Journal of Computer Science and Technology Studies
ISSN: 2709-104X
DOI: 10.32996/jcsts
Journal Homepage: www.al-kindipublisher.com/index.php/jcsts

JCSTS
AL-KINDI CENTER FOR RESEARCH AND DEVELOPMENT

| RESEARCH ARTICLE

Modern Service Discovery Architectures for Internet-Scale Distributed Systems

Sanjay Singh

LinkedIn Corporation, Mountain View, CA, USA
ORCID: 0009-0007-7953-2083
Corresponding Author: Sanjay Singh, **E-mail**: reachout.sanjaysingh@gmail.com

| ABSTRACT

Distributed applications increasingly cannot know in advance where the services they depend on will be running, since workloads move, scale, and fail over continuously across cloud and container environments. This article examines how service discovery has evolved from a simple endpoint lookup mechanism into a real-time control layer that maintains a continuously updated picture of service health, location, and metadata across a distributed system. It traces the shift from static configuration and manually maintained host mappings toward dynamic registration and continuous state synchronization, and it details the architectural components, service registries, health monitoring subsystems, metadata repositories, and control plane and data plane separation that make this shift operationally practical. Particular attention is given to the scalability, reliability, and fault tolerance demands placed on discovery infrastructure once it operates at internet scale, where every dependent application shares the same fate as the discovery layer beneath it. The analysis argues that discovery cannot be evaluated in isolation from the service mesh and traffic infrastructure typically layered on top of it, since the security and observability mechanisms bundled into modern mesh integration introduce real performance costs that only become visible once discovery and mesh architecture are assessed together rather than as separate concerns. The article also considers how discovery is converging with identity-aware networking, policy-driven communication, and predictive routing intelligence and what these trends imply for how platform teams coordinate discovery, mesh, and orchestration decisions going forward. The central contribution is a case for treating service discovery, service mesh integration, and infrastructure orchestration as a single coordinated architectural concern rather than three independently optimized layers, since only that coordinated view surfaces the tradeoffs that actually determine whether a distributed system scales gracefully or merely appears to.

| KEYWORDS

Service Discovery; Distributed Systems Architecture; Service Mesh; Microservices Orchestration; Fault Tolerance; Internet-Scale Infrastructure

| ARTICLE INFORMATION

ACCEPTED: 15 August 2026 **PUBLISHED:** 23 September 2026 **DOI:** 10.32996/jcsts.2026.8.9.11

1. Introduction

1.1 Service Discovery as a Foundational Architectural Capability

A distributed application rarely knows in advance where the services it depends on will actually be running. A workload can get rescheduled mid-deployment, a new replica can spin up to absorb load, an entire region can fail over to a backup, and hardcoding an address into a config file stops being viable the moment any of that happens. Something has to keep answering a simple but constantly shifting question: where is this service right now, and is it healthy? That is what service discovery does, and as distributed systems have grown from a handful of services to thousands, discovery has stopped being a minor operational convenience and become one of the components the whole platform actually depends on.

1.2 From Endpoint Resolution to a Real-Time Control Layer

Service discovery today does more than resolve a name to an address. It keeps a live, continuously updated picture of which instances exist, whether they are healthy, what version they are running, and what metadata should shape how traffic reaches them. That shift, from a static lookup table to an active control layer, is what lets applications stay resilient while the infrastructure

underneath keeps changing. What follows works through the architectural principles behind that shift, the components that make it work in practice, and the reliability and performance tradeoffs that show up once discovery has to operate at internet scale.

2. Evolution of Service Discovery in Distributed Systems

2.1 Limitations of Static Discovery Mechanisms

Early distributed systems leaned on static configuration files, manually maintained host mappings, and Domain Name System records to let services find each other. That worked fine when deployments changed rarely and infrastructure sat still for months at a time. Virtualization, containers, and elastic cloud infrastructure broke that assumption entirely. Once instances could appear and disappear within minutes, static discovery started generating configuration drift, stale entries, and delayed updates that directly undermined reliability, a failure pattern documented as far back as coordination services built specifically to manage this kind of dynamic membership state (Burrows, 2006).

2.2 Dynamic Registration and Continuous State Synchronization

Modern service discovery swaps those static assumptions for dynamic registration and continuous synchronization instead. A service registers itself the moment it becomes available, reports health on a regular cadence, and deregisters automatically once it can no longer serve traffic. Consensus-based coordination protocols made this automated lifecycle practical at scale since they give a distributed registry a way to converge on one consistent view of membership even while individual nodes fail or drop offline temporarily (Ongaro & Ousterhout, 2014). The wait-free coordination primitives built into early internet-scale coordination services proved that a dynamic registry of this kind could sustain very high read and write volumes without turning into a bottleneck itself (Hunt et al., 2010).

2.3 Standardized Discovery Protocols Predating the Cloud Era

Dynamic discovery is not purely a cloud native invention, and it is worth remembering that. Long before containers and elastic infrastructure became routine, network protocols already existed for advertising and locating services on a local network without manual configuration, built around standard naming and record lookup conventions that let a client find a service by type instead of by a fixed address (Internet Engineering Task Force, 2013). That early work established a principle modern architectures still lean on: discovery should describe what a service offers, not merely where it happens to sit at a given moment. Later systems generalized that idea from local network scope to internet and data center scope, trading broadcast-based lookup for centralized or replicated registries capable of handling far higher request volumes.

2.4 Service Registries as Shared Infrastructure

As discovery matured, purpose-built service mesh and registry platforms emerged specifically to centralize this capability rather than leaving every team to reimplement it on its own. General-purpose service networking platforms bundle registration, health checking, and key-value-based configuration storage into one operational surface, cutting the number of independent systems a platform team has to run and reason about mid-incident (HashiCorp, n.d.-a). This consolidation reflects the same lesson the earlier coordination services taught: a shared, carefully operated registry beats a scattered collection of independently maintained lists once infrastructure outgrows what a small team can track by hand.

3. Core Components of Modern Service Discovery Architectures

3.1 Registry, Health Monitoring, and Metadata Repository

Pull apart a modern service discovery architecture and a fairly consistent set of pieces shows up: a service registry, a health monitoring subsystem, a metadata repository, discovery clients, configuration distribution mechanisms, and traffic routing components. The registry is the authoritative record of what actually exists. Health monitoring narrows that record down to what is currently reachable and correct. The metadata repository carries everything else, deployment version, geographic location, and security posture, that later feeds smarter routing decisions. Distributed key-value stores purpose-built for this kind of critical coordination data have become a common foundation for the registry layer precisely because they are designed to stay consistent and available through partial infrastructure failure (etcd-io, n.d.).

Component	Function	Typical Enforcement Point
Service registry	Holds the authoritative record of what service instances exist	Central or replicated coordination store
Health monitoring subsystem	Determines which registered instances are reachable and correct	Continuous or periodic probing layer

Component	Function	Typical Enforcement Point
Metadata repository	Carries version, location, and security context for routing decisions	Attached to registry entries, queried by clients
Discovery clients	Query the registry to resolve service instances at request time	Embedded in application or sidecar proxy
Configuration distribution mechanism	Propagates routing and policy changes across the fleet	Control plane to data plane channel

Table 1. Core components of a modern service discovery architecture and their enforcement points

3.2 Metadata-Driven Routing Decisions

Metadata is what turns discovery from a lookup mechanism into something closer to an intelligent routing layer. Once a discovery system knows a service instance's version, geographic location, and health status, it can support locality-aware routing, version-based traffic shifting, canary releases, and workload isolation without ever requiring the calling application to understand any of that complexity itself. Configuration distribution protocols standardized across modern proxy ecosystems make this metadata propagation practical at scale, letting an entire fleet of proxies converge on a consistent routing picture within seconds of a change (Envoy Proxy Project, n.d.-a).

3.3 Registry Consistency Models and Their Tradeoffs

Not every registry needs the same consistency guarantees, and picking the wrong model for a given workload is a common architectural mistake. A registry backing critical coordination state, leader election, distributed locks, and configuration that must never be read inconsistently generally needs strong consistency even at some latency cost, which is exactly the design point distributed key-value stores built for this purpose optimize around (etcd-io, n.d.). A registry backing routine service lookup, on the other hand, can usually tolerate brief staleness in exchange for lower latency and higher read throughput, since a slightly outdated view of healthy instances rarely causes a real problem as long as health checking eventually corrects it. Sorting state into the right category, rather than using the strongest available consistency model by default, is one of the more consequential decisions a discovery architecture makes.

3.4 The Data Plane and Control Plane Split

A recurring pattern in mature discovery architectures is the split between a control plane, which computes routing and configuration decisions, and a data plane, which actually enforces those decisions against live traffic. Keeping the two separate means the control plane can evolve on its own schedule while a large, heterogeneous fleet of data plane proxies continues enforcing whatever it last received, without every proxy needing to reimplement discovery logic from scratch (Envoy Proxy Project, n.d.-b). Security-relevant configuration, certificate distribution, access policy, and identity verification tend to follow the same split once a service mesh enters the picture, an approach that has become standard practice across mesh implementations generally (Cloud Native Computing Foundation, 2018).

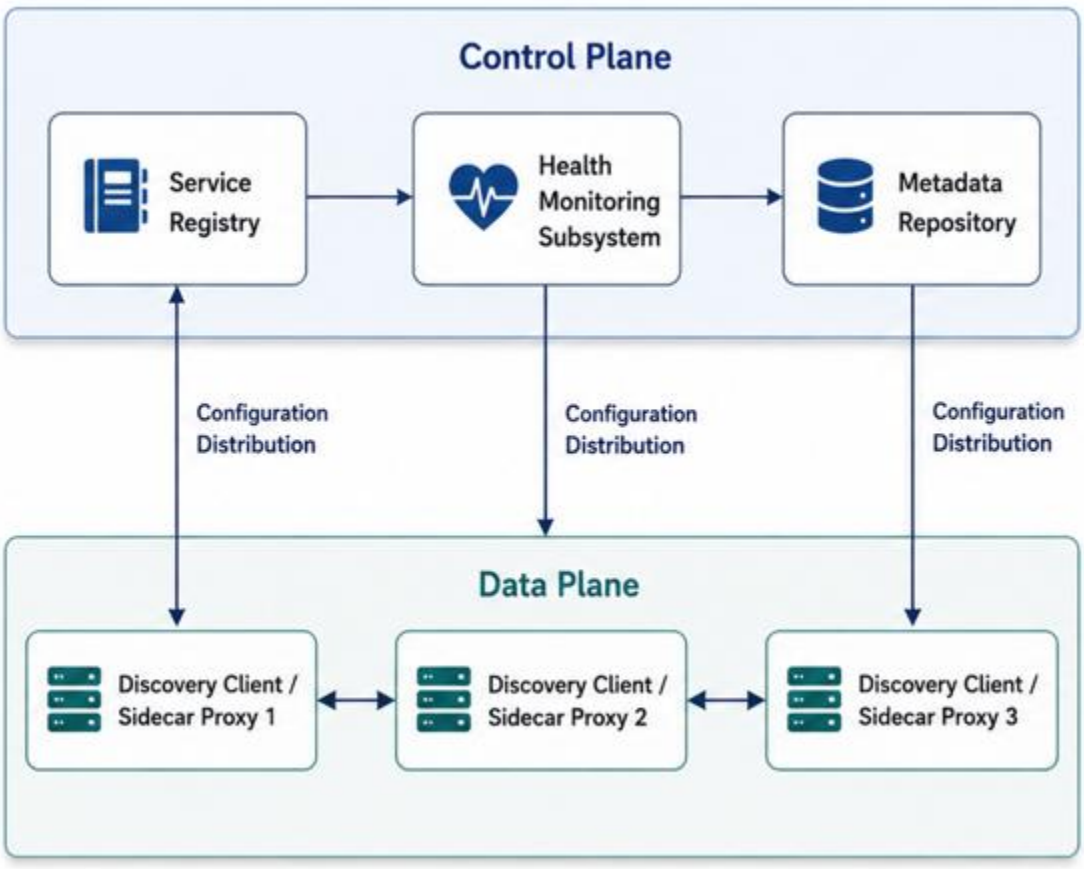

Figure 1. A modern service discovery architecture separates the control plane, where routing and configuration decisions are computed, from the data plane, where those decisions are enforced on live traffic.

4. Scalability, Reliability, and Fault Tolerance

4.1 High Availability Requirements at Discovery Scale

Service discovery has to run with exceptionally high availability, since every other distributed application sits on top of it. Large environments can generate millions of discovery lookups alongside a constant stream of health updates and registration events from thousands of workloads at once. Meeting that load while keeping response latency low takes efficient replication, distributed caching, and carefully chosen consistency models, the same architectural concerns that motivated early cluster coordination systems built to manage locking and configuration at very large scale (Burrows, 2006).

4.2 The Cost of Getting Discovery and Mesh Integration Wrong

Fault tolerance matters just as much as availability, since a failure inside discovery infrastructure propagates outward almost immediately to every dependent application. What gets discussed less often is that the mechanisms used to secure and observe service-to-service communication, frequently layered on top of discovery through a service mesh, carry a real performance cost that has to be accounted for rather than waved away. Measured benchmarks of sidecar-based service mesh proxying found request latency increasing by up to 185% in gRPC mode compared to a direct, mesh-free path (Zhu et al., 2022). Enabling mutual TLS across service mesh sidecars has, under certain configurations, been shown to cause a throughput drop of up to 95% relative to an unencrypted baseline (Bremler-Barr et al., 2024). Neither figure is a corner case worth waving away as a rounding error; both represent overhead that discovery and mesh architects genuinely need to design around.

4.3 Distributed Data Plane Configuration at Scale

Configuration distribution earns its own place in a fault tolerance discussion, distinct from the registry itself. Health-aware routing built directly into a general-purpose service networking platform's data plane lets individual proxies make local failover decisions without waiting on a round trip to a central control plane during an active incident, which meaningfully shortens the window where traffic keeps flowing to an already failed instance (HashiCorp, n.d.-b). Coordination frameworks originally built for internet scale membership and configuration management remain a common foundation underneath these data plane components precisely

because they were designed from the start to tolerate partial network failure without losing consistency (Apache Software Foundation, n.d.).

5. Integration with Traffic Infrastructure and Service Orchestration

5.1 Coordinating with Load Balancers, Proxies, and API Gateways

Service discovery rarely operates alone. It integrates with load balancers, reverse proxies, and API gateways to turn real-time infrastructure state into actual routing decisions. API gateways in particular have taken on an expanded role in this integration, providing centralized authentication, rate limiting, and protocol translation on top of the routing intelligence discovery supplies, which cuts the amount of cross-cutting logic individual services need to implement on their own (Neelan, 2025). Cloud provider load balancing services extend the same principle globally, using health and capacity signals to steer traffic toward the most appropriate backend across regions (Google Cloud, n.d.).

5.2 Automated Lifecycle Integration with Orchestration Platforms

Orchestration integration is what actually closes the loop between deployment and discovery in practice. Startup and shutdown become discovery events rather than manual configuration steps: a workload announces itself the instant it comes online, and once it stops passing health checks, it simply drops out of the routing picture with no operator intervention required. Kubernetes captures this relationship natively in its networking model, giving discovery clients and proxies service and endpoint abstractions to consume without needing any awareness of the scheduler underneath (Kubernetes Project, n.d.-a). More granular endpoint representations built on that same networking model let a cluster track individual pod readiness at a finer resolution than a single service-level abstraction would support, which starts to matter considerably once a service scales past a few dozen replicas and readiness state keeps changing (Kubernetes Project, n.d.-c). Cluster management systems built to operate at a very large scale demonstrated early on that treating scheduling, health, and networking as one coordinated system, rather than three separately operated pieces, produces meaningfully more reliable outcomes than bolting them together after the fact (Verma et al., 2015).

5.3 Data Plane Coordination Across a Heterogeneous Fleet

A real production fleet also rarely runs one uniform version of any given proxy or client library at any given time, and orchestration integration has to reckon with that directly. Rolling upgrades, staged rollouts, and mixed version fleets during a migration all mean discovery and its downstream data plane have to tolerate version skew gracefully instead of assuming every participant speaks the exact same protocol version. Load balancing platforms operated by major cloud providers handle a comparable problem at global scale, continuously reconciling capacity and health signals across regions that may be running different infrastructure generations at the same time (Google Cloud, n.d.).

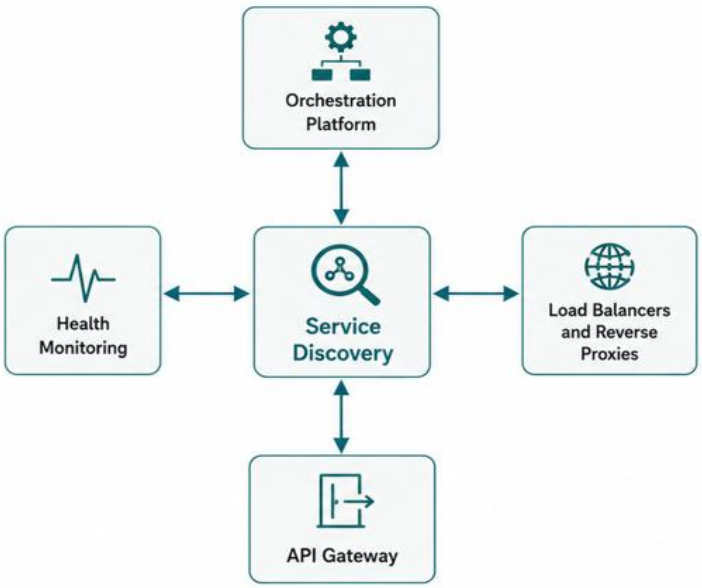

Figure 2. Service discovery operates as a hub coordinating with orchestration platforms, load balancers, API gateways, and health monitoring rather than functioning as an isolated capability.

6. Future Trends in Service Discovery

6.1 Predictive and Adaptive Routing Intelligence

The next generation of service discovery is likely to bring in predictive signals rather than simply reacting to health checks after the fact. Rather than checking only whether an instance currently passes a health probe, a future system could continuously weigh historical traffic patterns, resource trends, and application-level performance signals to catch problems before they turn into visible failures. Site reliability practice has been moving in that same proactive direction for years now, away from purely reactive incident response and toward anticipating capacity and failure problems before they surface (Beyer et al., 2016).

6.2 Convergence with Identity-Aware and Policy-Driven Infrastructure

A parallel trend is the merging of discovery with identity-aware networking and policy-driven communication rather than treating them as adjacent but separate concerns. Reference architectures for cloud infrastructure now commonly fold networking, security, and workload identity into a single design concern instead of three systems procured and run independently (Liu et al., 2011), and security guidance written specifically for container orchestration platforms echoes the same conclusion from the opposite direction, arguing that discovery and identity enforcement work best designed together rather than bolted on after the fact (Kubernetes Project, n.d.-b).

6.3 Observability as a Prerequisite for Autonomous Discovery

Every predictive or autonomous capability described above depends entirely on the quality of the observability data feeding it, a dependency that is easy to underestimate. Blind spots in telemetry translate directly into blind spots in prediction: a discovery system cannot anticipate a failure condition it never had visibility into in the first place, and an autonomous decision made against incomplete data can end up worse than a simple, well understood static rule would have been. Broadly adopted instrumentation standards give a heterogeneous fleet of services and proxies a common, comparable telemetry format instead of leaving every team to invent its own (OpenTelemetry Project, n.d.). Autonomous decision-making inside discovery only becomes trustworthy once the telemetry underneath it is treated as seriously as the registry itself.

6.4 Multi-Cloud and Edge Expansion

Organizations increasingly run services across more than one cloud provider and, in some cases, edge locations positioned close to end users rather than a handful of centralized regions. This expansion multiplies the number of environments a discovery system has to stay consistent across, and it raises the stakes on cross-environment latency, since a discovery lookup that has to cross a slow or unreliable link between providers introduces exactly the kind of delay that defeats the point of low-latency service resolution. Future discovery architectures will most likely have to build multicloud and edge distribution in from the start as a core design constraint, rather than patching it onto a design that originally assumed a single data center.

7. Methodology

7.1 Analytical Approach

The approach taken here is qualitative and architectural, not experimental. Its claims trace back to direct operational experience designing and operating service discovery infrastructure at internet scale, checked against publicly available standards documentation, vendor architecture references, and peer-reviewed measurement studies of the performance characteristics discussed in Sections 4 and 6.

7.2 Evidence Base and Boundaries

Where a claim describes a specific measured outcome rather than a general architectural principle, that outcome is attributed directly to its source study by name and year instead of presented as a figure the reader should expect to reproduce in their own environment, since measured overhead shifts with hardware, protocol choice, and workload shape. Observability tooling standardized across the industry provides the instrumentation baseline referenced throughout this analysis for how discovery and mesh behavior actually gets measured in practice (OpenTelemetry Project, n.d.).

8. Discussion and Practical Implications

8.1 Discovery as Shared-Fate Infrastructure

Because every dependent service relies on the same discovery layer, a failure or performance regression there has shared fate consequences across the whole application ecosystem rather than staying contained to one team's service. That is precisely why the overhead figures discussed earlier, the latency and throughput costs measured in sidecar-based mesh deployments, deserve more architectural attention than they typically get. Choosing a sidecarless mesh architecture over a traditional sidecar deployment has shown as little as an 8% latency increase under mutual TLS, compared to 166% for a conventional sidecar based deployment carrying the same load (Bremler-Barr et al., 2024), which is exactly the kind of tradeoff that only becomes visible once discovery and mesh integration get evaluated together rather than as separate concerns. Treat discovery as a solved, low-level utility and

mesh integration as a later, unrelated concern, and the risk is locking in an architecture whose combined overhead was never actually assessed as a whole, only piecemeal at the moment each component happened to get adopted.

8.2 Operational Lessons for Platform Teams

How much operational discipline the surrounding practices demand is something platform teams operating discovery at scale routinely underestimate. It is runbooks, postmortem review, and a disciplined on call rotation that actually shorten mean time to resolution and cut repeat incidents when discovery infrastructure breaks, not the choice of registry technology by itself. This distinction between the technology and the practice around it is not new: the early coordination services now treated as foundational examples of this kind of infrastructure made the point directly: the technology makes reliability possible while operational discipline is what actually delivers it (Hunt et al., 2010).

8.3 Choosing a Consistency Model Deliberately, Not by Default

Platform teams frequently inherit a discovery architecture's consistency model rather than choosing it deliberately, adopting whatever default a particular registry technology ships with rather than evaluating whether that default actually fits the workload at hand. Given the measured cost differences between sidecar and sidecarless mesh architectures under otherwise identical load, treating discovery and mesh consistency choices as a one-time technology selection rather than an ongoing architectural decision leaves real performance on the table. Reassessing that choice periodically, especially as workload characteristics shift or as sidecarless alternatives mature, is a low-cost practice few platform teams currently treat as a standing item in their architecture review process.

8.4 Toward Coordinated Discovery, Mesh, and Orchestration Design

Perhaps the single clearest practical takeaway is that discovery, mesh, and orchestration decisions should never sit with three separate teams each optimizing for its own metric in isolation. An interaction effect between a discovery choice and the mesh layer's security enforcement, or between discovery and the orchestration platform's rollout cadence, can quietly erase whatever efficiency gain looked promising on paper for the discovery layer alone. Coordinating these three areas deliberately, even through informal shared architecture review rather than a formally unified team, tends to surface that kind of interaction well before it reaches production rather than during an active incident.

8.5 Limitations of This Analysis

The approach taken here is architectural rather than experimental, and it is worth being explicit that the overhead figures pulled from the measurement literature reflect the specific benchmark conditions of those studies rather than universal constants anyone can assume will hold everywhere. Hardware generation, network topology, request size distribution, and the particular mesh or discovery implementation in play will all shift actual overhead in a given production environment away from any single published number. The cited figures are best read as evidence that these tradeoffs are real and worth measuring directly, not as values to carry over uncritically into an unrelated deployment.

9. Conclusion

This analysis set out to work through what modern service discovery actually has to provide at internet scale, beyond the basic function of turning a name into an address. The discovery has become a real-time control layer combining a registry, health monitoring, metadata-driven routing, and tight integration with orchestration and traffic infrastructure, and none of those pieces function adequately on their own. Each piece closes a specific gap a naive implementation would otherwise leave open: coordination protocols keep membership state consistent under failure, deliberate consistency model choices trade latency against correctness where each actually matters, and separating the control plane from the data plane lets routing logic evolve without touching every proxy enforcing it. Much of the existing writing on service discovery treats it as a problem that a registry and health checks solve once they exist. The case made here runs differently: discovery cannot be judged in isolation, and its real operational cost only becomes clear when it is considered alongside the service mesh and traffic infrastructure that sit on top of it, because the security and observability mechanisms typically bundled with mesh integration introduce latency and throughput costs that the measurement literature reviewed in Sections 4 and 8 documents directly. What separates infrastructure that scales gracefully from infrastructure that merely appears to is whether discovery, mesh integration, and orchestration get treated as one coordinated system or as three layers each optimized on its own.

Nothing in the architectural principles laid out here is meant to substitute for measuring actual behavior inside a specific production environment. The literature cited throughout makes clear that discovery and mesh integration choices carry real, sometimes substantial, performance consequences, though exactly how large those consequences turn out to be depends on local conditions that no general analysis can fully predict in advance. A platform team that makes a habit of measuring these tradeoffs directly, instead of assuming a published benchmark number will transfer cleanly onto its own infrastructure, ends up better positioned to make discovery and mesh decisions that actually hold up under its own production load.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

Publisher's Note: All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers.

References

[1] Apache Software Foundation. (n.d.). Apache ZooKeeper documentation. https://zookeeper.apache.org/doc/current/

[2] Beyer, B., Jones, C., Petoff, J., & Murphy, N. R. (2016). Site reliability engineering: How Google runs production systems. O'Reilly Media. https://www.oreilly.com/library/view/site-reliability-engineering/9781491929117/

[3] Bremler-Barr, A., Lavi, O., Naor, Y., Rampal, S., & Tavori, J. (2024). Performance comparison of service mesh frameworks: The mTLS test case. arXiv preprint arXiv:2411.02267. https://arxiv.org/abs/2411.02267

[4] Burrows, M. (2006). The Chubby lock service for loosely-coupled distributed systems. Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI). https://research.google/pubs/the-chubby-lock-service-for-loosely-coupled-distributed-systems/

[5] Cloud Native Computing Foundation. (2018). Service mesh: From technical selection to best practice [Whitepaper]. https://www.cncf.io/wp-content/uploads/2020/08/rfma-servicemesh-cncf.pdf

[6] Envoy Proxy Project. (n.d.-a). xDS REST and gRPC protocol. Envoy Documentation. https://www.envoyproxy.io/docs/envoy/latest/api-docs/xds_protocol

[7] Envoy Proxy Project. (n.d.-b). Envoy architecture overview. Envoy Documentation. https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/arch_overview

[8] etcd-io. (n.d.). etcd: Distributed reliable key-value store for the most critical data of a distributed system. https://etcd.io/

[9] Google Cloud. (n.d.). Cloud Load Balancing overview. Google Cloud Documentation. https://cloud.google.com/load-balancing/docs/load-balancing-overview

[10] HashiCorp. (n.d.-a). Consul architecture. Consul Documentation. https://developer.hashicorp.com/consul/docs/architecture

[11] HashiCorp. (n.d.-b). Consul data plane architecture. Consul Documentation. https://developer.hashicorp.com/consul/docs/architecture/data-plane

[12] Hunt, P., Konar, M., Junqueira, F. P., & Reed, B. (2010). ZooKeeper: Wait-free coordination for Internet-scale systems. Proceedings of the 2010 USENIX Annual Technical Conference. https://www.usenix.org/legacy/event/atc10/tech/full_papers/Hunt.pdf?ref=samueleresca.net

[13] Internet Engineering Task Force. (2013). RFC 6763: DNS-based service discovery. https://www.rfc-editor.org/rfc/rfc6763.html

[14] Kubernetes Project. (n.d.-a). Service. Kubernetes Documentation. https://kubernetes.io/docs/concepts/services-networking/service/

[15] Kubernetes Project. (n.d.-b). Cloud native security and Kubernetes. Kubernetes Documentation. https://kubernetes.io/docs/concepts/security/cloud-native-security/

[16] Kubernetes Project. (n.d.-c). EndpointSlice. Kubernetes Documentation. https://kubernetes.io/docs/reference/kubernetes-api/discovery/endpoint-slice-v1/

[17] Liu, F., Tong, J., Mao, J., Bohn, R. B., Messina, J. V., Badger, M. L., & Leaf, D. M. (2011). NIST cloud computing reference architecture (NIST Special Publication 500-292). National Institute of Standards and Technology. https://www.nist.gov/publications/nist-cloud-computing-reference-architecture

[18] Neelan, A. (2025). The evolving role of API gateways in scalable microservices architecture. International Journal of Emerging Trends in Computer Science and Information Technology, 6(4), 4-15. https://doi.org/10.63282/3050-9246.IJETCSIT-V6I4P102

[19] Ongaro, D., & Ousterhout, J. (2014). In search of an understandable consensus algorithm. Proceedings of the 2014 USENIX Annual Technical Conference. https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

[20] OpenTelemetry Project. (n.d.). Observability primer. OpenTelemetry Documentation, Cloud Native Computing Foundation. https://opentelemetry.io/docs/concepts/observability-primer/

[21] 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 (EuroSys). https://research.google/pubs/large-scale-cluster-management-at-google-with-borg/

[22] Zhu, X., She, G., Xue, B., et al. (2022). Dissecting service mesh overheads. arXiv preprint arXiv:2207.00592. https://arxiv.org/abs/2207.00592