Intent-based System Design and Operation

Vaastav Anand[§]
Max Planck Institute for Software Systems
Germany

Yichen Li[§]
The Chinese University of Hong Kong
Hong Kong

Alok Gautam Kumbhare
Microsoft
USA

Celine Irvene
Microsoft
USA

Chetan Bansal
Microsoft
USA

Gagan Somashekar
Microsoft
USA

Jonathan Mace
Microsoft
USA

Pedro Las-Casas
Microsoft
Brazil

Ricardo Bianchini
Microsoft
USA

Rodrigo Fonseca
Microsoft
USA

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 detail the necessary challenges that the community must address to ensure reliability and efficiency with intent-based systems in practical settings.

CCS Concepts

• **Networks** → **Cloud computing**; • **Software and its engineering** → **Cloud computing**.

Keywords

System Design, Autonomous Computing

This work is licensed under a Creative Commons Attribution 4.0 International License.
PACMI'25, October 13-16, 2025, Seoul, South Korea
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2205-9/2025/10
https://doi.org/10.1145/3766882.3767182

ACM Reference Format:
Vaastav Anand[§], Yichen Li[§], Alok Gautam Kumbhare, Celine Irvene, Chetan Bansal, Gagan Somashekar, Jonathan Mace, Pedro Las-Casas, Ricardo Bianchini, and Rodrigo Fonseca. 2025. Intent-based System Design and Operation. In *Practical Adoption Challenges of ML for Systems (PACMI '25), October 13–16, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3766882.3767182

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 [13, 14, 22, 23], 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 [17, 19] 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 [20]. 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 [17, 19]. Fourth, the environment in which

[§]Work done during internship at Microsoft

systems operate is continuously changing but the systems are not designed to be easily reconfigurable [5].

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 [33]; 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 [9, 11, 15]. Second, the recent rise of Large Language Models (LLMs) has provided developers and operators with increasingly sophisticated automatic coding and code understanding tools through compound AI systems [36] and autonomous agentic systems [7]. This is particularly visible with tools like Lovable [30] and AutoBE [32] which can generate designs and implementations of backends from functional requirements. However, these tools only focus on the functional aspect of systems and do not incorporate the operational aspects of the system.

For cloud systems to fully embrace automation across all aspects, systems 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 [12], where the network automatically configures itself to meet operators' intent, to the 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 and enlist all the practical challenges we would need to solve for the adoption of these 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. We enlist the current roadblocks and challenges that the community must solve to adopt intent-based cloud systems in practice.

2 Intent for Cloud System Design

We introduce *intent* [12] 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.

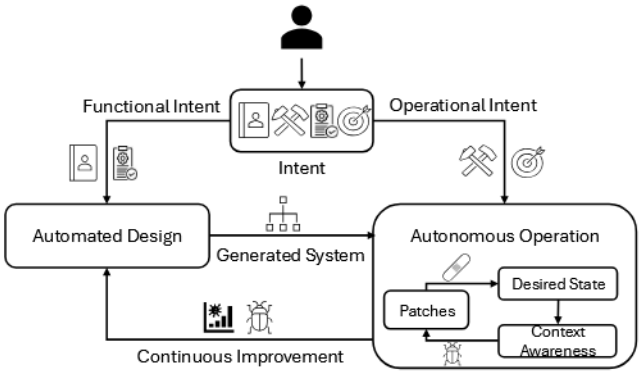

Figure 1: **Intent-based cloud system components**

```
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**

Intent Types. 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.

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 correctness, (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

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

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 [10, 24, 31].

Use Case: Automated Microservice Design. Microservices are a pervasive design architecture commonly used for developing modern cloud systems [13, 14]. Due to their importance, there has been a growing interest in automating the generation and deployment of microservice systems [1, 5, 18]. To make it easy for generating microservice implementations, we are building Cerulean [6], 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 [5]. 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.

3.2 Real-Time Context Awareness

We define *Real-time context awareness* as 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.*

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.

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*.

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.

3.3 Autonomous Operation

Systems are composed of many different components at different layers. Operating a system is a complex task.

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 and their outcomes and constraints. This requires a comprehensive understanding of the system's state and environment as well as the ability to anticipate potential deviations from the desired state, and to identify and mitigate these issues.

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.

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* evolves over time as the system evolves and the operational intent changes.

Use Case: Automated Incident Management. Current cloud providers rely on human intervention guided by troubleshooting guides (TSGs) to mitigate and resolve incidents. Automating the execution of TSGs can significantly reduce the time to mitigation (TTM) and reduce the burden of SREs. For example, we have built `Llexus` [28], a tool that automates 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. 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 by generating actionable instructions. `Llexus` can further use these instructions to generate executable plans and automatically mitigate and resolve possible issues that might happen to the system.

3.4 Continuous Improvement

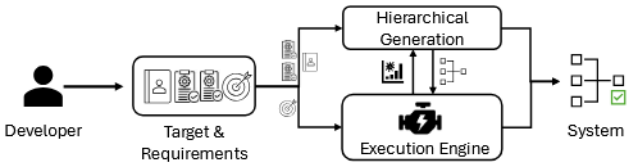

Figure 4: **System improvement with Hierarchical Generation**

As the system executes, 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 [10], or changes in the intent. We need the system to automatically detect intent violations and adjust itself, either by changing configurations, fixing the code, or redesigning parts, or the whole of, the system.

We detect intent violations by coupling the functional and operation intents, at different levels, with the real-time context awareness to generate the refinement intent. We then address the refinement intent either by dynamically re-configuring the system [4, 34] or by re-designing the system at the desired abstraction level using Cerulean. Figure 4 depicts the continuous improvement process that uses hierarchical generation process from Cerulean to continuously re-generate implementations of the system.

4 Adoption Challenges

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

Quality of Intent. The quality of the user-specified intent dictates the efficacy of the underlying system. Specifying good-quality intent is almost entirely dependent on the process of requirement engineering and elicitation [21, 35]. However, it remains unclear at the moment as to the level of detail and precision that the user-specified intent must meet in order to provide a good quality system. Ensuring high-quality of intent from the users might require developing future language-specific abstractions to aid the users in succinctly describing their intent. One possible of achieving this could be to leverage Paralegal's abstractions [3] to restrict the user-provided intent to first-order logic stylized English.

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. This is also an active area of research for Intent-Based Networking systems [26].

Intent Lifecycle Management. Functional and Operational requirements evolve over time. Consequently, the functional and operational intents of the system must also evolve to avoid drifting away from the desired requirements of the system. This is akin to how formal specifications of systems can deviate from the actual implementations. Just like how specifications must-be kept up-to-date, the intents must also be kept up-to-date. This adds additional burden onto the users and developers.

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 [27], and PetaBytes of logs [29] per day. Extracting relevant information to serve as context for inputs to the LLM is a challenging task that requires precision. Incorrect instructions can lead to catastrophic consequences like deletion of in-production databases [16].

Action Selection. LLMs suffer from instruction inconsistency [25], 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. Moreover, 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.

Reliability and Correctness. LLMs are well known to suffer from hallucinations [25] 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 [8].

Handling Model Drift. The underlying large language models used by various components may independently evolve over time. As a result, the quality of the outputs and the selected actions may vary leading to less than optimal system designs and operating conditions.

Debugging & Observability. Like any software, these systems will also suffer from bugs. Thus, there needs to be adequate frameworks and tooling that can help users find bugs in these systems. This is especially challenging as these systems are inherently non-deterministic due to the underlying non-deterministic models and consequently cannot be exhaustively tested in development.

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] J. Adam, C. Zech, L. Zhu, S. Rajesh, N. Harbison, M. Jethwa, W. Crichton, S. Krishnamurthi, and M. Schwarzkopf. Paralegal: Practical static analysis for privacy bugs. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pages 957–978, 2025.

[4] V. Anand, D. Garg, and A. Kaufmann. Iridescent: A framework enabling online system implementation specialization. *arXiv preprint arXiv:2508.16690*, 2025.

[5] 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.

[6] V. Anand, A. G. Kumbhare, C. Irvene, C. Bansal, G. Somashekar, J. Mace, P. Las-Casas, R. Bianchini, and R. Fonseca. Automated service design with cerulean (project showcase). In *2025 IEEE/ACM International Workshop on Cloud Intelligence & AIOps (AIOps)*, pages 1–3. IEEE, 2025.

[7] Anthropic Blog. Building effective agents. Accessed July 2025 from https://www.anthropic.com/engineering/building-effective-agents, 2024.

[8] 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.

[9] J. Bornholt, R. Joshi, V. Astrauskas, B. Cully, B. Kragl, S. Markle, K. Sauri, D. Schleit, G. Slatton, S. Tasiran, et al. Using lightweight formal methods to validate a key-value storage node in amazon s3. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 836–850, 2021.

[10] 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.

[11] M. Brooker and A. Desai. Systems correctness practices at aws: Leveraging formal and semi-formal methods. *Queue*, 22(6):79–96, 2024.

[12] A. Clemm, L. Ciavaglia, L. Z. Granville, and J. Tantsura. Rfc 9315: Intent-based networking - concepts and definitions, 2022.

[13] A. Cockcroft. The evolution of microservices. (April 2016). Retrieved October 2020 from https://www.slideshare.net/adriancockcroft/evolution-of-microservices-craft-conference, 2016.

[14] 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.

[15] P. Deligiannis, N. Ganapathy, A. Lal, and S. Qadeer. Building reliable cloud services using p#(experience report). *arXiv preprint arXiv:2002.04903*, 2020.

[16] A. Edser. 'i destroyed months of your work in seconds' says ai coding tool after deleting a dev's entire database during a code freeze: 'i panicked instead of thinking'. Accessed July 2025 from https://www.pcgamer.com/software/ai/i-destroyed-months-of-your-work-in-seconds-says-ai-coding-tool-after-deleting-a-devs-entire-database-during-a-code-freeze-i-panicked-instead-of-thinking/, 2025.

[17] 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.

[18] 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.

[19] 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.

[20] A. Gluck. Introducing domain-oriented microservice architecture. Accessed June 2024 from https://www.uber.com/blog/microservice-architecture/, 2020.

[21] J. A. Goguen and C. Linde. Techniques for requirements elicitation. In *[1993] Proceedings of the IEEE International Symposium on Requirements Engineering*, pages 152–164. IEEE, 1993.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] A. S. Jacobs, R. J. Pfitscher, R. H. Ribeiro, L. Z. Granville, R. A. Ferreira, W. Willinger, and S. G. Rao. Establishing trust for using natural language for intent-based networking. *IEEE Transactions on Network and Service Management*, 2025.

[27] 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.

[28] P. Las-Casas, A. G. Kumbhare, R. Fonseca, and S. Agarwal. Llexus: an ai agent system for incident management. *ACM SIGOPS Operating Systems Review*, 58(1):23–36, 2024.

[29] 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.

[30] Lovable. Build something lovable: Create apps and websites by chatting with ai. Accessed July 2025 from https://lovable.dev/, 2025.

[31] J. C. Mogul. Emergent (mis) behavior vs. complex software systems. *ACM SIGOPS Operating Systems Review*, 40(4):293–304, 2006.

[32] J. Nam. [autobe] we made ai-friendly compilers for vibe coding, achieving 100 Accessed July 2025 from https://dev.to/samchon/autobe-we-made-ai-friendly-compilers-for-vibe-coding-491k, 2025.

[33] 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.

[34] 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.

[35] A. Van Lamsweerde. Requirements engineering in the year 00: A research perspective. In *Proceedings of the 22nd international conference on Software engineering*, pages 5–19, 2000.

[36] M. Zaharia, O. Khattab, L. Chen, J. Q. Davis, H. Miller, C. Potts, J. Zou, M. Carbin, J. Frankle, N. Rao, and A. Ghodsi. The shift from models to compound ai systems. Accessed July 2025 from https://bair.berkeley.edu/blog/2024/02/18/compound-ai-systems/, 2024.