ICT academy

Uvod v oblačne tehnologije (Cloud Computing)

LTFE

Univerza v Ljubljani
Fakulteta za elektrotehniko

IKT Katedra za informacijske in komunikacijske tehnologije

LTFE Laboratorij za telekomunikacije

LMMFE Laboratorij za multimedijo

www.ict-academy.eu

ICT academy

Urnik

1. del

- Introduction to Cloud Computing
- Virtualization and Containerization
- Microservices

2. Del

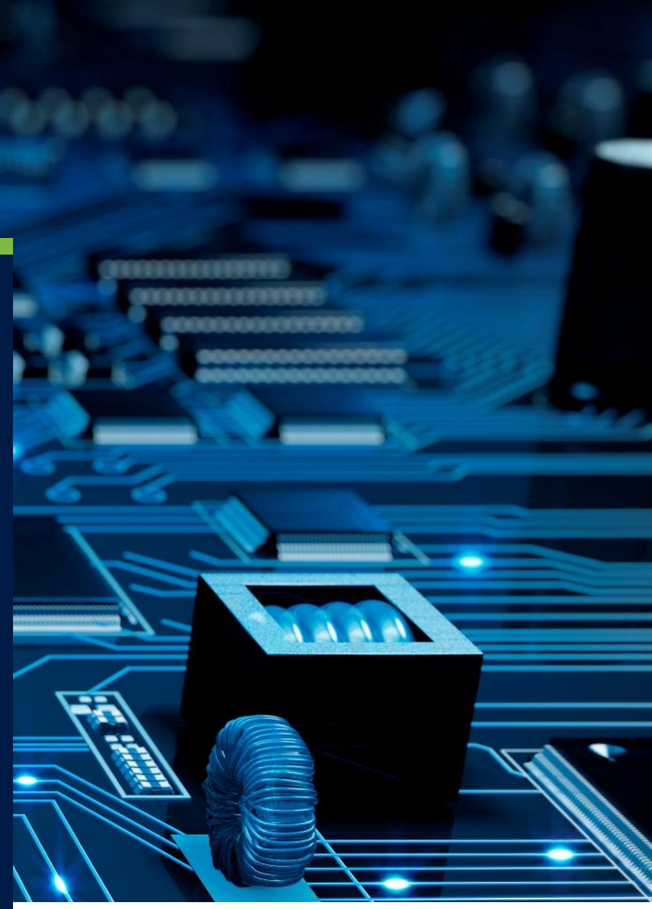
- Container Orchestration
- Emergent trends and landscape overview
- DevOps

Univerza v Ljubljani
Fakulteta za elektrotehniko

IKT Katedra za informacijske in komunikacijske tehnologije

LTFE Laboratorij za telekomunikacije

LMMFE Laboratorij za multimedijo



Avtorske pravice

Vsa gradiva usposabljanja so last Univerze v Ljubljani, Fakultete za elektrotehniko (UL FE).

- Uporaba je dovoljena izključno v okviru usposabljanja.
- Noben del gradiva ne sme biti reproduciran, shranjen ali prepisan v katerikoli obliki oziroma na katerikoli način (elektronsko, mehansko, s fotokopiranjem, snemanjem) brez predhodnega pisnega dovoljenja.

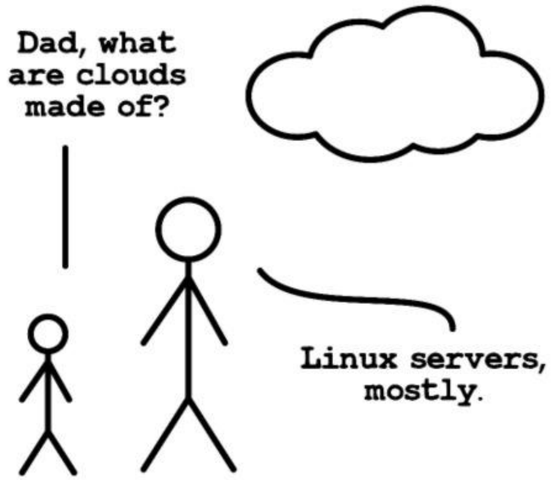
Univerza v Ljubljani
Fakulteta za elektrotehniko

Katedra za informacijske in komunikacijske tehnologije

Laboratorij za telekomunikacije

Laboratorij za multimedijo

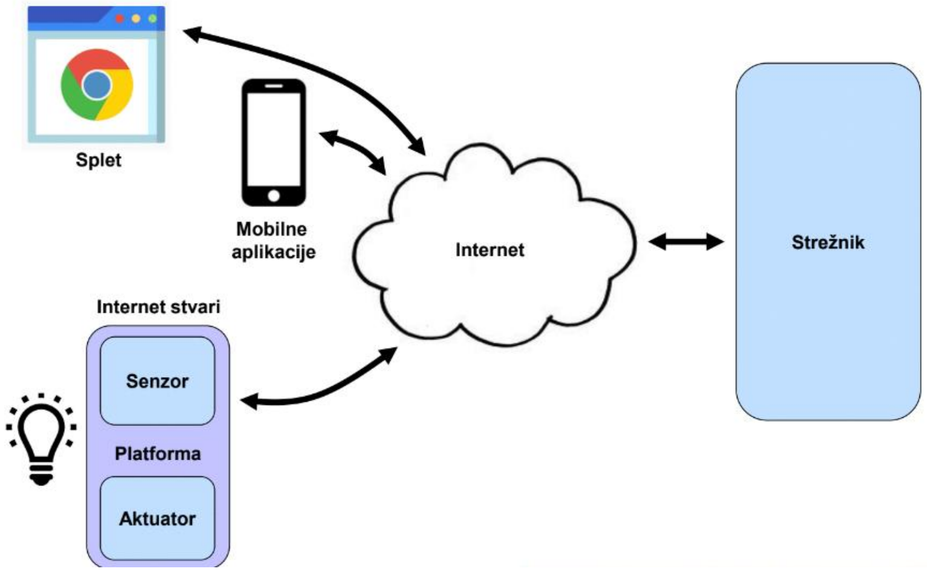
Introduction to Cloud Computing



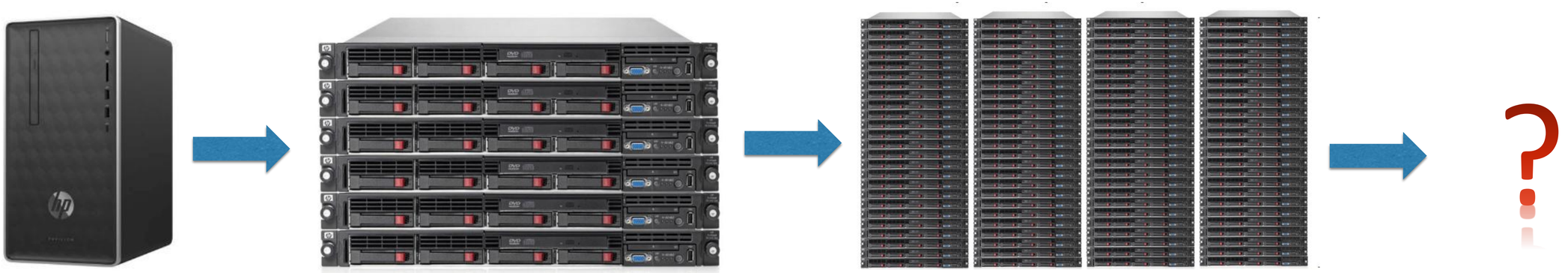
4

Scaling apps

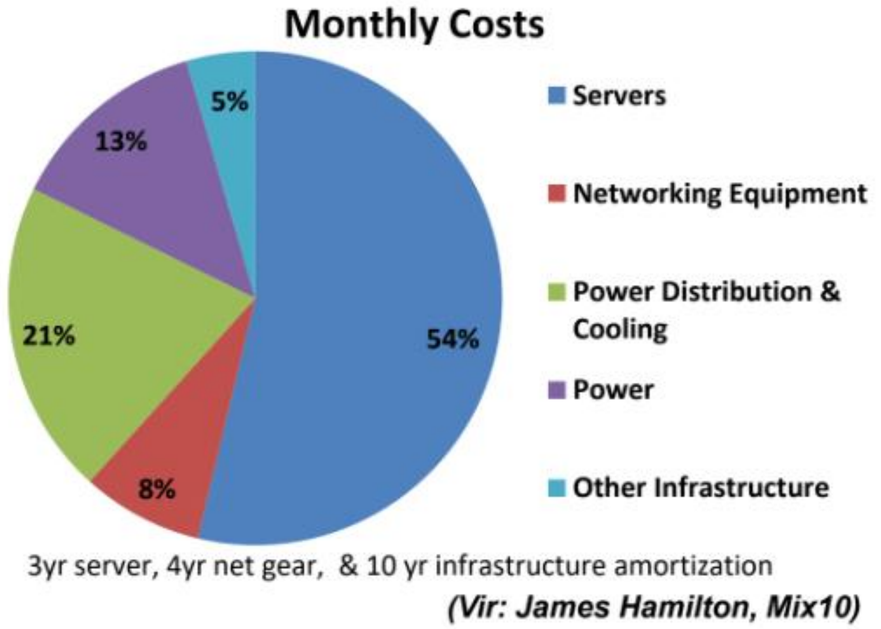
- Running a monolithic application usually requires a small number of powerful servers that can provide enough resources for running the application.
- To deal with increasing loads on the system:
  - **vertically** scale the servers (also known as scaling up) -> adding more CPUs, memory, and other server components
    - usually doesn't require any changes to the app
    - gets expensive relatively quickly
    - in practice always has an upper limit
  - scale the whole system **horizontally** (scaling out) -> setting up additional servers and running multiple copies of an application
    - relatively cheap hardware-wise
    - require big changes in the application code (sometimes impossible)



Servers scaling

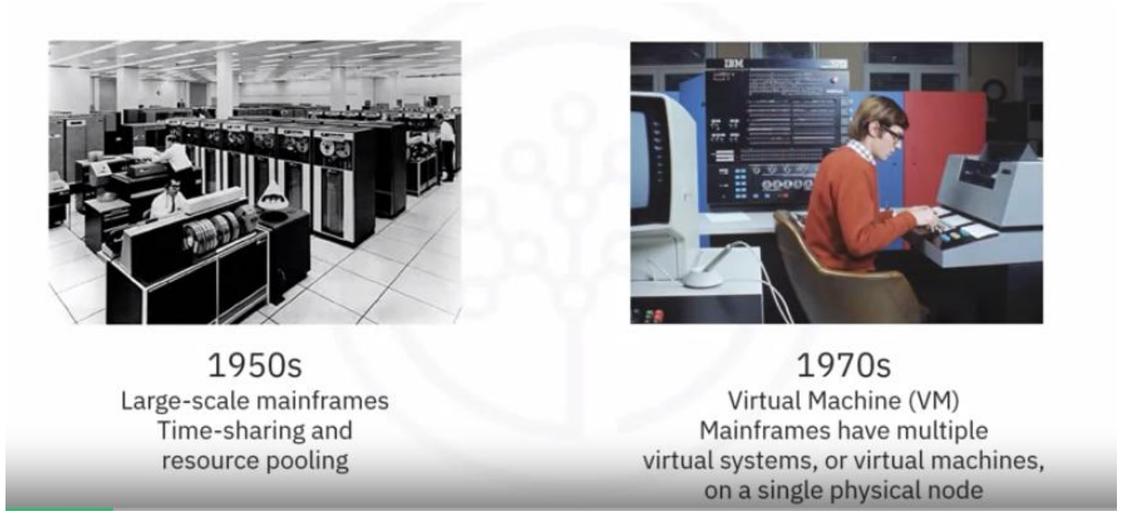


- Problems:
  - Managing physical hardware (power, cooling, networking, compatibility)
  - Administration (help of virtualization)
  - Price
  - Cloud computing tries to solve all of the above problems

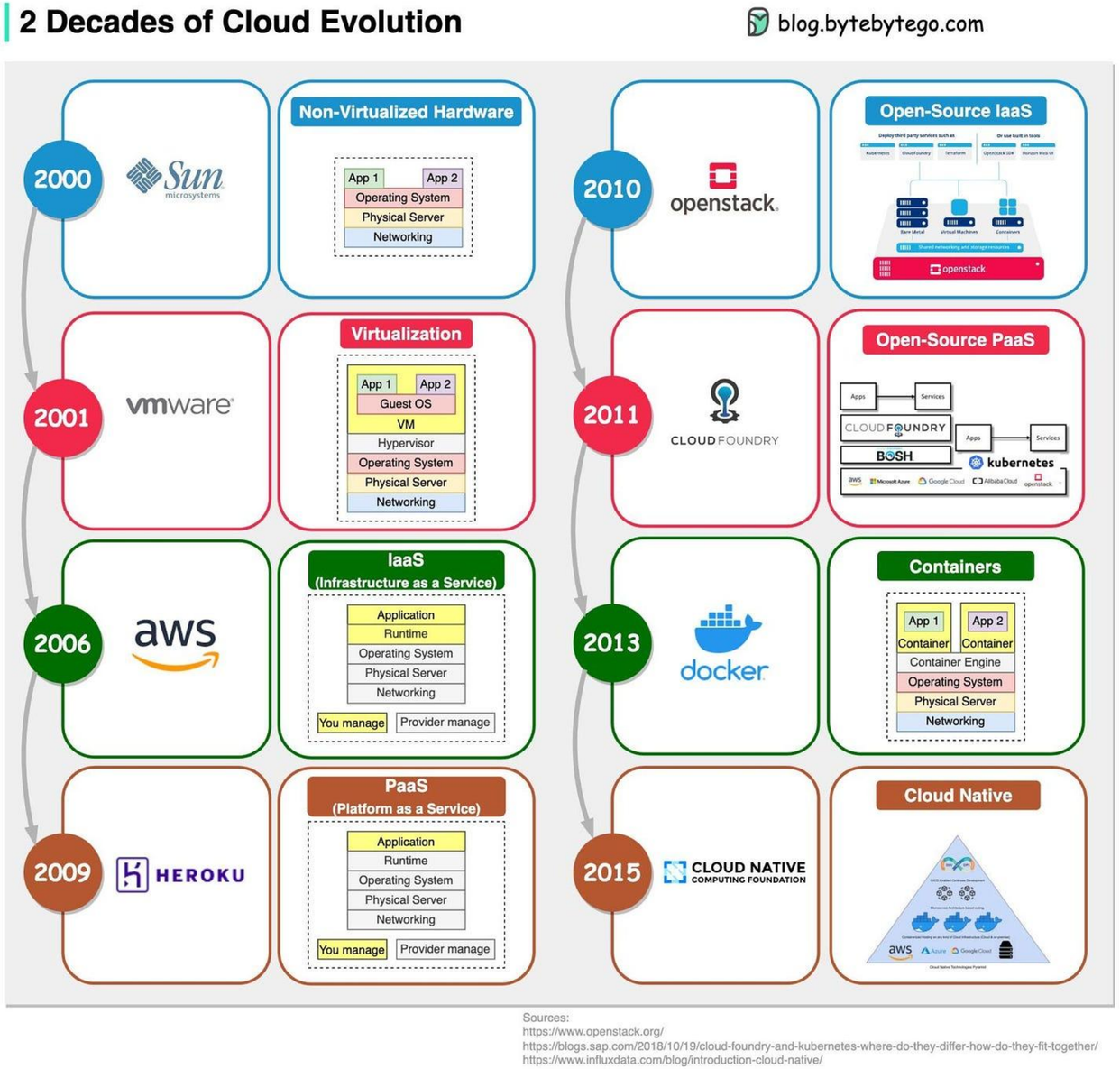


History and Evolution of Cloud Computing

- Cloud computing become mainstream and popular in the first decade of the 21st century:
  - Amazon’s Elastic Compute (EC2) and Simple Storage Service (S3) in 2006
  - Heroku in 2007
  - Google Cloud Platform in 2008
  - Alibaba Cloud in 2009,
  - Windows Azure (now Microsoft Azure) in 2010,
  - IBM’s SmartCloud in 2011
  - DigitalOcean in 2011



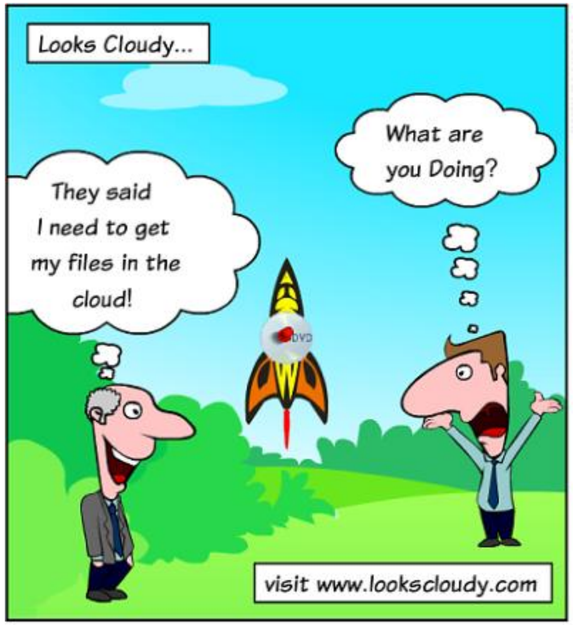
Source: https://www.coursera.org/learn/introduction-to-cloud/lecture/NbAsv/history-and-evolution-of-cloud-computing



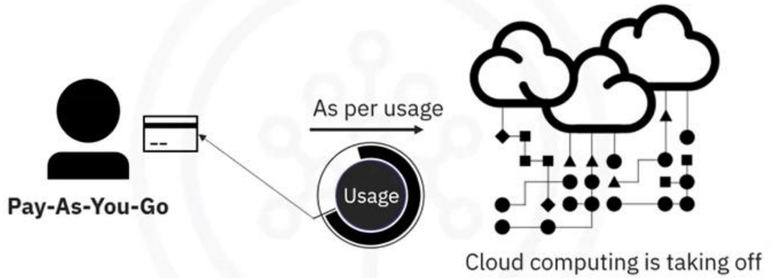
Source: https://blog.bytebytego.com/p/ep74-the-evolution-of-aws-services

What is Cloud Computing?

- Cloud Computing provides an alternative to the on-premises datacenter.
- Cloud computing is the delivery of computing services - including servers, storage, databases, networking, software, analytics, and intelligence - over the internet.
- Key **drivers for moving to cloud**:
  - Agility
  - Flexibility
  - Competitiveness



Characteristics

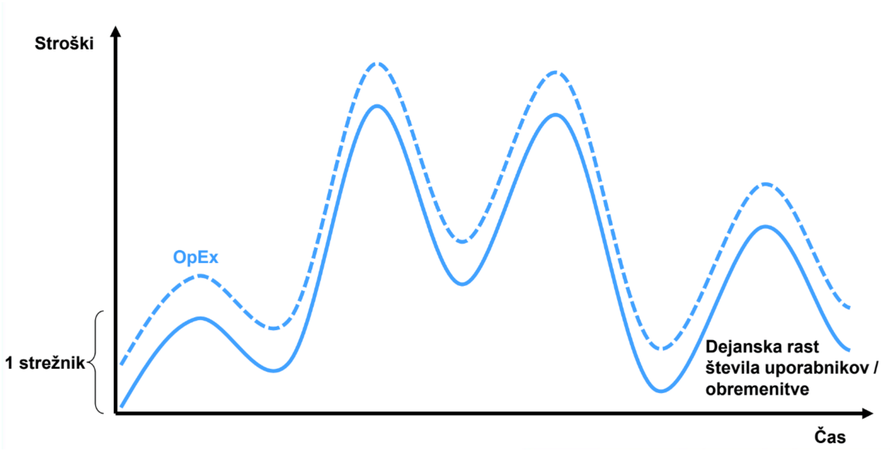


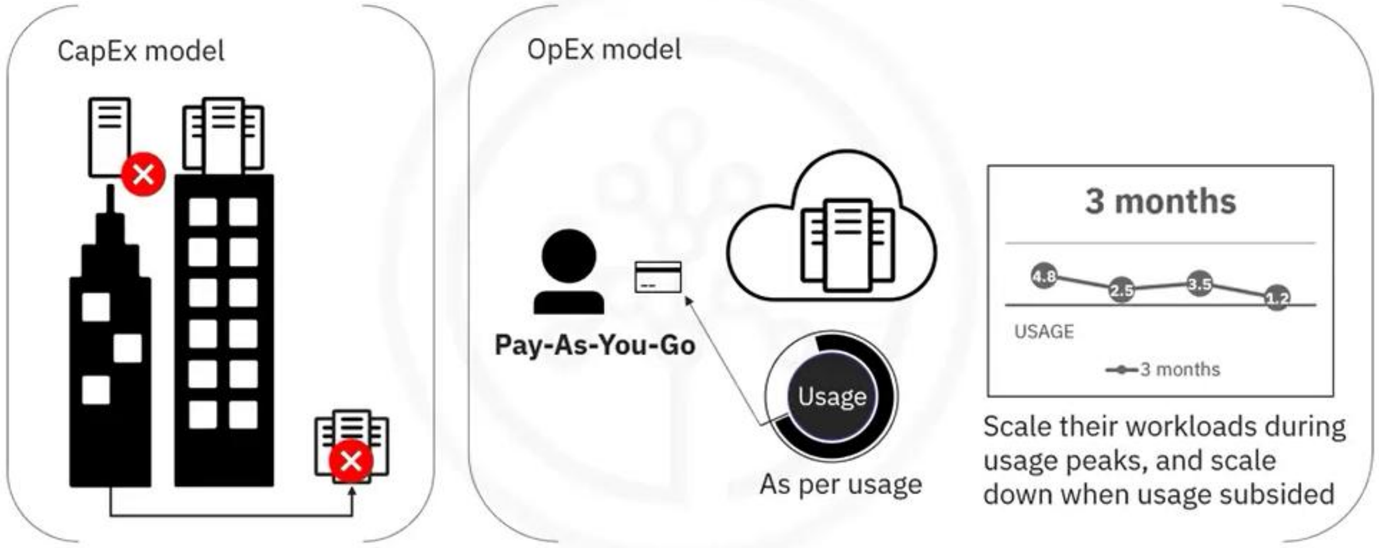
- **On-demand self-service**: access cloud resources whenever required
- **Broad Network Access:** Cloud computing resources needs to be accessed across the internet from a broad range of devices such as PCs, laptops, and mobile devices using standards-based APIs (for example, ones based on HTTP).
- **Shared Infrastructure:** Uses a virtualized software model, enabling the sharing of physical services, storage, and networking capabilities.
- **Dynamic Provisioning (Elasticity):** Allows for the provision of services based on current demand requirements. This is done automatically using software automation, enabling the expansion and contraction of service capability, as needed.
- **Managed Metering**: Uses metering for managing and optimizing the service and to provide reporting and billing information. In this way, consumers are billed for services according to how much they have actually used during the billing period.

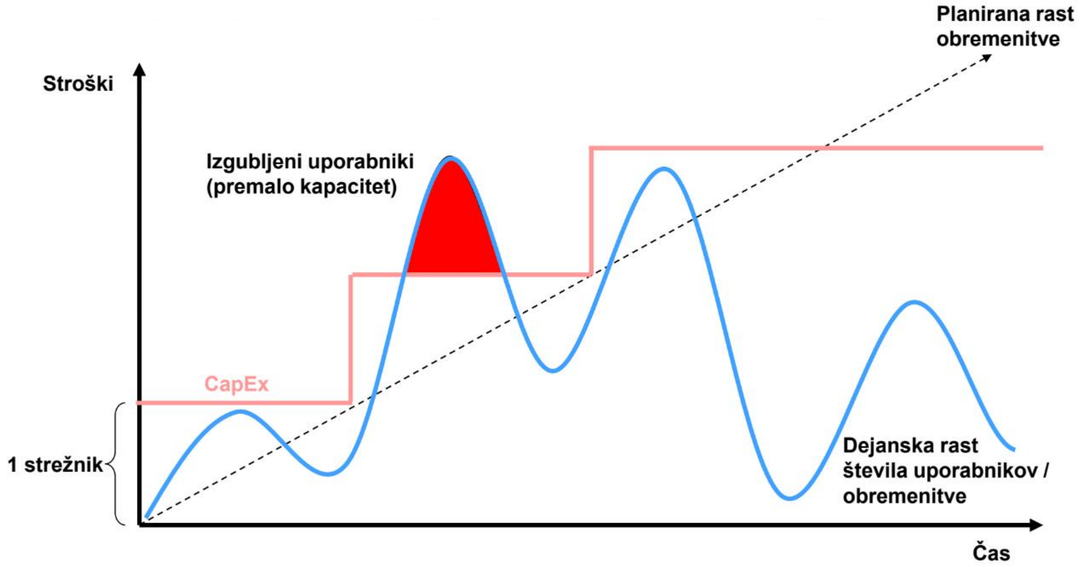
Cloud computing allows for the sharing and scalable deployment of services, as needed, from almost any location, and for which the customer can be billed based on actual usage.

Business case for Cloud Computing

- **Switch from CapEx to OpEx**:
  - Capital expenditures (CapEx) are funds used by a company to acquire or upgrade physical assets such as property, buildings, or equipment.
  - An operating expense (OpEx) is an expenditure that a business incurs because of performing its normal business operations.







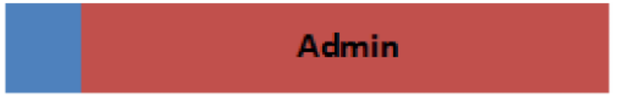
Source: https://www.coursera.org/learn/introduction-to-cloud/lecture/NbAsv/history-and-evolution-of-cloud-computing

Velik ponudnik [$13/Mb/s/mesec]: $0.04/GB

Srednji [$95/Mb/s/mesec]: $0.30/GB (7.1x)

Velik ponudnik : $4.6/GB/leto (2x in 2 Datacentra)

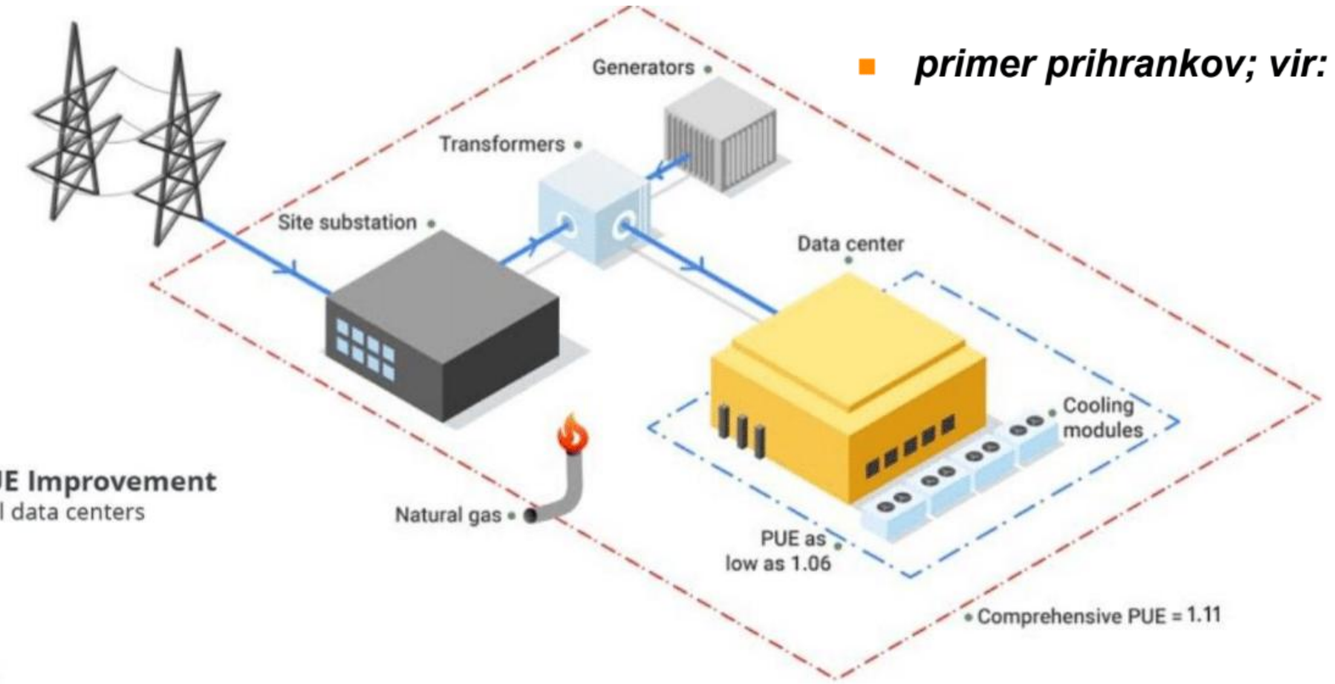
Srednji: $26.00/GB/leto (5.7x)

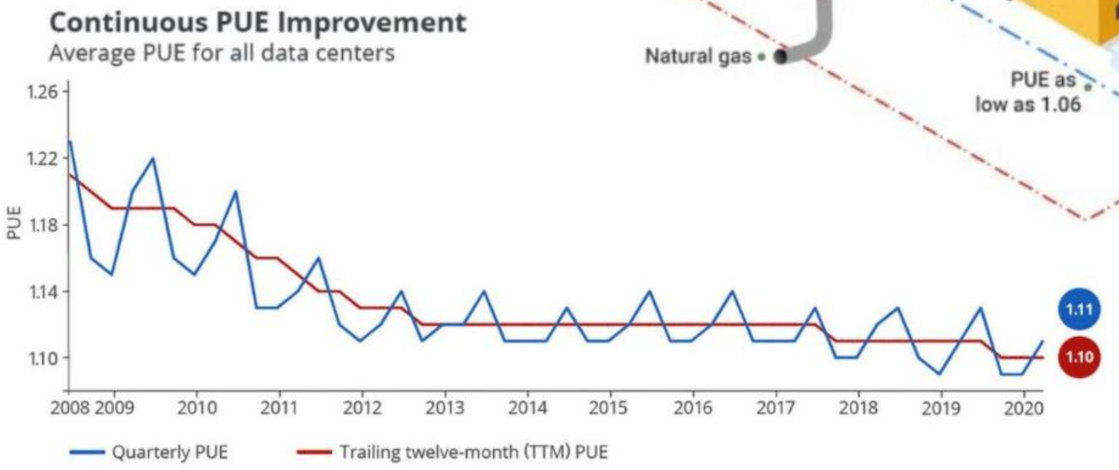


Velik ponudnik: preko 1.000 strežnikov/admina

Srednje podjetje: ~140 strežnikov/admina (7.1x)

- ***primer prihrankov; vir: James Hamilton, Mix10***





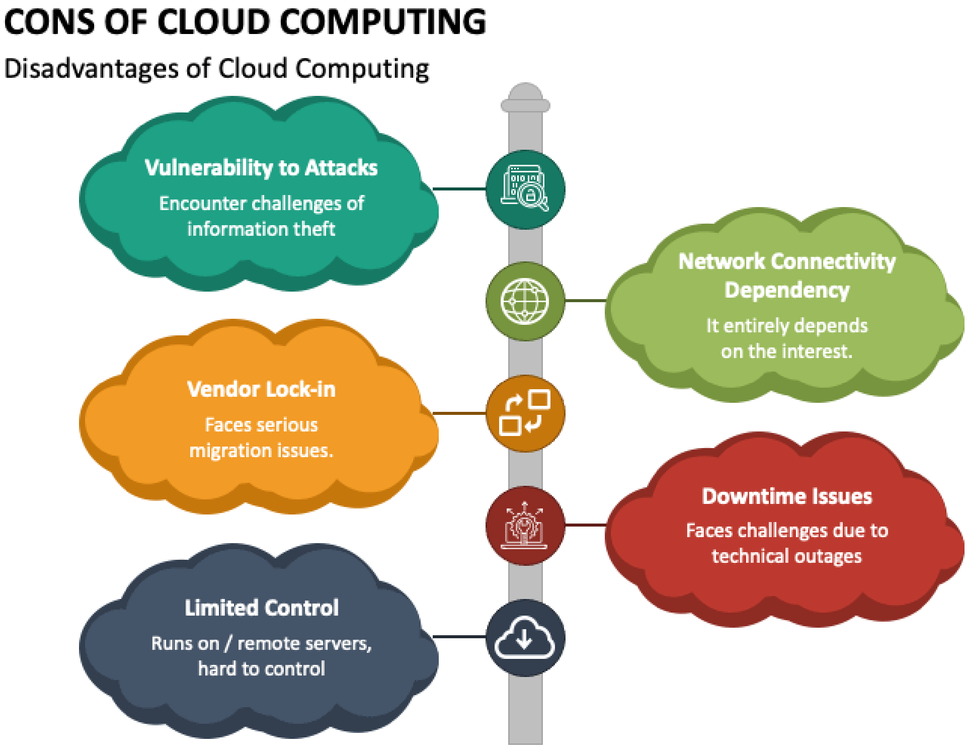
Vir: Google

Advantages of cloud computing

- **Speed**: Resources can be accessed in minutes, typically within a few clicks.
- **Scalability and flexibility (Rapid Elasticity)**: We can increase or decrease the requirement of resources according to the business requirements.
- **Productivity, efficiency (reduced deployment time)**: While using cloud computing, we put less operational effort. We do not need to apply patching, as well as no need to maintain hardware and software. So, in this way, the IT team can be more productive and focus on achieving business goals.
- **Reliability**: Backup and recovery of data are less expensive and very fast for business continuity.
- **Global Scale:** Deploy code anywhere and at any scale (“unlimited” capacity)
- **A business model with latest trends, strategic value:** Latest innovative technologies.
- **Security**: Many cloud vendors offer a broad set of policies, technologies, and controls that strengthen our data security. (!!! Also, a disadvantage - complexity)
- **Flexible Cost**: It reduces the huge capital costs of buying hardware and software – **low initial costs**. (!!! Also, a disadvantage)

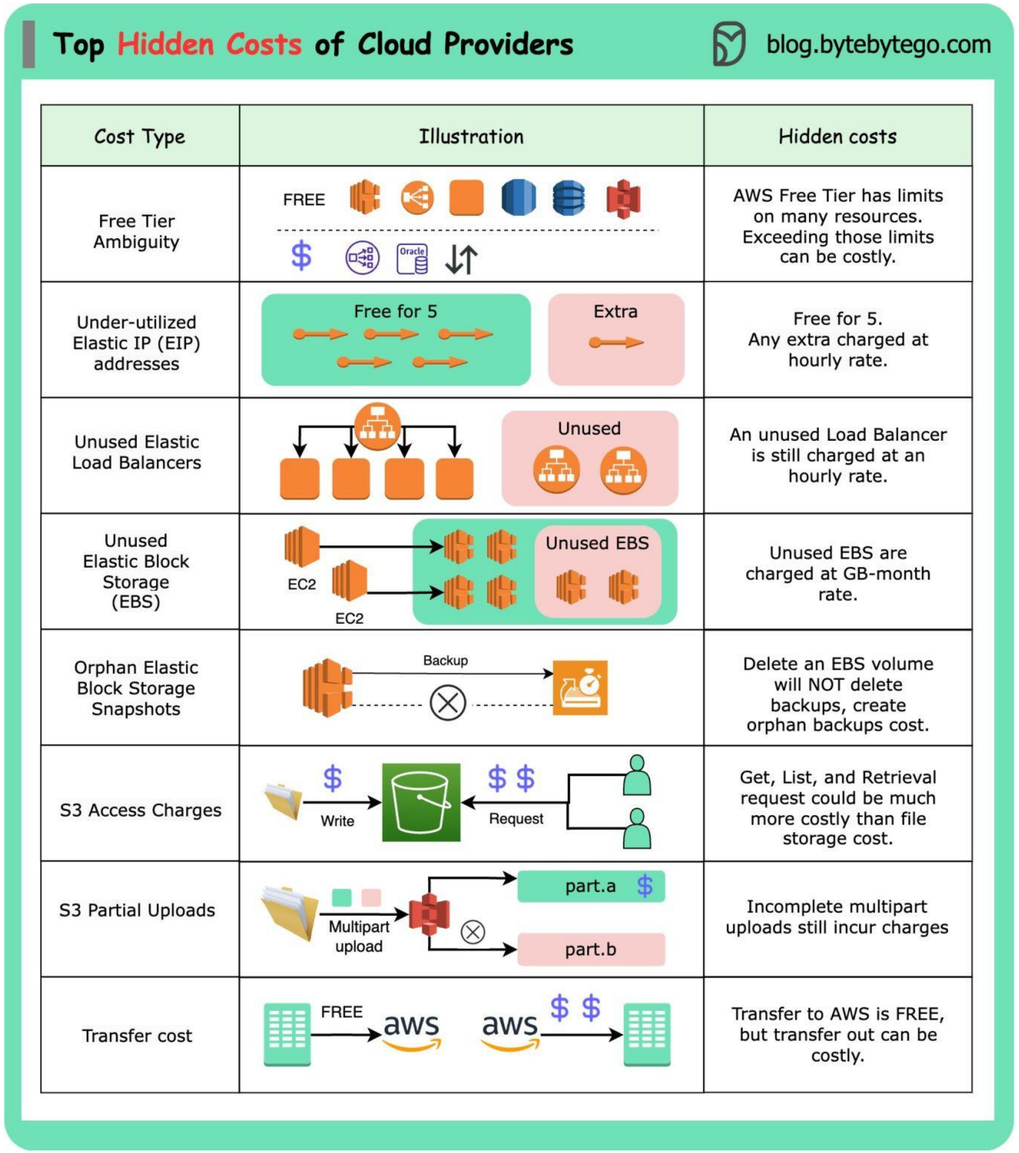
Disadvantages of cloud computing

- **Security**
- **Data loss (unavailability of data)**
- **Data persistence**
- **Costs (hidden costs)**
- **Vendor lock-in (Lack of Standards)**
- **Company use of data**
- **Company ethic**
- **Loss of user control and visibility (Limited Control)**
- **Regulation**
- **Complexity (Continuously Evolving)**



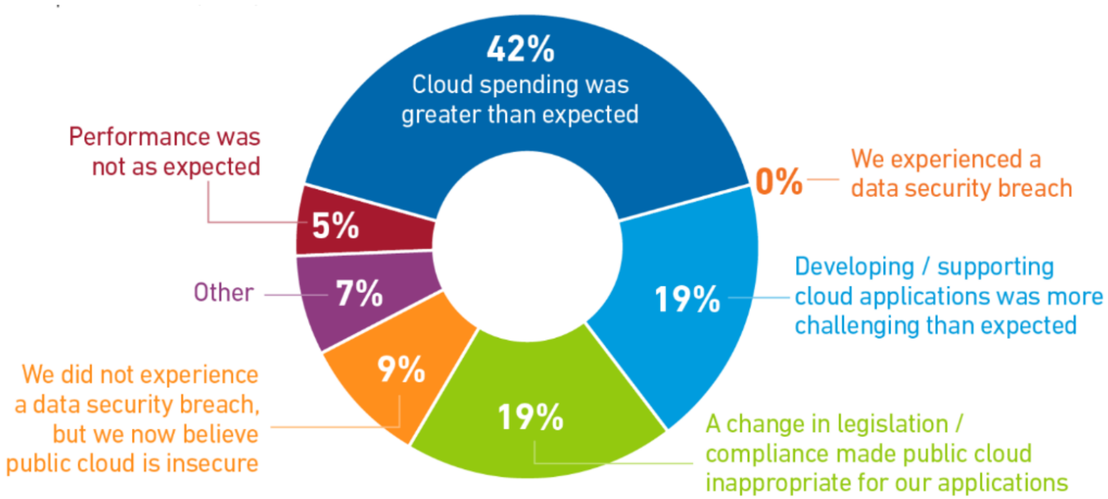
Hidden costs examples

- Free Tier Ambiguity
- Elastic IP Addresses
- Load Balancers
- Block Storage
- Snapshots
- Access Charges
- Data Transfer Costs



Cloud repatriation

- Cloud repatriation, also known as cloud exit, refers to the process of moving data, applications, and workloads from the cloud environment back to on-premises infrastructure or a different cloud provider.
  - **Dropbox** moved most of its infrastructure off the public cloud and onto its own custom-built infrastructure -> **gain more control and reduce costs**
  - GE, Hertz, Sony and CapitalOne (mostly large companies that have huge workloads)
- **Main reasons**: Costs, Data Control and Security, Performance and Latency, Compliance and Regulatory Requirements, Avoiding Vendor Lock-In



Source: https://journal.uptimeinstitute.com/high-costs-drive-cloud-repatriation-but-impact-is-overstated/

> Man I love this take. AWS is so so difficult to use properly. The dashboard is a maze, and a lot of very common use cases require you to deploy several (weirdly named) AWS products in harmony. The complexity was forgivable when cloud computing was new, 10+ years ago, but in the time since then AWS really hasn't gotten easier to use at all. Great job security for people who know how to do it properly.

Cloud Security Incidents

- Capital One (2019)
  - 30GB of credit application data affecting about 106 million people
  - misconfigured firewall that allowed an attacker to query internal metadata and gain credentials of an Amazon Web Services' IAM role
- Docker Hub (2019)
  - attackers managed to plant malicious images in the Docker hub
  - users unknowingly deployed cryptocurrency miners in the form of Docker containers that then diverted compute resources toward mining cryptocurrency for the attacker
- Half of Public Docker Hub Images Found to Have Critical Vulnerabilities

Source: Sophos, The state of Cloud Security 2020

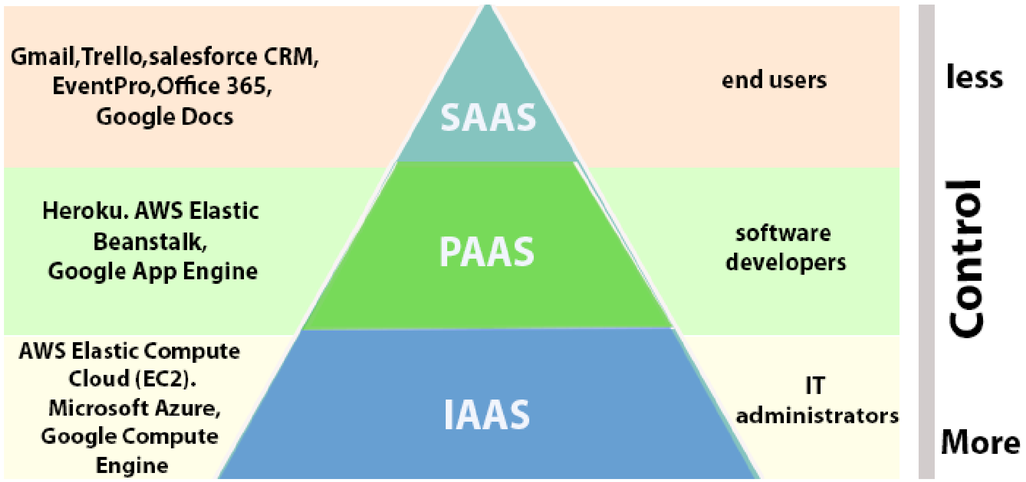
Source: https://www.thermofisher.com/blog/analyteguru/wp-content/uploads/sites/25/2021/10/091021-Image-4.jpg

- Microsoft Azure (2020)
  - a large-scale crypto mining attack against Kubernetes cluster in Azure
  - Targets were misconfigured Kubeflow containers -> turn them into crypto miners
- Tesla
  - hijacking resources has become a lot more lucrative than stealing info (cryptocurrencies prices)
  - cryptojacking when a Kubernetes cluster was compromised
  - an administrative console not being password protected
  - the misconfiguration had helped attackers get hold of Tesla's AWS S3 bucket credentials
  - stolen credentials were then used to run a crypto mining script on a pod
- Jenkins
  - cryptojacking, about $3.5 million, or 10,800 Monero in 18 months.

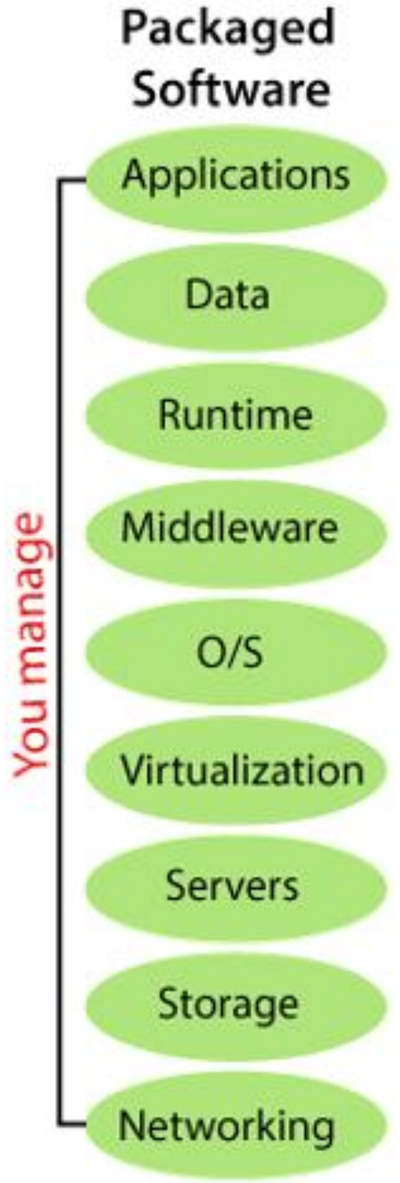
Source :https://img2.helpnetsecurity.com/posts2020/sophos-062020-1.jpg

Service Models

- Infrastructure as a Service (IaaS)
- Platform as a Service (PaaS)
- Software as a Service (SaaS)
- Function as a Service (FaaS)



Source: https://static.javatpoint.com/tutorial/microsoft-azure/images/introduction-to-cloud-computing-4.png

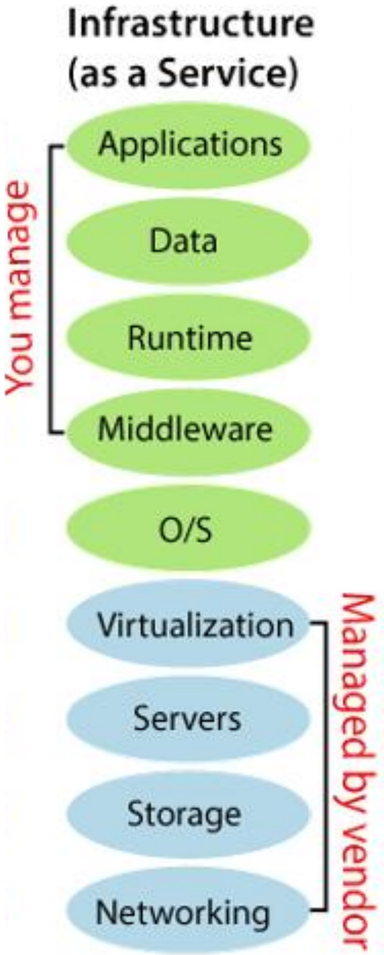


Source: https://static.javatpoint.com/tutorial/microsoft-azure/images/introduction-to-cloud-computing-5.png

19

Infrastructure as a Service (IaaS)

- Delivery of **computing infrastructure**, including operating systems, networking, storage, and other infrastructural components.
- No need to buy and maintain physical servers, easy scaling
- **Flexible** -> System administrators can oversee the installation, configuration, and management of operating systems, development tools, and other underlying infrastructure that they wish to use.



Amazon S3

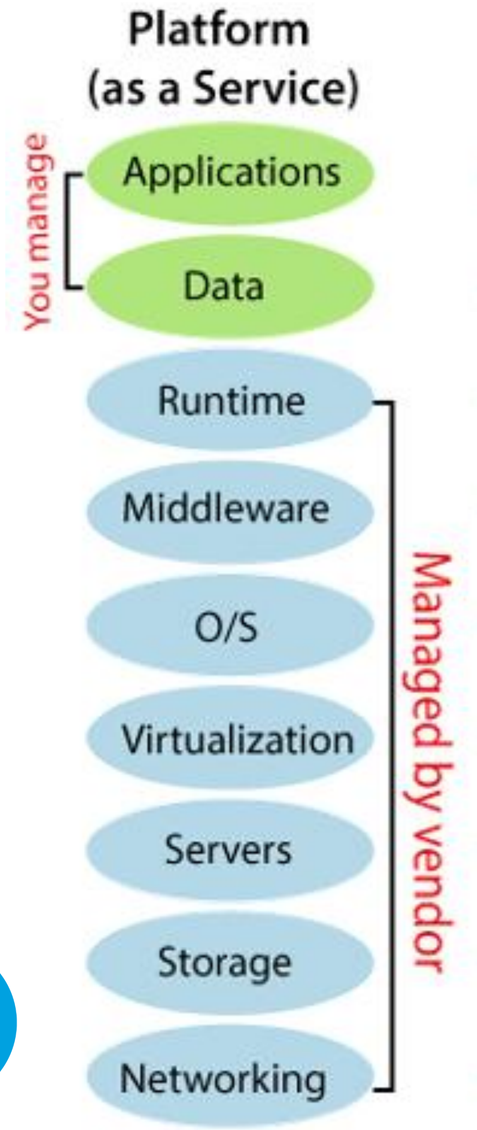
Amazon EC2

DigitalOcean

VM

Platform as a Service (PaaS)

- Underlying infrastructure (such as the operating system and other software) is installed, configured, and **maintained by the provider**.
- Used by software developers and developer teams (cuts down on the complexity of setting up and maintaining computer infrastructure)
- Focus their attention on development rather than DevOps and system administration.
- **ATTENTION: Total lock-in**



App Engine

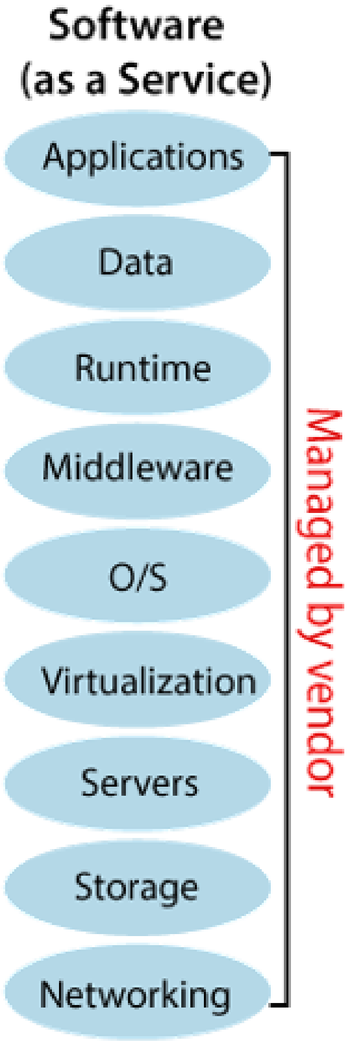
HEROKU

RED HAT® OPENSHIFT Container Platform

salesforce

Software as a Service (SaaS)

- SaaS providers are cloud-based applications that users access on demand from the internet **without needing to install or maintain the software.**
- SaaS applications are popular among businesses and general users.
- Accessible from any device, and have free, premium, and enterprise versions of their applications.
- Abstracts away the underlying infrastructure of the software application.



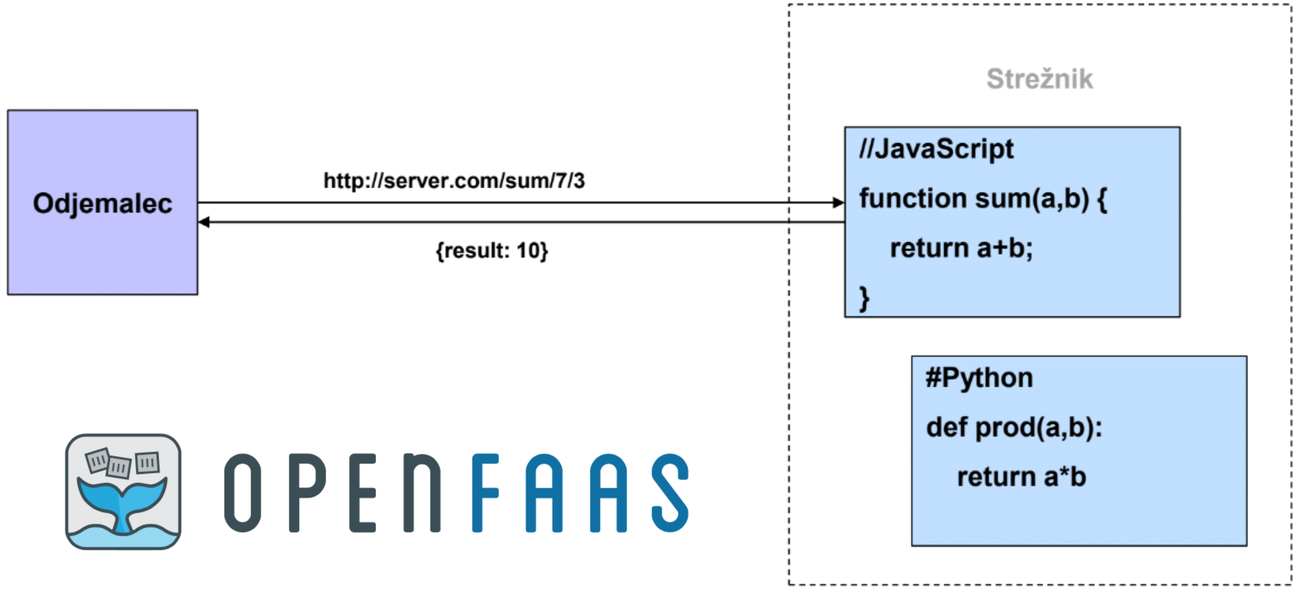
Google Docs

slack

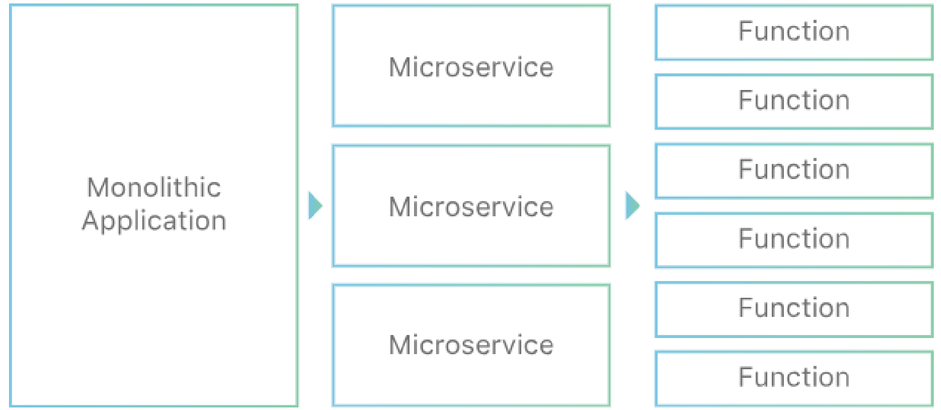
Adobe Creative Cloud

Function as a Service (FaaS)

- **Function-as-a-Service (FaaS) is a serverless way to execute modular pieces of code.**
- Examples: AWS lambda, Google Cloud Functions, Microsoft Azure Functions

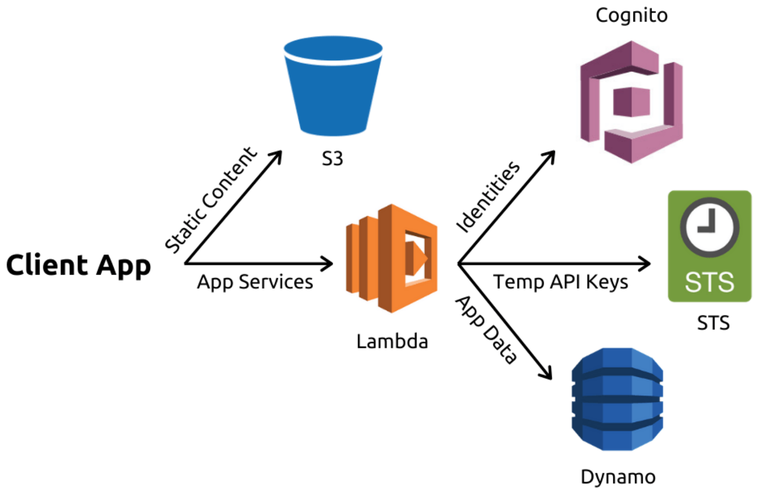


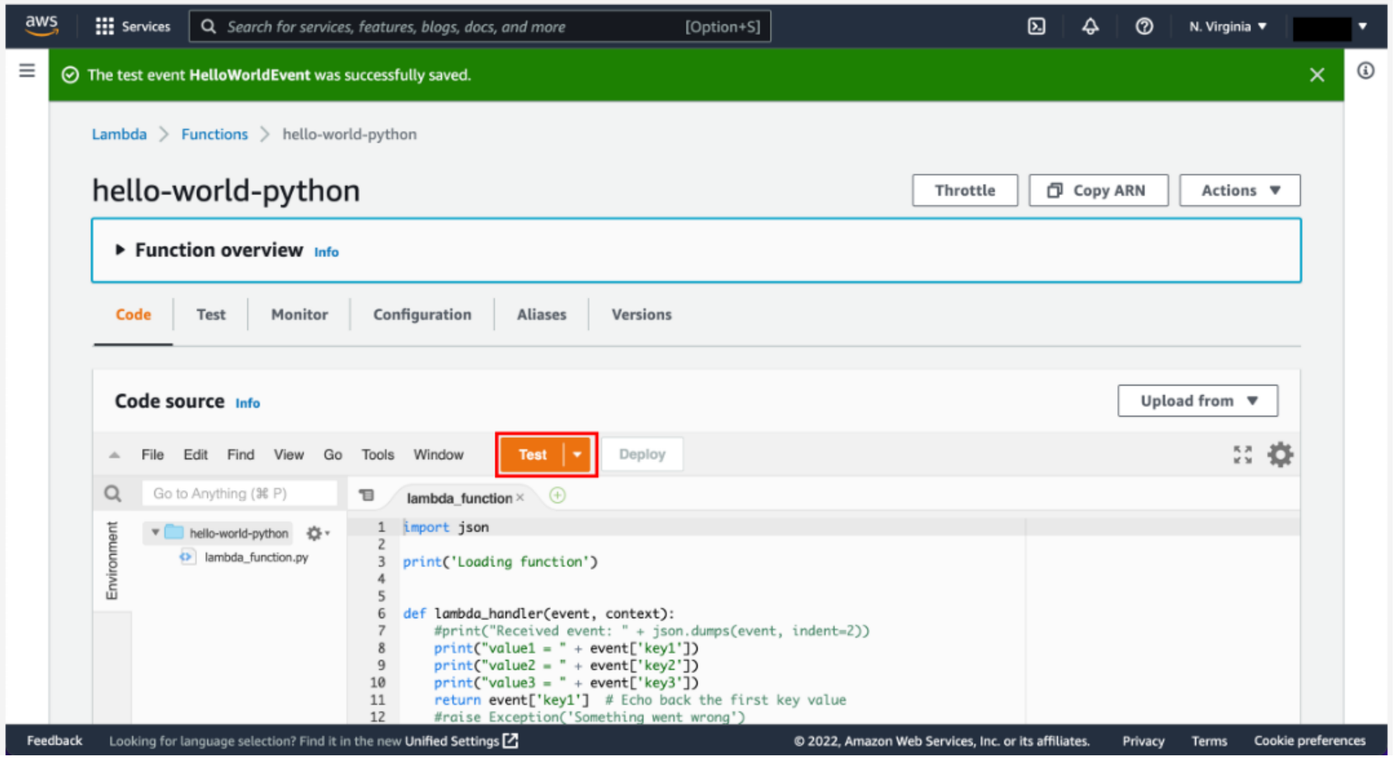
- FaaS lets developers write and **update a piece of code on the fly**, which can then be executed in response to an event, such as a user clicking on an element in a web application. -> **Stateless**



Source: https://www.cloudflare.com/en-gb/learning/serverless/glossary/function-as-a-service-faas/

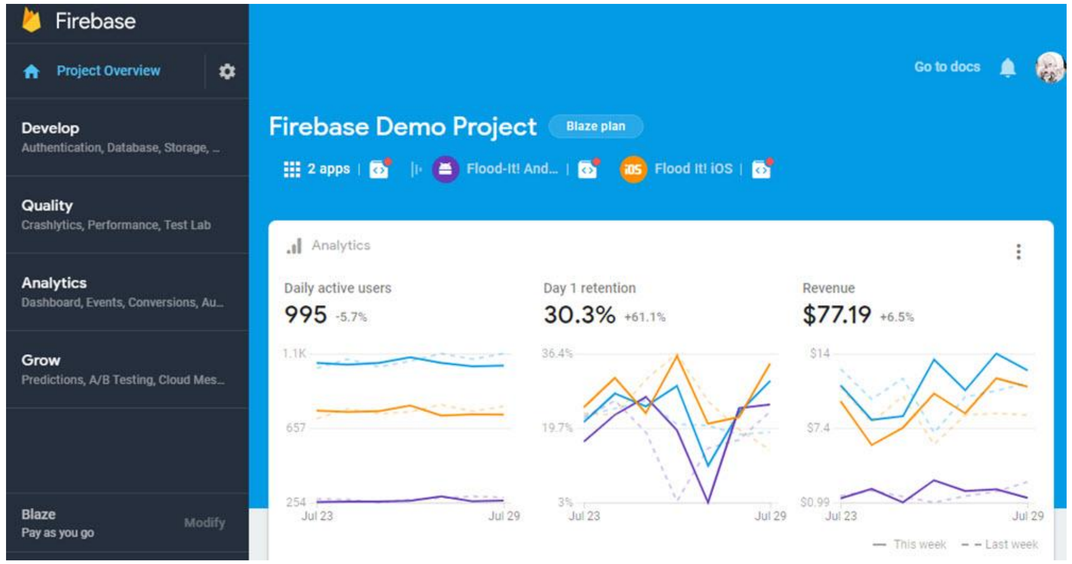
- **Advantages of using FaaS:**
  - Improved developer velocity (more time writing application logic)
  - Built-in scalability
  - Cost efficiency (providers do not charge their clients for idle computation time)
- **Drawbacks of FaaS:**
  - Less system control (tough to understand the whole system and adds debugging challenges)
  - More complexity required for testing (difficult to incorporate FaaS code into a local testing environment)

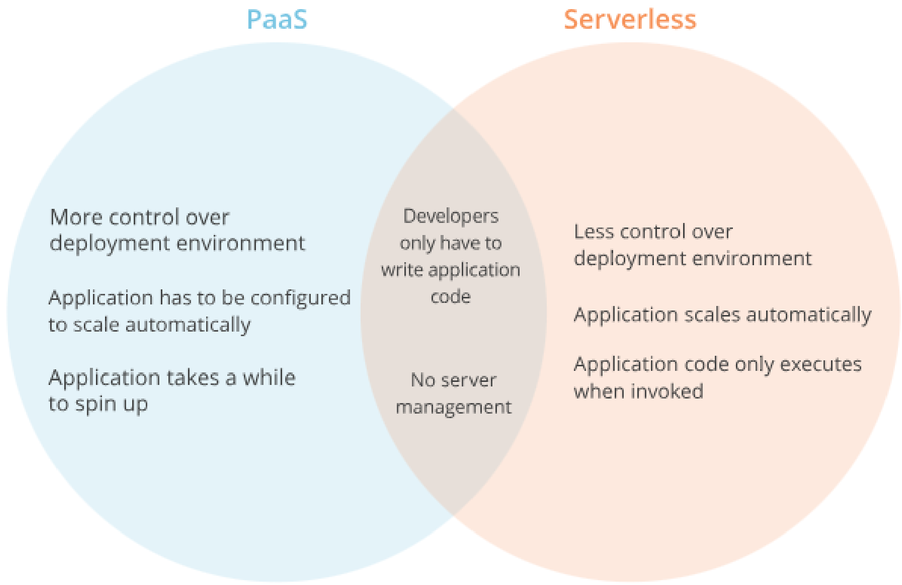




Serverless

- Serverless applications **scale instantly, automatically**, and on demand, without any extra configuration from the developer or the vendor.
- “Serverless” architectures have been built around services like Firebase, functions as a service

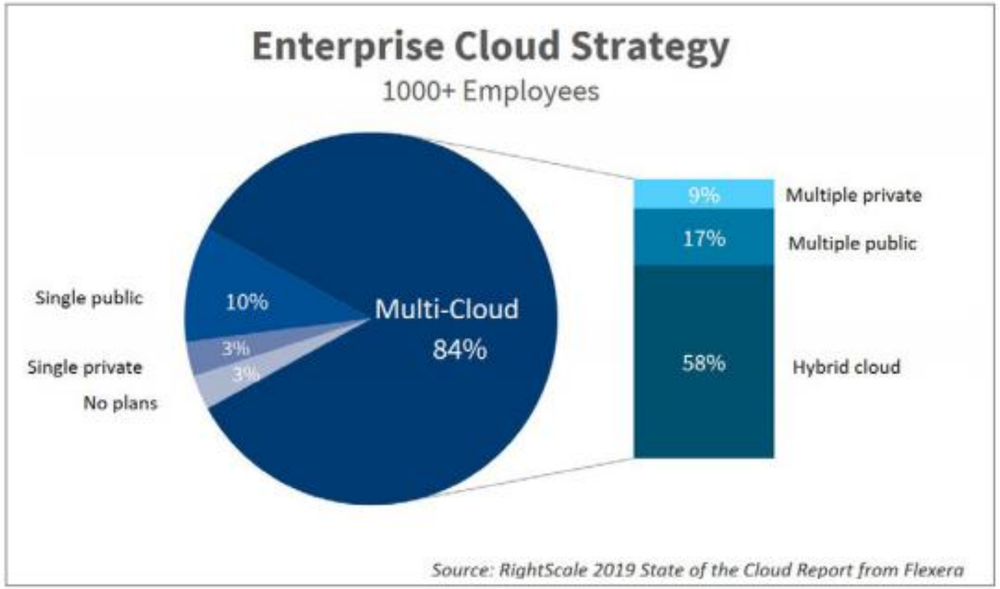


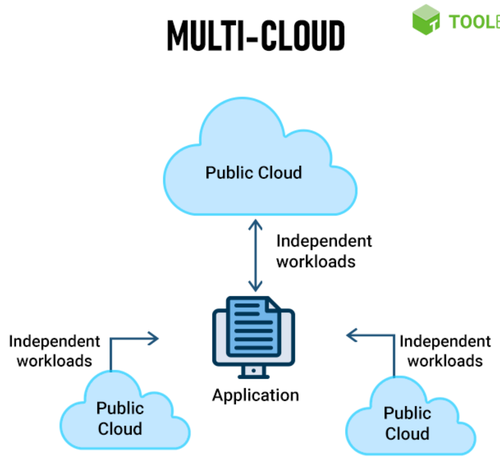


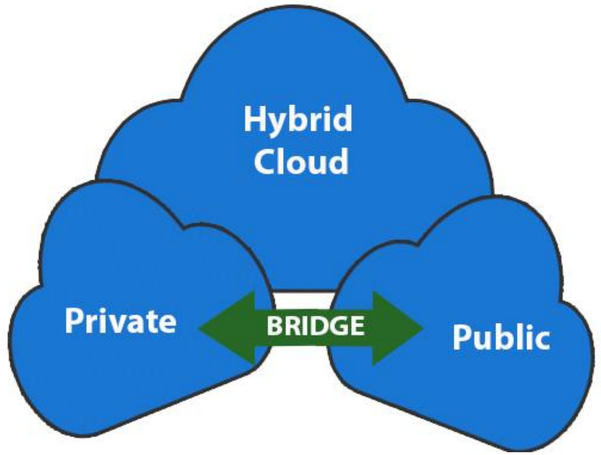
Source: https://www.cloudflare.com/en-gb/learning/serverless/glossary/serverless-vs-paas/

Deployment Models

- **Public Cloud**: The cloud resources that are owned and operated by a third-party cloud service provider are termed as public clouds. It delivers computing resources such as servers, software, and storage over the internet
- **Private Cloud**: The cloud computing resources that are exclusively used inside a single business or organization are termed as a private cloud. A private cloud may physically be located on the company's on-site datacentre or hosted by a third-party service provider.
- **Hybrid Cloud**: It is the combination of public and private clouds, which is bounded together by technology that allows data applications to be shared between them. Hybrid cloud provides flexibility and more deployment options to the business.
- **Multi Cloud**: Use of more than one public cloud provider



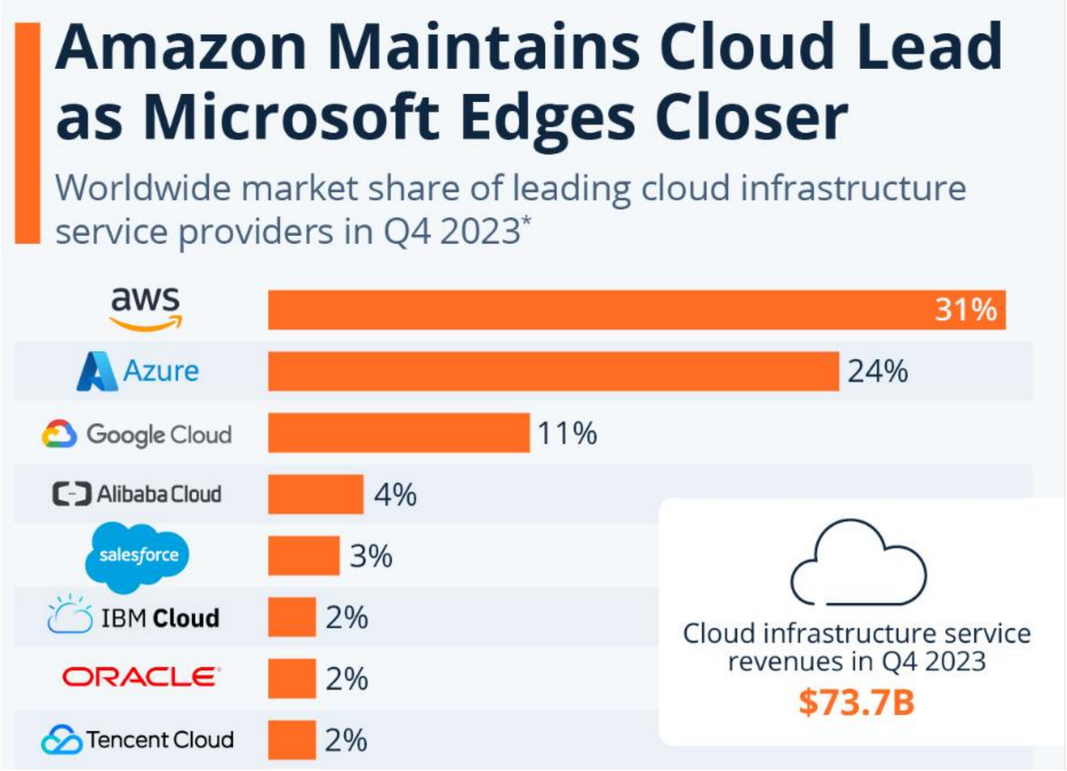




Source: https://static.javatpoint.com/tutorial/microsoft-azure/images/introduction-to-cloud-computing-3.png

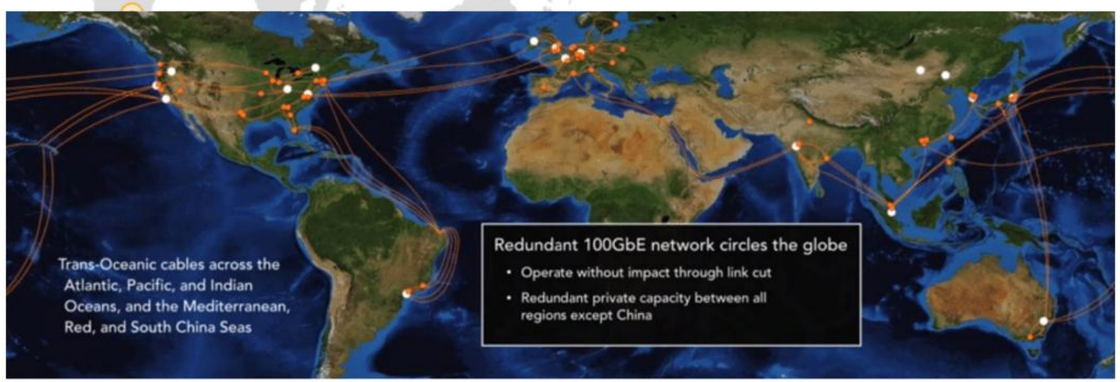
Key Cloud Service Providers

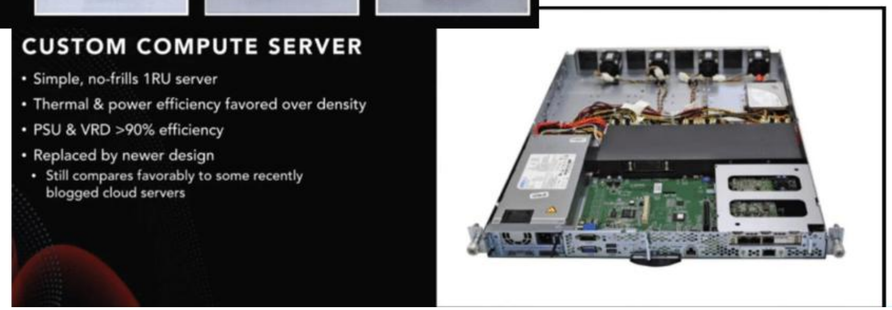
- https://getdeploying.com/ -> Compare cloud providers
- Amazon Web Services (AWS)
- Microsoft Azure
- Google Cloud
- Alibaba Cloud



Source: https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers/

The AWS Cloud spans 55 Availability Zones within 18 geographic Regions and 1 Local Region around the world



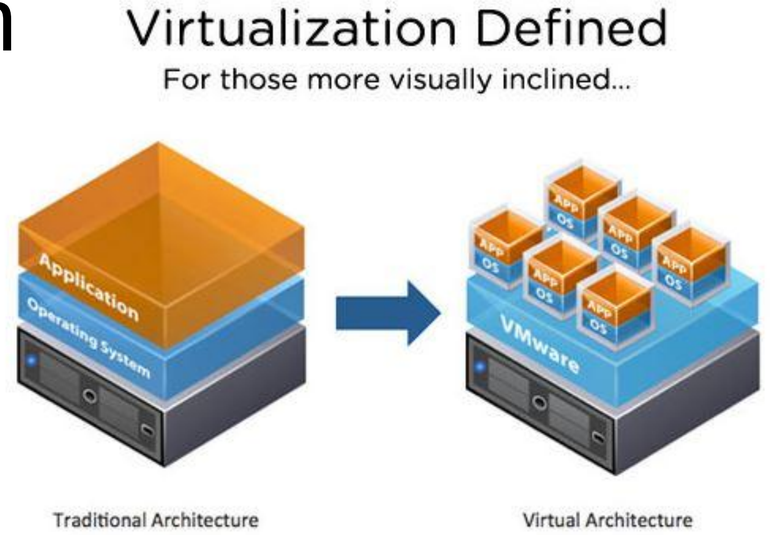


Virtualization and Containerization

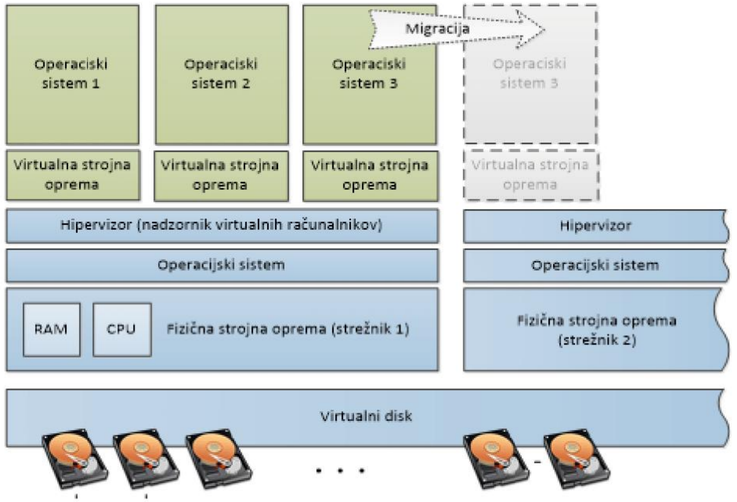
29

Resource virtualization

- Multiple resources combined in the pool
- User has no contact with the hardware
- Scale up -> Add more resources to the pool
- Hypervisor
- VM
- Easier migration

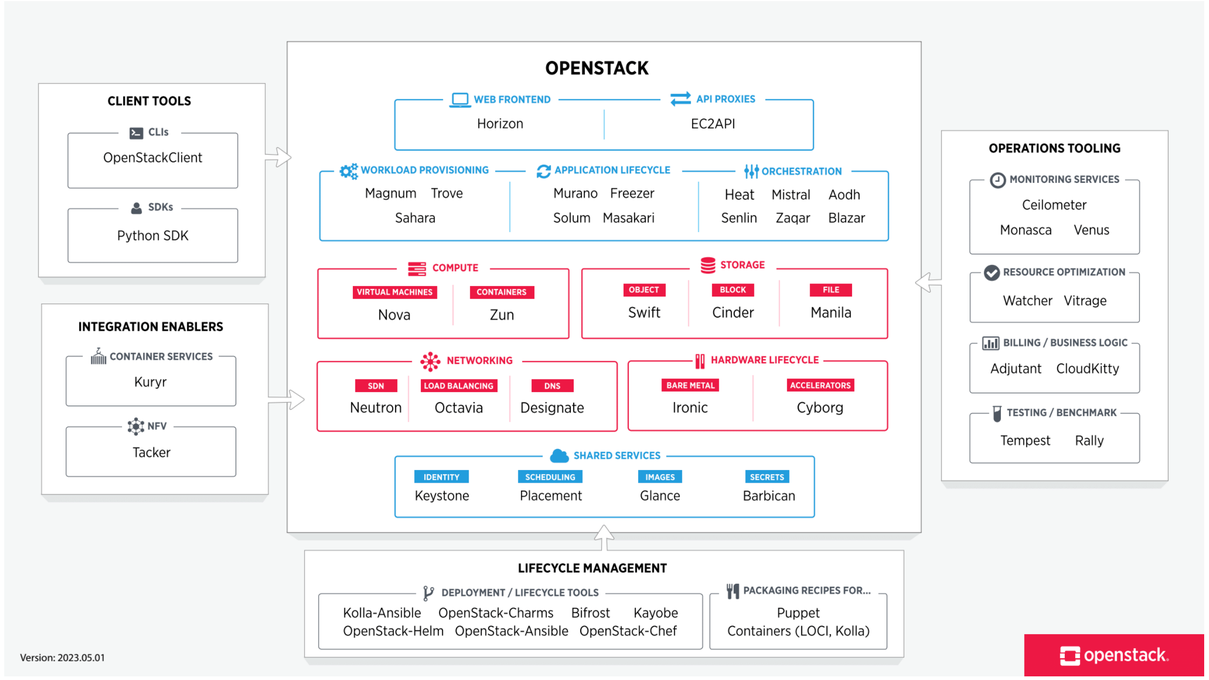


Source: https://viktec.files.wordpress.com/2014/09/virtulization.jpg



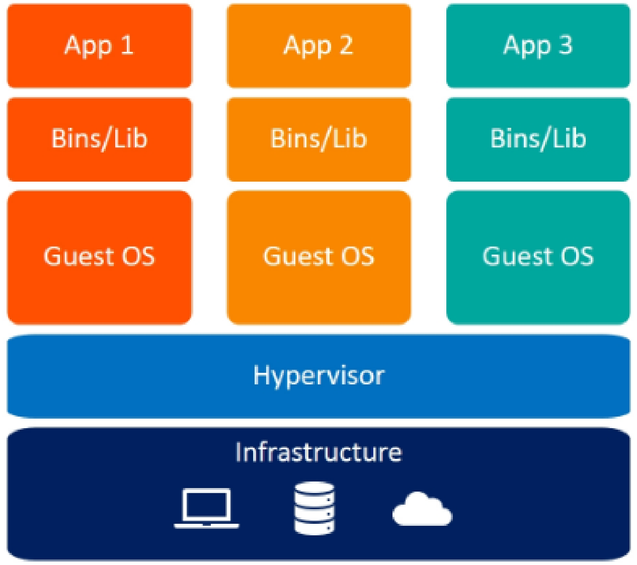
On-premises cloud projects

- The infrastructure that is housed at the client's business would be called on-premises cloud infrastructure.
- Managing on-premises environments involves complexities and costs related to skilled personnel, equipment upkeep, and ongoing maintenance.



Virtual Machines

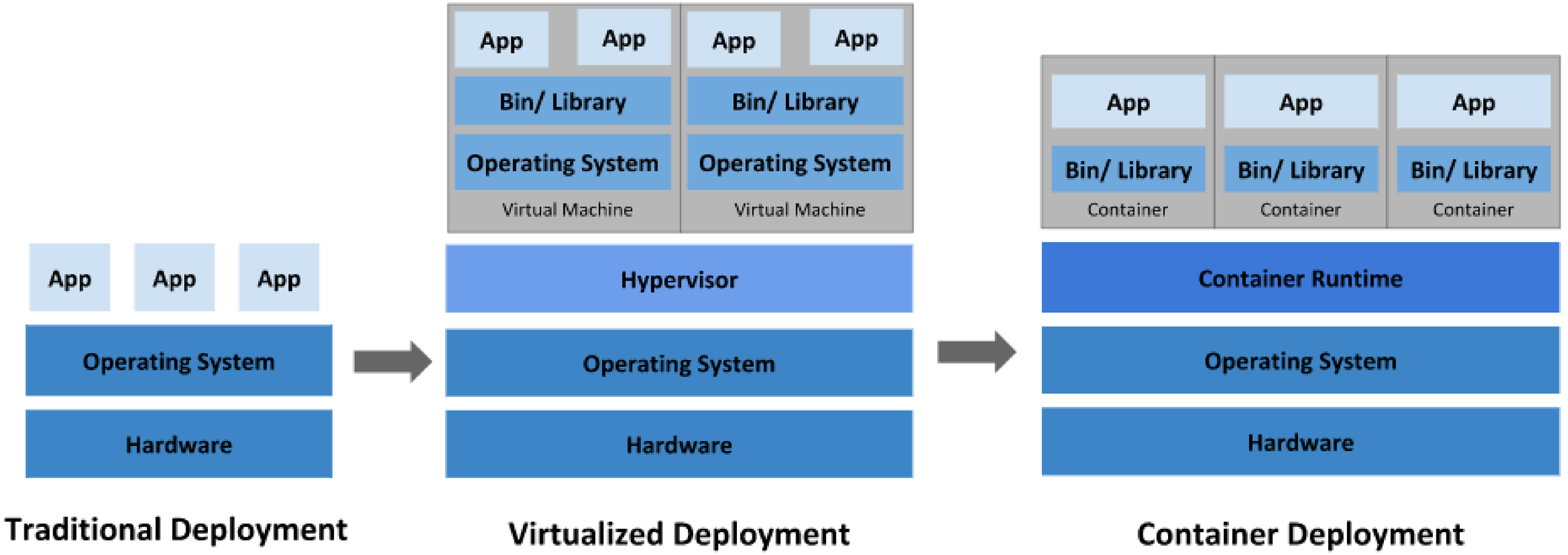
- VMware, Inc. gave the world a gift - the virtual machine (VM). IT departments no longer needed to procure a brand-new oversized server every time the business needed a new application.
- Problems:
  - Every VM requires its own dedicated operating system (OS)
  - Every OS consumes CPU, RAM and other resources
  - Every OS needs patching and monitoring.
  - In some cases, every OS requires a license.
  - VMs are slow to boot, and portability isn't great



Virtual Machines

Source: https://cloudzy.com/blog/virtual-machine-vm-what-why-when/

Virtualization History



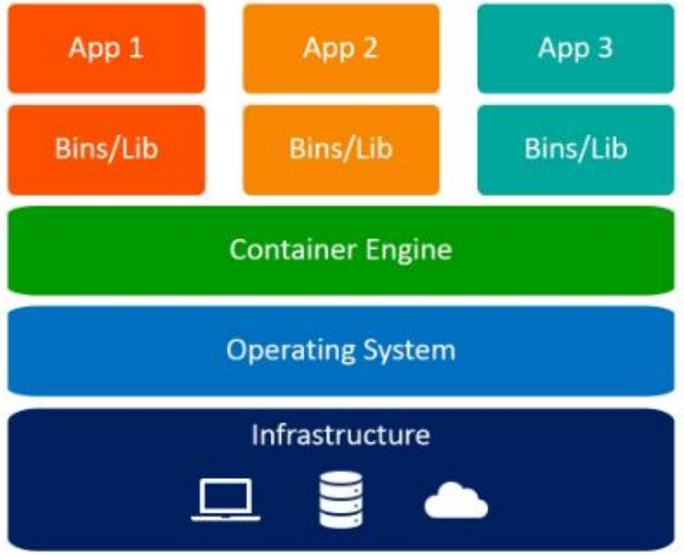
Source: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

Needs for containers

- Different software components require different environment requirements.
- Components start getting smaller, their numbers start to grow -> can’t give each of them their own VM
  - VM usually needs to be configured and managed individually
- Containerization is the next logical step in virtualization, and there is a huge buzz around this technology.
- Linux container technology provide isolation for running applications
- A process running in a container runs inside the host’s operating system
  - The process in the container is still isolated from other processes

Introduction to Containers

- Containers can provide virtualization at both the **operating system level** and the **application level**.
- Some of the possibilities with containers are as follows:
  - Provide a complete operating system environment that is sandboxed (isolated)
  - Allow packaging and isolation of applications with their entire runtime environment
  - Provide a portable and lightweight environment
  - Help to maximize resource utilization in data centers
  - Aid different development, test, and production deployment workflows



Containers

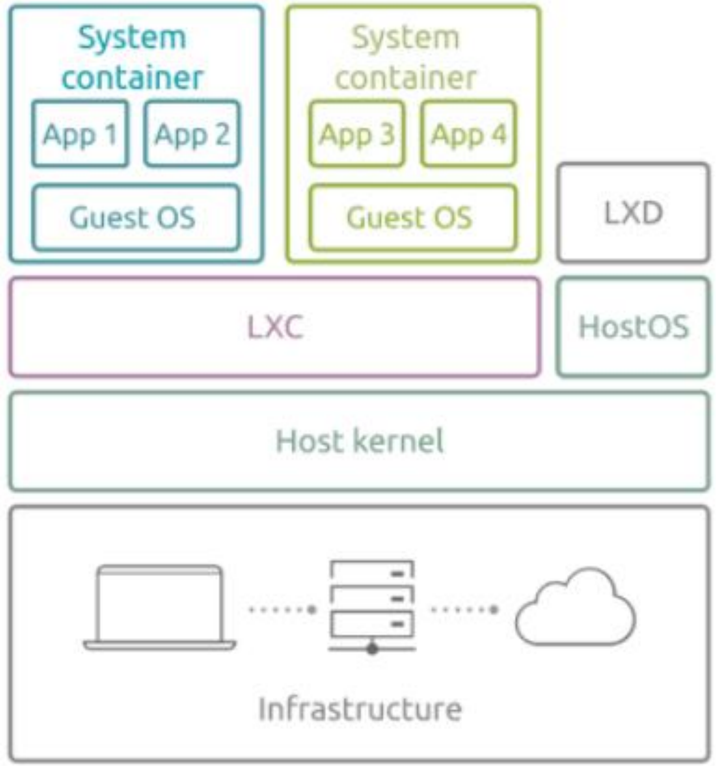
Source: https://cloudzy.com/blog/virtual-machine-vm-what-why-when/

Container Definition

- A container can be defined as a **single operating system image, bundling a set of isolated applications and their dependent resources** so that they run separated from the host machine.
- There may be multiple such containers running within the same host machine.
- Containers can be classified into two types:
  - **Operating system level**: An entire operating system runs in an isolated space within the host machine, sharing the same kernel as the host machine.
  - **Application level**: An application or service, and the minimal processes required by that application, runs in an isolated space within the host machine.

System containers

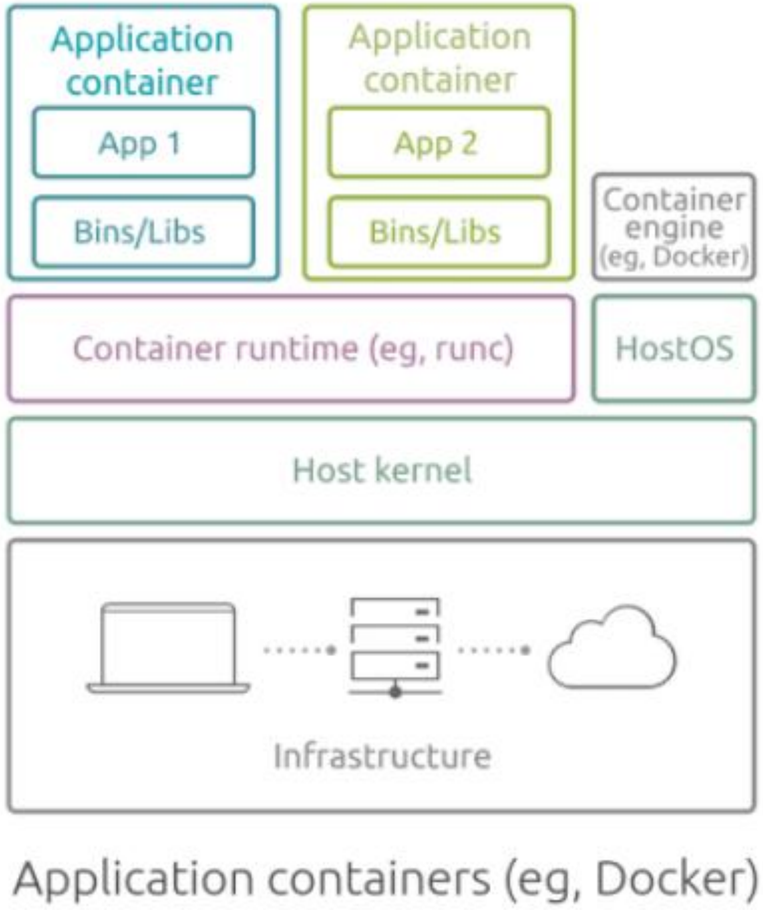
- Similar to virtual/physical machines.
- Capable of running a full operating system.
- Support for any type of workload, managed identically to virtual/physical machines.
- Typically long-lasting, akin to traditional machines (Installation of packages is possible, services within can be fully managed, backup policies can be defined, monitoring)
- Ability to host multiple applications within a single container.
- Equipped with all aspects of a virtual machine's management.
- System containers run a full operating system giving them flexibility for the workload types they support.
- Examples: LXC, LXD



System containers (eg, LXD)

Application Containers

- Application containers run a single app/process.
- Stateless, meant to be ephemeral (temporary).
- Minimal concern; containers can be easily created, deleted, and replaced.
- Examples: Docker containers
- Common Misconception: LXD is not an alternative to Docker or Kubernetes. LXD and Docker serve different purposes, not competing technologies.



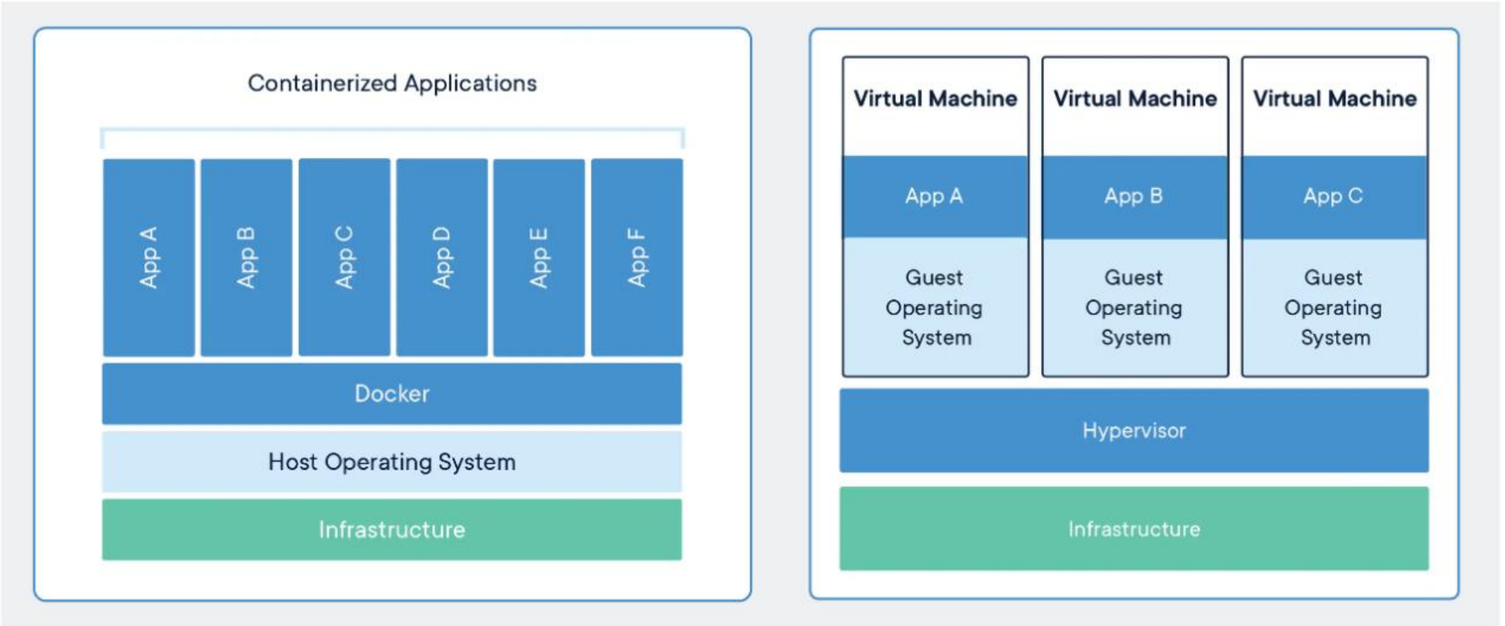
Application containers (eg, Docker)

Containerization vs traditional virtualization

- Containers are **lightweight** compared to traditional virtual machines.
- Starting a container happens nearly instantly compared to the slower boot process of virtual machines.
- Containers are portable.

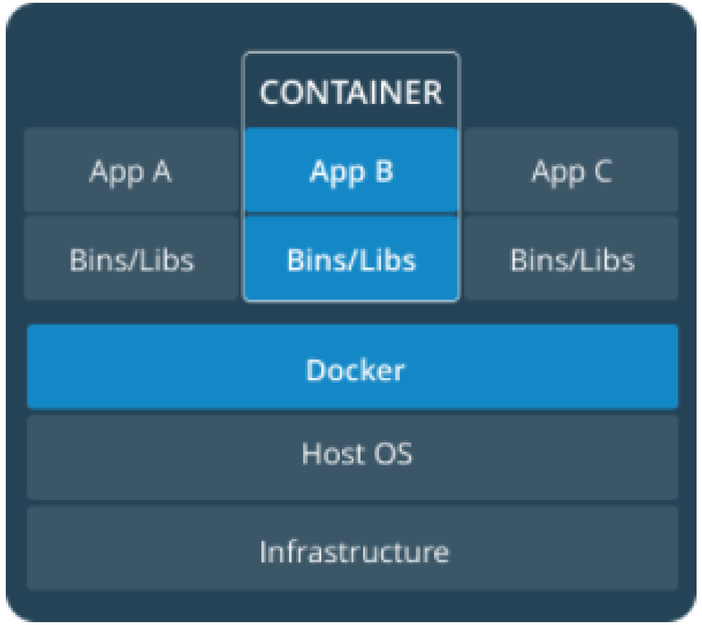
Feature	Container	VM
Defination	OS virtualization.	Server Virtualization.
Operating System	Containers shares Host OS	Each VM has its own OS
Size	Megabytes	Gigabytes
Time to start	Seconds	Minutes
Kernel	Shared with host	Each VM has its own Kernel
Density of Application workloads	Containers can run 2-3 times more applications.	
Benefits	Containers share a common OS which reduces mgmt. overhead. Containers are light weight and are more portable than VMs and can be deployed across public- private cloud and Traditional Data Centres.	Consolidates Multiple applications on a single system reducing server footprint.
Drawbacks	Higher Fault Domain and are less secure than VMs.	VMs are not portable across Public-Private Clouds.

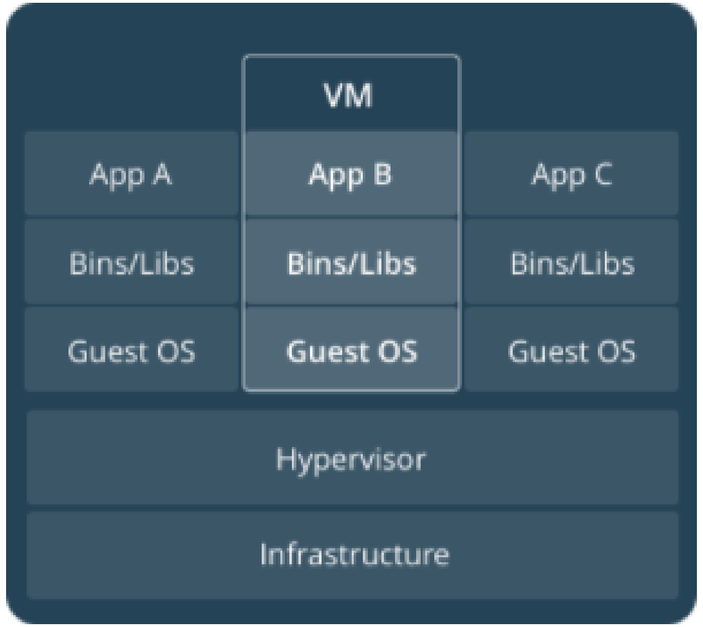
Source: https://mykloud.wordpress.com/2018/06/06/containers-deep-dive/



Source: https://www.docker.com/resources/what-container/

- Containers are lightweight because they don’t need the extra load of a hypervisor, but run directly within the host machine’s kernel.
- A container runs natively on Linux and shares the kernel of the host machine with other containers.
- It runs a discrete process, taking no more memory than any other executable, making it lightweight.

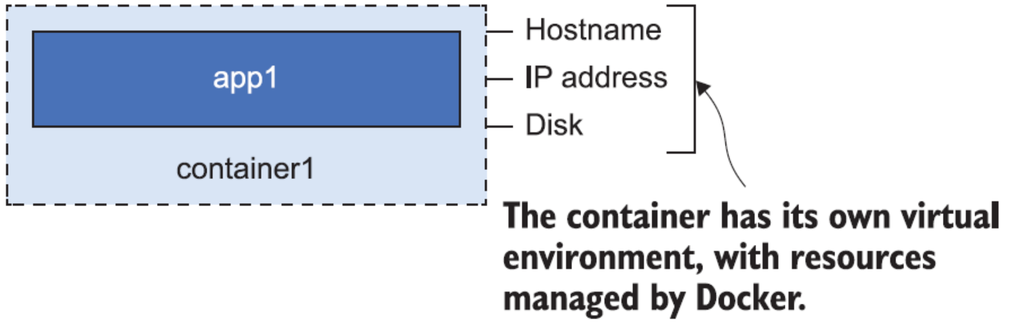


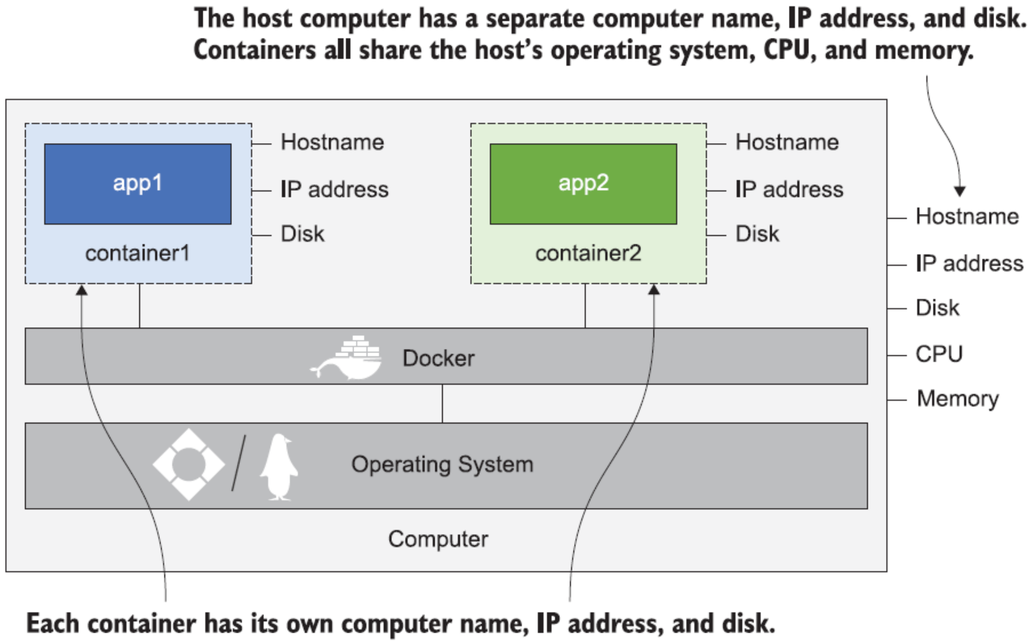


Source: https://docs.docker.com/get-started/

Docker containers

- Container has its **own machine name** and **IP address, disk drive.**
- Virtual resources created by Docker
- The application inside the box can’t see anything outside the box.
- All **share the CPU and memory** of the computer, and they all share the computer’s operating system.
- It fixes two conflicting problems in computing: Density, Isolation

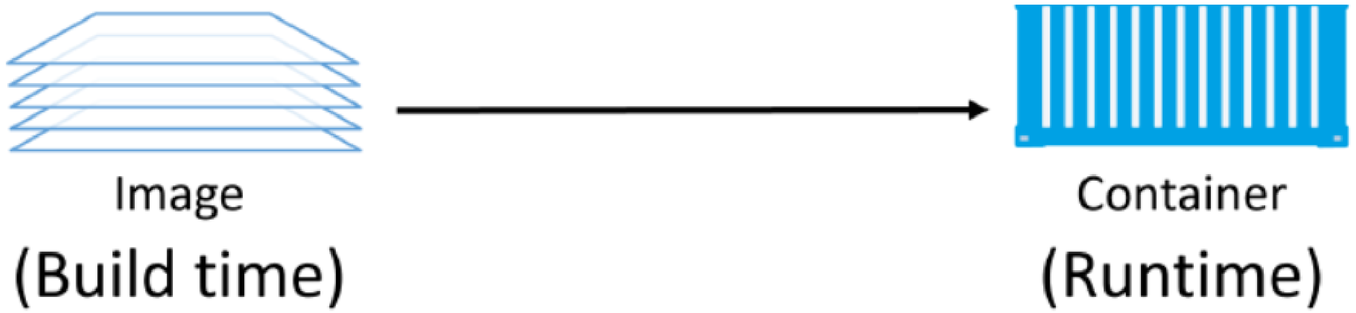




Container Images

- A image is a unit of packaging that **contains everything required for an application to run**.
  - application code
  - application dependencies
  - OS constructs.
- You get images by **pulling them from an image registry**.
- Images are made up of **multiple layers**, images **tend to be small**

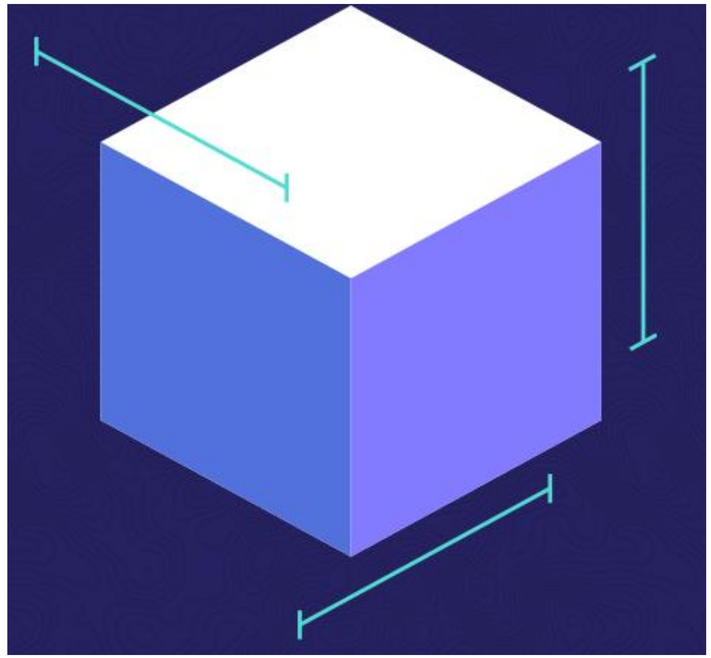
> You can think of Docker images as similar to VM templates. A VM template is like a stopped VM — a Docker image is like a stopped container.



Source: Docker Deep Dive, Nigel Poulton

OCI Image

- History: No standard, different implementations and capabilities
- To make sure that all container runtimes could run images produced by any build tool, the community started the Open Container Initiative to define industry standards around container image formats and runtimes.
- The Open Container Initiative is an open governance structure for the express purpose of creating open industry standards around container formats and runtimes.

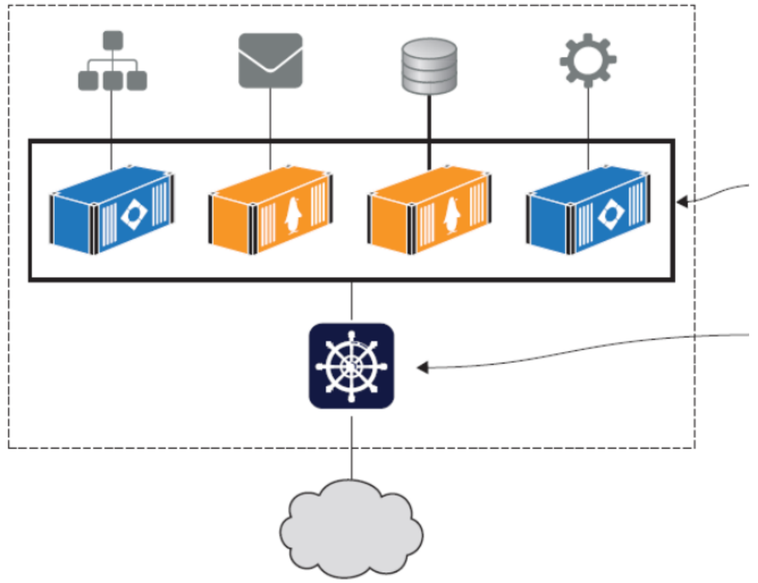


Docker overview

- Docker is an **open platform** for developing, shipping, and running applications.
- Docker is a platform for running applications in lightweight units called **containers**.
- Docker enables you to **separate your applications from your infrastructure** so you can deliver software quickly
- **Reduce the delay** between writing code and running it in production
- People’s number one “**most wanted**” technology

Why are containers and Docker so important?

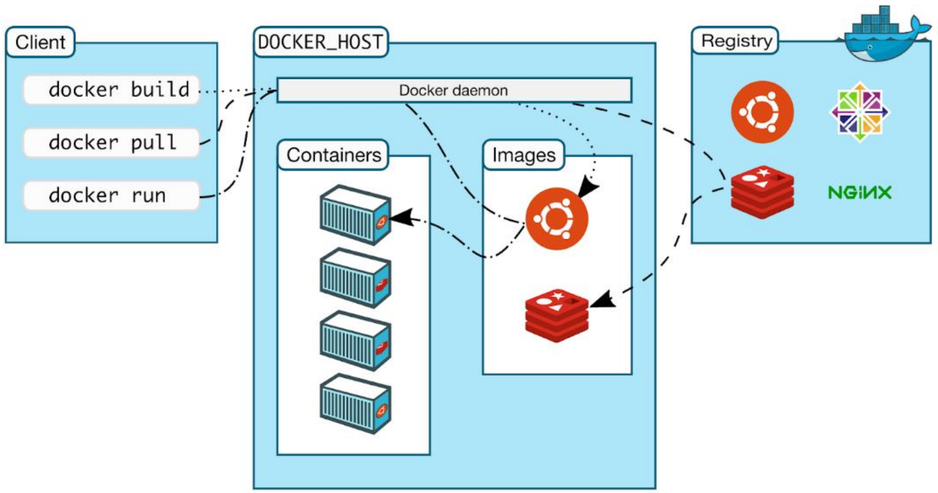
- Migrating apps to the cloud
- Modernizing legacy apps
- Building new cloud-native apps
- Technical innovation: Serverless and more
- Docker provides an abstraction
- Protecting your computer
- Improving portability
- Getting organized



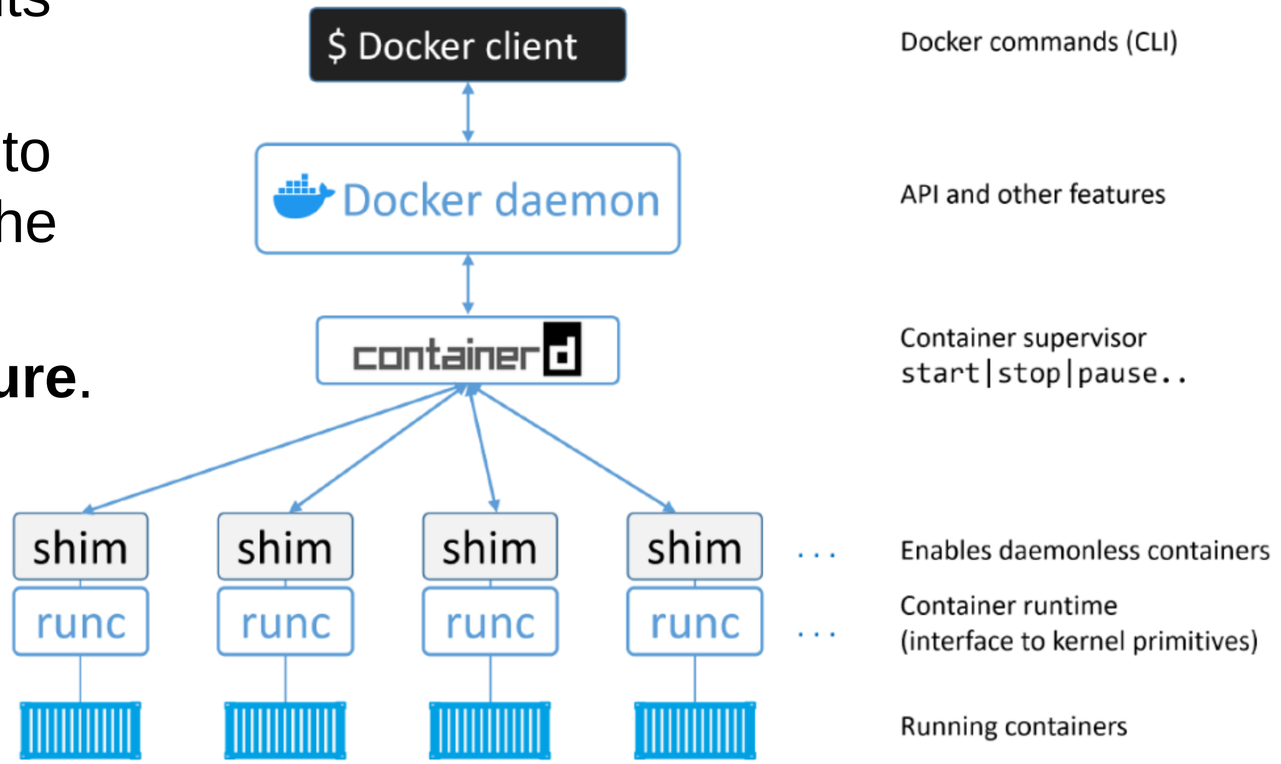
Source: Learn Docker in a Month of Lunches, ELTON STONEMAN

Docker Architecture

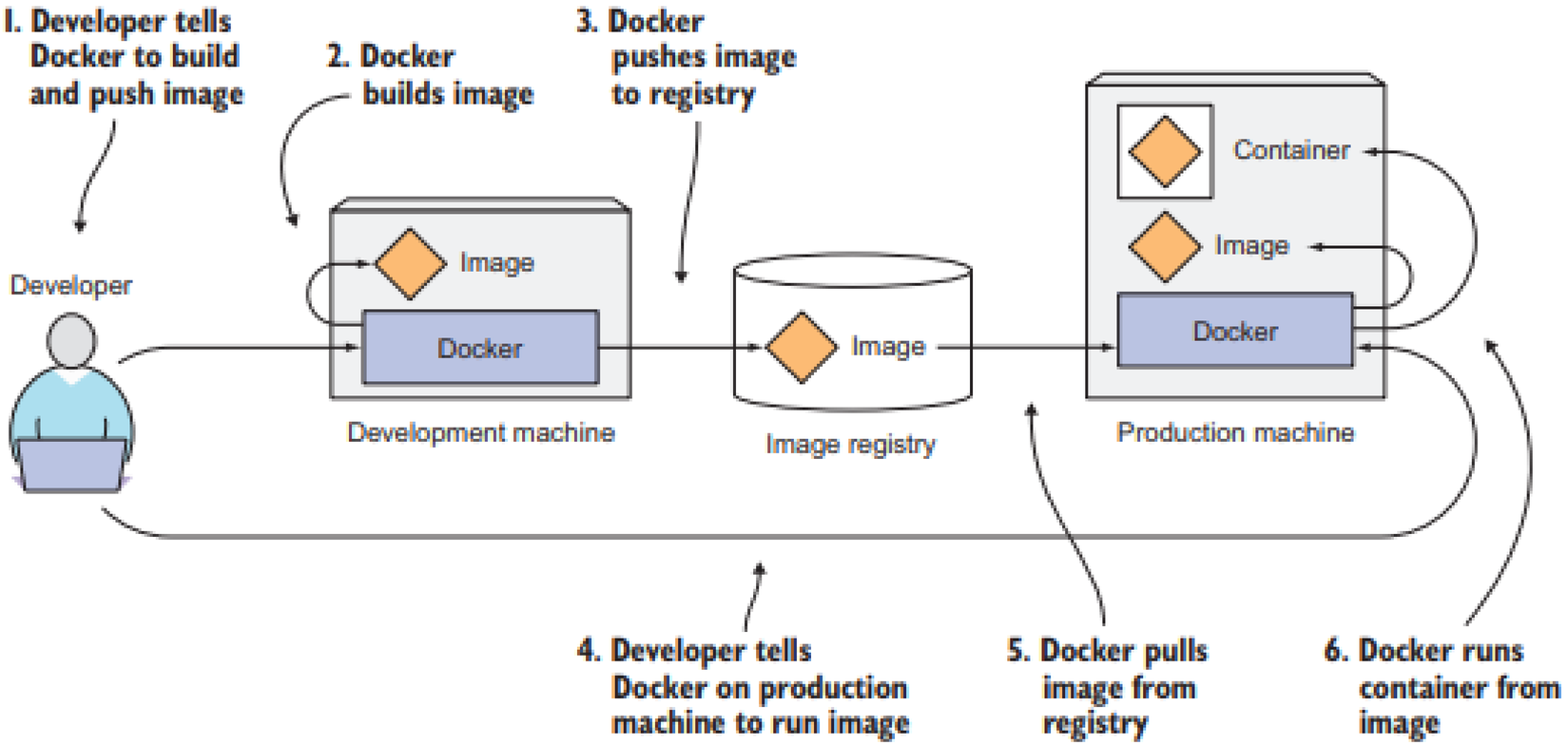
- Docker is written in the **Go programming language** and takes advantage of several features of the **Linux kernel** to deliver its functionality.
- Uses a technology called namespaces to provide the isolated workspace called the container.
- Docker uses a **client-server architecture**.



Source: https://docs.docker.com/get-started/overview/



Starting a simple container



Source: Marko Luksa - Kubernetes in Action-Manning Publications (2018)

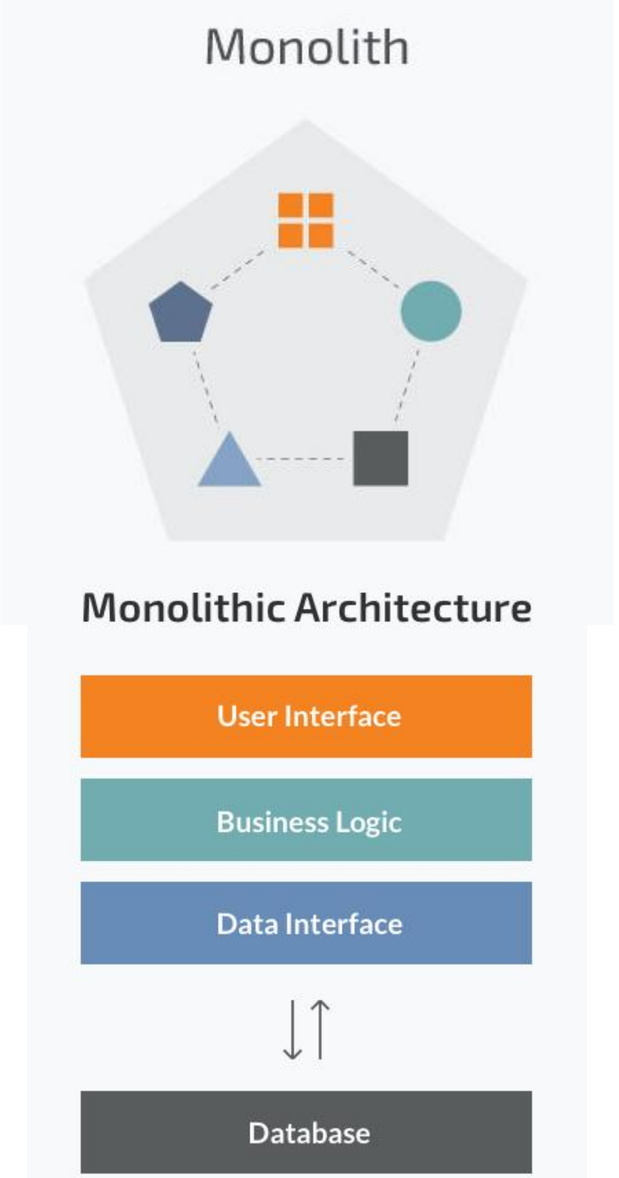
Microservices

Source: https://dev.to/lovepreetsingh/microservices-vs-monolithic-architecture-a-practical-approach-4m06

48

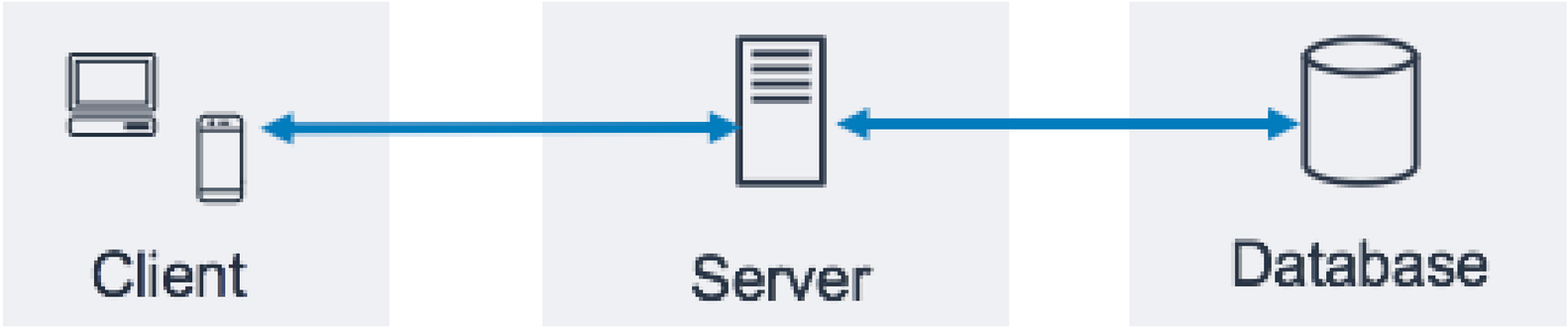
Traditional applications

- Monolithic application
- Years ago, most software applications were big monoliths
- Built as a single and indivisible unit
- Still widespread today
- Challenges
  - Handling a huge code base
  - Adopting a new technology
  - Scaling
  - Lack of modularity
  - Deployment
  - Implementing new changes -> slow (entire application has to be rewritten)



Source: Microservices vs Monolith: which architecture is the best choice for your business? https://www.n-ix.com/microservices-vs-monolith-which-architecture-best-choice-your-business/

- Client-side user interface, a server side-application, and a database.

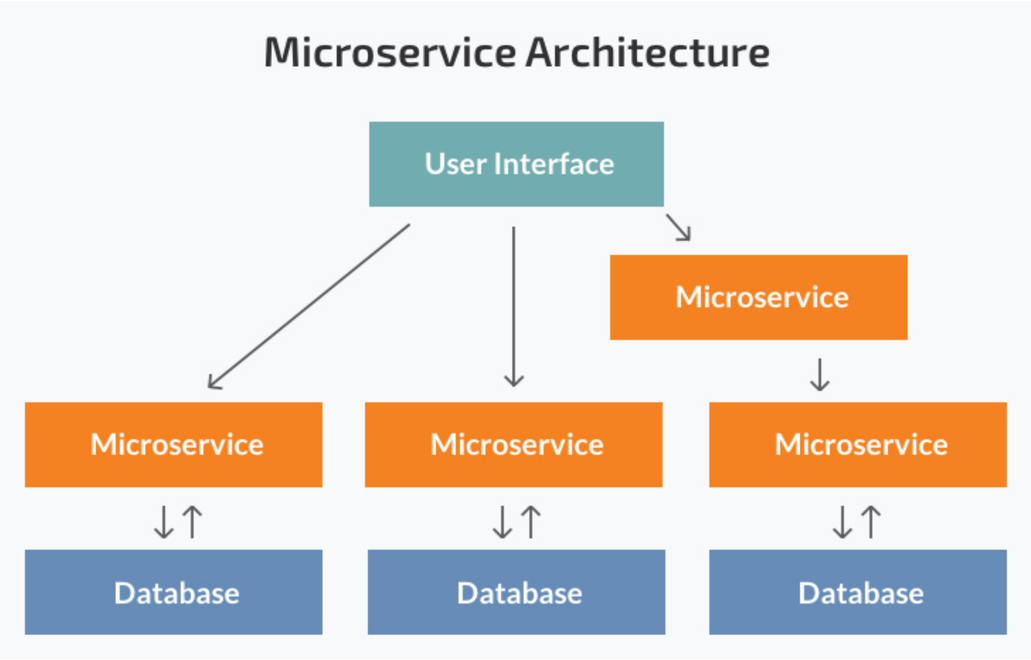


Source: Three-Tier Architecture Overview - https://docs.aws.amazon.com/whitepapers/latest/serverless-multi-tier-architectures-api-gateway-lambda/three-tier-architecture-overview.html

- Strengths
  - Less cross-cutting concerns (logging, handling, caching, and performance monitoring)
  - Little configuration it required
  - Easier debugging and testing
  - Simple to deploy (just one file or directory)
  - Simple to develop (knowledge – standard way of development)

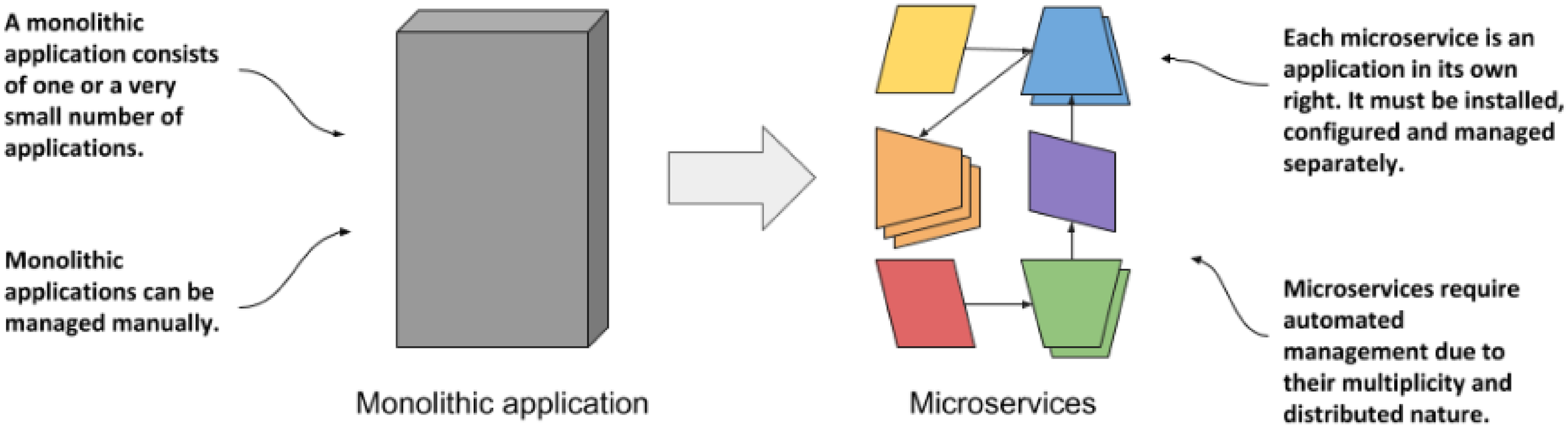
Microservices

- Today, these big monolithic legacy applications are slowly being broken down into smaller, independently running components called microservices.



Source: Microservices vs Monolith: which architecture is the best choice for your business? https://www.n-ix.com/microservices-vs-monolith-which-architecture-best-choice-your-business/

- The monoliths were divided into dozens, **sometimes hundreds, of separate processes**.
- This allowed organizations to divide their **development departments into smaller teams**.



Source: Marko Luksa - Kubernetes in Action-Manning Publications (2022)

- These units carry out every application process as a separate service.
- So all the services have their **own logic and the database**.
- Components **communicate** with each other through defined methods called **APIs**.

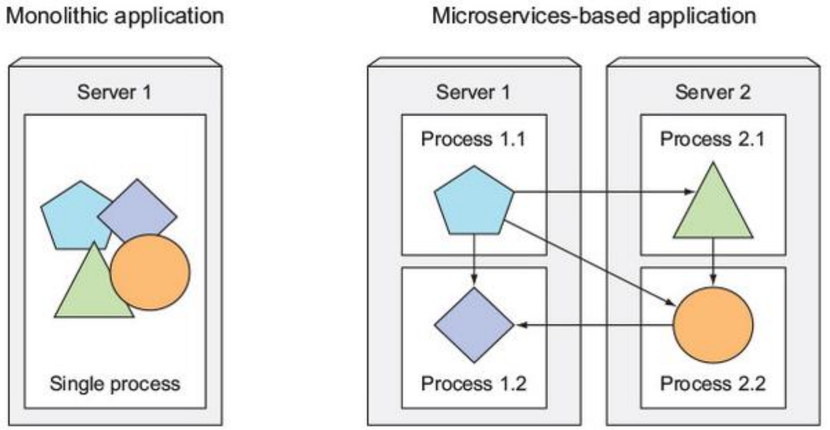
- Strengths:
  - Independent components
  - Easier understanding
  - Better scalability
  - Flexibility in choosing the technology
  - The higher level of agility

- Weaknesses:
  - Extra complexity
  - System distribution
  - Cross-cutting concerns (configuration, logging, metrics, health checks)
  - Testing

- **Automating the management of microservices:**
  - Each microservice is now a separate application with its **own development and release cycle**.
  - Individual parts of the entire application **no longer need to run on the same computer.** -> easier to scale the entire system
- **Bridging the dev and ops divide:**
  - The development team is now much more involved in the daily management of the deployed software.
- **Standardizing the cloud:**
  - Customers can now deploy applications to any cloud provider through a standard set of APIs provided by Kubernetes.

Splitting apps into microservices

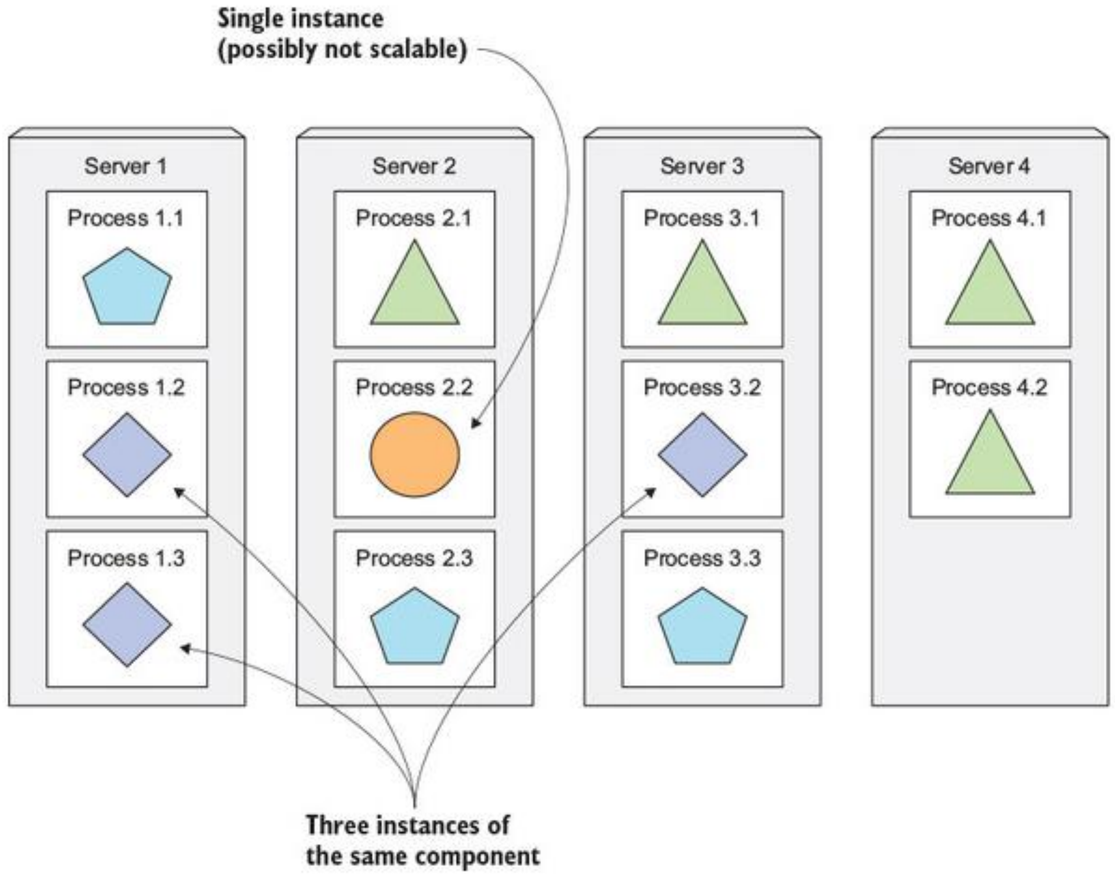
- We start splitting complex monolithic applications into smaller independently deployable components called microservices.
- Each microservice:
  - runs as an independent process,
  - communicates with other microservices through simple, well-defined interfaces (APIs)
  - can be written in the language that's most appropriate for implementing that specific microservice



Source: Marko Luksa - Kubernetes in Action-Manning Publications (2018)

Scaling microservices

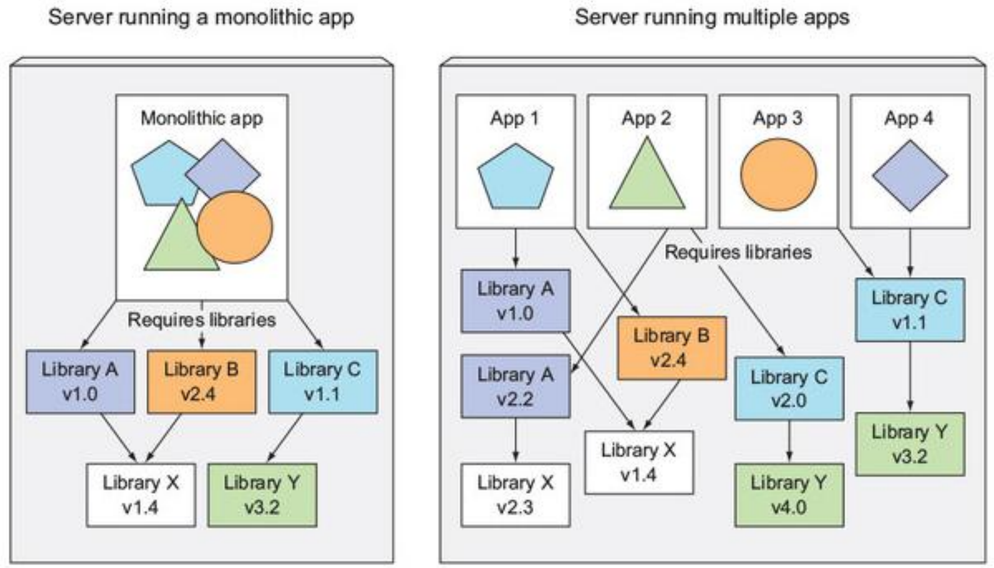
- Scaling is done on a per-service basis.
  - you have the option of scaling only those services that require more resources
- Splitting the app into microservices allows you to horizontally scale the parts that allow scaling out



Source: Marko Luksa - Kubernetes in Action-Manning Publications (2018)

Microservices environment requirements

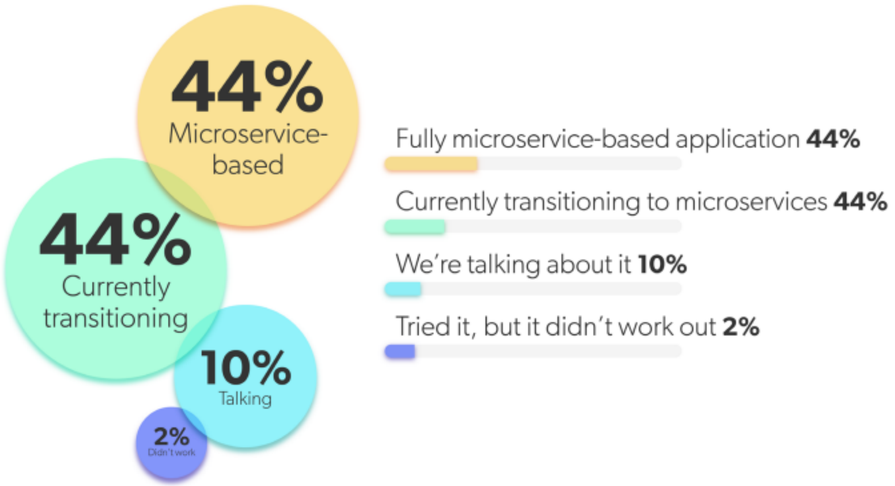
- Components in a microservices architecture are also developed independently
- common to have separate teams developing each component
- different libraries and different versions of the same libraries
- exact same environment during development, testing and in production -> container technologies



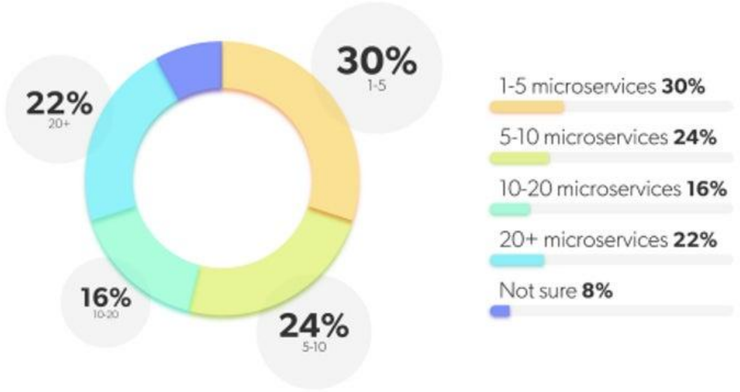
Source: Marko Luksa - Kubernetes in Action-Manning Publications (2018)

Microservices Trends

What Is Your Status for Microservice Adoption?

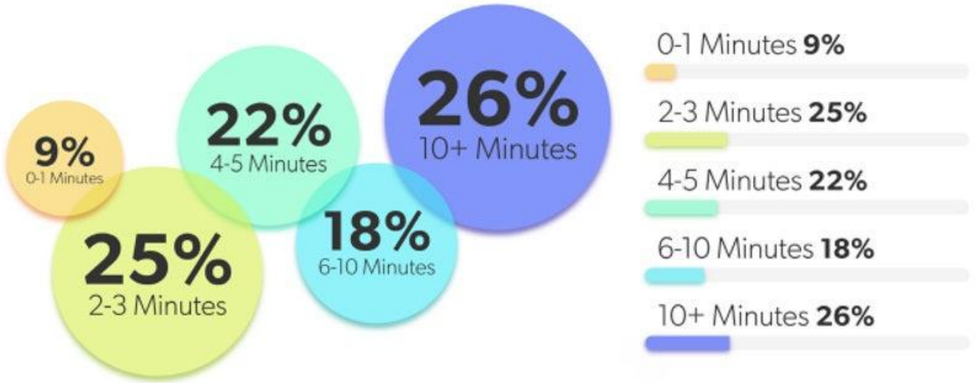


How Many Microservices Do You Have in Your Primary Application?



Have You Experienced an Increase in the Time it Takes to Start Up the Services in Your Microservice Application Since the Original Transition/Creation of the Microservice?

How Long Does it Take You to Remotely Deploy Your Containerized Environment?

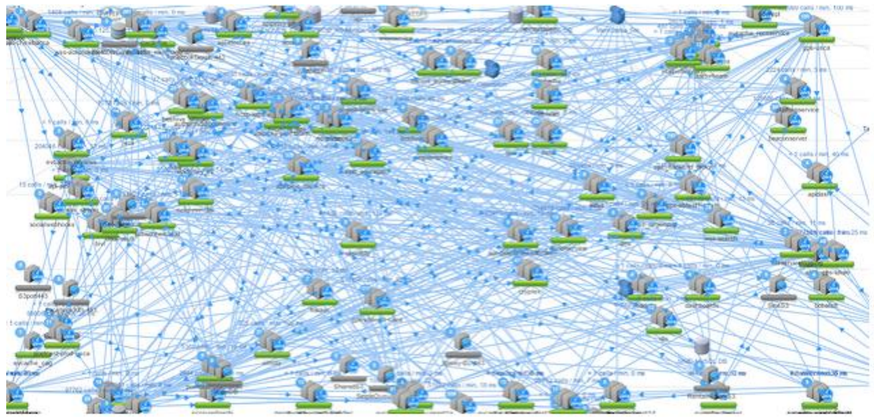


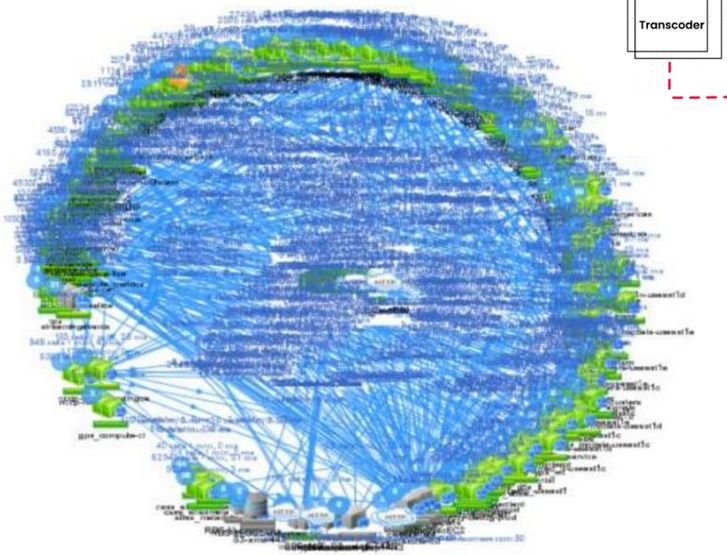
Source: https://www.jrebel.com/blog/2022-java-architecture-trends

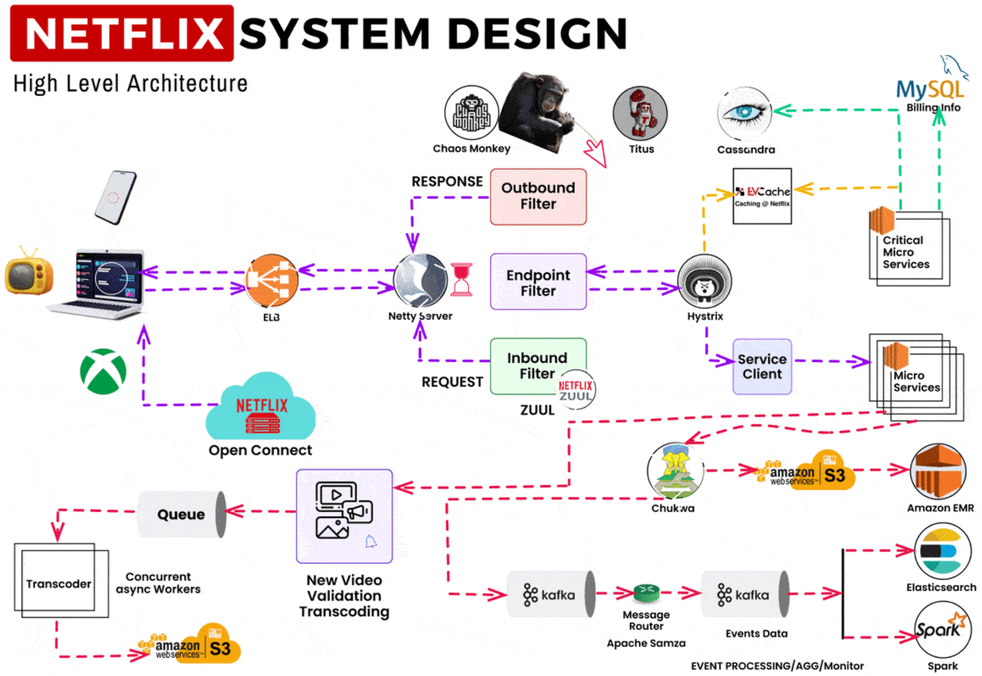
Complexity and problems of Microservices

Example: Netflix

- Netflix runs on 1000+ microservices
- In 2017 (!) launched over one million containers per week.

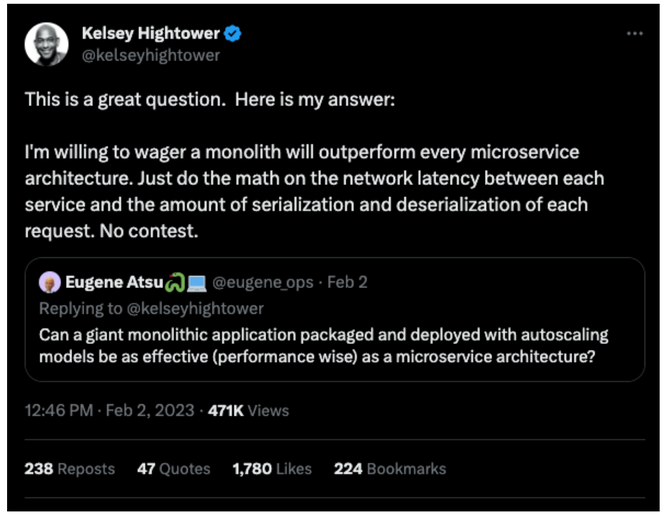






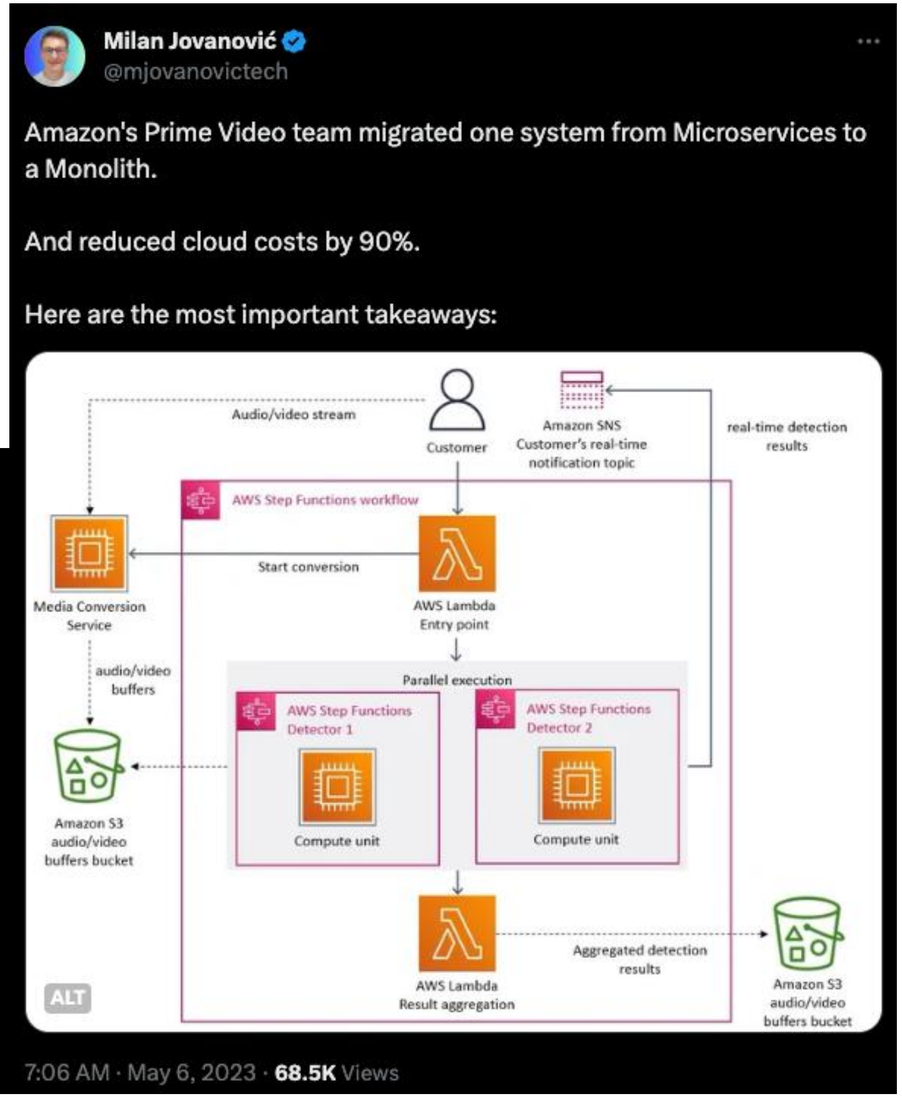
Source: https://miro.medium.com/v2/resize:fit:679/1*oT-ssryndQnVmbwBhjXNzw.gif

- Back in the days distributed systems were respected, feared, and generally avoided.
- There is no standard tooling for microservices-based development - there is no common framework.
- **Not starting out with microservices on day one** - no matter the problem -> MonolithFirst
  - Dropbox, Twitter, Netflix, Facebook, GitHub, Instagram, Shopify, StackOverflow - these companies and others started out as monolithic code bases.
    - WhatsApp went supernova with their Erlang monolith and a relatively small team (50 engineers)
    - Instagram was acquired for billions - with a crew of 12.
    - StackOverflow makes it a point of pride how little hardware they need to run the massive site.
  - What problem are you solving? Is it scale? *"The only thing harder than a distributed system is a BAD distributed system."*



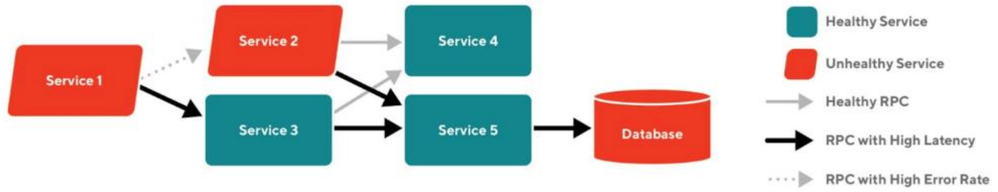
- Each microservice is being **maintained by a dedicated team**, walled off behind a beautiful, backward-compatible, versioned API. Rarely even have to communicate with that team - as if the microservice was maintained by a 3rd party vendor. **This rarely happens.**
- **YASS - “yet another stupid service”? This includes:**

✓ Developer privileges in GitHub/GitLab
✓ Default environment variables and configuration
✓ CI/CD
✓ Code quality checkers
✓ Code review settings
✓ Branch rules and protections
✓ Monitoring and observability
✓ Test harness
✓ Infrastructure-as-code

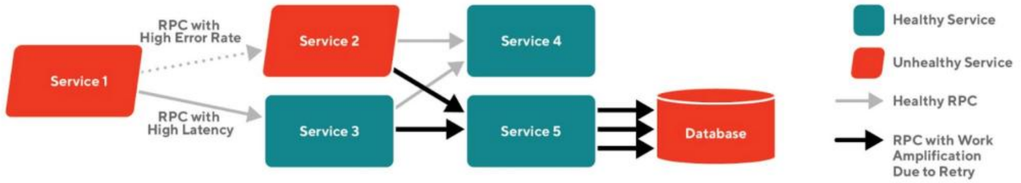


- Did you war-game the endless permutations of things that can and will go wrong? Is there backpressure? Circuit breakers? Queues? Jitter? Sensible timeouts on every endpoint?

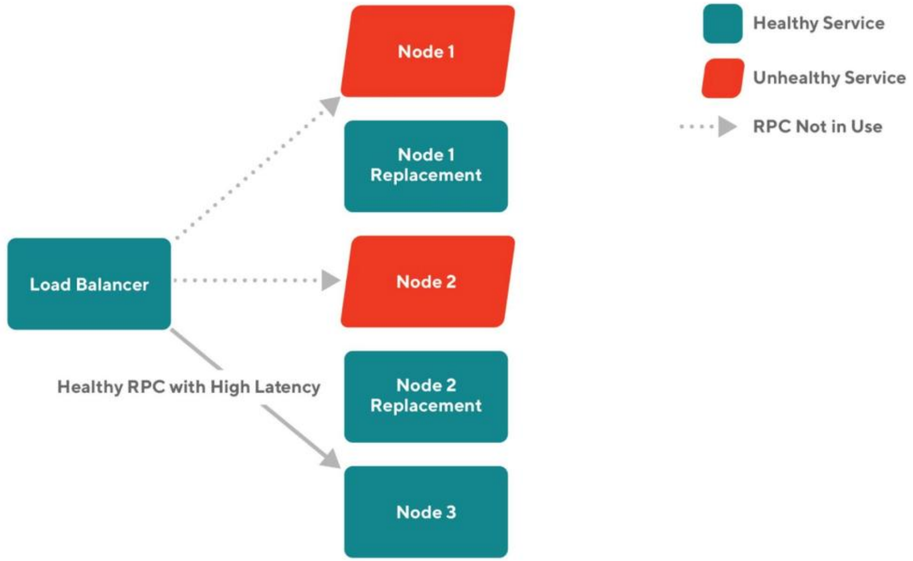
Cascading failure: a chain reaction of different interconnected services failing



•**Retry storm:** when retries put extra pressure on a degraded service



•**Death spiral:** some nodes fail, causing more traffic to be routed to the healthy nodes, making them fail too



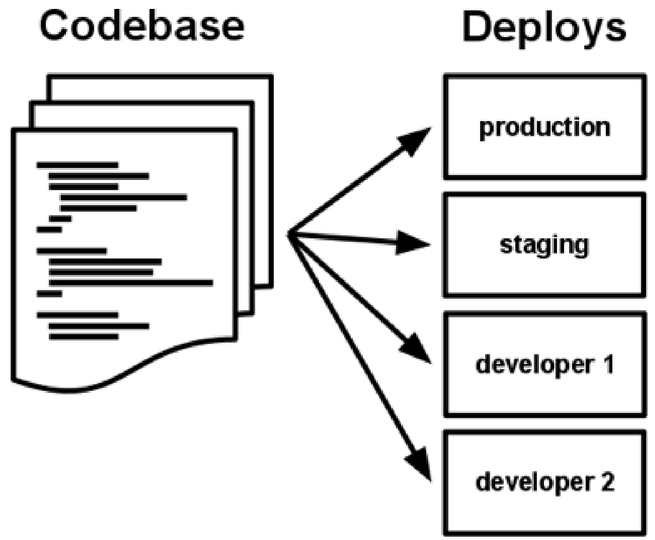
Source: https://doordash.engineering/2023/03/14/failure-mitigation-for-microservices-an-intro-to-aperture/

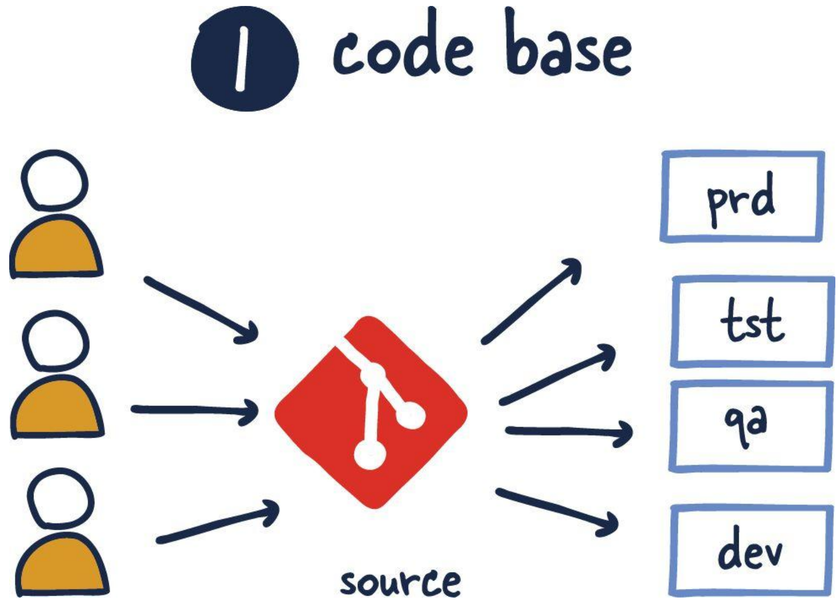
The Twelve Factor App

- The twelve-factor app is a methodology for building software-as-a-service apps that.
  - **Are suitable for deployment on modern cloud platforms**, obviating the need for servers and systems administration.
  - **Minimize divergence between development and production**, enabling continuous deployment for maximum agility;
  - And can **scale up without significant changes to tooling**, architecture, or development practices.

Codebase

- **One codebase tracked in revision control (Git); many deploys.**
  - App is always tracked in a version control system
  - If there are multiple codebases, it's not an app – it's a distributed system. Each component in a distributed system is an app, and each can individually comply with twelve-factor.
  - A deploy is a running instance of the app.
  - The codebase is the same across all deploys.
- Especially in a microservices-focused world, all the overhead needed to maintain multiple repositories can be tiresome.

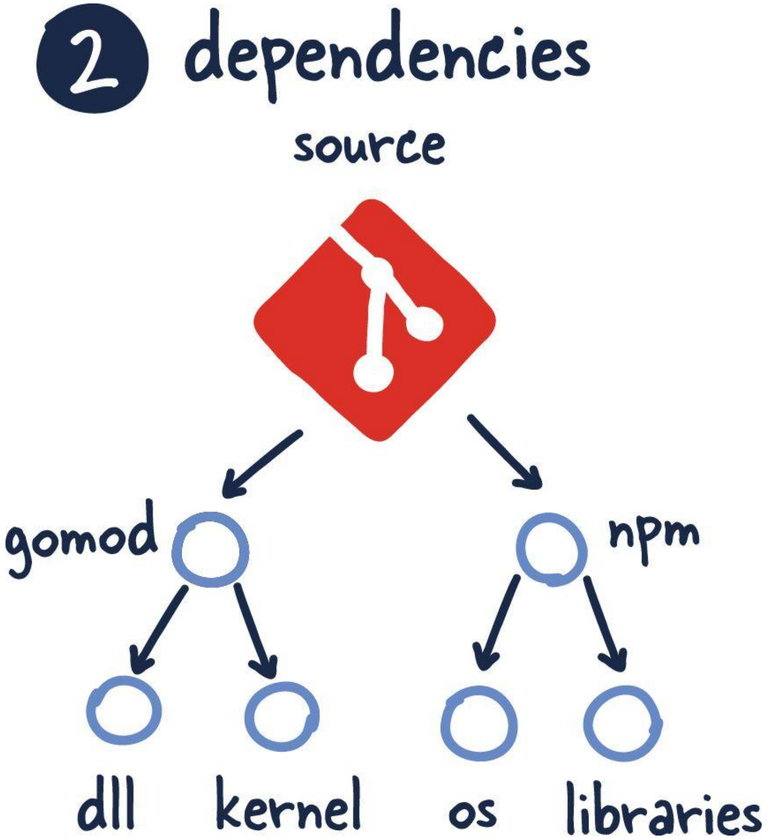




Source: https://architecturenotes.co/12-factor-app-revisited/

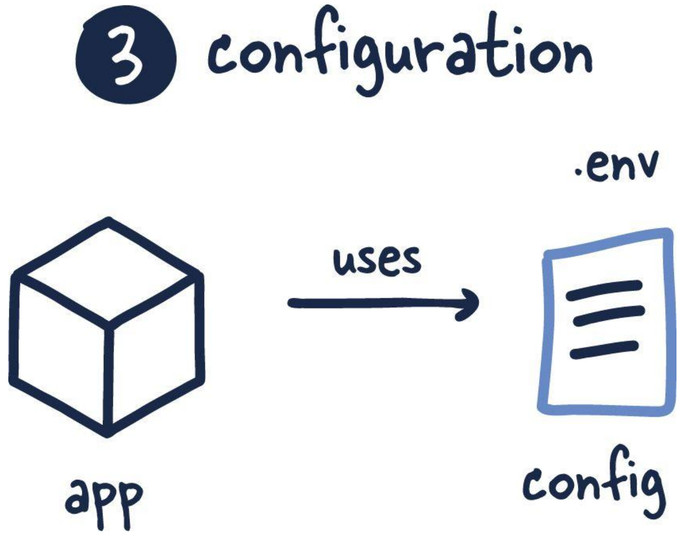
Dependencies

- **Explicitly declare and isolate dependencies**
- Libraries installed through a packaging system can be installed system-wide or scoped into the directory containing the app.
- A twelve-factor app never relies on implicit existence of system-wide packages.
- One benefit of explicit dependency declaration is that it simplifies setup for developers new to the app.
- Twelve-factor apps also do not rely on the implicit existence of any system tools.



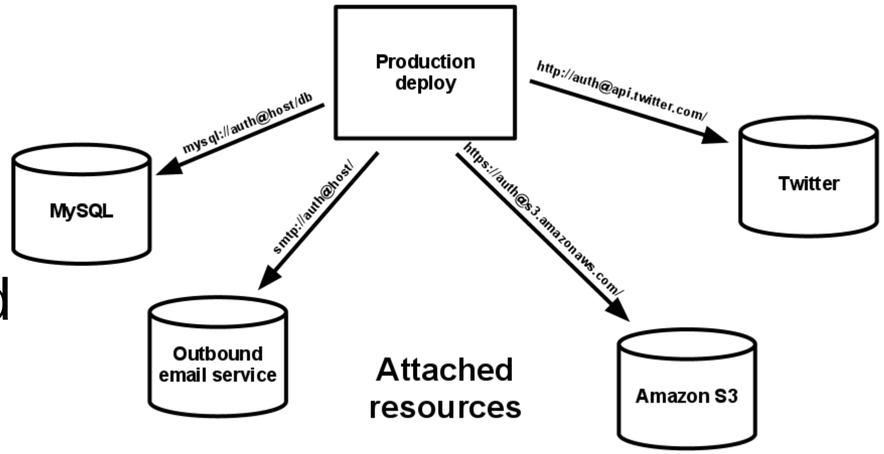
Configuration

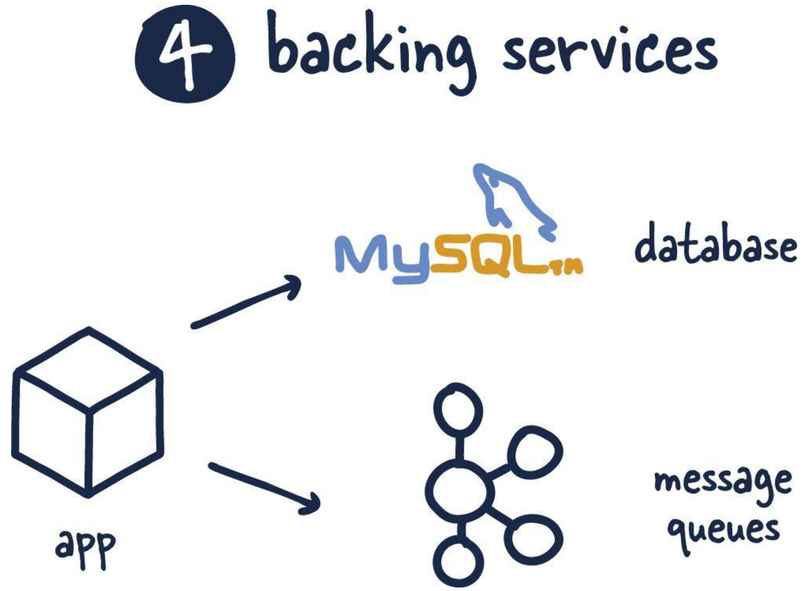
- **Store config in the environment.**
- An app’s config is everything that is likely to vary between deploys.
- Requires strict separation of config from code.
- A litmus test for whether an app has all config correctly factored out of the code is whether the codebase could be made open source at any moment, without compromising any credentials.
- The twelve-factor app stores config in **environment variables**.



Backing Services

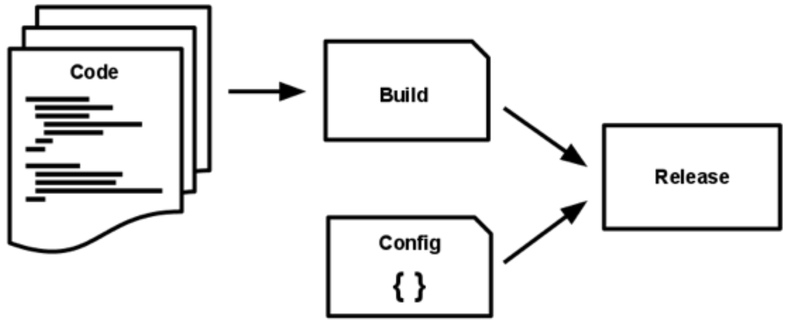
- **Treat [backing services] as attached resources.**
- **Backing Services**: Essential network services consumed by apps, such as databases, messaging systems, SMTP services, and caching systems.
- Traditionally handled by system administrators; can also include third-party managed services.
- **Twelve-Factor App Principle:** Treats local and third-party services equally as attached resources, with no code distinction. -> only config updates.
- Resources can be attached or detached to/from deployments as needed without code alterations.

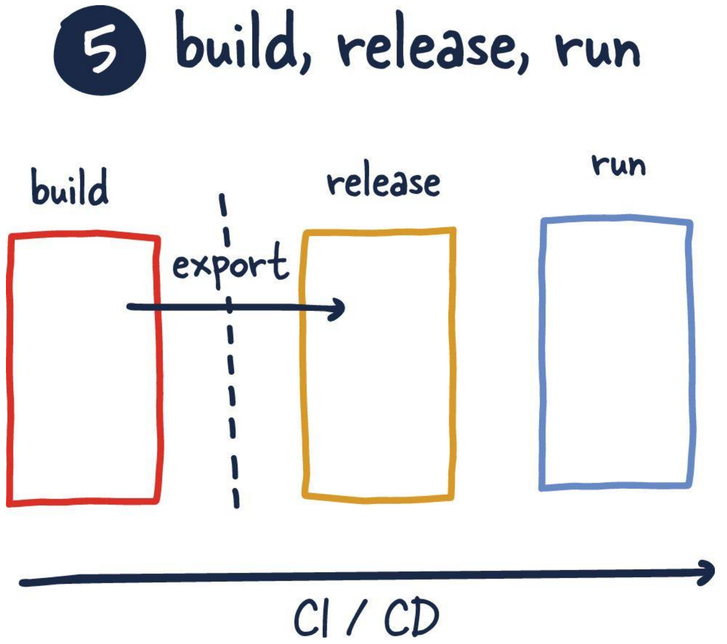




Build, release, run

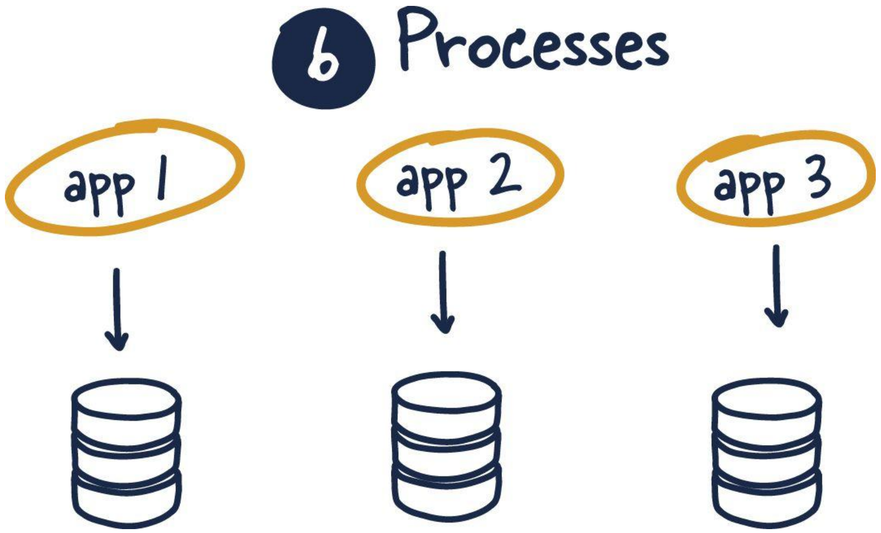
- **Strictly separate build and run stages.**
- Enforces strict separation of build, release, and run stages to prevent runtime code changes.
- Deployment tools typically offer release management tools, most notably the ability to roll back to a previous release.
- Every release should always have a unique release ID.





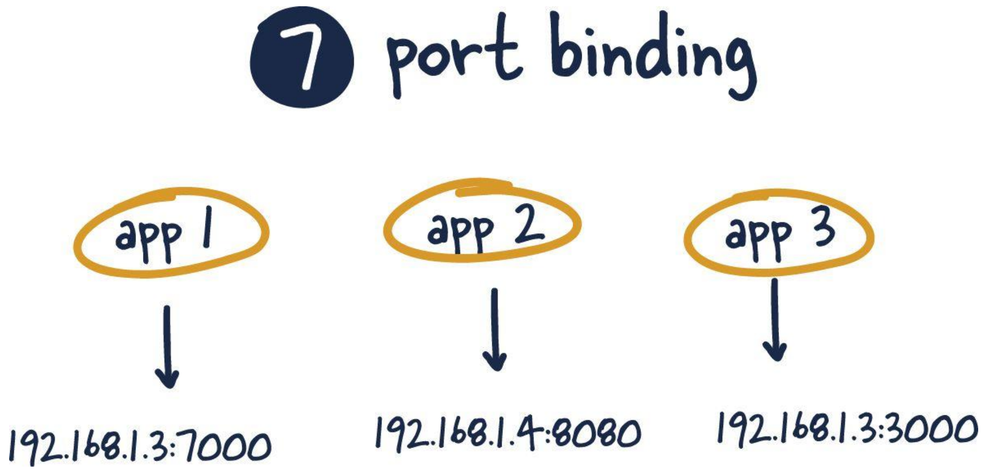
Processes

- **Execute the app as one or more stateless processes.**
- The app is executed in the execution environment as one or more processes.
- Stateless and share-nothing; persistent data must be stored in a stateful backing service like a database.
- Use of memory or filesystem for temporary caching only; no assumptions of persistence across requests or jobs.
- Avoid "sticky sessions"; use time-expiring datastores for session state data.



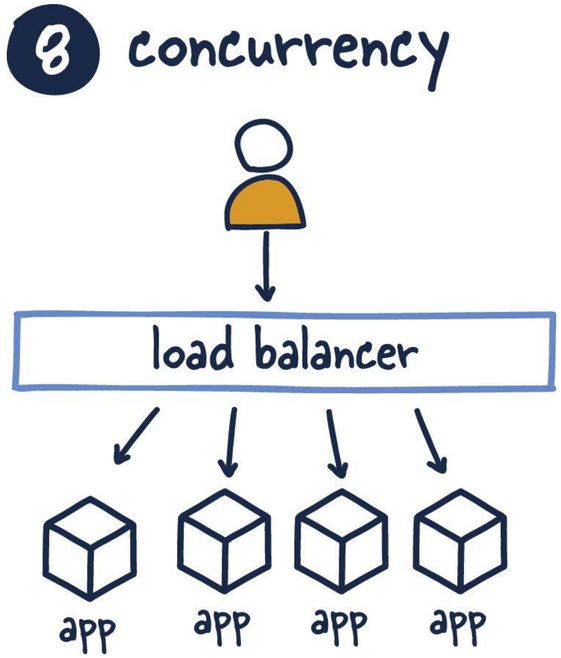
Port binding

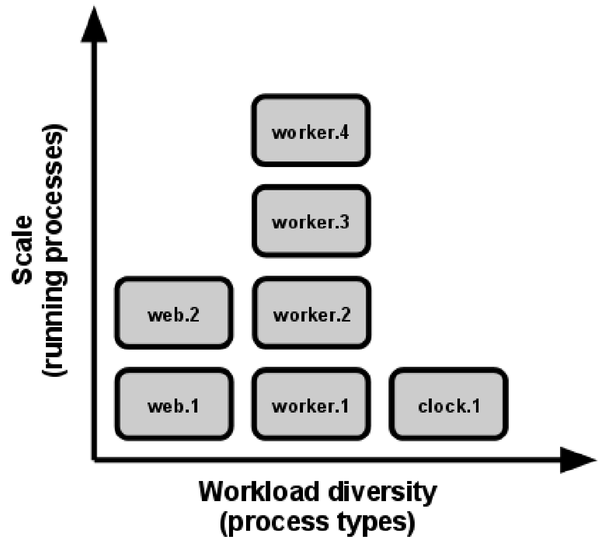
- **Export services via port binding.**
- Web apps are sometimes executed inside a webserver container.
- Self-contained, does not require runtime webserver injection, exports HTTP by binding to a port.



Concurrency

- **Scale out via the process model.**
- In the twelve-factor app, processes are a first class citizen.
- The process model truly shines when it comes time to scale out. The share-nothing, horizontally partitionable nature of twelve-factor app processes means that adding more concurrency is a simple and reliable operation.
- Twelve-factor app processes should never daemonize or write PID files.



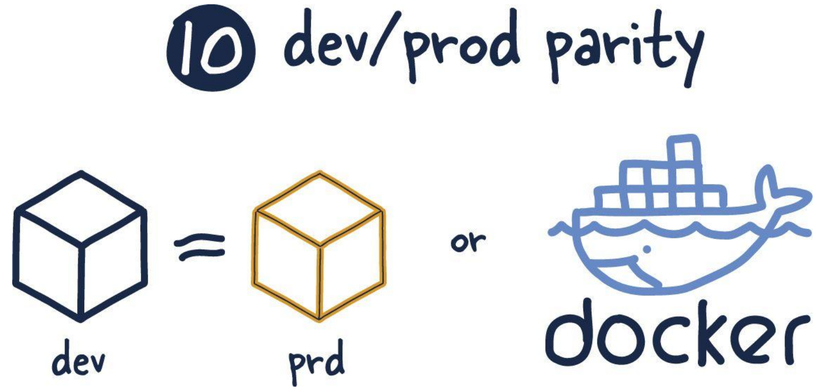


Disposability

- **Maximize robustness with fast startup and graceful shutdown.**
- The twelve-factor app's processes are disposable, **meaning they can be started or stopped at a moment's notice**.
- Processes should strive to minimize startup time.
- Processes shut down gracefully when they receive a SIGTERM signal from the process manager.
- A twelve-factor app is architected to handle unexpected, non-graceful terminations. Crash-only design takes this concept to its logical conclusion.

Dev/prod parity

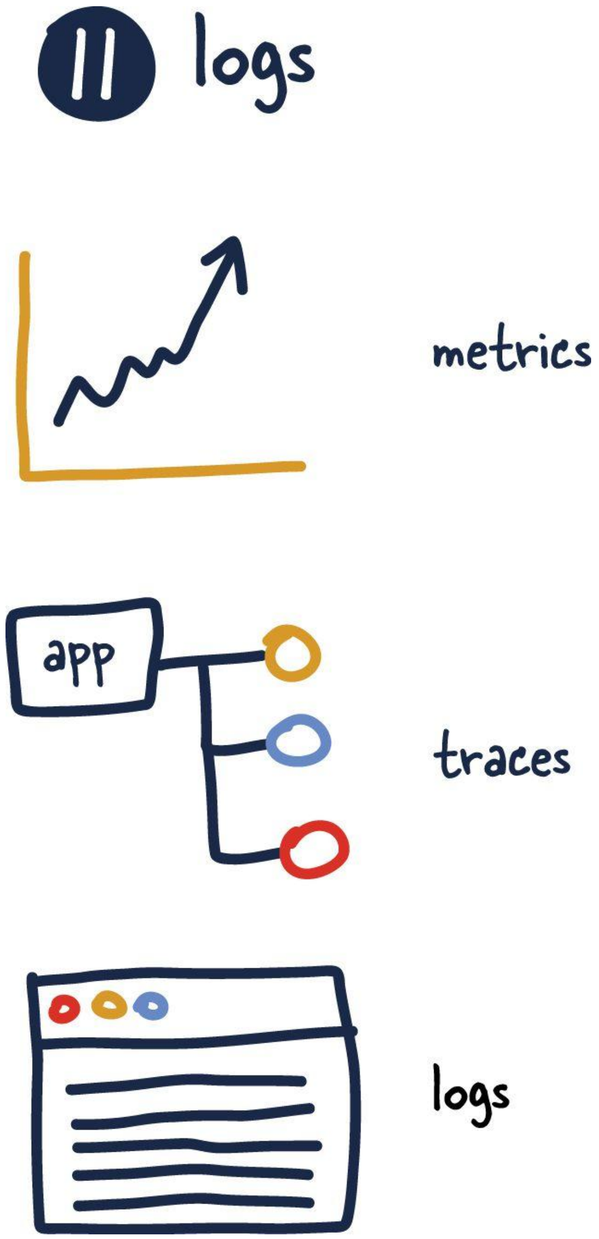
- **Keep development, staging, and production as similar as possible.**
- The twelve-factor app is designed for continuous deployment by keeping the gap between development and production small.
- The twelve-factor developer resists the urge to use different backing services between development and production.



	Traditional app	Twelve-factor app
Time between deploys	Weeks	Hours
Code authors vs code deployers	Different people	Same people
Dev vs production environments	Divergent	As similar as possible

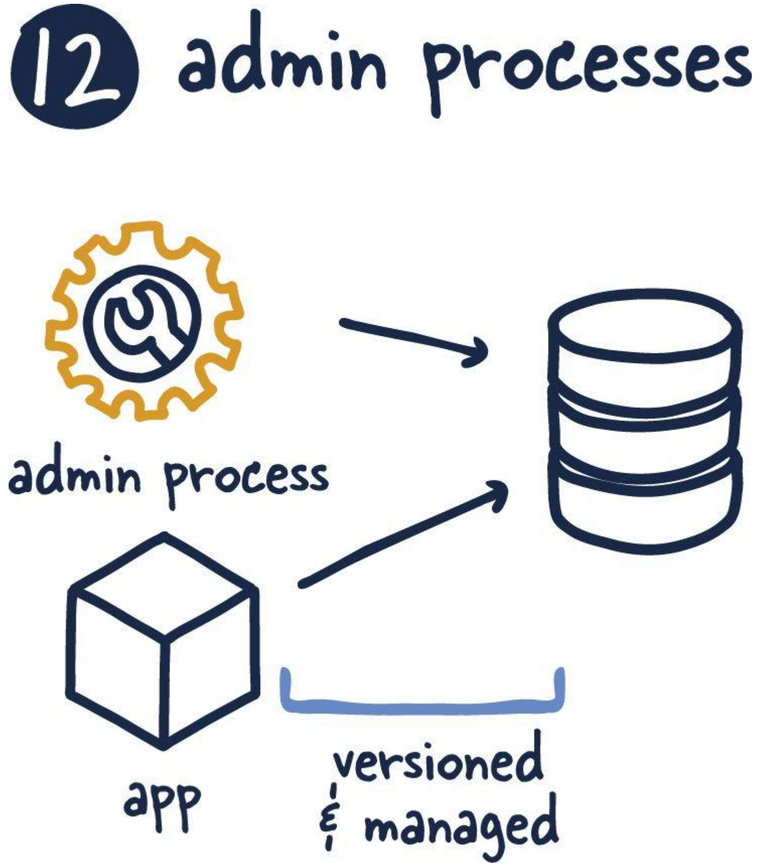
Logs

- **Treat logs as event streams.**
- A twelve-factor app never concerns itself with routing or storage of its output stream. It should not attempt to write to or manage logfiles.
- Each running process **writes its event stream, unbuffered, to stdout**.
- In staging or production deploys, each process' stream will be captured by the execution environment.



Admin processes

- **Run admin/management tasks as one-off processes**.
- One-off admin processes should be run in an identical environment as the regular long-running processes of the app.
- Admin code must ship with application code to avoid synchronization issues.

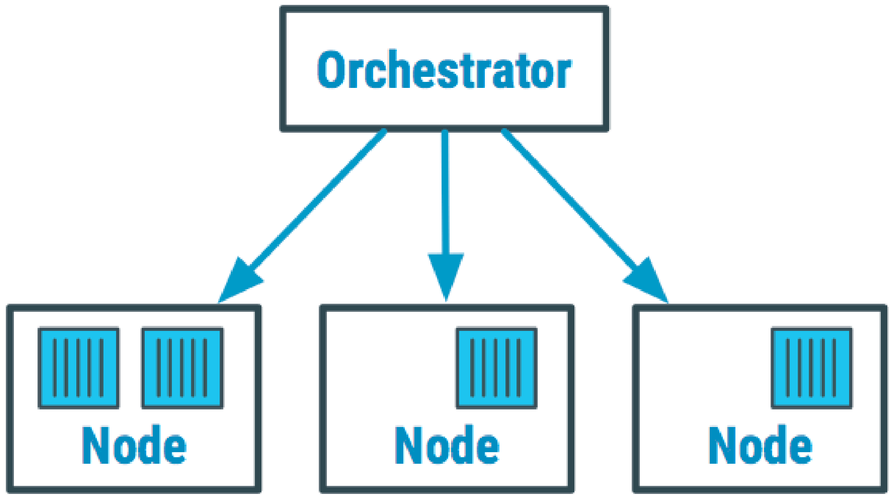


Container Orchestration

77

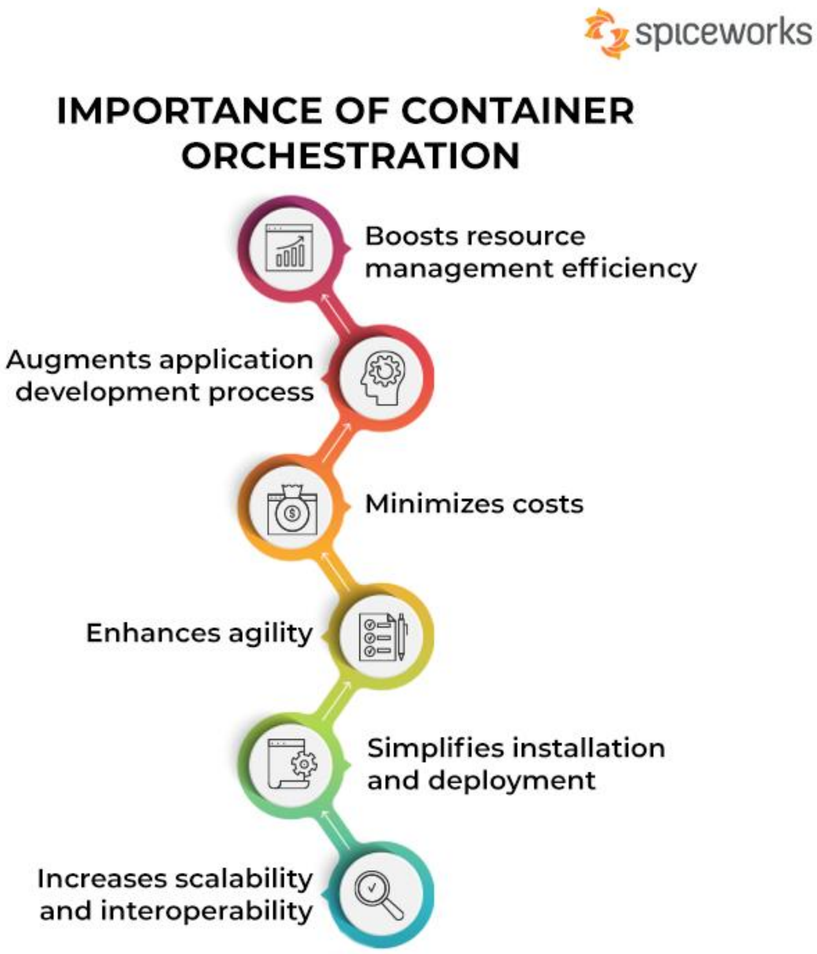
Container Orchestration

- Container orchestration is the **process of deploying containers on a compute cluster consisting of multiple nodes**.
- By abstracting the host infrastructure, container orchestration tools allow the users deploying to **entire cluster as a single deployment target.**



Source: https://devopedia.org/container-orchestration

- Container Orchestration envisions several features, some of which are mentioned below:
  - Provisioning hosts
  - Instantiating a set of containers
  - Rescheduling failed containers
  - Linking containers together through agreed interfaces
  - Exposing services to machines outside of the cluster
  - Scaling out or down the cluster by adding or removing containers
- Docker Swarm vs. Kubernetes



Source: https://images.spiceworks.com/wp-content/uploads/2022/08/12055735/Importance-of-Container-Orchestration.png

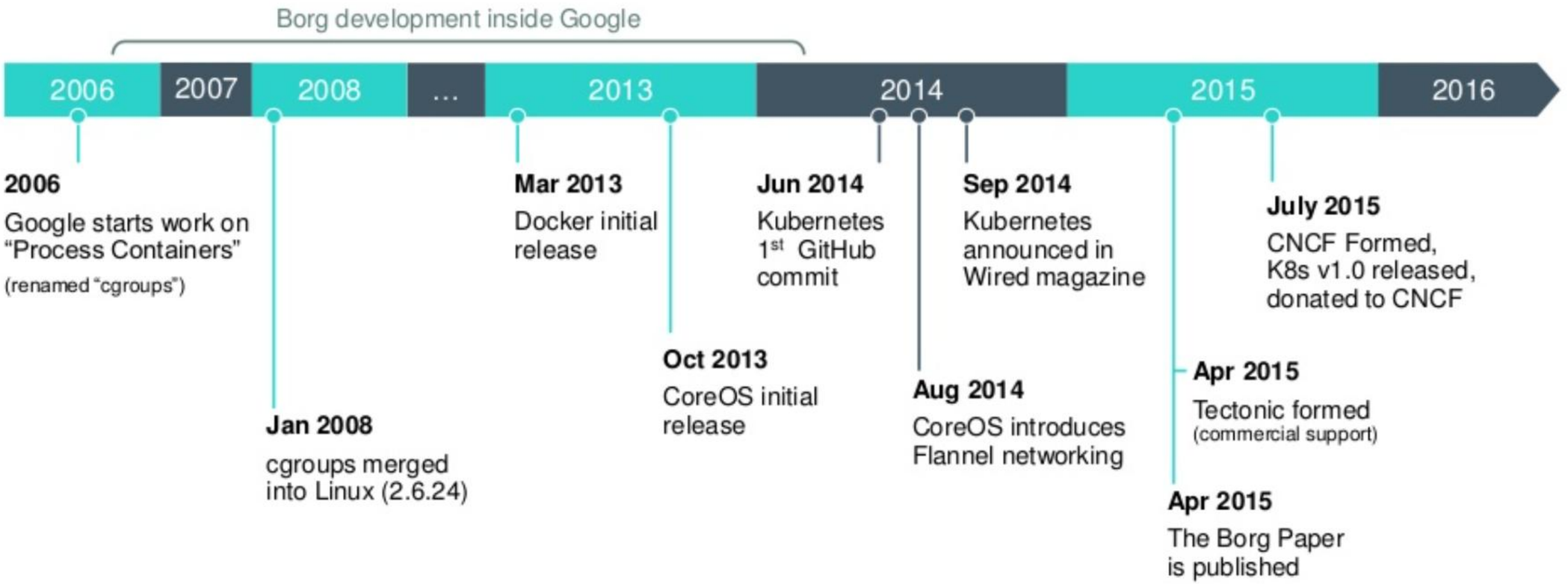
Source: https://www.simform.com/wp-content/uploads/2022/05/container-orchestration-tools.png

About Kubernetes

- https://kubernetes.io/
- Rapidly growing ecosystem
- The word **Kubernetes is Greek for pilot or helmsman**, the person who steers the ship
- **Kubernetes steers your applications** and reports on their status while **you** - the captain - **decide where you want the system to go.**
- HOW TO PRONOUNCE KUBERNETES AND WHAT IS K8S?
  - Most often it's **Koo-ber-netties** or **Koo-ber-nay'-tace**, but you may also hear **Koo-ber-nets**, although rarely.
  - It's also referred to **as Kube or K8s**, pronounced Kates, where the 8 signifies the number of letters omitted between the first and last letter.

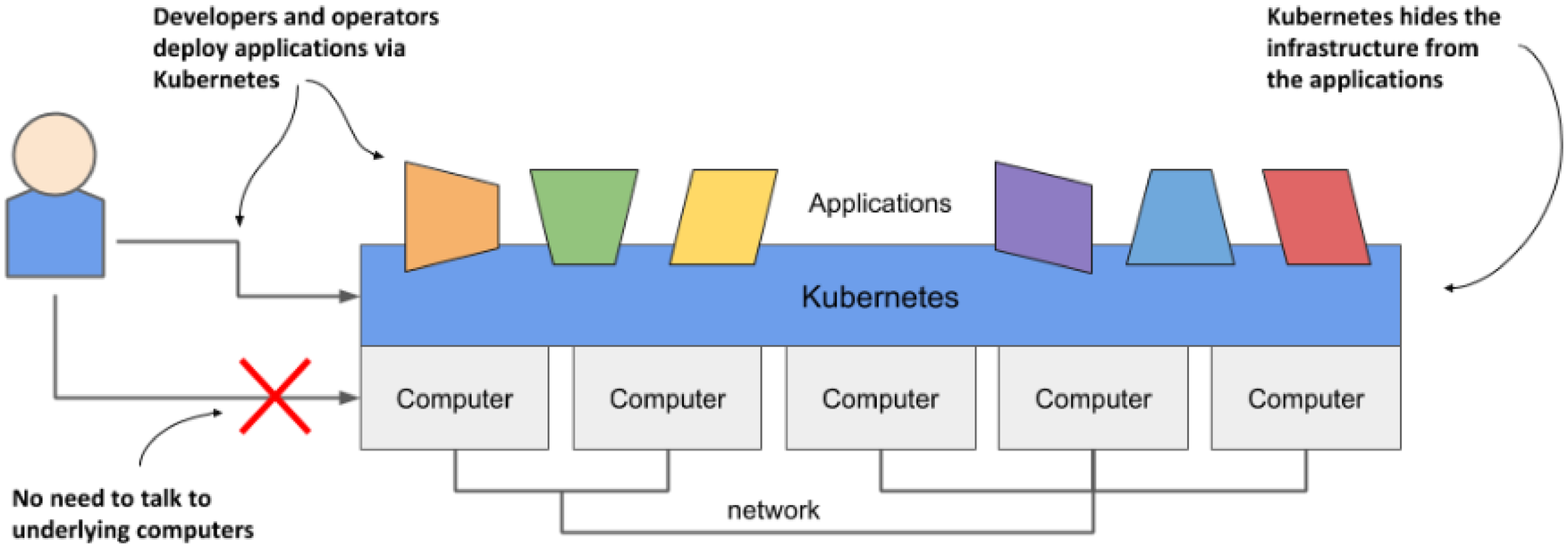
Kubernetes history

- Kubernetes was originally developed by **Google**.
  - As early as 2014, it was reported that Google start **two billion containers every week**. That's over **3,000 containers** per second
  - They run around 2.5 million servers (July 2016, Gartner).
- **Borg and Omega (2003)** - the predecessors of Kubernetes
  - Google developed an internal system called Borg (and later a new system called Omega) that helped both application developers and operators manage these thousands of applications
- In **2014 Google introduced Kubernetes**, an **open-source** project.
- Under the umbrella of the **Cloud Native Computing Foundation (CNCF)**, which is part of the Linux Foundation.
- CNCF organizes several KubeCon - CloudNativeCon conferences per year



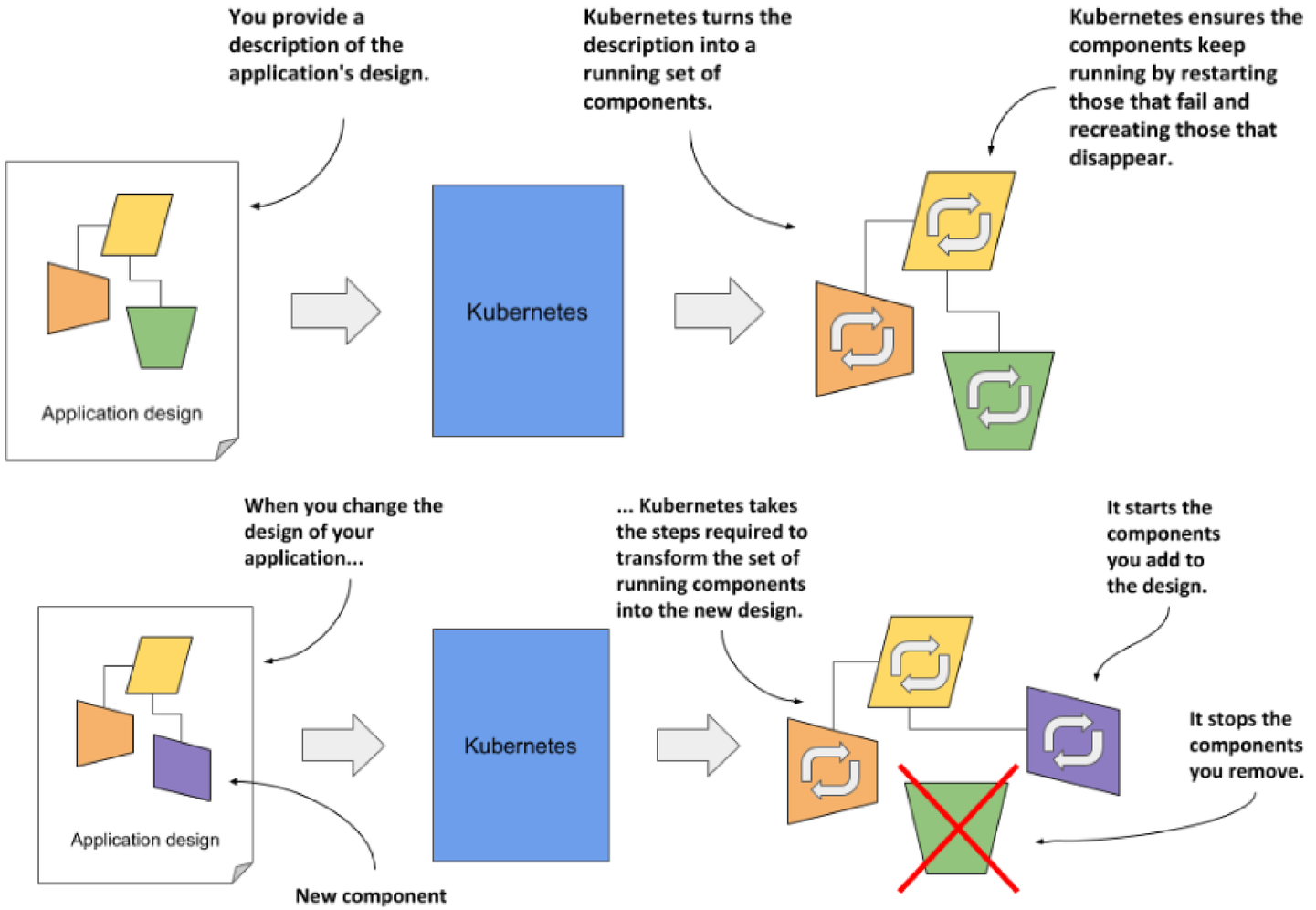
Source: https://www.slideshare.net/egg9/kubernetes-introduction

- **Kubernetes is a software system for automating the deployment and management of complex, large-scale application systems composed of computer processes running in containers.**
- **Main Kubernetes features**:
  - Abstracting away the infrastructure

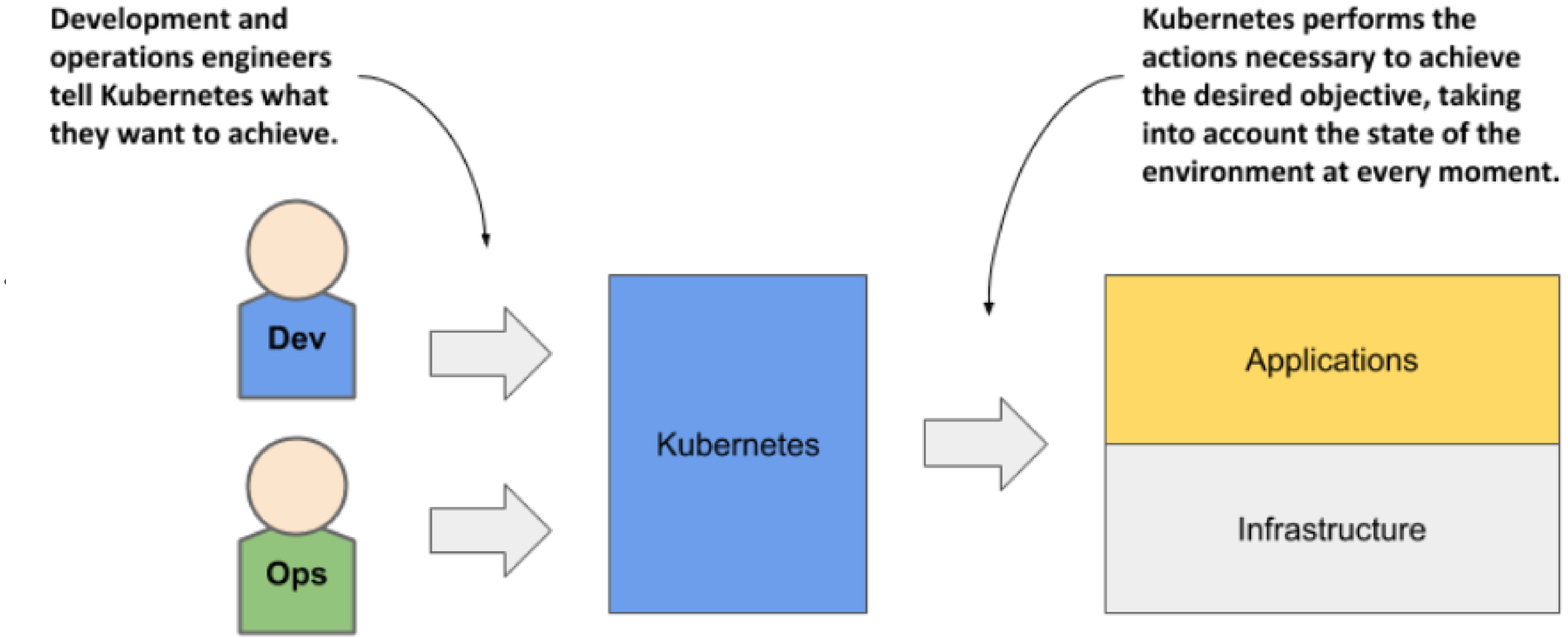


Source: Marko Luksa - Kubernetes in Action-Manning Publications (2022)

- Standardizing how we deploy applications
- Deploying applications declaratively



Source: Marko Luksa - Kubernetes in Action-Manning Publications (2022)

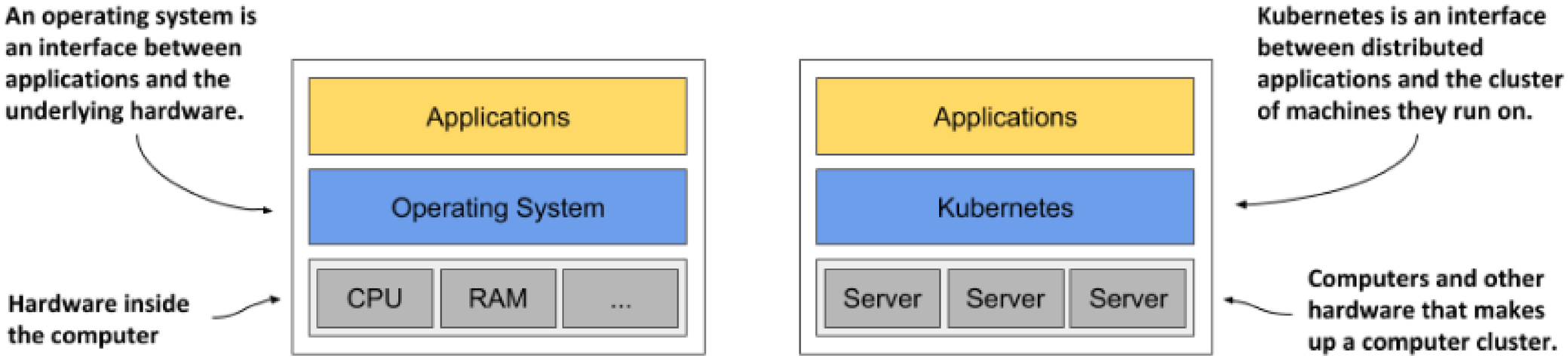


Source: Marko Luksa - Kubernetes in Action-Manning Publications (2022)

Understanding Kubernetes

- **Kubernetes is a platform for running containers.**
- It takes care of:
  - starting your containerized applications,
  - rolling out updates,
  - maintaining service levels,
  - scaling to meet demand,
  - securing access, and much more.

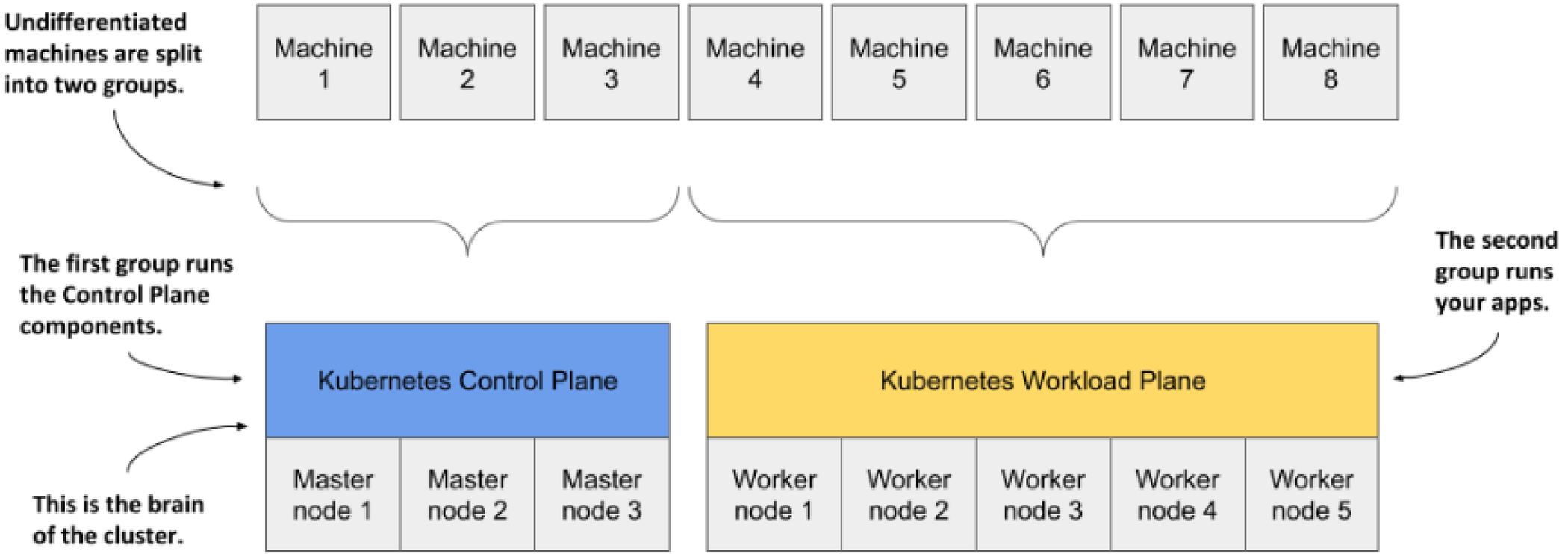
- Kubernetes is like an operating system for computer clusters



Source: Marko Luksa - Kubernetes in Action-Manning Publications (2022)

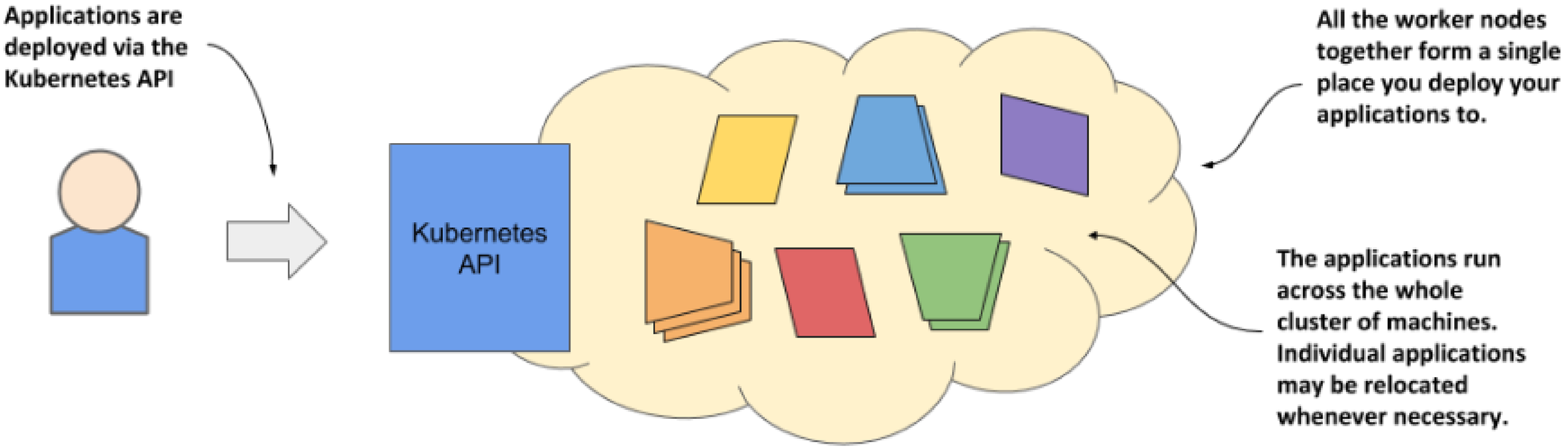
- Infrastructure-related mechanism provided by Kubernetes:
  - service discovery
  - horizontal scaling
  - load-balancing
  - self-healing
  - leader election

- How Kubernetes fits into a computer cluster?
  - master nodes -> run the **Kubernetes Control Plane**
  - worker nodes -> represent the **Workload Plane**
  - Non-production clusters can use a single master node, but highly available clusters use at least three physical master nodes to host the Control Plane. The number of worker nodes depends on the number of applications you'll deploy.



Source: Marko Luksa - Kubernetes in Action-Manning Publications (2022)

- How all cluster nodes become one large deployment area?
  - After Kubernetes is installed on the computers, **you no longer need to think about individual computers when deploying applications**.
  - You deploy your applications using the **Kubernetes API**, which is provided by the Kubernetes Control Plane.
  - Each application must be small enough to fit on one of the worker nodes.



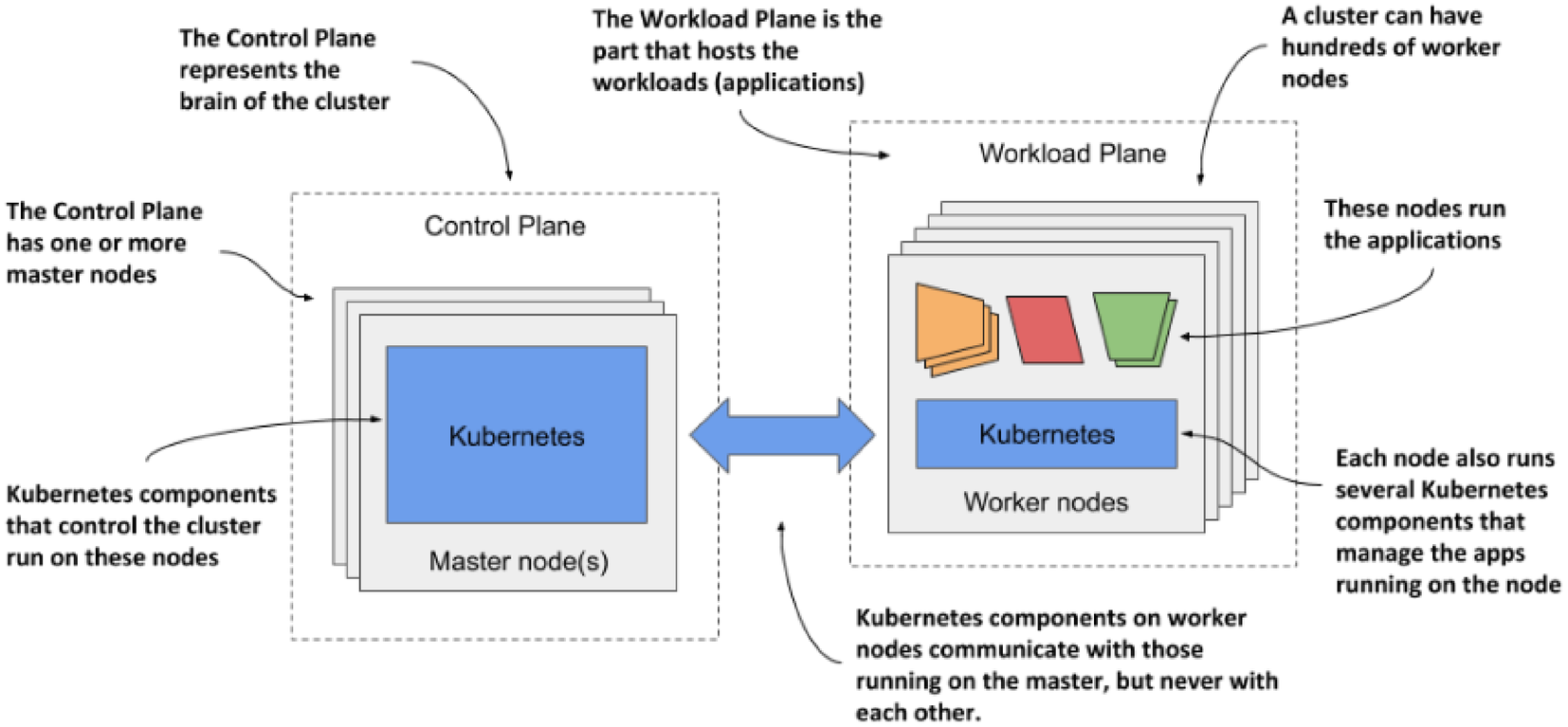
Source: Marko Luksa - Kubernetes in Action-Manning Publications (2022)

The benefits of using Kubernetes

- Self-service deployment of applications
- Reducing costs via better infrastructure utilization
- Automatically adjusting to changing load
- Keeping applications running smoothly (Health checking and self-healing)
- Simplifying application development

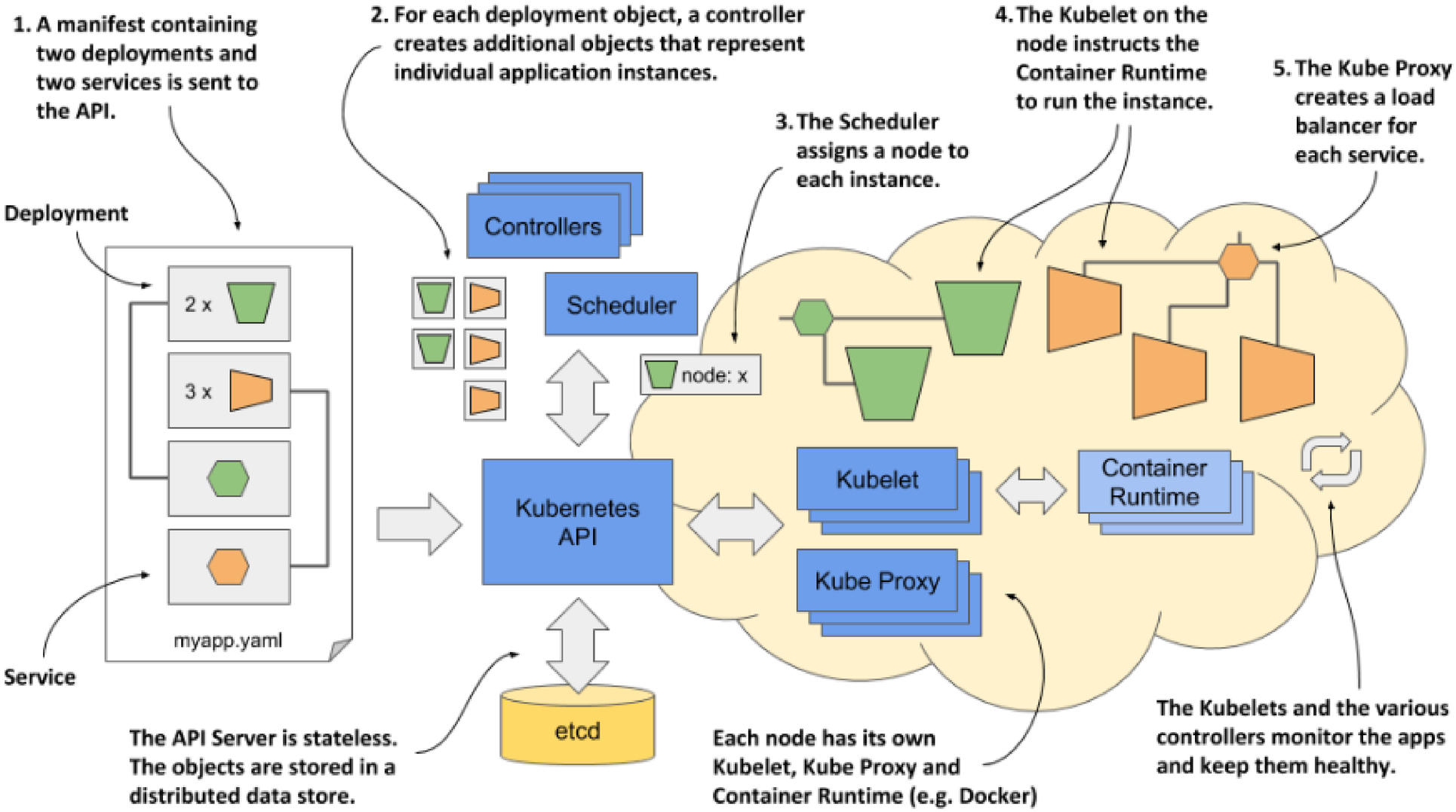
Kubernetes architecture

- A Kubernetes cluster consists of nodes divided into two groups:
  - A set of **master nodes** that host the **Control Plane components**, which are the **brains of the system**, since they control the entire cluster.
  - A set of **worker nodes** that form the **Workload Plane**, which is where your **workloads** (or applications) **run.**



Source: Marko Luksa - Kubernetes in Action-Manning Publications (2022)

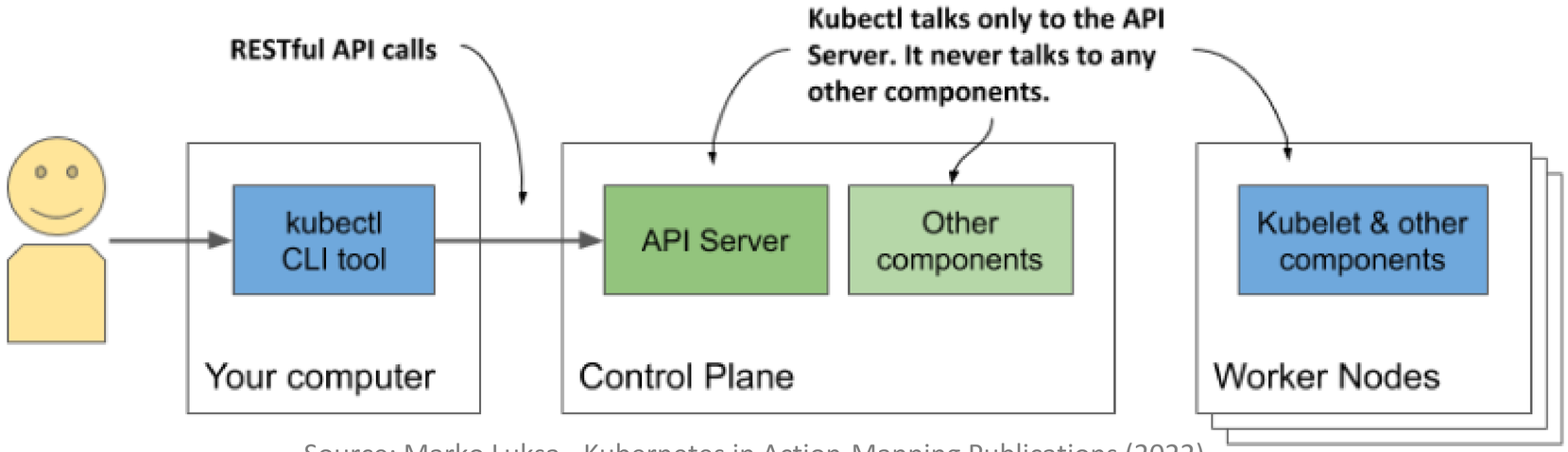
How Kubernetes runs an application



Source: Marko Luksa - Kubernetes in Action-Manning Publications (2022)

Deploying your first application

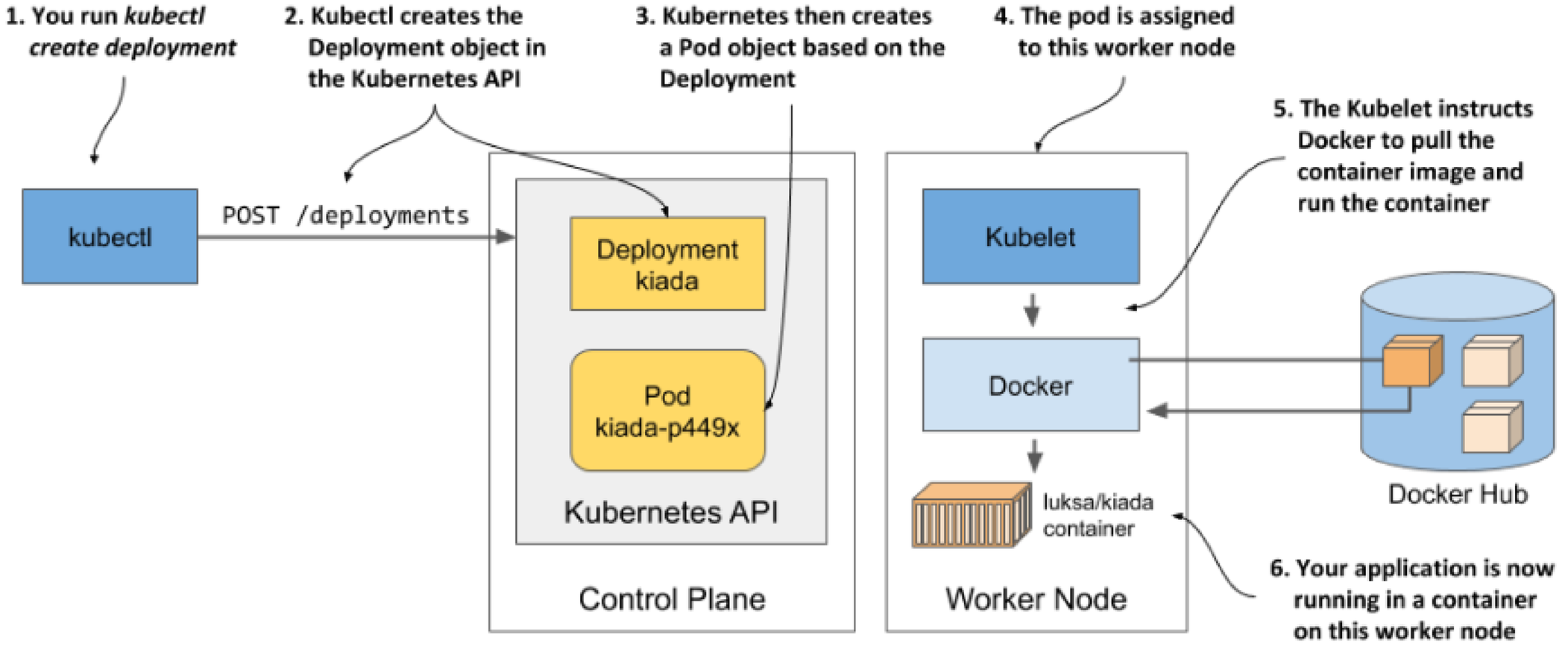
- To interact with Kubernetes, you use a command-line tool called **kubectl**, pronounced *kube-control*, *kube-C-T-L* or *kube-cuddle*.



Source: Marko Luksa - Kubernetes in Action-Manning Publications (2022)

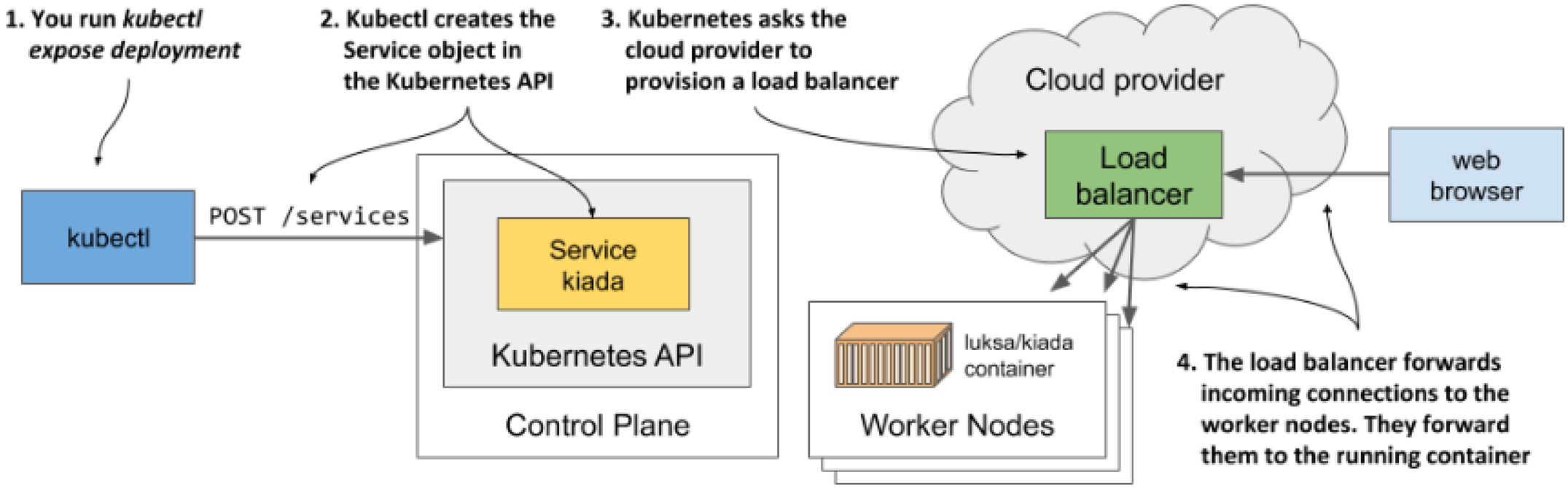
- Usually, to deploy an application, you'd prepare a JSON or YAML file describing all the components that your application consists of and apply that file to your cluster -> **declarative approach**

- The interaction with Kubernetes consists mainly of the creation and manipulation of objects via its API.



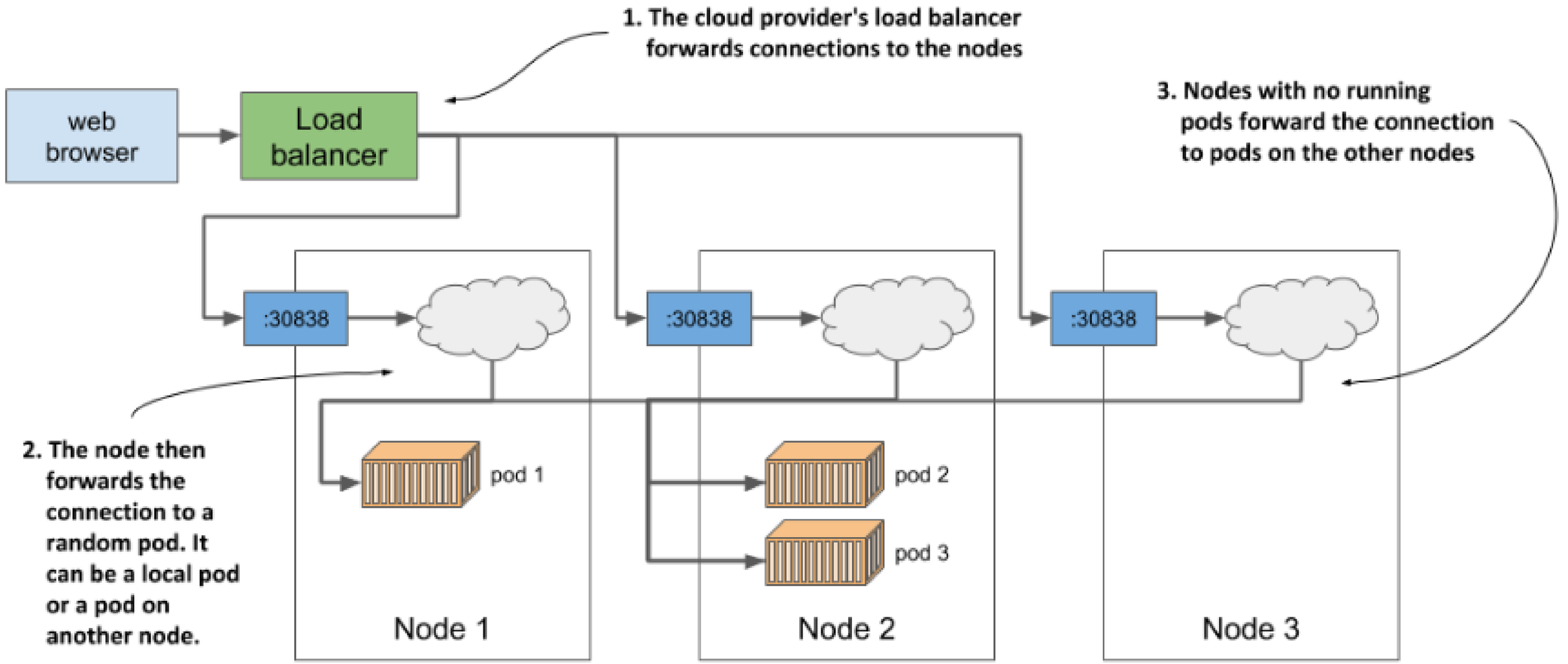
Source: Marko Luksa - Kubernetes in Action-Manning Publications (2022)

- To make the pod accessible externally, you'll expose it by creating a Service object.



Source: Marko Luksa - Kubernetes in Action-Manning Publications (2022)

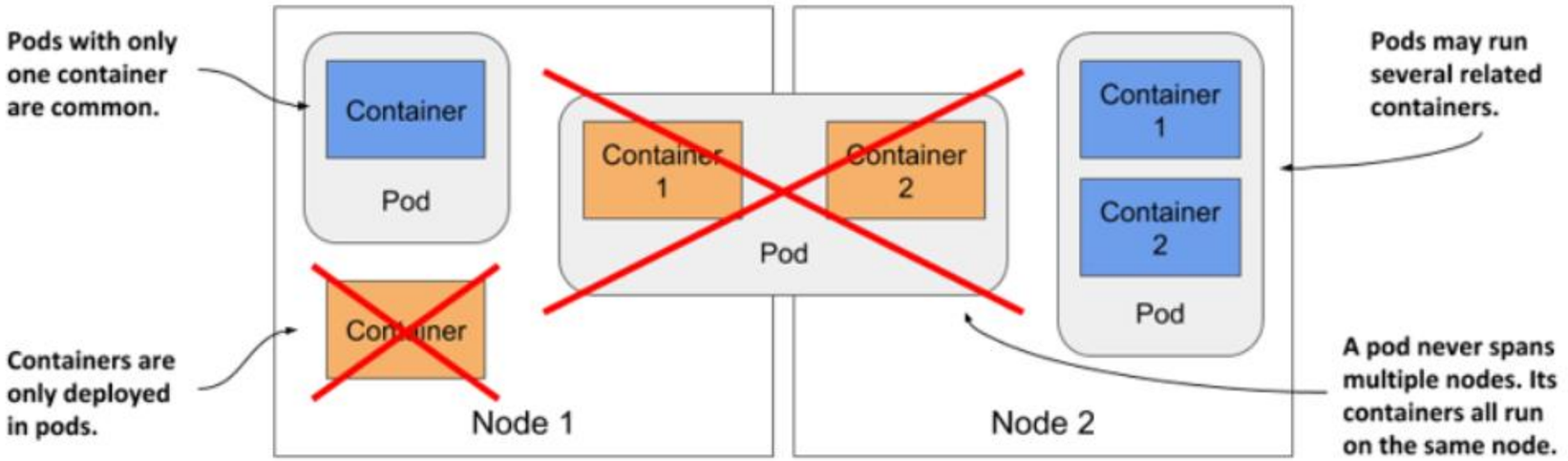
- One of the major benefits of running applications in containers is the ease with which you can scale your application deployments.



Source: Marko Luksa - Kubernetes in Action-Manning Publications (2022)

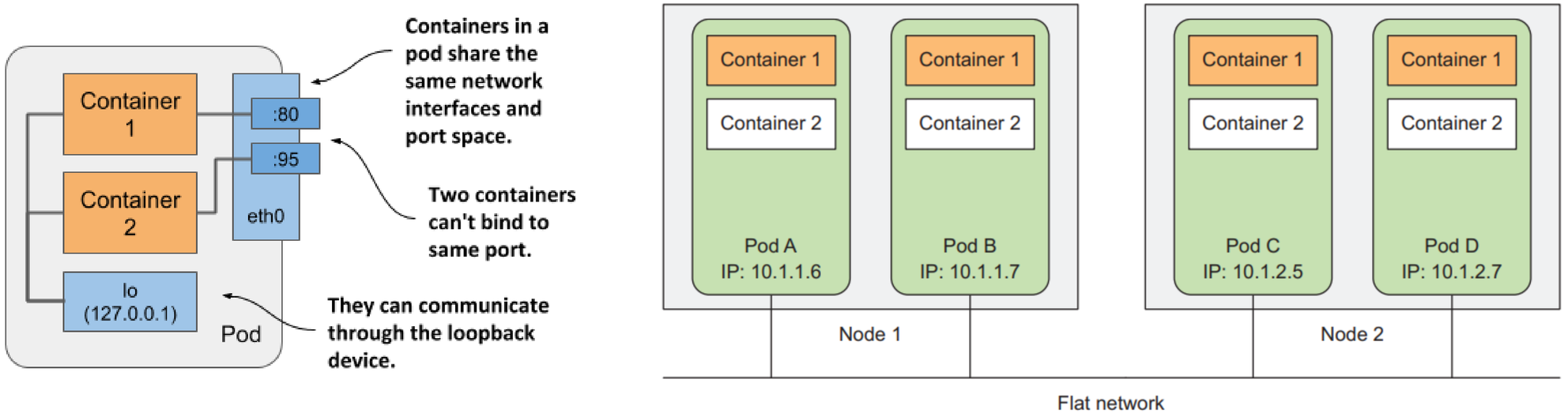
Pods

- Pod is a co-located group of containers and represents the **basic building block** in Kubernetes.
- You always deploy and operate on a pod of containers.
  - it’s common for pods to contain only a single container
- When a pod does contain multiple containers, all of them are always run on a single worker node



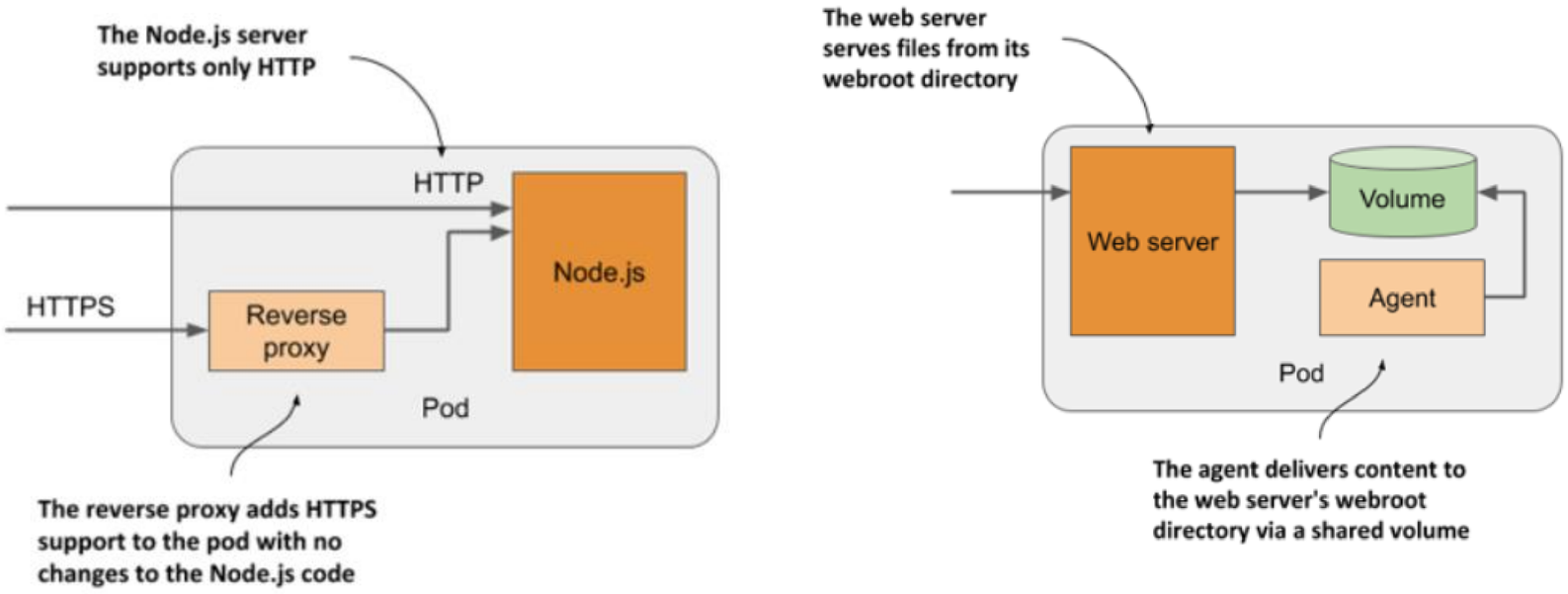
Vir: Marko Luksa - Kubernetes in Action-Manning Publications (2018)

- Containers in a pod share the same IP address and port space
  - processes running in containers of the same can’t bind to the same port numbers
- Container can communicate with other containers in the same pod through localhost
- All pods in a Kubernetes cluster reside in a single flat, shared, network-address space.



Vir: Marko Luksa - Kubernetes in Action-Manning Publications (2018)

- **Sidecar containers**
  - Multiple containers into a single pod -> when the application consists of one main process and one or more complementary processes
  - Sidecar containers include log rotators and collectors, data processors, communication adapters, and others



Source: Marko Luksa - Kubernetes in Action-Manning Publications (2018)

Install Kubernetes

- Decide whether you want to run them **locally**, with one of the major **cloud** providers, in a **hybrid** cloud solution.
- **Managing Kubernetes yourself:**
  - If you ask anyone in the Kubernetes community if this is a good idea, you'll usually get a very **definite "no".**
  - Kubernetes brings with it an **enormous amount of additional complexity**. Anyone who wants to run a Kubernetes cluster must be intimately familiar with its inner workings.
  - The management of production-ready Kubernetes clusters is a multi-billion-dollar industry.
  - On the other hand, trying out Kubernetes for non-production use-cases or using a managed Kubernetes cluster is much less problematic.

Kubernetes vanilla open-source vs enterprise Kubernetes

- **Using a vanilla version of Kubernetes:**
  - The open-source version of Kubernetes is maintained by the community and represents the cutting edge of Kubernetes development.
  - This also means that it may not be as stable as the other options.
  - It may also lack good security defaults.
  - Deploying the vanilla version requires a lot of fine tuning to set everything up for production use.
- **Using enterprise-grade Kubernetes distributions:**
  - A better option for using Kubernetes in production is to use an enterprise-quality Kubernetes distribution such as OpenShift or Rancher.
  - In addition to the increased security and performance provided by better defaults, they offer additional object types in addition to those provided in the upstream Kubernetes API.
  - These commercial Kubernetes distributions usually lag one or two versions behind the upstream version of Kubernetes. It’s not as bad as it sounds. The benefits usually outweigh the disadvantages.

Deploying Kubernetes in the cloud

- The advantage that you can **scale your cluster at any time** at short notice if required. -> **elasticity of the cluster is certainly one of the main benefits of running Kubernetes in the cloud.**
- Using a managed Kubernetes cluster in the cloud:
  - Using Kubernetes is ten times easier than managing it.
  - Most major cloud providers now offer Kubernetes-as-a-Service. They take care of managing Kubernetes and its components while you simply use the Kubernetes API like any of the other APIs the cloud provider offers.
- The top managed Kubernetes offerings include the following:
  - Google Kubernetes Engine (GKE)
  - Azure Kubernetes Service (AKS)
  - Amazon Elastic Kubernetes Service (EKS)
  - IBM Cloud Kubernetes Service
  - Red Hat OpenShift Online and Dedicated
  - VMware Cloud PKS
  - Alibaba Cloud Container Service for Kubernetes (ACK)

Amazon EKS

Azure Kubernetes Service (AKS)

Should you even use Kubernetes?

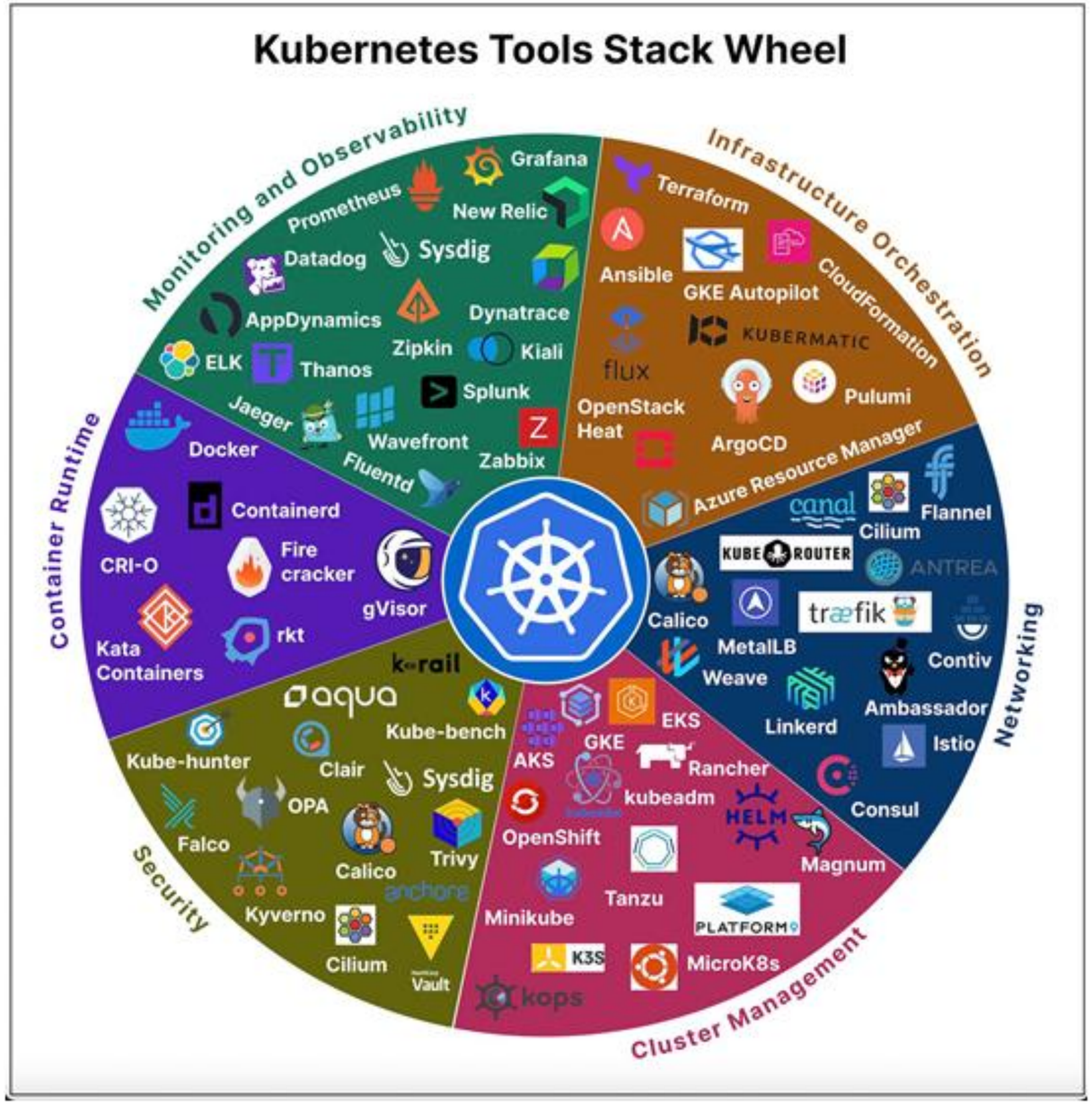
- **Do your workloads require automated management?**
  - If your application is a large monolith, you definitely don’t need Kubernetes.
  - Even if you deploy microservices, using Kubernetes may not be the best option, especially if the number of your microservices is very small.
  - But if your system consists of less than five microservices, throwing Kubernetes into the mix is probably not a good idea.
  - If your system has more than twenty microservices, you will most likely benefit from the integration of Kubernetes.
- **Can you afford to invest your engineers’ time into learning Kubernetes?**
  - While the applications themselves don’t need to be modified to run in Kubernetes, development engineers will inevitably spend a lot of time learning how to use Kubernetes, even though the operators are the only ones that actually need that knowledge.
- **Are you prepared for increased costs in the interim?**
  - While Kubernetes reduces long-term operational costs, introducing Kubernetes in your organization initially involves increased costs for training, hiring new engineers, building and purchasing new tools and possibly additional hardware.
  - Kubernetes requires additional computing resources in addition to the resources that the applications use.
- **Don’t believe the hype**
  - The initial excitement has just begun to calm down, but many engineers may still be unable to make rational decisions about whether the integration of Kubernetes is as necessary as it seems.

Kubernetes security report

- StackRox - State of Kubernetes and Container Security Report, Winter 2020
  - Nearly half the organizations have delayed deploying containerized apps into production due to security concerns.
  - Only 6% of organizations have avoided any security incident in their container and Kubernetes environments (69% misconfiguration).
  - Despite maturing container strategies, security remains the number one concern with container strategies.
  - Adoption of managed Kubernetes offerings from public cloud providers sees massive growth.

Case study: Urb-it, 8 Years Kubernetes In Production

- Wanted to join the cloud-native strategy trend.
- Migrating From Self-Managed To Managed Kubernetes -> we have never regretted the move
- Learnings:
  - Kubernetes Is Complex
  - Kubernetes Certificates (expiration dates)
  - Keep Kubernetes & Helm Up To Date
  - Disaster Recovery Plan -> make sure to have ways to recreate the cluster if needed
  - Backup Of Secrets (If your cluster goes away, all your secrets will be gone.)
  - Vendor-Agnostic VS “Go All In” -> While being vendor-agnostic is a great idea, for us, it came with a high opportunity cost.
  - Pre-Scaling During Known Peaks -> Even with the auto-scaler, we sometimes scaled too slowly.
  - Pick The Right Node Type
  - Ensure you track the usage of memory, CPU, etc., over time so you can observe how your cluster is performing.
  - Having all logs consolidated in one place, along with a robust trace ID strategy (e.g. OpenTelemetry or similar), is crucial for any microservices architecture.
  - Security
- **But in the long run, and especially in the latest years, it has proven to provide great value for us.**



Source: ByteByteGo

Emergent trends and landscape overview

108

Cloud Native

- Cloud native is the software approach of building, deploying, and managing modern applications in cloud computing environments.
- Modern companies want to build highly scalable, flexible, and resilient applications that they can update quickly to meet customer demands.
- **What are cloud-native applications?** Software programs that consist of multiple small, interdependent services called microservices.
- **What is cloud-native application architecture?** Immutable infrastructure, Microservices, API, Service mesh, Containers
- **What is cloud-native application development?** Continuous integration, Continuous delivery, DevOps, Serverless

CNCF

- The Cloud Native Computing Foundation (CNCF) is an open-source foundation that helps organizations kick start their cloud-native journey.
- Established in 2015, the CNCF supports the open-source community in developing critical cloud-native components, including Kubernetes.
- Part of the **Linux Foundation**

26 GRADUATED PROJECTS

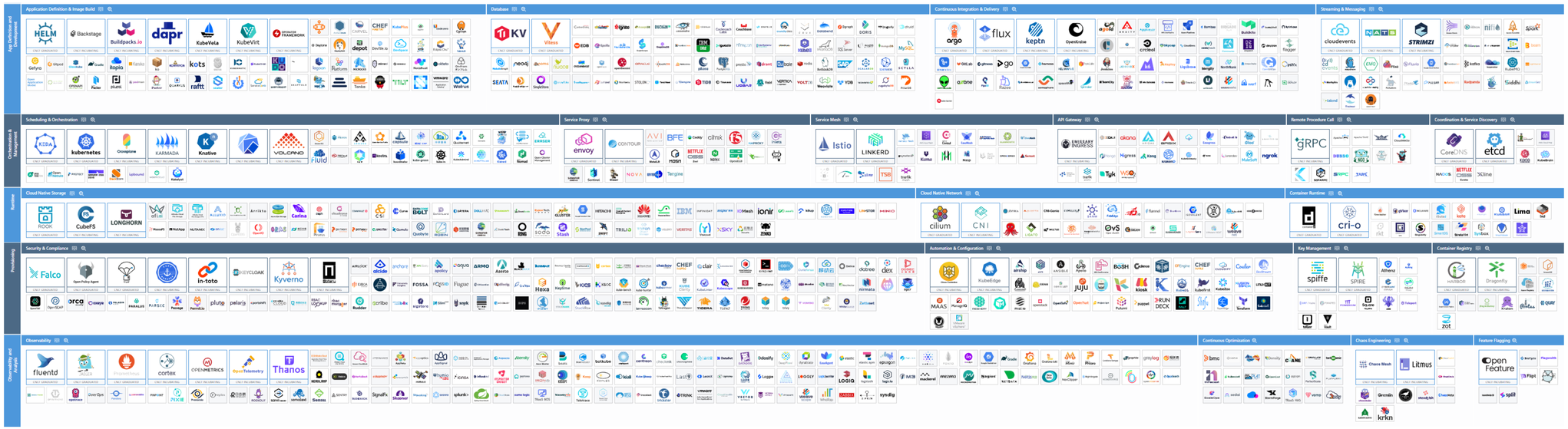
36 INCUBATING PROJECTS

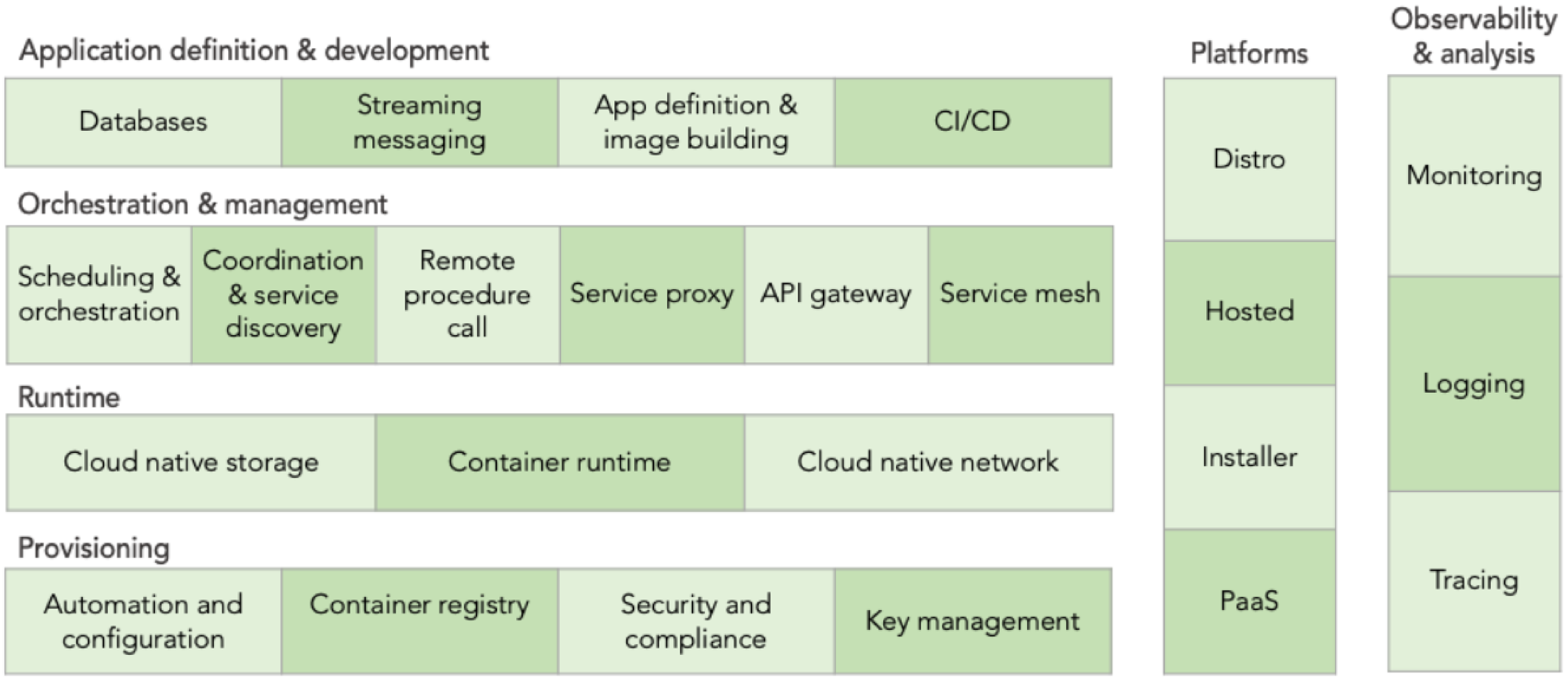
113 SANDBOX PROJECTS

CLOUD NATIVE
COMPUTING FOUNDATION

Cloud native landscape

- The number of project can be overwhelming
- **Cloud native landscape map -> Here to help!**
- The goal of the cloud native landscape is to **compile and organize all cloud native open source projects** and proprietary products into categories, providing an overview of the current ecosystem.





Source: https://thenewstack.io/cloud-native/an-introduction-to-the-cloud-native-landscape/

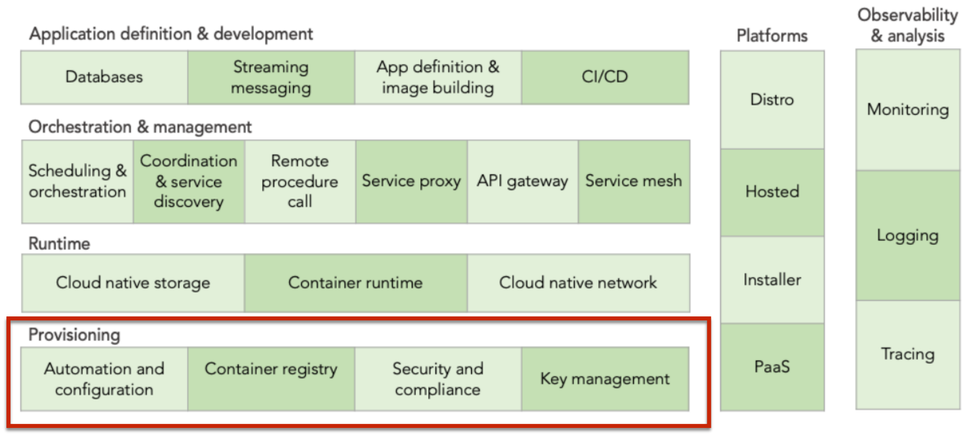
Cloud native stack

- **Provisioning**: tools that are used to *create and harden* the foundation on which cloud native apps are built
- **Runtime:** everything a container needs to run in a cloud native environment
- **Orchestration & Management:** tooling to handle running and connecting your cloud native applications
- **App Definition and Development**: tools that enable engineers to build apps
- **Observability and Analysis**

- **Platform:** Platforms bundle different tools from different layers together, solving a larger problem.

Provisioning

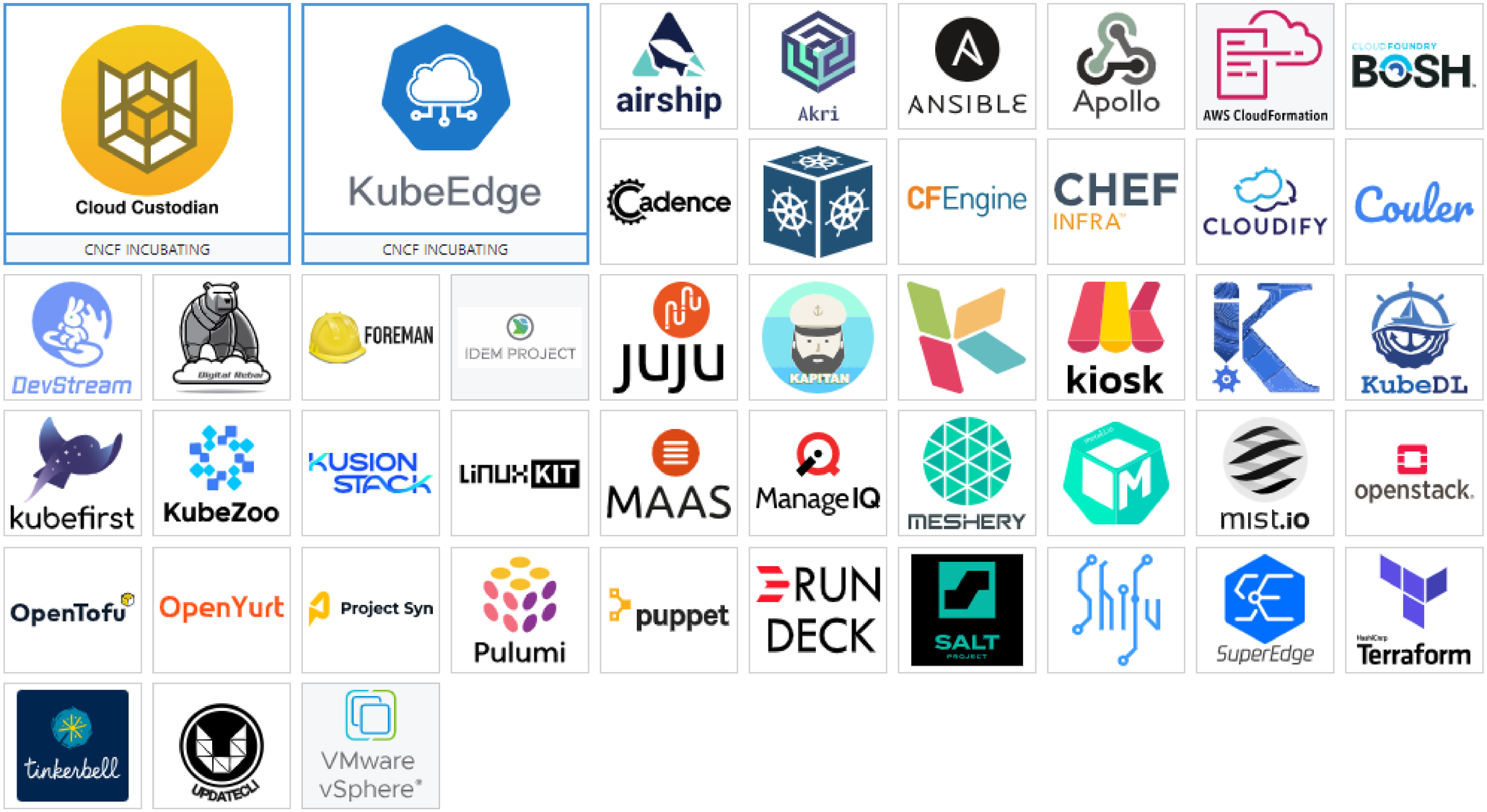
- Provisioning is the process of creating and setting up IT infrastructure, and includes the steps required to manage user and system access to various resources.
- **Topics:**
  - Automation & Configuration
  - Container Registry
  - Security & Compliance
  - Key Management



Source: https://www.servicenow.com/products/it-asset-management/what-is-provisioning.html

Automation & Configuration

- Speed up the **creation and configuration of compute resources** (virtual machines, networks, firewall rules, load balancers, etc.).
- Traditionally, IT processes relied on lengthy and labor intensive manual release cycles -> To be compatible with cloud native's **rapid development cycles**, infrastructure must be provisioned **dynamically and without human intervention**.
- Reducing the required work to provision resources through automation
- You'll also need a subset of these tools to create and manage the Kubernetes clusters themselves.


Cloud Custodian
CNCF INCUBATING
KubeEdge
CNCF INCUBATING
airship
Akri
ANSIBLE
Apollo
AWS CloudFormation
CLOUD FOUNDRY
BOSH
Cadence
CFEngine
CHEF
INFRA
CLOUDIFY
Couler
DevStream
Digital Rebar
FOREMAN
IDEM PROJECT
JUJU
KAPITAN
kiosk
KubeDL
kubefirst
KubeZoo
KUSION STACK
LINUX KIT
MAAS
ManageIQ
MESHERY
mist.io
openstack.
OpenTofu
OpenYurt
Project Syn
Pulumi
puppet
RUN DECK
SALT PROJECT
Shifu
SuperEdge
HashiCorp
Terraform
tinkerbell
UPDATECLI
VMware vSphere

Infrastructure-as-Code (IaC)

- Infrastructure as code (IaC) is the **ability to provision and support your computing infrastructure using code** instead of manual processes and settings.
- Manual infrastructure management is time-consuming and prone to error - especially when you manage applications at scale.
- It **automates infrastructure management** so developers can focus on building and improving applications instead of managing environments.
- Tools: Terraform, Puppet, Chef, and Ansible

Container Registry

- Container registries are specialized web applications that categorize and store repositories which is a space to store images.
- Cloud native applications are packaged and run as containers. Container registries store and provide the container images needed to run these apps.
- By centrally storing all container images in one place, they are easily accessible for any developer working on that app.

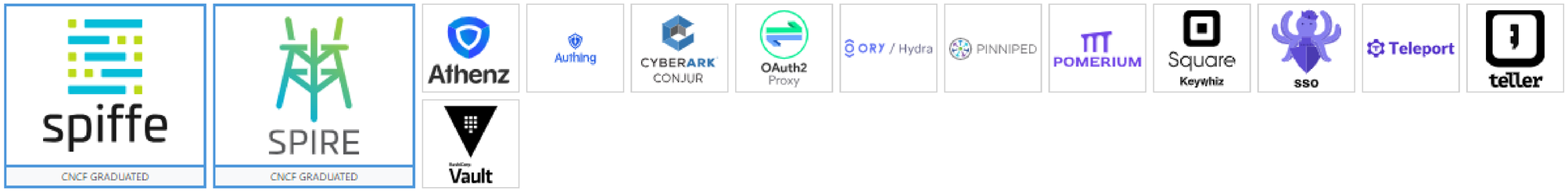
Security & Compliance

- Cloud native applications are designed to be rapidly iterated on. In order to release code on a regular cadence you must ensure that the code and operating environment are secure and only accessed by authorized engineers.
- Security and compliance tools help harden, monitor, and enforce platform and application security.
- To run containers securely:
  - containers must be scanned for known vulnerabilities
  - signed to ensure they haven't been tampered with
- Kubernetes has extremely permissive access control settings by default that are unsuitable for production use.
- Image scanning, Image signing, Policy enforcement, Audit, Certificate Management


Falco
CNCF GRADUATED
Open Policy Agent
CNCF GRADUATED
TUF
CNCF GRADUATED
CERT MANAGER
CNCF INCUBATING
in-toto
CNCF INCUBATING
KEYCLOAK
CNCF INCUBATING
Kyverno
CNCF INCUBATING
Notary
CNCF INCUBATING
AIRLOCK
alcide
anchore
APIClarity
apolicy
aqua
ARMO
Aserto
BLACKDUCK
BLOOMBASE
Bouncy Castle
CAPSULE8
cerbos
CHAITIN
Check Point
checkov
CHEF INSPEC
clair
CLOUDMATOS
CONFIDENTIAL CONTAINERS
ContainerSSH
COPA
Curiefense
移动云
Datica
datree
dex
DOSEC
EJBCA
Fairwinds Insights
FOSSA
FOSSID
Fugue
GitGuardian
Goldilocks
Grafeas
Hexa
Keylime
KICS
KSOC
kube-bench
kube-hunter
kubearmor
KUBE Clarity
KubeLinter
Kubescape
KUBEWARDEN
matano
Metarget
mondoo
默安科技 MoreSec
NeuVector
nirmata
opcr
OpenFGA
OpenSCAP
orca security
oxeye
PALADIN CLOUD
PARALUS
PARSEC
Passage
Permit.io
pluto
polaris
portshift
PRISMA CLOUD
青藤云安全
RBAC LOOKUP
rbac manager
Rudder
scribe
secure codebox
sigstore
Slim toolkit
snyk
Sonatype Nexus
SONOBUOY
SOPS
SPYDERBAT
STACKHAWK
StackRox
sysdig SECURE
探真科技 TensorSecurity
terrascan
Tetragon
ThreatMapper
TIGERA
TOPAZ
TREND MICRO
trivy
trivy
VEINMIND
VM Clarity
WhiteSource
Zettaset

Key Management

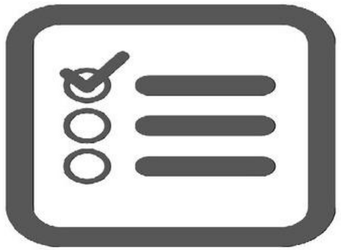
- **What is a key?** A key is a string of characters used to encrypt or sign data.
- The tools and projects from how to **securely store passwords** and other **secrets** (sensitive data such as API keys, encryption keys, etc.) to how to safely **eliminate** passwords and secrets from your microservices environment.
- The secret distribution must be **entirely programmatic** (no humans in the loop) and automated.



- Applications need to know if a given request comes from a valid source (authentication - **AuthN**) and if that request has the right to do whatever it's trying to do (authorization - **AuthZ**)
- Key generation, storage, management, and rotation
  - Vault, for example, is a rather generic key management tool allowing you to manage different types of keys.
- Single sign-on and identity management
  - Keycloak, on the other hand, is an identity broker which can be used to manage access keys for different services.

Authentication (AuthN)

Is it really you?

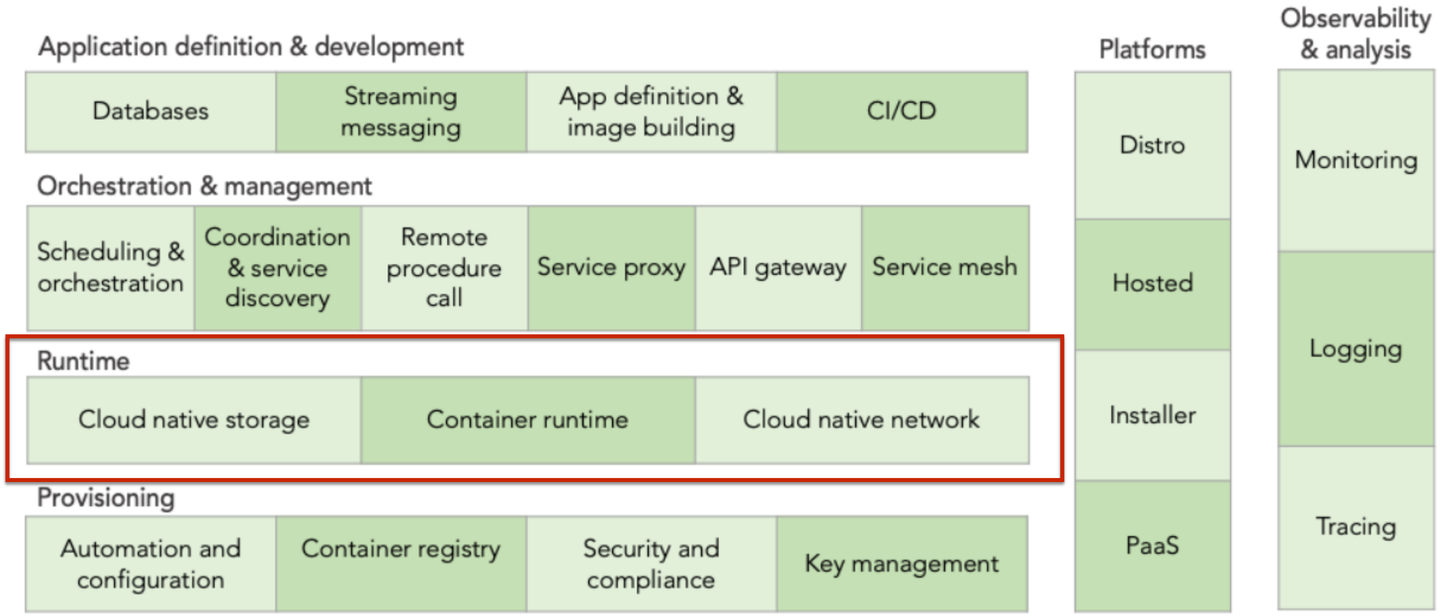


Authorization (AuthZ)

Who you are and what you can do

Runtime

- Runtime encompasses everything a container needs to run in a cloud native environment.
- That includes the code used to **start a container**, tools to make **persistent storage available** to containers; and those that manage the container **environment networks**.
- **Topics:**
  - Cloud Native Storage
  - Container Runtime
  - Cloud Native Network



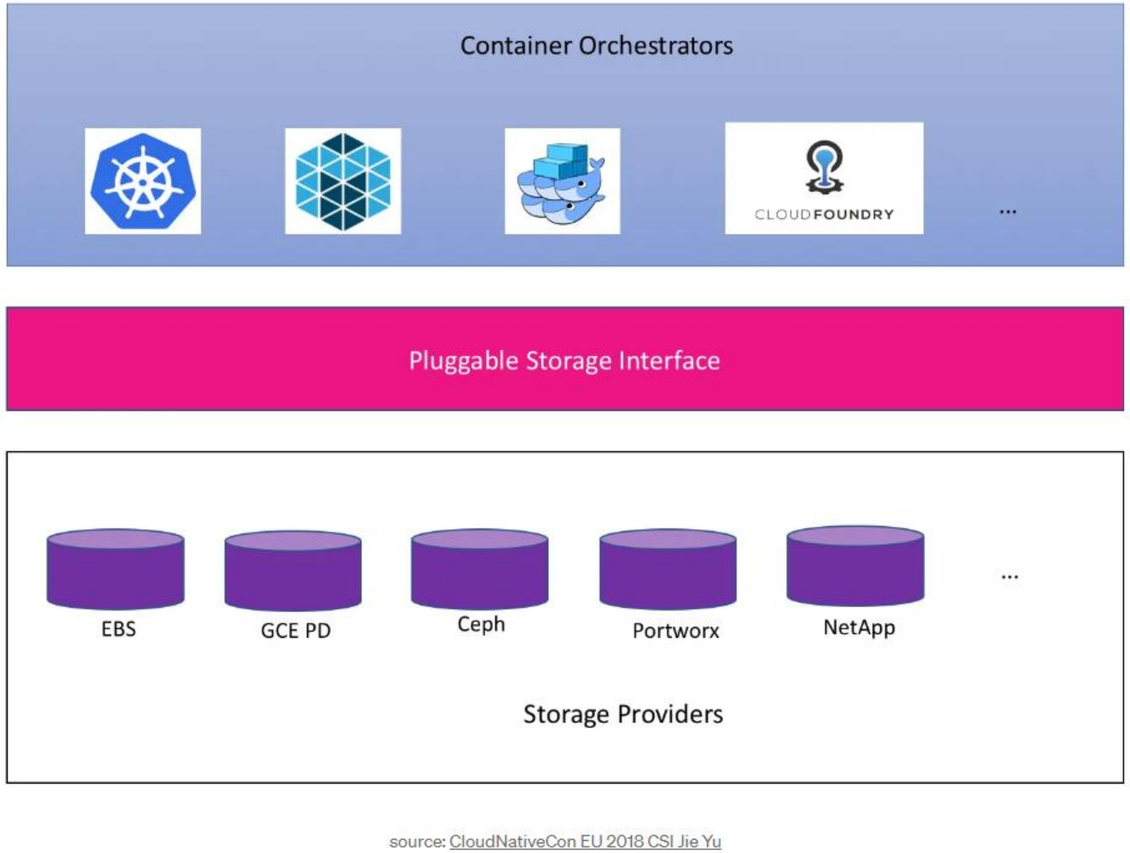
Cloud Native Storage

- **Storage** is where **the persistent data of an app is stored**, often referred to as a persistent volume.
- **Persistent data** -> things like databases, messages, or any other information we want to ensure doesn't disappear when an app gets restarted.
- Cloud native architectures are fluid, flexible, and elastic, making persisting data between restarts challenging.
- To benefit from the elasticity of the cloud, storage **must be provisioned automatically**.
- Cloud native storage is largely made possible by the **Container Storage Interface (CSI).**
- Examples: MinIO (S3-compatible API for object storage), Velero (backup solution)


ROOK
CNCF GRADUATED
CubeFS
CNCF INCUBATING
LONGHORN
CNCF INCUBATING
afi.ai
Alibaba Cloud File Storage
Alibaba Cloud File Storage CPFS
ALLUXIO
Amazon Elastic Block Store (EBS)
Arrikto
Azure Disk Storage
Carina
ceph
cloudcasa by Catalogic
COMMVAULT
CSI
Curve
DATACORE BOLT
DatenLord
DATERA
DELL EMC
DIAMANTI
DriveScale
ExponTech
GLUSTER
Google Persistent Disk
HITACHI
Hewlett Packard Enterprise
HUAWEI
HwameiStor
IBM
INFINIDAT
inspur 浪潮
IOMesh
ionir
JuiceFS
k8up
KANISTER
KASTEN by Veeam
LINSTOR
MINIO
MooseFS
NetApp
NUTANIX
ondat
OpenEBS
OpenIO
ORAS
鹏云网络
Piraeus
portworx by Pure Storage
portworx
QINGSTOR
Qumulo
Quobyte
ROBIN
彬岩数据 SANDSTONE
SANGFOR 深信服科技
ScaleFlash
RING
soda foundation
Stash
StorPool DISTRIBUTED STORAGE
SWIFT
TRILIO
TRITON Object Storage
VELERO
VERITAS
Vineyard
XSKY
焱融云 YAN RONG
YRCloudFile
ZENKO

Container Storage Interface (CSI)

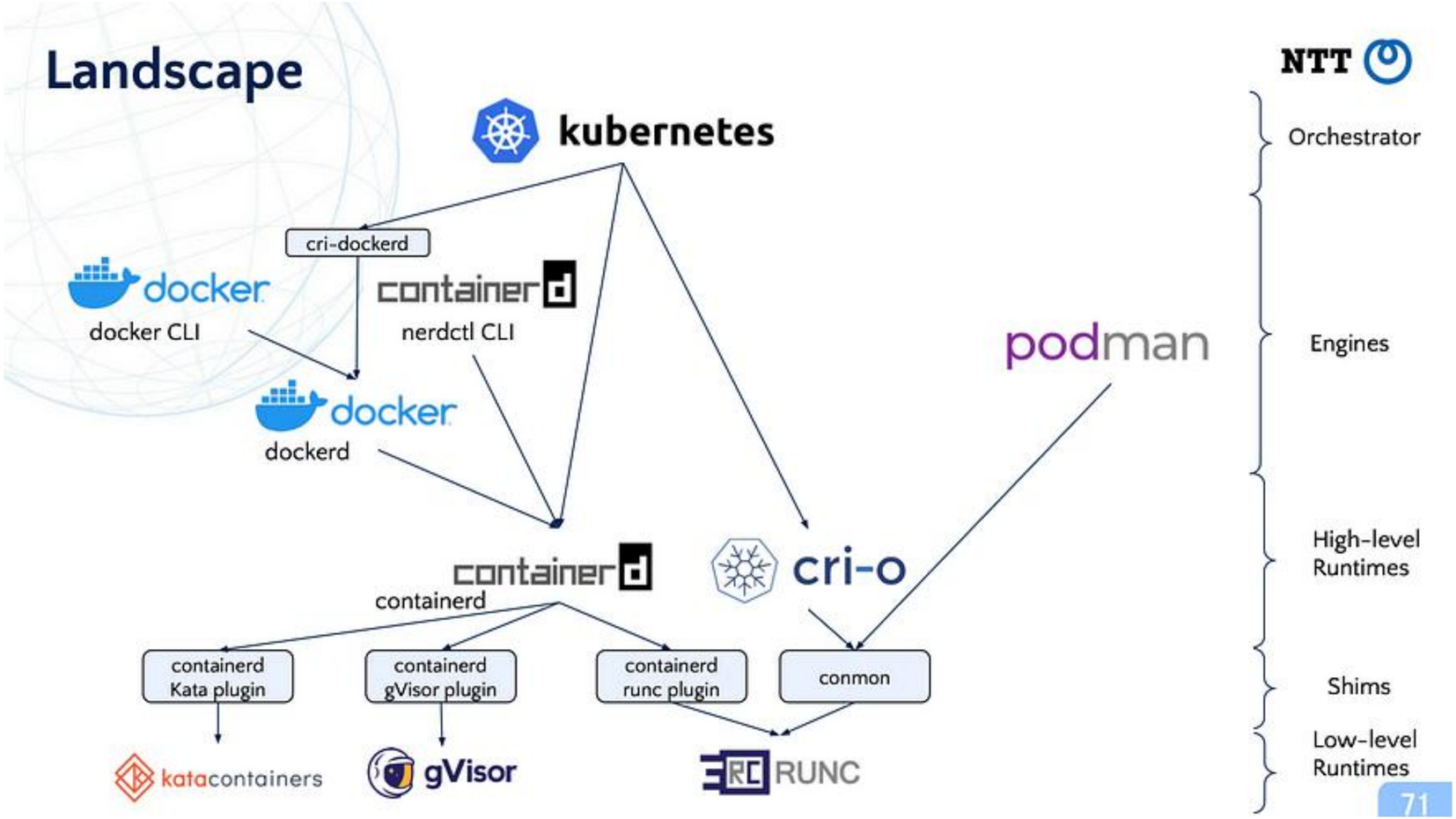
- Container Storage Interface (CSI) is **an initiative to unify the storage interface** of Container Orchestrator Systems (COs) like Kubernetes, Mesos, Docker swarm, cloud foundry, etc.
- How to write a Container Storage Interface (CSI) plugin



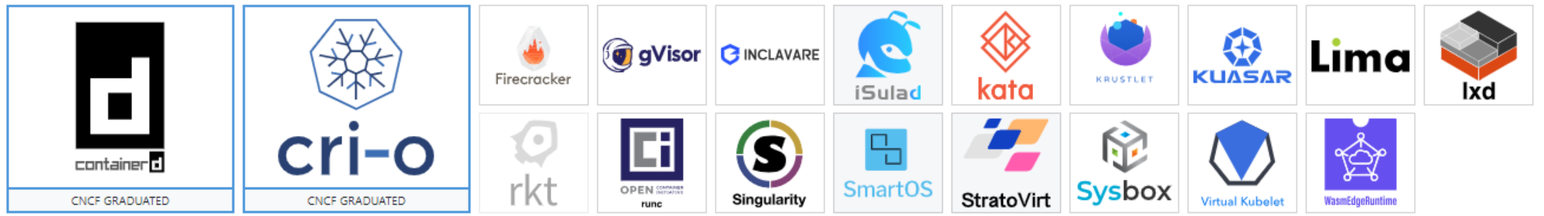
source: CloudNativeCon EU 2018 CSI Jie Yu

Container Runtime

- The **container runtime** is the software that **executes containerized** applications.
- The runtime will start an app within a container and provide it with the needed resources.
- Container images (the files with the application specs) must be launched in a standardized, secure, and isolated way.
- Examples:
  - Runtimes like **Containerd** (part of the famous Docker product) and **CRI-O** are standard container runtime implementations.
  - **gVisor** provides an additional security layer between containers and the OS.
  - **Kata** allows you to run containers as VMs.



Source: https://medium.com/nttlabs/the-internals-and-the-latest-trends-of-container-runtimes-2023-22aa111d7a93

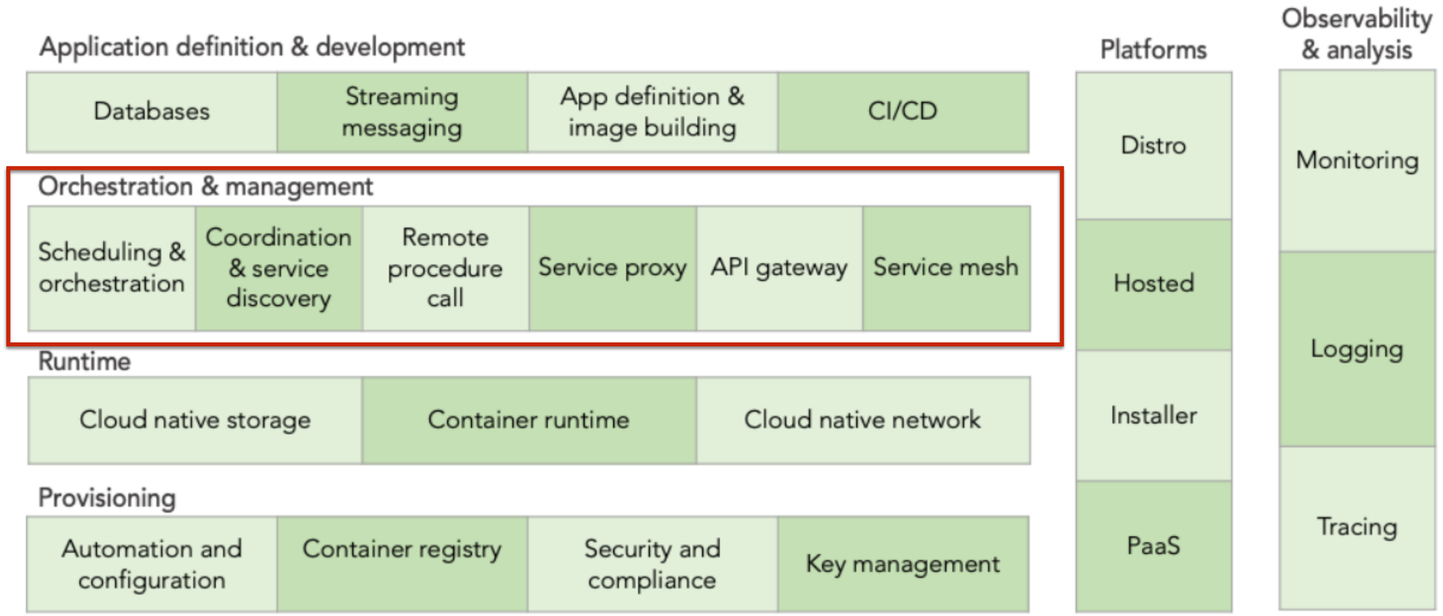


Cloud Native Network

- Containers need to communicate with each other privately.
- Container Network Interface (CNI)
- At a minimum, a container network needs to assign IP addresses to pods allowing other processes to access it.

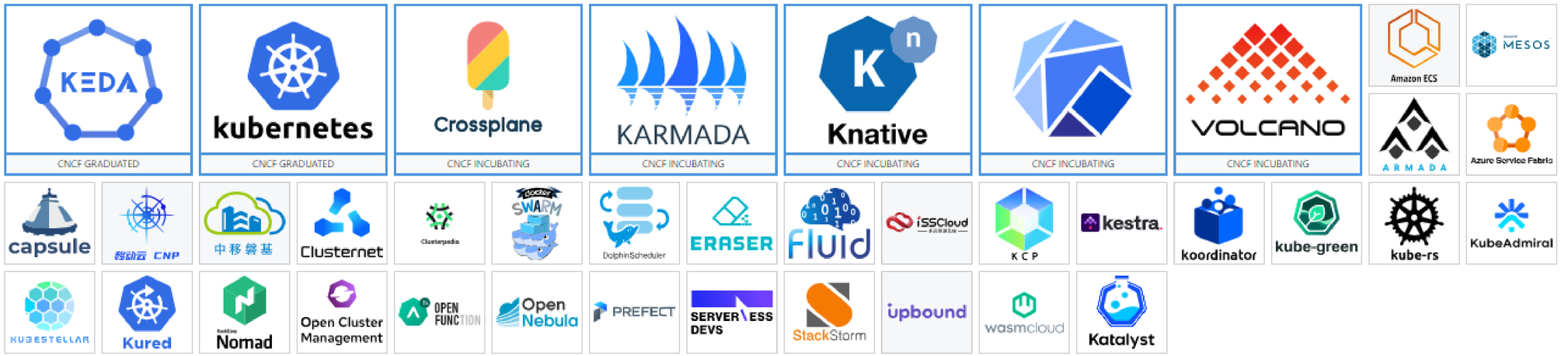
Orchestration & Management

- Tooling to handle **running and connecting your cloud native applications**.
- Inherently scalable, cloud native apps rely on automation and resilience, enabled by these tools.
- **Topics:**
  - Scheduling & Orchestration
  - Coordination & Service Discovery
  - Remote Procedure Call
  - Service Proxy
  - API Gateway
  - Service Mesh



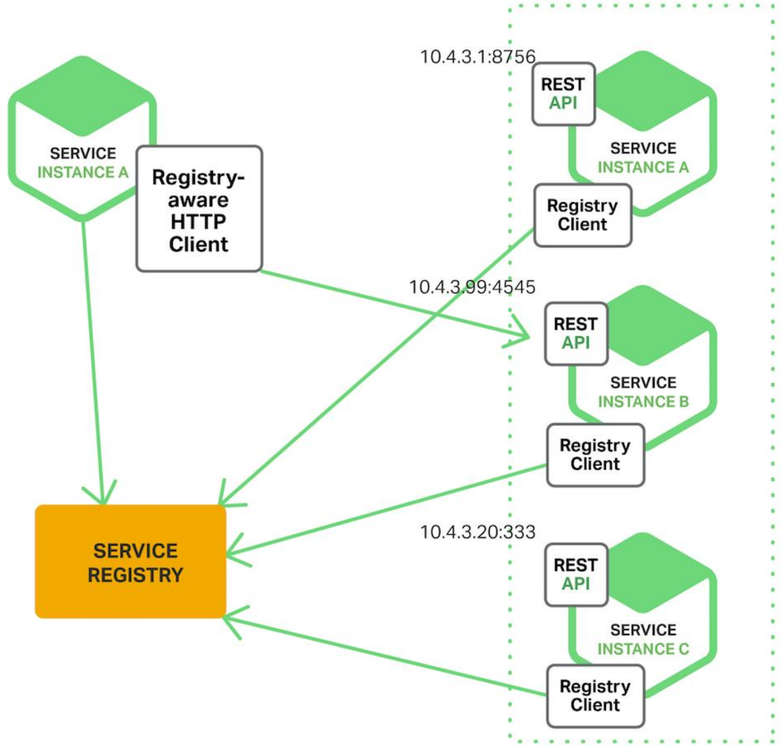
Scheduling & Orchestration

- Orchestration and scheduling refer to **running and managing containers across a cluster.**
- Kubernetes lives in the orchestration and scheduling section along with other less widely adopted orchestrators like Docker Swarm and Mesos.



Coordination & Service Discovery

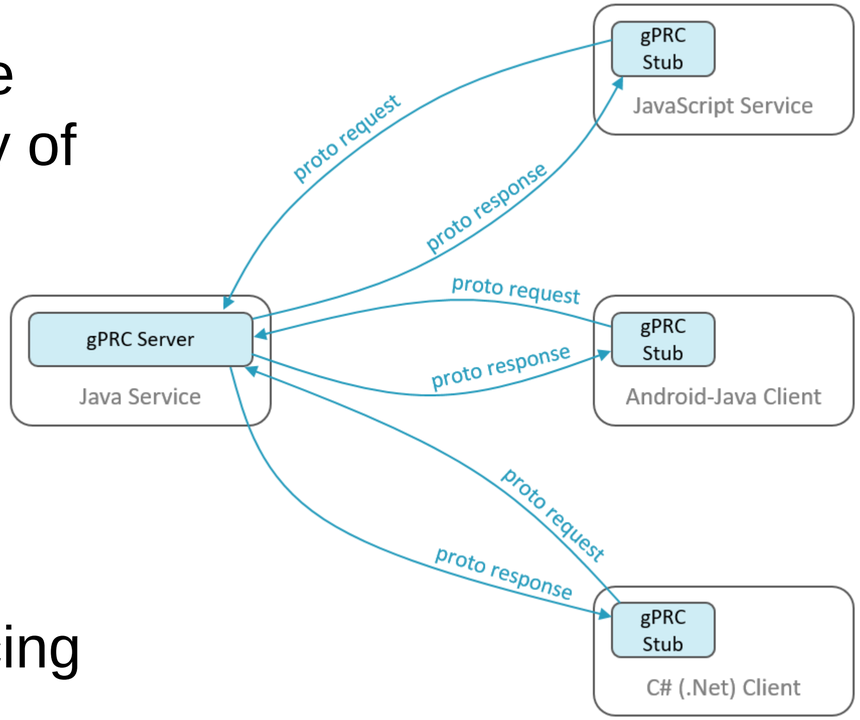
- Cloud native architectures are dynamic and fluid, meaning they are constantly changing.
- Tools in this category keep track of services within the network so services can find one another when needed.
  - **Service discovery engines**: database-like tools that store information on all services and how to locate them
  - **Name resolution tools**: tools that receive service location requests and return network address information (e.g. CoreDNS)

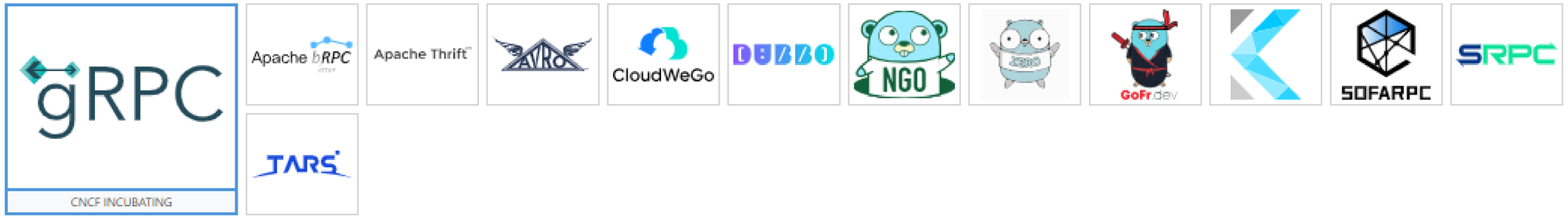


Source: https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/

Remote Procedure Call

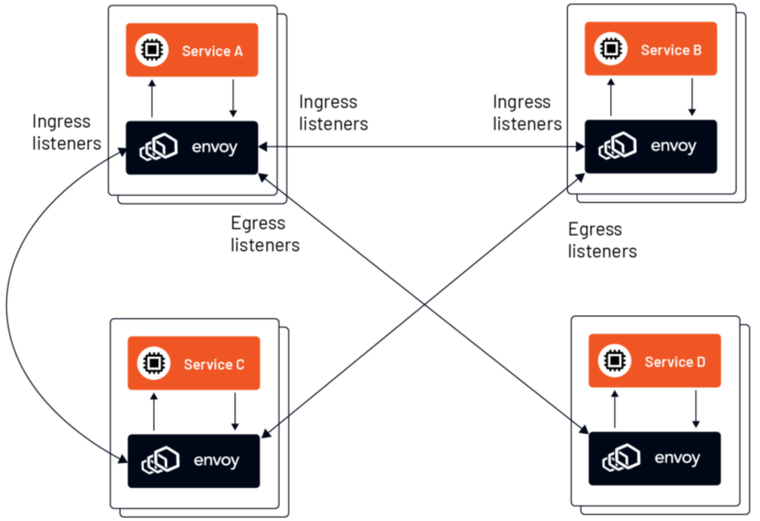
- Remote Procedure Call (RPC) is a particular technique enabling applications to talk to each other. It's one way of structuring app communication.
- RPC benefits:
  - extremely efficient use of the network layer
  - well-structured communications between services
- Criticized for creating brittle connection points and forcing users to do coordinated upgrades for multiple services.



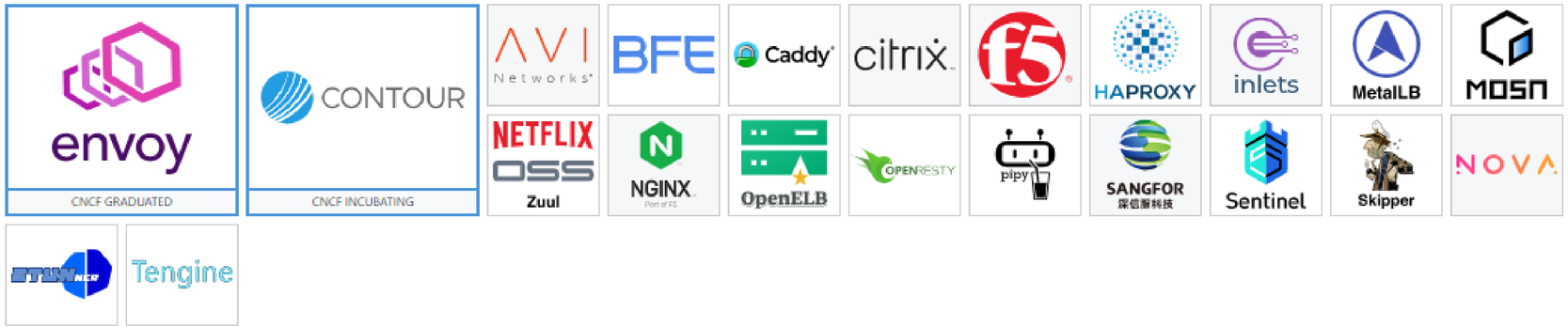


Service Proxy

- A service proxy is a tool that intercepts traffic to or from a given service, applies some logic to it, then forwards that traffic to another service.
  - Can collect information about network traffic as well as apply rules to it
- Traditionally -> code enabling data collection and network traffic management was embedded within each application. -> A service proxy "externalizes" this functionality.
- Routing, TLS termination, gather critical data, cache content

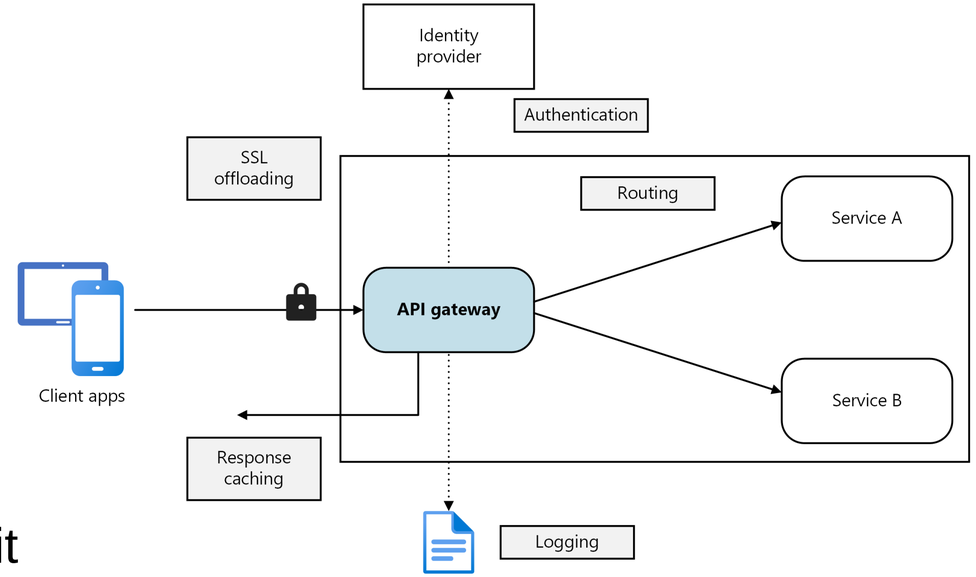


Source: https://tetrate.io/what-is-envoy-proxy/



API Gateway

- Computers interact with each other through APIs
- An API gateway allows organizations:
  - functions, such as authorizing or limiting the number of requests
  - common interface to (often external) API consumers
- An API gateway takes custom code out of our apps and brings it into a central system
- API gateways serve as a common entry point for a set of downstream applications



Microsoft Azure

Source: https://learn.microsoft.com/en-us/azure/architecture/microservices/images/gateway.png

EMISSARY INGRESS
CNCF INCUBATING

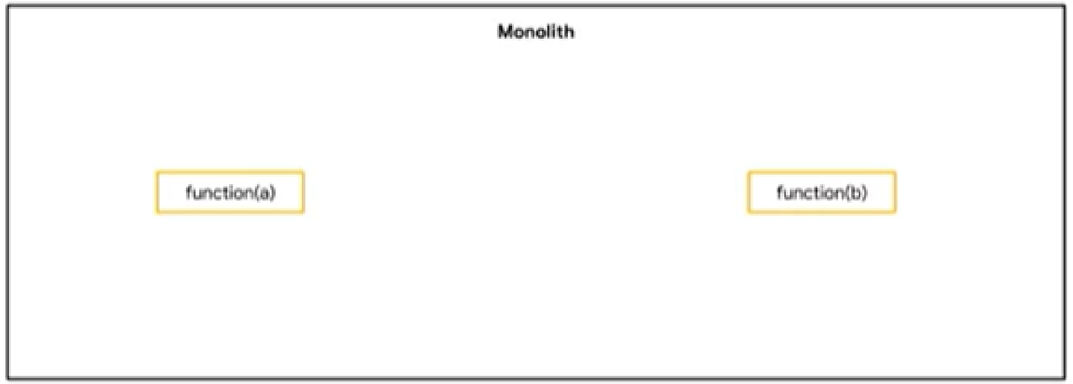
3SCALE
akana
APIOAK
APISIX
Easegress
saaras.io
Gloo
GRAVITEE.IO
Hango
Higress
Kong
krakenD
KubeGateway
KUSK
lunar.dev
MuleSoft
ngrok
Reactive Interaction Gateway
træfik proxy
Tyk
WSO2 API Microgateway

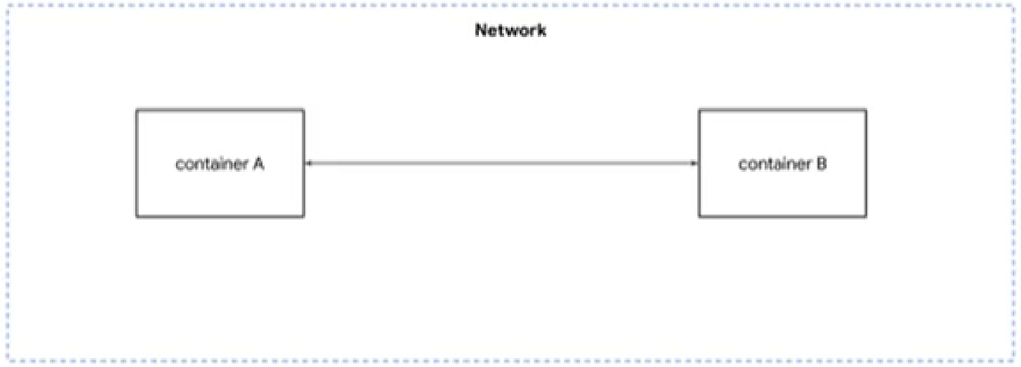
Service Mesh

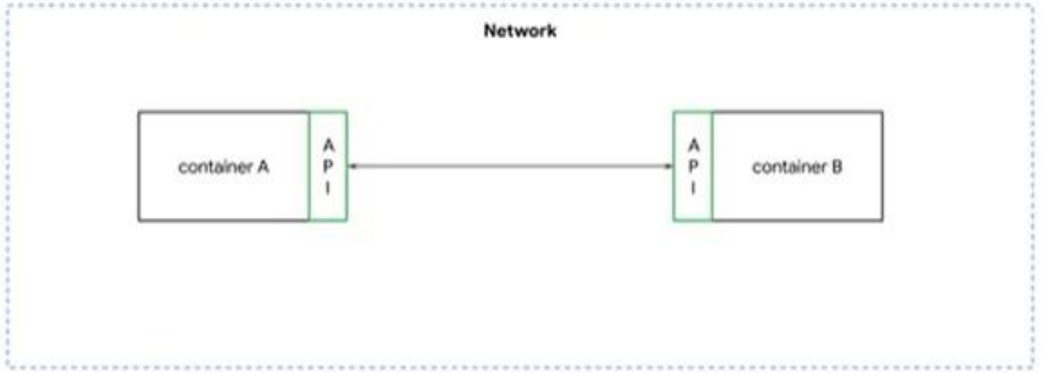
- Service meshes **manage traffic (i.e. communication) between services**. They enable platform teams to add **reliability**, **observability**, and **security** features uniformly across all services running within a cluster without requiring any code changes.

A need for a service mesh

- A microservice architecture requires considerably more management, monitoring and security.

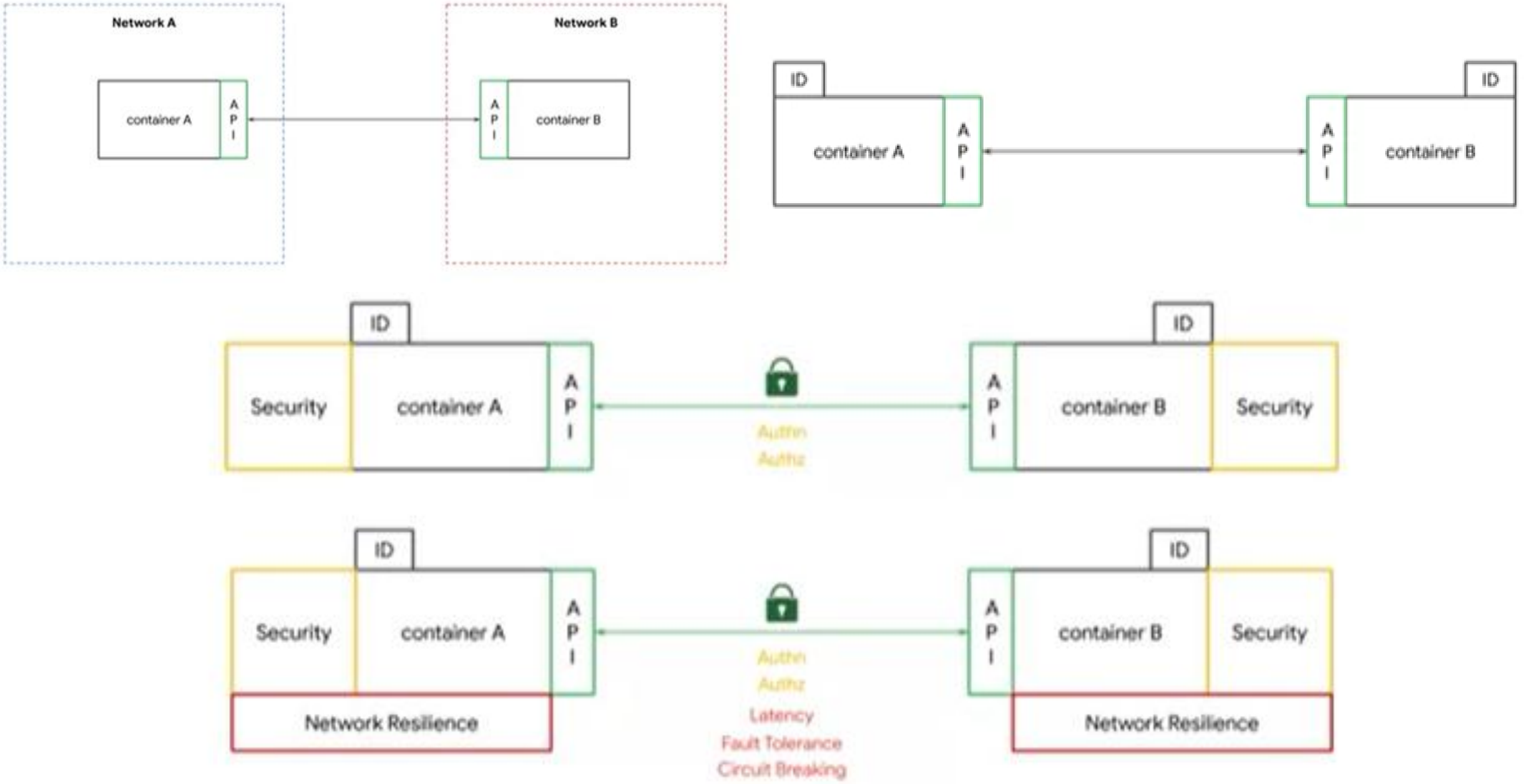




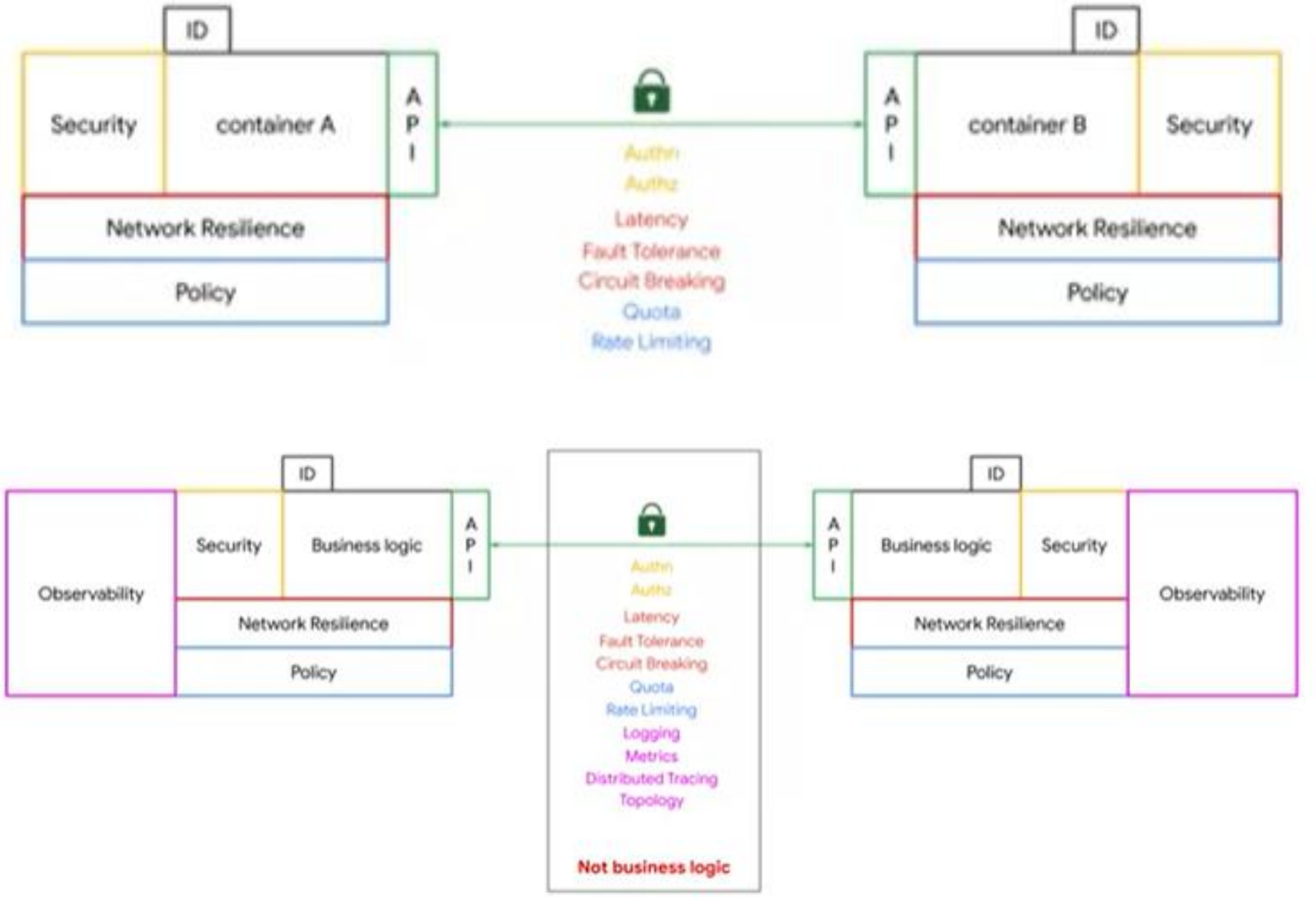


Source: https://www.coursera.org/lecture/hybrid-cloud-infrastructure-foundations-anthos/why-a-service-mesh-60VVl

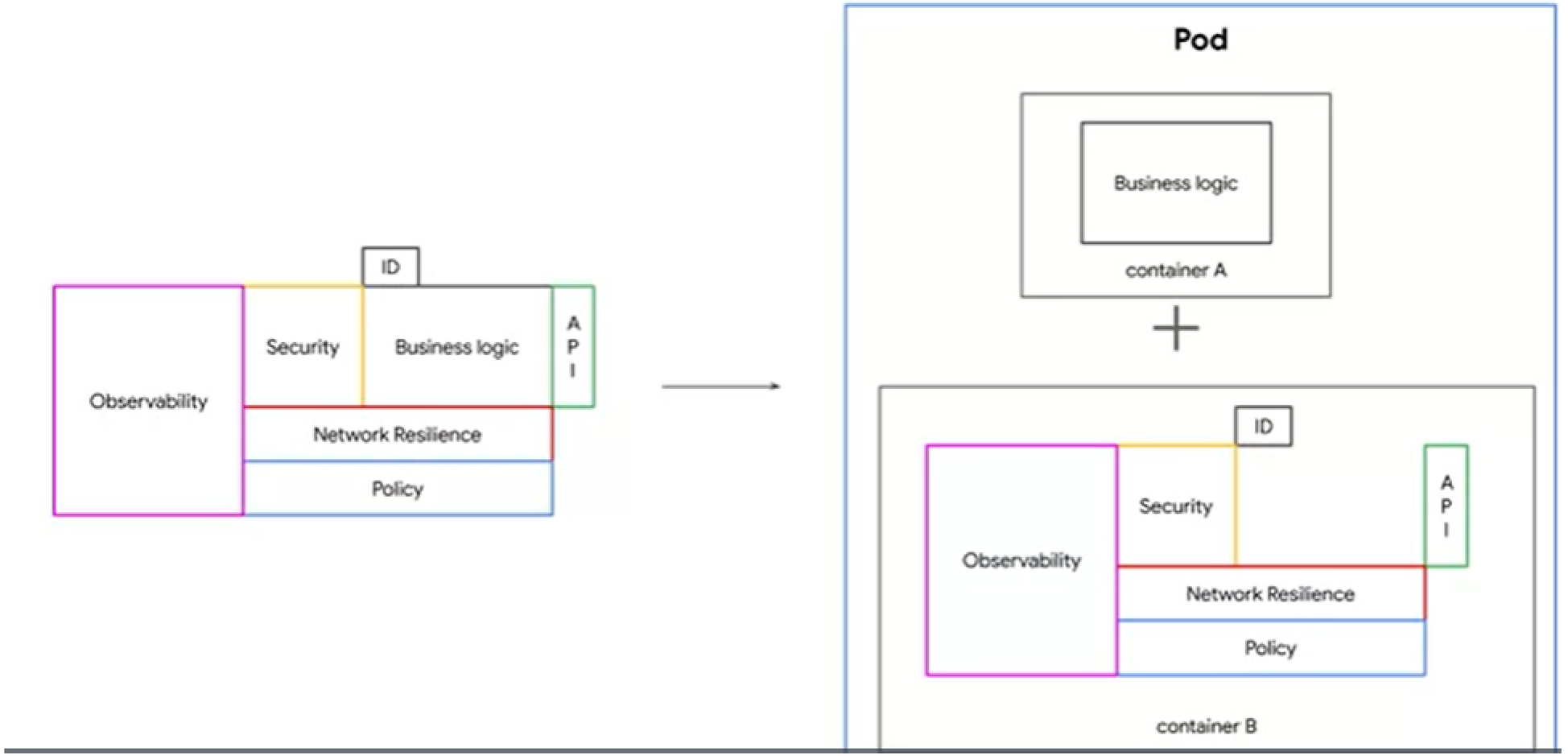
Zero trust network



Source: https://www.coursera.org/lecture/hybrid-cloud-infrastructure-foundations-anthos/why-a-service-mesh-60VVl



Source: https://www.coursera.org/lecture/hybrid-cloud-infrastructure-foundations-anthos/why-a-service-mesh-60VVl

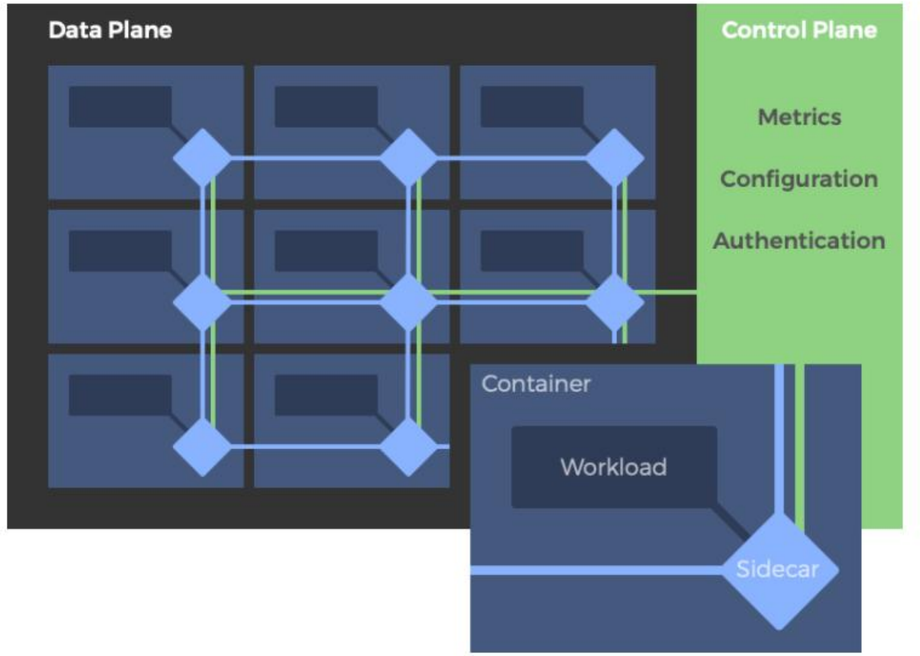


Source: https://www.coursera.org/lecture/hybrid-cloud-infrastructure-foundations-anthos/why-a-service-mesh-60VVl

Service Mesh

- **A service mesh is an infrastructure layer that handles communication between the microservices (or machines) in your backend.**
- Communication between these services **can be extremely complicated**, so the service mesh will handle tasks like:
  - **Service Discovery** - For each microservice, new instances are constantly being spun up/down. The service mesh keeps track of the IP addresses/port number of these instances and routes requests to/from them.
  - **Load Balancing** - When one microservice calls another, you want to send that request to an instance that's not busy (using round robin, least connections, consistent hashing, etc.). The service mesh can handle this for you.
  - **Observability** - As all communications get routed through the service mesh, it can keep track of metrics, logs and traces.
  - **Resiliency** - The service mesh can handle things like retrying requests, rate limiting, timeouts, etc. to make the backend more resilient.
  - **Security** - The mesh layer can encrypt and authenticate service-to-service communications. You can also configure access control policies to set limits on which microservice can talk to whom.
  - **Deployments** - You might have a new version for a microservice you're rolling out and you want to run an A/B test on this. You can set the service mesh to route a certain % of requests to the old version and the rest to the new version (or some other deployment pattern)

- A **service mesh** is a tool for adding observability, security, and reliability features to applications by inserting these features at the platform layer rather than the application layer.
- The service mesh is typically implemented as a scalable set of network proxies deployed *alongside* application code (***sidecar***).
- **Data plane**
  - interconnected set of sidecar proxies
- **Control plane**
  - components of a service mesh that are used to configure the proxies and collect metrics



Source: https://glasnostic.com/blog/service-mesh-istio-limits-and-benefits-part-1

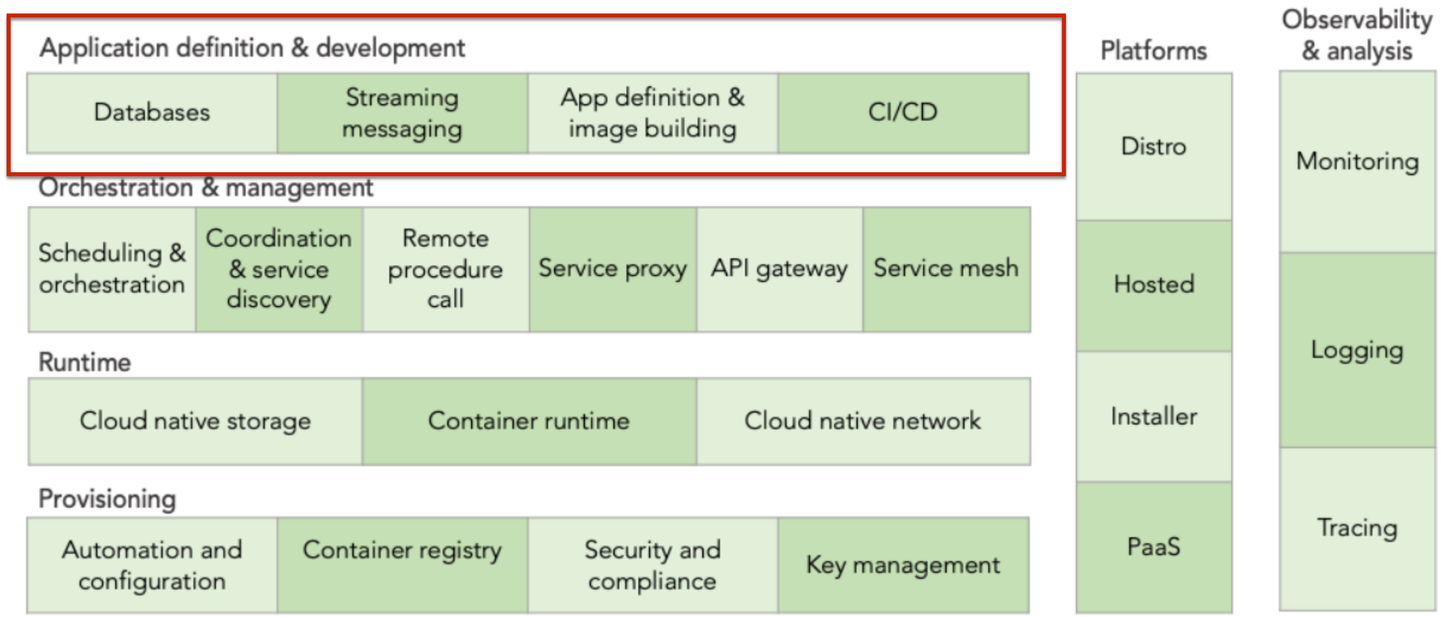
- Service meshes are designed to solve the many challenges developers face when talking to remote endpoint.
- **Benefits of Service Meshes**
  - **Observability**
    - ability to trace requests across remote services for debugging purposes
    - uniform telemetry metrics at the service call level (source, destination, protocol, URL, status codes, latency, duration)
    - data for all services collected and passed along to the monitoring tool of choice
  - **Traffic Control**
    - well suited to balance *individual calls* across a number of destination instances
    - resiliency patterns like retries, timeouts, deadlines, circuit breaking, canary releases, and A/B releases
    - Discovery and Routing, Load balancing
  - **Security**
    - The authentication of services (certificate authority to manage keys and certificates)
    - The encryption of traffic between services (mTLS)
    - Security-specific policy enforcement

- **Limitations of Service Meshes**
  - Added Complexity
    - introduction of proxies, sidecars and other components into an already sophisticated environment dramatically increases the complexity of development and operations
  - Required Expertise
    - Adding a service mesh on top of an orchestrator often requires operators to become experts in both technologies.
  - Slowness
    - Service meshes are an invasive and intricate technology that can add significant slowness to an architecture
  - Adoption of a Platform
    - Force operators to adapt to a highly opinionated platform and conform to its rules
- **Service mesh technologies**
  - Linkerd
  - Istio
  - AspenMesh by F5, Consul Connect by HashiCorp, Kong, AppMesh by AWS
  - Microsoft started an initiative to standardize the various service mesh interfaces, dubbed SMI

App Definition and Development

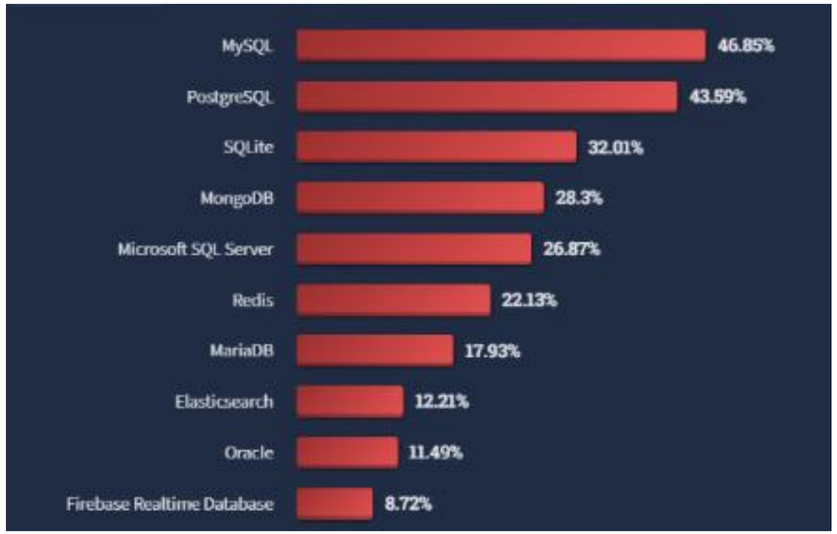
- The application definition and development layer focuses on the **tools that enable engineers to build apps**.

- **Topics:**
  - Database
  - Streaming & Messaging
  - Application Definition & Image Build
  - Continuous Integration & Delivery



Databases

- **A database is an application through which other apps can efficiently store and retrieve data.**
- Databases allow you to store data, ensure only authorized users access it, and enable users to retrieve it via specialized requests.
- Database types:
  - Structured query language (SQL) databases
  - no-SQL databases
- **New cloud native databases** aim to bring the scaling and availability benefits of Kubernetes to databases.

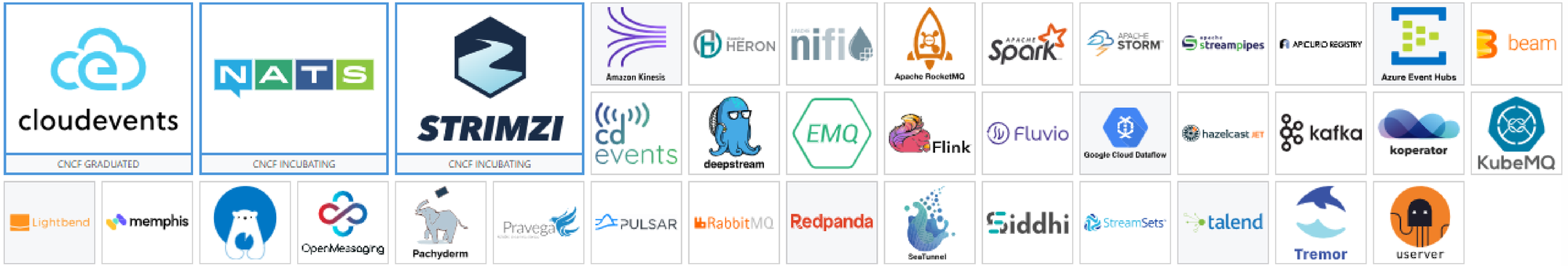


Source: https://www.linkedin.com/pulse/top-5-most-popular-databases-2023-learnsql-com/


TiKV
CNCF GRADUATED
Vitess
CNCF GRADUATED
CarbonData
Hadoop
Ignite
ArangoDB
BIGCHAINDB
cassandra
ClickHouse
Cockroach Labs
Couchbase
CRATE.IO
crunchy data
CRUX
Databend
Dgraph
DORIS
Dragonfly
druid
EDB
Epsilla
FoundationDB
GraphScope
GreptimeDB
hazelcast IMDG
IBM DB2
iguazio
Infinispan
InterSystems
KUBEBLOCKS
KubeDB
MariaDB
SQL Server
MogDB
mongoDB
MySQL
NebulaGraph
neo4j
noms
nuoDB
OCEANBASE
openGauss
openGemini
ORACLE
OrientDB
PERCONA
pilosa
PostgreSQL
presto
qdrant
Qubole
redis
RethinkDB
SAP
SCALARDB
SCHEMAHERO
SCYLLA
SEATA
ShardingSphere
SingleStore
snowflake
software
authzed
StarRocks
STOLON
Tarantool
TDengine
TiDB
Timescale
UQBAR
Vald
VERTICA
by opentext
VOLTDB
Weaviate
YDB
Yellowbrick
yugabyteDB
PolarDB

Streaming & Messaging

- Streaming and messaging tools enable service-to-service communication by transporting messages (i.e. events) between systems.
- This dynamic creates an environment where individual apps are either **publishers**, meaning they write events, or **subscribers** that read events, or more likely both.
- Highly **decoupled architecture** where services can collaborate without needing to know about one another.

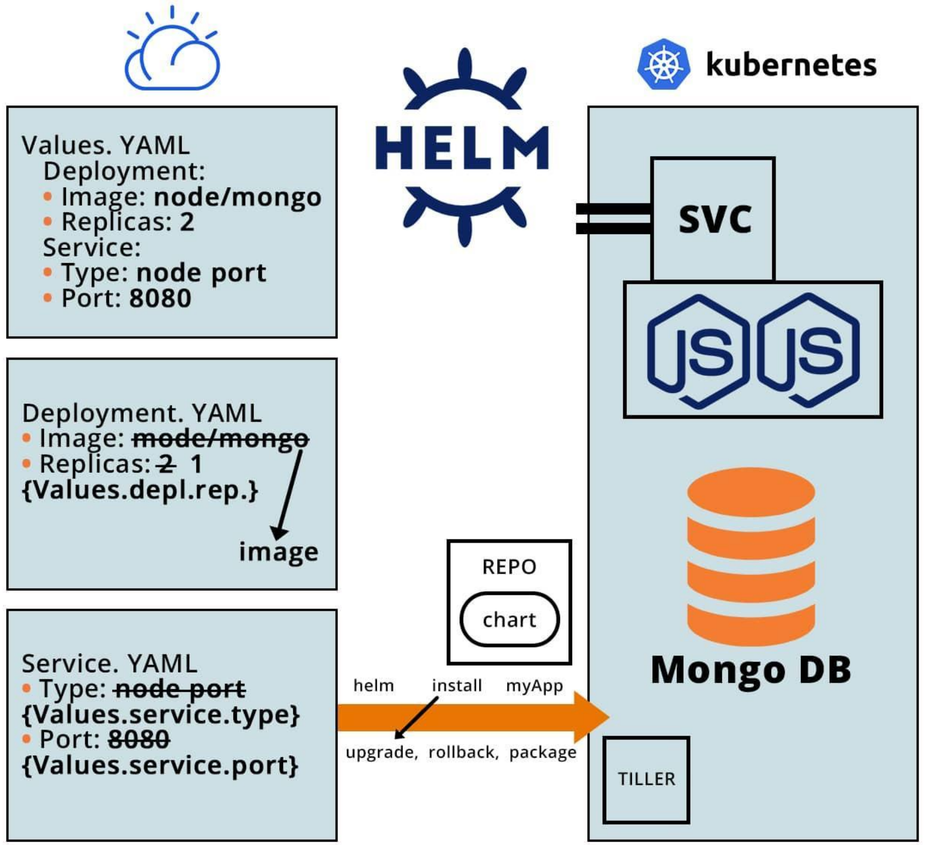


Application Definition & Image Build

- **Developer-focused tools** that help build application code into containers and/or Kubernetes.
- **Operations-focused tools** that deploy apps in a standardized way.

Helm

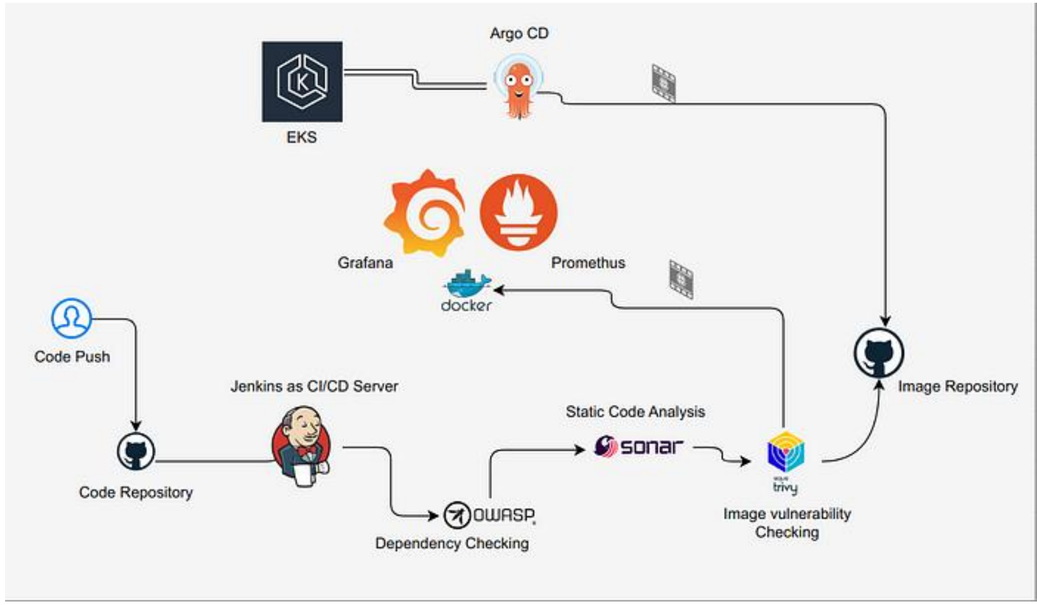
- https://helm.sh/
- The package manager for Kubernetes
- Helm Charts help you define, install, and upgrade even the most complex Kubernetes application.
- The Purpose of Helm:
  - Create new charts from scratch
  - Package charts into chart archive (tgz) files
  - Interact with chart repositories where charts are stored
  - Install and uninstall charts into an existing Kubernetes cluster
  - Manage the release cycle of charts that have been installed with Helm



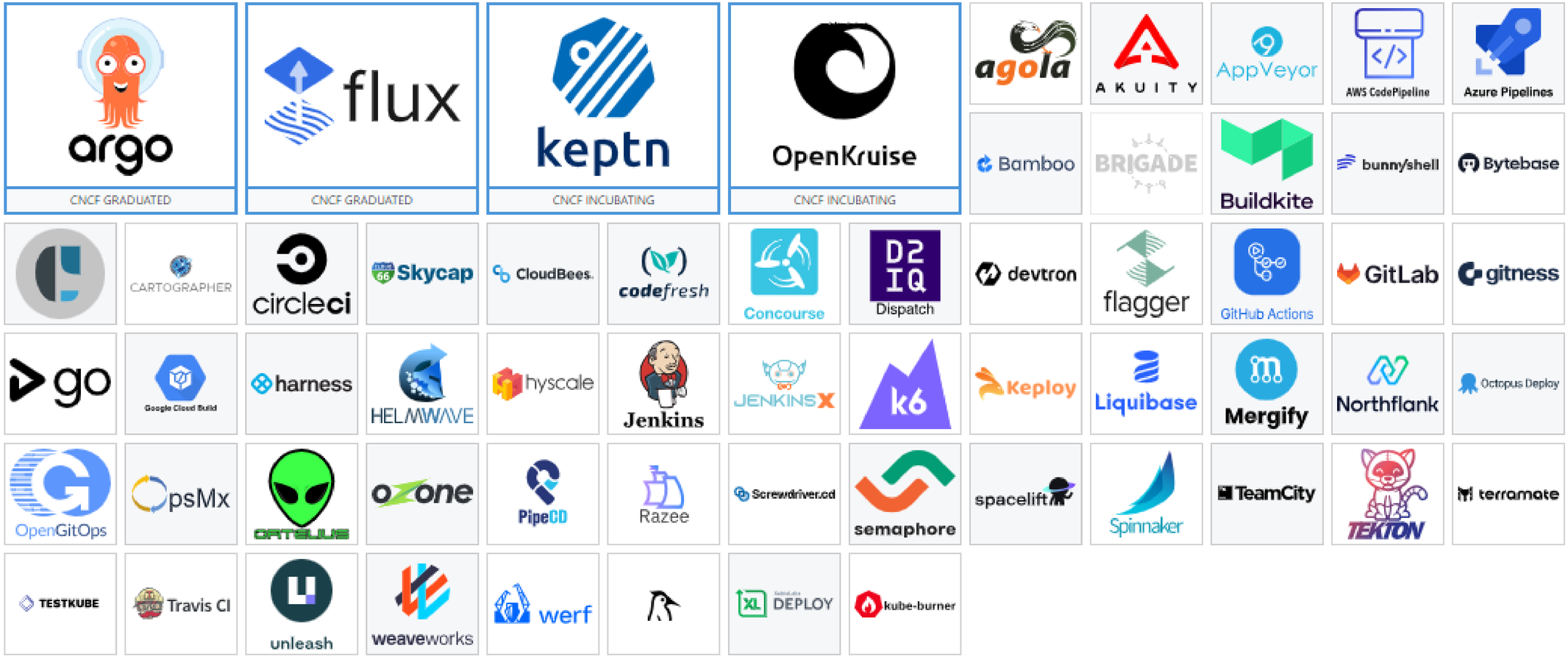
Source: https://kruschecompany.com/helm-kubernetes/

Continuous Integration & Delivery

- **Continuous integration (CI)** and **continuous delivery (CD)** tools enable fast and efficient development with embedded quality assurance.
  - **CI** automates code changes by immediately building and testing the code, ensuring it produces a deployable artifact.
  - **CD** goes one step further and pushes the artifact through the deployment phases.
- Mature CI/CD systems **watch source code** for changes, **automatically** build and test the code, then begin moving it from development to production where it has to pass a variety of tests or validation to determine if the process should continue or fail.

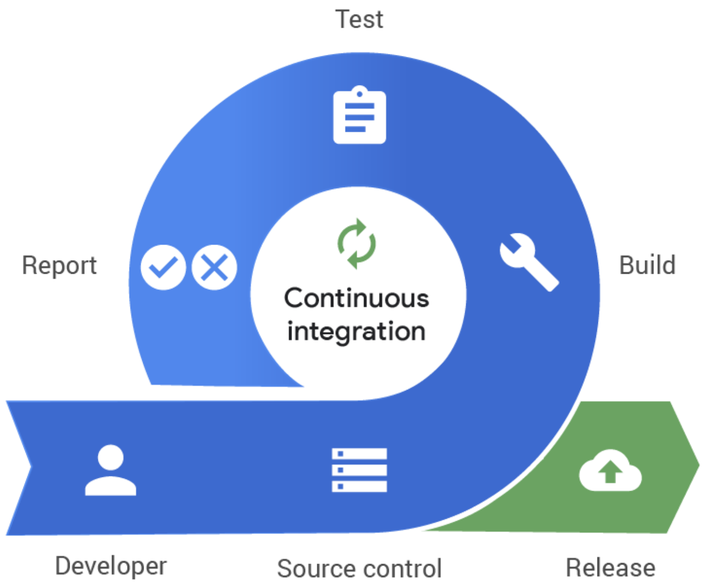


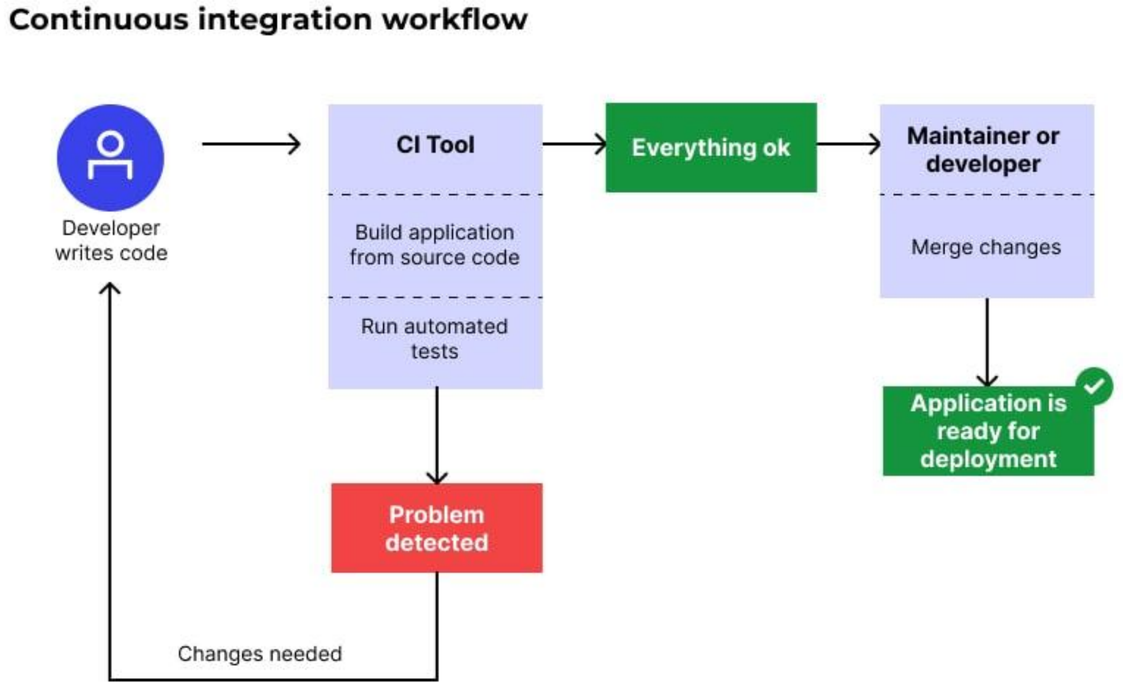
Source: https://medium.com/@inderjotsingh141/ultimate-ci-cd-using-aws-4d177ad79629


argo
CNCF GRADUATED
flux
CNCF GRADUATED
keptn
CNCF INCUBATING
OpenKruise
CNCF INCUBATING
agola
AKUITY
AppVeyor
AWS CodePipeline
Azure Pipelines
Bamboo
BRIGADE
Buildkite
bunnyshell
Bytebase
CARTOGRAPHER
circleci
Skycap
CloudBees.
codefresh
Concourse
D2 IQ
Dispatch
devtron
flagger
GitHub Actions
GitLab
gitness
go
Google Cloud Build
harness
HELMWAVE
hyscale
Jenkins
JENKINS X
k6
Keploy
Liquibase
Mergify
Northflank
Octopus Deploy
OpenGitOps
OpsMx
ORTELIUS
ozone
PipeCD
Razee
Screwdriver.cd
semaphore
spacelift
Spinnaker
TeamCity
TEKTON
terramate
TESTKUBE
Travis CI
unleash
weaveworks
werf
DEPLOY
kube-burner

Continuous integration

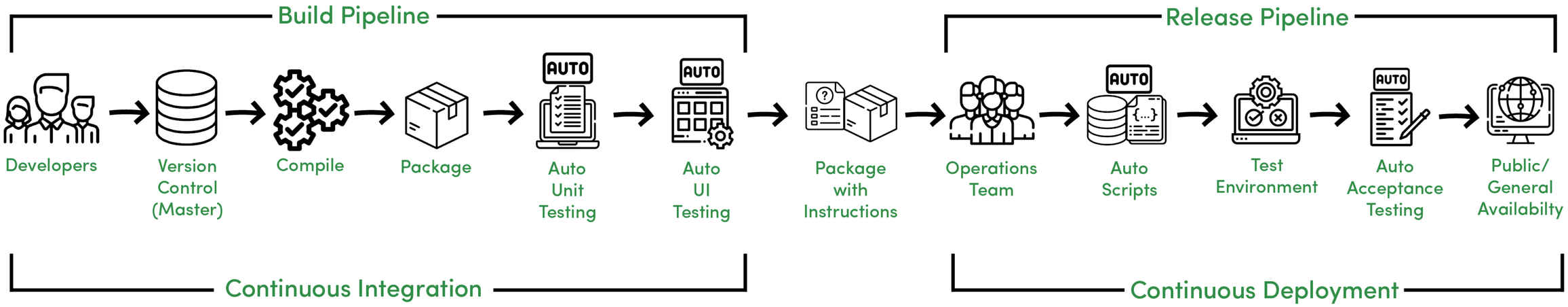
- CI tools ensure that any code change or updates developers introduce are **built**, **validated**, and **integrated** with other changes **automatically and continuously**. Each time a developer adds an update, automated testing is triggered to ensure only good code makes it into the system.





Continuous delivery

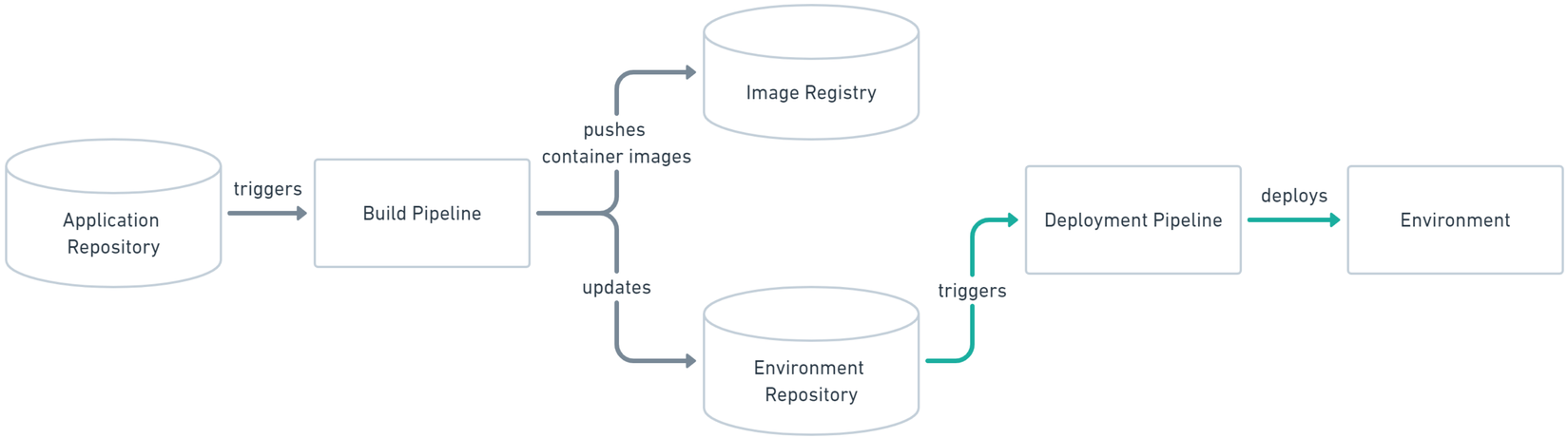
- Continuous Delivery is the ability to get **changes of all types**—including new features, configuration changes, bug fixes and experiments—**into production**, or into the hands of users, **safely and quickly in a sustainable way**.
- Code is always in a deployable state, even in the face of teams of thousands of developers making changes on a daily basis.



Source: https://www.geeksforgeeks.org/ci-cd-continuous-integration-and-continuous-delivery/

GitOps

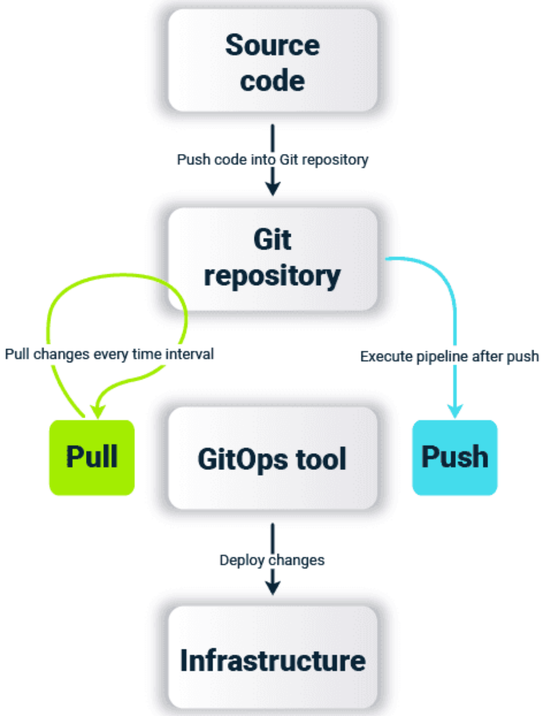
- GitOps is a way of implementing Continuous Deployment for cloud native applications.
- Focuses on a developer-centric experience when operating infrastructure.
- The core idea of GitOps is having a **Git repository** that always contains declarative descriptions of **the infrastructure currently desired in the production** environment.
- Deploy a new application or update an existing one -> you **only need to update the repository** - the automated process handles everything else



Source: https://www.gitops.tech/

Why use GitOps?

- **Deploy Faster More Often**
  - don’t have to switch tools for deploying your application
  - Everything happens in the version control system you use for developing the application anyways.
- **Easy and Fast Error Recovery**
  - error recovery -> issuing a git revert
- **Easier Credential Management**
  - environment only needs access to your repository and image registry
- **Self-documenting Deployments**
  - every change to any environment must happen through the repository
- **Shared Knowledge in Teams**
  - Git to store complete descriptions of your deployed infrastructure

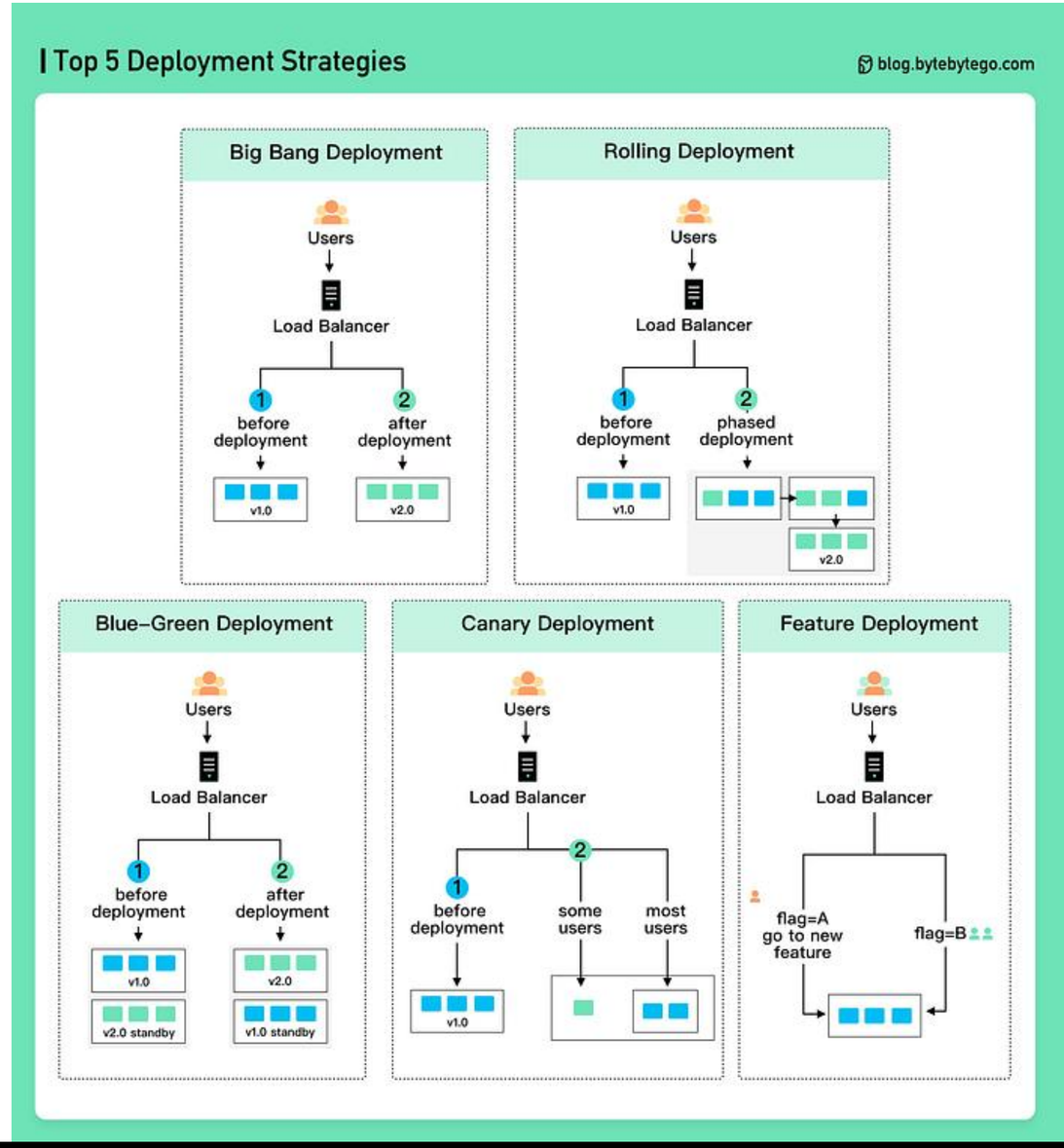


codilime

Source: https://codilime.com/static/31502eb6a1279025d21bbbb799ecd257/f6b72/unlocking-the-power-of-gitops-for-network-automation-10-1-.png

Deployment Strategies

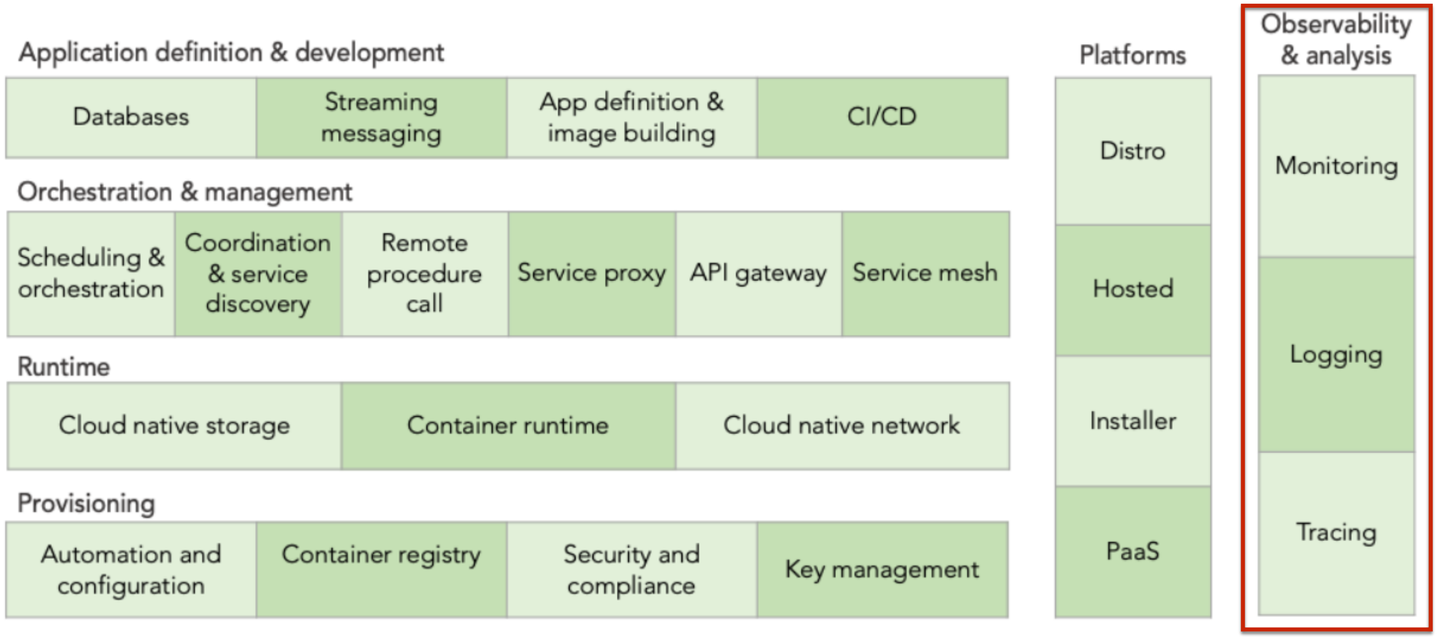
- **Big Bang Deployment** (the entire system is deployed in one go)
- **Rolling Deployment** (updates are deployed gradually)
- **Blue Green Deployment** (two identical environments, “blue” and “green,” are maintained)
- **Canary Deployment** (deployed to a small subset of users or servers)
- **Feature Deployment** (where specific features or functionalities are deployed)



Observability and Analysis

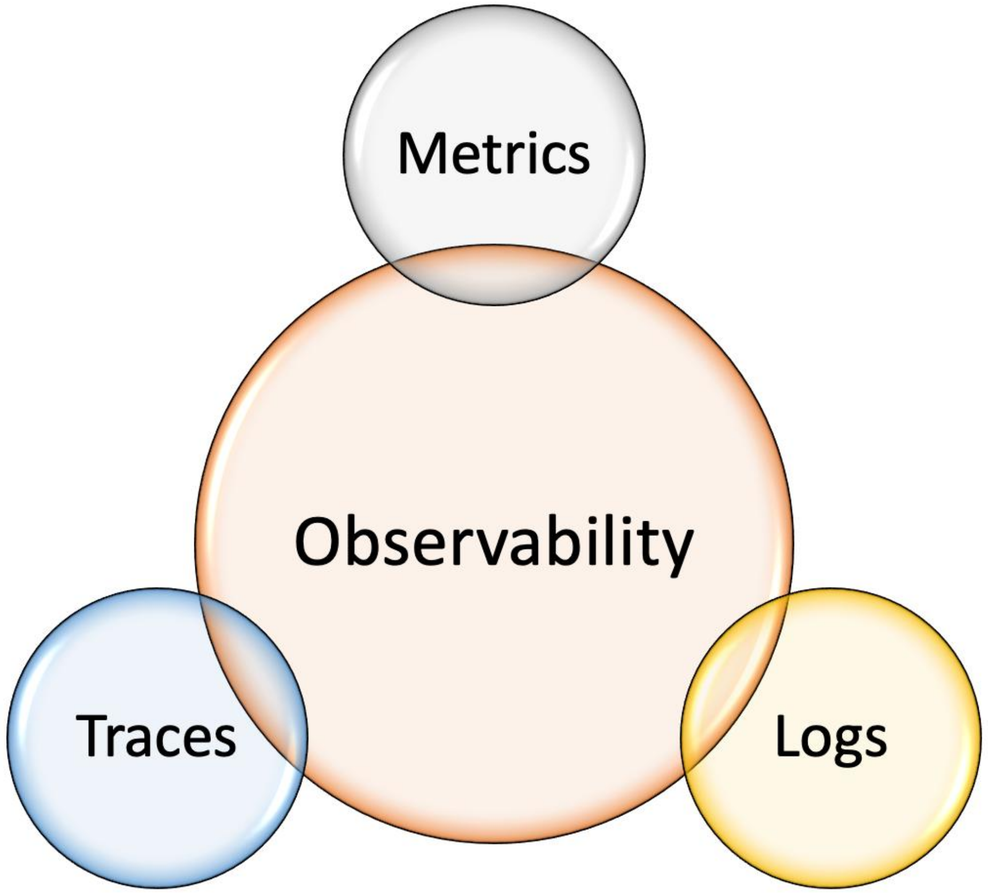
- Observability is a system characteristic describing the degree to which a system can be understood from its external outputs.
- To ensure there is **no service disruption**, you'll need to **observe and analyze** every aspect of your application, so every anomaly gets detected and rectified right away.

- **Topics:**
  - Observability
  - Chaos Engineering



Observability

- Systems are complex, and can fail or degrade in performance in many different ways.
- Observability is the practice and ability of a **system to be understood from its external outputs.**
- Observability frameworks emit telemetry data, that is consumed by analysis tools which aid in querying or visualizing the data.
- These frameworks **emit logs, metrics, and traces** from application code, underlying node infrastructure, orchestration systems.



Source: https://linkedin.github.io/school-of-sre/level101/metrics_and_monitoring/images/image7.png

Logging, Metrics, Traces

- **The 3 pillars of observability**
- **Logs:**
  - Logs are files that record events, warnings and errors as they occur within a software environment.
  - Most logs include contextual information, such as the time an event occurred and which user or endpoint was associated with it.
  - Observability tools aggregate data from multiple log files and analyze it collectively.
  - Provide a comprehensive record of all events and errors that take place during the lifecycle of software resources.
  - One of the biggest is that they record only the events, warnings and errors the logging software has been configured to record.
  - Log data isn't always persistent (containerized applications) – centralized storage

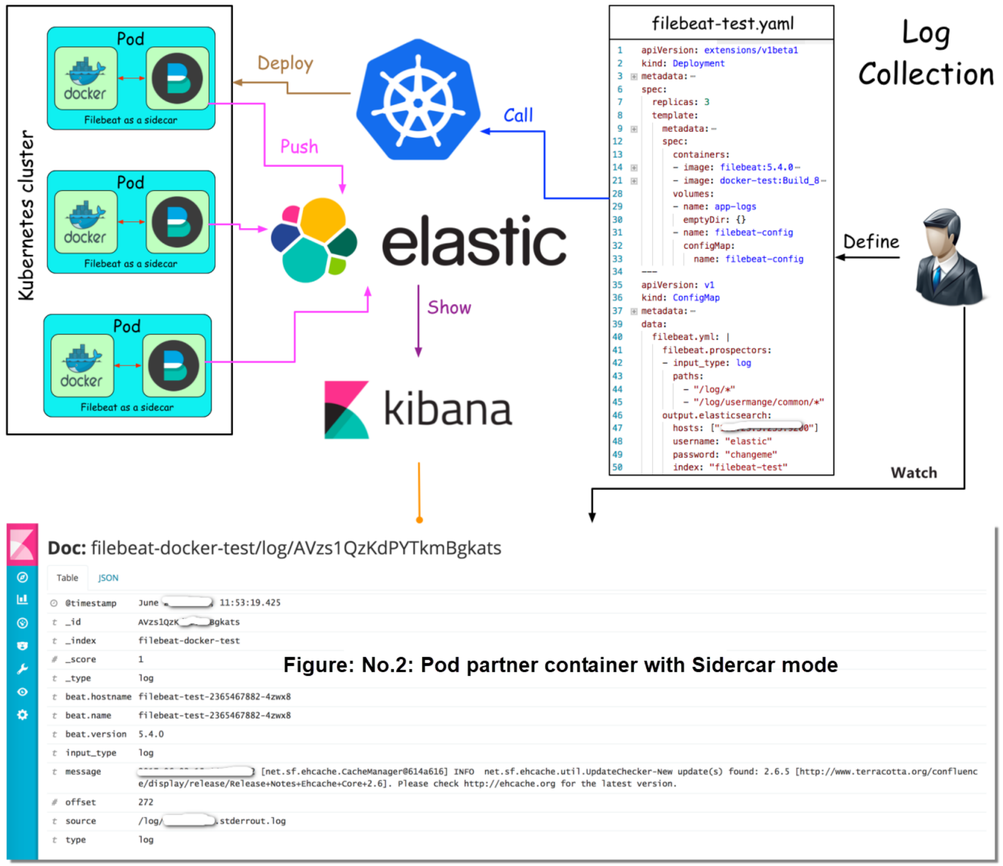
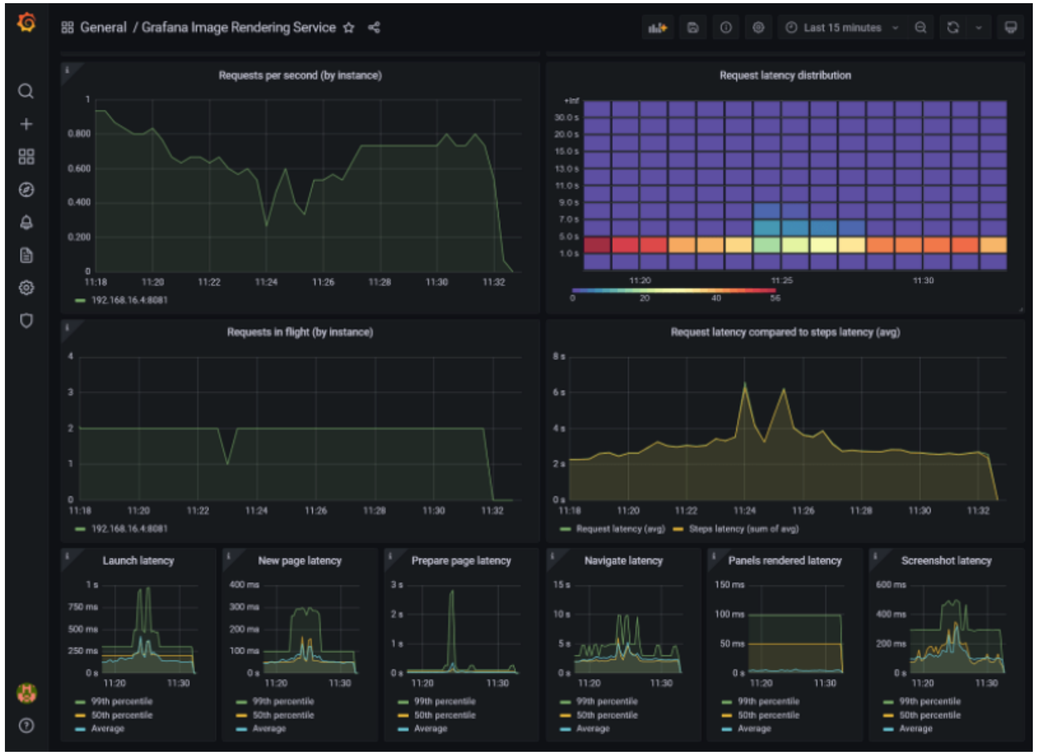


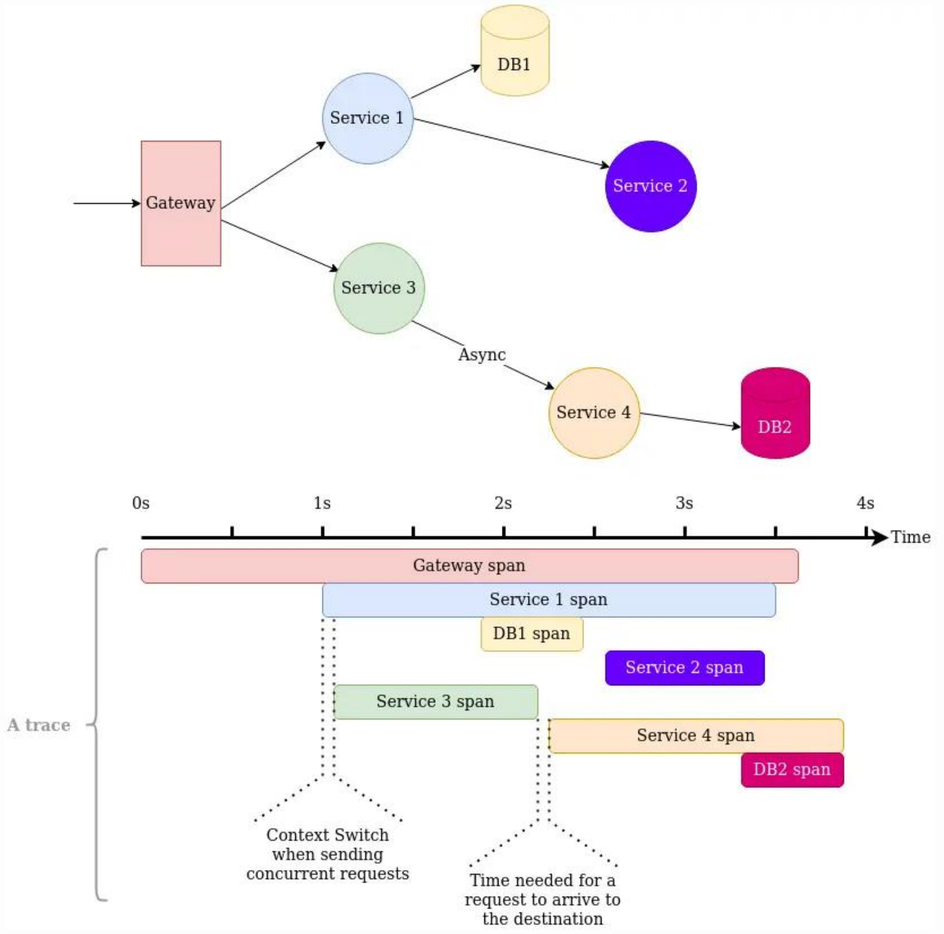
Figure: No.2: Pod partner container with Sidercar mode

Source: https://www.alibabacloud.com/blog/log-platform-solution-in-the-cloud-native-architecture_597654

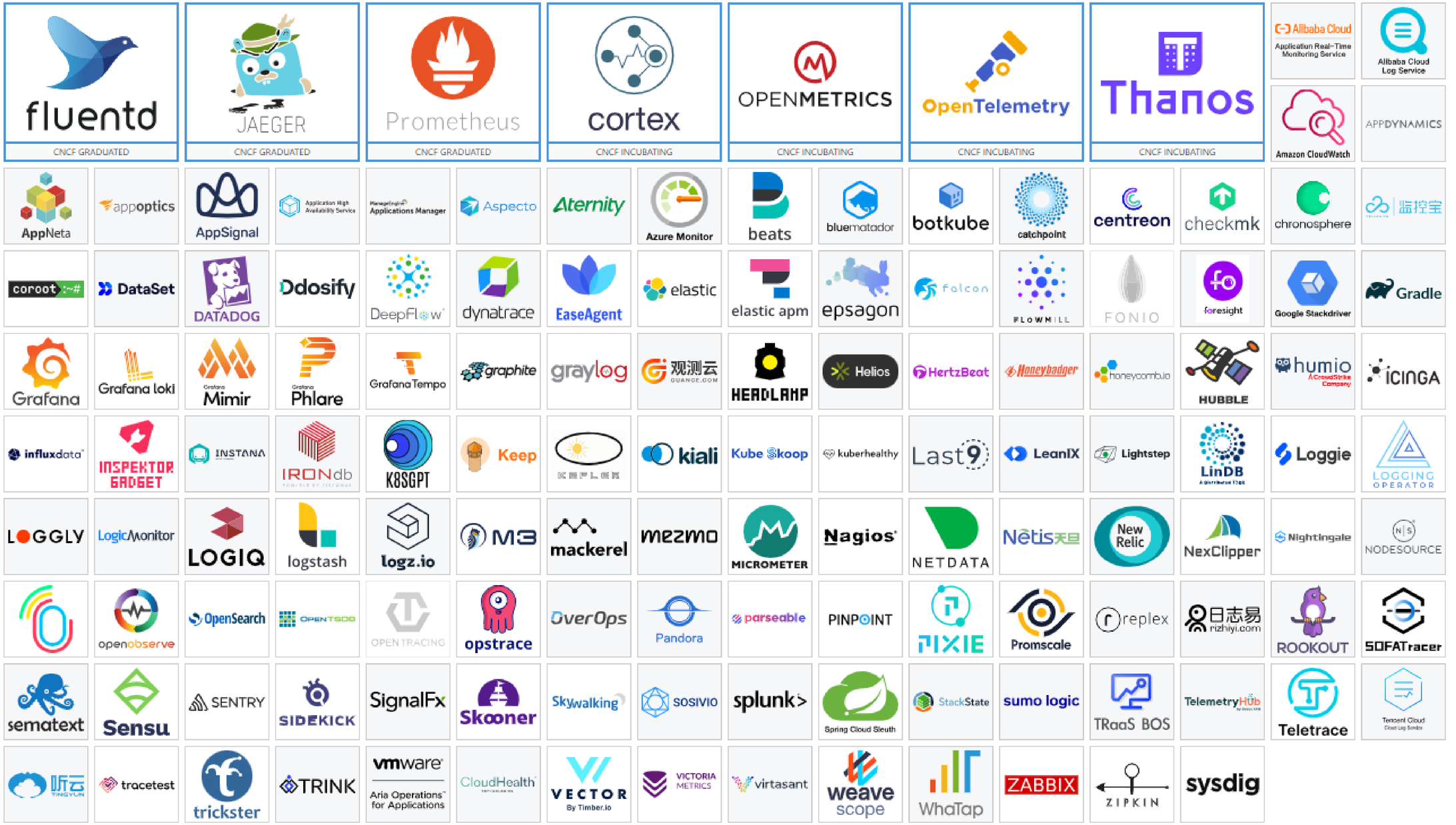
- **Metrics:**
  - Metrics are quantifiable measurements that reflect the health and performance of applications or infrastructure.
  - Example: transaction per second, CPU, memory consumption
  - The main benefit of metrics is that they provide real-time insight into the state of resources.
  - By correlating metrics with data from logs and traces, organizations gain the fullest possible context on system performance.
  - Metrics aren't typically useful for pinpointing the source of a problem, especially in a complex distributed system.



- **Traces:**
  - A distributed trace is data that tracks an application **request as it flows** through the various parts of an application.
  - The trace records how long it takes each application component to process the request and pass the result to the next component.
  - Traces can also identify which parts of the application trigger an error.
  - Distributed traces are the most effective way to research the root cause of a problem.
  - The major limitation of distributed traces is that only a fraction of all application requests are traced in most cases.
    - traces takes too much time and consumes too many resources to trace every request an application receives

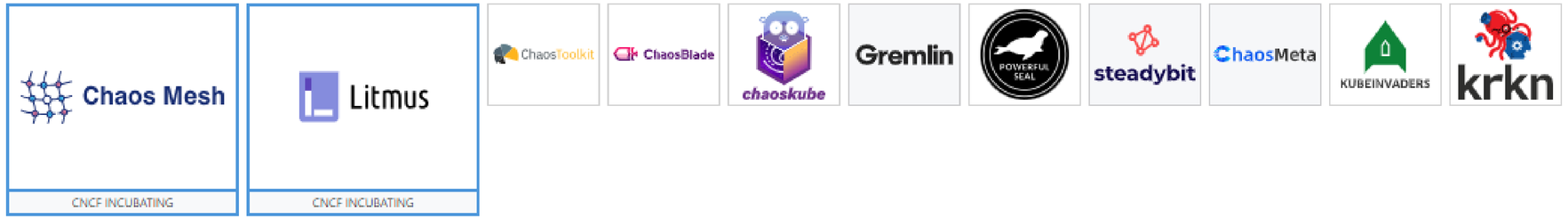


Source: https://alexandruburlacu.github.io/posts/2021-05-20-logs-traces-how-to


fluentd
CNCF GRADUATED
JAEGER
CNCF GRADUATED
Prometheus
CNCF GRADUATED
cortex
CNCF INCUBATING
OPENMETRICS
CNCF INCUBATING
OpenTelemetry
CNCF INCUBATING
Thanos
CNCF INCUBATING
Alibaba Cloud
Application Real-Time Monitoring Service
Alibaba Cloud Log Service
Amazon CloudWatch
APPDYNAMICS
AppNeta
appoptics
AppSignal
Application High Availability Service
Applications Manager
Aspecto
Aternity
Azure Monitor
beats
bluematador
botkube
catchpoint
centreon
checkmk
chronosphere
监控宝
coroot
DataSet
DATADOG
Ddosify
DeepFlow
dynatrace
EaseAgent
elastic
elastic apm
epsagon
falcon
FLOWMILL
FONIO
foresight
Google Stackdriver
Gradle
Grafana
Grafana loki
Grafana Mimir
Grafana Phlare
Grafana Tempo
graphite
graylog
观测云
HEADLAMP
Helios
HertzBeat
Honeybadger
honeycomb.io
HUBBLE
humio
ICINGA
influxdata
INSPEKTOR GADGET
INSTANA
IRONdb
K8SGPT
Keep
KPLON
kiali
Kube Skoop
kuberhealthy
Last9
LeanIX
Lightstep
LinDB
Loggie
LOGGING OPERATOR
LOGGLY
LogicMonitor
LOGIQ
logstash
logz.io
M3
mackerel
mezmo
MICROMETER
Nagios
NETDATA
Nētis天旦
New Relic
NexClipper
Nightingale
NODESOURCE
openobserve
OpenSearch
OPENTSDB
OPEN TRACING
opstrace
OverOps
Pandora
parseable
PINPOINT
PIXIE
Promscale
replex
日志易 rizhiyi.com
ROOKOUT
SOFATracer
sematext
Sensu
SENTRY
SIDEKICK
SignalFx
Skooner
Skywalking
SOSIVIO
splunk>
Spring Cloud Sleuth
StackState
sumo logic
TRaaS BOS
TelemetryHub
Teletrace
Tencent Cloud
听云
tracetest
trickster
TRINK
vmware
Aria Operations for Applications
CloudHealth
VECTOR
By Timber.io
VICTORIA METRICS
virtasant
weave scope
WhaTap
ZABBIX
ZIPKIN
sysdig

Chaos Engineering

- Chaos engineering refers to the practice of **intentionally introducing faults into a system** in order to test its resilience and ensure applications and engineering teams are able to withstand turbulent and unexpected events.
- In a cloud native world, applications must **dynamically adjust to failures**.
- Chaos engineering tools enable you to experiment on a software system in production to ensure they perform gracefully should a real failure occur.
- Chaos engineering tools and practices are critical to **achieving high availability for your applications**.

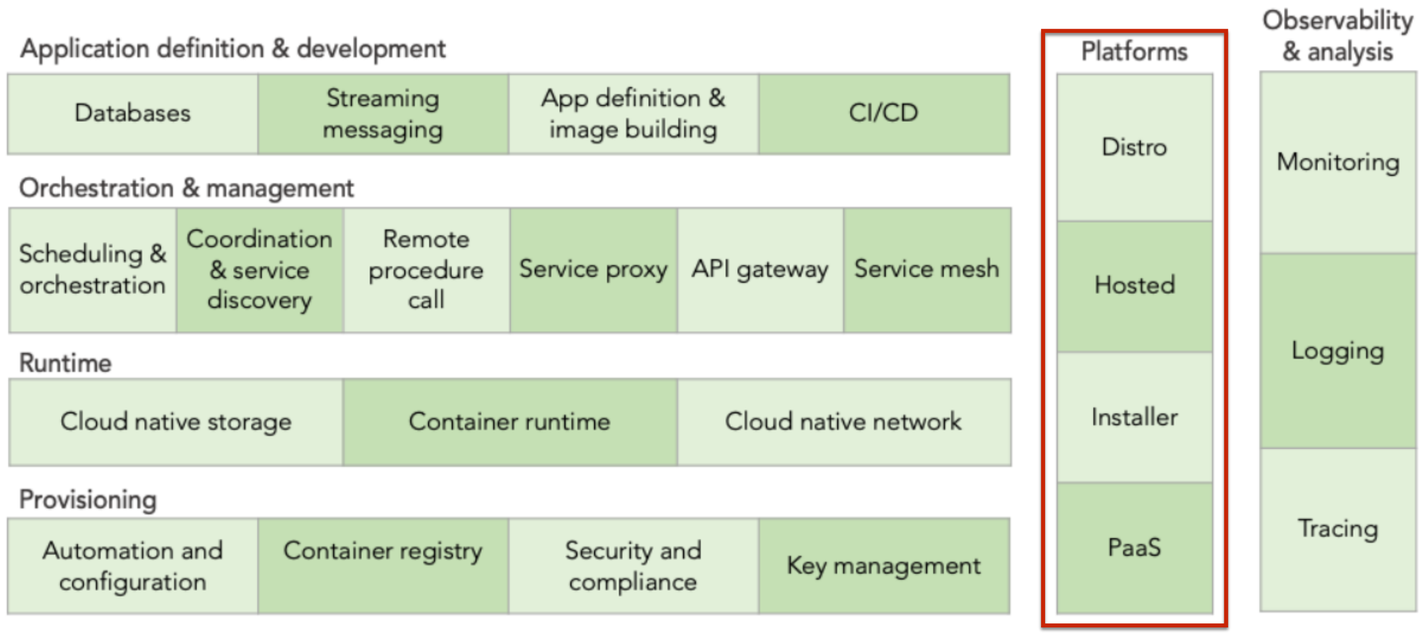


Platform

- Platforms **bundle different tools** from different layers together, solving a larger problem.
- For some organizations, especially those with small engineering teams, platforms are the only way to adopt a cloud native approach.
- All platforms revolve around Kubernetes (core of the cloud native stack)

Topics:

- Certified Kubernetes - Distribution
- Certified Kubernetes – Hosted
- Certified Kubernetes – Installer
- PaaS/Container Service



Certified Kubernetes - Distribution

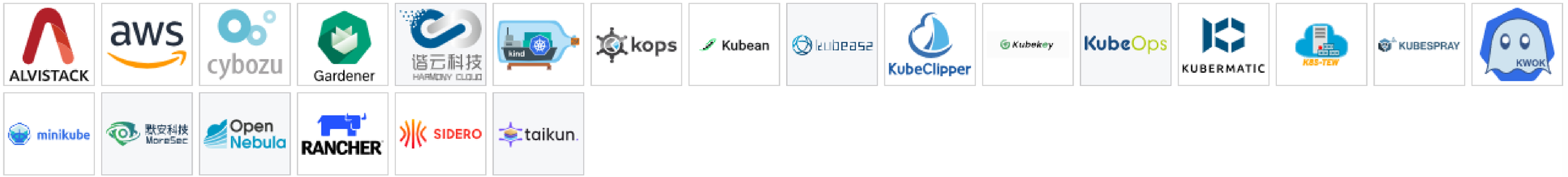
- A distribution, or distro, is when a vendor takes core Kubernetes — that's the unmodified, open source code (although some modify it) — and packages it for redistribution.
- Usually this entails finding and validating the Kubernetes software and providing a mechanism to handle cluster installation and upgrades.
- Kubernetes distributions provide a trusted and reliable way to install Kubernetes and provide opinionated defaults that create a better and more secure operating environment.

Certified Kubernetes - Hosted

- Hosted Kubernetes is a service offered by infrastructure providers like AWS, Digital Ocean, Azure, and Google, allowing customers to spin up a Kubernetes cluster on-demand.
- The cloud provider **takes responsibility** for **managing part of the Kubernetes cluster**, usually called the control plane.
- They are similar to distributions but managed by the cloud provider on their infrastructure.
- Hosted Kubernetes is the easiest way to get started with cloud native.
- Managed clusters provide stricter limits on configuring your Kubernetes cluster than DIY Kubernetes clusters.

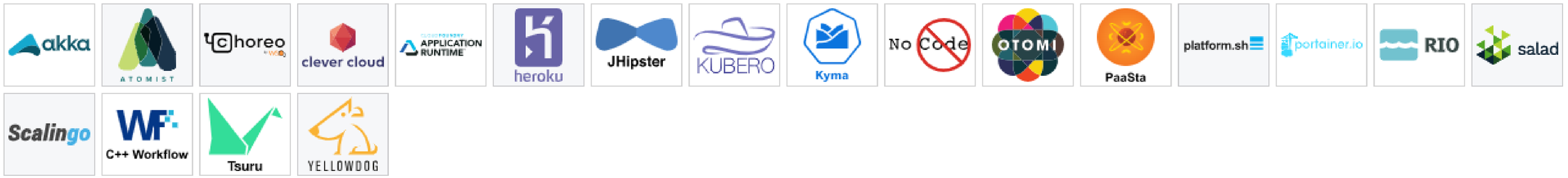
Certified Kubernetes - Installer

- Kubernetes installers help install Kubernetes on a machine.
- They automate the Kubernetes installation and configuration process and may even help with upgrades. Kubernetes installers are often coupled with or used by Kubernetes distributions or hosted Kubernetes offerings.
- Kubernetes installers like kind (Kubernetes in Docker) allow you to get a Kubernetes cluster with a single command.



PaaS/Container Service

- A Platform-as-a-Service, or PaaS, is an environment that allows users to run applications without necessarily concerning themselves with the details of the underlying compute resources.
- A PaaS attempts to connect many of the technologies found in this landscape in a way that provides direct value to developers.
- Trade-offs and restrictions: Stateless applications tend to do very well in a PaaS but stateful applications like databases usually don't.

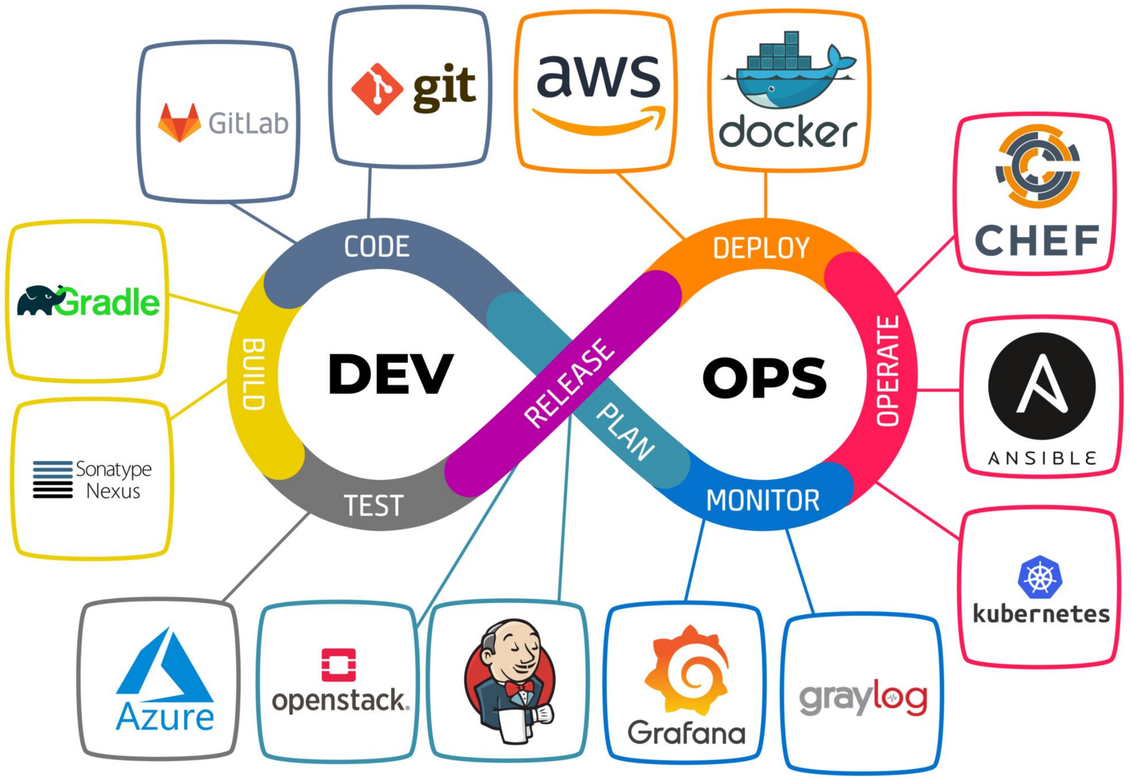


DevOps

170

DevOps

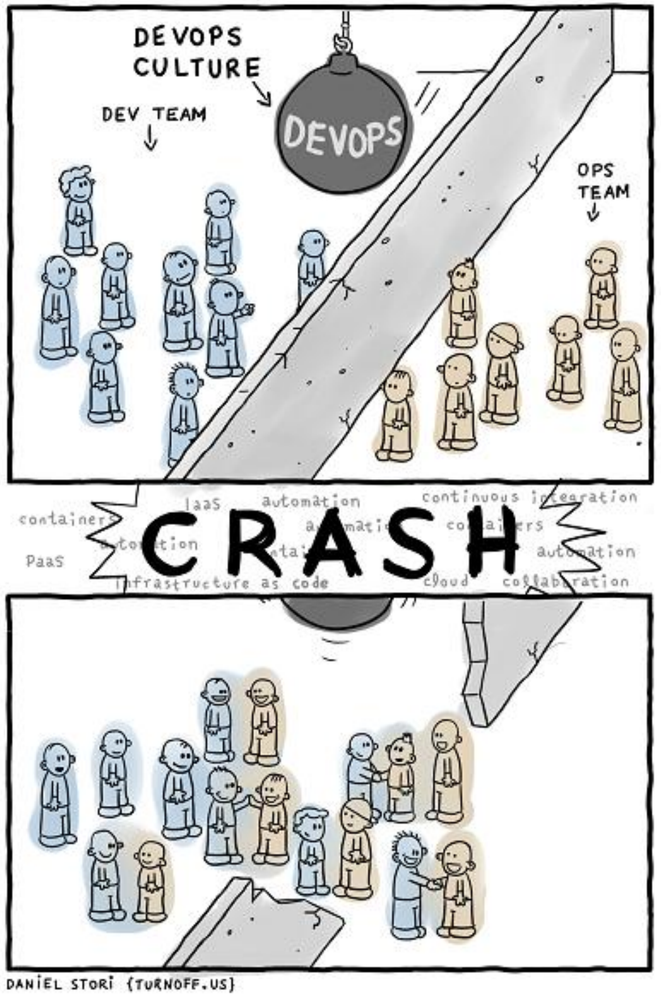
- **DevOps combines development (Dev) and operations (Ops) to increase the efficiency, speed, and security of software development and delivery compared to traditional processes.**
- DevOps is not a role. DevOps is a set of practices.



Source: https://shalb.com/blog/what-is-devops-and-where-is-it-applied/

DevOps explained

- DevOps is a combination of software development (dev) and operations (ops).
- People working together to conceive, build and deliver secure software at top speed.
- Accelerate delivery through **automation**, **collaboration**, **fast feedback**, and **iterative improvement**.
- DevOps represents a change in mindset for IT culture.
- DevOps focuses on incremental development and rapid delivery of software.

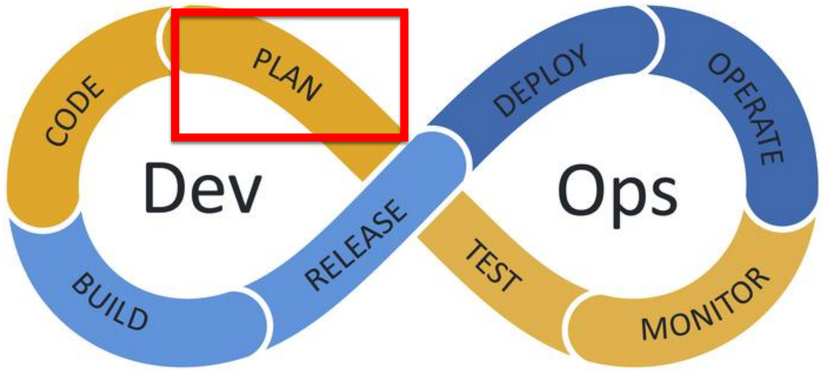


Source: https://www.neilmillard.com/2017/08/19/what-is-devops.html

DevOps Lifecycle

1. Plan:

- planning the project's lifecycle
- DevOps workflow is planned with the likelihood of future iterations and likely prior versions in mind
- we will likely have information from past iterations that will better inform the next iteration

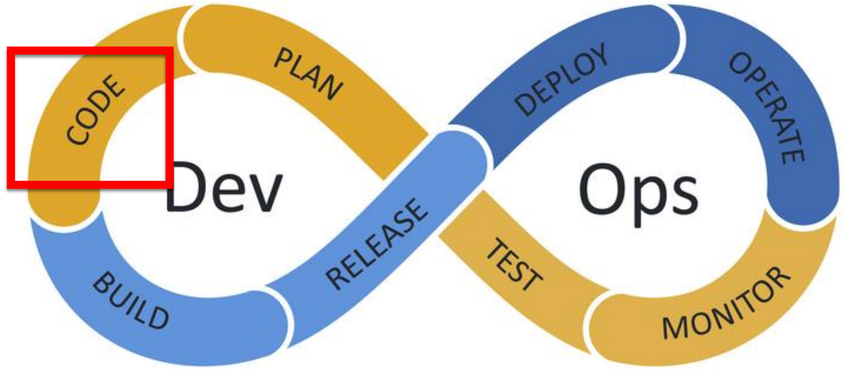


Jira Software Confluence slack

DevOps Lifecycle

2. Code:

- The developers will write the code and prepare it for the next phase during the coding stage.
- Developers will write code in accordance with the specifications outlined in the planning phase and will ensure that the code is created with the project's operations in mind.



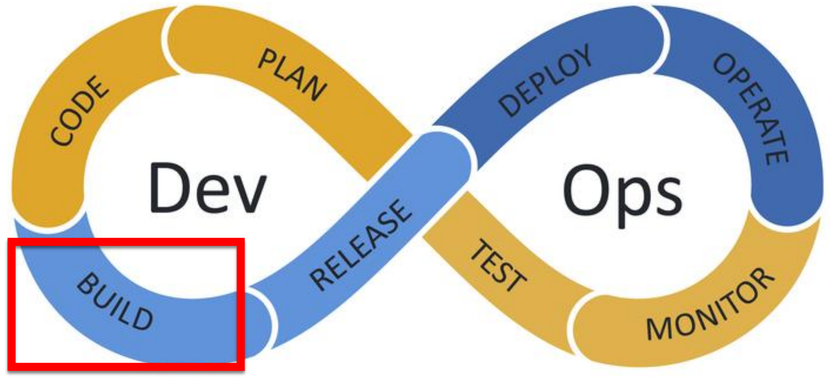
GitHub

GitLab

DevOps Lifecycle

3. Build:

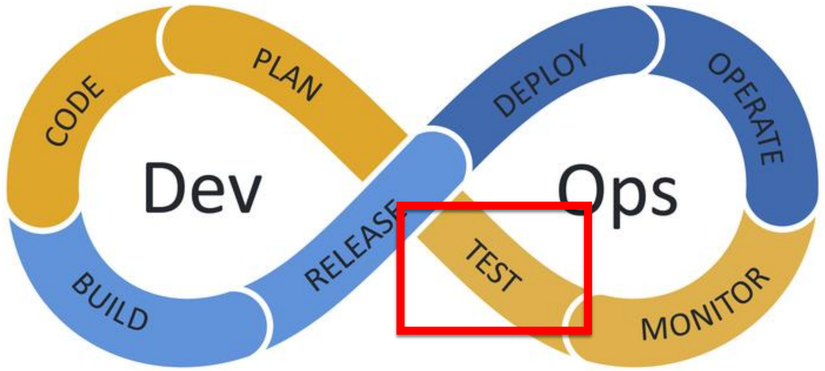
- Code will be introduced to the project during the construction phase, and if necessary, the project will be rebuilt to accommodate the new code.
- During this phase, you essentially manage various software builds and versions with the help of automated tools that assist in compiling and packaging code for future release to production.



DevOps Lifecycle

4. Test:

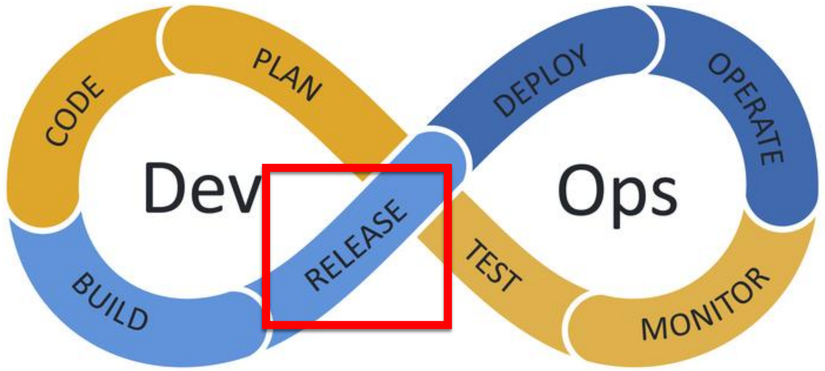
- It is the phase of continuous testing that ensures optimal code quality.
- Teams will also test for edge and corner case issues at this stage.



DevOps Lifecycle

5. Release:

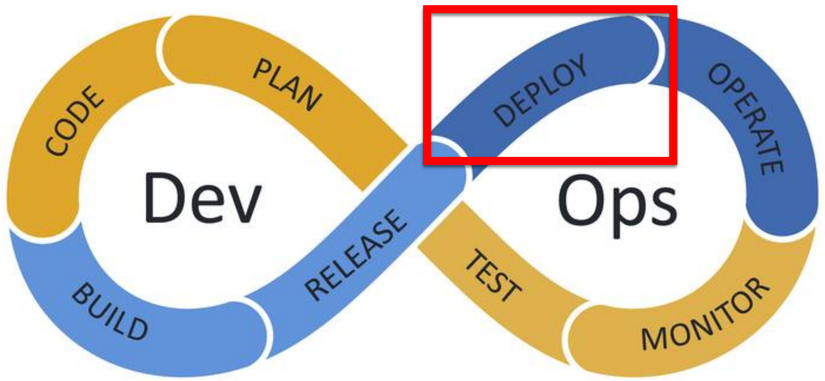
- The release phase occurs when the code has been verified as ready for deployment and a last check for production readiness has been performed.
- The project will subsequently enter the deployment phase if it satisfies all requirements and has been thoroughly inspected for bugs and other problems.



DevOps Lifecycle

6. Deploy:

- This is the phase of managing, scheduling, coordinating, and automating various product releases into production.
- Traditionally, this would be the responsibility of the operations team; in DevOps, it is a shared responsibility. This shared duty pushes team members to collaborate to guarantee a successful deployment.



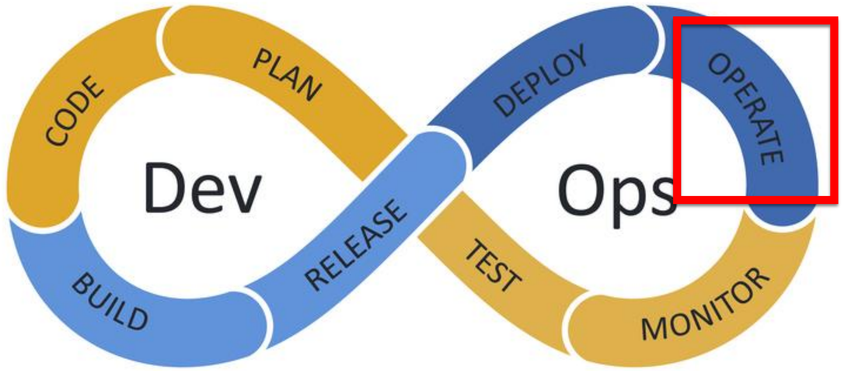
OPENSHIFT

kubernetes

DevOps Lifecycle

7. Operate:

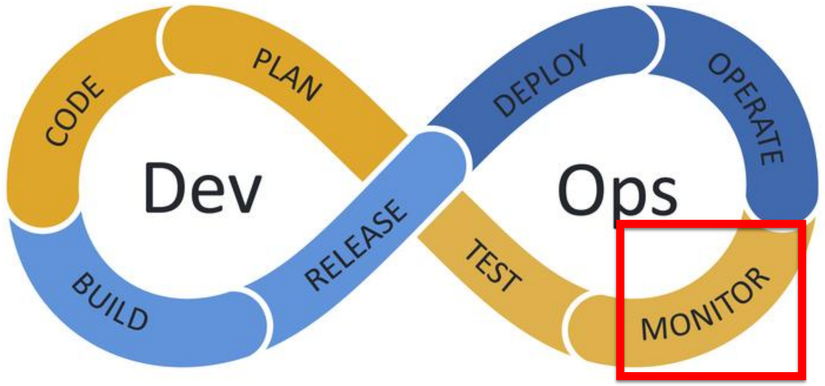
- In the operating phase, teams test the project in a production environment, and end users utilize the product.
- This crucial stage is by no means the final step.
- It informs future development cycles and manages the configuration of the production environment and the implementation of any runtime requirements.



DevOps Lifecycle

8. Monitor:

- During the monitoring phase, product usage, as well as any feedback, issues, or possibilities for improvement, are recognized and documented.
- This information is then conveyed to the subsequent iteration to aid in the development process.
- This phase is essential for planning the next iteration and streamlines the pipeline’s development process.

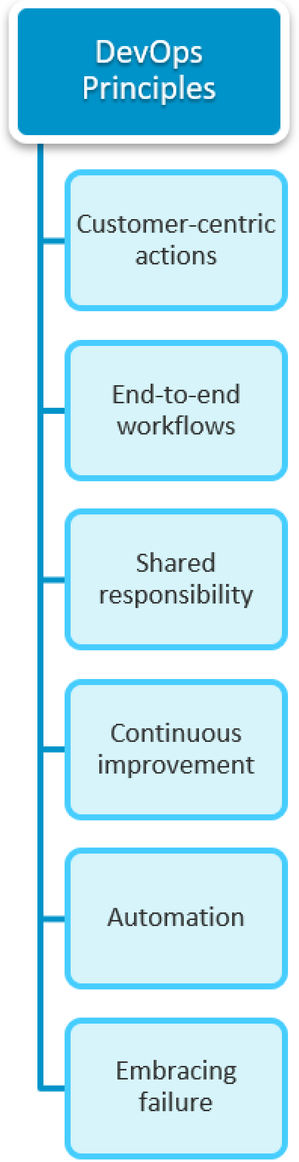


splunk> slack

DATADOG

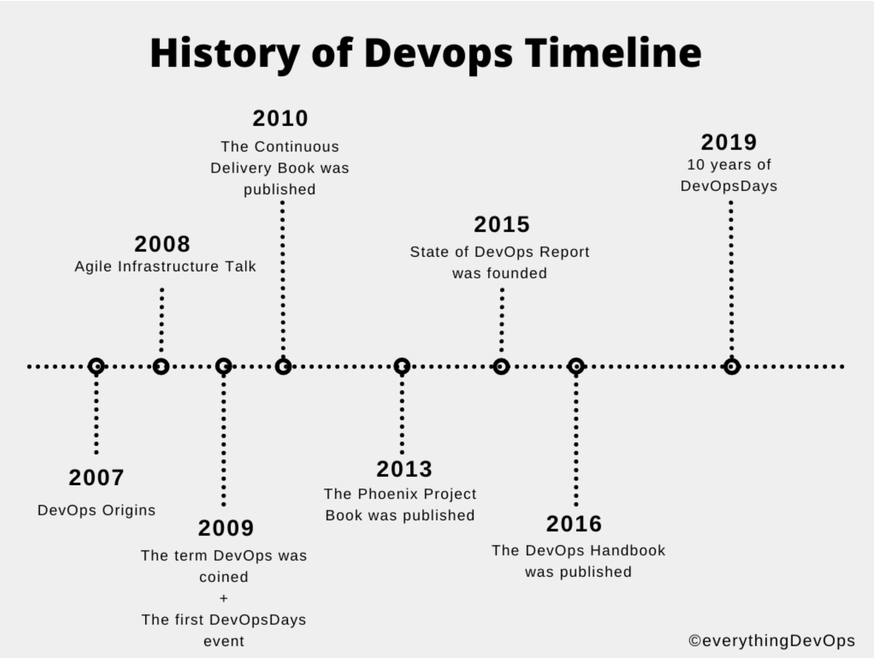
Core DevOps principles

- **Automation of the software development lifecycle**
  - automating testing, builds, releases, the provisioning of development environments
- **Collaboration and communication**
  - effective collaboration and communication in the team
- **Continuous improvement and minimization of waste**
  - automating repetitive tasks
  - watching performance metrics for ways to reduce release times or mean-time-to-recovery
- **Hyperfocus on user needs with short feedback loops**
  - Through automation, improved communication and collaboration, and continuous improvement, DevOps teams can take a moment and focus on what real users really want, and how to give it to them.



Phases of DevOps

- **Phase 1: Bring Your Own DevOps**
  - each team selected its own tools
  - This approach caused problems when teams attempted to work together because they were not familiar with the tools of other teams
- **Phase 2: Best-in-class DevOps**
  - organizations standardized on the same set of tools, with one preferred tool for each stage of the DevOps lifecycle
  - It helped teams collaborate with one another, but the problem then became moving software changes through the tools for each stage.
- **Phase 3: Do-it-yourself (DIY) DevOps**
  - building on top of and between their tools
  - maintaining DIY DevOps was a significant effort and resulted in higher costs
- **Phase 4: DevOps Platform**
  - A single-application platform approach improves the team experience and business efficiency.



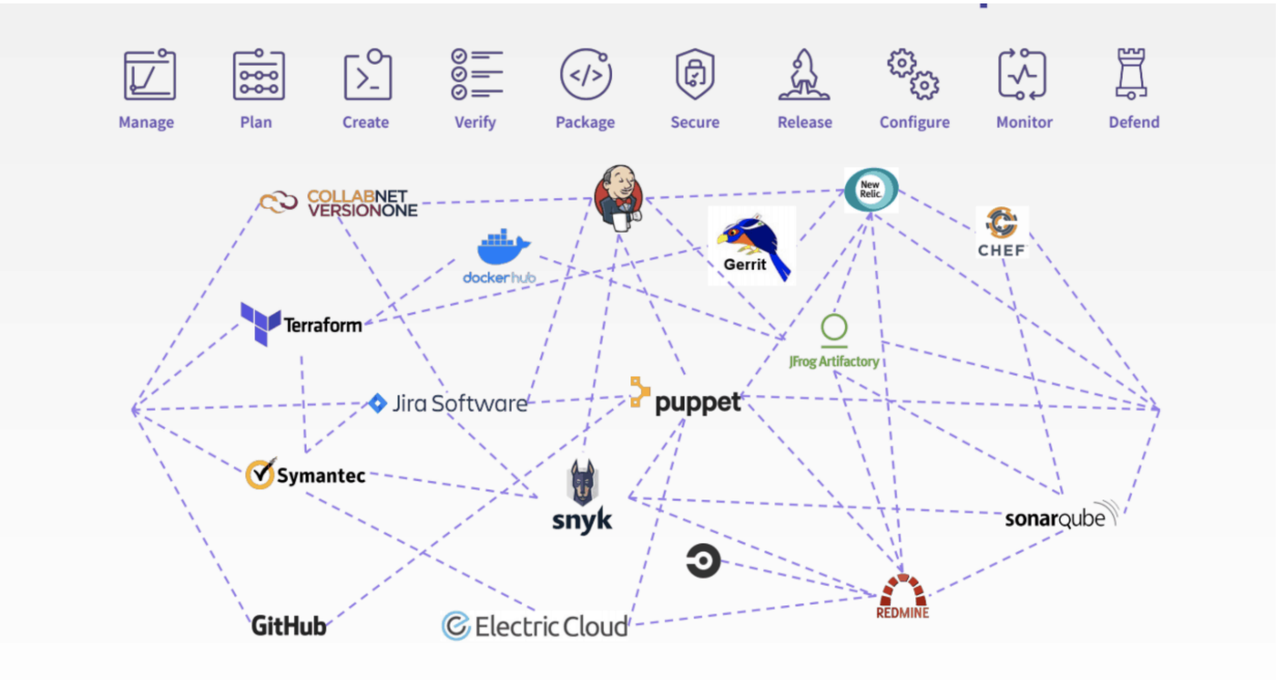
DevOps platforms

- A DevOps platform combines the ability to develop, secure, and operate software in **a single application** so everyone involved in the software development process - from a product manager to an ops pro - can seamlessly work together to release software faster.

Jenkins

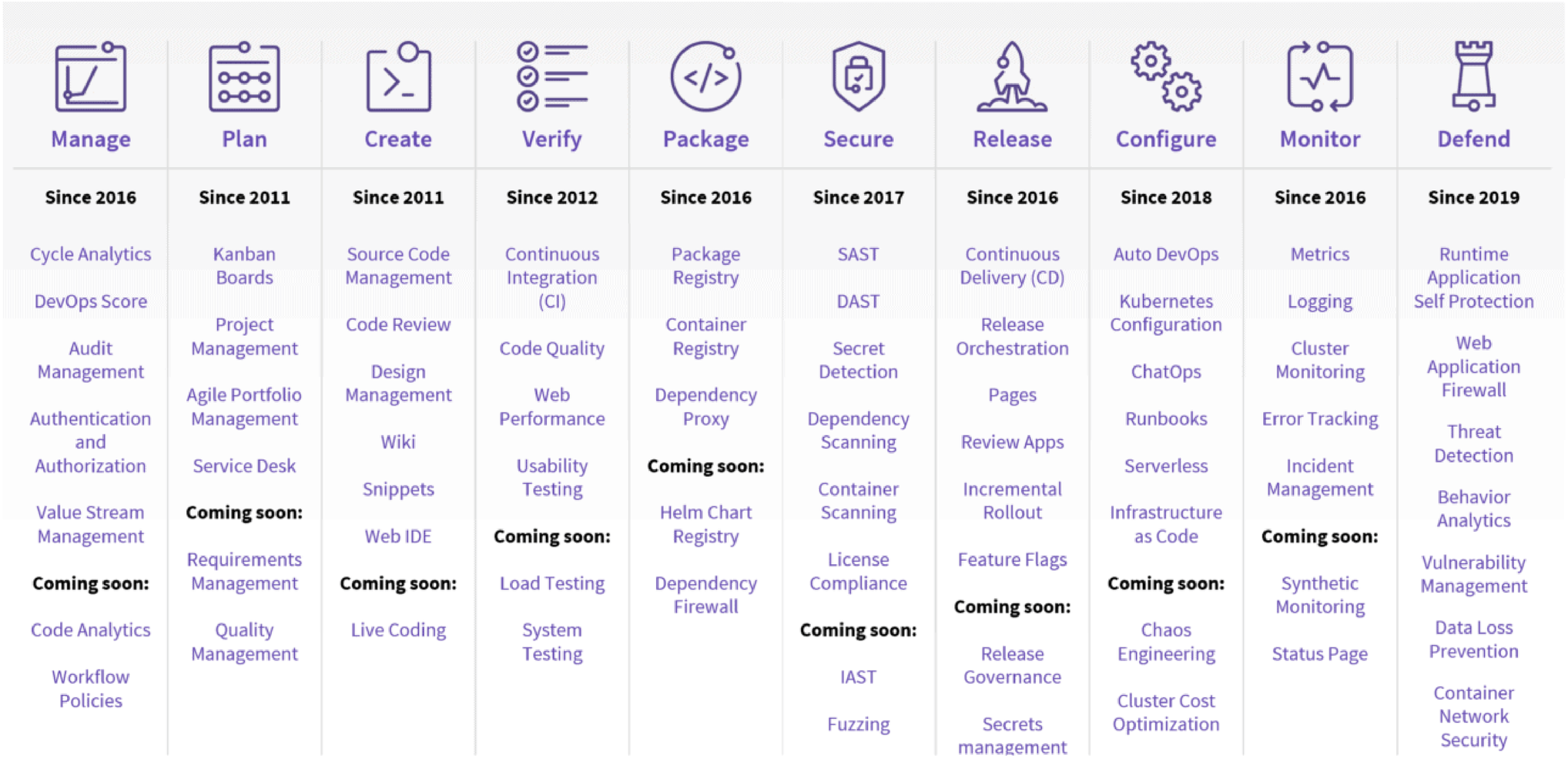
GitLab

GitHub



Source: https://www.techzine.eu/blogs/devops/43256/gitlab-is-a-devsecops-platform-with-open-source-at-its-core/

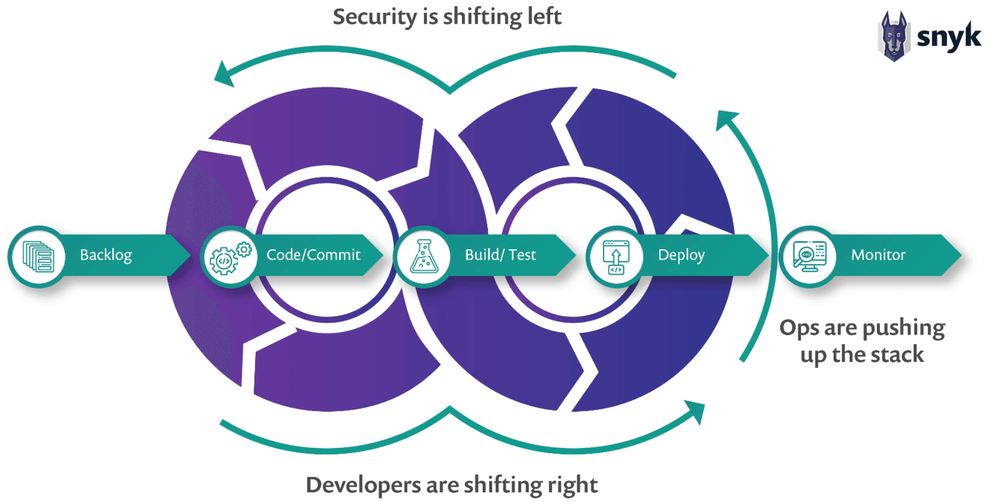
GitLab’s DevSecOps platform

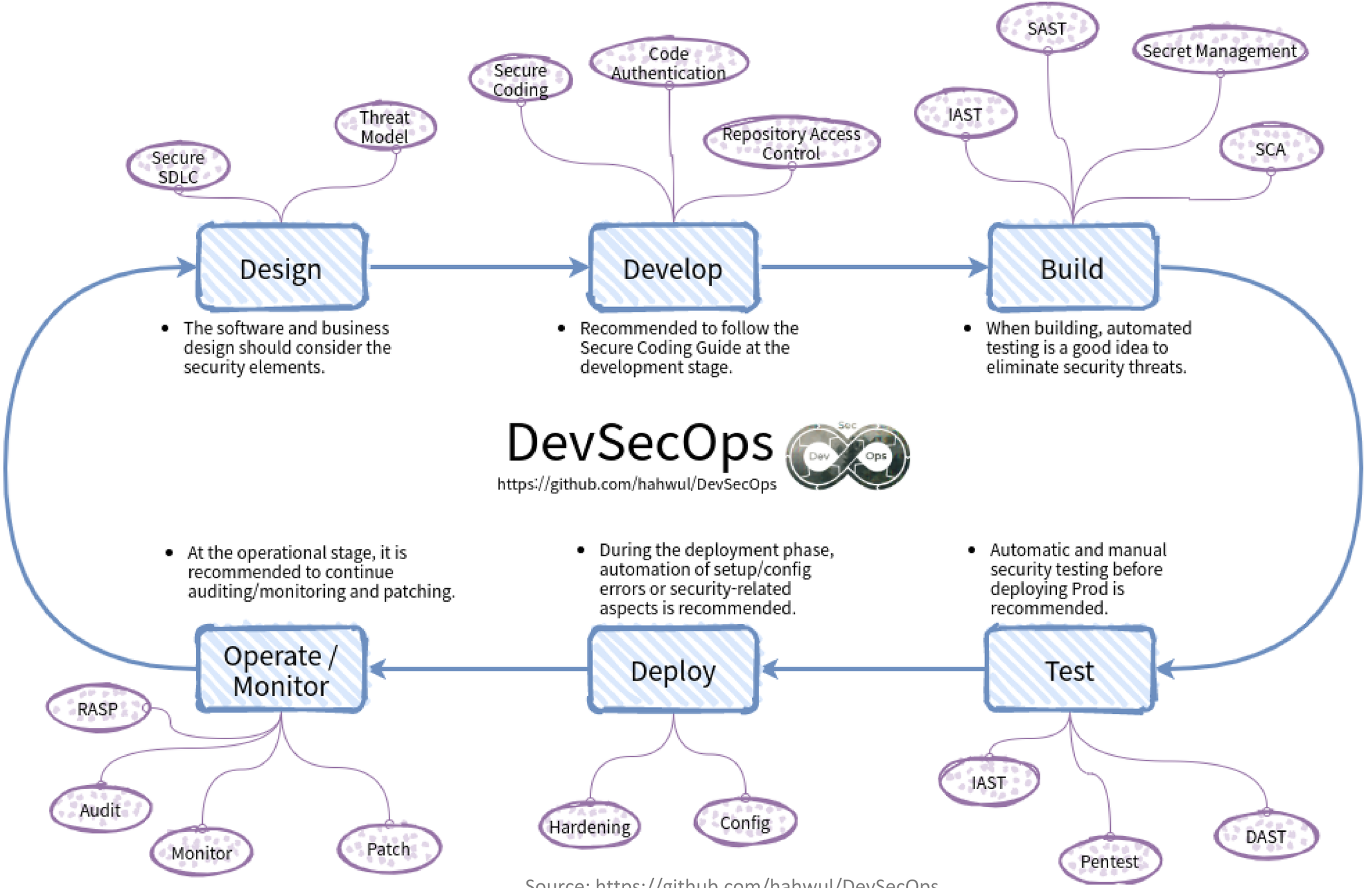


Source: https://www.techzine.eu/blogs/devops/43256/gitlab-is-a-devsecops-platform-with-open-source-at-its-core/

DevSecOps

- DevSecOps stands for **development**, **security**, and **operations**. It's an approach to culture, automation, and platform design that **integrates security as a shared responsibility throughout the entire IT lifecycle**.
- **Shift-Left Security** is the practice of **moving security checks as early and often in the SDLC as possible** as part of a DevSecOps shift. Vulnerabilities found earlier in development are much easier and cheaper to fix.
- Integrating Security into the CI/CD Pipeline





Source: https://github.com/hahwul/DevSecOps

How to continue

- https://github.com/csantanapr/awesome-learn
- Engineering blogs
  - Netflix Tech Blog
  - Engineering at Meta
  - GitHub Engineering
  - LinkedIN Engineering Blog
- Podcasts
- Conference talks
  - YT- CNCF [Cloud Native Computing Foundation]

Thank you!


LTFE
Laboratorij za telekomunikacije
LMMFE
Laboratorij za multimedijo
IKT
Katedra za informacijske
in komunikacijske
tehnologije
Univerza v Ljubljani
Fakulteta za elektrotehniko