ahomé

Cloud as a Service

Agenda

1. DevOps and Agile
2. Infrastructure as a Code
3. Git and Gitflow
4. CI/CD pipelines
5. Containers and Kubernetes
6. Microservices and API
7. Ansible and AWX
8. Python scripting

DevOps and Agile

What is Agile?

Agile Methodology involves continuous iteration of development and testing in the Software Development LifeCycle process. This software development method emphasizes on iterative, incremental, and evolutionary development.

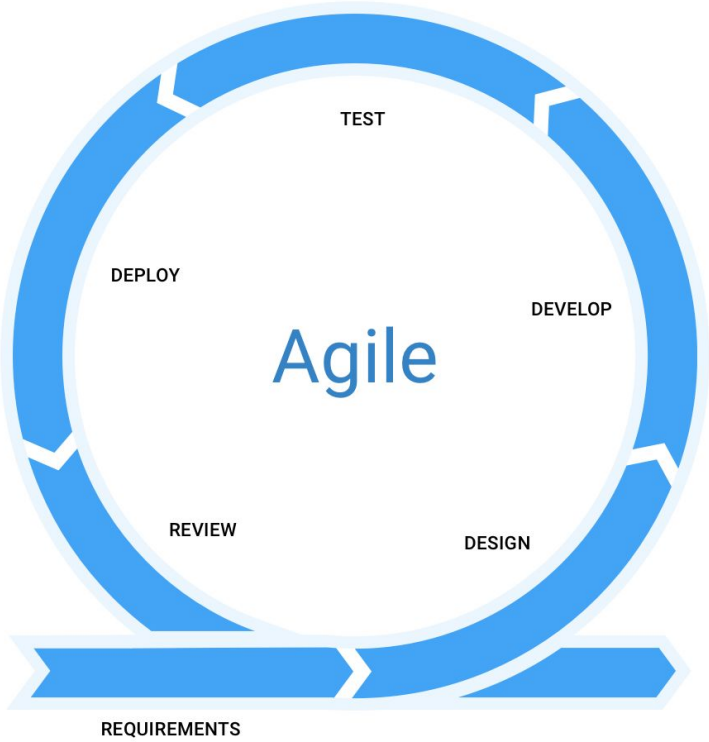
Agile development process breaks the product into smaller pieces and integrates them for final testing. It can be implemented in many ways, including scrum, kanban, scrum, XP, etc.

DevOps and Agile

What is Agile?

Agile addresses gaps in Customer and Developer communications

MVP : Minimum Viable Product



DevOps and Agile

What is DevOps?

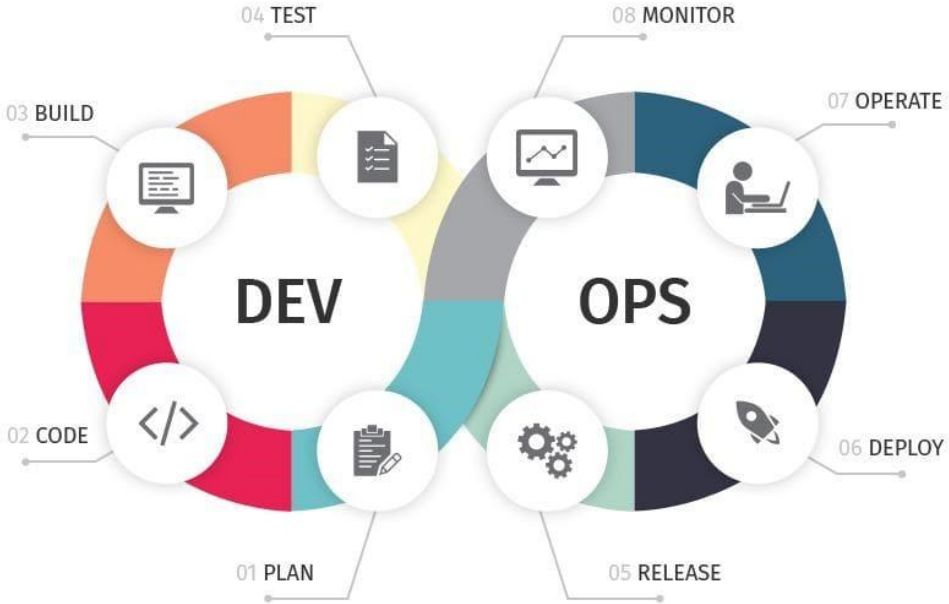
DevOps is a software development method which focuses on communication, integration, and collaboration among IT professionals to enables rapid deployment of products.

DevOps is a culture that promotes collaboration between Development and Operations Team. This allows deploying code to production faster and in an automated way. It helps to increases an organization's speed to deliver application and services. It can be defined as an alignment of development and IT operation.

DevOps and Agile

What is DevOps?

DevOps addresses gaps in Developer and IT Operations communications



DevOps and Agile

IT world is changing... oups!!! Has Changed already

Business requirements:

- ➔ Time to market
- ➔ Flexibility
- ➔ Costs
- ➔ Competitivity

Traditional methods and processes are not working anymore in the new business model

DevOps and Agile

Agility has replaced Waterfall methodology in Developers world.

DevOps is the new word to define Sysadmin and Network Engineers.

What about the infrastructure itself ?

Is the infrastructure ready to handle changes, performance, flexibility, cost reduction?

Infrastructure as a Code

Infrastructure as code (IaC) is the process of managing and provisioning computer data centers through machine-readable definition files, rather than physical hardware configuration or interactive configuration tools The IT infrastructure managed by this comprises both physical equipment such as bare-metal servers as well as virtual machines and associated configuration resources. The definitions may be in a version control system. It can use either scripts or declarative definitions, rather than manual processes, but the term is more often used to promote declarative approaches.

source: Wikipedia

Infrastructure as a Code

Infrastructure as code (IaC) is not just about automation

IaC is a concept that extends beyond simple infrastructure automation.
IaC requires applying DevOps practices to automation scripts to ensure they're free of errors, are able to be redeployed on multiple servers, can be rolled back in case of problems, and can be engaged by both operations and development teams. The use of modern coding systems like Ansible or Puppet is designed to make **IaC environments accessible to anyone with basic knowledge of modern coding techniques and structures.**

Infrastructure as a Code

Infrastructure as code (IaC) is not just about automation

"...IaC environments accessible to anyone with basic knowledge of modern coding techniques and structures".

Big paradigm shift...

So ... IT infrastructure for everyone is possible while we are growing in a very complex environment?

Infrastructure as a Code

Best practices of IaC

Manage infrastructure via source control, thus providing a detailed audit trail for changes.

Apply testing to infrastructure in the form of unit testing, functional testing, and integration testing.

Avoid written documentation, since the code itself will document the state of the machine. This is particularly powerful because it means, for the first time, that infrastructure documentation is always up to date.

Enable collaboration around infrastructure configuration and provisioning, most notably between dev and ops.

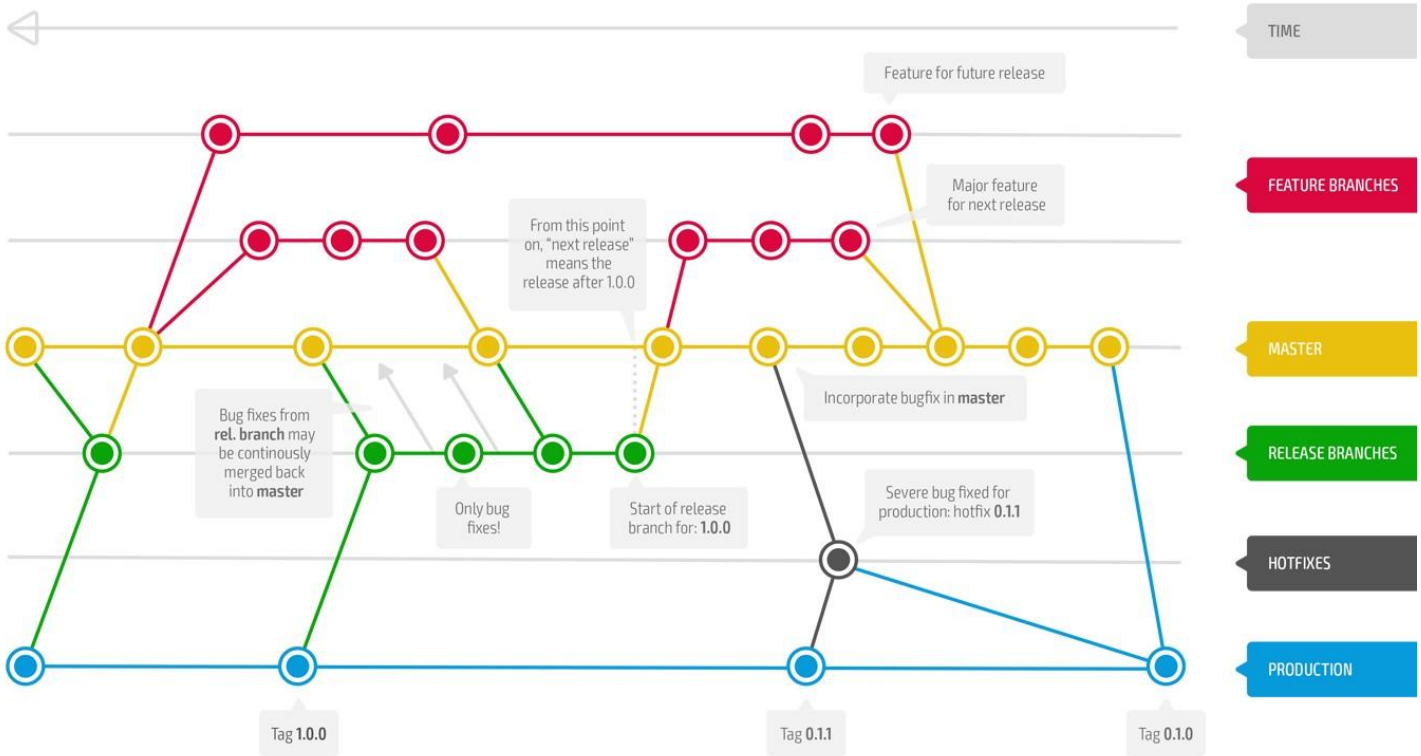
Git and Git flow

Git definition

Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency.

Git and Git flow

Git-flow



CI/CD pipeline

Why CI/CD ?

As everything is a code, if you want your engineering team to deliver bug-free code at high velocity? A fast and reliable CI/CD pipeline is crucial for doing that sustainably over time.

What is a CI/CD pipeline?

A CI/CD pipeline helps you automate steps in your software delivery process, such as initiating code builds, running automated tests, and deploying to a staging or production environment. Automated pipelines remove manual errors, provide standardized development feedback loops and enable fast product iterations.

CI/CD pipeline

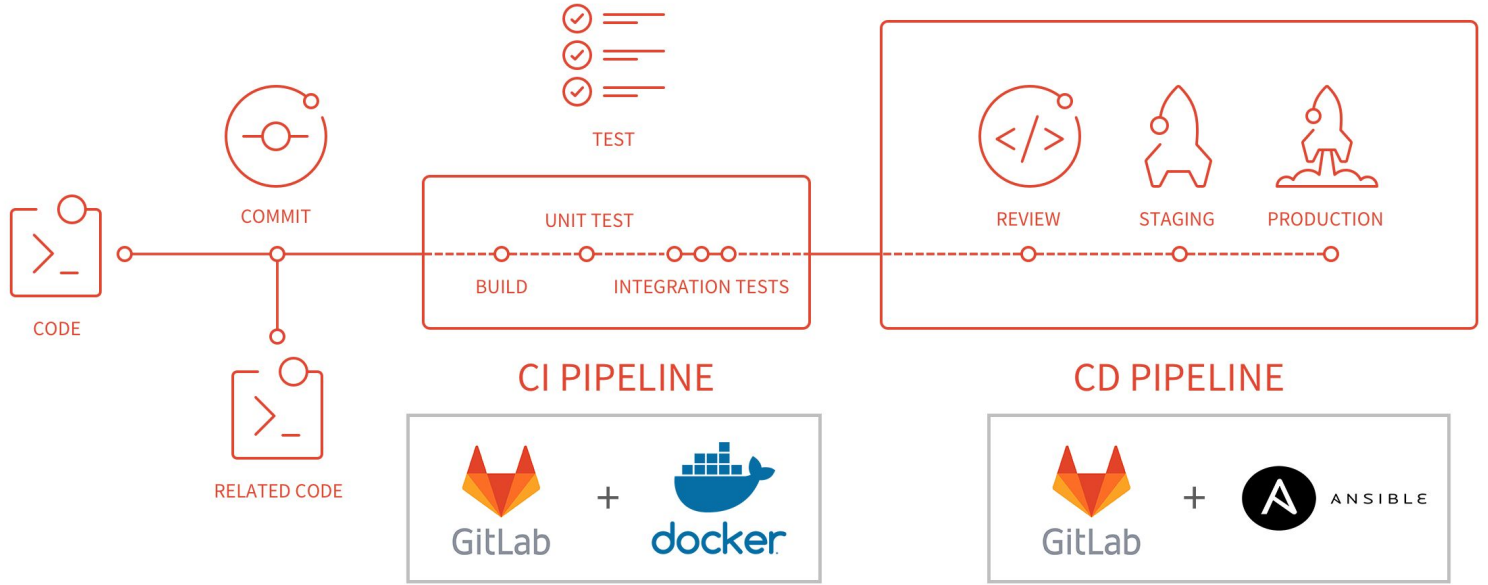
CI (Continuous Integration)

Continuous Integration, is a software development practice in which all developers merge code changes in a central repository multiple times a day. CD stands for Continuous Delivery, which on top of Continuous Integration adds the practice of automating the entire software release process.

CD (Continuous Delivery)

Continuous Delivery includes infrastructure provisioning and deployment, which may be manual and consist of multiple stages. What's important is that all these processes are fully automated, with each run fully logged and visible to the entire team.

CI/CD pipeline



Example of CI/CD with gitlab

CI/CD pipeline

Wait a minute... is it Docker in the CI pipeline ?

Why do we need to integrate Docker or a container based application like Kubernetes in our pipeline?

Make sense for developers... but why it's important for DevOps? Or even Infrastructure as a Code?

CI/CD pipeline

benefits of pipelines

- Stay focused on writing code and monitoring the behavior of the system in production.
- QA and product stakeholders have easy access to the latest, or any, version of the system.
- Product updates are not stressful.
- Logs of all code changes, test and deployments are available for inspection at any time.
- Rolling back to a previous version in the event of a problem is a routine push-button action.
- A fast feedback loop helps build an organizational culture of learning and responsibility.

CI/CD pipeline

What makes a good pipeline?

- Stay focused on writing code and monitoring the behavior of the system in production.
- QA and product stakeholders have easy access to the latest, or any, version of the system.
- Product updates are not stressful.
- Logs of all code changes, test and deployments are available for inspection at any time.
- Rolling back to a previous version in the event of a problem is a routine push-button action.
- A fast feedback loop helps build an organizational culture of learning and responsibility.

CI/CD pipeline

A good CI/CD pipeline is fast, reliable and accurate.

- How quickly do we get feedback on the correctness of our work?
- How long does it take us to build, test and deploy a simple code commit?
- Do our CI/CD pipelines scale to meet development demands in real time?
- How quickly can we set up a new pipeline?
- **Reliability**
  A reliable pipeline always produces the same output for a given input, and with no oscillations in runtime. Intermittent failures cause intense frustration among developers.
- **Accuracy:**
  Any degree of automation is a positive change. However, the job is not fully complete until the CI/CD pipeline accurately runs and visualizes the entire software delivery process.

CI/CD pipeline

Good things to do

- When the master is broken, drop what you're doing and fix it. Maintain a "no broken windows" policy on the pipeline.
- Run fast and fundamental tests first.
- Always use exactly the same environment.
- Build in quality checking.
- Include pull requests.
- Peer-review each pull request.

Containers and Kubernetes

What is a container?

A standardized unit of software

Package Software into Standardized Units for Development, Shipment and Deployment

A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.

Containers and Kubernetes

What is Docker?

Docker containers that run on Docker Engine:

Standard: Docker created the industry standard for containers, so they could be portable anywhere

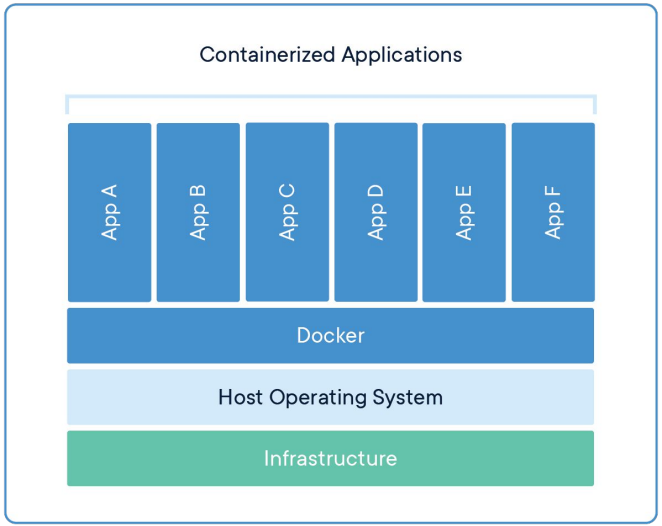
Lightweight: Containers share the machine's OS system kernel and therefore do not require an OS per application, driving higher server efficiencies and reducing server and licensing costs

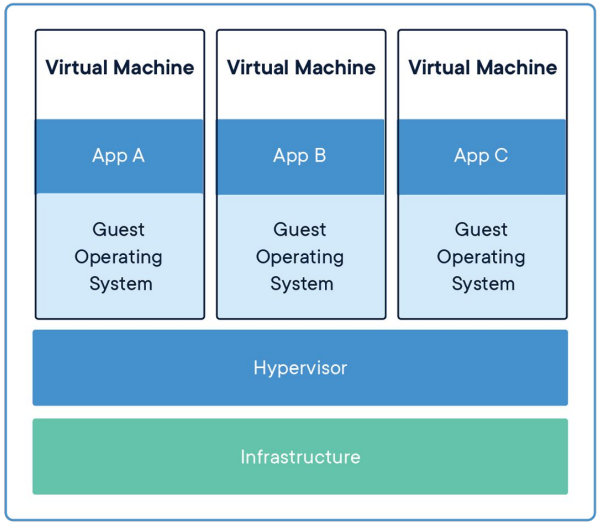
Secure: Applications are safer in containers and Docker provides the strongest default

isolation capabilities in the industry

Containers and Kubernetes

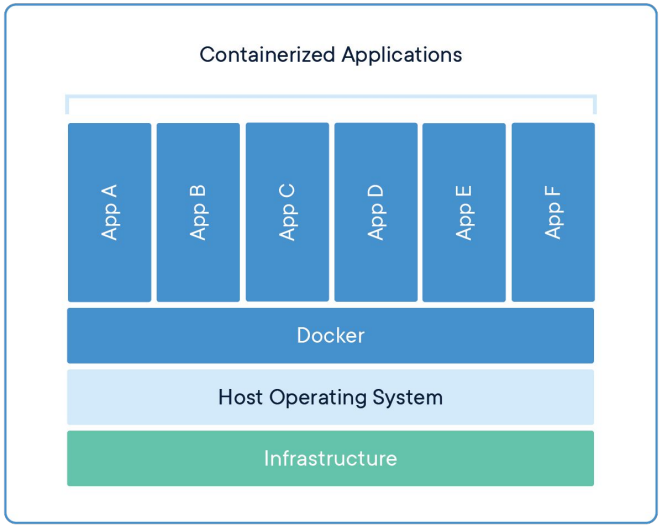
Docker vs Virtual Machine





Containers and Kubernetes

Docker vs Virtual Machine

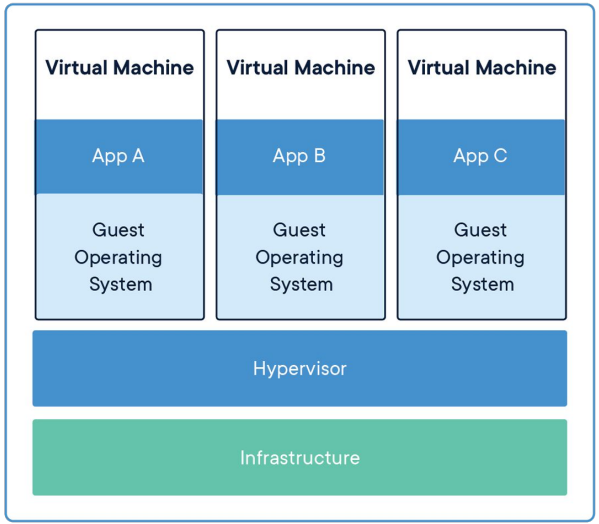


CONTAINERS

Containers are an abstraction at the app layer that packages code and dependencies together. Multiple containers can run on the same machine and share the OS kernel with other containers, each running as isolated processes in user space. Containers take up less space than VMs (container images are typically tens of MBs in size), can handle more applications and require fewer VMs and Operating systems.

Containers and Kubernetes

Docker vs Virtual Machine



VIRTUAL MACHINES

Virtual machines (VMs) are an abstraction of physical hardware turning one server into many servers. The hypervisor allows multiple VMs to run on a single machine. Each VM includes a full copy of an operating system, the application, necessary binaries and libraries - taking up tens of GBs. VMs can also be slow to boot.

Containers and Kubernetes

What is Kubernetes?

Kubernetes is a portable, extensible open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.

Google open-sourced the Kubernetes project in 2014. Kubernetes builds upon a decade and a half of experience that Google has with running production workloads at scale, combined with best-of-breed ideas and practices from the community.

Containers and Kubernetes

Why do I need Kubernetes?

Why do I need Kubernetes and what can it do
Kubernetes has a number of features. It can be thought of as:

- a container platform
- a microservices platform
- a portable cloud platform and a lot more.

Kubernetes provides a container-centric management environment. It orchestrates computing, networking, and storage infrastructure on behalf of user workloads. This provides much of the simplicity of Platform as a Service (PaaS) with the flexibility of Infrastructure as a Service (IaaS), and enables portability across infrastructure providers.

API and Microservices

What is API (application programming interface)?

a set of subroutine definitions, communication protocols, and tools for building software. In general terms, it is a set of clearly defined methods of communication between various components.

An easy way to think about an API is to think of it as a contract of actions you can request for a particular service. APIs are in use today in a multitude of web applications, such as social media, banking software, and much more. The standardized contract allows for external applications to interface with another.

API and Microservices

Common API Actions

In today's world, APIs are usually developed using a RESTful style. These APIs will have a series of verbs associating with HTTP actions, like the following:

GET (get a single item or a collection)
POST (add an item to a collection)
PUT (edit an item that already exists in a collection)
DELETE (delete an item in a collection)

The advantage of this consistency through different applications is having a standard when performing various actions. The four different HTTP verbs above correlate with the common CRUD capabilities that many applications use today. When working with different APIs in one application, this makes for a recognizable way to understand the implications of the actions taken across different interfaces.

API and Microservices

What is Microservices

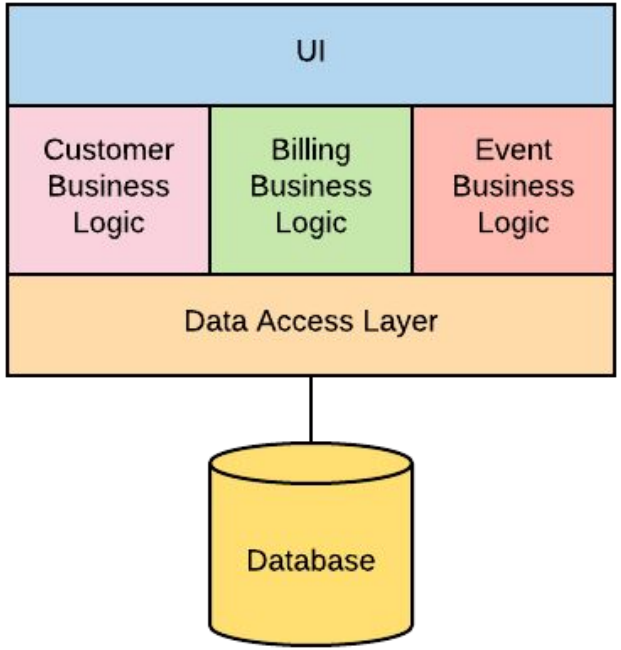
Wikipedia defines a microservice as:

a software development technique—a variant of the service-oriented architecture (SOA) architectural style that structures an application as a collection of loosely coupled services. In a microservices architecture, services are fine-grained and the protocols are lightweight.

API and Microservices

The Precursor to Microservices: Monoliths

In the early days of software development (and continuing in many large enterprise environments today), there's the concept of a monolith. A monolith is a single application that holds a full collection of functionality, serving as one place to store everything. Architecturally, it looks like this:



API and Microservices

The Precursor to Microservices: Monoliths

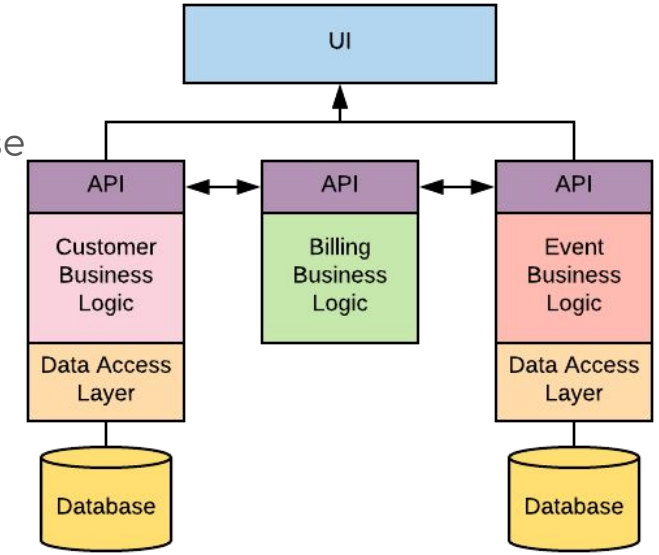
All of the components of the application reside in one area, including the UI layer, the business logic layer, and the data access layer. Building applications in a monolith is an easy and natural process, and most projects start this way. But adding functionality to the codebase causes an increase in both the size and complexity of the monolith, and allowing a monolith to grow large comes with disadvantages over time. Some of these include:

Risk of falling into the big ball of mud anti-pattern, not having any rhyme or reason in their architecture and difficult to understand from a high level.
Restriction of the technology stack inside the monolith. Especially as the application grows, the ability to move to a different technology stack becomes more and more difficult, even when the technology proves to no longer be the best choice.
Making changes to the codebase affects the entire application, no matter how small. For example, if just one of the business logic sections is receiving constant changes, this forces redeployment of the entire application, wasting time and increasing risk.

API and Microservices

Microservices:

The monolith example from above and convert it to use microservices. In that case, the application architecture would change to look like this:



API and Microservices

Microservices:

With this kind of architecture comes a whole host of advantages:

It's easier to separate concerns. These boundaries between areas help with development (you only need to concern yourself with your microservice, not the entire application) and with understanding the architecture of the application.

Unlike with a monolith, a microservice can use a different tech stack as needed. Considering rewriting everything in a new language? Just change one microservice to use the new tech stack, assess the benefits gained, and determine whether to proceed.

Deployments of the application as a whole become more focused. Microservices give you the flexibility to deploy different services as needed.

API and Microservices

Hold-on... are we still talking about infrastructure?

Thank you.