Journal of Information Systems Engineering and Management
2026, 11(2s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

Designing for Zero Downtime: Migrating a Legacy Gateway and Achieving High-Availability Platform Engineering

Kalyan Inturi
Independent Researcher, USA

ARTICLE INFO

Received: 01 Jan 2026

Revised: 11 Feb 2026

Accepted: 20 Feb 2026

ABSTRACT

Infrastructure migrations present substantial risks to service continuity, particularly for high-throughput SaaS environments. This article examines a gateway migration initiative at a leading enterprise SaaS provider, where a legacy Ruby/Sinatra monolithic gateway handling 50 million daily requests was migrated to a Kong Enterprise architecture. The migration employed phased rollout strategies, automated traffic replay for functional validation, and comprehensive infrastructure-as-code practices to achieve a seamless transition without service interruption. The initiative addressed multiple technical challenges, including the decoupling of tightly bound authentication and session logic into an OpenID Connect (OIDC) model, performance risk management during shadow traffic replication, and the enabling of rapid rollbacks. Results demonstrated successful service continuity throughout the transition period alongside operational improvements in reliability, reduced compute overhead, and enhanced security posture. This experience validates that detailed migration planning and incremental deployment strategies enable complex infrastructure transformations while maintaining strict Service Level Objectives (SLOs).

Keywords: Zero-Downtime Migration, Phased Rollout, Infrastructure as Code, Gateway Modernization, Shadow Traffic Replay

1. Introduction

Cloud infrastructure migrations often fail because organizations rely on synthetic testing that misses the complex, undocumented edge cases inherent in legacy systems. For SaaS providers operating at the scale of tens of millions of daily requests, maintaining high availability (typically above 99.99%) is commercially critical, making "rip and replace" strategies unviable [1]. Research indicates that investing in automated reliability engineering and continuous validation is the only proven method to prevent service disruptions during such critical transitions [3].

This paper presents a case study on the modernization of core ingress infrastructure at a high-growth enterprise SaaS organization. The legacy system, an 8-year-old Ruby/Sinatra monolith, had become a bottleneck for feature velocity and scalability. As the platform grew to process 50 million requests per day, the tightly coupled nature of the legacy gateway—handling routing, authentication, and session management in a single runtime—posed significant reliability risks [2].

While this study focuses on a specific transition from a legacy Ruby gateway to Kong Enterprise, the architectural patterns detailed herein are broadly adaptable. These methods serve as a reference blueprint for any organization transitioning from monolithic architectures to distributed microservices, particularly where service interruption is not an option [7]. As noted in recent industry studies, leveraging modern enterprise gateways allows organizations to offload complex cross-cutting concerns, freeing engineering teams to focus on core product capabilities [6].

A central contribution of this initiative was the implementation of an innovative functional validation framework. Recognizing that standard regression testing could not cover years of accumulated "tribal knowledge" in the legacy code, the team engineered a Shadow Traffic Replay system. This mechanism duplicated live production traffic to the new infrastructure asynchronously, enabling a direct, data-driven comparison of responses between the monolith and the new microservices-based architecture. This approach, which aligns with emerging zero-downtime strategies for large-scale distributed services [5], allowed for the identification of behavioral discrepancies that synthetic test suites missed.

By detailing this validation innovation alongside phased rollout strategies [8], this article provides a comprehensive methodology for executing complex infrastructure transformations without service disruption.

2. Gateway Modernization Drivers

Companies running older all-in-one gateway systems encounter growing problems as business needs expand. At the subject organization, the legacy Ruby gateway suffered from three critical limitations:

- Tight Coupling of Concerns: The legacy system combined request routing, user authentication, and session tracking within the same monolithic codebase. Changing a simple route often required redeploying the entire authentication stack, increasing the "blast radius" of minor updates. This tight coupling meant that changes in one area produced unexpected failures elsewhere, extending incident resolution times [2].
- Scalability Inefficiency: Authentication workloads consume different computational resources than simple request routing, yet the monolithic design forced the organization to scale the entire stack uniformly. Ruby's Global Interpreter Lock (GIL) further constrained concurrency, making it difficult to utilize multi-core instances effectively. This aligns with industry findings that monolithic gateways create resource scaling inefficiencies that impede growth [6].
- Operational Risk: The legacy codebase relied on older language versions where internal expertise was becoming scarce. Security patches for the underlying framework were becoming difficult to backport, creating a compliance risk where the organization had to choose between accepting known vulnerabilities or manual patching [1].

3. Migration Challenges

Executing gateway migrations while maintaining continuous service availability presents organizations with interconnected technical and operational obstacles. The fundamental requirement to maintain continuous service availability changes the migration from a purely technical task into a complex coordination effort [2].

- **Undocumented Legacy Behaviors:** Legacy gateways often develop "tribal knowledge" characteristics through years of operation, such as implicit assumptions regarding request formats or undocumented edge case handling [5]. Synthetic test suites cannot reliably detect these discrepancies, necessitating validation against actual production traffic.
- **Performance Impact Risks:** Duplicating production requests for validation introduces processing overhead. A critical risk in standard shadowing is "double execution," where replayed requests might trigger duplicate backend transactions (e.g., double payments). The migration strategy must ensure that replayed traffic validates the gateway's logic without executing side effects on downstream services [5].

Journal of Information Systems Engineering and Management
2026, 11(2s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/ **Research Article**

- **Coupled Authentication Logic:** Monolithic gateways typically embed authentication enforcement within the routing code. Migrating this logic to independent microservices while ensuring consistent security enforcement across distributed architectures requires careful coordination to prevent authorization gaps [7].

4. Methodology: Interceptor-Based Shadow Validation

To address the challenge of "double execution" during validation, the team engineered a custom State-Mocking Shadow Framework. Unlike simple traffic mirroring, this approach allowed the new Kong gateway to process live production traffic in a "sandbox" mode that verified logic without impacting backend state.

Interceptor-Based Shadow Traffic Flow

To address the challenge of 'double execution' during validation, the team engineered a custom State-Mocking Shadow Framework. **Figure 1** illustrates this innovative 'No-Double-Execution' replay mechanism, detailing the interaction between the legacy gateway, the NoSQL store, and the interceptor service.

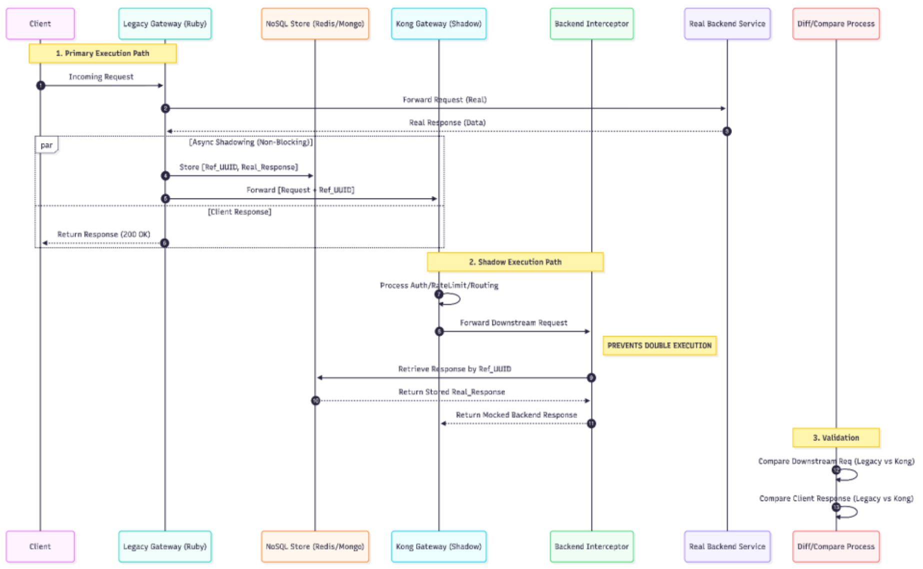
Client
Legacy Gateway (Ruby)
NoSQL Store (Redis/Mongo)
Kong Gateway (Shadow)
Backend Interceptor
Real Backend Service
Diff/Compare Process
1. Primary Execution Path
Incoming Request
Forward Request (Real)
Real Response (Data)
par
[Async Shadowing (Non-Blocking)]
Store [Ref_UUID, Real_Response]
Forward [Request + Ref_UUID]
[Client Response]
Return Response (200 OK)
2. Shadow Execution Path
Process Auth/RateLimit/Routing
Forward Downstream Request
PREVENTS DOUBLE EXECUTION
Retrieve Response by Ref_UUID
Return Stored Real_Response
Return Mocked Backend Response
3. Validation
Compare Downstream Req (Legacy vs Kong)
Compare Client Response (Legacy vs Kong)
Client
Legacy Gateway (Ruby)
NoSQL Store (Redis/Mongo)
Kong Gateway (Shadow)
Backend Interceptor
Real Backend Service
Diff/Compare Process

Figure 1 illustrates this innovative 'No-Double-Execution' replay mechanism, detailing the interaction between the legacy gateway, the NoSQL store, and the interceptor service

4.1 The Shadow Replay Mechanism

The implementation utilized a "Store-and-Forward" pattern to decouple execution:

1. **Request Capture:** When the Legacy Gateway processed a request, it generated a unique reference ID (UUID). After receiving the response from the *real* backend service, it asynchronously stored this response payload **(via a non-blocking worker)** in a temporary NoSQL database (e.g., Redis or MongoDB), keyed by the UUID [5].
2. **Shadow Forwarding:** The Legacy Gateway then forwarded the original incoming request, tagged with the UUID, to the Kong Gateway.

Journal of Information Systems Engineering and Management
2026, 11(2s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/ **Research Article**

3. **The Backend Interceptor:** Kong processed the request normally (applying OIDC verification, rate limiting, and transformations). However, instead of routing to the live backend service, Kong was configured to route shadow traffic to a specialized **Interceptor Service**.
4. **State Mocking:** The Interceptor extracted the UUID from the request headers, looked up the stored response in the NoSQL database, and returned it to Kong. This simulation enabled Kong to complete the request lifecycle as if it had successfully communicated with the backend, preventing duplicate transactions while validating the full routing logic.

4.2 Automated Discrepancy Analysis

A background process compared the artifacts from both pathways [8]:

- **Downstream Request Verification:** It compared the request payload Kong *attempted* to send to the backend against what the Legacy Gateway *actually* sent. This ensured that Kong's request transformation plugins were correctly configured.
- **Client Response Verification:** It compared the final response Kong generated for the client against the actual response sent by the Legacy Gateway. Any differences (e.g., missing headers, incorrect status codes) were flagged as regressions.

5. Implementation Framework

The technical implementation progressed through coordinated phases that built validation confidence while maintaining production stability [4]. As detailed in **Figure 2**, the migration followed a sequential eight-stage path, moving from initial planning and provisioning through to the final legacy retirement.

Figure 2: Phased Implementation Steps

This flowchart details the sequential stages of the zero-downtime migration.

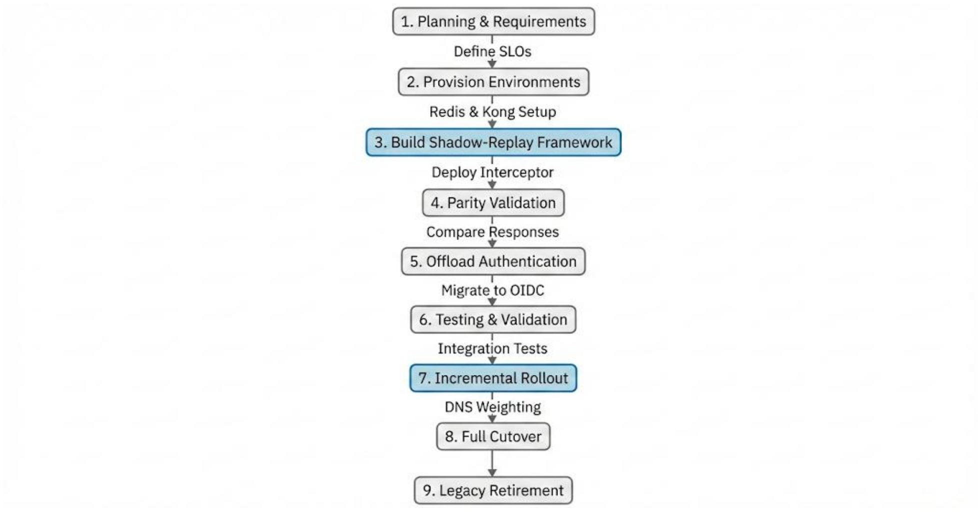

Figure 2: Enhanced Phased Steps for Zero-Downtime Migration. This vertical flowchart details the eight stages of the migration: planning and requirements gathering, provisioning environments, building the shadow-replay framework, offloading authentication, testing and validation, incremental rollout, full cutover, and retiring the legacy system.

Planning & Provisioning: Teams mapped authentication flows and routing rules to ensure comprehensive functional coverage. Parallel infrastructure was deployed using Infrastructure as Code (IaC) to ensure environment consistency [4].

Shadow Framework Construction: Engineers deployed the lightweight proxy layer and the NoSQL store. The validation began with **traffic sampling**, starting with a small percentage of production traffic to avoid resource overuse [5].

Authentication Offloading: A critical step involved extracting the authentication logic into independent microservices, enabling the new gateway to verify tokens via OIDC without relying on the monolithic database [7].

Incremental Rollout: Feature flags and DNS rules controlled traffic allocation. Traffic was shifted incrementally (1% $\rightarrow$ 5% $\rightarrow$ 100%) based on observed system behavior, allowing migration velocity to adapt to real-world validation results [8].

Phase	Key Activities	Technologies/Tools	Validation Method
Planning & Requirements	Service-level objective definition, legacy feature mapping, dependency identification	Documentation frameworks, requirement tools	Requirements verification, stakeholder alignment
Environment Provisioning	Parallel infrastructure deployment, CI/CD pipeline setup	Docker containers, orchestration platforms	Configuration testing, environment parity checks
Shadow Traffic Framework	Live request duplication with progressive traffic sampling, starting with a small percentage of production traffic and gradually increasing sample rates until 100% parity validation, response capture, and storage	Lua-based proxy, MongoDB database	Automated response comparison, behavioral diffing
Microservice Migration	Authentication and authorization logic extraction	Service mesh, secure communication protocols	Shadow validation, functional equivalence testing
Automated Testing	Response comparison across millions of requests, edge case validation	Integration test frameworks, performance tools	Error handling verification, throughput validation
Incremental Rollout	Gradual traffic shifting with monitoring	Feature flags, load balancer rules	Real-time metrics monitoring, error rate tracking
Full Cutover & Decommission	Complete traffic migration, legacy system retirement	Rollback procedures, cutover automation	SLO compliance verification, final validation period

Table 1: Migration Implementation Phases [4, 5, 7, 8]

Journal of Information Systems Engineering and Management
2026, 11(2s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/ **Research Article**

6. Outcomes and Operational Impact

The infrastructure transition maintained continuous service availability without disrupting customer access throughout all migration phases. No customer-facing incidents were recorded during the incremental rollout, demonstrating that shadow traffic validation combined with gradual deployment mechanisms effectively prevented service interruptions. This outcome proved particularly significant given that the system processed substantial daily request volumes, where even brief disruptions would have affected numerous customer interactions.

Financial implications extended beyond avoided downtime costs to include operational expense reductions from modernized infrastructure. The replacement gateway required fewer computational resources for equivalent workload processing compared to legacy implementations, translating to measurable cost reductions in hosting expenses. Organizations adopting modern platform architectures report infrastructure cost decreases exceeding twenty percent while simultaneously improving performance characteristics [6]. These savings compound over multi-year periods as reduced operational overhead frees budget allocation for feature development rather than infrastructure maintenance.

Reliability improvements manifested through access to vendor-supported security patches and compliance certifications previously unavailable for deprecated gateway platforms. The modern gateway received regular security updates addressing newly discovered vulnerabilities, eliminating the technical debt accumulated from running unsupported software versions. Compliance framework adherence became straightforward with vendor-provided documentation rather than requiring custom attestation efforts for legacy systems [1]. Automated deployment capabilities reduced mean time to recovery for incidents by enabling rapid rollback to known-good configurations when issues emerged.

Operational efficiency gains resulted from eliminating manual deployment procedures susceptible to human error. Engineering teams could deploy changes through automated pipelines with confidence that testing gates would catch regressions before production deployment. This automation freed engineering capacity previously consumed by deployment coordination for higher-value feature development activities [3]. The architectural separation of authentication logic into independent microservices enabled autonomous scaling and update cycles, preventing authentication bottlenecks from constraining overall system capacity. Multi-tenant capabilities positioned the infrastructure to accommodate future business growth and acquisition integrations without requiring additional migration initiatives.

Outcome Category	Achievement	Business Impact
Service Continuity	Zero customer-facing incidents throughout migration	Maintained customer trust, no revenue disruption
Cost Efficiency	Infrastructure cost reduction exceeding 20%, projected multi-year savings	Freed budget for feature development vs. maintenance
Security Posture	Vendor-supported security patches, SOC 2 compliance certifications	Eliminated technical debt, simplified compliance
Operational Efficiency	Automated deployment pipelines eliminated manual errors	Freed engineering capacity for high-value activities

System Reliability	Reduced mean time to recovery through automated rollback	Improved incident response capabilities
Scalability	Independent microservice scaling, multi-tenant architecture support	Future-ready infrastructure for business growth

Table 2: Outcomes and Operational Impact [1, 3, 6]

7. Lessons and Replicability

Several critical insights emerged from executing this zero-downtime migration that inform future infrastructure transformation initiatives. Automation investment proved essential for achieving migration success despite requiring substantial upfront engineering effort. Organizations attempting similar transitions without comprehensive automation capabilities face a significantly higher risk of service disruption and prolonged migration timelines. The shadow traffic framework and automated deployment pipelines provided the confidence necessary for executing cutover decisions, demonstrating that automation serves as risk mitigation rather than an optional enhancement [8].

Functional parity validation using production traffic patterns revealed behavioral discrepancies that synthetic testing could not detect. Edge cases embedded in years of production operation manifested only when replacement systems processed actual customer requests. This observation suggests that migration validation strategies should prioritize real-world traffic replay over synthetic test scenario development when legacy systems exhibit complex or undocumented behaviors.

Architectural separation of authentication and authorization concerns simplified gateway responsibilities while enabling independent component evolution. Organizations maintaining monolithic architectures should evaluate opportunities for extracting cross-cutting concerns into dedicated services before attempting infrastructure migrations. This separation reduces migration complexity by decomposing the problem into smaller, independently validatable transformations [2].

Feature flag capabilities combined with incremental traffic shifting provided essential safety mechanisms throughout migration execution. The ability to rapidly revert traffic allocation when unexpected issues emerged prevented minor problems from escalating into major incidents. Migration planners should incorporate similar rollback capabilities at each phase transition rather than committing to irreversible cutover decisions.

These techniques extend beyond gateway replacements to broader infrastructure modernization contexts. Organizations transitioning toward microservice architectures benefit from similar shadow validation and incremental deployment strategies. Legacy system replacements across various domains can employ automated behavioral comparison between old and new implementations. Cloud-native platform adoption leverages infrastructure as code practices for reproducible environment provisioning and simplified rollback capabilities. The methodological patterns described prove adaptable to diverse technical contexts requiring continuous service availability throughout transformation execution.

Lesson Learned	Rationale	Applicable Contexts
Invest in automation early	Shadow replay frameworks and CI/CD pipelines provide migration confidence despite upfront effort	Complex infrastructure migrations, distributed system transitions
Prioritize production traffic validation	Real traffic reveals edge cases and undocumented behaviors that synthetic tests cannot detect	Legacy system replacements, behavioral parity requirements
Separate architectural concerns	Extracting cross-cutting concerns enables independent evolution and reduces migration complexity	Monolithic to microservice transitions, modular architecture adoption
Implement comprehensive rollback mechanisms	Feature flags with incremental traffic shifting prevent minor issues from escalating	Any phased deployment, high-availability requirements
Adopt infrastructure as code	Code-managed infrastructure ensures reproducibility, governance, and team collaboration	Cloud-native platform adoption, multi-environment deployments

Table 3: Key Lessons and Replicability Contexts [2][4][8]

Conclusion

Gateway modernization initiatives present organizations with technical complexity that demands systematic execution strategies, preventing service disruption. Phased migration methodologies combining shadow traffic replay, infrastructure as code practices, and incremental rollout mechanisms enable organizations to transition from monolithic architectures to contemporary gateway platforms without impacting customer experiences. The implementation demonstrates that comprehensive automation investments, particularly in traffic validation frameworks and deployment pipelines, provide the confidence necessary for executing complex infrastructure transitions. Functional parity verification through production traffic comparison reveals behavioral differences that synthetic testing cannot detect, ensuring replacement systems match legacy functionality before production cutover. Separating authentication and authorization concerns into independent microservices simplifies gateway responsibilities while enabling independent scaling and update cycles. Feature flag strategies combined with incremental traffic shifting provide essential rollback capabilities at each migration stage, preventing catastrophic failures during deployment. Managing infrastructure, configuration settings, and deployment processes through code enhances repeatability and oversight while allowing teams to share patterns throughout their organizations. These methods apply not only to gateway migrations but also to moving toward microservices, updating legacy systems, and any infrastructure change that needs to keep services running continuously. Organizations adopting these systematic approaches to migration planning, automation investment, and incremental deployment establish capabilities for executing complex technical transformations while maintaining service reliability and customer satisfaction throughout transition periods.

Journal of Information Systems Engineering and Management
2026, 11(2s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/ **Research Article**

References

[1] Gianluca Caricato, Marco Capotosto, Silvio Orsini, and Pietro Tiberi, "TIPS: A Zero-Downtime Platform Powered by Automation," SSRN, Bank of Italy Markets, Infrastructures, Payment Systems Working Paper No. 28, Mar. 2023. https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4381966

[2] Maheshbhai Kansara, "Cloud Migration Strategies and Challenges in Highly Regulated and Data-Intensive Industries: A Technical Perspective," International Journal of Applied Machine Learning and Computational Intelligence, ResearchGate, Dec. 2021. https://www.researchgate.net/profile/Maheshbhai-Kansara/publication/389254166_Cloud_Migration_Strategies_and_Challenges_in_Highly_Regulated_and_Data-Intensive_Industries_A_Technical_Perspective/links/67ba3a618311ce680c705e69/Cloud-Migration-Strategies-and-Challenges-in-Highly-Regulated-and-Data-Intensive-Industries-A-Technical-Perspective.pdf

[3] Maheeza Bhamidipati, "AI-Driven Automation and Reliability Engineering: Optimizing Cloud Infrastructure for Zero Downtime and Scalable Performance," Journal of Computer Science and Technology Studies, May 2025.https://al-kindipublishers.org/index.php/jcsts/article/view/9725

[4] M. Vaidhegi and G. Glory, "AI-Powered Cloud Migration: Automating the Transition from On-Premises to Cloud Environments with Zero Downtime," International Journal of Innovative Research in Science Engineering and Technology (IJIRSET), vol. 14, no. 1, Jan. 2025. https://philpapers.org/rec/GGLACM

[5] Tharun Damera, "Zero-Downtime Migration Strategies for Large-Scale Distributed Services," International Journal of Scientific Research in Computer Science Engineering and Information Technology, ResearchGate, Mar. 2025. https://www.researchgate.net/publication/389582692_Zero-Downtime_Migration_Strategies_for_Large-Scale_Distributed_Services

[6] Ivan Ivanov, "How GameStake, a Technology Startup, Got to Zero Downtime and 25% Cloud Cost Reduction with Massdriver," GameStake Technologies, Massdriver, 2019. https://www.massdriver.cloud/case-studies/gamestake-achieves-zero-downtime-deployments-with-a-25-reduction-in-cloud-costs

[7] Sesha Sai Sravanthi Valiveti, ".NET Core Microservices for Zero-Downtime AuthHub Migrations," European Journal of Engineering and Technology Research (EJ-ENG), Sep. 2025. https://ej-eng.org/index.php/ejeng/article/view/3288

[8] Sandeep Kumar Gond, "Phased Migration Strategy for Zero Downtime in Systems," DZone, Jan. 2025.https://dzone.com/articles/phased-migration-strategy-zero-downtime