Reference Architecture for a Cloud Native Digital Enterprise

Version Winter-2020

Original Author

- Lakmal Warusawithana | Senior Director - Technology Evangelism | WSO2, Inc | lakmal@wso2.com | @lakwarus

> *This document describes a vendor/technology-neutral reference architecture for a cloud native digital enterprise. The architecture defined in this paper can be mapped into different cloud native platforms (Kubernetes and service mesh), different cloud providers (Microsoft Azure, Amazon AWS, and Google GCP), and infrastructure services to perform the implementation. These reference implementations will be covered in separate papers.*

Introduction

In an era of digital transformation, **(digital) enterprises** are looking for fast innovation through effective collaboration to deliver more value to their customers with dramatically less effort. Digital enterprises enable companies of every sector to integrate, expose, and monetize their business capabilities by digitizing entire value chains.

As a result, APIs have become the norm to expose integrated business functionalities to deliver enhanced digital experience. Enterprises can start their digital transformation in greenfield or brownfield; in both cases, having a well defined **API-led integration architecture** is important. Apart from integration and API platforms, these architectures should be able to provide agility, flexibility, and scalability. This paper focuses on how to use cloud native technologies along with an API-led integration platform to create an effective architecture, i.e., a **reference architecture for a cloud native digital enterprise**, to increase productivity by having agility, flexibility, and scalability through automation and services.

What is Cloud Native?

"Cloud native" is combination of a philosophical approach and a set of technologies that allow organizations to build, deploy, and operate software applications more frequently, resiliently, and reliably. The culture is an important one that is mostly ignored when describing the cloud native. In addition to the architectural and technical requirements, it should include cultural changes like more agile processes, dynamic and decentralized decision-making, and autonomy teams that will allow the organizations to utilize the benefits of all those modern technologies. In this document, we mainly focused on architectures and technologies that important for cloud native digital enterprise, and we will discuss essential cultural changes in a separate methodology paper.

Cloud native has its own foundation: the **Cloud Native Computing Foundation (CNCF)**, which was launched in 2015 by the Linux Foundation. The main goal of the CNCF is to build sustainable ecosystems and foster communities to support the growth and health of cloud native open-source software.

Cloud Native Reference Architecture

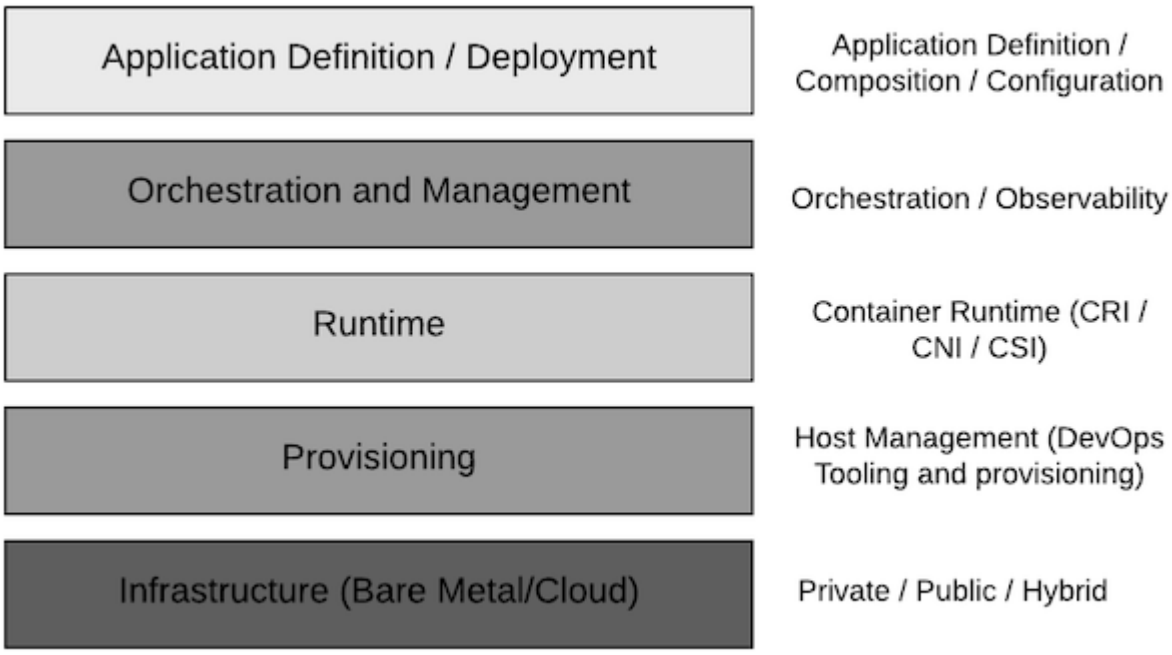


Figure 1 - Cloud native reference architecture by the CNCF

Figure 1 illustrates the cloud native reference architecture presented by the CNCF. Each layer has its own specialized cloud native software stacks and many of them are governed by the CNCF.

Infrastructure

The infrastructure layer represents the actual computing resources. These computing resources can be composed by using a set of bare metal machines networked together in a local data center. If the digital enterprise already runs a private cloud with the support of hypervisor-based virtualization technologies like VMware, OpenStack, and CloudStack, then the infrastructure layer can be composed by using a set of virtual machines connected and worked in the same virtual network. Alternatively, an enterprise can use virtual infrastructures, which is provided by public cloud providers such as Google, Microsoft, and Amazon. Or it can be a hybrid cloud by combining private cloud and public cloud computing resources.

Provisioning

The provisioning layer covers the host management activities such as installation and setting up

operating systems. It has a set of DevOps (maintenance) and management (software updates, security patches, etc) activities. Operating systems like CoreOS and RancherOS are specialized host operating systems to run containerized environments.

Runtime

The runtime layer mainly consists of the container runtime. The **container runtime interface** (CRI) allows to plug different implementations of container times. Docker is the widely used container runtime; alternatively, CRI-O (Open Container Initiative compatible runtimes) or rkt container runtimes can be used. You can even plug a hypervisor-based container runtime like Frakti with support from CRI.

The **container network interface** (CNI) enables APIs to plug different container network runtime implementations. The CNI comes with inbuilt network plugins such as BRIDGE, VLAN, IPVLAN, DHCP, loopback, etc. Also, it allows the plugin of the container network from third-party originations such as Weave, Calico, Cilium, Flannel, WMWare, and NSX. All of the network runtimes implement CNI specifications.

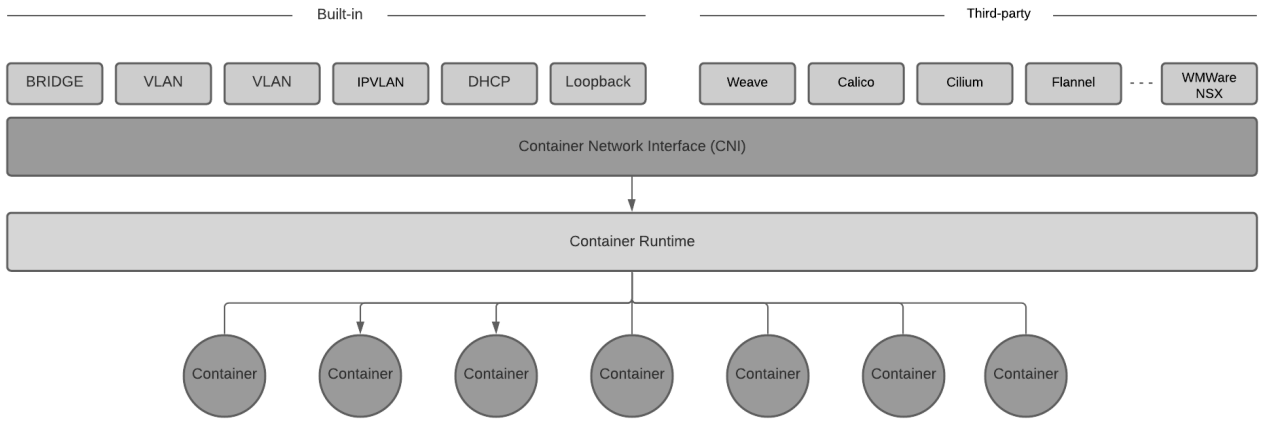


Figure 2 - Container network interface (CNI)

Docker does not implement the CNI and it has its own implementation known as the container network model (CNM) and it only works with the Docker container runtime.

A **Container storage interface** (CSI) provides a common standard to connect container orchestration platforms to plugin to persistent storage. With the help of CSI, storage vendors can write a plugin to a single specification and this works on many orchestration platforms. Dynamic provisioning and decommissioning of volumes, attachment and detachment of volumes from a host node, and mounting and unmounting of a volume from a host node are the main capabilities that are provided by the CSI.

Orchestration and Management

Container orchestrators help to manage a large number of containerized application deployments across multiple container host machines. Cloud Foundry, Mesos, Nomad, and Kubernetes are

popular container orchestrators used in the cloud native space. Container scheduling, provisioning, launching, and discovery; system monitoring, tracing, and crash recovery; declarative system configuration; routing, load balancing, and policy enforcements are a few common features that are managed by these orchestrators.

These base orchestrators can be used to create higher-level orchestrators such as service mesh and serverless platforms. Istio, Linkerd, and OpenPaaS are some service mesh platforms created on top of the Kubernetes orchestrator.

Application Definition / Deployment

The application definition layer defines application composition, application-specific configurations, deployment properties, image repositories, continuous integration / continuous delivery, etc. Cloud native application developers are mainly engaged with the functionality of this layer.

Reference Architecture for a Cloud Native Digital Enterprise

Cloud native applications are all about dynamism. **Microservice architecture (MSA)** is critical to accomplish agility. To gain the benefits of MSA (e.g., faster to develop, test, and deploy, and much easier to understand and maintain) these microservices need to integrate with different SaaS endpoints, legacy applications, and other microservices to perform the defined business functionality. To align with business requirements, it is required to composite and integrate multiple microservices and expose them as business APIs. These APIs should be secured, managed, observed, and monetized. API-led integration platforms combined with cloud native technologies are critical for a digital enterprise.

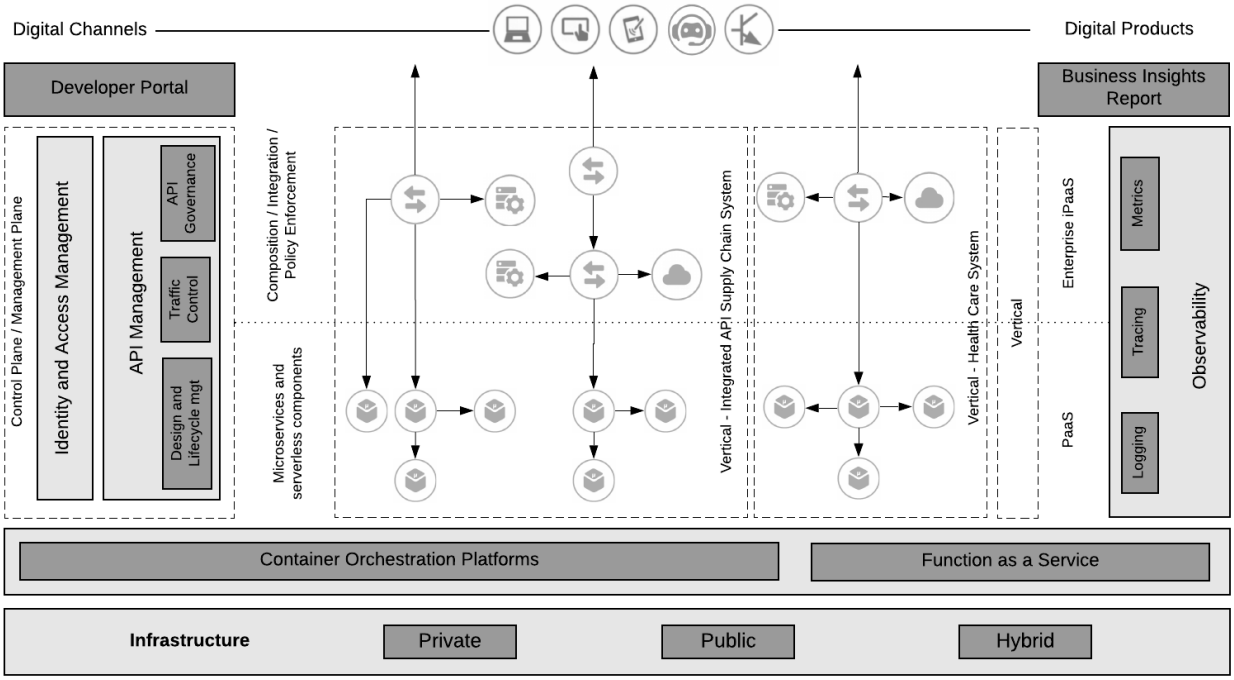


Figure 3 - Reference Architecture for a Cloud Native Digital Enterprise

Infrastructure

This layer represents the same functionality that we discussed in the cloud native reference architecture. Container Orchestration Platform / Function as a Service Platform

This layer also represents the same functionality that we discussed in the cloud native reference architecture. Cloud Foundry, Mesos, Nomad, Kubernetes, Istio, Linkerd, and OpenPaaS are some examples of current industry-leading container orchestration platforms. Knative, AWS Lambda, Azure Functions, Google Functions, Oracle Functions are a few examples of functions as a service platform (FaaS).

Microservices and Serverless Components

Decompositioning a complex problem into a set of smaller problems will be easier to tackle and faster to develop, test, deploy, scale, and much easier to update. This smaller problem can be implemented as microservices or a serverless function. Each microservice or serverless function is developed by a smaller team with the freedom of choosing appropriate technologies. Digital enterprises can have in-house or cloud orchestration platforms to deploy these MSA-based applications. Some cloud providers offer PaaS on top of these orchestration platforms and enterprises can use them with a pay-as-you-go model. If enterprises use serverless functions, then it is recommended to use a FaaS platform provided by a well-known cloud provider. Cloud locking might be a downside for the use of a FaaS platform, but it can depend on your enterprise's policies

Containers Images

"The Twelve Factor App" defines a methodology to develop and deploy scalable applications and many of them are nicely fit into MSA.

Once the microservices are defined and implemented, they should be bundled with all their dependencies and shipped as container images. Environment-specific configurations should be defined externally and injected into containers at the runtime. These container images should be stored in a registry where other developers as well as runtime environments cloud-pull and create containers out of these images.

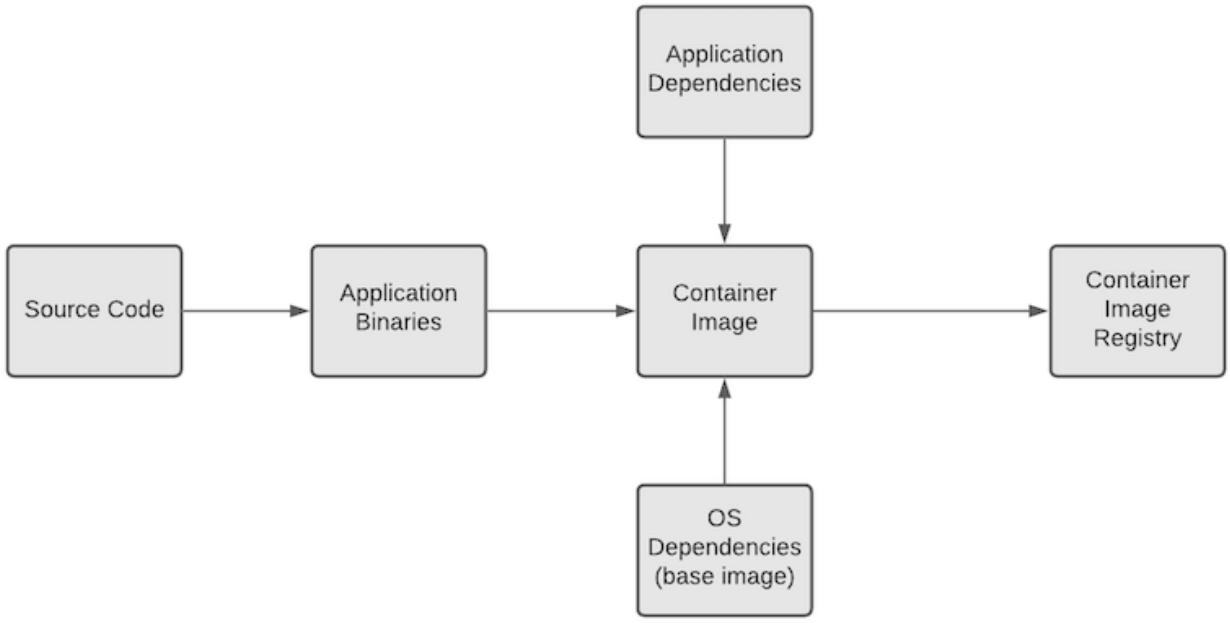


Figure 4 - Container image creation

One of key benefits of shipping applications as container images is the universal packaging model, which is supported by all the cloud providers, and the property of immutability. Once container images have been built, then it is guaranteed all the required dependencies will be met when the container runtime is created. It helps to maintain consistency from developer machines to production servers. Every release should create a new container image with proper versioning. These versions can be used as a container image tag.

Container Image Naming

Container Runtime

The container orchestration platform is scheduled and creates a container (runtime) in a worker node. Each container gets its own IP address, storage, and a namespace with the allocated CPU and memory resources. These resource allocations should be able to pass as runtime properties or be allocated with default values.

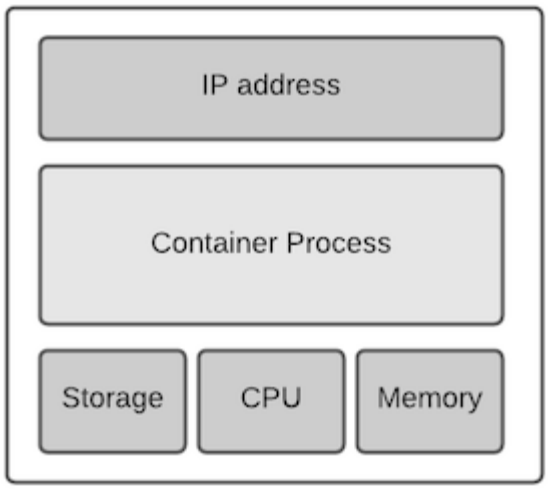


Figure 5 - Container (runtime)

In addition to the all application dependencies that come with the container image, the container runtime needs to be associated with some environment-specific properties such as configurations, certificates, and credentials. These properties can be changed in the developer, test, and production environments. Therefore, these properties should not burn into the container image but should be associated with the container runtime.

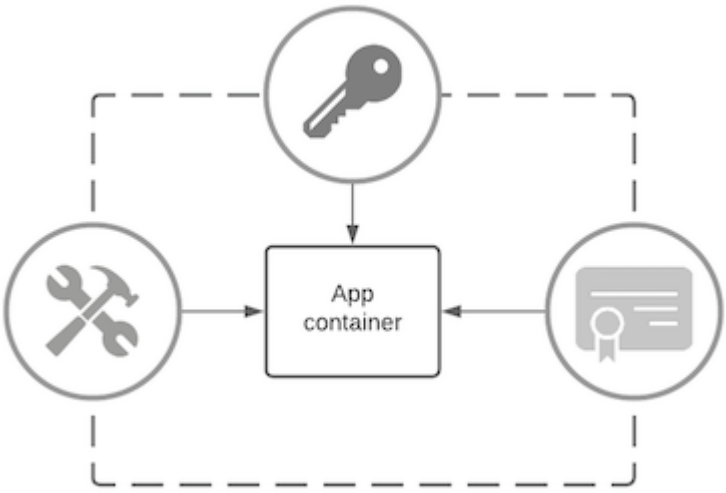


Figure 6 - Configs, credential, and certificate association with the container

Scalability, Load Balancing and Service Discovery

Microservices communicate with each other to complete a given business task. When the number of services increase, we need to have a proper discovery service and should be able to communicate with a unique name (service-name) such as a domain name service (DNS). These service names should not be bound to a specific environment (dev, pod, etc.) and they should be resolved to correct the IP address of the services that are running in the given environment.

Compared to hypervisor-based virtual machine instances, a container runtime's overhead is minimal. Owing to the combination of container properties and MSA best practises, these containers can be scaled out very fast. When scaling out, ingress traffic should be routed to each container with a proper load balancing mechanism. Every application should have a proper load balancer bound to a service name.

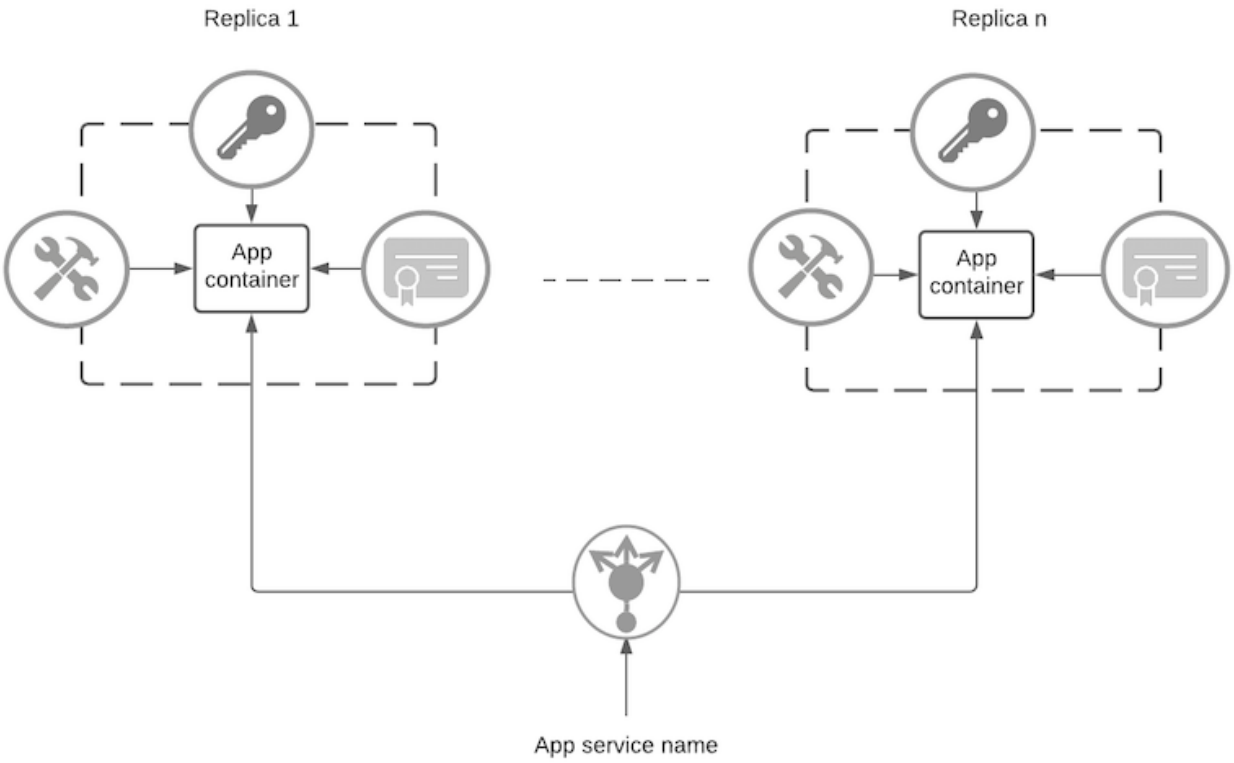


Figure 7 - Scaling, load balancing, and service name resolving

Health Check and Auto-Healing

An important feature of orchestration platforms is doing the health check probe for each container and being able to auto-heal if something is wrong. Health checks can be done in multiple levels of the container lifecycle.

Some applications are required to carry out some initialization tasks when the container is booted up. Container orchestration platforms should be able to monitor the progress of these initialization tasks and advertise the successful completion before accepting any workload. These kinds of health check probes are known as startup probes.

After the container boots up, container orchestrators do a health check to confirm the application readiness to accept the workload, then notify the load balancer to route incoming traffic to the container. In some cases, a running application can become unhealthy due to some temporary load spike. In these kinds of scenarios, orchestrators can detect the unhealthiness of the application from the health check probe and notify load balancers to skip further traffic routing. This helps affected

containers to recover. When it recovers, then again opened traffic routes through the load balancers.

Sometimes applications might not fully recover from an unhealthy situation or can be in a fatal error. In such a scenario, orchestrators should be able to identify the situation through the health check probes and replace the error container with a new container. These health check probes are known as liveness probes.

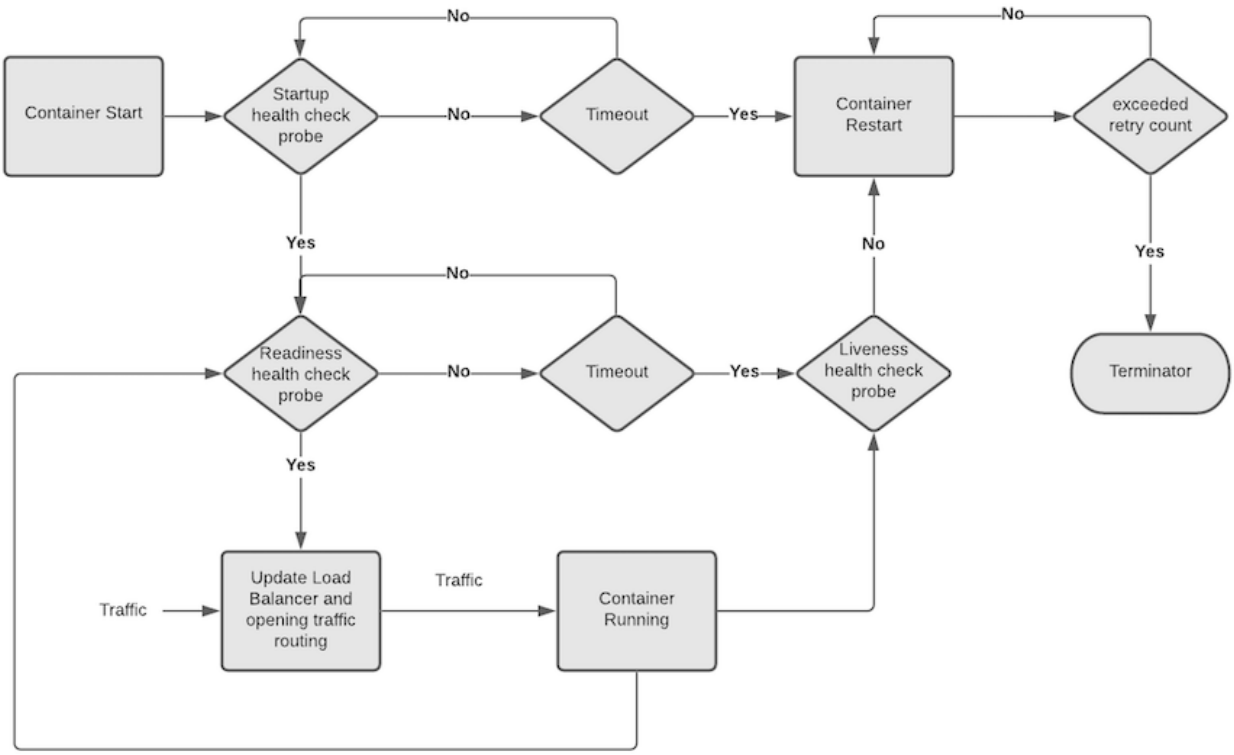


Figure 8 - Container health check probes

Autoscaling

Autoscaling is a critical function to accomplish a scalable architecture. Individual microservices that are deployed as containers should be able to scale in and out depending on the load spikes. Running unnecessary containers wastes computing resources and having a short number of containers can cause a service downtime.

Container orchestrators can monitor these load spikes and are able to remove unnecessary containers or add additional containers to perform scale in and out. CPU usage, memory usage, and in-flight-request counts (load balancer routing queue) are few well known load spike monitoring factors and help to compute scale in and out decisions. Some orchestrators use advanced auto scaling algorithms, where it can predict future load spikes and do scale out early to avoid service disruptions.

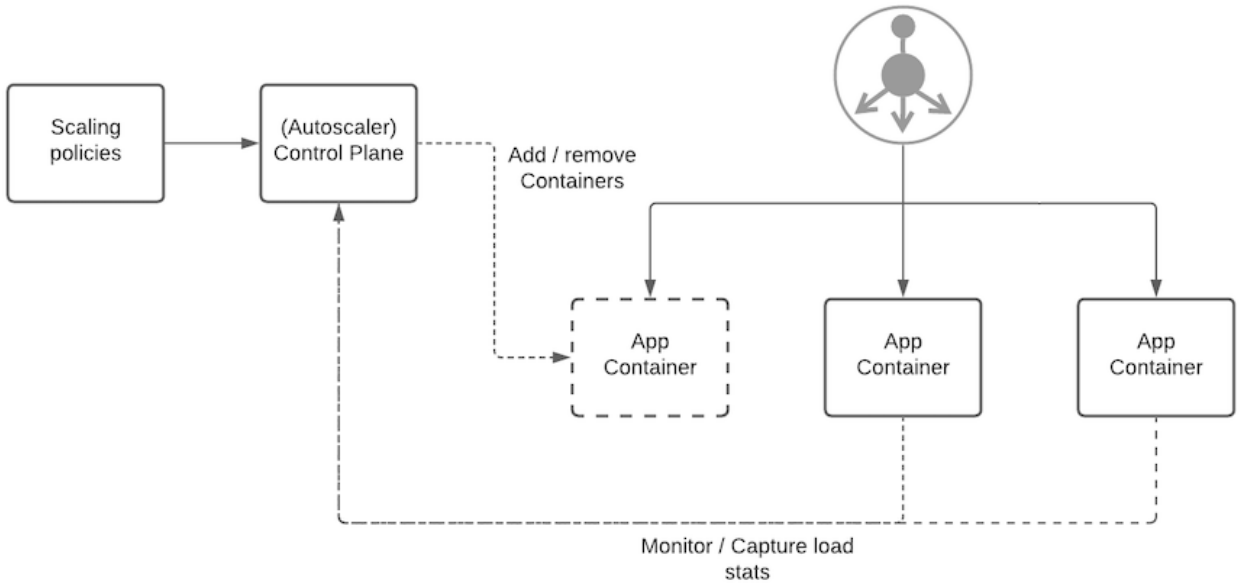


Figure 9 - Autoscaling

Rollouts and Rollbacks

MSA produces frequent releases and these releases need to be seamlessly rolled out into production. As we know, even though we do thorough testing, sometimes we need to roll back to a stable state due to some late-found error. To mitigate these kinds of situations, we should have different deployment strategies. The deployment strategies, ramped, Blue/Green, Canary, A/B testing, shadow help to have zero downtime in rollouts and rollbacks.

Composition / Integration / Policy Enforcement

Microservices are fine-grained and are developed as smaller logical components to make systems more agile and scalable. Microservices are not designed from the end users point of view, where users really need access to the system with their business needs. To expose system functionality as business APIs, these microservices need to integrate with different SaaS endpoints, legacy applications, and other microservices to perform the defined business functionality.

Every enterprise already has some kind of system. When introducing new business functionalities, it is necessary to integrate with these legacy systems. These integrations are often supported by enterprise service bus (ESB) functionality such as routing, transformations, orchestration, aggregation, and resilient patterns.

However, in general, ESB is a monolithic system and does not fit well with MSA. Alternatively in MSA, integrations are achieved by using integration microservices. These integration microservices can have either a codebase implementation approach or a configuration-driven approach. Integration-focused specifical programming languages and frameworks help with the code-based approach by providing necessary abstractions and libraries. For a configuration approach, a modern microservices-friendly lightweight ESB runtime, known as a micro integrator, can be used.

Microservice expose APIs to be consumed by other microservices to complete a given business functionality. These individual microservices need to be aggregated to meet user needs. This aggregation can be done in another microservice to expose meaningful APIs to consumers. These APIs should be secured, managed, observed, and monetized. This requires a governance model with policy enforcement. This is where API gateways are important.

API Gateway

An API gateway can be used as a policy enforcement point of API governance while working in sync with the control and management plane components like lifecycle management, traffic control, policy control and identity and access management. The following are the key functionalities of an API gateway;

Authentication and Security:

While microservices are mainly focusing on business logic, authentication and security can be implemented in the service level as well. But it duplicates functionality and increases maintainability issues. One main benefit of MSA is heterogeneous language support for service implementation. An API Gateway enforces standard authentication and security across all microservices.

API Rate Limiting:

The load handling capacity differs from microservice to microservice. Overloading a few microservices might lead to an unresponsive application and recovering from this kind of a situation is hard. An API gateway only allows requests that can be handled in each microservice and it controls requests that go over the limit.

Dynamic API Discovery and Routing:

Applications go through a series of different lifecycle stages. These life cycles can differ from enterprise to enterprise. Development, test, staging, and production are common lifecycle stages; these are sometimes called environments. In the development stage, developers need to discover other microservices for inter-communication. In addition to discoverability, these inter-communication links should work in different environments without altering anything. Dynamic discovery and dynamic routing should be a key functionality in an API gateway.

API Loadbance and Failover:

To make microservices highly available or scalable, we need to run two or more of the same microservice in a deployment. In such a case, an API gateway can handle the load balancing or failover functionality.

API Shaping:

An API gateway can optimize the API response depending on the nature of the API consumer. For example, if the API consumer is a mobile device, we can strip down some content of the response to optimize bandwidth usage compared to a web consumer.

API Composition:

Especially when we expose APIs to external parties, we might need to aggregate multiple microservice responses and create a single composite API response. An API gateway plays a major role in these kinds of edge compositions.

API Mediation and Transformation:

The same microservice may need to support new microservice consumers as well as legacy consumers. Mediation and message transformation in the API gateway is very useful in such a situation.

Response Caching:

Response caching could be handy in some situations rather than expecting the backend to process each and every request. An API gateway comes in handy in this kind of requirement.

API Gateway Deployment Patterns

Aligning with the MSA, API governance can be achieved via three main API gateway deployment patterns.

- Centralized/shared API gateway
- Private jet API gateway
- Sidecar API gateway

Centralized/Shared Gateway

A centralized API gateway is a well-established and popular deployment pattern. The shared cluster of API gateways handle all API requests. These requests can be internal as well as external API calls. An API gateway cluster can be scaled horizontally and the load is distributed among all the API gateway containers.

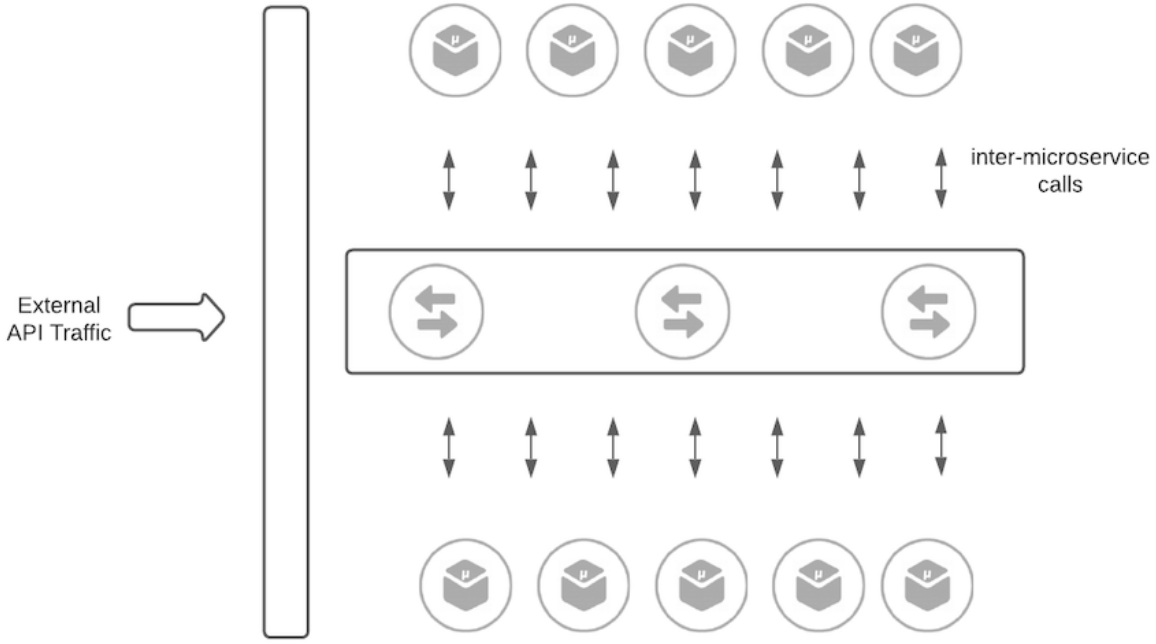


Figure 15 - Centralized/shared API gateway

In this deployment, the API gateway adds an additional hop into inter-microservice communications. The same gateway cluster can be used to manage external APIs as well as internal APIs or can have a dedicated API gateway layer to manage external traffic.

Private Jet Gateway

In this pattern, each individual microservice has a dedicated API Gateway. This provides maximum security as well as guarantees resource allocation for API execution. A single private jet API gateway can be attached to a cluster of microservices of the same type. Load balancing, failover features will be necessary and naturally fit into this kind of scenario.

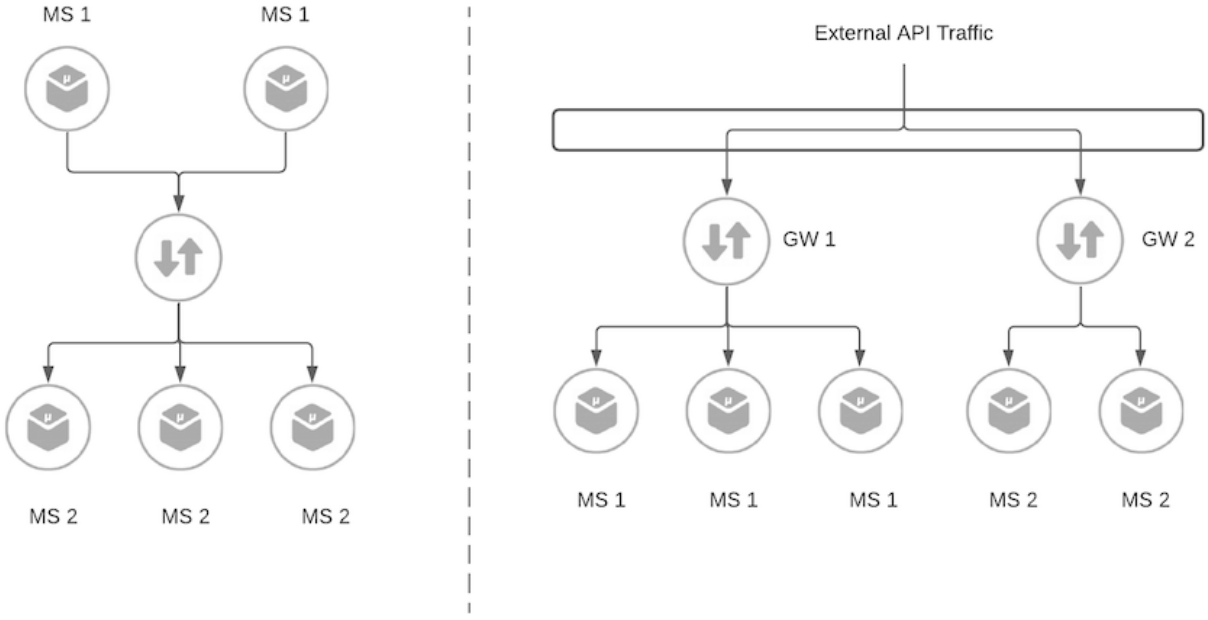


Figure 16 - Private Jet API Gateway

A private jet API gateway itself can be scaled independently. This pattern also increases one network hop in inter-microservice communication similar to a centralized API gateway.

Sidecar Gateway

Hetrogeneos services are one of the key benefits in the MSA. Microservices can be implemented in different languages depending on the benefits. An API gateway can be attached as a sidecar to the microservice, which can then benefit from all the capabilities of the API gateway.

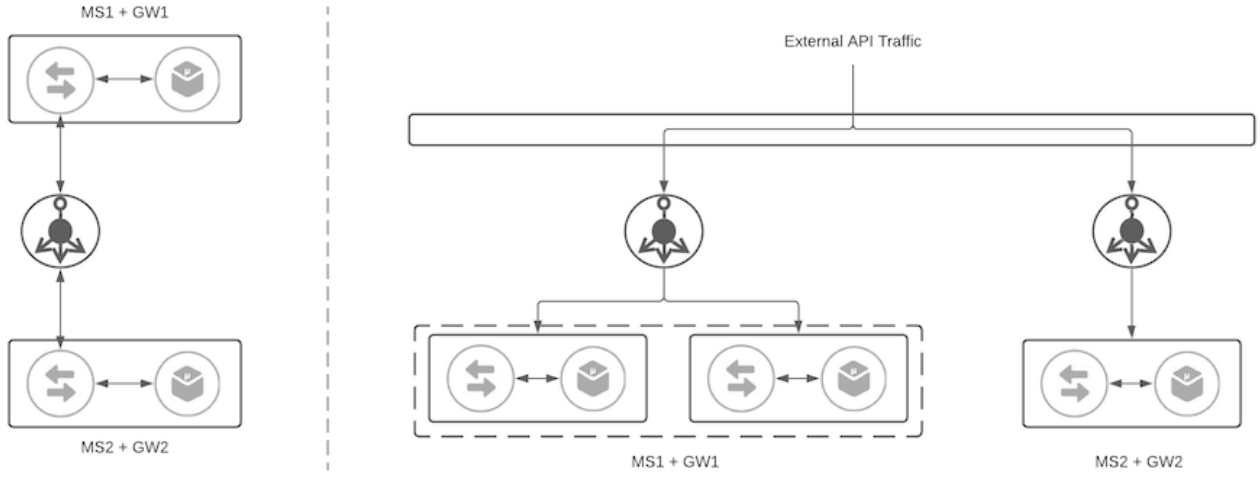


Figure 17 - Sidecar API gateway

The sidecar pattern reduces the additional external network hops that are required in the centralized and private jet gateway patterns, while having the local network call to communicate.

A sidecar is heavily used in service mesh architectural patterns. Offloading all service to service communication matters, such as discovery, reliable delivery, routing, failover, load balancing, etc., into a mesh sidecar will give freedom to developers to focus on business functionality.

A sidecar API gateway pattern can be used when and where you want to have service-mesh architecture.

All-in-One Gateway

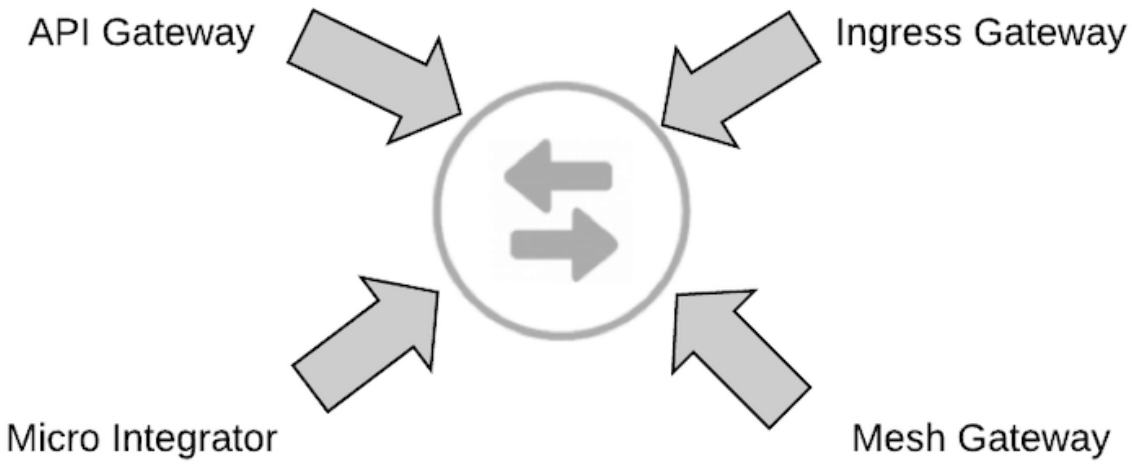


Figure 18 - Gateway convergence

Technology is evolving in a way that all types of gateways such as API gateways, ingress gateways, service mesh gateways, and micro integrators are merging into one single all-in-one gateway.

But even if these gateways are merging into a single gateway concept, depending on the use case and the requirement, in some cases, it is good to use multiple gateways to have a clean and scalable architecture.

Control and Management Plane

API gateways are the interception point to policy enforcement, capture stats, metrics, and analyze to find out how APIs are behaving. Managing these APIs is a necessity in today's digital economy. Control and management planes should provide the API management capabilities.

Design and Lifecycle Management

API design is a very important phase in the software development life cycle (SDLC). The design phase helps to gather developer feedback before implementing (API-first design). OpenAPI (Swagger) is the common industry standard to define the API design. Deploying a prototyped API, providing early access to APIs, creating mock API implementations, and getting early feedback are

some of the functionalities that are provided by design and lifecycle management.

Control Access and Enforce Security

Security is paramount when exposing business capabilities via APIs. This not only involves authentication and authorization but also covers policies to protect attacks, sensitive data leaking, revoke compromise APIs, blocking subscriptions due to non-payment, threat protection, bot detection, token-fraud detection, etc. APIs can be secured by using OAuth2.0, OIDC, Basic Auth, API Key, and Mutual TLS. The control and management planes can be used to define these security policies.

Manage and Scale API Traffic

API management enables users to control traffic flows to backend business services. API quotas and spike arrest helps to protect backend systems from being properly throttled and managed. The control and management planes should be able to define these policies and enforce them in the data plan via API gateways.

Observability

Unlike monolith architecture, auditing and tracing are hard problems in decentralized architectures such as MSA. Having the necessary interceptors to collect metrics, stats, and data is critical. The control and management planes should have capable analytics tools and engines to analyze these collected metrics, stats, and data to generate business intelligence reports. These reports can be utilized to monetize business capabilities by combining them with the defined business plans.

Developer Portal

It also important that these APIs are listed in an externally accessible self-service developer portal, where application developers or API users can easily discover these APIs and use them with a well-defined business plan. The developer experience is key to the adoption and success of your APIs, and having a feedback mechanism, such as customer ratings and forums, is key for a developer portal.

Business Insight Report

To become a successful digital enterprise, it is important to collect data, analyze, and get meaningful business insights on how these APIs are behaving. Comprehensive observability and business insight reporting systems play a major role here. These dashboards and reports can be used by both business and operations leaders to gain a 360-degree view of their digital business.

GitOps

Continuous integration and continuous deployment is critical to achieving agility. GitOps is a way of

implementing continuous deployment to cloud native applications. It combines the functionalities of Git and continuous deployment tools and provides a developer-centric experience when operating infrastructure.

A Git repository keeps all declarative deployment descriptions of the infrastructure in the given environment (dev, test, stage, prod, etc.) and continuous deployment tools automate the process to make the environment match the described state in the repository. During an unfortunate event, it is easy to rollback to a working state by referring to the Git revision.

In GitOps, as a general practice, you can have two Git repositories, one to keep application code revisions and another to keep track of declarative descriptions for deployments.

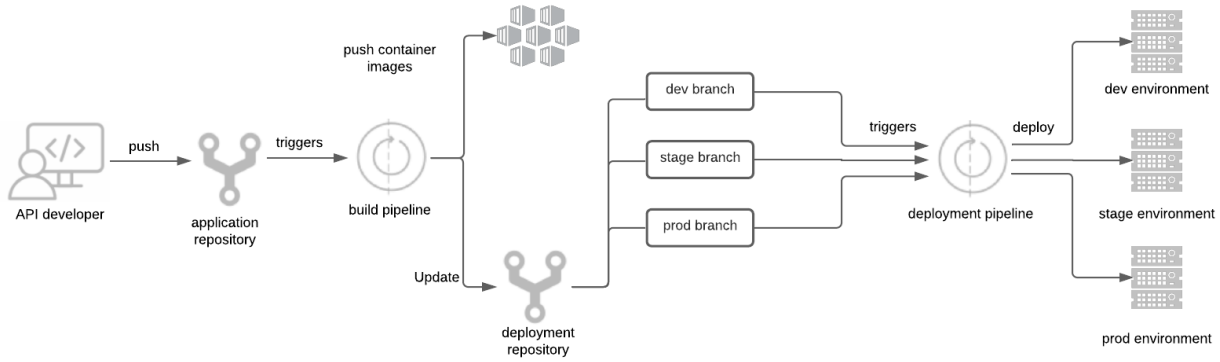


Figure 19 - GitOps

By configuring Git triggers for application source code push events, build pipelines can start the configured pipeline steps depending on the application requirements. One step could be building and pushing relevant container images. Another step could be creating declarative deployment descriptors and committing and pushing to the separate deployment Git repository.

If you have many deployment environments, then you can have separate Git branches for each environment. After completing environment testing, we can then promote to upper environments (e.g., stage to prod) by merging into the relevant Git branch. With this mode, If you want to deploy a new application or update an existing one, you only need to update the repository; the automated process handles everything else.

Conclusion

The digital enterprise enables companies of every sector to integrate and expose their business capabilities as APIs by digitalization of the entire value chain in their digital transformation journey. These APIs should be secured, managed, observed, and monetized. An API-led integration platform is essential for digital enterprises whether they start with a greenfield or a brownfield.

Cloud native applications are all about dynamism. Microservice architecture (MSA) is critical to accomplish agility. Cloud native technologies, such as containers and orchestration platforms, are

critical for successful microservice implementation and deployments. API gateways play a key role by enforcing policies that are defined in the control and management plane.

A self-service developer portal is important to build an effective API ecosystem. Dashboards and reports help both business and operations to gain a 360-degree view of their digital business.

Combining cloud native technologies with an API-led integration platform creates an effective architecture for a digital enterprise to increase productivity by having automation, production or operation, and services.

Appendix

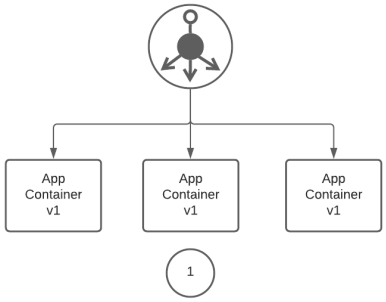
Abstractions

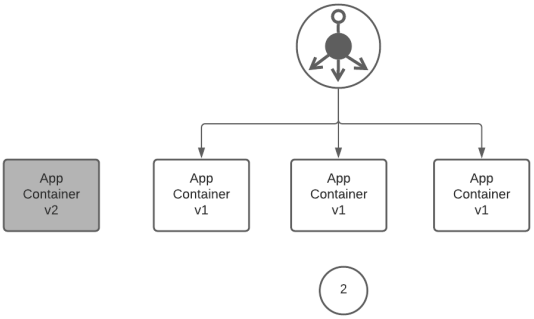
Icon	Name	Description
Component	Microservices and serverless components	Core business logic, aggregation and service composition, transformation.
Component	Gateways	API gateways, ingress gateways, mesh gateways, micro integrators, exposed APIs, events and streams, policy enforcement points
Component	Legacy and data services	Databases, existing systems, registries and repositories, user stores, business processes
SaaS EPR	External endpoint	Access using APIs, events, and streams, cloud systems, and SaaS
Front end Client	API consumers	Mobile apps, reactive apps, API consumers
Desktop Client	API consumers	Mobile apps, reactive apps, API consumers
Mobile Client	API consumers	Mobile apps, reactive apps, API consumers
Bot Client	API consumers	Bots, API consumers
IOT Client	API consumers	IoT devices, apps, API consumers
Key	Credentials	Credentials, keys, passwords
Config	Configurations	Configurations
Cert	Certificates	Certificates
Load Balancer	Load balancer	Load balancer

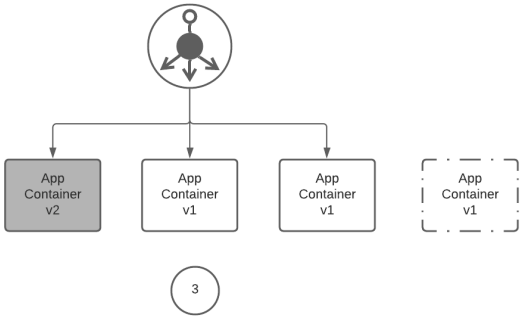
Deployment Strategies

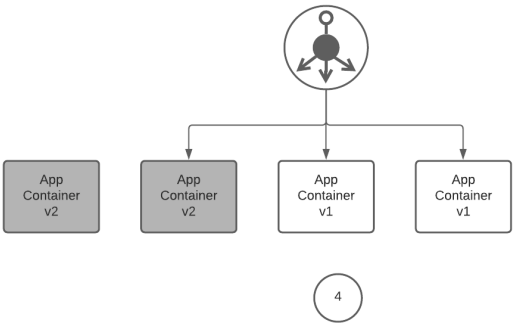
Ramped

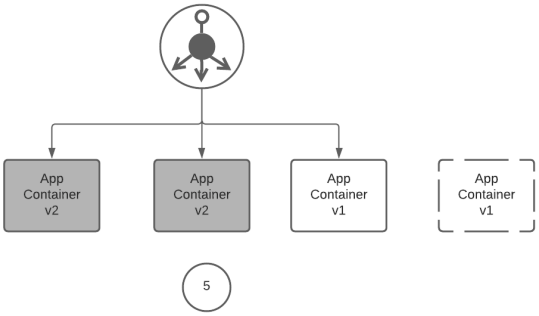
Ramped (also known as rolling-update) is the simplest rollout strategy that can be achieved with zero downtime. In this, a new version (e.g., version 2) is achieved by replacing containers one after another until all the containers are rolled out.

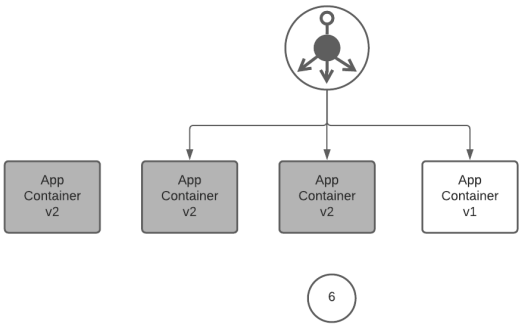

App Container v1
App Container v1
App Container v1
1

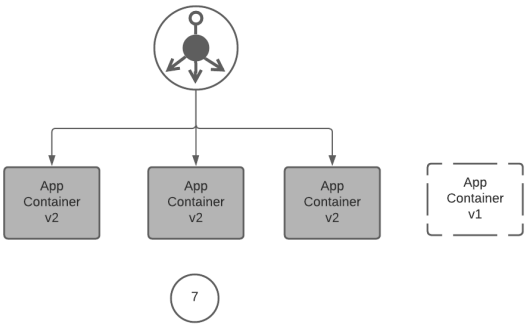

App Container v2
App Container v1
App Container v1
App Container v1
2


App Container v2
App Container v1
App Container v1
App Container v1
3


App Container v2
App Container v2
App Container v1
App Container v1
4


App Container v2
App Container v2
App Container v1
App Container v1
5


App Container v2
App Container v2
App Container v2
App Container v1
6


App Container v2
App Container v2
App Container v2
App Container v1
7

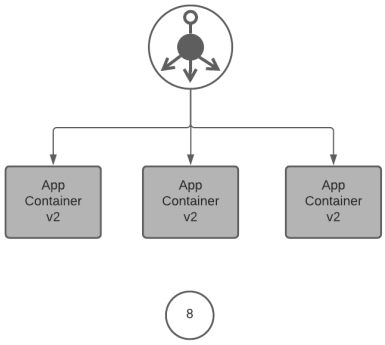

App Container v2
App Container v2
App Container v2
8

Figure 10 - Ramped deployment strategy

Blue/Green

The blue/green deployment strategy deploys version 2 (green) alongside version 1 (blue) with exactly the same amount of containers. After testing by running some for a time period, the new version traffic is switched from version 1 to version 2 at the load balancer level.

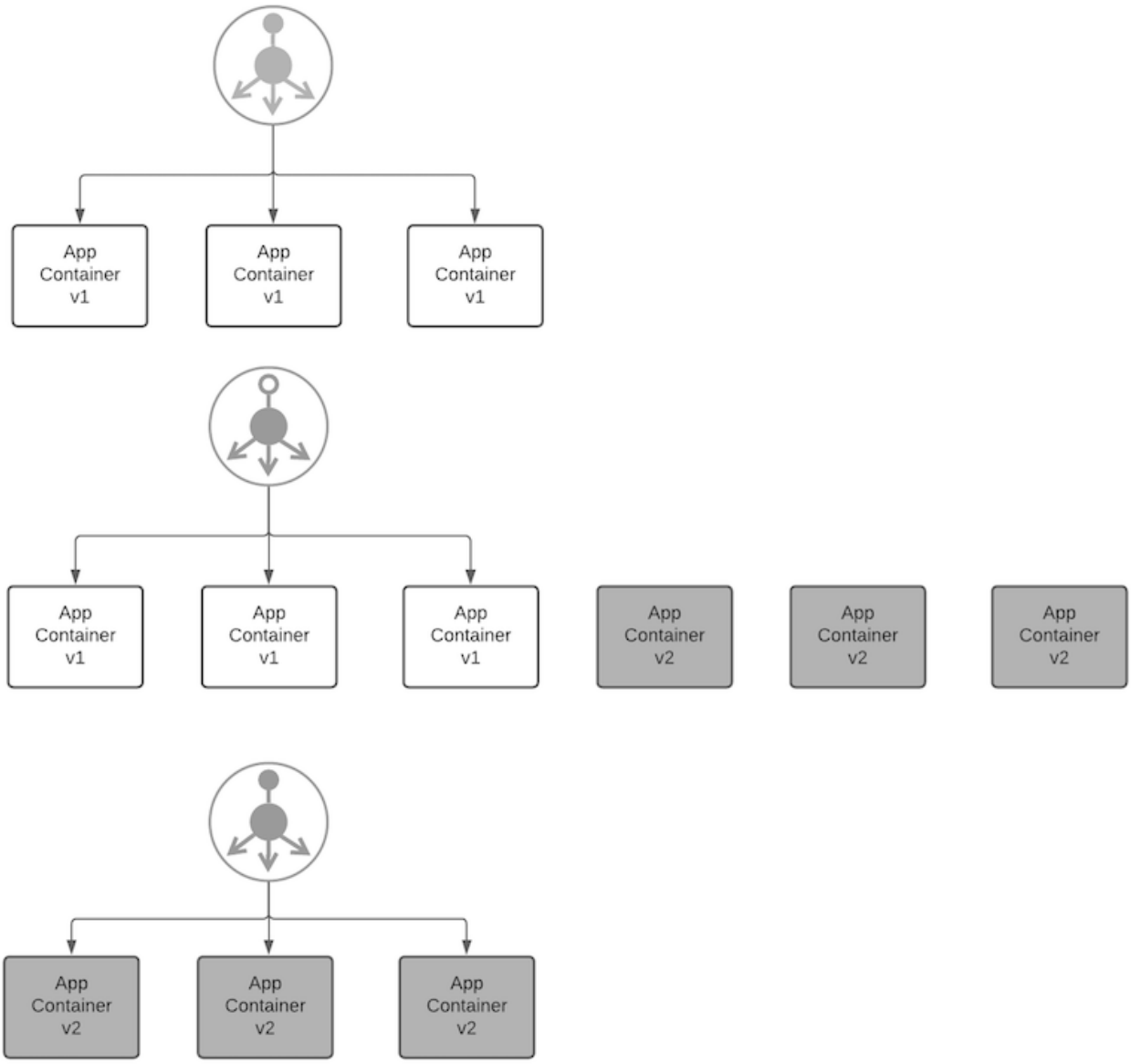


Figure 11 - Blue/Green deployment strategy

Canary

A canary deployment gradually shifts production traffic from version 1 to version 2. Initially a smaller percentage of traffic will be routed to the new version. Canary is mostly used when the tests are lacking or there is little confidence about the stability of the new version.

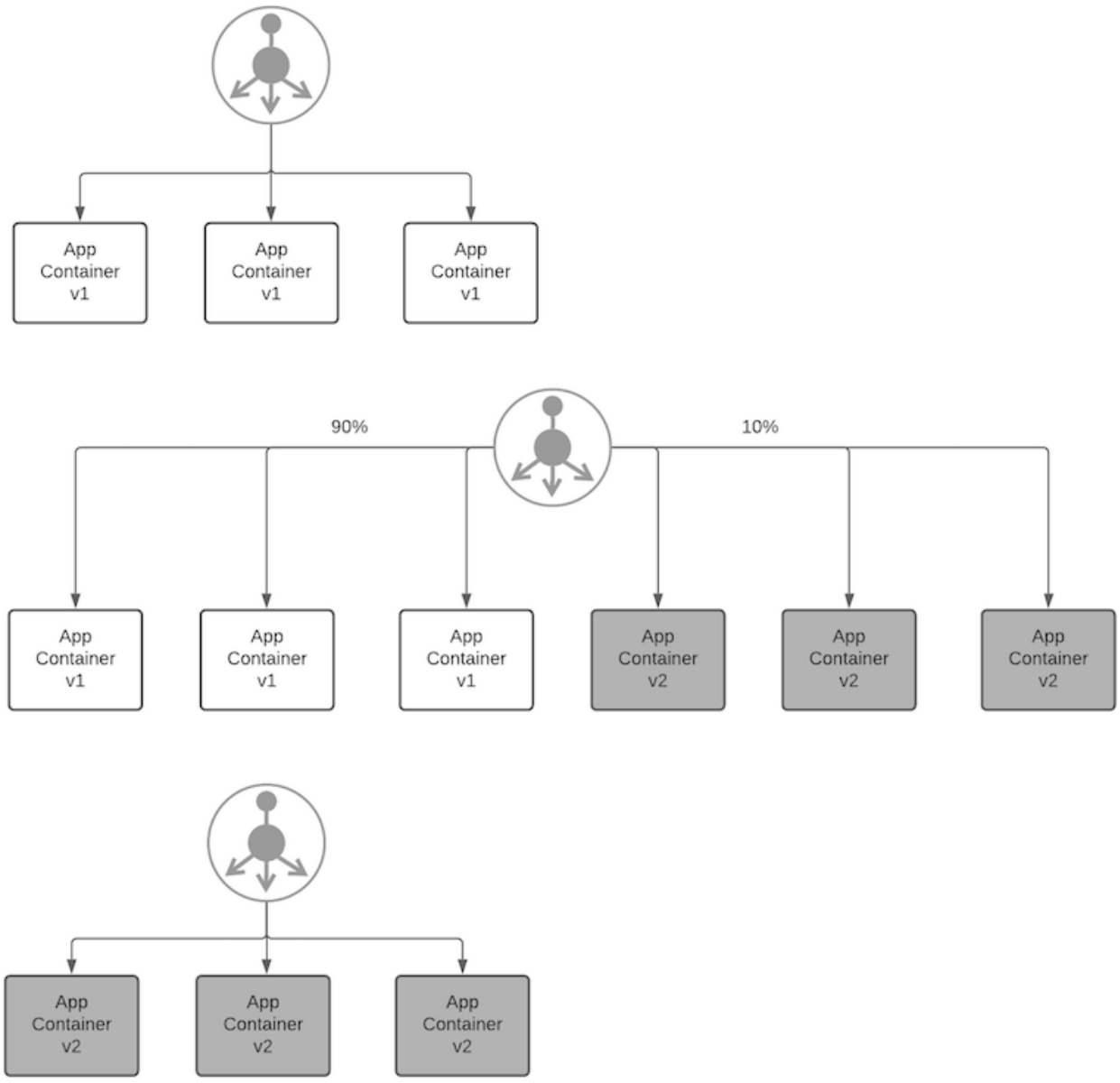


Figure 12 - Canary deployment strategy

A/B Testing

A/B testing deployments routes a subset of users to a new version (functionality) under specific conditions. This deployment strategy is used to test the conversion of a given feature and only rolls out the version that converts the most.

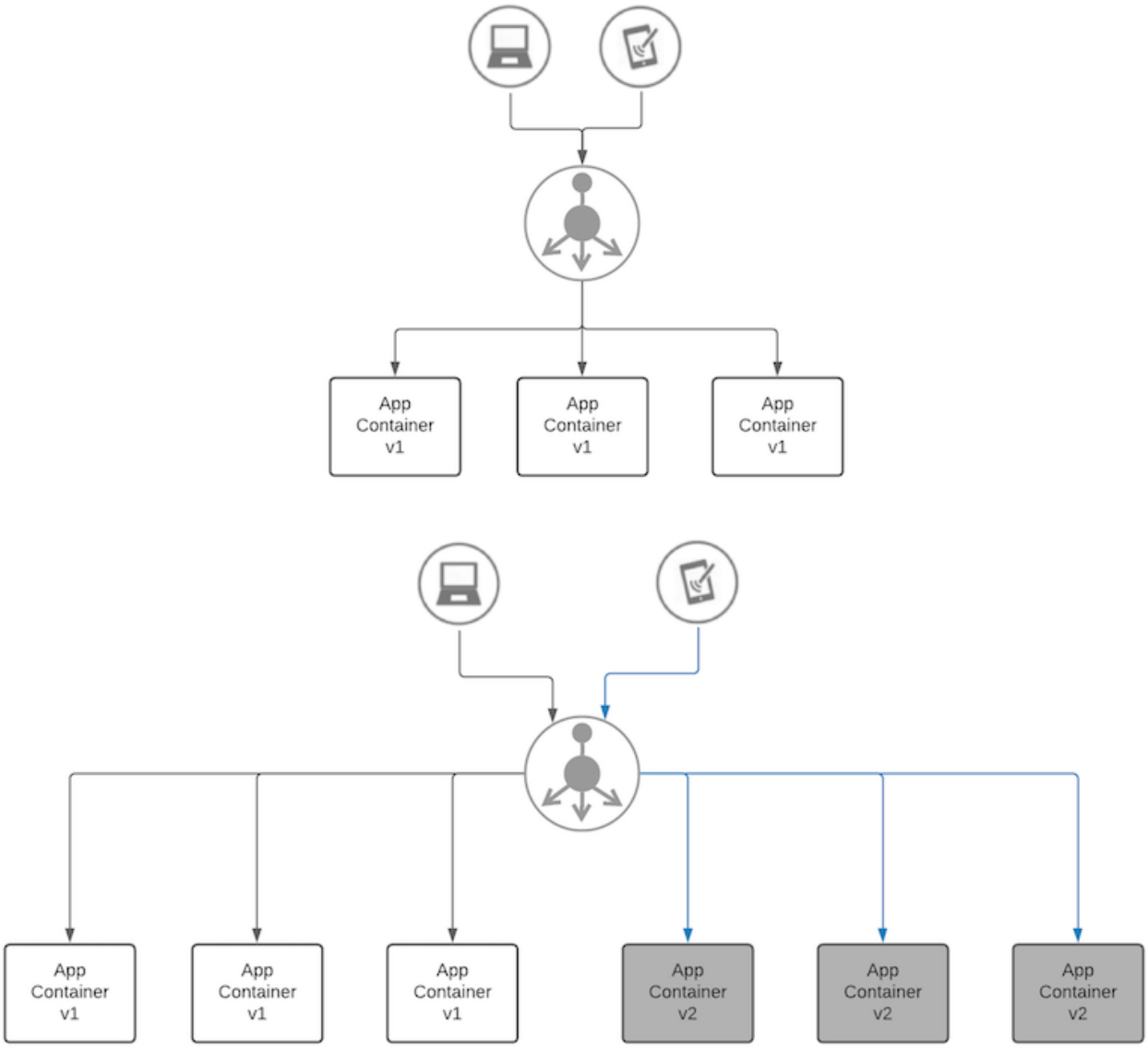


Figure 13 - A/B testing deployment strategy

Shadow

Shadow deployment strategies run version 1 and 2 together and fork version 1 incoming requests and send them to version 2 as well without impacting production traffic. This is a fairly complex deployment strategy and is mainly used to test production load on a new feature. When the required stability and performance are met, the new version of the application can be rolled out.

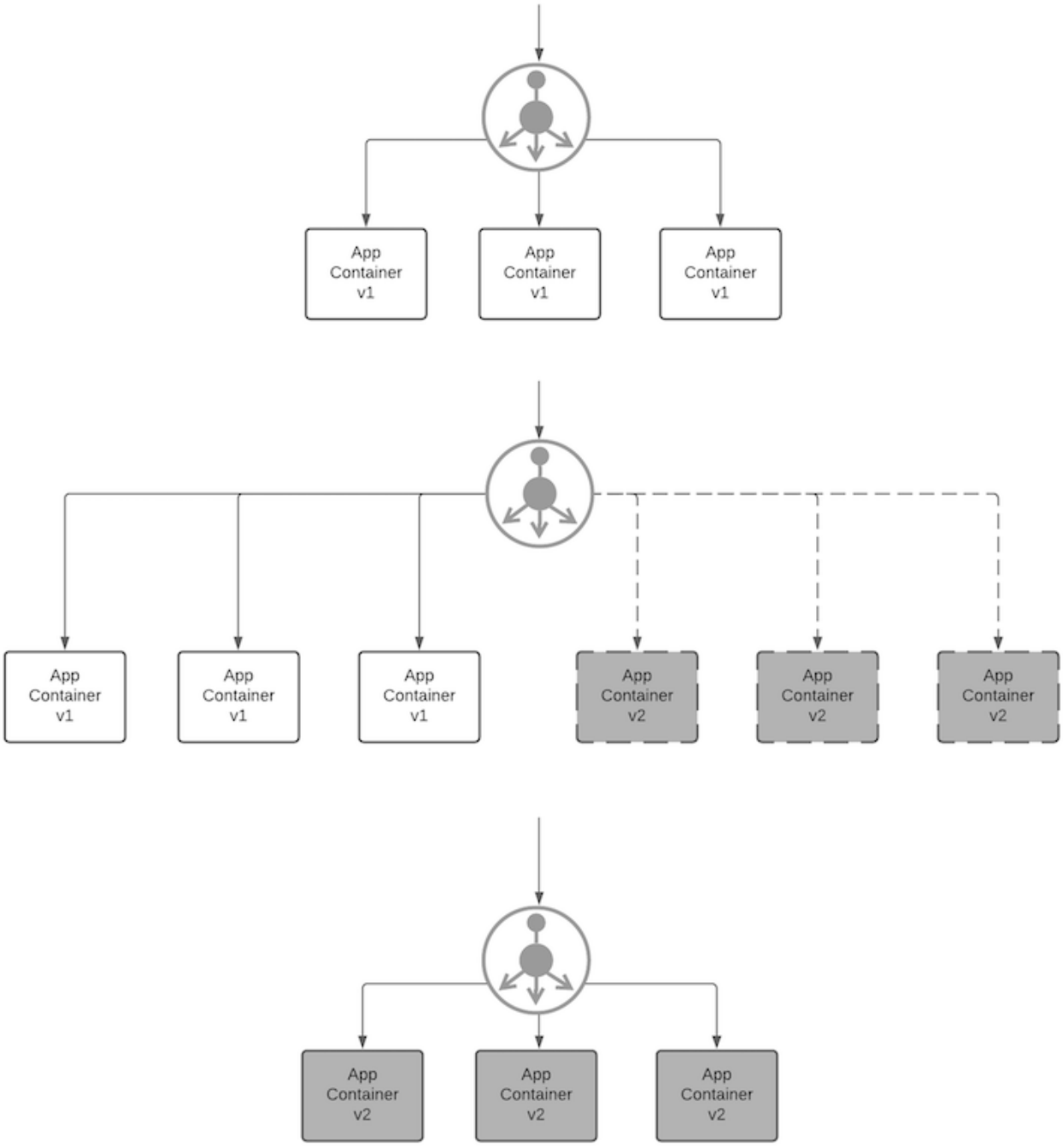


Figure 14 - Shadow deployment strategy

References:

- [1] Six Strategies for Application Deployment
- [2] API Microgateway
- [3] The twelve-factor Apps
- [4] GitOps