, Pengjun Xie, and Gongshen Liu. “Hierarchy-aware global model for hierarchical text classification.” In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 1106–1117, 2020.
- [ZYJ21a] Yongle Zhang, Junwen Yang, Zhuqi Jin, Utsav Sethi, Kirk Rodrigues, Shan Lu, and Ding Yuan. “Understanding and Detecting Software Upgrade Failures in Distributed Systems.” In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, SOSP ’21*, p. 116–131, New York, NY, USA, 2021. Association for Computing Machinery.
- [ZYJ21b] Yongle Zhang, Junwen Yang, Zhuqi Jin, Utsav Sethi, Kirk Rodrigues, Shan Lu, and Ding Yuan. “Understanding and detecting software upgrade failures in distributed systems.” In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pp. 116–131, 2021.
- [ZZL12] Jian Zhou, Hongyu Zhang, and David Lo. “Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports.” In *2012 34th International Conference on Software Engineering (ICSE)*, pp. 14–24. IEEE, 2012.
- [ZZS20] Jingqing Zhang, Yao Zhao, Mohammad Saleh, and Peter Liu. “Pegasus: Pre-training with extracted gap-sentences for abstractive summarization.” In *International Conference on Machine Learning*, pp. 11328–11339. PMLR, 2020.