Deep Science Publishing, 2026
https://doi.org/10.70593/978-81-69589-32-1_9

DeepScience
Open Access Books

Chapter 9: Resilience, Observability, and Performance at Infrastructure Scale

9.1. Introduction

Infrastructure supporting Cloud and other large-scale applications often demand high resilience against faults and performance agility in responding to varying usage patterns. Understanding and addressing these challenges are mathematical, engineering, and operational disciplines in their own right. Clear, operationally relevant definitions, models, and technologies can simplify parts of the problems of large-scale resiliency, observability, and performance. Nevertheless, mathematical foundations and assistive technologies cannot relieve engineering and operations teams of planning for and working through the wide variety of faults that occur at scale.

Resilience of large-scale infrastructures implies the joint property that the infrastructure is always operational (cannot be irrecoverably broken), and that service disruptions are not only avoided but detected and reported, ideally by those affected. Team central to maintaining this resilience formally define the following essential concepts: Recoverable Fault—non-catastrophic fault in infrastructure layer that can be detected and is independently resolvable. Capacity/fault-resilience—resilience property such that, unless all nodes of a certain type fail simultaneously, no service disruption occurs. Performance Testing and Benchmarking is one of the essential considerations of any system targeting cloud and other large-scale application deployments, whether for customer-facing usage or for internal project development.

9.1.1. Cache Design and Data Locality

Most web requests can be served quickly from cache; a small fraction requires round trips to back-end storage systems, which contributes to the bulk of the latency. The delay in resolving a critical path is due to network or slowest-device latency. People notice chat conversations taking a long time to appear on a device. The evolution of personal

chat history improved performance and chats check for recent message history when sending, performance is further enhanced by horizontal partition of other features. Care has been taken in understanding root causes and testing all changes. Country or region information that is densely used such as for upload or view material can reside in a separate redis located in the same region. The concern with redis is the high number of keys. For some infrequently changing large keys, alternative modes of caching have been tried. For china market, material video is cached in cdn located in china. A caching layer designed using a cloud provider's object storage system supports very high read throughput at extremely low unit cost.

New applications and features run much slower than established ones. Load testing must be done to ensure the new functionality is On par with the existing one in terms of performance. Binomial distribution reflects the statistical behaviour for load-tester response time and number of requests. Negative exponential distribution represents the time taken by any component of a system to respond and is heavily influenced by the component's link with external system. Prediction of expected response time before deploying a new feature will help remediate latency issues faster. Relating network latency with number of hops gives a better perspective of hits at backend servers. Distributed system benchmarking on throughput, latency and requests are drawn through the black-box method. During black box testing, proxies are used to aggregate and analyse all the logs instead of scanning through individual log files on the backend storage.

Fig 9.1: Resilience, Observability, and Performance at Infrastructure Scale

9.1.2. Performance Testing and Benchmarking

Information technology that provides services to infrastructure operators takes many forms, with widely different characteristics, daily usage patterns, and failure modes. Supporting such a heterogeneous environment requires extensive testing and benchmarking, followed by runtime performance engineering, both of which must take into account user-defined service-level agreements (SLAs) specifying acceptable latency and throughput. Performance testing and benchmarking share many goals, yet they are distinct activities deserving of separate focus. Testing occurs through constant load and periodically increased stress, whereas benchmarking executes workloads with known characteristics over limited periods of time.

All systems must be tested under regular use to ensure that they meet the expectations of their owners and users. The simplest way to devise such tests is to redefine them as service-level guarantees: targets defined by the ultimate users of the service. Such tests can help quantify the service's current capability and detect performance regressions introduced by code changes, configuration changes, or insufficient resource allocation. Periodically increasing the load beyond normal levels can ensure that the service can handle unexpected surges; that latency remains acceptable even under extreme load; and that the service recovers gracefully from resource exhaustion.

9.2. Foundations of Resilience in Large-Scale Infrastructure

The wealth of definitions and concepts surrounding resilience cannot rule out the impressions one gains from observing resilient behaviour in systems, without the need for formal specification or theoretical underpinnings. Resilience largely manifests itself in the fabric of a system as certain attributes or properties, whose interconnected nature may not lend itself to the systematic reductionist approach of defining each in isolation. There are, however, key constraints or principles, among others that lend themselves well to definition, which form the foundations for resilience. These are a comparatively simple need for fault tolerance or recoverability, a need for sound capacity planning* in order to ensure that the demand on resources does not exceed availability, a demand for readily accessible and useful telemetry that can alert to developing or imminent problems, and a need for performance engineering in order to ensure suitable throughput at a reasonable cost. Having sufficient definition or specification of a system to enable accurate simulation of these aspects is equally important.

Aside from preserving real options for relatively rapid recovery from the majority of potential failures via a well-thought-out and architected disaster recovery plan, fault tolerance generally refers to the degree to which a system can continue to function in the presence of component failures. It may be defined in terms of repair time, the expected

time to repair failed components thus restoring full operation, or} degraded operational behaviour reflecting the continued ability to meet business objectives even under failure conditions. Indeed, fault tolerance is sometimes considered in the broader context of recovery, with complete restoration of normal service as the primary goal, while fault tolerance is then viewed as achieving a useful degree of service during component failures – for example, reduced response speed.safe on-call depth.

Fig 9.2: Foundations of Resilience in Large-Scale Infrastructure

9.2.1. Definitions and Concepts

While the commonly adopted terminology of infrastructure resilience is very much shaped by the infrastructure that provides it, at least three insightful definitions of resilience at large scales have been proposed:

Security was defined in one of Abadi's LISA keynote talks as follows: "finally, security is a property of a system that allows it to continue operating normally despite the presence of an adversary; these adversaries are assumed to be not only knowledgeable about the key and protocol, but also much richer than we are and to occupy a privileged situation (for example, they can launch crypto-systems weaker than ours). This is a property that, for instance, the telephone system does not have." Resilience may be defined similarly: a system is resilient if it can withstand any type of modulation without becoming totally non-operational. The modulations are not only the ones due to the failure of system components, but also the ones triggered by attackers working with the specific aim of making the functionality of the system break. Modulation can be

smoothed during the system's lifetime by introducing redundancy, which is costly but sometimes certainly needed.

The second definition of resilience was proposed in a paper presented at HPTS'07 and reads: "Resilience is the ability of a system to tolerate bad events. A bad event may be a failure, an attack, or some other disturbance. A disturbance can be natural, accidental, or intentional." Resilience is very close to fault tolerance. Both terms defined the concept of fault tolerance long before the term fault tolerance was coined. H. M. R. T. follows the general practice of calling fault tolerance what is tolerated when the disturbance is not intentional, and resilience what is tolerated when the disturbance is intentional.

9.2.2. Fault Tolerance and Recoverability

Large-scale infrastructures must remain adaptable and maintain service continuity despite production faults of any kind. The key question a resilient design must answer is: how fast can a service recover from a fault? A fault may crush a system's performance, bringing the service close to a state of denial-of-service; quickly restoring performance back to, or near to, its normal operating condition is key for resilience. The metrics of fault tolerance are therefore time-centric—recovery time and meanwhile service unavailability—and the strategies for design involve both planning ahead so that faults can be quickly detected and also planning systems with redundancy or similar characteristics that allow them to be repaired quickly. In short, attention must be directed to Fault Tolerance and Recoverability.

Fault Tolerance specifically addresses withstandability under fatal production faults. Services that provide strong quality-of-service guarantees often include a layered redundancy where the upper layers can be sacrificed during catastrophic events and where new instances can be provisioned automatically when the fault of a layer is detected. Similarly, the ability to discover and take advantage of nearby instances in failure domains is often critical for fast recovery (e.g. nearby CDN caches exploiting DNS resolution failures). Even for services not providing such strong guarantees, resiliency can still be improved by ensuring that early detection leads to speedy isolation and repair of faulted elements.

9.2.3. Capacity Planning and Demand Forecasting

Infrastructure systems frequently support services with unanticipated peaks in load and no opportunity for checkpointing or persisting state during the load. Such increases can create damage via data loss or corruption, significant increase in response times, and cost through increased resource allocation. Such overload failures can be tolerated if capacity

planning, empirically informed demand forecasting, or active demand management are successful.

Capacity planning aims to define the resource capability needed to meet expected demand with acceptable risk. Demand forecasting within a planning horizon can be accomplished either using tracking-based techniques that are often little more than anti-persistence fitted curves or more data-intensive regressions. Demand management techniques attempt to change the distribution of load over time rather than changing present capacity usage. Ideally, demand management not only flattens periods of peak usage but also reduces total usage during peak periods, often through incentives.

In-level caches, widely used in microprocessor designs, can also be used to mask hardware failure in large-scale systems. The idea is quite simple: often-requested data is cached closer to the CPU where it can be accessed more quickly. Hardware caches exploit locality of reference; if data recently fetched is replaced by new data it's likely the new data will also be accessed shortly thereafter. In large-scale systems the essence of the same problem arises. Frequently accessed data represents a relatively small subset of the total data set but is still quite large, on order-of-magnitude 1TB. Requests for cached data are also more likely to be serviced successfully because of the underlying structure of the user base; certain classes of lookup queries are common to all. Hence loading popular data into a more accessible faster cache-layer can reduce response times and improve throughput.

9.3. Observability Architectures for Scale

Informing a large-scale service's operational management entails building observability, the capability to monitor whether a system is behaving correctly. Such observability can be delivered through three different types of instrumentation. Brain signals are recorded by different senses that populate environments with observable scalar properties: visual, auditory and tactile. Information theory identifies these signal sources as telemetry. For a visual representation of what's happening, images are collected from cameras; acoustic waves are captured by microphones; and for the tactile sense, the temperature of an object might be measured. A distributed system can also utilize additional signal sources such as tracers and metrics. The data collected through these videos, lectures, temperatures, and words are the telemetry of candidates in an environment. However, telemetry alone is not useful information: in a painting, each pixel has some color, but without combining all pixel colors into a holistic interpretation, a person's mind will see only noise. Telemetry needs to be combined with data created by people's minds in order to produce new information. Telemetry collects a growing amount of signals of different types that need to be managed by pipelines in a scalable way. These pipelines can then

process, analyze, summarize, enrich and store the data to be used for managerial purposes.

Observability encompasses the infrastructure necessary to use these additional types of telemetry: distributed tracing, timing measurements, aggregated metrics and activity logs. Distributed tracing provides the ability to follow user requests as they traverse the different entities of a distributed system. Timing measurements capture the response time associated with a broader set of specific operations that are usually internally triggered within each entity of the system. Aggregated metrics signal the degree of utilization of different resources and high-level summary information regarding redundancy and failures. Finally, activity logs provide a narrative of important events that were triggered by the system.

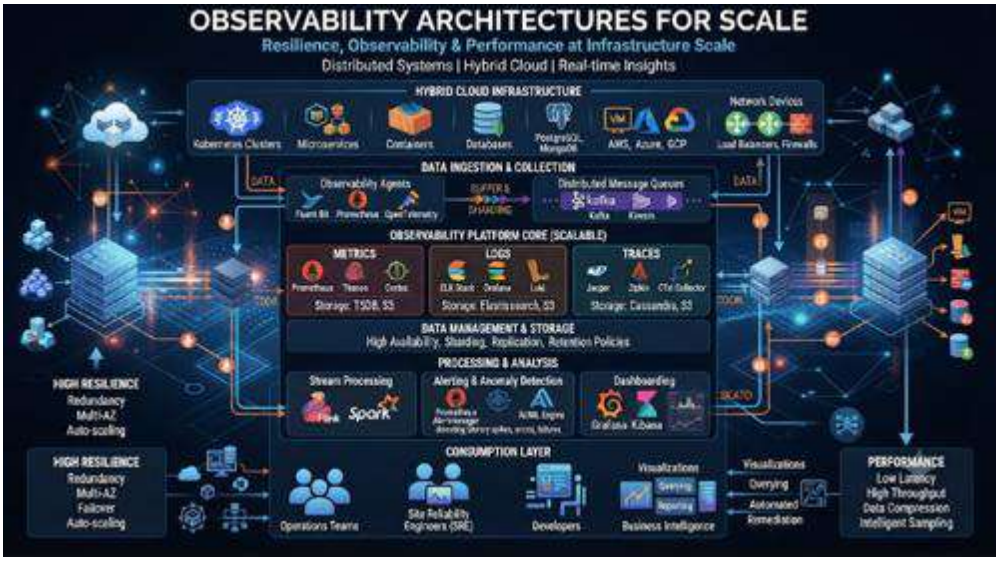

Fig 9.3: Observability Architectures for Scale

9.3.1. Instrumentation and Telemetry

Robust and efficient telemetry is a cornerstone of observability in large-scale systems. Any instrumentation at massive input and output scales must be undertaken judiciously, as it has the potential to impact system performance and reliability. Often, one of the biggest challenges of adding enhanced instrumentation lies within the collection, processing, and analysis of the telemetry data. In addition to the computational load added by the new data source, ingestion points become more critical as their impact on end-to-end performance and reliability increases with scale. The following are some key aspects of instrumentation and telemetry that have proven to greatly improve the robustness and efficacy of large infrastructures.

At large scale, a single set of logs, a set of metrics, and distributed traces do not convey a complete picture of the system's health. Instead, embedded telemetry should be deliberately separated to produce independent sources of truth, with a separate set of log-like structures dedicated to telemetry and separated from the core functionality. These log-like telemetry sources can then be sampled at a different rate from the core system, providing the necessary granularity while minimizing the impact of the instrumented collection. The activations of profile- or sample-based telemetry should also be chosen carefully with consideration of their location within the control flow to minimize the risk of covert side channels and inadvertent bias. In environments with an easily-controlled separation of traffic or behavior and telemetry, care should be taken to introduce additional system overload conditions to provide coverage for different system regions and workloads during the collection of telemetry.

9.3.2. Distributed Tracing, Metrics, and Logs

Instrumentation of large systems produces vast amounts of telemetry data—traces, metrics, logs—that can be mined for information. Distributed tracing solutions instrument applications to measure request paths through a system. Components spanning large distributed systems often represent broadly different abstractions (e.g., big data frameworks, messaging systems, specialized servers) that evolve independently, or that combine functional use cases in complex ways. Telemetry information on application latency, varying load along request paths, and the interaction between different abstractions can assist in Quality of Service (QoS) management and resource scheduling. Logs, used for dynamic and systematic telemetry as well as audit and service information, need to be collected, processed, archived and searched.

Distributed tracing solutions instrument web services to measure request paths through the system. These systems leverage routing information in requests together with an undertheory of stateful interactions in applications, which causes unrelated services to appear stateful in end-to-end interactions, to relate events observed in different services. Telemetry data can be used for SLAs management, resource scheduling, dynamic partitioning, middleware dimensioning, and for Application Performance Management (APM) solution validation and improvement. Latency profiles reveal application paths that contribute disproportionately to end-to-end user experience. Telemetry on different time scales—from service-level to infrastructure-level—enables Trade-off analysis between various operating currencies, such as monetary cost versus application throughput or latency.

9.3.3. Data Pipelines and Observability data management

An observability architecture generates considerable telemetry data about system operations. To fully realize its benefits, a data management strategy is essential.

Data Pipelines for Large-Scale Ingest

An application complex enough to generate several hundred million traces per day, tens of billions of events per hour across thousands of critical long-lived metrics, and several hundred terabytes of logs within a 30-day retention period must rely on a partitioned ingest pipeline capable of supporting extreme volume. This can be accomplished using a large number of distributed write sharded queues such as Kafka partitions as the primary accelerators collecting and moving time-ordered data. Such first-in-first-out stored queues can be partitioned across available parallelism in the system on timestamp windows, dimensioned for expected volume within these windows under peak conditions, and rapidly auto-scaled elastically if auto-scaling can be translated from the consuming side down to the ingest pipelines. Ingest nodes should be point-of-connection abstractions mapped to the available underlying buffer/message queue storage.

Observability Data Management Strategy

An effective observability data management strategy at scale comprises a complete ingest pipeline with flow-control mechanisms that safeguard other layers from overload or saturation, a second layer designed for bulk processing under super-computing or cluster conditions to digest, filter, transmute, summarize, and batch data at the highest possible throughput with moderate cost, and one or more relatively slower long-term expensive but energy-efficient storage layers. The second data processing layer can be horizontally partitioned with input partitions controlling the number of processing instances and proceedings channels with number of lines processing being the throughput control. The elements of the third long-term storage layer can be relatively low latency and high redundancy.

9.4. Performance Engineering at Scale

Any infrastructure will have limiting throughput and latency associated with it, and understanding these trade-offs at design time and coordinating application use of the infrastructure accordingly can improve performance, resource utilization, and operational costs. Most services face a “quantity goes up, latency goes down” trend across their lifespan, but some are governed by a “quantity goes up, latency goes up” trend due to resource competition. With hyper-scale user demand capacity expansion is done frequently yet usually are simple processes, if care is not taken in planning then these add more costs than value. Autoscaling reduces wasteful over-provisioning in

services but requires sufficient levels of load and demand forecast accuracy for it to have an impact on total cost. Some systems have seasonal behaviours that predictable and are better suited to manual scaling though improved cloud support can help automate it.

Although cloud scaling provides convenient scaling of various hardware resources it is not free. Servers are provisioned with excess memory to support normal operation while the Cloud providers use memory as oversubscription model to reduce costs and increase flexibly but such oversubscription requires monitoring of the metrics and prediction of trends for such events not to result in service interruptions or degraded performance. Multitenancy can decrease costs compared to dedicated VM but the resource contention due to other tenants in the same shared VM must be acknowledged and planned for in any Quality of Service (QoS) commitments. Selecting the right resource type is also important as different hardware types hold different advantages and disadvantages for workloads and using the wrong type could result in network overhead. Usage of Serverless offers different tradeofss and should be considered ultimately the decision comes down to cost.

Fig 9.4: Performance Engineering at Scale

9.4.1. Throughput, Latency, and QoS Trade-offs

Large-scale infrastructure support provisioning of complex functionality to a broad set of customers at affordable prices. Nevertheless, scaling services to meet user requirements often implicitly assumes that quality-of-service (QoS) is also preserved at large scales. In many practical deployments, this is not necessarily true because increased throughput involves increased latency or reduced QoS levels for some users

or a subset of requests. Such trade-offs can be captured explicitly through formal modeling, enabling effective height-matching operations for latency-sensitive requests at ultra-low QoS levels, freeing resources for other requests.

Understanding QoS at scale requires a detailed analysis of how small- and large-scale resources are utilized over time; throughput; and failure rates and latencies. Various deployment techniques further help understand trade-offs between latency, throughput, and QoS. Users, providers, and other interested groups are often concerned about their QoS.

9.4.2. Resource Scheduling and Autoscaling

In the cloud, service requests are frequently dispatched to a large number of instances of a service, ignoring the cost of starting and stopping these instances. In contrast, network routing typically directs incoming requests to only one or a few instances, as these are costlier to provision than the individual requests they process. However, within an individual class of requests, individual latencies are typically less critical than summed throughput or completion time. Different classes of requests may have different Quality of Service requirements, often varying with time of day. These benefits can be exploited without incurring high costs for individual requests by steering requests to “rented” resources at otherwise idle spots in the infrastructure.

In situations where a service is composed of inter-related components where requests cross component boundaries at varying rates, autoscaling techniques can be employed for all components simultaneously. A formal scheduling model for an arbitrary directed graph with stochastic demand was constructed, and showed that in such cases demand may be met at a cost still bounded by O(1) times optimal. Autoscalers based on this model can track demand adaptively with a controller that does not require demand prediction or scaling-characteristic estimation. Such controllers can be implemented for arbitrary components with demand trade-offs between throughput and completion time.

9.5. Conclusion

The foundations of performance, resiliency, and observable architecture at scale are characterized and a number of considerations, concepts, and potential techniques are presented. Large-scale distributed architectures deliver an unprecedented capacity to support real-time services to billions of users. Yet, large-scale distributed systems remain hard; hard to build, hard to maintain, and hard to operate. A multitude of events, including failures, misconfigurations, overload, and natural disasters can threaten operational integrity and system responsiveness. Without appropriate engineering

practice these events can degrade service delivery, introduce downtime, manifest as performance regressions, and mutate into catastrophic failures. Specific considerations and techniques require special attention. Resiliency engineering requires a focus on reducing the effects of faults on the live system and minimizing the latency and resources required to bring the system to a healthy state. Resiliency must be incorporated into the architecture from the ground up. Furthermore, in order to support the demands of test-and-code frequency followed by rapid deployment, continuous observation is critical. The environment, as well as the code running within it, must be continually probed for data that informs architectural health, functional correctness, and responsiveness of critical workflows.

Far from a buzzword synonym for performance, observability encompasses a suite of techniques, patterns, and infrastructure that ensures meaningful operational telemetry is created, collected, processed, and available in a timely manner for live (or very recently concluded) workloads. Controlled performance testing and the appropriate definition of service-level objectives provides hot path observations in lower environments when the live system isn't producing sufficient volumes of traffic to justify probing. Performance remains yet another essential aspect requiring careful attention. Users' quality of experience covers a number of non-functional concerns, the most prominent being throughput, latency, and quality of service delivered, collectively described as performance. Performance engineering encompasses the entire space, from resource scheduling through demand prediction and service-level objective definition down to the architectural shape.

References

Aceto, G., Botta, A., de Donato, W., & Pescapé, A. (2018). Cloud monitoring: A survey. Computer Networks, 57(9), 2093–2115.

Kolla, S. K., Bandi, V. D. V. K., & Meda, R. (2026). Comment on "Predicting self-image satisfaction after adult spinal deformity surgery: a machine learning approach using patient phenotypes." Spine Deformity. https://doi.org/10.1007/s43390-026-01400-3

Basiri, A., Behnam, N., de Rooij, R., Hochstein, L., Kosewski, L., Reynolds, J., & Rosenthal, C. (2016). Chaos engineering. IEEE Software, 33(3), 35–41.

Raghunath Loganathan. (2026). SaaS Entitlement Governance: A Reproducible Model for Detecting and Reclaiming Over-Provisioned Access at Enterprise Scale. Journal of Intelligent Decision Making and Information Science, 3(7s), 2519–2530. https://doi.org/10.59543/jidmis.v3.1957

Padma, G., Reddy, V. A. R., KA, S. D., Buvaneswari, B., & Kumar, K. S. (2026, April). Privacy-Preserved Face Recognition Biometric Authentication Using FaceNet and Zero-Knowledge Proofs for Secure, Access Control on Decentralized Blockchain Networks. In 2026 2nd International Conference on Intelligent Systems and Computational Networks (ICISCN) (pp. 1-8). IEEE.

Dean, J., & Barroso, L. A. (2013). The tail at scale. Communications of the ACM, 56(2), 74–80.

Kolla, S. K., Khaparkar, S., Kumar, D., Yadav, S., Shankar, S., & Shamila, M. (2026, June). Deep Learning Based Real-Time Threat Monitoring for Cloud and IoT-Enabled Healthcare Systems. In 2026 5th OPJU International Technology Conference (OTCON) on Smart Computing for Innovation and Advancement in Industry 5.0 (pp. 1-6). IEEE.

Gan, Y., Zhang, Y., Hu, K., Cheng, D., He, Y., Pancholi, M., & Delimitrou, C. (2019). Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, 19–33.

Gunawi, H. S., Hao, M., Leesatapornwongsa, T., Patana-anake, T., Do, T., Adityatama, J., Eliazar, K. J., Laksono, A., Lukman, J. F., Martin, V., & Satria, A. D. (2014). What bugs live in the cloud? A study of 3000+ issues in cloud systems. Proceedings of the ACM Symposium on Cloud Computing, 1–14.

Bhavani, B. D., SR, S., Loganathan, R., & Nagaraj, S. (2026, April). Evolutionary Gravitational Neocognitron Neural Network, Snow Leopard Optimization and Deep Graph Reinforcement Learning for Routing Protocol in WSN. In 2026 2nd International Conference on Intelligent Systems and Computational Networks (ICISCN) (pp. 1-8). IEEE.

Huang, P., Guo, C., Zhou, L., Lorch, J. R., Dang, Y., Chintalapati, M., & Yao, R. (2017). Gray failure: The Achilles' heel of cloud-scale systems. Proceedings of the 16th Workshop on Hot Topics in Operating Systems, 150–155.

Jayathilaka, H., Krintz, C., & Wolski, R. (2017). Performance monitoring and root cause analysis for cloud-hosted web applications. Proceedings of the 26th International Conference on World Wide Web, 469–478.

Kaldor, J., Mace, J., Bejda, M., Gao, E., Kuropatwa, W., O'Neill, J., Ong, K. W., Schaller, B., Shan, P., Viscomi, B., Venkataraman, V., Veeraraghavan, K., & Song, Y. J. (2017). Canopy: An end-to-end performance tracing and analysis system. Proceedings of the 26th Symposium on Operating Systems Principles, 34–50.

Mace, J., Roelke, R., & Fonseca, R. (2015). Pivot tracing: Dynamic causal monitoring for distributed systems. Proceedings of the 25th Symposium on Operating Systems Principles, 378–393.

Kolla, T. (2026). Multi-Agent AI Framework for Predictive Healthcare Interoperability. International Journal of Multidisciplinary Research in Science, Engineering, Technology & Management, 2(5), 1-15.

Mangalampalli, B. M., Peddi, R. K., Kolla, S. K., Reddy, V. A. R., Mangala, N., & Seenu, A. (2026, June). Explainable Clinical Graph Intelligence Framework for Longitudinal Risk Modeling and Care Pathway Optimization. In 2026 Third International Conference on Innovations in Cybersecurity and Data Science (ICICDS) (pp. 1-6). IEEE.

Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., & Wilkes, J. (2015). Large-scale cluster management at Google with Borg. Proceedings of the Tenth European Conference on Computer Systems, 1–17.

Wang, T., Zhang, W., Ye, C., Wei, J., Zhong, H., & Huang, T. (2012). FD4C: Automatic fault diagnosis framework for web applications in cloud computing. IEEE Transactions on Systems, Man, and Cybernetics, Part A: Systems and Humans, 42(1), 61–75.

Xu, W., Huang, L., Fox, A., Patterson, D., & Jordan, M. I. (2009). Detecting large-scale system problems by mining console logs. Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, 117–132.

Yuan, D., Luo, Y., Zhuang, X., Rodrigues, G. R., Zhao, X., Zhang, Y., Jain, P. U., & Stumm, M. (2014). Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation, 249–265.