International Journal of Emerging Trends in Computer Science and Information Technology
ISSN: 3050-9246 | https://doi.org/10.63282/3050-9246.IJETCSIT-V6I3P101
Eureka Vision Publication | Volume 6, Issue 3, 1-12, 2025

Case Study

Modern Approach to Kubernetes Traffic Management: Migrating from Ingress to Gateway API

[1]Upendra Kanuru, [2]Upendra Kumar Gurugubelli
[1] Software Engineer, USA.
[2] Software Engineer, Japan.

Received On: 16/05/2025 ***Revised On: 04/06/2025*** ***Accepted On: 17/06/2025*** ***Published On: 03/07/2025***

Abstract *- This paper presents a comprehensive analysis of the migration from Kubernetes Ingress to the emerging Gateway API for traffic management in cloud-native environments. While Kubernetes Ingress has served as a foundational component for exposing services, its limitations in managing complex, multi-tenant, and highly dynamic microservices architectures have become increasingly apparent. The Kubernetes Gateway API, designed with a role-oriented, portable, expressive, and extensible model, offers a significant evolution by decoupling infrastructure and application concerns, providing richer routing capabilities, and enhancing role-based access control. This study details the motivations behind adopting the Gateway API, outlines a practical migration methodology, including architectural comparisons and the utility of tools like ingress2gateway, and discusses the challenges encountered. The observed improvements in operational clarity, routing flexibility, scalability, and security post-migration position the Gateway API as a future-forward standard for robust and maintainable Kubernetes traffic control.*

Keywords *- Kubernetes, Gateway API, Ingress, Traffic Management, Cloud-Native, Microservices, Networking, Migration, Enterprise Security.*

1. Introduction

1.1. Background on Kubernetes and Cloud-Native Architectures

Kubernetes (K8s) has solidified its position as the de facto standard for container orchestration, fundamentally transforming how applications are deployed, managed, and scaled within cloud-native paradigms [5]. This platform automates critical tasks such as the deployment, scaling, and operational management of containerized applications, thereby facilitating faster, more reliable software updates and supporting modern DevOps methodologies. The core functionality of Kubernetes is intricately linked to its robust networking model, which ensures seamless communication among containerized workloads and facilitates external access to the cluster [3].

The rapid adoption of Kubernetes and its instrumental role in enabling complex microservices architectures have significantly expanded the demands placed on its initial networking constructs. The foundational elements, including Services, Ingress, and DNS, were initially conceived to address a certain level of complexity. However, the escalating requirements of modern, distributed applications, particularly in multi-tenant or highly dynamic environments, have surpassed these original designs. This progression highlights a continuous imperative for Kubernetes' networking capabilities to evolve. The shift extends beyond merely introducing new features; it encompasses a fundamental re-architecture to support sophisticated traffic management, enhanced security, and granular policy enforcement, reflecting the increasing maturity and inherent complexity of contemporary cloud-native deployments.

1.2. Overview of Kubernetes Ingress as the Traditional Solution

For an extended period, Kubernetes Ingress served as the primary solution for managing external HTTP(S) access to services within a Kubernetes cluster. It functioned as an intelligent Layer 7 reverse proxy, directing incoming traffic to internal services based on predefined rules such as hostnames or URL paths [8]. This approach centralized routing logic, which simplified initial traffic management and reduced the need for individual services to expose public IP addresses, thereby mitigating potential security risks [9].

While Ingress represented a substantial improvement over direct service exposure via NodePort or LoadBalancer services, it was primarily designed for more straightforward HTTP/HTTPS routing scenarios. Its widespread adoption, however, led to its application in use cases for which it was not optimally designed, particularly in complex enterprise environments. This situation illustrates a common pattern in rapidly evolving technological landscapes: a solution initially developed for a specific problem (basic external HTTP routing) is often extended through ad-hoc mechanisms, such as custom annotations, to accommodate more advanced requirements. Over time, this stretching of the original design reveals inherent limitations, ultimately necessitating the development of a more purpose-built successor.

1.3. Problem Statement: Limitations and Growing Challenges with Ingress

Despite its widespread utility, Ingress presented several

significant challenges for production workloads. These included a tight coupling between routing rules and underlying infrastructure details, which diminished flexibility and complicated management as environments scaled. Furthermore, managing multi-tenant workloads became increasingly difficult, leading to complexities in isolating and controlling traffic for disparate users or teams. The demand for more granular routing policies also frequently exceeded Ingress's native capabilities.

Additional limitations of Ingress included its primary restriction to HTTP/HTTPS protocols, inherent difficulties in extending its functionality, and the prevalence of controller-specific behaviors. Advanced features like header-based matching and traffic weighting often relied on custom annotations, which varied significantly across different Ingress controller implementations. This reliance on custom annotations and the tight coupling of routing logic resulted in a significant accumulation of technical debt. Each Ingress controller, such as NGINX, Traefik, or HAProxy, implemented extended features through its own set of proprietary annotations. This fragmentation of the ecosystem hindered portability of configurations across different environments or controllers and created a substantial maintenance burden. It also increased the learning curve for new teams and made it challenging to switch between different Ingress controllers or cloud providers. This situation underscores how ad-hoc extensibility, while offering immediate expediency, can lead to long-term operational complexities and impede innovation by locking users into specific vendor implementations. These challenges directly motivated the Kubernetes community to pursue a more standardized and flexible API.

1.4. Introduction to the Kubernetes Gateway API as an Evolved Solution

The Kubernetes Gateway API represents a profound evolution in traffic management within Kubernetes, fundamentally rethinking how routing and traffic control should operate in modern, scalable cloud-native environments. It is designed as a more flexible, powerful, and extensible alternative to the traditional Ingress API. The Kubernetes community is actively championing the Gateway API as the "eventual successor to Ingress", signaling a clear strategic direction for future development and adoption within the ecosystem.

The development of the Gateway API by the Kubernetes SIG-NETWORK group, coupled with its progression towards general availability, reflects a strong community consensus regarding the necessity for a standardized, future-proof solution for traffic management. This initiative transcends a mere feature addition; it signifies a strategic architectural reorientation. This community-led standardization effort aims to mitigate vendor lock-in, enhance interoperability, and deliver a consistent user experience across diverse Kubernetes environments and controller implementations. It demonstrates a maturing ecosystem where foundational APIs are being refined to accommodate increasingly complex and varied use cases, thereby ensuring long-term sustainability and reducing operational friction for organizations deploying cloud-native applications.

1.5. Paper Objective

This paper aims to provide a comprehensive analysis of the motivations, practical implementation process, and observed benefits associated with migrating from Kubernetes Ingress to the Gateway API. It will serve as a detailed guide for organizations contemplating or actively planning such a transition, drawing upon real-world experiences and incorporating academic insights into the broader Kubernetes networking landscape.

2. Kubernetes Networking Landscape

2.1. Services, Pods, and Internal Communication

Within a Kubernetes cluster, Services provide stable and consistent access points to Pods, which are inherently ephemeral. This abstraction ensures that workloads remain reachable irrespective of individual Pod failures or restarts. Services are capable of load-balancing traffic across a group of Pods that are identified by common labels. Kubernetes offers several types of Services, each designed for specific communication patterns: Cluster IP exposes the Service on a cluster-internal IP address; NodePort exposes the Service on a static port across all nodes; LoadBalancer provisions an external IP address through integration with a cloud provider; and External Name maps a Service to an external DNS name.

The fundamental problem Kubernetes addresses is the management of volatile containers. Services are a critical abstraction layer that provides stable network identities, such as virtual IPs and DNS names, for highly dynamic Pods. This design choice is paramount for microservices architectures, where individual instances are frequently scaled up or down, or restarted. This abstraction is foundational to Kubernetes' resilience and scalability, allowing application developers to concentrate on their service logic without being burdened by the intricacies of the underlying Pod lifecycle. This promotes a decoupled architecture where components can evolve independently. However, this internal abstraction necessitates an external entry point for traffic, which is precisely the role fulfilled by Ingress and, subsequently, the Gateway API.

2.2. Kubernetes Ingress: Architecture, Capabilities, and Use Cases

Kubernetes Ingress functions as an intelligent Layer 7 reverse proxy, specifically designed to route external HTTP(S) traffic to internal services based on rules defined by hostnames, URL paths, or other advanced criteria. To operate, an Ingress resource requires an Ingress Controller, such as NGINX, Traefik, or HAProxy, which interprets the Ingress definitions and configures the actual routing mechanisms.

The capabilities of Ingress include SSL termination for secure traffic handling, URL rewriting, redirection, and basic

rate limiting or authentication, often implemented through annotations. It proved particularly beneficial for microservices architectures where multiple services shared a common domain, as it consolidated routing logic and reduced the need for individual services to possess public IP addresses.

The Ingress resource itself defines the routing rules, but the actual traffic handling and feature implementation are delegated to an external Ingress Controller. This design centralizes external traffic management, but it also means that the effective features available to users are dictated by the capabilities and limitations of the specific controller in use. This architectural characteristic, while simplifying initial external exposure, inadvertently led to a fragmented ecosystem. Each controller implemented advanced features through its own set of custom, often proprietary, annotations. This lack of standardization at the advanced feature level meant that Ingress configurations were not portable across different controller implementations, resulting in tight coupling to a specific vendor's solution. This ultimately emerged as a significant limitation, underscoring the need for a more unified and extensible API like the Gateway API.

2.3. DNS and Service Discovery in Kubernetes

Kubernetes automatically creates DNS records for both Services and Pods, enabling workloads within the cluster to discover and communicate with Services using consistent DNS names rather than ephemeral IP addresses (Kubernetes.io). The DNS system dynamically updates these records to reflect the current state of Services and Pods, thereby underpinning microservice communication by simplifying service discovery and facilitating dynamic scaling.

DNS, often an overlooked component, is nevertheless fundamental to Kubernetes' dynamic nature. The ability for services to locate each other by name, rather than relying on potentially changing IP addresses, is crucial for maintaining self-healing capabilities and enabling seamless scaling operations. However, this reliance on DNS also positions it as a critical single point of failure or a potential performance bottleneck if misconfigured, or if propagation issues arise during changes. This highlights that while API changes are important, fundamental network services like DNS require meticulous planning. The importance of careful DNS planning during significant network shifts, such as migrating from Ingress to Gateway API, is therefore paramount. Unforeseen DNS propagation delays can cause substantial short-term disruptions, emphasizing that even well-designed API changes necessitate careful operational planning around underlying network fundamentals. This underscores the need for robust DNS management strategies in any Kubernetes deployment.

2.4. Container Network Interfaces (CNIs) and Network Performance

The Kubernetes networking model is central to its functionality, with Container Network Interfaces (CNIs) playing a pivotal role in seamlessly connecting containerized workloads. Popular CNI plugins include Cilium, Flannel, Calico, and Antrea. These plugins assign each Pod a globally unique virtual IP address, facilitating communication between Pods located on the same node or across different nodes within the cluster.

Each CNI plugin presents a unique set of trade-offs concerning simplicity, security, throughput, and scalability. For example, Cilium is noted for its high performance and advanced security features, leveraging eBPF (extended Berkeley Packet Filter) to often outperform other CNIs like Calico in terms of latency and throughput in specific scenarios. The selection of a CNI plugin is not a minor configuration detail but a foundational decision that significantly influences the performance, reliability, and security characteristics of the entire Kubernetes cluster. Different CNIs offer varying levels of capabilities, ranging from basic overlay networking (e.g., Flannel) to sophisticated policy enforcement and observability (e.g., Cilium, Calico). This implies that while Ingress and Gateway API manage external traffic, the efficiency and security of internal Pod-to-Pod communication are heavily dependent on the chosen CNI. A high-performing CNI can maximize the benefits derived from advanced traffic management APIs by ensuring that the underlying network fabric is robust and efficient. This illustrates the layered nature of Kubernetes networking, where each component contributes synergistically to the overall system's capabilities.

3. Challenges with Traditional Kubernetes Ingress

3.1. Tight Coupling and Lack of Role Separation

Ingress configurations frequently intertwine routing logic with specific details of the underlying infrastructure, resulting in a tight coupling between these layers. This design characteristic renders configurations less flexible and more challenging to manage effectively as the environment scales. The tight coupling inherent in Ingress means that infrastructure providers, who are responsible for managing load balancers and network policies, and application developers, who define routing rules for their services, often operate on the same Ingress resource. This blurs the traditional ownership boundaries between these distinct roles. When changes are required, this interdependency necessitates close coordination, which increases the potential for misconfigurations or delays.

This architectural limitation directly translates into operational friction, leading to slower deployment cycles and reduced agility for development teams. It impedes the ability to implement true GitOps principles or self-service models, as infrastructure changes might inadvertently disrupt application routing, and vice versa. The absence of clear role separation thus becomes a significant impediment to scaling operations efficiently within large organizations.

3.2. Limitations in Advanced Routing and Protocol Support

The capabilities of Ingress for defining fine-grained routing rules were often found to be insufficient, necessitating the search for more powerful solutions. Ingress was primarily designed to handle HTTP/HTTPS (Layer 7) traffic, lacking native support for other essential protocols such as TCP or UDP (Layer 4). Furthermore, advanced features like header-based matching and traffic weighting frequently required the use of custom, non-standard annotations, which varied across different Ingress controller implementations.

The core Ingress specification was intentionally minimal, leaving advanced routing features to be implemented by individual Ingress controllers through proprietary annotations. While this approach fostered rapid innovation among controller vendors, it inadvertently created a fragmented landscape where advanced routing capabilities were neither standardized nor portable. This situation compelled users to adopt vendor-specific workarounds for common advanced routing scenarios, such as canary deployments or A/B testing, thereby limiting their flexibility and increasing the complexity of managing diverse traffic patterns. The absence of native multi-protocol support further constrained Ingress's utility for modern microservices that increasingly rely on protocols like gRPC or other Layer 4 communication.

3.3. Difficulties in Multi-Tenancy Management

In environments supporting multiple tenants, Ingress proved cumbersome to manage, leading to significant complexities in isolating and controlling traffic for different users or teams. Ensuring strict separation of data and resources within multi-tenant SaaS platforms is paramount for security and performance, but Ingress's design made achieving this challenging [2].

While Ingress did support domain-based multi-tenancy, it lacked robust, built-in mechanisms for fine-grained multi-tenant isolation and policy enforcement at the routing layer. This often necessitated complex manual configurations of network policies or reliance on external tools to achieve adequate tenant separation. Such an approach increased the risk of "noisy neighbor" issues, where one tenant's resource consumption negatively impacts others , or even data leakage if configurations were erroneous. The inherent limitations of Ingress for multi-tenancy translated into significant security and performance risks for platform operators. It required extensive custom tooling and operational overhead to enforce isolation, making it less ideal for building robust "routing-as-a-service" platforms for internal product teams or external customers. This directly highlighted the need for a more opinionated and structured approach to multi-tenancy, which the Gateway API aims to provide.

3.4. Operational Overhead and Extensibility Constraints

The fragmented nature of Ingress setups, characterized by diverse controller implementations and custom annotations, often created considerable operational challenges. These included the burden of managing multiple configurations and ensuring consistency across different environments. Even conversion tools like ingress2gateway, while helpful, frequently required post-editing of the generated YAML configurations to align with specific load balancer or controller nuances [6].

The fact that ingress2gateway serves as a "scaffold, not a final product" and necessitates "post-editing" underscores a critical aspect of complex system migrations. While automation tools can significantly accelerate the initial conversion, they cannot fully capture the intricacies of existing custom configurations, specific vendor implementations, or unique operational requirements. This implies that human expertise and manual refinement remain indispensable in such migrations. It further highlights that even declarative APIs, while simplifying management, still demand a deep understanding when transitioning from legacy systems. The "last mile" of migration often involves bespoke adjustments, emphasizing the continued value of skilled engineers in an increasingly automated world. This situation reflects the industry's continuous push towards more declarative, automated, and standardized solutions in cloud-native environments. The shortcomings of Ingress in this regard provided a strong impetus for the Gateway API to be designed with explicit extensibility points and a more unified approach to traffic management, aiming to reduce manual intervention and improve overall operational efficiency.

4. The Kubernetes Gateway API: Design and Capabilities

4.1. Core Design Principles: Role-Oriented, Portable, Expressive, Extensible

The Kubernetes Gateway API is fundamentally shaped by four core design principles:

- **Role-oriented:** This principle models API kinds after the distinct organizational roles responsible for managing Kubernetes service networking: the Infrastructure Provider, the Cluster Operator, and the Application Developer. This design allows shared network infrastructure to be utilized by multiple, potentially non-coordinating teams while maintaining centralized control and governance.
- **Portable:** Defined as Custom Resources (CRDs), the Gateway API specifications ensure broad support across a wide range of implementations. This directly addresses the issue of vendor-specific annotations that plagued Ingress, promoting greater interoperability.
- **Expressive:** The API provides built-in functionality for common and advanced traffic routing scenarios, such as header-based matching and traffic weighting. These capabilities previously required custom Ingress annotations, making the Gateway API more intuitive and powerful out-of-the-box.

- **Extensible:** The design allows for custom resources to be linked at various layers of the API, enabling granular customization at appropriate points within the API structure.

The "role-oriented" principle represents a profound architectural shift from Ingress. Instead of a single, monolithic Ingress resource attempting to serve all purposes, the Gateway API explicitly separates concerns and responsibilities across different personas. This extends beyond mere technical separation; it directly influences organizational design and workflow. The Infrastructure Provider defines the Gateway Class, which specifies the type of gateway controller. The Cluster Operator then deploys Gateways based on these classes and establishes overarching policies. Finally, the Application Developer defines Routes for their specific services, linking them to the appropriate Gateways. This design enables decentralized application development and deployment, granting developers greater autonomy over their routing configurations, while simultaneously maintaining centralized governance and policy enforcement through the operators' control over gateways and policies. This architectural pattern directly addresses the tight coupling and multi-tenancy challenges faced with Ingress by providing clearer ownership boundaries and a structured method for managing shared infrastructure. Consequently, it fosters greater agility, mitigates misconfiguration risks, and scales more effectively with organizational growth, aligning seamlessly with modern platform engineering principles.

4.2. Key Resources: Gateway Class, Gateway, and Route Types (e.g., HTTP Route)

The Gateway API is composed of three primary resource types that work in concert to manage traffic:

- Gateway Class: This resource functions as a blueprint or template for Gateways. It defines a group of Gateways that share a common configuration and are managed by a specific controller, such as k8s.io/gateway-nginx.
- Gateway: This resource represents the actual traffic handling infrastructure, acting as an entry point to the cluster, often backed by a cloud load balancer. It defines listeners for specific protocols (e.g., HTTP) on particular ports and hostnames.
- Route Types (e.g., HTTPRoute): These resources define the rules for how traffic is routed from a Gateway to backend services. HTTPRoute specifically handles HTTP traffic, allowing for sophisticated matching based on paths, headers, and hosts. Beyond HTTP/HTTPS, the Gateway API extends support to other protocols through dedicated route types like TCPRoute, TLSRoute, and UDPRoute.

The relationship between Gateway and Route objects introduces a "bidirectional trust model". A Gateway can explicitly filter which Routes are permitted to attach to its listeners, and conversely, Routes reference specific Gateways. This contrasts sharply with Ingress, where any Ingress resource could potentially attach to an Ingress Controller without explicit permission. This explicit association and filtering mechanism significantly enhances both security and control within the cluster. Cluster operators can define stringent policies at the Gateway level, ensuring that application developers can only attach routes that conform to organizational security and networking policies. This prevents unauthorized traffic exposure and misconfigurations, making the system inherently more robust and auditable, especially crucial in multi-tenant or highly regulated environments.

4.3. Enhanced Traffic Management Features: Header Matching, Traffic Splitting, Policy Attachment

The Gateway API provides a comprehensive suite of rich routing capabilities, including advanced HTTP routing, granular header matching, flexible path rewriting, and sophisticated traffic splitting mechanisms. These features offer a significantly greater degree of control and flexibility over how traffic is directed within the Kubernetes environment. The API natively supports advanced traffic control requirements, such as implementing canary releases and blue-green deployments, through features like weighted routing. Furthermore, the Policy Attachment model allows for the decoration of Gateway API objects with implementation-specific Custom Resource Definitions (CRDs) to apply cross-cutting concerns like security policies and rate limiting.

The native inclusion of features such as header-based matching and traffic weighting marks a substantial improvement. These capabilities were previously complex or non-standard with Ingress, often requiring cumbersome workarounds. Their direct integration into the Gateway API means that advanced deployment strategies like canary releases and A/B testing can now be implemented directly and portably within the Kubernetes API itself, rather than relying on external tools or controller-specific annotations. This development significantly streamlines DevOps practices, enabling faster and safer software delivery. It reduces the need for complex external orchestration for common deployment patterns, making the Kubernetes platform more self-sufficient and lowering the cognitive load on development teams. The native support for these features reflects a maturation of the API, designed to directly support and facilitate modern software engineering methodologies.

5. Practical Migration from Ingress to Gateway API

5.1. Pre-Migration Planning and Assessment of Dependencies

Before embarking on a migration from Ingress to Gateway API, a thorough assessment of several critical factors is imperative:

- Ingress Controller Support: It is essential to determine if the currently deployed Ingress

controllers are compatible with or supported by the Gateway API.

- Migration Strategy: A strategic decision must be made regarding the migration approach: either a complete, one-time replacement of Ingress or a phased transition where both Ingress and Gateway API operate concurrently. Experience suggests that a one-time conversion for primary resources, followed by gradual onboarding of other services, can be an effective strategy.
- Custom Dependencies: Organizations must meticulously identify any custom annotations, BackendConfigs, or specific load balancer features that the existing Ingress setup relies upon. It is important to note that some Ingress annotations may not have direct 1:1 mappings in the Gateway API, necessitating alternative solutions or equivalent controller support.
- DNS Management: A clear understanding of the current DNS setup is crucial, with a detailed plan for potential A record updates that may be required if the new Gateway exposes a different IP address.
- Certificate Management: The existing certificate management approach (e.g., pre-shared certificates, cert-manager, or cloud provider-managed certificates) must be reviewed, and a plan for its seamless integration with the Gateway API should be developed.

The planning considerations underscore that migration is not merely a YAML conversion but a complex process that impacts multiple layers of the infrastructure, including DNS, certificates, and custom configurations. The challenge of non-1:1 annotation mapping highlights that unique, controller-specific Ingress features might require re-architecting or finding new solutions within the Gateway API, rather than a direct translation. This reveals the inherent costs and complexities of transitioning from a less standardized API, such as Ingress with its numerous annotations, to a more standardized one. It necessitates a holistic assessment of the entire networking stack and its dependencies, extending beyond just the Ingress resources themselves. A thorough pre-migration audit is therefore critical to identify potential friction points and avoid unexpected disruptions, emphasizing that technical migrations are often as much about meticulous operational planning as they are about code changes.

5.2. Architectural Comparison: Ingress-Based vs. Gateway API-Based Deployments

The architectural shift from an Ingress-based setup to a Gateway API-based deployment represents a significant evolution in how traffic is managed within Kubernetes.

5.2.1. Before (Ingress-Based):

In a traditional Ingress setup, external traffic flows through an External Load Balancer, which then directs requests to an Ingress Controller. This controller, in turn, routes traffic to the appropriate Kubernetes Service, which finally distributes it among the backend Pods. DNS A records typically point to the Ingress IP, and BackendConfig resources are often used for specific load balancer features or timeouts.

Figure 1. Traffic Flow

A key architectural characteristic of this model is its monolithic nature. The Ingress resource itself conflates concerns from different operational roles. It defines both infrastructure-level configuration (the logical entry point for traffic) and application-level routing rules (path-based routing to services). This often leads to conflicts and a lack of clear ownership. Furthermore, to implement advanced features such as TLS configuration, timeouts, or weighted traffic splitting, the Ingress specification relies heavily on non-standard, vendor-specific annotations or Custom Resource Definitions (CRDs) like Backend Config. This results in a lack of portability and a fragmented user experience across different Kubernetes environments.

5.2.2. After (Gateway API-Based):

With the Gateway API, the flow is more granular. An External Load Balancer directs traffic to a Gateway resource. The Gateway then uses HTTP Route (or other Route types) to define how traffic is forwarded to the Service, which ultimately reaches the Pods. DNS A records are updated to point to the Gateway's IP, and certificate management is integrated at the Gateway level. Additional configurations like Backend Config and Gateway policies can be applied.

5.2.3. Certificate Management:

The following table provides a side-by-side comparison of the architectural components:

Table 1. Architectural Components: Ingress vs. Gateway API

Component Type	Ingress-Based Deployment	Gateway API-Based Deployment
External Load Balancer	External LB	External LB
Controller	Ingress Controller	Gateway (managed by a GatewayClass controller)
Routing Resource	Ingress	HTTPRoute (or other Route types like TCPRoute,

		TLS Route)
Backend Service	Service	Service
DNS	DNS A Record (pointing to Ingress IP)	DNS A Record (pointing to Gateway IP)
Certificate Management	Often external or via Ingress annotations	Integrated via Gateway annotations or cert-manager
Policy/Config	Backend Config (controller-specific annotations)	Backend Config / Gateway Policy (explicit resources/CRDs)

This table offers a clear, side-by-side visual representation of the architectural transformation, aiding in a quick understanding of the shift in responsibility and component interaction. By explicitly mapping components, it clarifies how familiar concepts are re-represented or replaced in the new API, facilitating a deeper comprehension of the conceptual transition beyond mere YAML changes.

The architectural comparison clearly illustrates a decomposition of the Ingress Controller's responsibilities into distinct Gateway API resources (Gateway Class, Gateway, HTTP Route). The Ingress Controller often bundled the load balancer logic, routing rules, and policy enforcement into a single entity. In contrast, the Gateway API explicitly separates these concerns. This decoupling enables greater modularity and independent evolution of components. Infrastructure providers can manage the underlying Gateways, cluster operators can enforce policies on them, and application developers can define their specific routes, all with clearer boundaries of responsibility. This modularity is fundamental to scaling traffic management effectively in large, complex organizations, as it reduces inter-team dependencies and allows for more specialized tooling and expertise at each layer.

5.3. Leveraging Migration Tools: The Role and Limitations of ingress2gateway

The ingress2gateway CLI tool is designed to assist in the migration process by converting existing Ingress resources into the Gateway API format. This tool can serve as a valuable starting point for organizations undertaking the transition. However, its utility has specific limitations: it currently supports only the ingress-nginx provider, which restricts its universal applicability. Furthermore, the tool often requires subsequent manual editing of the generated YAML configurations to precisely align with the nuances of specific load balancers or controller implementations.

The observation that ingress2gateway functions as a "scaffold, not a final product" and necessitates "post-editing" highlights a crucial aspect of complex system migrations. While automation tools can significantly accelerate the initial phase of conversion, they are inherently limited in their ability to fully capture the intricacies of existing custom configurations, specific vendor implementations, or unique operational requirements. This implies that human expertise and manual refinement remain indispensable throughout such migrations. It underscores that declarative APIs, while simplifying ongoing management, still demand a deep understanding and hands-on intervention when transitioning from legacy systems. The "last mile" of migration frequently involves bespoke adjustments, emphasizing the enduring value of skilled engineers in an increasingly automated technological landscape.

5.4. Step-by-Step Migration Walkthrough with Configuration Examples

To illustrate the practical migration process, consider a common original setup and the corresponding steps to transition to the Gateway API.

5.4.1. Original Setup Example:

A typical Ingress setup might involve an Ingress resource exposed via an external load balancer. A DNS A record points to the Ingress Controller's IP address. Backend Config resources are defined for specific cloud provider features like Cloud Armor or custom timeouts. The backend application runs within a Deployment and Pods, exposed via a Kubernetes Service of type NodePort or Cluster IP on a specific port, such as 8080.

Migration Process Steps:

1. Create a Gateway Class: This resource defines the controller responsible for implementing the Gateway. For instance, if using an NGINX-based gateway controller, the controller Name would specify k8s.io/gateway-nginx.
   YAML
   ```
   apiVersion: gateway.networking.k8s.io/v1
   kind: GatewayClass
   metadata:
     name: external-gateway
   spec:
     controllerName: k8s.io/gateway-nginx
   ```

2. Define a Gateway: This resource represents the actual entry point for traffic, such as a cloud load balancer. It specifies listeners for particular protocols (e.g., HTTP) on designated ports (e.g., 80) and hostnames (e.g., "example.com").
   YAML
   ```
   apiVersion: gateway.networking.k8s.io/v1
   kind: Gateway
   metadata:
     name: my-gateway
     namespace: default
   spec:
     gatewayClassName: external-gateway
     listeners:
     - name: http
   ```

```
    port: 80
    protocol: HTTP
    hostname: "example.com"
```

3. Replace Ingress with HTTP Route: The HTTP Route resource defines the specific routing rules for HTTP traffic, effectively replacing the functionality of the legacy Ingress resource. It specifies how incoming requests, based on criteria like path prefixes, should be directed to backend services.

```
YAML
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: my-app-route
  namespace: default
spec:
  parentRefs:
  - name: my-gateway
  rules:
  - matches:
    - path:
        type: PathPrefix
        value: "/"
    backendRefs:
    - name: my-service
      port: 8080
```

The step-by-step process illustrates how the Gateway API achieves its flexibility and power through the composition of multiple, smaller, and role-specific resources (Gateway Class, Gateway, HTTP Route). This contrasts with the more monolithic nature of the Ingress resource. This compositional approach significantly enhances clarity, auditability, and the ability to delegate management responsibilities. Each resource can be managed by a different team or automated process, allowing for more granular control and reducing the blast radius of changes. This embodies the core Kubernetes philosophy of declarative configuration, but applied with a more refined and modular design, ultimately leading to a more maintainable and scalable system.

5.5. Addressing Common Migration Challenges: DNS, Certificate Management, and Annotation Equivalencies

During the migration to Gateway API, several common challenges typically arise, requiring careful planning and execution.

- DNS Consideration: If the newly created Gateway exposes a new external load balancer IP address, it is imperative to update the corresponding A record in the DNS system to point to this new IP. DNS propagation during this switch can introduce short-term disruptions if not meticulously planned. It is advisable to adjust the Time-To-Live (TTL) settings for DNS records accordingly to minimize potential downtime.
- Certificates: The Gateway API facilitates a transition towards more streamlined certificate management. Organizations can now leverage managed certificates through tools like cert-manager or directly utilize cloud provider-managed certificates via Gateway annotations or controller-specific configurations. Centralizing TLS termination at the Gateway simplifies certificate management and enhances overall security for HTTPS traffic across applications.
- Backend Config and Gateway Policy: The Gateway API supports the creation of equivalent Backend Config and other Gateway policies for various configurations, such as SSL settings, health checks, and other required parameters. However, a notable challenge is that some annotations from the legacy Ingress API, such as backendConfig.cloud.google.com, may not have direct 1:1 mappings in the Gateway API. This necessitates ensuring that the chosen Gateway API controller provides equivalent functionality or requires re-architecting the specific feature.

The challenges highlighted, including DNS propagation, certificate management, and annotation equivalencies, demonstrate that a seemingly "simple" API migration can have ripple effects across the entire cloud-native stack. DNS is critical for service reachability, certificates are fundamental for security, and annotations provide custom features. A change in one layer, specifically the traffic management API, necessitates careful consideration and potential adjustments in other interconnected components. This reinforces the concept that cloud-native systems are inherently highly interconnected. Successful migrations require not only a thorough understanding of the new API but also a deep knowledge of the existing infrastructure, its dependencies, and how various components (e.g., DNS, cert-manager, cloud load balancers) interact. This interconnectedness emphasizes the critical need for comprehensive testing and robust rollback strategies that account for the potential ripple effects throughout the entire system.

5.6 Migration Execution: A Strategy for Monitoring and Validation

While Section 5.5 outlined the static challenges inherent in the migration, the dynamic execution phase requires a robust strategy to ensure service continuity and validate success. This section details a phased methodology for executing the migration, monitoring its progress in real-time, and formally asserting its successful completion [1].

5.6.1. Phased Rollout and Live Traffic Shaping

A direct, "big bang" cutover from Ingress to Gateway is fraught with risk. A more prudent, industry-standard approach involves a parallel deployment and a gradual traffic shift, often referred to as a canary or blue-green deployment strategy [7].

- Parallel Deployment: The first step is to deploy the new Gateway API resources (Gateway,

HTTPRoute, GatewayClass, and associated policies) in parallel with the existing Ingress and BackendConfig resources. At this stage, the new Gateway will be provisioned with a distinct external IP address, but no production traffic will be directed to it. This allows for internal validation of the new configuration against a non-production endpoint.

- DNS-Based Traffic Shaping: The transition of live traffic is most effectively managed at the DNS layer. By utilizing weighted DNS records (e.g., AWS Route 53 Weighted Routing, Google Cloud DNS Weighted Round Robin), an operator can precisely control the percentage of traffic directed to the legacy Ingress IP versus the new Gateway IP. A typical rollout schedule proceeds as follows:
  - Phase 1 (1-5% Traffic): A small fraction of traffic is shifted to the Gateway API endpoint. This phase is critical for observing the new stack under real-world load without significant user impact.
  - Phase 2 (10-50% Traffic): As confidence in the new stack grows, the traffic percentage is incrementally increased. Continuous monitoring is essential during this phase.
  - Phase 3 (100% Traffic): Once all metrics indicate stability, 100% of traffic is directed to the Gateway. The legacy Ingress remains operational as a rapid rollback path.

This phased approach provides a crucial safety mechanism; any detected anomaly can trigger an immediate rollback by reverting the DNS weights to favor the legacy Ingress IP, minimizing the mean time to recovery (MTTR).

5.6.2. Multi-Layered Monitoring During Transition

Effective monitoring during the migration requires observing telemetry from multiple layers of the stack to gain a holistic view of system health.

- Infrastructure-Level Metrics: Monitor the cloud provider's load balancer metrics for both the old and new endpoints. Key Performance Indicators (KPIs) include:
  - Request Count: Should decrease on the legacy load balancer and proportionally increase on the new one.
  - End-to-End Latency (p50, p90, p99): Must remain within or below established baselines
  - HTTP Error Rates: Server-side 5xx and client-side 4xx error rates should not increase. A spike in 5xx errors on the new endpoint is a critical signal for immediate rollback.
- Controller-Level Metrics: The Gateway and Ingress controllers themselves expose vital Prometheus metrics. Monitor the request throughput, configuration reloads successes/failures, and processing latency for both controllers to ensure they are functioning as expected.
- Application-Level Metrics: The ultimate source of truth is the application itself. Monitor application-specific dashboards for business-critical metrics, such as user transaction success rates, application error logs, and internal service-to-service communication latency. Any degradation here indicates a potential issue with the new traffic routing configuration.

5.6.3. Asserting Migration Success and Decommissioning

The migration cannot be considered complete until its success is formally verified and legacy artifacts are retired.

- Defining Success Criteria: Success should be measured against a pre-defined set of criteria: A typical rollout schedule proceeds as follows:
  - Sustained Stability: The new Gateway API stack has handled 100% of production traffic for a sustained observation period (e.g., 24-72 hours) without any degradation in the KPIs mentioned above(5.6.2).
  - Functional Parity Verification: A comprehensive suite of automated end-to-end and integration tests must be executed and passed against the new endpoint. This confirms that all required functionality previously managed by Ingress annotations or BackendConfig (e.g., custom timeouts, health check configurations, security policies) is correctly implemented in the Gateway API configuration.
  - Observability Confirmation: All monitoring dashboards, logging queries, and alerting rules have been successfully migrated to target the new Gateway resources and are confirmed to be reporting accurately.
- Decommissioning Legacy Resources: Once all success criteria are met, the final step is to decommission the legacy resources to reduce complexity and cost. This involves the systematic deletion of the Ingress and BackendConfig resources, followed by the release of the associated external load balancer and its IP address. This final act formally concludes the migration project.

6. Results and Discussion

6.1. Observed Improvements in Separation of Concerns and Operational Clarity

The adoption of the Gateway API demonstrably improves the separation of concerns, leading to a cleaner decoupling of infrastructure and application layer routing. This architectural refinement significantly enhances maintainability and empowers different teams to manage their respective domains more independently. Consequently, clearer ownership boundaries are established, which in turn reduces cross-team friction and improves the clarity of Responsible, Accountable, Consulted, and Informed (RACI) charts within an organization.

The "separation of concerns" and "clearer ownership boundaries" are not merely technical benefits; they directly translate into improved organizational efficiency and a reduction in inter-team conflicts. By aligning API resources with distinct organizational roles - Infrastructure Provider, Cluster Operator, and Application Developer - the Gateway API facilitates a more streamlined and efficient workflow. This highlights that architectural decisions in cloud-native environments have profound organizational impacts. A well-designed API can significantly reduce communication overhead, empower teams with greater autonomy over their specific domain, and ultimately accelerate software delivery cycles. This demonstrates how technical elegance can lead to operational excellence and foster better team collaboration, effectively illustrating Conway's Law in action.

6.2. Analysis of Enhanced Routing Flexibility and Scalability

The Gateway API provides a robust set of rich routing capabilities, including advanced HTTP routing, header matching, path rewriting, and sophisticated traffic splitting. These advanced features offer significantly greater control and flexibility over how traffic is directed within the cluster. A key improvement over Ingress is its support for multi-protocol traffic, encompassing HTTP, HTTPS, TCP, UDP, and TLS, thereby addressing a critical limitation of its predecessor. Furthermore, the API enables the scaling of routing logic independently per team or environment without interference, which promotes superior scalability across diverse teams and multiple clusters.

The native support for advanced routing features such as traffic splitting and header matching directly enables more sophisticated deployment strategies, including canary releases and A/B testing, to be implemented consistently across an organization. These strategies were previously cumbersome or non-standard with Ingress. This significantly streamlines DevOps practices, leading to faster and safer software delivery. It reduces the reliance on complex external tooling or custom scripts, simplifying the CI/CD pipeline and enhancing the overall reliability of application updates. This positions the traffic management layer as a crucial enabler for continuous delivery and experimentation within cloud-native environments [4].

6.3. Impact on Role-Based Access Control and Security Post-Migration

The Gateway API significantly improves Role-Based Access Control (RBAC) through the establishment of clear resource boundaries among Gateway, Gateway Class, and HTTP Route objects. This structured approach allows for more precise control over which roles can define and manage different aspects of traffic routing. Policies within the Gateway API further aid in enforcing security and access controls, enabling functionalities such as limiting traffic to specific IP addresses or applying rate limiting policies. Additionally, centralizing TLS termination at the Gateway simplifies certificate management and enhances the overall security posture by ensuring consistent HTTPS enforcement across applications.

The enhanced RBAC and policy attachment capabilities of the Gateway API facilitate a "shift-left" approach to security and governance. Instead of security being an afterthought or applied inconsistently, policies can be defined and enforced at the Gateway and Route levels by cluster operators, providing a centralized control point for ingress traffic. This leads to a more secure and compliant posture for cloud-native applications. By embedding security and access control directly into the API definition and enforcing it through explicit resource boundaries, organizations can effectively reduce the attack surface, ensure consistent policy application, and simplify auditing processes. This move towards declarative security policies at the network edge is critical for managing risk effectively in dynamic microservices environments.

6.4. Future-Proofing Infrastructure with Gateway API

The Gateway API is explicitly designed to support future advancements in cloud-native networking, including protocols like gRPC, mesh-style routing, and other advanced features. This forward-looking design positions it as a more sustainable long-term solution for traffic management. The Kubernetes community's active promotion of the Gateway API as the successor to Ingress clearly indicates its strategic importance, ensuring that infrastructure adopting it is prepared for upcoming developments in the ecosystem. The API integrates seamlessly with Kubernetes' native resources and is designed to work effectively with service meshes such as Istio and Linkerd.

The explicit design for "future-ready extensibility" and its strong alignment with the Kubernetes SIG-NETWORK roadmap signifies that adopting the Gateway API is not merely a tactical migration but a strategic investment. It positions an organization's infrastructure to seamlessly integrate with evolving cloud-native patterns like service meshes and new protocols (e.g., gRPC). This provides a compelling argument for early adoption, as it reduces future technical debt and ensures compatibility with upcoming ecosystem advancements. Organizations that migrate now will be better equipped to leverage emerging technologies and respond effectively to changing business requirements, demonstrating foresight and a commitment to modern cloud-native practices.

6.5. Comparison of Practical Outcomes Against Theoretical Advantages

The practical outcomes observed during real-world migrations largely validate the theoretical advantages attributed to the Gateway API. The benefits of separation of concerns, rich routing capabilities, enhanced RBAC, and future-readiness, as detailed in the API's design principles, are indeed realized in operational environments. While challenges such as DNS propagation during transitions and

the mapping of legacy Ingress annotations to Gateway API equivalents exist, these can be effectively mitigated through meticulous planning and careful post-editing of tool-generated configurations. Despite the upfront effort required for migration, the long-term benefits in terms of maintainability, scalability, and enhanced security are consistently deemed worthwhile. The following table provides a direct, comparative summary of the two APIs' capabilities, reinforcing the central argument of this paper. This table serves as a concise, comparative summary of the two APIs' capabilities. It allows for a quick identification of the specific areas where the Gateway API offers significant improvements over Ingress, thereby reinforcing the central argument of this paper. It also provides a valuable reference point for decision-makers evaluating the benefits of migration.

Table 2. Comparison of Kubernetes Ingress vs. Gateway API Features

Feature	Kubernetes Ingress	Kubernetes Gateway API
Protocol Support	Primarily HTTP/HTTPS	HTTP/HTTPS, TCP, UDP, TLS, gRPC
Extensibility Model	Annotation-based, controller-specific	Explicit API extension points (CRDs)
Role Separation	Limited/Implicit	Explicit (Role-Oriented: Infra, Operator, Developer)
Multi-Tenancy Support	Challenging, often requires external tooling	Built-in, robust via role separation and policy attachment
Advanced Routing	Limited (via annotations)	Native, declarative (header matching, traffic splitting)
Policy Enforcement	Limited (via annotations)	Native, declarative (via Policy Attachment)
Community Adoption/Future-Proofing	Legacy, successor planned	Future-forward, community-backed, active development
Configuration Complexity	Higher for advanced use cases due to ad-hoc extensions	Lower for advanced use cases due to structured design

7. Conclusion

The migration from Kubernetes Ingress to the Gateway API represents a pivotal advancement in Kubernetes traffic management, marking a strategic shift from simple HTTP routing to a more sophisticated, extensible, and role-oriented paradigm. This paper has detailed the inherent limitations of traditional Ingress, particularly its tight coupling of configuration, restricted routing capabilities, and challenges in managing multi-tenant environments. In stark contrast, the Gateway API offers a robust and forward-looking solution through its principled design, clearly defined and composable resources (Gateway Class, Gateway, Route), and native support for advanced traffic management features.

While the practical aspects of migration involve careful planning, meticulous attention to DNS considerations and necessary refinements of tool-generated configurations, the observed benefits unequivocally justify the transition. The enhanced operational clarity, superior routing flexibility, improved security posture through refined RBAC and policy enforcement, and the inherent future-proofing of infrastructure collectively demonstrate the significant value proposition of the Gateway API. As the Kubernetes community continues to champion the Gateway API as the future standard for ingress and service mesh functionalities, its adoption becomes crucial for organizations striving to build resilient, scalable, and maintainable cloud-native applications that can adapt to evolving demands and technological landscapes.

References

[1] Adusumilli, L. V. P. (2025). Serverless Kubernetes: The Evolution of Container Orchestration. European Journal of Computer Science and Information Technology, 13(30), 20-36.DOI: 10.37745/ejcsit.2013/vol13n302036.

[2] Al-Dhuraibi, M., Al-Khawlani, M., & Al-Hakimi, A. (2025). Kubernetes Security: A Comprehensive Analysis of Architectural Hardening, Access Control, and Compliance. MDPI. https://doi.org/10.3390/jcp5020030

[3] Chippagiri, S., Ravula, P., & Gangwani, D. (2024). Optimizing Kubernetes Network Performance: A Study of Container Network Interfaces and System Tuning Profiles. European Journal of Theoretical and Applied Sciences, 2(6), 651-668. DOI: 10.59324/ejtas.2024.2(6).58.

[4] Egbuna, J. (2025). The Evolution and Impact of Kubernetes in Modern Software Engineering: A Review. International Journal of Academic and Applied Research (IJAAR), 9(4), 56-63. Retrieved from http://ijeais.org/wp-content/uploads/2025/4/IJAAR250406.pdf

[5] Gudelli, V. R. (2021). Kubernetes-based orchestration for scalable cloud solutions. International Journal of Novel Research and Development (IJNRD), 6(9), 36-40. https://ijnrd.org/papers/IJNRD2109006.pdf

[6] Kubernetes Gateway API. (n.d.). https://gateway-api.sigs.k8s.io

[7] O'Dwyer, N., & O'Sullivan, B. (2025). Continuous Deployment: A Review of Current Practices and Future Directions. arXiv preprint arXiv:2501.07204. Retrieved from https://arxiv.org/pdf/2501.07204

[8] Ogunrinde, V., Mehra, A., & Martin, O. (2025). Understanding Kubernetes Networking: A Practical

Guide to Architecture, Implementation, and Best Practices. DOI: 10.34218/IJRCAIT_08_01_234

[9] Upendra, Kanuru. (2025). Enterprise Security Strategy Framework for Electronic Health Record Organizations. International Journal of Advanced Research in Computer and Communication Engineering, 14(5), 1-8. https://doi.org/10.17148/IJARCCE.2025.1450 1