OpsRamp

EXECUTIVE'S GUIDE TO KUBERNETES

MICROSERVICES AND CONTAINERS IN PLAIN ENGLISH

The Executive's Guide to Kubernetes

Kubernetes is emblematic of a movement in technology -- a movement away from monolithic architectures, towards what's known as microservices architecture, where services are decoupled, isolated, and only as big as they absolutely have to be. These individual microservices are deployed in containers that are launched in seconds and may be terminated only after minutes of usage.

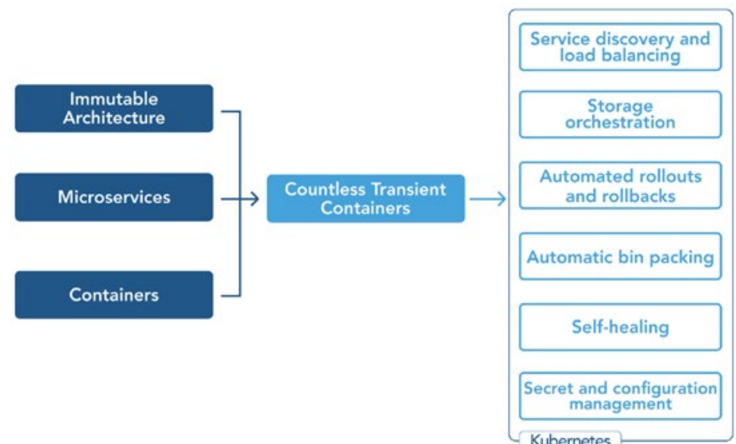
Since none of this is visible to an end user, why go through the work of unbundling and containerizing your monolith? Simple: it's easier to develop and maintain in nearly every way. From executing upgrades, to allocating resources with precision, to isolating performance issues, the advantages of a microservices architecture over a monolithic one are vast. In this guide, we'll go over everything you need to know to be an effective leader as your team tackles the process of building out microservices. But first, let's start with some basic ideas you'll see mentioned throughout all of these chapters.

Key Terms & Concepts

- **Containers:** A standalone, executable package of software that includes all necessary code and dependencies.
- **Immutable Architecture:** An infrastructure paradigm where servers are never modified, only replaced.
- **Infrastructure-as-Code:** The practice of provisioning and managing data center resources using humanly-readable declarative definition files (e.g., YAML).
- **Microservices:** A series of independently deployable software services that, together, make up an application.
- **Vertical Scaling:** Where you allocate more CPU or memory to your individual machines or containers.
- **Horizontal Scaling:** Where you add more machines or containers to your load-balanced computing resource pool.

Putting it all Together

By decoupling your services and converting them into micro services, you enable containerization for each individual service. This containerization compartmentalizes any resource needs, configuration needs, and performance issues for a particular service--allowing your technical leadership to align individual contributors with teams specialized to support a unique tech stack best tailored to power that service. Once containerized, your architecture can then become immutable to ensure the stability of production environments while also allowing for accurate and thorough testing of new image configurations.



OpsRamp
© OpsRamp, Inc. All Rights Reserved

Each of the following chapters provides a deeper look into the concepts, best practices, and challenges of transitioning to a Kubernetes-based containerized set of microservices.

CHAPTER 1: **Comparing Containers to Virtual Machines**

Learn why containers are more lightweight, faster to deploy, and easy to terminate when compared to virtual machines.

CHAPTER 2: **Benefits of Microservice-Based Architecture**

Get some guidance on how to prepare your organization for success when converting from monolithic- to microservice-based architecture.

CHAPTER 3: **Reasons to Use Immutable Architecture**

Discover why mutable architecture is more vulnerable to performance issues than immutable architecture and how you can employ infrastructure-as-code methodologies and tools to easily manage your servers.

CHAPTER 4: **Why Containers are Challenging without Kubernetes**

Understand the challenges unique to containerization regarding deployment, lifecycle management, network configuration, and scaling -- and how Kuberenetes solves them.

CHAPTER 5: **History of Kubernetes**

Walk through the evolution of Kubernetes since its origins, the main reasons for its dominance, and its rise to a highly automated and extensible platform configured with declarative files.

OpsRamp

© OpsRamp, Inc. All Rights Reserved

Container Vs. Virtual Machine

Containerization is hot right now. In 2020, 86% of technology leaders reported prioritizing containerization projects for their applications; by 2023, 70% of global businesses are expected to be running at least two containerized applications in a production environment. With such explosive adoption rates, more and more organizations debate whether to embrace the containerization trend or stay put on virtual machines. In this chapter, we'll review the basics of both solutions and make a direct comparison.

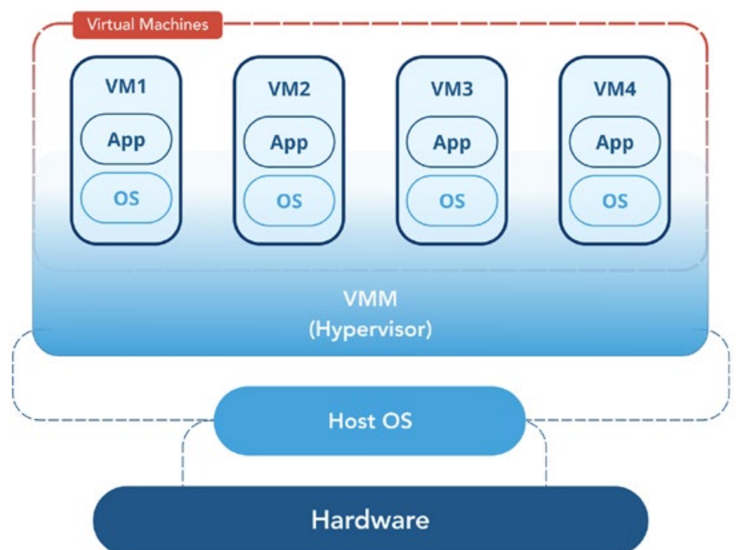
The simplest way to think of a container in relation to a virtual machine (VM) is that a VM virtualizes a physical server, whereas a container virtualizes an operating system (OS). The OS that containers virtualize may run on either a physical server or a virtual machine. But before we dig deeper into explaining each, let's quickly review the key benefits of both containers and virtual machines.

Container Benefits	Virtual Machine Benefits
Small memory footprint since the OS instance is shared across containers	You can install different OS types (Linux, Windows) in VMs hosted on the same server
Launch a new instance in seconds	Launch a new instance in minutes
Ideal for maintaining a fixed preset configuration and replacing it as quickly as needs change	Ideal for changing its configuration over time as you would when using a physical server
Ideal for a lifespan (from launch to termination) ranging from minutes to days	Ideal for a lifespan ranging from days to years
Ideally suited to host microservices	Ideally suited to support client-server applications or to host containers

WHAT IS A VIRTUAL MACHINE?

A virtual machine creates a software entity that functions the same way a computer system would. To run VMs, your physical machines must have a virtualization software known as a hypervisor (or Virtual Machine Monitor) to host the VMs and handle VM management. Even though VMware vSphere is the most popular commercially-licensed hypervisor, Microsoft Hyper-V is a common choice in a Windows environment, while free versions such as XEN (an early open source hypervisor project) and KVM (even though architected a bit differently) are commonly used to virtualize a Linux server. The hypervisor emulates the server's physical hardware so that the installed operating system (Linux, Windows, or macOS) won't know the difference between a VM and an underlying physical server. So each virtual machine has its dedicated OS and kernel. The memory footprint of a VM is almost as high as a small server since it requires a dedicated OS instance.

Virtual machines help organizations save on IT infrastructure costs by running many operating systems and applications on a small number of physical servers. VMs also allow you to configure an ideal environment for the changing needs of your applications over time.



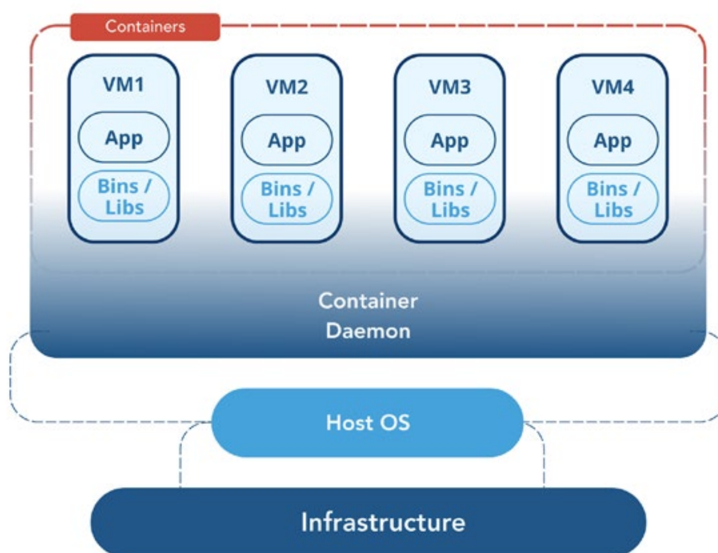
VMs running on top of a hypervisor that emulates the underlying hardware

© OpsRamp, Inc. All Rights Reserved

WHAT IS CONTAINERIZATION?

Containerization aims to provide a consistent and portable way to deliver software applications through containing the application software and its dependencies in “boxes” that can be rapidly copied and multiplied to scale up or down based on changing workload demands.

Containers work because they package all of your application’s dependencies and configurations (such as system files, software libraries, and device drivers) to ensure a predictable and consistent runtime--regardless of which environment they are deployed on, provided a Container Runtime Environment (also known as OS-level virtualization) such as Docker is installed on the Operating System. Docker remains by far the most popular container runtime engine while containerd and CRI-O are the runner-ups.



Conceptual depiction of a container

Host OS & The Kernel

The Hardware component represents a physical or virtual machine with an Operating System (OS). The OS has what is known as a kernel, which is a program that acts as a governing layer between all of the programs that run on the host machine and its physical components. The kernel plays a critical role in containerization, and because of this, you must deploy containers that are compatible with the underlying OS kernel (meaning containers are OS-specific).

Namespaces & Control Groups

Linux Operating System kernels have two distinct features known as *namespacing* and *control groups*.

- **Namespacing:** isolates resources (hard drive, networking, hostnames, users, etc) on a machine for a particular process
- **Control Groups (CGroups):** limits the amount of resources that a process can use

These two features combined allow you to isolate a single process (representing a container) and limit the amount of computing resources it can consume. Effectively, this combination is what we call containerization.

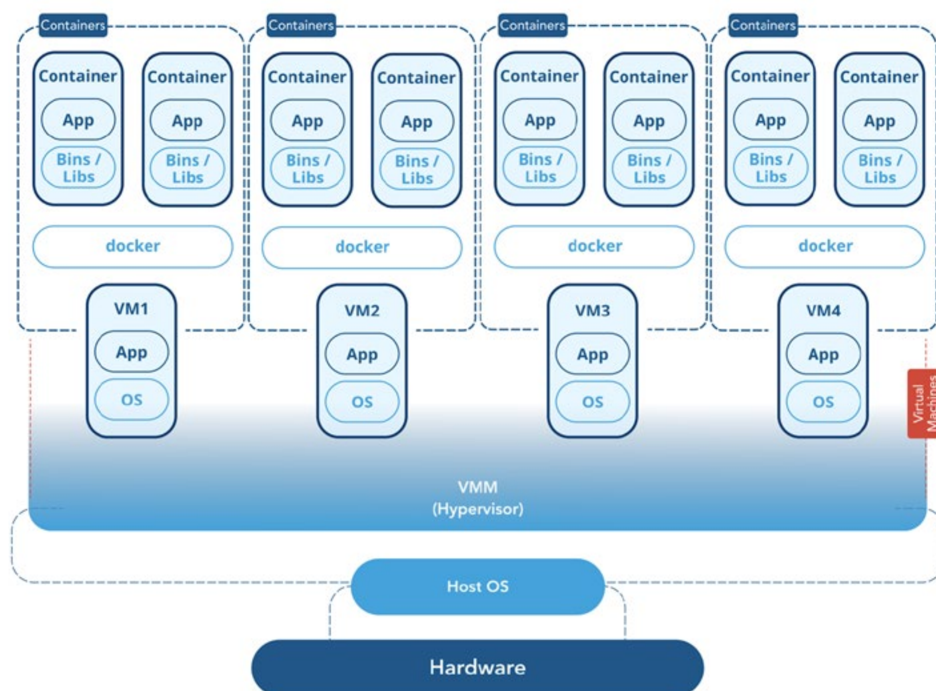
Although these features are unique to Linux, you can still use a Mac or Windows desktop machine. To do so, simply install a Container Runtime Environment such as Docker (or use native tools like Hyper-V and HyperKit) and deploy a Linux virtual machine to host the containers (while your desktop continues to function as before). The Linux VM provides the required kernel used to constrain access to the hardware resources on your machine.

© OpsRamp, Inc. All Rights Reserved

Containers vs VMs

Containers can be much lighter than VMs because they don't need their own OS to run. As a result, you can run multiple containers on a single server without wasting as much of your host machine's resources. This setup also translates into less computational overhead and faster startup times.

Here is a side-by-side visual comparison of their architectural differences:



COMPARING VIRTUAL MACHINE AND CONTAINER ARCHITECTURES

Let's summarize the information that we covered in this article in the form of a tabular comparison:

Although VMs are heavier, slower, and more costly, they still have a critical role to play in application delivery. VMs are critical to optimizing server hardware usage in a data center environment. In fact, the most popular AWS service is still the EC2 which is nothing other than a virtual machine running on a hypervisor hosted on an physical AWS server. The EC2 is often used by AWS clients to host a container runtime engine. Together, both technologies are extremely powerful and complimentary. When it comes to hosting applications architected to use microservices (see our next article), however, containers are the answer.

Containers	VMs
OS-level virtualisation	Hardware-level virtualization
Share a single host OS	Dedicated OS and kernel
Lightweight (computing footprint)	Relatively heavyweight
Quick startup time (seconds)	Slower startup time (minutes)
Process-level isolation	Fully isolated system
Much lower costs of running containers	Enterprise virtualization tech-nology is relatively costly

OpsRamp
© OpsRamp, Inc. All Rights Reserved

Microservices design has become a go-to architectural framework for teams developing distributed enterprise software. But what is this design, and how does an organization build or transition applications to properly become part of the microservices movement?

In this article, we'll explain the basic concepts of microservices, their advantages over traditional monolithic architectures, and how to build a culture that thrives using this design approach.

WHAT ARE MICROSERVICES?

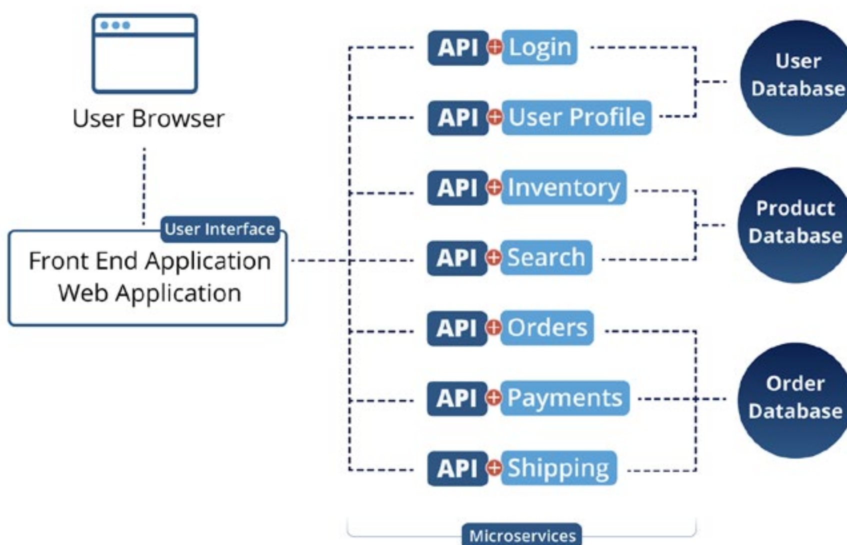
Microservices are small discrete software services communicating via an Application Programming Interface (API) that collectively form a business application. Each microservice is aligned with a business functionality and owned by a small team responsible for maintaining and testing each microservice.

As such, a microservice-based architecture is a type of software design that modularizes a system into a collection of services and sub-modules. The paradigm of separating software modules into services has existed since the dawn of computing including when Service Oriented Architecture (SOA) articulated it at the turn of the millennium. At a high level, the difference is that microservices are more granular and lightweight than ever conceived before.

These software modules each serve a unique purpose (such as login, user profile, search, reviews), but together make up one application. From a technical perspective, this architecture aims to create network-accessible services that meet the following criteria:

- Organized around single-purpose functionality
- Highly maintainable and testable
- Loosely coupled
- Independently deployable
- Owned by a small team

Let's use the example of an e-commerce platform to depict the benefits of a microservice-based architecture. At a micro level, you have business-specific workflows like inventory management, order management, shipping, and payment processing. To achieve a workflow, one or many microservices can be used to build the application feature; each microservice communicates with the front-end User Interface (UI) and each other via an Application Programming Interface (API).



Microservices are tied to APIs

With a model based on specific business domains, each microservice is treated as a black box accessible only via its API from the outside. As long as the API remains stable and consistent, the functionality of the microservice may independently change over time.

OpsRamp © OpsRamp, Inc. All Rights Reserved

4 WAYS TO EMBRACE MICROSERVICE DESIGN

Contrary to what some may think, microservices as an idea represents more than just a technical architectural model. It actually represents a culture-first approach to designing software that emphasizes development autonomy, accountability, flexibility, and technical freedom to achieve application excellence.

The success of microservice-style architecture requires mature and efficient communication to appease *Conway's law*, which suggests that organizations are destined to build design systems that mirror their communication structures. After all, it is easy to imagine that a poorly communicating organization would struggle to create a seamless application experience devoid of glaring inconsistencies such as conflicting schemas, unhelpful errors, etc.

Fortunately, there are four clear ways you can ensure that your organization is set up for success when building applications through microservices. Let's take a look.

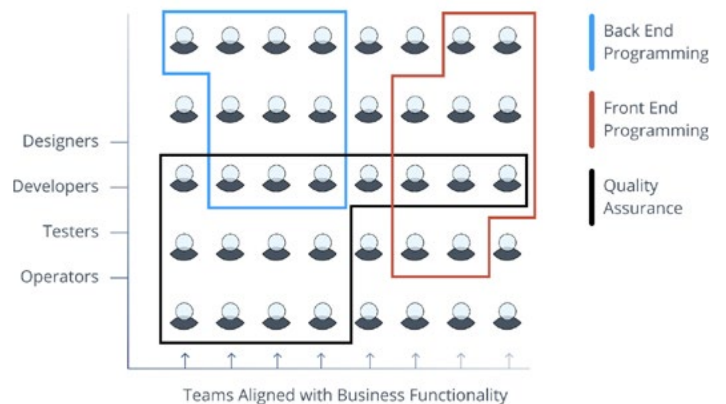
1. Structure Teams by Business Service

In most businesses, teams operate in silos which creates a very conventional and inefficient approach to developing and delivering software. A traditional organization separates functions such as:

- Product management
- Product marketing
- User experience design
- Front-end development
- Backend development
- Database administration
- Test engineering
- IT operations

Instead, under the microservices model, a multidisciplinary and cross-functional team owns a particular business service. Each team then consists of individuals who make up the respective service's lifecycle from end to end. This structure improves understanding of business requirements, team cadence, and software delivery.

Spotify has expanded on this theme to create an often-referenced matrixed organizational model (made up of squads, tribes, chapters, and guilds). Spotify's model creates teams aligned by business functionality while still keeping a role-based domain affinity for each contributor.



Teams Aligned with Business Functionality Implemented in Microservices

© OpsRamp, Inc. All Rights Reserved

2. Make Deployment Streams Independent

Since microservice teams work on services independent of other components of the larger system, they can adopt agile practices such as Continuous Integration and Continuous Delivery (CI/CD). A centralized, cross-functional team develops, tests, problem-solves, deploys, and updates services much faster than traditional teams who might get bottlenecked by other supporting teams.

For example, a microservice team working on a payments service for an application can add support for a new payment method and independently release it into a live application environment; a traditional team would likely have to coordinate with back-end, front-end, product, and potentially other service teams affected by their development changes. With so many additional complexities, simply aligning on scope, bandwidth, and priority often drags the velocity down for everyone involved.

3. Grant Technology Heterogeneity

One of the biggest lures to a microservice architecture is the technical heterogeneity that it offers. As you can imagine, technology heterogeneity only works if teams have strong service ownership (independence) instead of collective service ownership.

In a strong ownership model, each microservice team is in control of its own programming paradigms, technological decisions, deployment practices, and tools. They can independently choose technology stacks that are best suited for their respective microservices without worrying about scaling niche skillets across a broad team.

The strong ownership model is not without its downsides. As each team standardizes on a different technology, the movement of personnel across teams becomes constrained, as does procurement management and training. Spotify solves this challenge by introducing “guilds” where subject matter experts coordinate their selections of technologies, best practices, and tools.

4. Avoid Tight Coupling

Errors have a wide blast radius in a tightly coupled software system. In some cases, a single bug can render an entire application unusable. With microservice-based architecture, any given software fault or error can only ever impact that one underlying service. The remaining system can continue to function normally as long as dependent services don’t require its functionality. Additionally, resolutions can be turned around faster due to the smaller troubleshooting scope and strong ownership of the responsible team.

THE MODERNIZATION CHALLENGE

As infrastructure ages and new solutions become available, application owners must choose between modernizing or maintaining their products.

Modernizing Applications	Maintaining Applications
Requires costly long-term investments that may take months to complete.	Keeps additional costs at a minimum for higher margins.
Ensures competitiveness against emerging players.	Prevents user loss due to significant experience changes and instability.
Requires an organization to make structural changes.	Avoids risky organizational turmoil if changes are controversial.
Requires motivating employees to adopt new practices and technology.	Avoids the costly search of recruiting new experts.

© OpsRamp, Inc. All Rights Reserved

As you can see, there is a lot of risks involved in modernizing your applications -- however, the payoff is arguably the only thing that matters: continued application relevance in a fast-evolving landscape. That being said, you can find many applications that still run on backend technologies from the 1980s that have survived on UI updates alone.

Modernizing your applications and adopting a microservices approach should happen only when you choose to move away from the client-server application model for a Software as a Service (SaaS) model. This transition typically happens either when an application requires to scale beyond its originally designed capacity or when end-users no longer wish to install client software on their desktops. The investment required to support such a transition would be an ideal trigger to adopt a modern application architecture.

Conclusion

Adopting the microservices design for building applications requires setting up a culture of service independence, multi-disciplinary team ownership, and open communication. Transitioning to this model of development can be expensive and it is not without risks to both application stability and organizational harmony, however, the rewards can be great. Applications built using a microservices design are less vulnerable to global performance issues, faster to release, and easier to upgrade.

© OpsRamp, Inc. All Rights Reserved

An Introduction to Infrastructure as Code & Immutable Architecture

The old saying "the only constant in life is change" is especially true when talking about technology. In software development, concepts such as agile scrum and continuous delivery attempt to embrace change and streamline its delivery. A more drastic approach to making constant operational change more efficient is the use of an immutable architecture.

Knowing that iterative changes and upgrades are essential to the success of applications, this might seem odd. After all, change also affects the underlying infrastructure and its configurations that support your applications. So how would a business benefit from adopting anything described as immutable?

In this chapter, we'll explain the concept of immutable architecture, the advantages it offers, and how it supports the notion of Infrastructure as Code (IaC).

WHAT IS IMMUTABLE ARCHITECTURE?

Immutable architecture, also known as immutable infrastructure, is a term that can be a bit misleading. Coined by Chad Fowler, an immutable architecture doesn't mean that your environment should never change, but rather, once a specific instance (such as a container or virtual machine) is started, its configuration should never change.

Instead of upgrading or re-configuring the underlying infrastructure of that instance (of a container or virtual machine), you should simply replace it entirely with a new instance that has all of your required changes. You may need to replace the instance within minutes, days, or weeks due to a workload change, an architecture change, or simply to keep up with changes going on elsewhere in your environment. Either way, replacing the instance allows for discrete versioning in your application environment. You can think of each version as a configuration snapshot at an exact point in time. Systematic versioning, in turn, lowers the risk of making mistakes during upgrades, offers the ability to test before rolling out, and enables rolling back (to the previous version) if your application encounters a problem. Combining flexibility with precision is the primary goal of an immutable architecture.

The practice of immutable architecture doesn't have to be limited to just the underlying infrastructure of your workloads; you can use this technique to support middleware components (such as a messaging bus, a database, or a data cache) and application software as well. You would simply release application source code as new, immutable, and versioned artifacts. You may version each package in the form of a new Docker image, a new Virtual Machine Image, or a new .jar file (Java code).

MUTABLE VS. IMMUTABLE ARCHITECTURE

Let's use a real-world example to illustrate the differences between a mutable and an immutable architecture. Say your company has an application web server running on a VM in the cloud. This web server has Nginx (webserver) installed on it and a specific web application version. After some time passes, you decide it's time to upgrade the version of Nginx or switch to Apache.

OpsRamp

© OpsRamp, Inc. All Rights Reserved

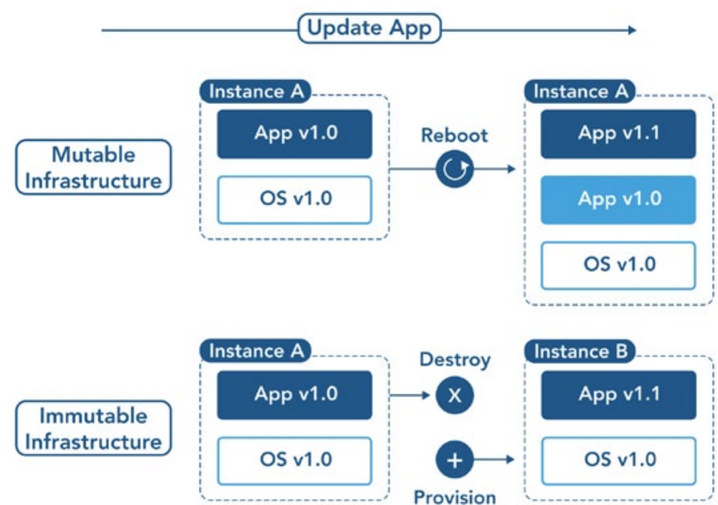
Upgrading Mutable Architecture

In a mutable architecture, you would simply upgrade your existing web server to the new version. You would affect such an upgrade using a configuration tool such as Chef, Puppet, Ansible, or SaltStack, or complete a manual upgrade.

However, what happens if the upgrade doesn't go as planned? There's a host of factors that could disrupt this process (network issues, DNS failure, dependency repository unavailability). These disruptions can result in your system being only partially upgraded, in-between your current and desired state. The implications of this can be unexpected behavior from the application because this in-between state would not have been validated and tested. This scenario might seem simple enough when considering a single VM, but this becomes exponentially complex with a large fleet of VMs.

Upgrading Immutable Architecture

In an immutable architecture, you would not upgrade your web server currently in place. To use the new version of Nginx, you would deploy the web server on a new VM. By using a different machine, you circumvent the need to upgrade any existing infrastructure. If the new machine encounters any errors, you can abort; if it's working as expected, you can redirect traffic to the new web server and decommission the old instance, as illustrated in the diagram right.



Here's a summary of the benefits of adopting an immutable architecture.

Immutable Architecture	Mutable Architecture
Streamlines operations	Requires reviews to ensure configuration consistency across nodes
Supports continuous deployment of a software application by matching infrastructure version to an application version	Requires ongoing configuration changes of the underlying infrastructure to support application updates
Mitigates manual errors that may result in security threats	Exposes risk of configuration inconsistency across instances
Supports scaling of infrastructure by adding and removing nodes as needed	Offers less control in rapidly replicating an exact configuration
Reduces operational costs	Increases operational overhead

OpsRamp
© OpsRamp, Inc. All Rights Reserved

WHAT IS INFRASTRUCTURE AS CODE?

The adoption of the public cloud accelerated the appeal of an immutable architecture. Not long ago, companies would have had to deal with a lot of physical infrastructure management overhead to replace computing nodes. Nowadays, implementing an immutable architecture has been simplified by cloud providers who automate resource provisioning on their self-service platforms. Today, your application's underlying infrastructure can be more easily versioned using code.

The main idea behind Infrastructure as Code (IaC) is to enable writing and executing code to define, deploy, update and destroy infrastructure by declaring the desired state. This trend has propelled Terraform to become the most popular open-source provisioning tool used by companies like Uber, Slack, Udemy, and Twitch.

TYPES OF IAC TOOLS

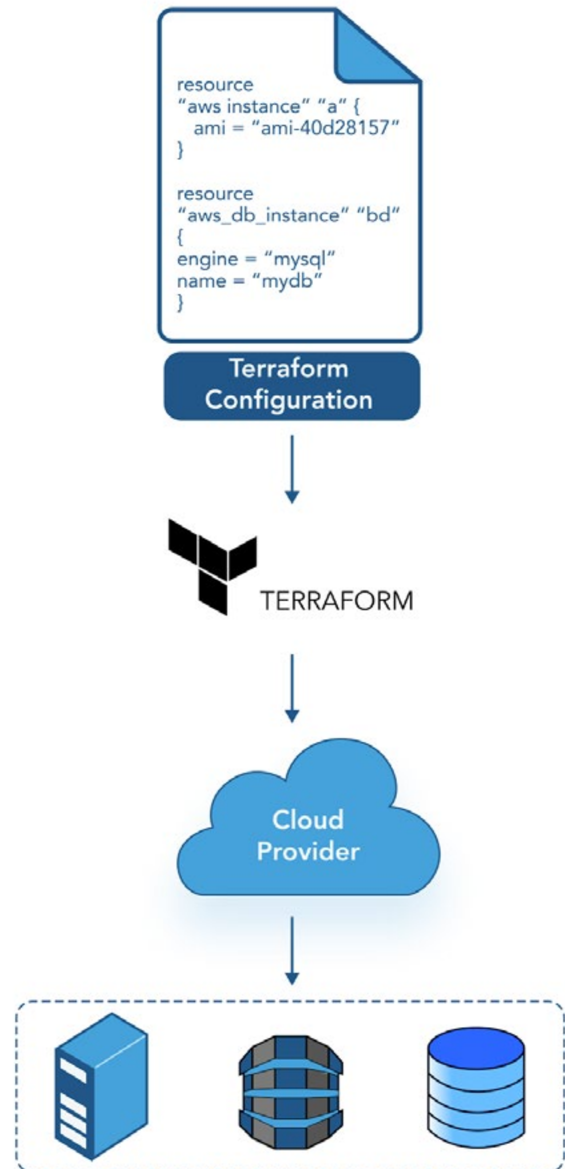
There are five broad categories of tools used to configure and orchestrate infrastructure and application stacks, even though the last category on our list is the only one recognized as a proper Infrastructure as Code (IaC) tool. It is helpful to see them defined and represented by examples, as summarized in the table below.

Ad hoc scripts	The most straightforward approach to automating anything is to write an ad hoc script. You take whatever task you were doing manually, break it down into discrete steps, using scripting languages like Bash, Ruby, and Python to define each of those steps in code, and execute that script on your server.
Configuration management tools	Chef, Puppet, Ansible, and SaltStack are all configuration management tools designed to install and configure software on existing servers that perpetually exist.
Server templating tools	An alternative to configuration management that has been recently growing in popularity is server templating tools such as Docker, Packer, and Vagrant. Instead of launching and then configuring servers, the idea behind server templating is to create an image of a server that captures a fully self-contained "snapshot" of the operating system (OS), the software, the files, and all other relevant dependencies.
Orchestration tools	Kubernetes would be an example of an orchestration tool. Kubernetes allows you to define how to manage your Docker containers as code. You first deploy the Kubernetes cluster, which is a group of servers that Kubernetes will manage and use to run your Docker containers. Most major cloud providers have native support for deploying managed Kubernetes clusters, such as Amazon Elastic Container Service for Kubernetes (Amazon EKS), Google Kubernetes Engine (GKE), and Azure Kubernetes Service (AKS).
Infrastructure as Code Provisioning tools	Whereas configuration management, server templating, and orchestration tools define the code that runs on each server or container, infrastructure as code provisioning tools such as Terraform, AWS CloudFormation, and OpenStack Heat define infrastructure configuration across public clouds and data centers. You use such tools to create servers, databases, caches, load balancers, queues, monitoring, subnet configurations, firewall settings, routing rules, and Secure Sockets Layer (SSL) certificates.

OpsRamp
© OpsRamp, Inc. All Rights Reserved

The image on the right depicts an example flow of using Terraform to create a Virtual Machine instance and a database in the AWS cloud environment.

Using Infrastructure as Code to define application environments, companies can eliminate the risks of configuration drifts and accomplish more reliable outcomes in their architectures.



Conclusion

Companies looking to extend the benefits of discrete and immutable versioning from their software applications to the entire architecture would benefit a great deal from adopting the Infrastructure as Code model presented in this article. Cloud-native services, automation, immutable architecture, and Infrastructure as Code (IaC) are integral components for scaling modern applications.

© OpsRamp, Inc. All Rights Reserved

The Challenges of Container Management Without Kubernetes

For the last decade, Virtual Machines (VMs) have been the backbone for software applications deployed to a cloud environment and still offer a great deal of trusted maturity. However, when it comes to application portability and delivery, containerization has overtaken virtualization.

Today, it's common for organizations to operate thousands of short-lived containers, each configured according to different workload requirements. These containers must be provisioned, connected to a network, secured, replicated, and eventually terminated. Although one person can manually configure dozens of containers, a large team must operate thousands of containers across a large enterprise environment.

As this management-scaling problem became more apparent for enterprise-level container use cases, an opportunity arose for orchestration tools like Kubernetes. In this article, we'll take a look at the challenges of managing containerized applications without Kubernetes, especially in a public cloud -- but first, let's talk a bit more about Kubernetes.

WHAT IS KUBERNETES?

Kubernetes is a container orchestrator platform that we introduce in this article and cover in-depth in our future articles. Commonly referred to as K8s (the number 8 represents the eight letters between "K" and "s"), Kubernetes is an open-source project and the industry's de-facto standard for automating deployment, scaling, and administration of containerized applications.

SCALING REQUIRES TASK AUTOMATION

Managing containers is challenging without automating resource allocation, load-balancing, and security enforcement. The list below highlights some of the tasks that require automation:

- Provisioning containers based on predefined images (OS, dependencies, libraries)
- Balancing incoming traffic across groups of similar containers
- Adjusting the number of containers based on workload demand (horizontal scaling)
- Allocating the right amount of CPU and memory for each container (vertical scaling)
- Configuring network ports to enable secure communication between containers
- Connecting and removing storage systems attached to containers
- Restarting containers if they fail

 © OpsRamp, Inc. All Rights Reserved

4 CONTAINER MANAGEMENT CHALLENGES MADE EASIER WITH AN ORCHESTRATION TOOL

Below, we dig deeper into four of the many tasks requiring automation in container administration.

1. Deploying

Deploying an application requires many steps.
Here are a few examples:

- Installing device drivers that manage the server hardware modules
- Installing the latest operating system patches
- Installing software libraries required by the application to run
- Ensuring communication with needed services such as a database
- Making sure the IP address is registered with the domain name service
- Installing virus and vulnerability scanners
- Scheduling backups

Most infrastructure deployment automation technology that existed before Kubernetes uses a procedural approach towards deployment configuration steps. This approach is known as imperative. Examples of such configuration automation tools are Ansible, Chef, and Puppet.

An early and important decision during Kubernetes' inception was the adoption of a declarative model. A declarative approach eliminates the need to define steps for the desired outcome. Instead, the final desired state is what is declared or defined. The Kubernetes platform automatically adjusts the configuration to reach and maintain the desired state. The declarative approach saves a lot of time as it abstracts the complex steps. The focus shifts from the 'how' to the 'what.'

2. Managing the Lifecycle

Docker is just one example of a container runtime engine that packages your application and all its dependencies together for delivery to any runtime environment. Another example is containerd. Below are some of the more common events in the lifecycle of a container handled by a runtime engine.

- Pull an image from a registry, or Docker Hub
- Create a container based on an image
- Start one or more stopped containers
- Stop one or more running containers

One person can complete these tasks when managing a small number of containers on a few hosts. However, attempting to carry this out manually falls far short in an enterprise environment with hundreds of nodes and thousands of containers. Kubernetes allows you to simply "declare" what you want to accomplish rather than code the intermediate steps. By using a container orchestration platform, you achieve the following benefits:

- Scaling your applications and infrastructure easily
- Service discovery and container networking
- Improved governance and security controls
- Container health monitoring
- Load balancing of containers evenly among hosts
- Optimal resource allocation
- Container lifecycle management

3. Configuring the Network

One of the most complex and least appreciated aspects of managing multiple containers is the network configuration. This step is required so the containers can communicate with each other and with other networks beyond the cluster. What complicates this process is that the containers rapidly start and terminate, and one mistake could lead to a security exposure. In the absence of automation tools, teams must configure networking identity for all applications and load balancing components and set up security features for ingress and egress of traffic.

With Kubernetes, software teams declare the desired state of networking for the application before being deployed. Kubernetes then maps a single IP address to a Pod (the smallest unit of container aggregation and management) that hosts multiple containers. This approach aligns the network identity with the application identity, simplifying the complexity required of the container networking layer and enabling easier maintenance at scale.

 © OpsRamp, Inc. All Rights Reserved

4. Scaling

Scaling the infrastructure to match the application workload requirement has always been a challenge.

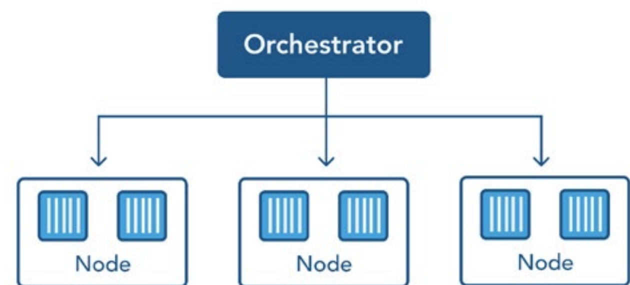
Suppose you are managing a few microservices running on a single server, and you are responsible for deploying, scaling, and securing these applications. Management may not be too difficult, assuming they're all similarly developed (same language, same OS). But what if you need to scale to a thousand deployments of different types, moving between local servers and the cloud? The challenges in such a scenario are:

- Identifying containers that are under or over-allocated
- Knowing whether your applications are appropriately load-balanced across multiple servers
- Knowing whether your resource cluster contains enough nodes for peak usage times
- Rolling back or updating all applications
- Modifying all deployments through a centralized portal or CLI
- Enforcing your security standards across all infrastructure

Kubernetes automates the workflows required to provide these types of functionality. Kubernetes organizes the hardware nodes as clusters and the containers as pods. You configure your desired state in a declarative fashion, and it takes over its administration to achieve your desired state of availability, performance, and security.

CONTAINER ORCHESTRATION

Container runtime engines such as Docker provide an OS virtualization platform to operate containers on a physical server or a virtual machine; however, they don't handle administrative or orchestration tasks. Instead, the runtime engines rely on Kubernetes to take on the orchestration responsibilities. Container orchestration automates the scheduling, deployment, networking, scaling, health monitoring, and container management.



Here is a summarized list of the functionality provided by Kubernetes:

Service discovery and load balancing	Identifies containers and balances traffic across them
Storage orchestration	Automatically mounts storage of your choice
Automated rollouts and rollbacks	Launches, stops, or re-assigns containers as needed
Automatic bin packing	Provisions desired container CPU and memory
Self-healing	Restarts failed or unhealthy containers
Secret and configuration management	Stores sensitive passwords and security keys

Conclusion

Managing containerized applications in a large-scale production environment presents DevOps teams with multiple tasks related to optimal deployment methods, networking configurations, security management, and scalability, to name a few. Automation and orchestration are required to manage hundreds or thousands of containers. Kubernetes offers functionality such as self-healing, automated rollouts/rollbacks, container lifecycle management, a declarative deployment model, and rich scaling capabilities for both nodes and containers.

This quote from Spotify summarizes well the benefits of Kubernetes: *"Our internal teams have less of a need to focus on manual capacity provisioning and more time to focus on delivering features for Spotify."*

© OpsRamp, Inc. All Rights Reserved

Who Made Kubernetes And Why Is It Popular?

Today, Kubernetes is the container orchestration solution to use. What began as an idea between Craig McLuckie, Joe Beda, and Brendan Burns in 2013 has quickly grown to become one of the most popular open-source projects in the world. Even as the leading solution, Kubernetes continues to experience exponential growth from enterprise adoption. And for the last two years, developers on StackOverflow have ranked Kubernetes as one of the most "loved" and "wanted" technologies.

This surge in popularity is directly tied to the rapid adoption of containers — like Docker containers — in application delivery. Containers have been at the heart of modern microservices architectures, cloud-native software, and DevOps workflows.

However, orchestrating container deployments can be difficult, time-consuming, and complex to scale without the right tools. Kubernetes is purpose-built to address these challenges. K8s (a shorthand where the number 8 represents the eight letters between "K" and "s") greatly reduces the complexity and manual work of container orchestration. As a result, major enterprises including Google, Spotify, Niantic, The New York Times, Asana, and China Unicom have used Kubernetes to streamline and scale up their container deployments.

In this article, we'll explore the history of Kubernetes and the main reasons it has grown to become the most popular Container Orchestration Engine available today. availability, performance, and security.

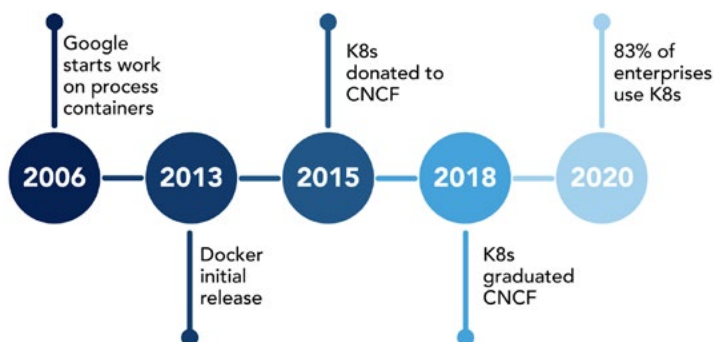
THE HISTORY OF KUBERNETES

Kubernetes has its roots in Google's internal Borg System, introduced between 2003 and 2004. Later, in 2013, Google released another project known as Omega, a flexible, scalable scheduler for large compute clusters. In that same year, McLuckie, Beda, and Burns set out to develop a "minimally viable orchestrator." The desired set of essential features for the orchestrator included:

- **Replication:** to deploy multiple instances of an application
- **Load balancing and service discovery:** to route traffic to these replicated containers
- **Basic health checking and repair:** to ensure a self-healing system
- **Scheduling:** to group many machines into a single pool and distribute work to them

The following year, Kubernetes was released.

Today, Kubernetes has 1800+ contributors, 500+ meetups worldwide, and 42,000+ users (many of them joining the public #kubernetes-dev channel on Slack). 83% of enterprises surveyed by the Cloud Native Computing Foundation (CNCF) in 2020 are using Kubernetes.



A brief history of Kubernetes

OpsRamp
© OpsRamp, Inc. All Rights Reserved

4 REASONS FOR KUBERNETES' POPULARITY

#1 The Rise of Containers

Containers paved the way for Kubernetes. But what's a container?

Containers are lightweight software components that bundle or package an entire application and its dependencies (such as software libraries) and configuration (such as network settings) to run as expected. This approach is becoming increasingly popular as an alternative to Virtual Machines when it comes to application portability.

Containerization of applications brings a number of benefits, including the following:

- **Portability:** Containers are truly "write once, run anywhere" (cloud, physical server, virtual server, etc.)
- **Efficiency:** Containers use fewer resources than virtual machines (VMs) and better utilize computing resources.
- **Agility:** With containers, developers can easily integrate into — or develop new — automated DevOps code delivery pipelines.
- **Higher development throughput:** Containers are ideal for hosting microservices owned by smaller independent development teams. As a result, they can deliver application upgrades faster.
- **Faster start-up:** Containers virtualize the operating system the same way virtual machines virtualize the physical server hardware. As such, they are lightweight, which helps them launch in seconds instead of minutes.
- **Flexibility:** Containers can run on virtual machines or bare metal hardware.

To take advantage of all these benefits at scale, software teams required orchestration tools to deploy and manage hundreds or thousands of containers which drives the adoption of Kubernetes.

#2 The Rise of Cloud

The growth of cloud computing has also been a major contributing factor to Kubernetes' widespread adoption. Cloud computing offers businesses the opportunity to use as many resources as they need when they need them. The pay-per-use model combined with the ability to rapidly provision and decommission resources make it an ideal platform for hosting a Kubernetes cluster requiring varying node count to accommodate changing workloads.

Kubernetes, by nature, is a cloud-agnostic system that allows companies to provision the same containers across public clouds and private clouds (also referred to as the hybrid cloud). The hybrid cloud model is a popular choice for enterprises, making Kubernetes an ideal solution for their use case.

Benefits of Kubernetes for hybrid cloud models include:

- Consistency across on-premise and public cloud.
- Portability of workload across platforms.
- Automation of provisioning processes spanning data center and cloud.
- Automated scaling of computing resources to maintain performance.

© OpsRamp, Inc. All Rights Reserved

#3 The Declarative Model

Before the release of Kubernetes, comparable tools automated the step-by-step procedures of deployment activities. But Kubernetes took a different approach that declares what the desired state of the system should be. Once the desired state is defined, Kubernetes continuously updates the underlying configuration necessary to achieve and maintain the targeted state.

This declarative paradigm removes the complexity of planning every step involved in the deployment and scaling processes and is therefore significantly more scalable in large environments.

#4 Extensibility

Kubernetes is a highly extensible platform consisting of native resource definitions such as Pods, Deployments, ConfigMaps, Secrets, and Jobs. Each resource serves a specific purpose and is key to running applications in the cluster. Software developers can also add Custom Resource Definitions (CRD) via the Kubernetes API server.

Also, Kubernetes enables software teams to write custom Operators, a specific process running in a Kubernetes cluster that follows what is known as the control pattern. An Operator allows users to automate the management of Custom Resource Definitions by talking to the Kubernetes API. Because K8s is so open and extensible, it can meet the demands of a wide range of use cases, limiting the need for engineers to turn to other tools for container orchestration.

Conclusion

As the cloud-native space continues to grow, more businesses will look to cloud-computing solutions to offer elasticity. Going hand-in-hand with this elastic cloud model is the use of containers for easier portability and rapid delivery of application workloads.

While these paradigms have solved many problems, they've also introduced new administrative complexities at a large scale that require the automated orchestration offered by K8s. Kubernetes has received more attention and adoption than any other infrastructure technology in recent memory. This level of adoption should last for years to come since K8s is free, sophisticated, stable, and well ahead of any other competing technology.

 © OpsRamp, Inc. All Rights Reserved