OFFICE OF THE CTO

Distributed Gateway Actors: Evolving API Management

By Rajesh Narayanan, Applications and Security

Reviewed by and Contributions from: Ian Smith, Sam Bisbee, Andrew Stiefel, Lori MacVittie, Mike Wiley and others.

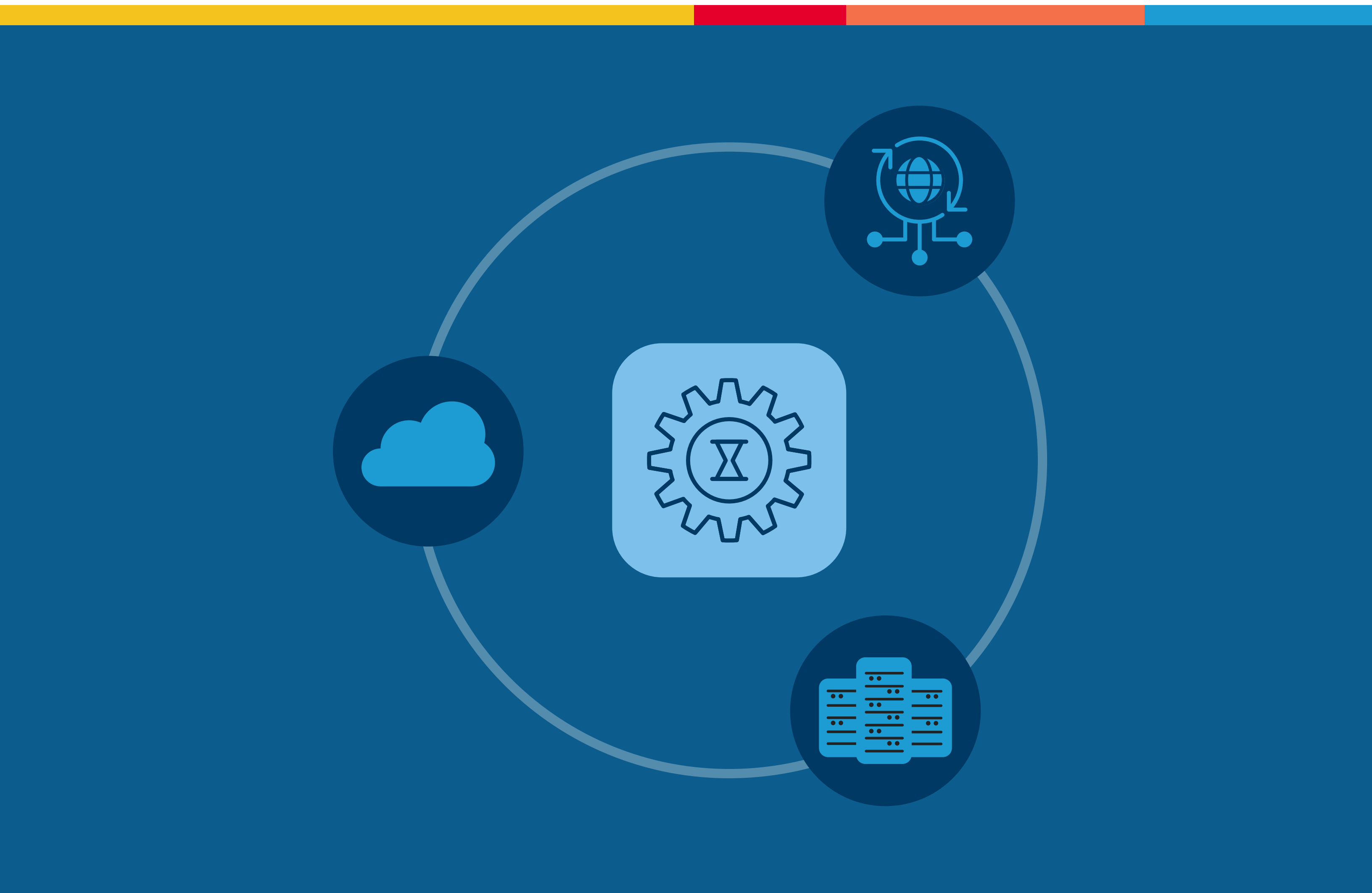


Table of Contents

F5 Office of the CTO Opinion

The Office of the CTO team at F5 has been exploring the technology field related to APIs for over a year now. This latest white paper is a continuation of our efforts to understand the ever-evolving API ecosystem.

The challenges we detailed with managing API sprawl will lead to challenges with API *gateway* sprawl, and traditional approaches to addressing these challenges will not be sufficient. While graph technologies such as GraphQL hold great promise with respect to taming the complexity associated with APIs, it is only part of the solution as the challenges go beyond connectivity, security, and routing. The right approach, based on nearly thirty years of expertise with scaling systems and applications, is based on a distributed—not federated—architecture that employs GraphQL but is not solely reliant on it.

This paper explores a distributed architectural approach to addressing the challenges of API gateway sprawl.

Abstract

The digital economy will be powered through APIs, giving us an *API-driven economy*. Following the API Sprawl white paper, our pursuit was to understand how to eliminate or alleviate the impact of API sprawl. While GraphQL seemed promising to reduce the ramifications of API sprawl, it tends to put the onus on developers to rewrite a large part of their API codebase. However, an emerging viewpoint around GraphQL is its ability to be used as an effective *gateway actor*. The gateway actor is a function or process created near the client that initiates an API request. This gateway actor acts as the initial API gateway terminating the API request. It can also be ephemeral, such that it can be destroyed after servicing the request.

In addition to developers and operations teams prioritizing API management over API gateways, we also discovered the issue of *API gateway sprawl* because of API sprawl. From a developer's perspective, the main concern is ensuring the API functions correctly and in a timely manner. On the other hand, the operations team is more focused on aspects such as discovery, security, usability, and access controls. Since API gateways today are a critical component of API infrastructure, it became evident that the proliferation of APIs increases the deployment of API gateways—leading to API gateway sprawl.

The future of the API architecture needs to evolve to accept API sprawl while simplifying deployment and operations. The architecture proposed is the next evolution of where API gateway design patterns need to be. While GraphQL is not central to the architecture, or a necessity, it has an ability to enhance the gateway actor pattern.

Summary

The API management ecosystem must move away from managing API gateway monoliths towards a more contemporary system design approach. This will result in a more agile and effective API management ecosystem.

API GATEWAY SPRAWL– MANAGING DISTRIBUTED MONOLITHS

API sprawl, already a challenge within the API economy, also results in *API gateway sprawl*, a situation where managing APIs has become uncontrollable due to diverse API gateway vendor technologies and opinionated teams managing the API gateways. We are at an inflection point in API architectures as the legacy API gateway (API-GWs)—a dedicated software layer that acts as the entry point for API calls—is no longer sufficient to manage the scale and complexity of the emerging API ecosystem.

Figure 1 illustrates how we have moved from managing API sprawl to managing API gateway sprawl.

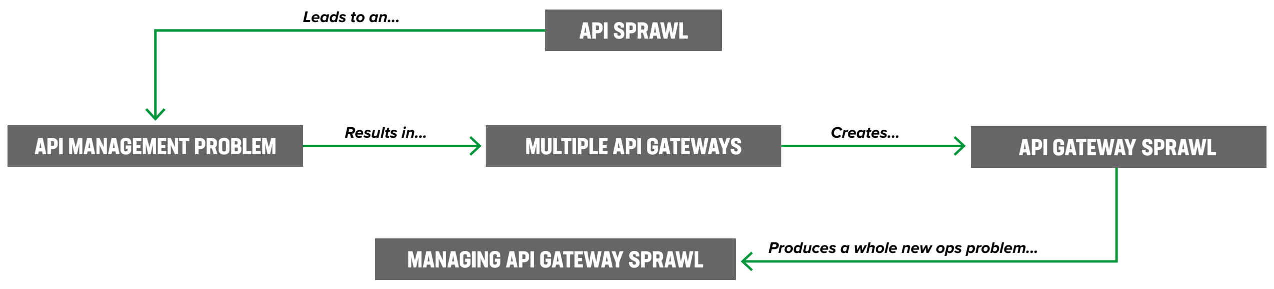


Figure 1: From API sprawl to API gateway sprawl

The design pattern above alludes to a centralized control plane, with the API gateways forming the distributed data plane as shown in Figure 2.

API gateways are an essential component of modern software architectures, providing a central point of control and security for APIs. However, as the number of features offered by API gateways has grown, they have become increasingly complex and difficult to manage. In many cases, these gateways have evolved into monolithic systems, with a wide range of functionalities bundled together in a single package.

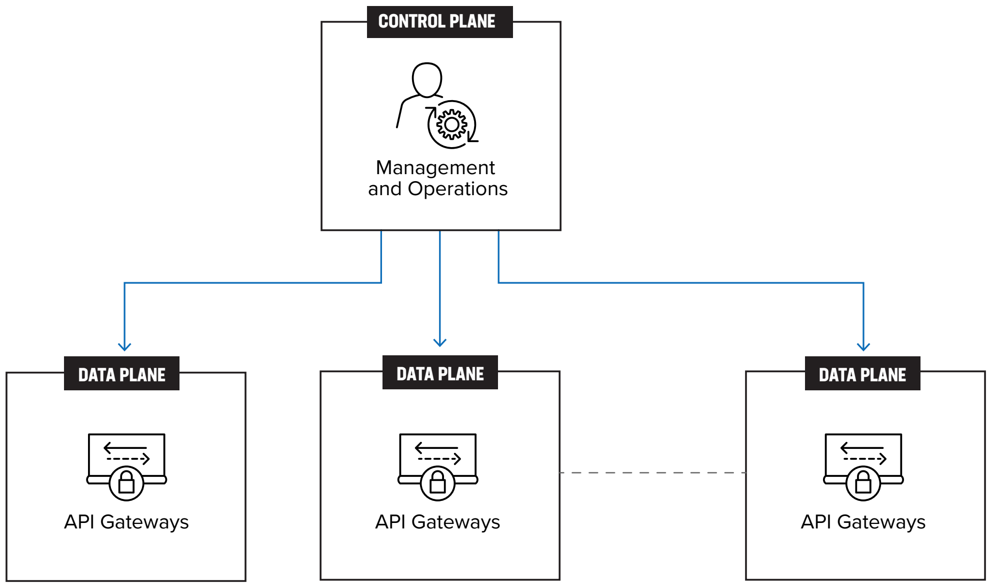


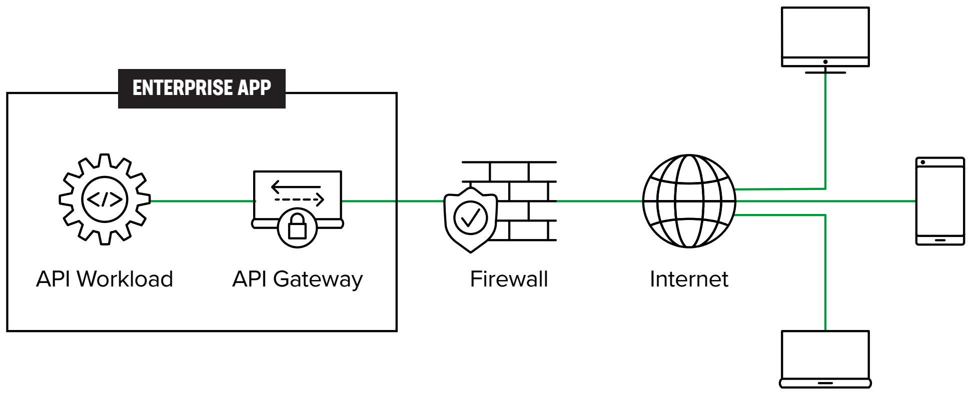
Figure 2: Manage distributed monoliths

While managing multiple gateways may appear to be a distributed design, the reality is that it falls short of true distribution. This is because the gateways are still tightly coupled, sharing resources and configurations that are difficult to separate. As a result, many enterprises end up managing distributed monoliths and all the challenges that creates.

LEGACY API GATEWAY ARCHITECTURES

Figure 3 shows the architecture of existing API gateways. API requests originating from the client are first transmitted via a shared or dedicated network, passing through a firewall before reaching the API gateway. The API gateway, which acts as a reverse proxy, then forwards the traffic to the API workload.

Figure 3: Legacy API gateway architectures



The legacy API-GW becomes an API management choke point when API gateways are deployed across the enterprise, or when the API workloads operationally move across regions, zones, multiple clouds, or to the edge while contending with API sprawl. Multiple API-GWs are spun up by different teams without a single point of enterprise management and control. If an individual or group of API services moves to a different infrastructure (cloud or otherwise), the operations team must have a method to migrate all the aspects of API management—registration, discovery, authentication, networking, security, and governance-risk-and-compliance (GRC) policies.

The architecture depicted in Figure 3 is thus not scalable or practical long term, as over time it leads to managing distributed monoliths (Figure 2).

There are two problems creating the API gateway choke point: (1) API gateway vendor sprawl, and (2) scale as you go when an enterprise has more applications running in more places.

1. **API gateway vendor sprawl:** Dealing with API gateway vendor sprawl is a human challenge, as it can be difficult to convince all teams to adopt a single API gateway vendor, and migrating to a new vendor can be a cumbersome task. As a result, organizations end up spending time and resources managing multiple gateway vendors. Although this problem can be addressed, it may not be entirely feasible in reality.

2. **Application scaling:** Scaling applications is a problem when the application needs to support more users within a single location or needs to be deployed at multiple locations. This requires the application to scale horizontally or vertically. However, as the application scales, the API gateways need to scale as well, and in some cases, they need to be deployed at multiple locations to support the scaling based on current architecture patterns. This can make managing the API gateways operationally complex.

SOLUTION: A DISTRIBUTED GATEWAY ACTOR PATTERN

Figure 4 depicts a distributed gateway actors' pattern to address API gateway sprawl. While the distributed pattern itself is not new, this paper formalizes it within the context of API gateways. The clients initiate the API request. The distributed gateway actors are ephemeral and instantiated on-demand as close to the client as possible. It becomes the responsibility of the API transport layer to send the API request from the gateway actor to the simplified API gateway, which then routes the request to the appropriate API workload. While the use of GraphQL support in the distributed actors is more of a detail than a requirement for this pattern to work, it does enable supporting features like service orchestration. So instead of creating a new service orchestration layer, GraphQL could provide that support.

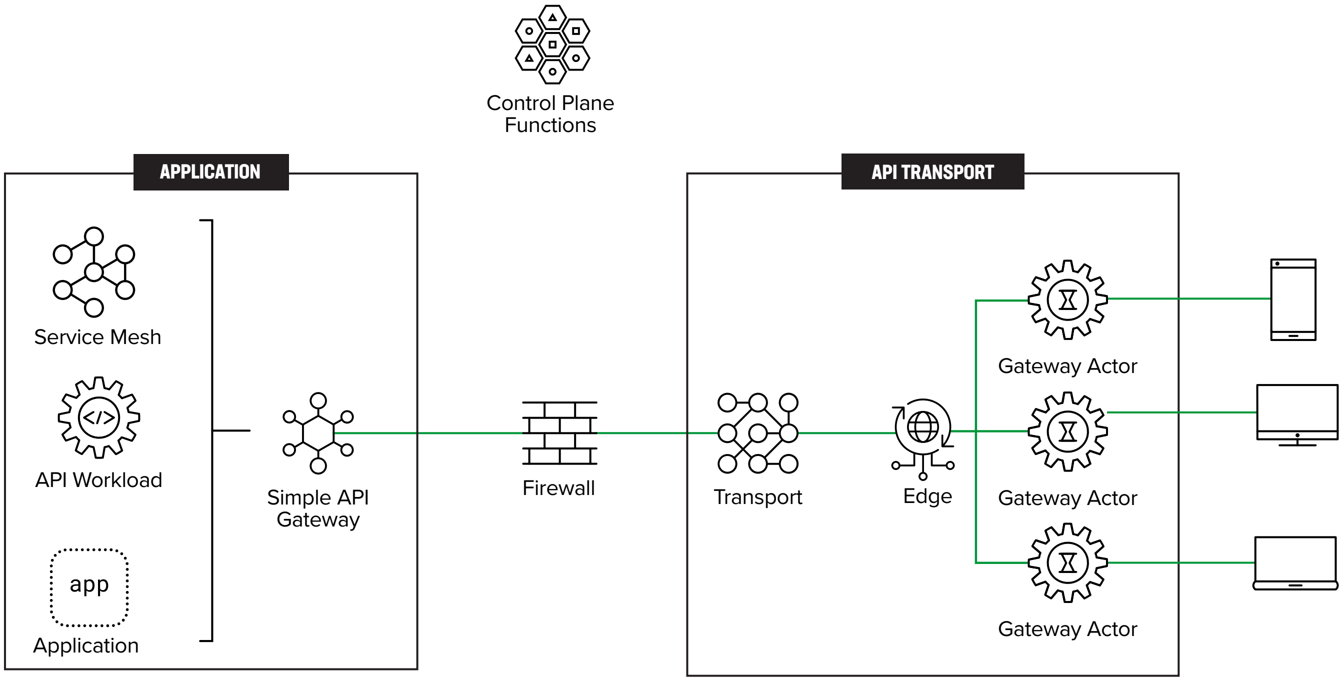


Figure 4: GraphQL-based distributed gateway actors

To clarify, the traffic pattern is from right to left. That is, the clients are on the right and the API requests are initiated by the client as shown in Figure 5.

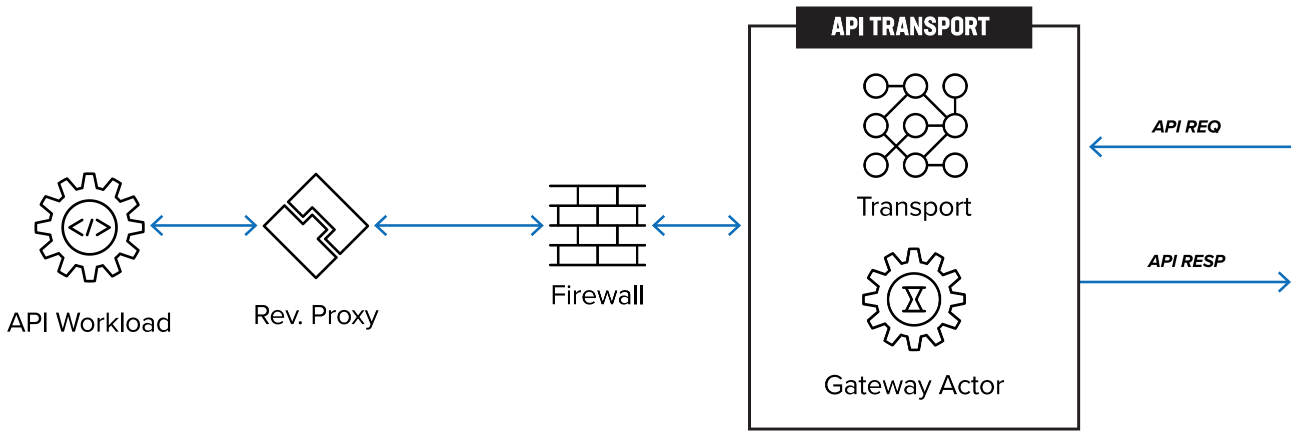


Figure 5: Traffic flows from client (right) to API workload (left)

There is an emerging deployment pattern using gateway actors to replace or reduce over dependency on API gateways for managing APIs within and across an enterprise. While GraphQL is not necessary for the architecture, the introduction of it is timely as the gateway actor is the right vehicle to support GraphQL.

THERE IS AN EMERGING DEPLOYMENT PATTERN USING GATEWAY ACTORS TO REPLACE OR REDUCE OVER DEPENDENCY ON API GATEWAYS FOR MANAGING APIS WITHIN AND ACROSS AN ENTERPRISE.

To gain a deeper understanding of the gateway actor as a potential solution, it's necessary to closely examine the current state of API infrastructures, particularly API gateways. This is because we have identified gateway sprawl as a significant contributor to the challenges of operating and scaling API infrastructures.

Role of API Gateways in API Management

To gain a better understanding of API gateways, it is important to first examine the various components of the modern API management infrastructure.

API MANAGEMENT INFRASTRUCTURE AND FUNCTIONS

Figure 6 offers a comprehensive visual representation of the various features and components that are integral to API management. In addition to API gateways, which are required for routing and managing traffic to backend services, there are several other important infrastructure components. These may include solutions for authentication, rate limiting, caching, and service mesh, among others. By incorporating these features, organizations can achieve control over their APIs, enhance security, and optimize performance.

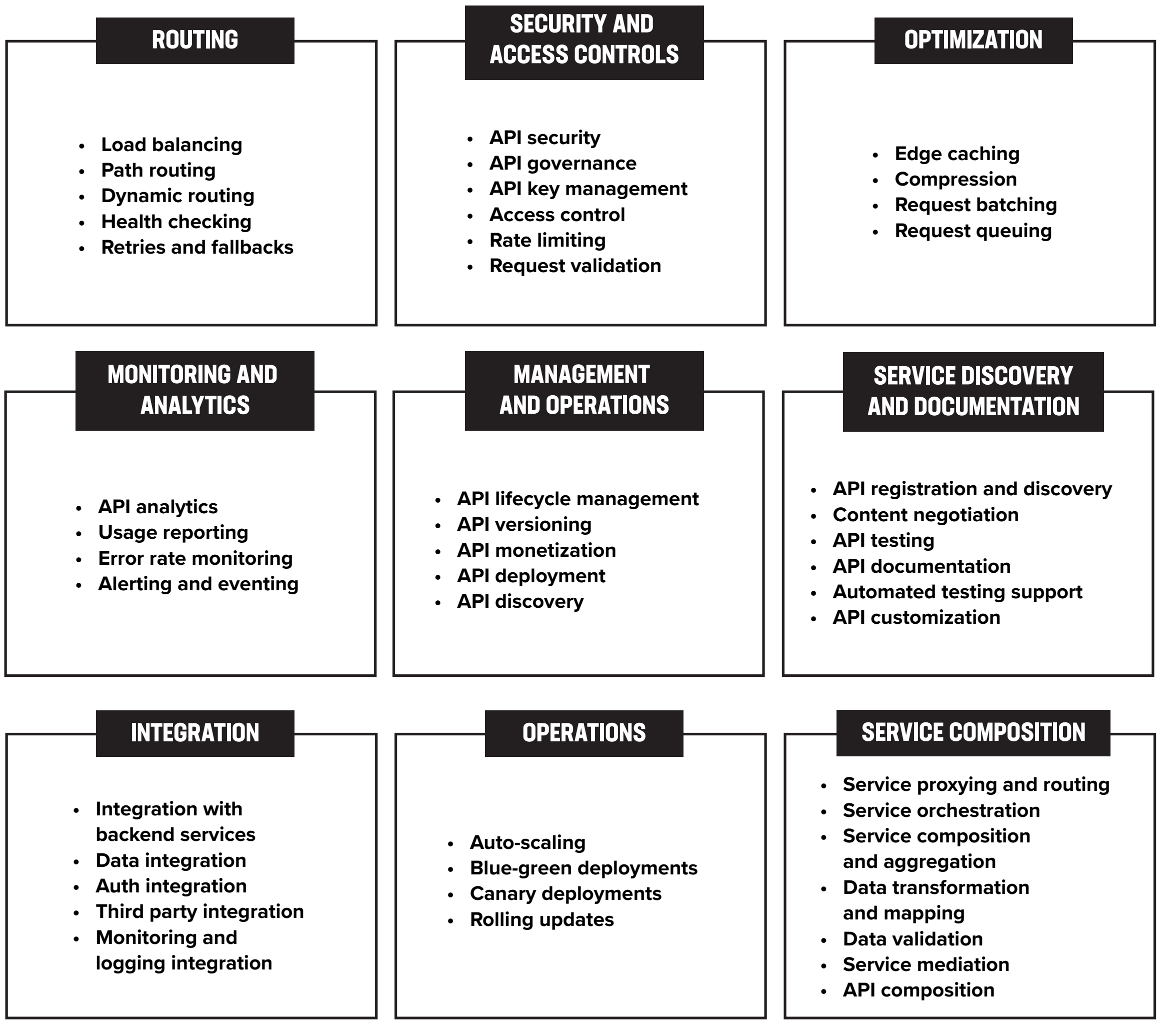


Figure 6: API management and infrastructure features

API Gateway Common Features

Let us examine the commonly recognized and familiar features of API gateways:

1. **Routing:** Routes requests to the appropriate backend service based on the path or content of the request.

2. **Authentication and authorization:** Authenticates and authorizes requests as a single ingress point, ensuring that only authorized clients can access the backend services.

3. **Rate limiting:** Limits the rate at which clients can make requests to the underlying APIs, preventing overuse and protecting the backend services from overload.

4. **Caching:** Caches responses from the underlying APIs, reducing the number of requests needed to be made to the backend services and improving performance.

5. **Protocol translation:** Translates between different protocols, such as HTTP and WebSockets, allowing clients to access the underlying APIs using different protocols.

6. **Load balancing:** Distributes requests to multiple backend services, improving scalability and reliability.

7. **Security:** Handles security tasks, such as encryption and decryption, to ensure data is transmitted securely.

8. **Analytics and monitoring:** Tracks and reports usage metrics and error information, providing visibility into how the system is being used and helping to identify performance bottlenecks and errors.

9. **Versioning:** Handles versioning of the underlying APIs, allowing clients to access different versions of the API depending on their needs.

10. **Service discovery:** Discovers available backend services and dynamically route requests to them, allowing for more dynamic and flexible service discovery.

Context: API Gateways in Focus

After analyzing API management infrastructure features, we identified the need to better understand the role of API gateways and explore alternatives to the current monolithic API gateway design.

With the growth in the number of APIs already leading to API sprawl and API gateway sprawl, an increasing number of clients are becoming mobile or highly distributed, and compute infrastructure has become available everywhere, including at the edge. We thus need an approach which can improve the agility, flexibility, scalability, performance, and security of the API ecosystem.

This new approach requires a streamlined design capable of fully leveraging the benefits of a truly distributed architecture.

API GATEWAYS

We further analyze the functionality and scope of an API gateway to tease out its nuances and limitations.

An API gateway is a critical component of today's API management infrastructure. At its core, the API gateway is a reverse proxy; it acts as a middleman between clients and backend services while performing a variety of tasks on the incoming request. For example, the gateway can authenticate, rate limit, request route, apply security policies, monitor, and apply versioning before forwarding the request to an appropriate backend service. Once the backend service has processed the request and returned a response, the API gateway can perform tasks such as caching, protocol translation, and response handling before forwarding the response back to the client.

As the number of APIs have grown, API gateways have evolved to include a variety of new functionalities beyond basic routing, effectively becoming monolithic systems. These gateways now manage tasks like authentication and rate limiting to improve performance and reduce the burden on backend services. However, even with this enhanced functionality, API gateways remain a single access point to the backend service, which can present challenges in highly distributed environments.

In particular, the rise of cloud, hybrid multi-cloud, and edge infrastructures has made it more difficult to maintain agility, security, and manageability with an API gateway approach. With clients, devices, and application workloads potentially spread out across a wide range of locations, an API gateway may not be well suited to provide the necessary level of edge-friendly architecture.

API GATEWAY CHALLENGES

Since they handle a wide range of tasks and need to integrate with multiple systems, API gateways are typically hard to deploy and manage. While managing API gateways can be complex, it is nevertheless a critical task for ensuring the availability, security, and scalability of an API. Enterprises tend to use specialized tools, such as API management platforms, to help manage and monitor their API gateways more effectively.

WHILE MANAGING API GATEWAYS CAN BE COMPLEX, IT IS NEVERTHELESS A CRITICAL TASK FOR ENSURING THE AVAILABILITY, SECURITY, AND SCALABILITY OF AN API.

The list below is not comprehensive, but some of the elements contributing to API gateway complexity include:

1. **Configuration management**: API gateways often have a wide range of configuration options and settings that need to be managed and maintained, such as routing rules, rate limiting, and security settings. Managing these settings can be complex and time-consuming, especially as the number of underlying APIs and clients grows.

2. **Integration with other systems:** The gateways need to integrate with a wide range of other systems, such as authentication and authorization systems, load balancers, and databases. This can be complex, especially when the underlying systems are not well integrated, or when the API gateway needs to handle multiple protocols or data formats. This becomes more problematic when an enterprise has multiple API gateway deployments from multiple vendors.

3. **Scalability and availability:** API gateways need to be able to handle large numbers of requests and ensure high availability, which can be complex to manage, especially when dealing with large numbers of backend services and clients.

4. **Security:** Being a critical API security component, security API gateways must be configured and managed to ensure sensitive data is protected and access is controlled. This can be complex and requires ongoing monitoring and management.

5. **Versioning:** As the number of underlying APIs and clients grow, it can become increasingly difficult to manage different versions of the API and ensure clients are accessing the correct version.

6. **Monitoring and troubleshooting:** API gateways can collect and generate large amounts of data. In a large enterprise the gateways can be distributed across many locations, making it hard to correlate events affecting the overall health of the application and troubleshoot issues.

API GATEWAY SPRAWL REFERS TO THE PROLIFERATION OF MULTIPLE, INDEPENDENT API GATEWAYS WITHIN AN ORGANIZATION.

API GATEWAY SPRAWL

API gateway sprawl refers to the proliferation of multiple, independent API gateways within an organization. Different teams or business units often create their own APIs, which can lead to the creation of multiple, independent API gateways to handle these different APIs. Different teams may also use API gateways from different vendors or solutions to handle different types of APIs.

This leads to the deployment of multiple API gateways, all with varying sets of capabilities.

API gateway sprawl creates several additional challenges:

1. **Scaling API gateway management:** Having multiple, independent API gateways can make it difficult to manage and maintain the gateways, especially as the number of underlying APIs and clients grows.

2. **Inefficiencies in east-west traffic:** Multiple gateways can result in requests needing to pass through said multiple gateways, adding latency and reducing performance.

3. **Uniformity of security policies:** Managing multiple gateways can be difficult and may lead to inconsistent security policies, making it harder to ensure sensitive data is protected and that access is controlled.

4. **Standardized governance:** With multiple gateways, it can be difficult to ensure all APIs are properly governed and conform to standards and best practices.

5. **Increased cost:** Having multiple gateways may lead to higher costs for infrastructure, development resources, and maintenance.

6. **Amplified integration challenges:** Having multiple gateways makes it harder to integrate with other systems and technologies, such as other databases, data warehouses, and data analytics tools.

Enterprises should strive to centralize their API management and governance and use a single type of API gateway. While this will alleviate the above challenges of API gateway sprawl, a homogenized strategy for API gateways is anything but simple.

CHALLENGES IN STANDARDIZING API GATEWAYS IN AN ENTERPRISE

As enterprises grow organically or through acquisitions, internal teams aligned to specific business units (BUs) will be at odds with each other while selecting an API gateway technology or vendor. Some reasons for this include the following:

- **Technology:** Different teams or business units have different technology stacks or prefer different API gateway solutions, making it difficult to standardize on a single type of gateway.

- **Legacy systems:** Some teams have existing systems that were built using a different type of API gateway, and it would be difficult to replace these systems with a new gateway, especially if they are still in use.

- **Customization:** Some teams will customize their existing API gateways to meet specific requirements and will find it difficult to replicate these customizations on a new gateway.

- **Replacement cost:** Replacing existing API gateways with a new, standardized gateway can be costly, both in terms of development and maintenance.

- **Training developers:** Teams vary in their levels of expertise and may need to become familiar with, or undergo training on, a new gateway technology from a different vendor—a process that can be both time-consuming and expensive.

- **Limited resources:** Organizations have limited resources in terms of developers, budget, and time to make the change, making it difficult to implement a new gateway.

- **Application dependencies**: Different teams or business units have different dependencies on their existing gateways, like being integrated with specific systems, or other custom integrations, making it difficult to switch to a new one.
- **Third-party solutions:** Teams using third-party solutions that integrate with the gateway will find it difficult to migrate to a new solution that doesn't support these third-party solutions

Thus, if an existing application has a well-established and opinionated team, the team will not want to pivot to a different deployment pattern so as not to cause disruption in their service.

We can thus conclude there is a need for a new approach which takes into consideration multiple limitations of existing API infrastructure while highlighting API gateway sprawl as one of the more important considerations.

Design Considerations

The following is not an exhaustive list but summarizes some of the high-level design considerations we believe should be incorporated when building the solution:

1. **Coexist with current deployments:** As organizations strive to keep up with the ever-changing technological landscape, it is common for enterprises to have a diverse range of API gateway deployments. It is not feasible to forklift the existing API infrastructure, as this can disrupt critical business operations. Thus, any new deployment must be designed in a way that can coexist with the currently deployed architecture.
2. **Standardize API gateway functions:** The primary goal of an enterprise's API strategy should be to standardize their API gateway functionality, which can be a challenging task due to the diverse range of APIs and the varying needs of different business units. Nevertheless, this standardization is crucial for creating a stable and secure infrastructure that can support the organization's digital transformation.
3. **Leverage edge deployments:** The edge infrastructure not only has the potential to exponentially increase the number of APIs, but also provides an opportunity for enterprises to move their gateway actors closer to the edge. This is possible as the same infrastructure used to create APIs can also be utilized to create distributed gateway actors. Therefore, the solution must fully leverage the proximity to the edge infrastructure to the clients that initiate the API request.
4. **Transport agnostic:** An important consideration for the implementation of distributed gateway actors' architecture is that it should not be dependent on any

specific transport mechanism. Whether an enterprise wants to utilize traditional IP networks, overlay networks, VPNs, or low latency messaging infrastructure, the solution must be agnostic to the transport mechanism. This allows for greater flexibility and enables enterprises to choose the transport mechanism that best suits their specific needs and requirements.

5. **GraphQL support:** GraphQL is becoming an increasingly popular choice for API development due to its many advantages over traditional REST APIs. One key advantage is its ability to provide fine-grained access to data, allowing clients to specify exactly what data they need in a single request. Additionally, GraphQL can simplify the process of aggregating data from multiple services, making it the right architecture to do service composability and orchestration. This can reduce network overhead and improve performance, especially in a distributed system where multiple API services are used to fulfill a single request. Finally, with its strongly typed schema and query language, GraphQL can improve API discoverability and enable easier client development.

6. **Security is table stakes:** The paramount design goal is it should be additive to the API security posture of the enterprise. The solution could subsume some of the functionality, like the ability to authenticate and authorize API requests, implement access controls, and protect against common security threats such as cross-site scripting (XSS) and SQL injection attacks. Under no circumstance should the new solution compromise the existing threat model or change it so significantly that the attack surface changes. By prioritizing security as a design goal, the architecture must provide a more secure environment for API communication, reducing the risk of data breaches and other security incidents.

Specific requirements can be derived based on these design considerations, which we've incorporated into our distributed gateway actors (DGA) solution.

Distributed Gateway Actors

THE DISTRIBUTED GATEWAY ACTORS (DGA) DESIGN PATTERN TAKES INSPIRATION FROM EDGE COMPUTING AND COMPUTING AVAILABLE CLOSE TO A CLIENT.

Now having fully explored API gateways, we can explain the distributed gateway actor solution.

DISTRIBUTED GATEWAY ACTORS DESIGN

The distributed gateway actors (DGA) design pattern takes inspiration from edge computing and computing available close to a client. The crux of the idea lies in hyper-distributing each gateway actor as close as possible to the client and having the gateway actor exist only for the duration of the transaction before it's cleared at the end.

ASSUMPTIONS

Here are some of the underlying assumptions to this solution.

Edge compute has become more pervasive, and we can now find some type of compute that is geographically available closer to the client. The gateway actors can be instantiated on these edge compute nodes. The intent is to have an implementation where the DGA is very lightweight and ephemeral so it can be instantiated on any edge compute.

API transport is a crucial component of the infrastructure since it involves establishing a network between the gateway actors and the API gateway. The type of infrastructure required is dependent upon the vendor or enterprise and can vary. For example, a large public cloud may use their own backbone to run the API transport. Another implementation could be a low-latency messaging bus layered on top of an existing high-bandwidth and low-latency network within an enterprise data center.

API GATEWAY FUNCTIONS

To reiterate, the API gateway is essentially a reverse proxy whose principal function is to route the HTTP traffic to the appropriate API workloads. This approach makes sense where all the APIs are collocated, like within a local on-prem network or within a virtual private cloud (VPC) inside an availability zone. But as the number of APIs scale, move across a hybrid infrastructure, or just become mobile, this approach of API gateway design is rendered inefficient, complex to operate, and expensive.

While there might be different views on how to classify all the features descried in Figure 6, we can agree that *API management infrastructure* has become a complex deployment of many components that need to be carefully orchestrated.

Figure 7 maps the various features and functions of the API management from Figure 6 to the distributed gateway actors architecture. This mapping visually illustrates how each feature and function is aligned with the DGA approach, highlighting the seamless integration of the API management components within the architecture.

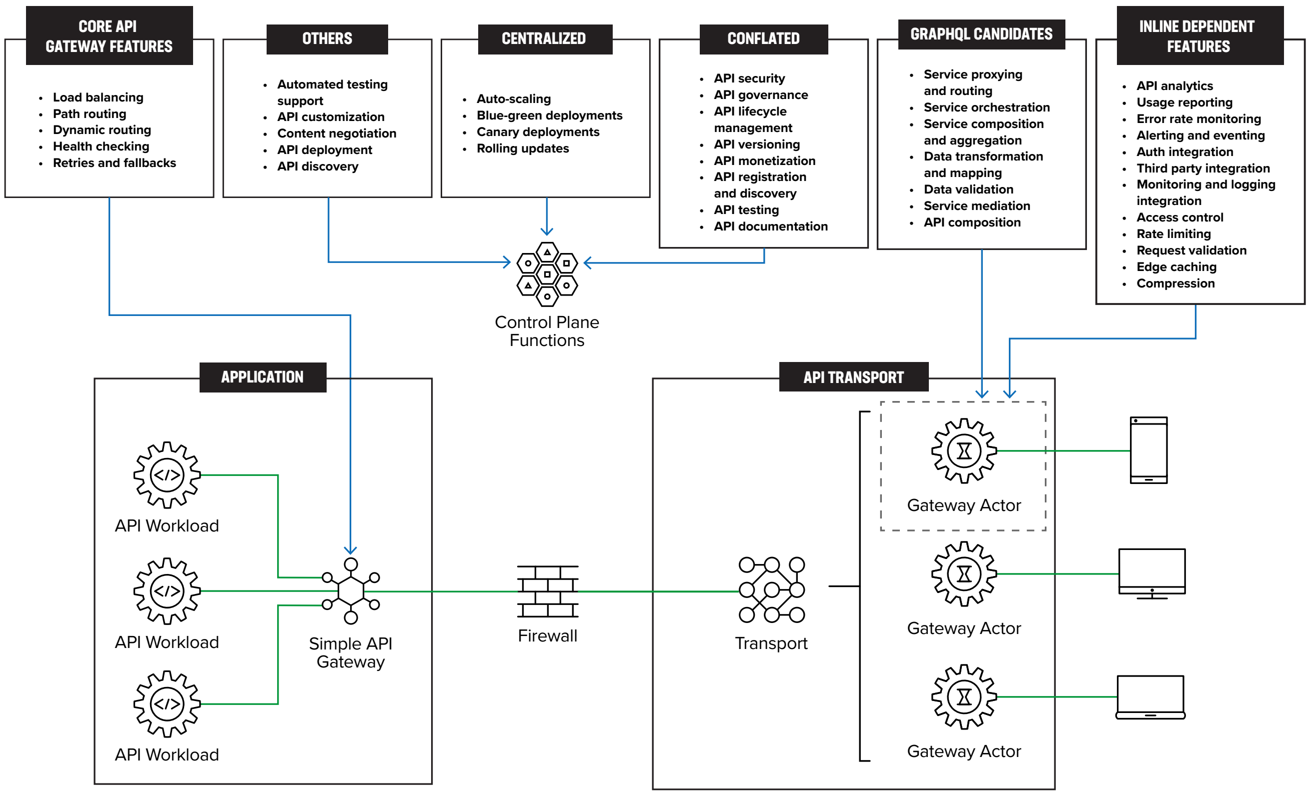


Figure 7: GraphQL-based distributed gateway actors

Centralized Functions

Most of the features listed above have some management component which can be logically centralized. Our goal is not to rearchitect the management plane but, if possible, remove certain functions.

Core API Gateway Functions

These are data plane functions and thus best implemented in the API and collocated with application workloads. API gateways are a crucial component of modern microservices architecture that serves as the entry point for all external traffic. We categorized its core functions to include routing, load balancing, dynamic routing, health checking, retries, and fallbacks.

An API gateway directs incoming requests to the appropriate microservice, distributes traffic across multiple instances, supports dynamic routing, monitors the health of microservices and their instances, retries failed requests, and provides fallback responses when a microservice is unavailable or has failed. Other functions such as authentication, authorization, rate limiting, caching, and logging can be distributed to the edge or centralized functions depending on the specific requirements of the system.

This modular approach allows for a more flexible and optimized architecture and is at the heart of our recommendation for simplifying, modernizing, and scaling the enterprise API infrastructure.

Conflated

API gateway and API management are often mistakenly conflated by vendors as part of the API gateway function, but they are actually two distinct functions that should be treated separately. An API gateway is responsible for routing requests from clients to backend services, while API management encompasses a broader set of features such as access control, rate limiting, analytics, and developer portal management.

WHILE SOME VENDORS MAY OFFER BOTH API GATEWAY AND API MANAGEMENT FUNCTIONS AS PART OF A SINGLE PRODUCT, IT'S IMPORTANT TO UNDERSTAND THE DIFFERENCES BETWEEN THESE FUNCTIONS AND EVALUATE THEM SEPARATELY BASED ON THEIR SPECIFIC REQUIREMENTS.

While some vendors may offer both API gateway and API management functions as part of a single product, it's important to understand the differences between these functions and evaluate them separately based on their specific requirements. Combining these functions can lead to confusion and potentially limit the flexibility and scalability of an organization's API infrastructure. This is also critical in understanding our position on distributing the API gateway functionality.

Gateway Actor - Inline Features

The features listed here are core functions which need to be inline to the data path. In a distributed gateway pattern, some of the inline functions of the API gateway also become distributed. These functions include access control, rate limiting, request validation, API analytics, usage reporting, error rate monitoring, alerting and eventing, auth integration, third-party integration, monitoring and logging integration, edge caching, and compression.

Moving these functions to the distributed gateway pattern has the following benefits:

- **Reduced load on API gateway:** Help to reduce the load on the centralized API gateway and improve the performance and scalability of the system.
- **Enable faster response times:** Enable faster response times and reduced latency by deploying these functions closer to the clients. By leveraging caching of the data and the function, the edge hosted API gateways can rapidly respond to incoming requests.

For example, access control, rate limiting, and request validation can be handled by an edge gateway actor, which is deployed closer to the clients. This can help to reduce the number of requests needing to be handled by the centralized API gateway, improving its performance and scalability. Similarly, API analytics, usage reporting, error rate monitoring, alerting and eventing, and monitoring and logging integration can be handled by edge gateways, which can collect and aggregate data from multiple microservices.

Gateway Actors - GraphQL Candidates

Today, an important capability that API gateways support is service composition and orchestration. While this can get rather complex, it would become a concern if this feature were not supported by the simplified API gateway. We believe GraphQL can be an interesting approach to service composition, acting as a sort of middleware to existing APIs.

Due to its visibility of all the APIs, the API gateway becomes a logical place to perform service composition, enabling developers to combine the APIs behind the gateway to enhance the business logic without needing to write new services which can be more easily combined in a service composition layer.

Service composition in GraphQL is made possible through its use of a strongly typed schema, which defines the shape of the data available to clients. Clients can use this schema to construct queries that specify the exact data they need, regardless of which services or sources provide it. Resolvers, which are functions that map queries to data sources, are used to retrieve data from the appropriate service or source. However, while GraphQL promises better security, it is only as good as the developer who writes the code.

Others

There are still some remaining features not highlighted from *Figure 6: API management and infrastructure features*. These remaining features and functions are candidates that can be moved to individual management and operations or data-plane functions.

We deliberately choose to use the term “actor” to avoid suggesting a specific implementation or vendor technology. The gateway actor’s implementation could be based on methods, functions, workers, threads, and processes, implemented using infrastructures based on VMs, containers, serverless, or other approaches specific to a vendor.

COMPONENT’S FEATURES AND BEHAVIOR

The approach taken with the distributed gateway actors (DGA) architecture simplifies the API gateway functions and moves other inline features either to the edge, or to the control plane.

Control Plane

Apart from the API gateway features, the DGA architecture also recommends identifying functions that could be better served in the control plane as a logically centralized component rather than implemented within a monolithic API gateway. The management and control of API infrastructure that already exists can be extended and expanded to include this additional functionality.

AN ENTERPRISE WANTING TO MAKE THIS TRANSFORMATION COULD ROLL OUT THE DGA ARCHITECTURE ONE APPLICATION AT A TIME, WITHOUT DISTURBING THE EXISTING DEPLOYMENTS AND WITHOUT THE NEED FOR A FORKLIFT OPERATION.

Simplified API Gateway

The simplification of the API gateway thus becomes an exercise to derive a standard set of core functions that all API gateway vendors can manage through a common set of configuration parameters.

An enterprise wanting to make this transformation could roll out the DGA architecture one application at a time, without disturbing the existing deployments and without the need for a forklift operation.

Distributed Gateway Actors

An important aspect of the DGA is that each *gateway actor* is ephemeral, being instantiated only for the duration of an API session (i.e., one client making one API call).

A distributed gateway actor can be more flexible, scalable, and cost-effective than the traditional API gateway. Rather than relying on multiple monolithic API gateways from different vendors to aggregate and handle API traffic, the gateway actor allows developers to specify and deploy individual gateway instances for each API closer to the edge of the network. The APIs themselves could provide greater control and customization for their specific needs.

This new approach also allows for greater scalability, as developers can easily spin up the gateway actor instances as needed to handle increased traffic without worrying about the overhead of managing a large, centralized gateway.

In addition to its technical benefits, the gateway actor also offers cost savings over the traditional API gateway, by allowing enterprises to only pay for the ephemeral gateway actor instances they use. This deployment model can open opportunities for additional revenue models.

GATEWAY ACTOR PLACEMENT

By leveraging an API transport layer, the DGAs essentially decouple the API ingress location from the API gateway. The DGAs can be moved to the edge, i.e., closer to the client making the API call. The APIs can terminate in the DGAs and then be transported by any means. This is different from *Figure 3: Legacy API gateway architectures* where the client traffic ingresses topologically adjacent to the API gateways.

Conclusion

Our intent thus has been to propose a vendor and deployment agnostic approach, as different vendors may have their own intellectual property to build these components to achieve similar objectives as outlined.

In this paper we have summarized our learnings from multiple quarters researching API sprawl, Edge 2.0 architectures, the API economy, and investigations into GraphQL. While the jury is still out with respect to legacy API infrastructure, we believe there is a need for a new approach.

Just the promise of unlocking value within every individual device or entity globally provides a strong enough reason to explore a new approach. We need to move away from the legacy API infrastructure today, because even though it looks distributed, it is monolithic under the hood.

We propose the distributed GraphQL based gateway actor approach as a vendor agnostic way to unlock the full potential of the emerging API economy.