

Intent-based System Design and Operation

Vaastav Anand[§]
Max Planck Institute for Software
Systems
Germany
vaastav@mpi-sws.org

Yichen Li[§]
The Chinese University of Hong
Kong
Hong Kong
ycli21@cse.cuhk.edu.hk

Alok Gautam Kumbhare
Microsoft
USA
alokk@microsoft.com

Celine Irvine
Microsoft
USA
celineirvene@microsoft.com

Chetan Bansal
Microsoft
USA
chetanb@microsoft.com

Gagan Somashekar
Microsoft
USA
gsomashekar@microsoft.com

Jonathan Mace
Microsoft
USA
jonathanmace@microsoft.com

Pedro Las-Casas
Microsoft
Brazil
pedrobr@microsoft.com

Rodrigo Fonseca
Microsoft
USA
fonseca.rodrigo@microsoft.com

Abstract

Cloud systems are the backbone of today’s computing industry. Yet, these systems remain complicated to design, build, operate, and improve. All these tasks require significant manual effort by both developers and operators of these systems. To reduce this manual burden, in this paper we set forth a vision for achieving holistic automation, *intent-based system design and operation*. We propose *intent* as a new abstraction within the context of system design and operation. Intent encodes the functional and operational requirements of the system at a high-level, which can be used to automate design, implementation, operation, and evolution of systems. We detail our vision of intent-based system design, highlight its four key components, and provide a roadmap for the community to enable autonomous systems.

1 Introduction

Cloud-based services are systems that are deployed in public cloud, run continuously, and are in a constant cycle of development and operation. These systems are typically distributed, have many components, and are always evolving. They have increasingly adopted a microservice architecture [11, 12, 18, 20], where each of these components are loosely coupled, can be developed separately using different libraries and frameworks, and scaled independently.

Full automation of cloud systems has been a long standing goal for developers and operators. However, despite the persistent need, automation of the design, building, operation, and maintenance [13, 15] of these systems has been elusive for multiple reasons. First, designing systems is a long and

arduous process which requires correctly handling a myriad of complex and intertwined requirements [16]. Second, there has been a lack of standardization and a dearth of available tooling that can allow developers to fully offload tasks. Third, cloud systems tend to be large and complex which prevents easy operation and understanding as it is impossible for any single operator to have full insight into the operational context of the system [13, 15]. Fourth, the environment in which systems operate is continuously changing but the systems are not designed to be easily reconfigurable [4].

We believe that we are now on the cusp of achieving the elusive goal of automation. This is for two reasons. First, there has been a confluence of standardization and automation tools, such as Kubernetes for deployment, maintenance, and convergence of systems to desired states [36]; OpenTelemetry for monitoring and observability via logs, traces, and metrics; and automatic bug-finding and verification tools that can be utilized in both development and production. Second, the recent rise of Large Language Models (LLMs) has provided increasingly sophisticated automatic coding and code understanding tools, and ways for operators to interact with their system at a higher level of abstraction. This has the potential to reduce the significant manual effort usually required by operators for tasks such as incident detection [38], incident management and mitigation [3, 17, 19, 23, 39], and root cause analysis [3, 6, 9, 35, 42, 43].

For cloud systems to fully embrace automation across all aspects, we need to be able to automatically carry out actions according to the user’s intent. To do so, we extend the idea of intent-based networking [10], where the network automatically configures itself to meet operators’ intent, to the

[§]Work done during internship at Microsoft

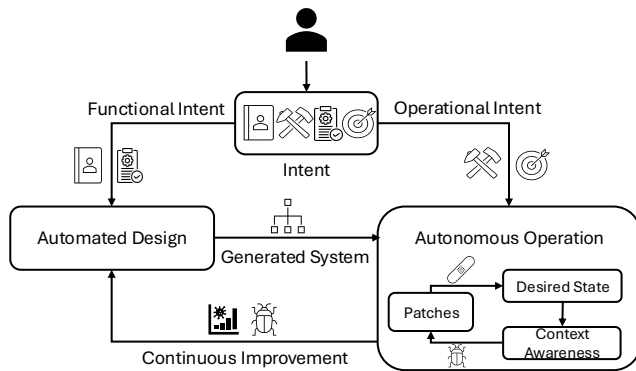


Figure 1: Intent-based cloud system components

Design a hotel reservation app as microservices. The app should allow searching for hotels near a location, search for activities near hotel, reserve and pay for a hotel room, read and write reviews about the hotels, and manage user's data.

Figure 2: Functional Intent Example

The app should have timely responses under high load and maintain a 100ms 99th percentile latency.

Figure 3: Operational Intent Example

broader context of cloud systems by combining automation tools with the generative capabilities of LLMs.

In this paper, we introduce our vision of intent-based self-managing cloud systems. Our goal is to enable users to specify high-level intents for the system, and have the system automatically *designed, developed, and operated*. We envision that the creators and operators of cloud systems will be able to describe at a high level their functional and operational intent for the system, and automatic, intelligent tools will be able to design, implement, and test the system, while integrating the monitoring necessary to operate the system within desired availability, reliability, and safety constraints.

2 Intent for Cloud System Design

We introduce *intent* [10] as a high-level abstraction within the context of system design and operations. Intent represents the potentially changing *functional* and *operational* requirements of the system from the user. Figure 1 illustrates the proposed components for a intent-based cloud system. The proposed components include automated design, automatic operation, and continuous improvement of systems.

We propose that there exist two broad classes of intent. First, *functional intent* represents the feature requirements of the system. These include the functional requirements, security requirements, as well as design requirements. Figure 2 shows an example functional intent for a hotel reservation application. Second, *operational intent* represents the operating

properties of the system which is used to derive the system SLA as well as metrics and monitors for gaining detailed insights into system behavior, ways to detect deviations from intended behavior, and strategies for mitigating issues and incidents. Figure 3 shows an example operational intent for an application. To accommodate changes to systems over time, we define the desired changes in the functional and operational intent as *refinement intent*. Refinement intent represents the delta between the initial intent and the new intent and is used for automatically improving the system.

2.1 Manifesting Intent

Currently, developers and operators manifest intent in cloud systems as part of the software development lifecycle (SDLC). SDLC is commonly divided into six phases [2] - (i) requirements engineering to extract desired features, (ii) prioritizing features and estimating effort, (iii) designing the system, (iv) implementing the selected design, (v) testing and productizing to ensure that the system is correct and understandable, (vi) deploying and maintaining the system.

As part of the SDLC, the functional and operational intent are extracted during the requirements engineering phase and converted into concrete actions *manually* taken by developers and operators through the other phases of the SDLC.

Instead of manifesting intent manually, we instead propose a human-in-the-loop approach for automating different phases of the SDLC to reduce the manual burden on developers and operators. Our human-in-the-loop approach uses LLMs to generate concrete actions that are then applied on the target system via automation tools.

2.2 Challenges

Reliability and Correctness. LLMs are well known to suffer from hallucinations [22] that lead to correctness issues in their output. Additionally, LLMs struggle with numerical and logical reasoning tasks leading to factually incorrect or inconsistent output. The output quality issue is further compounded for code generation tasks as the knowledge base of LLMs may consist of buggy, incorrect, or poorly written code. Ensuring the reliability of their outputs requires robust verification mechanisms and careful human oversight.

Explainability. Human operators must be able to verify and understand the rationale behind the actions selected by LLMs to ensure they align with the user intent. This requires implementing mechanisms for clear documentation, justification of actions, and validation processes to build trust and enable effective oversight. Failure to do so can cause unforeseen issues and monetary loss [7].

Providing accurate context. The outputs of LLM is directly dependent on the quality of the information provided as context in the input prompt. Cloud systems tend to be large in size spanning thousands of lines of code and documentation,

generating billions of traces [24], and PetaBytes of logs [31] per day. Extracting relevant information to serve as context for inputs to the LLM is a challenging task.

Action Selection. LLMs suffer from instruction inconsistency [22], in which LLMs deviate from user directives. This deviation can lead the LLM to misinterpret the user’s intent and select inappropriate actions. The problem of selecting the correct action is exacerbated by the large number of potential actions in large-scale cloud systems.

Dynamic Adaptation. Certain actions selected by the LLM might require changing the system online. This requires the systems to dynamically adapt while still running without requiring bringing the system offline. The system needs to have the ability to dynamically reconfigure itself and continuously update and adapt.

3 Intent-based self-managing cloud systems

In typical design and implementation of cloud systems, developers and operators manually manifest intent for four high level tasks. In this section, we detail an LLM-based approach for automating intent manifestation for these four tasks.

3.1 Distributed System Design

Developers often struggle with designing distributed systems because it requires managing the complexities of multiple independent moving pieces while ensuring the correctness, performance, and reliability of the system. To combat these issues, developers rely on principles and hints [25, 26] to select an appropriate design which they then manually implement using standardized tools such as Docker and OpenTelemetry. Naturally, this is a manually arduous process.

Key Idea. To alleviate the manual effort on developer, we propose using LLMs to convert the intent of the users, provided as user requirements, into concrete implementations.

User Requirements. The user requirements represent the intent of the various stakeholders. The functional requirements and the desired architecture pattern comprise the functional intent of the system. The behavioral properties of the system are the operational intent of the system.

Requirements. To generate a reliable, effective design of a distributed system, there are three different classes of requirements that a automatically generated system must provide. First, correctness guarantees, which include correctness with respect to user requirements, test suites, and formal specifications of the system. Second, explainability guarantees: the generated code must be understandable by humans and should provide more artifacts that can be used by developers to gain insights into the system. Third, performance guarantees, which may include scalability, SLOs, and absence of emergent misbehaviors [8, 21, 32].

3.1.1 Use Case: Generating Microservices

Microservices are a pervasive design architecture commonly used for developing modern cloud systems [11, 12]. Due to their importance, there has been a growing interest in automating the generation and deployment of microservice systems [1, 4, 14]. However, the effort has largely been focused on automating the generation of infrastructure components of the system rather than the business logic.

Cerulean [5] is a human-in-the-loop system that combines the generative capabilities of LLMs to generate the business logic of the system and then converts the business logic of the system into input specifications of Blueprint [4]. To generate the business logic, Cerulean proposes a *hierarchical generation* procedure that decomposes the system generation process into multiple steps at different levels of abstraction of the system design process including high-level design, low-level design, and unit-test generation. This process decomposes the functional intent and iteratively converts the intent into specific design choices of the system.

We leverage the modularity of Cerulean and extend the hierarchical generation process with two novel components to show how the *hierarchical generation* process can be extended to improve the quality of the generated system. First, we introduce an end-to-end test-case generation component that extends the *implementation generation* phase of the hierarchical generation process to also generate end-to-end tests of the system. To do this, first the component uses LLMs to extract end-to-end use-cases from the functional intent of the user. It then generates an implementation plan consisting of API calls to the frontend service(s) of the system using the generated interfaces in a previous phase. The component then uses this plan to generate a concrete end-to-end test that can be executed as a traditional go test or be compiled as a blackbox test that can be run against the deployed system. Second, we introduce a workload generator component that automatically generates workload processes that can be used by developers to benchmark the deployed system. The workload generator component expects that the input operational intent includes a description of the target workload. If the description is missing, then the user is prompted to provide a description. The component then uses this description along with the previously generated interfaces to generate a process that exercises the target workload when executed against the deployed system.

We believe that Cerulean’s hierarchical generation process can be further extended to provide additional guarantees for the generated system such as verification guarantees by incorporating the use and generation of formal models.

3.2 Real-Time Context Awareness

Real-time context awareness refers to the ability of the system to continuously and accurately understand its current operational state and the context in which it operates. *This understanding is essential for the system to make informed decisions and take appropriate actions to meet the user's intent.*

Key idea. To provide a cloud system the ability to continuously and autonomously comprehend its state and operational environment in alignment with the user's specified intent, we combine advanced monitoring techniques with LLMs. By correlating the different sources and forms of runtime data, we create a unified representation of the system's operational state. This representation is then connected with the system's domain knowledge (e.g., code, documentation) to generate the system *context*. As opposed to existing single-modal data approaches for analyses tasks [28–30, 33], context encapsulates and connects relevant information from various sources to provide a holistic view which can be used for further automated operations or for providing developers with relevant information for analysis tasks.

Context. Context represents the current operational information of the system comprising both runtime information and domain knowledge. Runtime information includes metrics, logs, traces, and monitors. Domain knowledge includes code, documentation, and operational guidelines such as troubleshooting guides. The real-time context awareness framework provides the intent-related contextualized and summarized knowledge for analyses tasks.

Requirements. To achieve effective real-time context awareness, it is essential to have comprehensive observability that covers different aspects of the system, including performance metrics, logs and traces. Cloud systems generate a large amount of data. It is crucial to understand the intent and use it a guiding principle to filter the relevant data.

3.2.1 Use Case: Behavior Comprehension

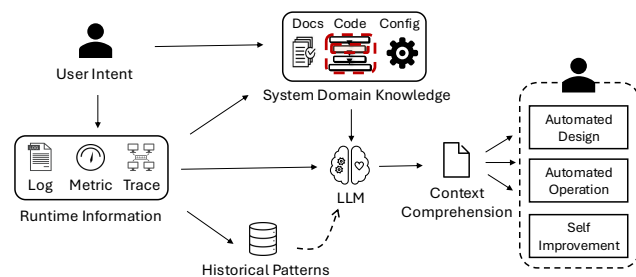


Figure 4: Real-time context awareness.

We bridge runtime data with service domain knowledge based on user intent, as shown in Figure 4. The intent is essential to determining the scope of the data that is needed. This framework allows us to extract and refine pertinent

system domain knowledge, offering real-time system comprehension based on user intents.

Runtime Data Modeling. We model multi-modal runtime data as loosely unified event graphs [41], representing the system's current execution state for real-time retrieval and modeling based on user intent. The runtime data includes metrics, monitors, logs, and traces. For the metrics, we represent their events as anomalies or deviations from the normal behavior. For logs, we mine patterns to identify sequences that require particular attention and understanding, treating each subsequence as an event. Traces function as connectors between these events, delineating the interactions and dependencies among various components. This integrated approach allows us to create a real-time and comprehensive mapping of the system behavior.

System Domain Knowledge Extraction. We extract relevant descriptions from documentation for logs and alerts that involve specific terms. We extract the related source code for logs and integrate it into our context.

Comprehension Generation. We continuously correlate and integrate context data to provide real-time system comprehension. Pattern mining and comprehension generation are triggered on-demand, avoiding unnecessary cost. The generated system comprehension serves as fundamental, shareable knowledge for users and different components.

3.3 System Operation

Operating a system is a complex task. Systems can be composed of many different components, can be built at different layers and usually evolves over time. Despite several advances in automating different aspects of system operation [34, 40], there is still a constant stream of failures and incidents that require human intervention.

Key idea. To fully automate system operation, we translate operational intent provided by the user into an *ops model* that is used to automate the system operation. We combine the model with the real-time insights from the context comprehension component, which enables the system to anticipate potential issues, automate decision-making, and execute operational tasks with minimal human intervention.

Ops model. The operational intent provides the operational properties of the system including the SLOs and SLAs. The operation intent is translated to an operational model, *ops model*, that is used to guide the system operation. The *ops model* defines the desired state of the system and identifies and prioritizes the potential risks and vulnerabilities in the system operations to SLAs. It formalizes the set of observability (metrics, monitors, logs, traces) required to identify and diagnose potential issues and defines the set of mitigations and countermeasures to be taken in case of failures. The *ops model* is not static and can evolve over time as the system evolves and the user requirements change.

Requirements. To effectively operate a system based on the user’s intent, it is essential to have a clear definition of the desired state, the operations, their outcomes and constraints. It is also important to have comprehensive understanding of the system’s state and environment, including the ability to anticipate potential deviations from the desired state and to identify and mitigate these issues. Furthermore, it is required an intelligent decision-making engine that can interpret the operational intent and contextual insights, enabling automated actions and proactive mitigation strategies.

3.3.1 Use Case: Automated Incident Management

Current cloud providers rely on human intervention guided by troubleshooting guides (TSGs) to mitigate and resolve issues that frequently occur to their services. Automating the execution of TSGs can significantly reduce the time to mitigation (TTM) and reduce the burden of SREs. For example, Llexus [27] is a tool that aims to automate the execution of TSGs by using LLM agents to produce executable plans from a source TSG. Llexus uses human-generated troubleshooting guides to create executable plans. These plans are executed when new incidents occur, enabling automatic mitigation and resolution of issues in cloud services.

As noted, this current approach improves the process of automatically mitigating and resolving issues in cloud services, but it still requires human curated TSGs with high quality and coverage as input. We envision that leveraging the *ops model* along with real-time context comprehension can provide a powerful framework for automating incident management in cloud services. The ops model’s detailed definition of operational requirements, observability metrics, and potential mitigation, combined with the context comprehension service’s ability to dynamically monitor and understand the system state, can enable the automatic generation of actionable instructions. These instructions can be used as input to Llexus to generate executable plans and automatically mitigate and resolve possible issues that might happen to the system.

3.4 System Improvement

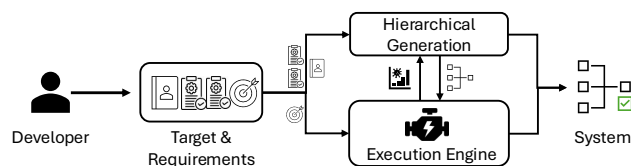


Figure 5: System improvement with Hierarchical Generation

As the system is deployed and runs, there can be deviations from intent due to several causes. There can be many reasons for intent violation, including unseen bugs, changes

in workloads, metastable failures [8], or changes in the intent itself. We need the system to automatically detect such intent violations and adjust itself, either by changing configurations, fixing the code, or redesigning parts, or the whole of, the system itself.

Key Idea. We detect intent violations by coupling the functional and operation intents, at different levels, with the real-time context awareness described in §3.2 to generate the refinement intent. We then address the refinement intent either by dynamically re-configuring the system [37] or by re-designing the system §3.1 at the desired abstraction level. Figure 5 depicts the continuous improvement process that uses hierarchical generation process from Cerulean [5] to continuously re-generate implementations of the system.

3.4.1 Use Case: Mitigating Metastable Failures

Metastable failures [8, 21] are caused by a trigger that either increases the load on the system or reduces the capacity of the system to handle the load. These triggers are often unknown at development time; thus preventing the systems from testing and safeguarding against such scenarios.

Once these metastable failures happen in deployed systems, we use the data collected from the context-awareness and autonomous operations components to generate trigger-scenarios which are used to reproduce the metastable failures in controlled settings of the system. We can then use hierarchical generation to generate new candidate designs for the system and then re-run the trigger scenarios to test if the system can avoid going into a metastable state while still adhering to the functional and operational intent. This process is repeated until a suitable design is found.

4 Future Directions

The realization of fully autonomous intent-based systems necessitates research advancements in several areas. We outline the key directions to achieve this vision.

Manifesting Intent. It is crucial to help users specify, refine, and understand their intent, balancing specificity, ambiguity, and user-friendliness. The goal is to minimize user burden while efficiently translating intent into service and operation models. It is also critical to keep the users engaged, instead of just pressing ‘yes’ in key human-in-the-loop moments.

Autonomous Design and Implementation. Challenges include deterministic translation from models to systems, providing explainability and verifiability for generated artifacts, reasoning about design choices, optimizing for given intents and metrics, and building testable solutions. Mitigating LLMs hallucinations and non-determinism, integrating formal verification, and dealing with the cases in which both the generated systems and tests agree, but are wrong, are concrete areas of research.

Service and Operations Modeling. These models, capturing both functional and operational requirements, form the first step in translating intent into system design. The challenge lies in developing comprehensive, machine-readable models that can generate and operate the system while providing explainability and verifiability.

Reasoning and Decision Making. The system should incorporate intent and service context to determine the best course of action for an intent violation. The challenge is to develop reasoning mechanisms that can handle uncertainty, conflicting intents, and changing contexts, while providing human-consumable explanations.

Autonomous Operations and Self-healing. These involve short-term mitigations and long-term fixes. A key challenge is to find the right level of intervention. Continuous learning and adaptation, an integral part of the system, presents challenges including representation, guard-rails, reliability of operations, and security.

Software Engineering Processes must evolve to support intent-based systems. Traditional methodologies like Agile will need to incorporate autonomy to support the design, implementation, testing, and deployment of such systems.

5 Conclusion

We presented a vision for an intent-based cloud system design and operation that aims to enable fully autonomous systems that can understand, design, operate, and improve themselves based on user intent. We believe that such a grand vision would require multiple research communities to explore solutions in tandem to address these challenges and lead to a new class of systems and services equipped with improved productivity, reliability, and maintenance.

References

- [1] Dapr: Distributed application runtime. <https://dapr.io/>.
- [2] P. Abrahamsson, O. Salo, J. Ronkainen, and J. Warsta. Agile software development methods: Review and analysis. *arXiv preprint arXiv:1709.08439*, 2017.
- [3] T. Ahmed, S. Ghosh, C. Bansal, T. Zimmermann, X. Zhang, and S. Rajmohan. Recommending root-cause and mitigation steps for cloud incidents using large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1737–1749. IEEE, 2023.
- [4] V. Anand, D. Garg, A. Kaufmann, and J. Mace. Blueprint: A toolchain for highly-reconfigurable microservice applications. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 482–497, 2023.
- [5] V. Anand, A. Kumbhare, C. Irvine, C. Bansal, G. Somashekar, J. Mace, P. Las-Casas, and R. Fonseca. Automated service design with cerulean. To appear in 6th International Workshop on Cloud Intelligence / AIOps (AIOps '25), 2025.
- [6] V. Anand, P. Las-Casas, R. Fonseca, and A. Kaufmann. Towards using llms for distributed trace comparison. To appear in 6th International Workshop on Cloud Intelligence / AIOps (AIOps '25), 2025.
- [7] Asim. How a single chatgpt mistake cost us \$10,000+. Accessed 9th June, 2024 from <https://web.archive.org/web/20240610032818/https://asim.bearblog.dev/how-a-single-chatgpt-mistake-cost-us-10000/>, 2024.
- [8] N. Bronson, A. Aghayev, A. Charapko, and T. Zhu. Metastable failures in distributed systems. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 221–227, 2021.
- [9] Y. Chen, H. Xie, M. Ma, Y. Kang, X. Gao, L. Shi, Y. Cao, X. Gao, H. Fan, M. Wen, et al. Empowering practical root cause analysis by large language models for cloud incidents. *arXiv preprint arXiv:2305.15778*, 2023.
- [10] A. Clemm, L. Ciavaglia, L. Z. Granville, and J. Tantsura. Rfc 9315: Intent-based networking - concepts and definitions, 2022.
- [11] A. Cockcroft. The evolution of microservices. (April 2016). Retrieved October 2020 from <https://www.slideshare.net/adriancockcroft/evolution-of-microservices-craft-conference>, 2016.
- [12] A. Cockcroft. Microservices workshop: Why, what, and how to get there. (April 2016). Retrieved October 2020 from <https://www.slideshare.net/adriancockcroft/microservices-workshop-craft-conference>, 2016.
- [13] V. Ganatra, A. Parayil, S. Ghosh, Y. Kang, M. Ma, C. Bansal, S. Nath, and J. Mace. Detection is better than cure: A cloud incidents perspective. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1891–1902, 2023.
- [14] S. Ghemawat, R. Grandl, S. Petrovic, M. Whittaker, P. Patel, I. Posva, and A. Vahdat. Towards modern development of cloud applications. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, pages 110–117, 2023.
- [15] S. Ghosh, M. Shetty, C. Bansal, and S. Nath. How to fight production incidents? an empirical study on a large-scale cloud service. In *Proceedings of the 13th Symposium on Cloud Computing*, pages 126–141, 2022.
- [16] A. Gluck. Introducing domain-oriented microservice architecture. Accessed June 2024 from <https://www.uber.com/blog/microservice-architecture/>, 2020.
- [17] D. Goel, F. Husain, A. Singh, S. Ghosh, A. Parayil, C. Bansal, X. Zhang, and S. Rajmohan. X-lifecycle learning for cloud incident management using llms. *arXiv preprint arXiv:2404.03662*, 2024.
- [18] E. Haddad. Service-oriented architecture: Scaling the uber engineering codebase as we grow. (September 2015). Retrieved October 2020 from <https://eng.uber.com/service-oriented-architecture/>, 2015.
- [19] P. Hamadani, B. Arzani, S. Fouladi, S. K. R. Kakarla, R. Fonseca, D. Billor, A. Cheema, E. Nkposong, and R. Chandra. A holistic view of ai-driven network incident management. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, pages 180–188, 2023.
- [20] M. Hashemi. The infrastructure behind twitter : Scale. (January 2017). Retrieved February 2021 from https://blog.twitter.com/engineering/en_us/topics/infrastructure/2017/the-infrastructure-behind-twitter-scale.html, 2017.
- [21] L. Huang, M. Magnusson, A. B. Muralikrishna, S. Estyak, R. Isaacs, A. Aghayev, T. Zhu, and A. Charapko. Metastable failures in the wild. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 73–90, 2022.
- [22] L. Huang, W. Yu, W. Ma, W. Zhong, Z. Feng, H. Wang, Q. Chen, W. Peng, X. Feng, B. Qin, et al. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *arXiv preprint arXiv:2311.05232*, 2023.
- [23] Y. Jiang, C. Zhang, S. He, Z. Yang, M. Ma, S. Qin, Y. Kang, Y. Dang, S. Rajmohan, Q. Lin, et al. Xpert: Empowering incident management with query recommendations via large language models. *arXiv preprint arXiv:2312.11988*, 2023.

- [24] J. Kaldor, J. Mace, M. Bejda, E. Gao, W. Kuropatwa, J. O'Neill, K. W. Ong, B. Schaller, P. Shan, B. Viscomi, et al. Canopy: An end-to-end performance tracing and analysis system. In *Proceedings of the 26th symposium on operating systems principles*, pages 34–50, 2017.
- [25] B. Lampson. Hints and principles for computer system design. *arXiv preprint arXiv:2011.02455*, 2020.
- [26] B. W. Lampson. Hints for computer system design. In *Proceedings of the ninth ACM symposium on Operating systems principles*, pages 33–48, 1983.
- [27] P. Las-Casas, A. Kumbhare, R. Fonseca, and S. Agarwal. Llexus: an ai agent system for incident management. *SIGOPS Oper. Syst. Rev.*, 58(1), 2024. To appear.
- [28] C. Lee, T. Yang, Z. Chen, Y. Su, and M. Lyu. Maat: Performance metric anomaly anticipation for cloud services with conditional diffusion. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 116–128. IEEE, 2023.
- [29] Y. Li, X. Zhang, S. He, Z. Chen, Y. Kang, J. Liu, L. Li, Y. Dang, F. Gao, Z. Xu, et al. An intelligent framework for timely, accurate, and comprehensive cloud incident detection. *ACM SIGOPS Operating Systems Review*, 56(1):1–7, 2022.
- [30] F. Lin, K. Muzumdar, N. P. Laptev, M.-V. Curelea, S. Lee, and S. Sankar. Fast dimensional analysis for root cause investigation in a large-scale service environment. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 4(2):1–23, 2020.
- [31] J. Liu, J. Zhu, S. He, P. He, Z. Zheng, and M. R. Lyu. Logzip: Extracting hidden structures via iterative clustering for log compression. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 863–873. IEEE, 2019.
- [32] J. C. Mogul. Emergent (mis) behavior vs. complex software systems. *ACM SIGOPS Operating Systems Review*, 40(4):293–304, 2006.
- [33] C. M. Rosenberg and L. Moonen. Spectrum-based log diagnosis. In *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–12, 2020.
- [34] D. Roy, X. Zhang, R. Bhavne, C. Bansal, P. Las-Casas, R. Fonseca, and S. Rajmohan. Exploring llm-based agents for root cause analysis. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, FSE 2024, page 208–219, New York, NY, USA, 2024. Association for Computing Machinery.
- [35] V. Seshagiri, S. Balyan, V. Anand, K. Dhole, I. Sharma, A. Wildani, J. Cambronero, and A. Züfle. Chatting with logs: An exploratory study on finetuning llms for logql. *arXiv preprint arXiv:2412.03612*, 2024.
- [36] E. Shanks. Kubernetes - desired state and control loops. Accessed July, 2024 from <https://theithollow.com/2019/09/16/kubernetes-desired-state-and-control-loops/>, 2019.
- [37] G. Somashekar, K. Tandon, A. Kini, C.-C. Chang, P. Husak, R. Bhagwan, M. Das, A. Gandhi, and N. Natarajan. OPPerTune: Post-Deployment configuration tuning of services made easy. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1101–1120, Santa Clara, CA, Apr. 2024. USENIX Association.
- [38] P. Srinivas, F. Husain, A. Parayil, A. Choure, C. Bansal, and S. Rajmohan. Intelligent monitoring framework for cloud services: A data-driven approach. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*, pages 381–391, 2024.
- [39] H. Wang, G. K. Tangirala, G. P. Naidu, C. Mayville, A. Roy, J. Sun, and R. B. Mandava. Anomaly detection for incident response at scale. *arXiv preprint arXiv:2404.16887*, 2024.
- [40] Z. Xie, Y. Zheng, L. Ottens, K. Zhang, C. Kozyrakis, and J. Mace. Cloud atlas: Efficient fault localization for clou systems using language models and causal insight. Accessed 11th July, 2024 from <https://people.mpi-sws.org/~jcmace/papers/xie2024cloud.pdf>, 2024.
- [41] G. Yu, P. Chen, Y. Li, H. Chen, X. Li, and Z. Zheng. Nezha: Interpretable fine-grained root causes analysis for microservices on multi-modal observability data. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 553–565, 2023.
- [42] D. Zhang, X. Zhang, C. Bansal, P. Las-Casas, R. Fonseca, and S. Rajmohan. Lm-pace: Confidence estimation by large language models for effective root causing of cloud incidents. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 388–398, 2024.
- [43] X. Zhang, S. Ghosh, C. Bansal, R. Wang, M. Ma, Y. Kang, and S. Rajmohan. Automated root causing of cloud incidents using in-context learning with gpt-4. *arXiv preprint arXiv:2401.13810*, 2024.