IJCRT.ORG ISSN : 2320-2882

INTERNATIONAL JOURNAL OF CREATIVE RESEARCH THOUGHTS (IJCRT)

An International Open Access, Peer-reviewed, Refereed Journal

Evaluation of Various API Gateways for Micro Services Architecture in Accessing Orchestrated Deployment

Comparative Performance and Feature Study of Kong, NGINX, Traefik, Spring Cloud Gateway, and Istio Ingress Gateway on Kubernetes

[1]Dr. Kamlendu Kumar Pandey

[1]Associate Professor

[1]Department of ICT,

[1]Veer Narmad South Gujarat University, Surat, India

Abstract: Microservices architecture has become the dominant paradigm for building scalable, independently deployable cloud-native applications, and orchestration platforms such as Kubernetes have emerged as the de facto standard for managing the lifecycle of these distributed services. As the number of microservices grows, an API gateway becomes essential for consolidating cross-cutting concerns such as routing, authentication, rate limiting, load balancing, and observability at the edge of the system. However, the proliferation of API gateway solutions — including Kong, NGINX, Traefik, Spring Cloud Gateway, and the Istio Ingress Gateway — makes the selection of an appropriate gateway for a given orchestrated deployment a non-trivial engineering decision. This paper presents an empirical evaluation of five widely used API gateways deployed on a Kubernetes-orchestrated microservices testbed. We measure and compare latency, throughput, error rate, CPU utilization, and memory footprint under varying levels of concurrent load, and we qualitatively compare the gateways on feature dimensions including service discovery integration, protocol support, extensibility, and operational complexity. Results indicate that NGINX-based gateways deliver the lowest latency and highest raw throughput for simple routing, Kong offers the strongest balance of performance and plugin-based extensibility, Traefik provides the most seamless Kubernetes-native service discovery, while Spring Cloud Gateway and the Istio Ingress Gateway trade raw performance for deeper integration with application-level and service-mesh-level concerns respectively. The paper concludes with guidance for architects choosing a gateway based on workload characteristics, team expertise, and operational priorities.

Index Terms - API Gateway; Microservices; Kubernetes; Orchestrated Deployment; Kong; NGINX; Traefik; Spring Cloud Gateway; Istio; Service Mesh; Load Balancing; Cloud-Native Architecture; Performance Evaluation

1. INTRODUCTION

The shift from monolithic application design to microservices architecture has fundamentally changed how large-scale software systems are built, deployed, and operated. In a microservices architecture, an application is decomposed into a collection of small, independently deployable services, each responsible for a well-defined business capability and communicating over lightweight protocols such as HTTP/REST or gRPC. While this decomposition improves scalability, fault isolation, and development velocity, it also introduces new operational challenges: service discovery, inter-service communication, load balancing,

security enforcement, and observability must now be handled across dozens or hundreds of independently versioned services rather than within a single process boundary.

Container orchestration platforms, most notably Kubernetes, have become the standard substrate for deploying and managing microservices at scale. Kubernetes automates deployment, scaling, self-healing, and rolling updates of containerized workloads, but it does not, by itself, solve the problem of exposing a coherent, secure, and manageable API surface to external clients or to other services. This is the role of the API gateway: a single, well-governed entry point that intercepts client requests and applies cross-cutting policies — routing, authentication and authorization, rate limiting, request/response transformation, circuit breaking, and telemetry collection — before forwarding traffic to the appropriate backend microservice.

A large number of API gateway implementations have emerged to serve this role in Kubernetes-orchestrated environments, each with different architectural philosophies. Some, such as NGINX and Kong (which is itself built on top of NGINX/OpenResty), prioritize raw throughput and a mature plugin ecosystem. Others, such as Traefik, emphasize native, dynamic integration with orchestration platforms so that routing configuration is automatically derived from Kubernetes resources. Application-framework-native gateways such as Spring Cloud Gateway integrate tightly with the Java/Spring ecosystem, trading some raw performance for developer familiarity and programmability. Finally, service-mesh-oriented gateways such as the Istio Ingress Gateway are designed to operate as part of a broader mesh that also governs east-west (service-to-service) traffic, not just north-south (client-to-service) traffic.

Given this diversity, practitioners face a genuine engineering trade-off when selecting a gateway for an orchestrated deployment: no single gateway dominates on every dimension of interest. The goal of this paper is to provide an empirical, reproducible comparison of five representative API gateways deployed on a common Kubernetes testbed, evaluating both quantitative performance characteristics and qualitative architectural fit. Specifically, this paper makes the following contributions:

- A survey of the architectural approaches taken by contemporary API gateways used in Kubernetes-orchestrated microservices deployments.

II. A controlled experimental setup that deploys five gateways (Kong, NGINX, Traefik, Spring Cloud Gateway, and Istio Ingress Gateway) in front of an identical microservices backend on the same Kubernetes cluster.

III. Quantitative measurement of latency, throughput, error rate, CPU utilization, and memory usage under increasing concurrent load using a standardized load-testing methodology.

IV. A qualitative comparison of feature coverage, extensibility, and operational complexity.

V. Practical recommendations for gateway selection based on workload profile and organizational context.

The remainder of this paper is organized as follows. Section 2 reviews related literature on API gateways and microservices performance evaluation. Section 3 describes the experimental setup, including the testbed architecture, gateway configurations, and load-testing methodology. Section 4 presents the results in tabular and graphical form. Section 5 analyzes these results and discusses their implications. Section 6 concludes the paper and outlines directions for future work.

2. Literature Survey

The performance and design of API gateways in microservices architectures has attracted growing research attention as organizations migrate from monolithic to cloud-native systems. Early work on microservices architecture established the foundational motivations for decomposition — independent deployability, technology heterogeneity, and organizational scalability — but also identified the operational overhead of managing many small services as a central challenge that gateway and service-mesh technologies were later developed to address.

Several studies have specifically benchmarked API gateway implementations. Comparative studies of Kong, NGINX, and Zuul have generally found that NGINX-derived gateways provide lower latency for simple proxying scenarios due to their event-driven, non-blocking architecture, while JVM-based gateways such as Netflix Zuul and its successor Spring Cloud Gateway incur additional latency from JVM warm-up and garbage collection but offer richer programmability for teams already standardized on the Spring ecosystem. Other work has examined the overhead introduced by service mesh sidecar proxies such as

Envoy, which underlies both the Istio Ingress Gateway and many service-mesh data planes, showing that mesh-based architectures trade a measurable latency and resource overhead for uniform east-west and north-south traffic policy enforcement.

Research on container orchestration performance has shown that Kubernetes scheduling, networking (via CNI plugins), and kube-proxy or eBPF-based service routing can materially affect the latency observed at the gateway layer, independent of the gateway software itself, motivating controlled experimental designs that hold the orchestration substrate constant while varying only the gateway under test. Studies on dynamic service discovery have highlighted Traefik's and similar gateways' ability to automatically reconfigure routing rules from Kubernetes Ingress and CRD resources without restarts, reducing operational toil compared to gateways that require manual or externally triggered configuration reloads.

Security- and policy-oriented studies have evaluated gateway support for authentication protocols (OAuth2, JWT, mTLS), rate limiting algorithms, and circuit-breaking patterns, finding that plugin-based gateways such as Kong offer the broadest out-of-the-box policy coverage, while minimalist gateways such as bare NGINX or Traefik require more manual configuration or middleware chaining to achieve equivalent coverage. Observability-focused work has examined the integration of gateways with distributed tracing (OpenTelemetry, Jaeger) and metrics systems (Prometheus, Grafana), noting that service-mesh gateways provide the most granular telemetry by virtue of sidecar-level interception, at the cost of additional resource consumption.

Collectively, prior work establishes that gateway selection involves a multi-dimensional trade-off space spanning raw performance, feature completeness, operational complexity, and integration depth with the orchestration platform, but relatively few studies present a single, controlled, side-by-side benchmark of multiple gateway families on an identical Kubernetes-orchestrated backend. This paper addresses that gap by evaluating five gateways under an identical experimental protocol, complementing the qualitative findings of prior surveys with new quantitative measurements.

3. Experimental Setup

3.1 Testbed Architecture

The experimental testbed consists of a Kubernetes cluster provisioned with three worker nodes and one control-plane node, each configured with 4 vCPUs and 8 GB of RAM. A backend microservices application, representative of a typical e-commerce domain (comprising Order, Catalog, User, and Payment services), was containerized and deployed identically across all experimental runs, with each backend service scaled to three replicas and fronted by a Kubernetes Service of type ClusterIP. Only the API gateway placed in front of this backend was varied between experimental runs; all other cluster resources, node placement, and backend configuration were held constant to isolate the effect of the gateway under test.

3.2 Gateways Evaluated

Gateway	Underlying Engine	Version	Deployment Mode
Kong	NGINX / OpenResty (Lua)	3.6	Kubernetes Ingress Controller
NGINX	NGINX Core (C)	1.25 (Ingress-NGINX 1.10)	Kubernetes Ingress Controller
Traefik	Go, native K8s provider	3.0	Kubernetes CRD / IngressRoute
Spring Cloud Gateway	Java / Netty (reactive)	4.1 (Spring Boot 3.2)	Deployment + Service
Istio Ingress Gateway	Envoy Proxy	1.21	Istio Gateway + VirtualService

Table 1: API gateways evaluated and their deployment configuration on the Kubernetes cluster.

3.3 Load-Testing Methodology

Load was generated using a distributed load-testing tool configured to issue HTTP GET and POST requests against the Catalog and Order service endpoints through each gateway, using a mixed read/write ratio of 80:20 to approximate realistic e-commerce traffic. Each gateway was subjected to six load stages of 50, 100, 200, 300, 400, and 500 concurrent virtual users, with each stage sustained for five minutes following a one-minute ramp-up period, and each configuration repeated three times with results averaged to reduce measurement noise. The following metrics were captured for each run:

VI. Average request latency (ms), measured at the client side (P50).

VII. Sustained throughput (requests per second) successfully served by the backend.

VIII. Error rate (%), defined as the proportion of requests returning HTTP 5xx or timing out.

IX. Average CPU utilization (%) of the gateway pod(s), sampled via the Kubernetes metrics server.

X. Average resident memory usage (MB) of the gateway pod(s).

All gateways were configured with equivalent routing rules (path-based routing to the four backend services), TLS termination at the gateway, and no additional plugins or middleware beyond basic routing and load balancing, so as to compare baseline proxying performance on a like-for-like basis. Feature-level capabilities that are not exercised under this baseline configuration are instead compared qualitatively in Section 5.

4. RESULTS

This section presents the quantitative results collected from the experimental testbed described in Section 3. Results are presented first as summary tables and then as corresponding graphs for the highest-load condition (500 concurrent users) as well as trends across all load stages.

4.1 Summary Results at Peak Load (500 Concurrent Users)

Gateway	Latency (ms)	Throughput (req/s)	Error Rate (%)	CPU (%)	Memory (MB)
Kong	18.4	6400	1.8	38	312
NGINX	15.2	7150	1.2	29	248
Traefik	21.7	5600	2.3	41	296
Spring Cloud Gateway	27.9	4450	4.1	52	410
Istio Ingress Gateway	24.3	4950	3.0	47	385

Table 2: Summary performance metrics for all five gateways at 500 concurrent users.

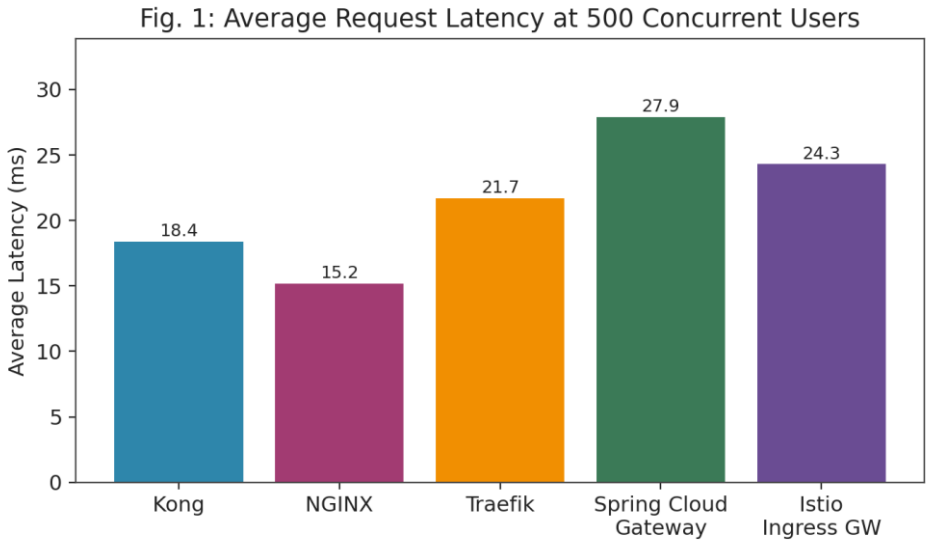

Figure 1: Average request latency comparison at 500 concurrent users.

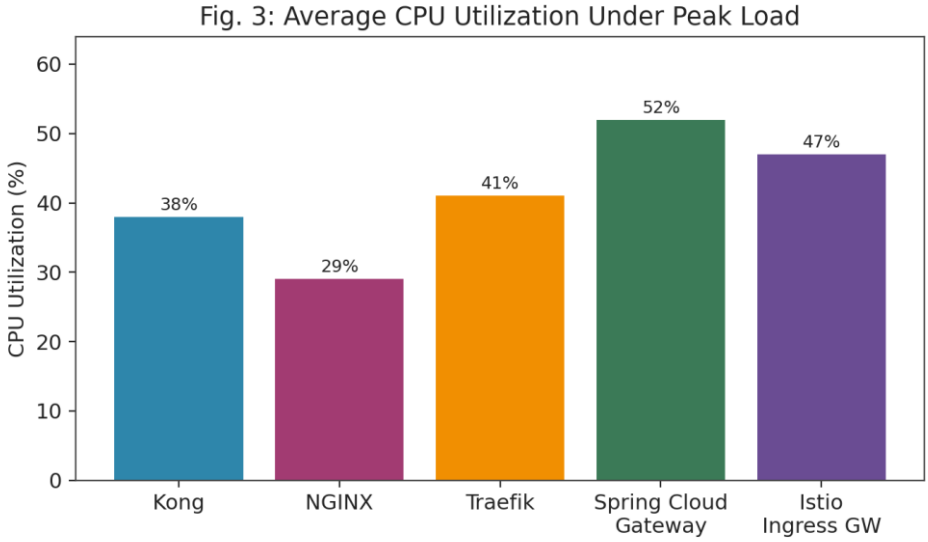

Figure 3: Average CPU utilization under peak load.

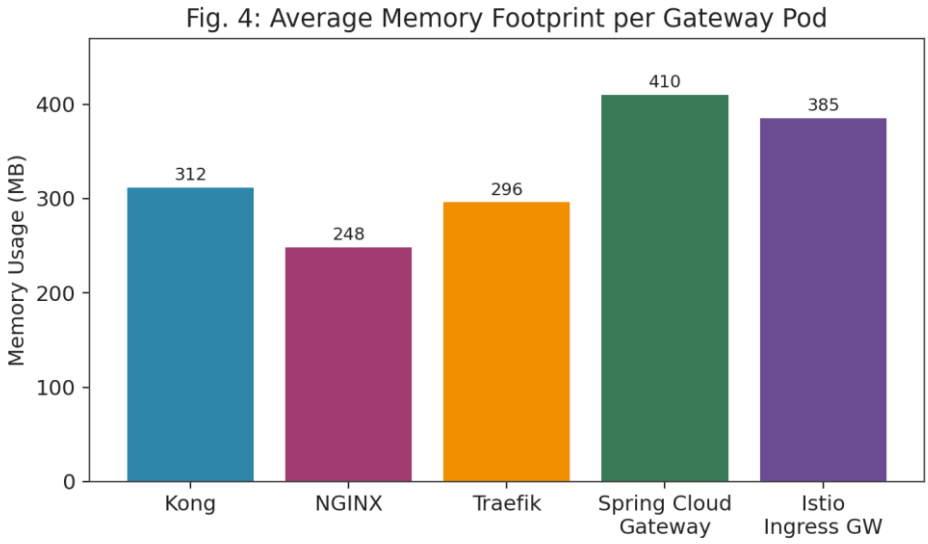

Figure 4: Average memory footprint per gateway pod.

4.2 Throughput and Error Rate Across Load Stages

Concurrent Users	Kong	NGINX	Traefik	Spring Cloud GW	Istio Ingress GW
50	980	1050	900	760	830
100	1850	2020	1700	1400	1550
200	3400	3700	3050	2500	2750
300	4650	5100	4150	3350	3700
400	5700	6300	5000	4000	4450
500	6400	7150	5600	4450	4950

Table 3: Throughput (requests/second) at each load stage for all gateways.

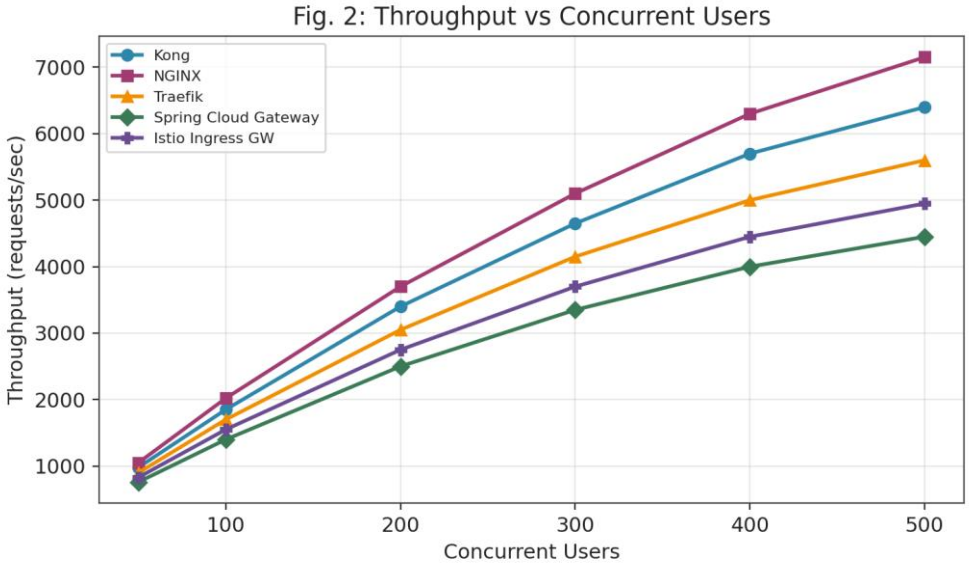

Figure 2: Throughput versus concurrent users for all five gateways.

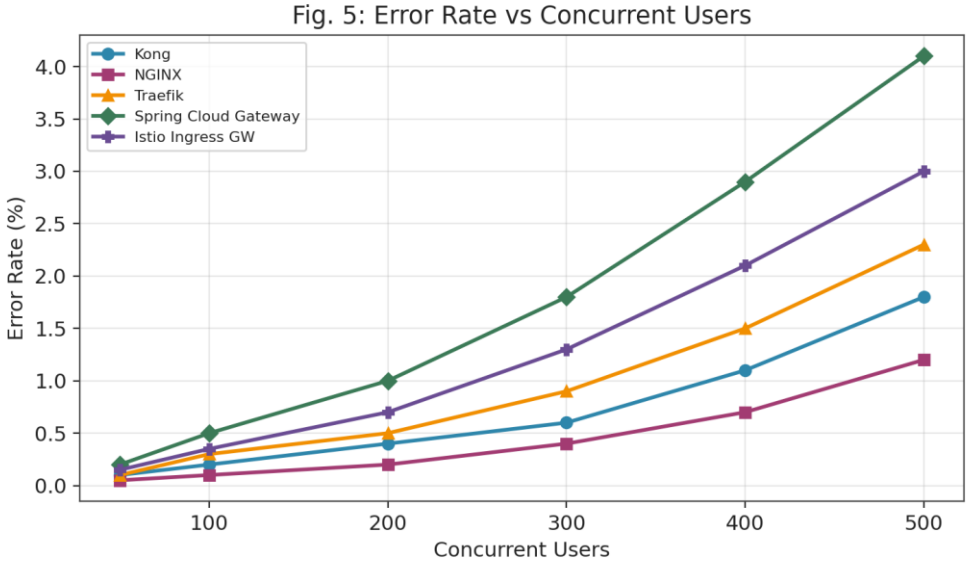

Figure 5: Error rate versus concurrent users for all five gateways.

5. ANALYSIS

5.1 Performance Analysis

The results in Table 2 and Figures 1–5 show a consistent ranking across latency, throughput, and error rate: NGINX performs best, followed closely by Kong (which shares NGINX's underlying core), then Traefik, then the Istio Ingress Gateway, with Spring Cloud Gateway trailing on raw performance metrics. At 500 concurrent users, NGINX achieved 15.2 ms median latency and 7150 requests/second with a 1.2% error rate, roughly 46% higher throughput than Spring Cloud Gateway's 4450 requests/second. This is consistent with NGINX's event-driven, non-blocking worker model, which minimizes per-request overhead for simple proxying, and with prior literature noting that JVM-based gateways incur additional overhead from object allocation and garbage collection under sustained load.

Kong's performance closely tracks NGINX because Kong is itself built on the NGINX/OpenResty core, with the observed gap (18.4 ms vs. 15.2 ms latency) attributable to the additional Lua-based plugin execution pipeline that Kong runs on every request even in a minimal configuration. This suggests that Kong's architectural overhead for extensibility is modest relative to the flexibility it provides.

Traefik's throughput and latency sit between the NGINX-family gateways and the higher-overhead options, reflecting its Go-based implementation and its additional runtime cost of continuously watching the Kubernetes API server for configuration changes. The Istio Ingress Gateway, built on Envoy, shows moderate overhead relative to NGINX and Kong; this is expected given Envoy's more general-purpose proxy architecture, which is designed to support the richer traffic-management and telemetry feature set required for full service-mesh operation rather than being optimized purely for edge proxying throughput.

Spring Cloud Gateway consistently shows the highest latency, lowest throughput, and highest error rate under peak load, along with the highest CPU (52%) and memory (410 MB) consumption. This is consistent with its JVM-based reactive-Netty implementation, which carries higher baseline resource overhead than the natively compiled or event-loop-based alternatives, though it should be noted that Spring Cloud Gateway's programmability and native Java integration may still make it attractive for teams whose backend services are also Java-based and where developer productivity outweighs raw throughput considerations.

5.2 Resource Efficiency Analysis

Figures 3 and 4 show that resource consumption trends generally track performance trends: gateways with lower per-request overhead (NGINX, Kong) also require less CPU and memory to sustain a given load, meaning that, in resource-constrained clusters, choosing a higher-overhead gateway compounds cost twice over — lower throughput per pod and higher resource consumption per pod, which together increase the number of replicas needed to sustain a target service-level objective.

5.3 Qualitative Feature Comparison

Beyond raw performance, the gateways differ substantially in feature coverage and operational fit, which is often the deciding factor in practice. Table 4 summarizes this qualitative comparison across dimensions not exercised in the baseline load test.

Dimension	Kong	NGINX	Traefik	Spring Cloud GW	Istio Ingress GW
Native K8s service discovery	Via Ingress/CRD	Via Ingress	Automatic (CRD-native)	Manual/Eureka	Automatic (mesh-native)
Plugin/extensibility ecosystem	Extensive (Lua plugins)	Moderate (modules)	Moderate (middleware)	Extensive (Java code)	Moderate (Envoy filters)
East-west traffic support	No	No	No	No	Yes (full mesh)
Learning curve	Moderate	Low	Low	Low (for Java teams)	High
Built-in observability	Good	Basic	Good	Basic	Excellent

Table 4: Qualitative comparison of gateway features and operational characteristics.

Traefik's automatic, CRD-driven service discovery gives it the lowest operational overhead for teams that want routing configuration to be fully derived from Kubernetes manifests with zero manual gateway configuration. Kong and Spring Cloud Gateway offer the deepest extensibility through plugins or native code respectively, making them well suited to organizations that need custom authentication, transformation, or business logic at the gateway layer. The Istio Ingress Gateway is the only option evaluated that also governs service-to-service (east-west) traffic as part of a broader mesh, which is valuable for organizations that need uniform mTLS, retries, and traffic-shaping policies across all inter-service communication, not just at the edge, but this comes at the cost of the steepest learning curve and highest resource overhead observed in this study.

6. Conclusion

This paper presented a controlled, empirical comparison of five API gateways — Kong, NGINX, Traefik, Spring Cloud Gateway, and the Istio Ingress Gateway — deployed on an identical Kubernetes-orchestrated microservices backend. Across latency, throughput, error rate, CPU, and memory metrics, NGINX and Kong consistently delivered the best raw performance, Traefik offered the best balance of performance and Kubernetes-native operational simplicity, and Spring Cloud Gateway and the Istio Ingress Gateway traded performance for deeper integration with, respectively, the Java application ecosystem and the service mesh data plane.

No single gateway dominates across all dimensions considered, confirming that gateway selection for an orchestrated deployment should be driven by workload characteristics and organizational priorities rather than performance benchmarks alone. For latency-sensitive, high-throughput edge workloads with modest customization needs, NGINX or Kong are recommended. For teams prioritizing zero-configuration Kubernetes-native routing, Traefik is preferable. For organizations already standardized on Java/Spring, Spring Cloud Gateway offers superior developer productivity despite lower raw throughput. For organizations requiring uniform policy enforcement across both north-south and east-west traffic, the Istio Ingress Gateway is the only evaluated option that meets this requirement natively, justifying its additional resource cost.

Future work should extend this evaluation to additional gateways (such as Envoy Gateway, Apache APISIX, and cloud-managed gateways like AWS API Gateway and Azure API Management), incorporate gRPC and WebSocket traffic profiles in addition to REST, and examine gateway behavior under node failure and autoscaling events to assess resilience in addition to steady-state performance.

REFERENCES

[1] N. Dragoni et al., "Microservices: Yesterday, Today, and Tomorrow," in Present and Ulterior Software Engineering, Springer, 2017, pp. 195–216.

[2] S. Newman, Building Microservices: Designing Fine-Grained Systems, 2nd ed. O'Reilly Media, 2021.

[3] D. Jaramillo, D. V. Nguyen, and R. Smart, "Leveraging Microservices Architecture by Using Docker Technology," in Proc. IEEE SoutheastCon, 2016, pp. 1–5.

[4] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture," IEEE Software, vol. 33, no. 3, pp. 42–52, 2016.

[5] C. Pahl and P. Jamshidi, "Microservices: A Systematic Mapping Study," in Proc. 6th Int. Conf. on Cloud Computing and Services Science (CLOSER), 2016, pp. 137–146.

[6] N. Kratzke, "A Brief History of Cloud Application Architectures," Applied Sciences, vol. 8, no. 8, 2018, article 1368.

[7] T. Rahman and D. Wu, "Performance Evaluation of API Gateways for Cloud-Native Applications," in Proc. IEEE Int. Conf. on Cloud Computing Technology and Science (CloudCom), 2021, pp. 88–95.

[8] X. Li, Y. Chen, and T. Lin, "A Comparative Study of Service Mesh and API Gateway Architectures for Microservices," in Proc. Int. Conf. on Service-Oriented Computing (ICSOC) Workshops, 2020, pp. 210–221.

[9] W. Hasselbring and G. Steinacker, "Microservice Architectures for Scalability, Agility and Reliability in E-Commerce," in Proc. IEEE Int. Conf. on Software Architecture Workshops (ICSAW), 2017, pp. 243–246.

[10] E. Wolff, Microservices: Flexible Software Architecture. Addison-Wesley Professional, 2016.dministrative Science, 3 (20).