CHAPTER 01

Course Coverage

DevOps ➔ The culture of collaboration between Dev and Ops.

DevOps Practices ➔ The practises which support the goals of DevOps culture.

DevOps Tools ➔ The tools that help implement DevOps practices

DevOps and the Cloud ➔ The closer relationships between DevOps and the cloud

Agile Software Development

DevOps grew out of the Agile software development movement

Agile seeks to develop software in small, frequent cycles in order to deliver functionality to customers quickly and quickly respond to changing business goals.

DevOps and Agile go hand-in-hand

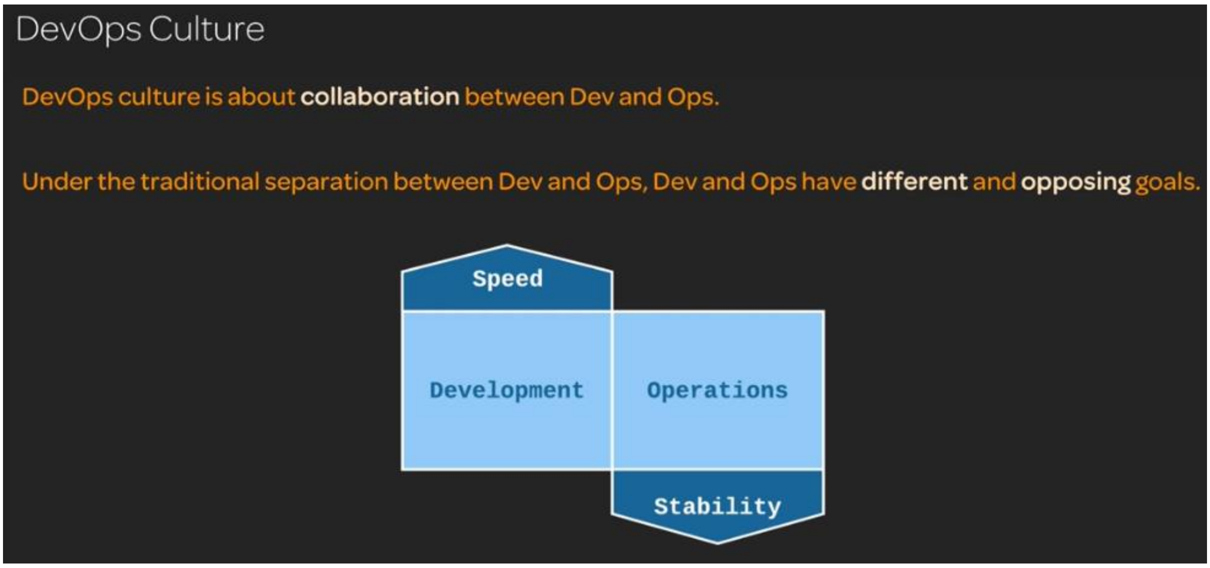
2007 Patrick Debois + Andrew shafer

2009 John Allspaw + Paul Hammond livestream

Patric hosted first DevOps Days conference in Belgium

CHAPTER 02

DevOps Essentials



DevOps Culture

DevOps culture is about **collaboration** between Dev and Ops.

Under the traditional separation between Dev and Ops, Dev and Ops have **different** and **opposing** goals.

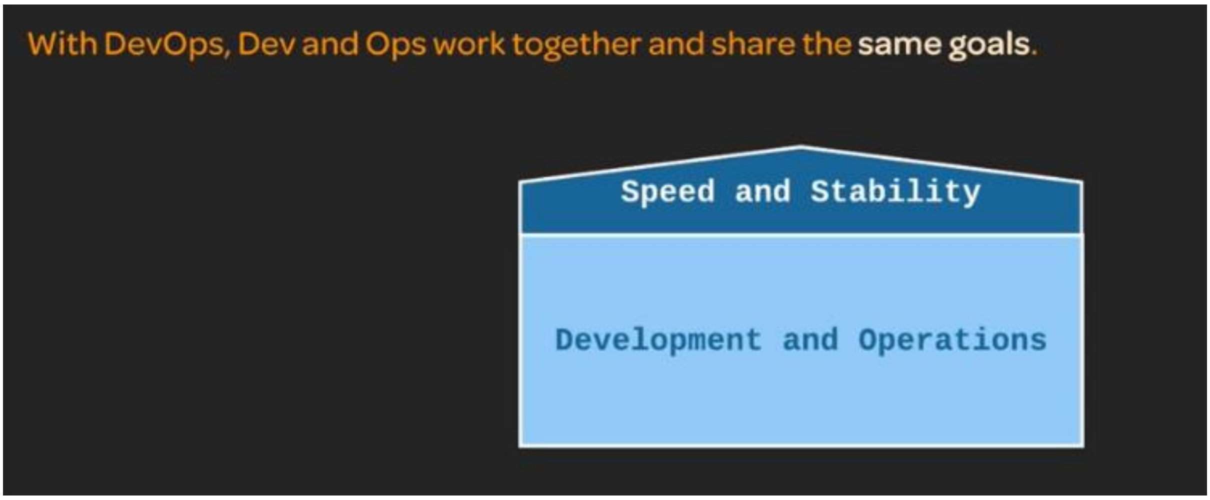
Speed

Development

Operations

Stability

After DevOps



With DevOps, Dev and Ops work together and share the **same goals**.

Speed and Stability

Development and Operations

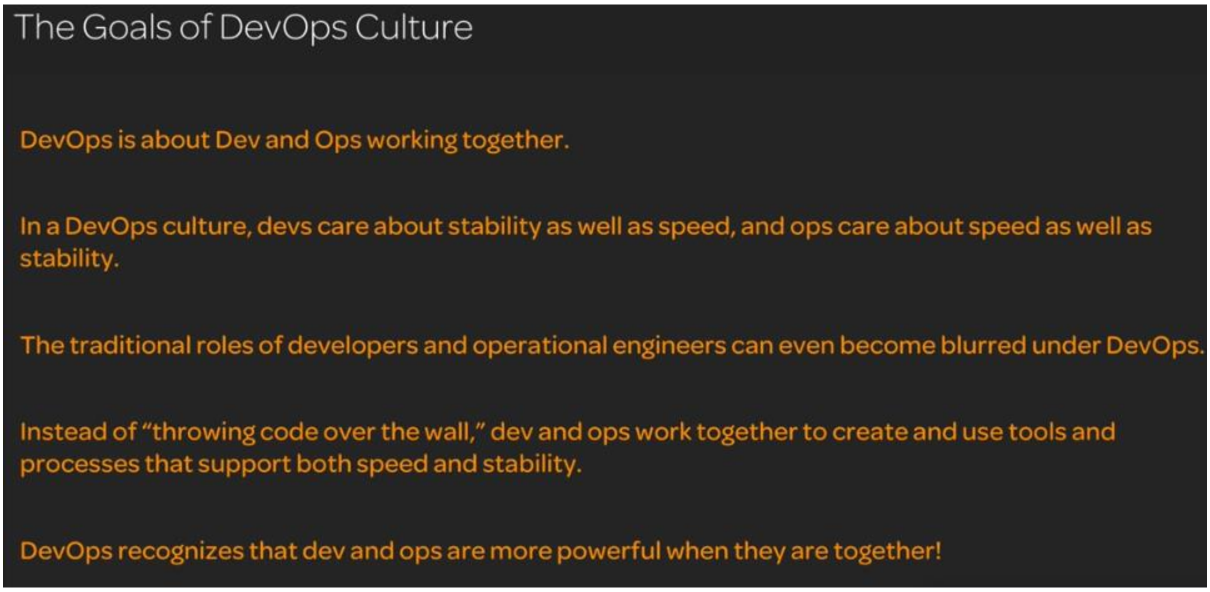
Pros

Dev and ops are playing on the same team

Dev and Ops share the same goals ➔ Speed and Stability

DevOps Goals

- Fast time to market (TTM)
- Few productions failures
- Immediate recovery from failures



The Goals of DevOps Culture

DevOps is about Dev and Ops working together.

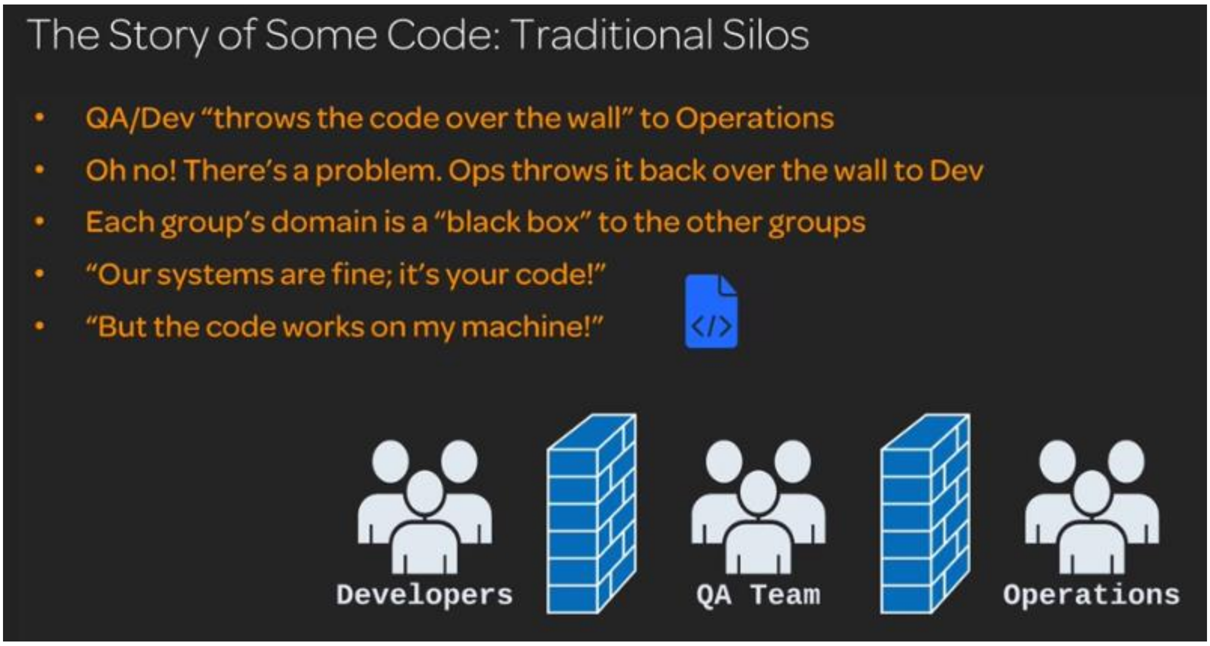
In a DevOps culture, devs care about stability as well as speed, and ops care about speed as well as stability.

The traditional roles of developers and operational engineers can even become blurred under DevOps.

Instead of "throwing code over the wall," dev and ops work together to create and use tools and processes that support both speed and stability.

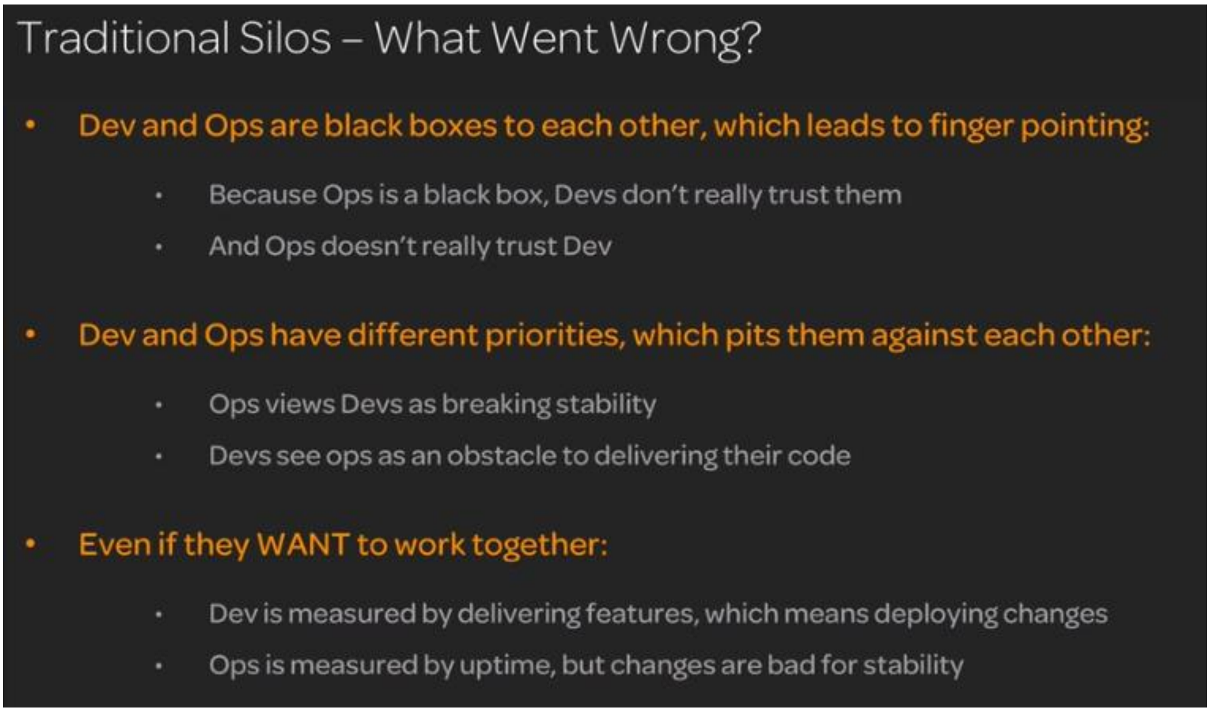
DevOps recognizes that dev and ops are more powerful when they are together!

DevOps vs Traditional Silos



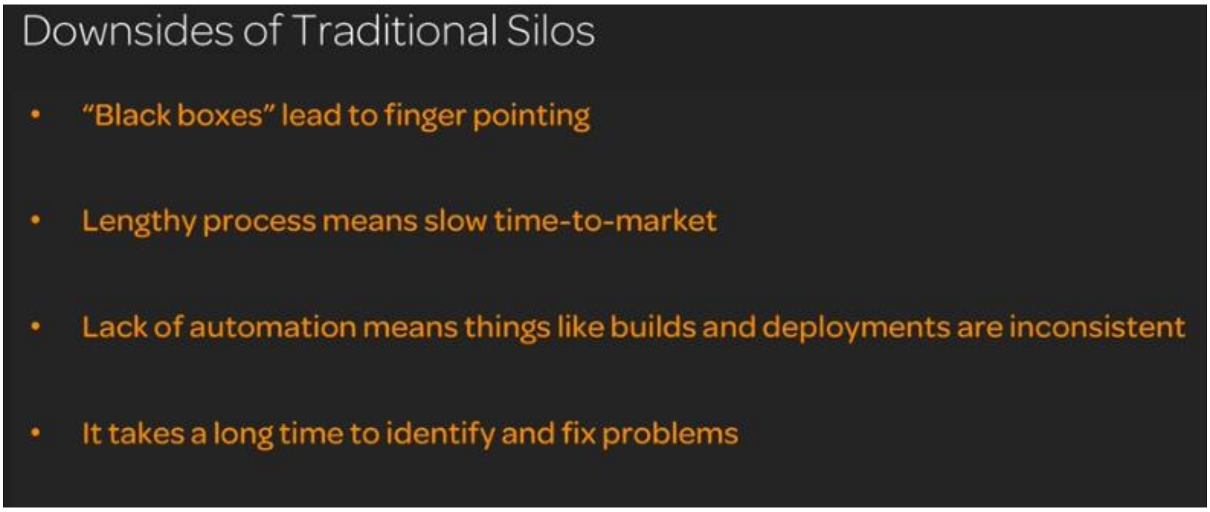
The Story of Some Code: Traditional Silos

- QA/Dev "throws the code over the wall" to Operations
- Oh no! There's a problem. Ops throws it back over the wall to Dev
- Each group's domain is a "black box" to the other groups
- "Our systems are fine; it's your code!"
- "But the code works on my machine!"



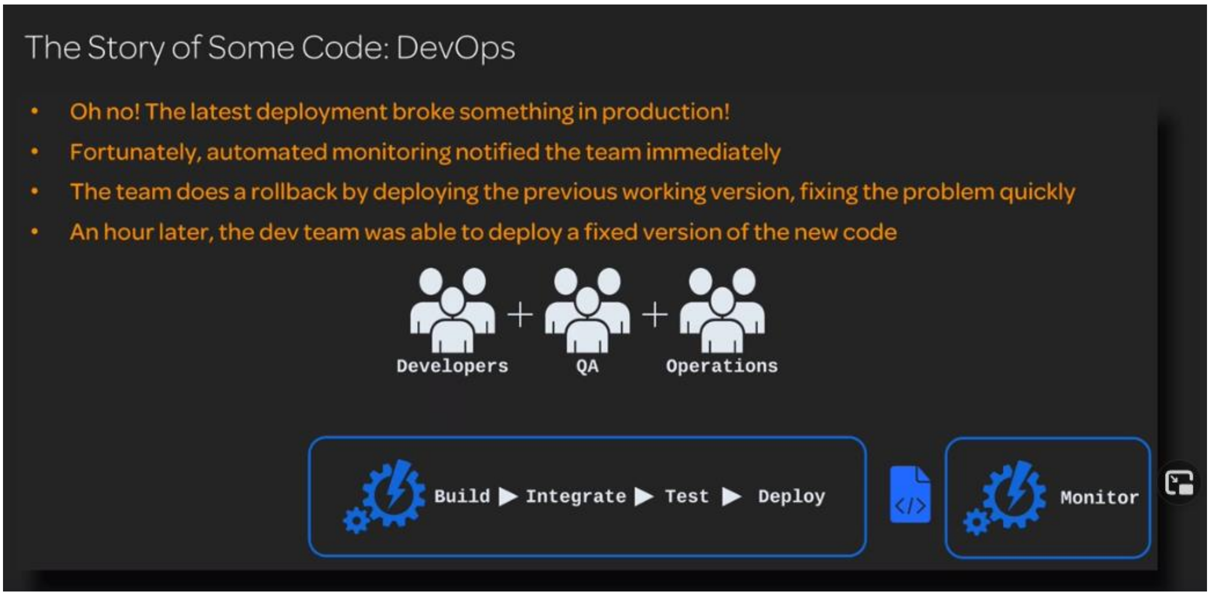
Traditional Silos – What Went Wrong?

- Dev and Ops are black boxes to each other, which leads to finger pointing:
  - Because Ops is a black box, Devs don't really trust them
  - And Ops doesn't really trust Dev
- Dev and Ops have different priorities, which pits them against each other:
  - Ops views Devs as breaking stability
  - Devs see ops as an obstacle to delivering their code
- Even if they WANT to work together:
  - Dev is measured by delivering features, which means deploying changes
  - Ops is measured by uptime, but changes are bad for stability



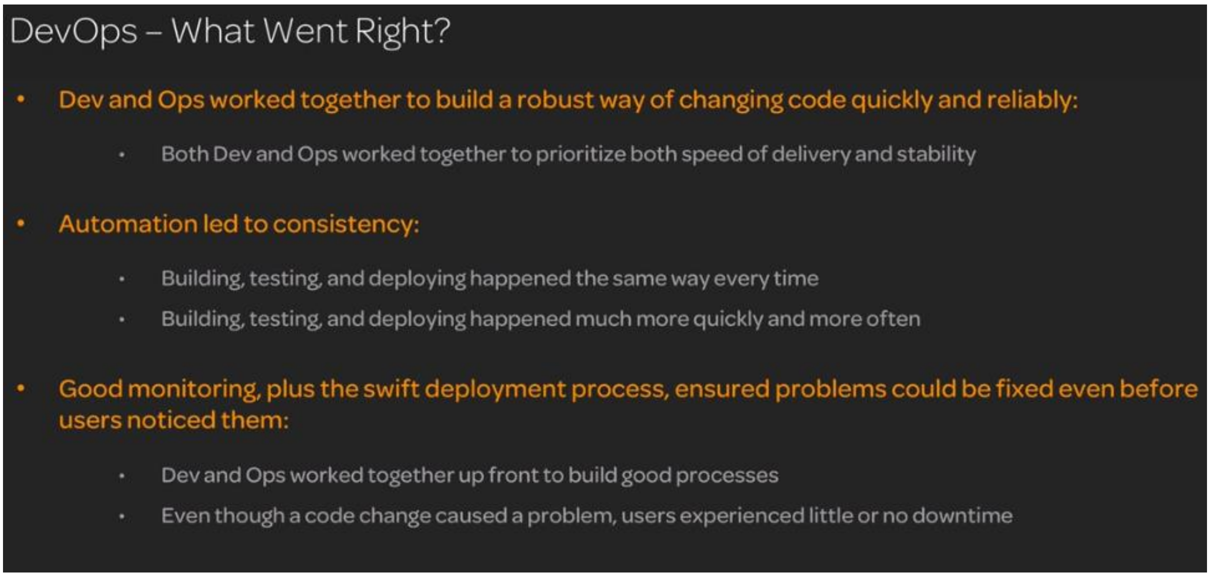
Downsides of Traditional Silos

- "Black boxes" lead to finger pointing
- Lengthy process means slow time-to-market
- Lack of automation means things like builds and deployments are inconsistent
- It takes a long time to identify and fix problems



The Story of Some Code: DevOps

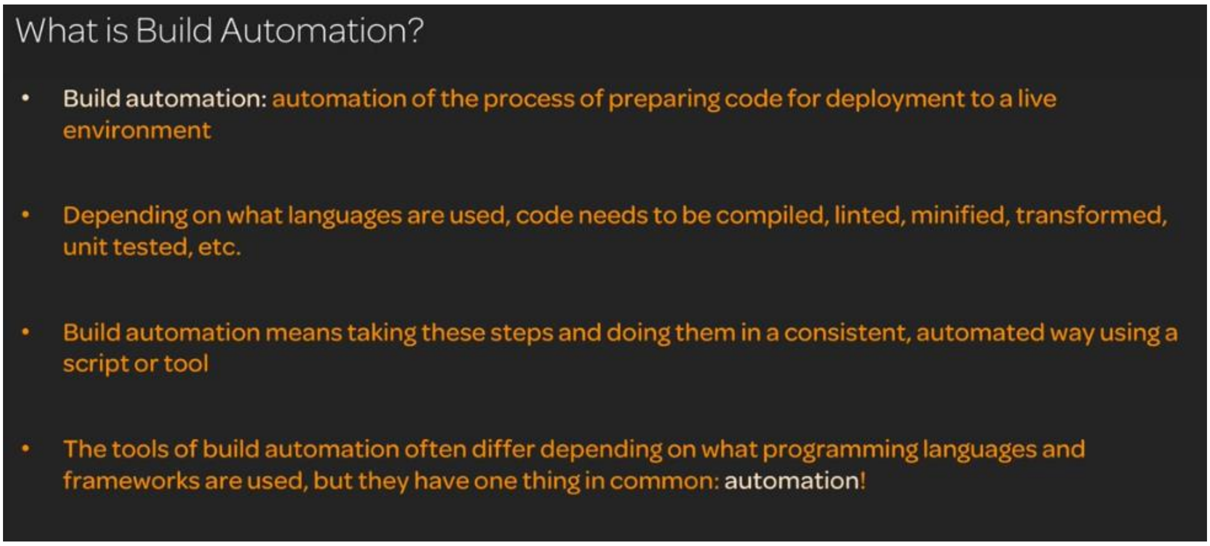
- Oh no! The latest deployment broke something in production!
- Fortunately, automated monitoring notified the team immediately
- The team does a rollback by deploying the previous working version, fixing the problem quickly
- An hour later, the dev team was able to deploy a fixed version of the new code



DevOps – What Went Right?

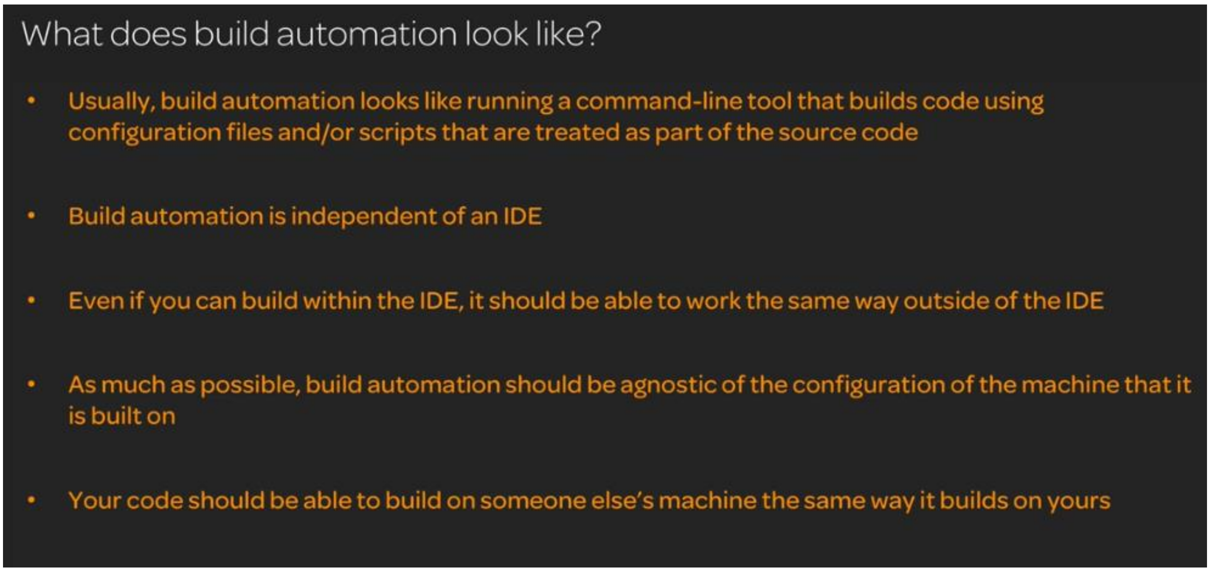
- Dev and Ops worked together to build a robust way of changing code quickly and reliably:
  - Both Dev and Ops worked together to prioritize both speed of delivery and stability
- Automation led to consistency:
  - Building, testing, and deploying happened the same way every time
  - Building, testing, and deploying happened much more quickly and more often
- Good monitoring, plus the swift deployment process, ensured problems could be fixed even before users noticed them:
  - Dev and Ops worked together up front to build good processes
  - Even though a code change caused a problem, users experienced little or no downtime

Continuous Build



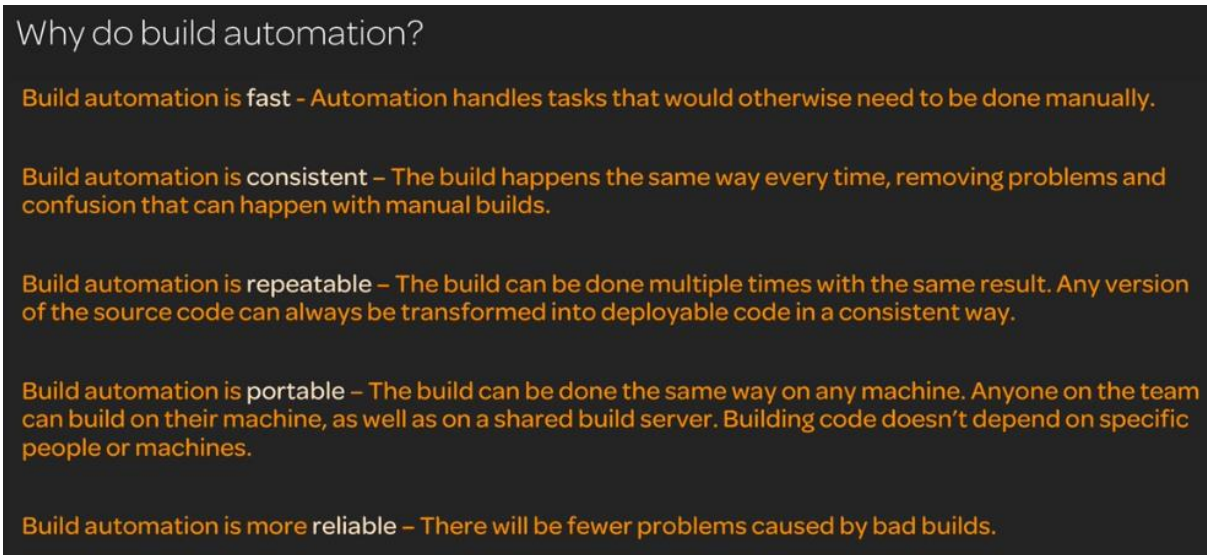
What is Build Automation?

- Build automation: automation of the process of preparing code for deployment to a live environment
- Depending on what languages are used, code needs to be compiled, linted, minified, transformed, unit tested, etc.
- Build automation means taking these steps and doing them in a consistent, automated way using a script or tool
- The tools of build automation often differ depending on what programming languages and frameworks are used, but they have one thing in common: automation!



What does build automation look like?

- Usually, build automation looks like running a command-line tool that builds code using configuration files and/or scripts that are treated as part of the source code
- Build automation is independent of an IDE
- Even if you can build within the IDE, it should be able to work the same way outside of the IDE
- As much as possible, build automation should be agnostic of the configuration of the machine that it is built on
- Your code should be able to build on someone else's machine the same way it builds on yours



Why do build automation?

Build automation is **fast** - Automation handles tasks that would otherwise need to be done manually.

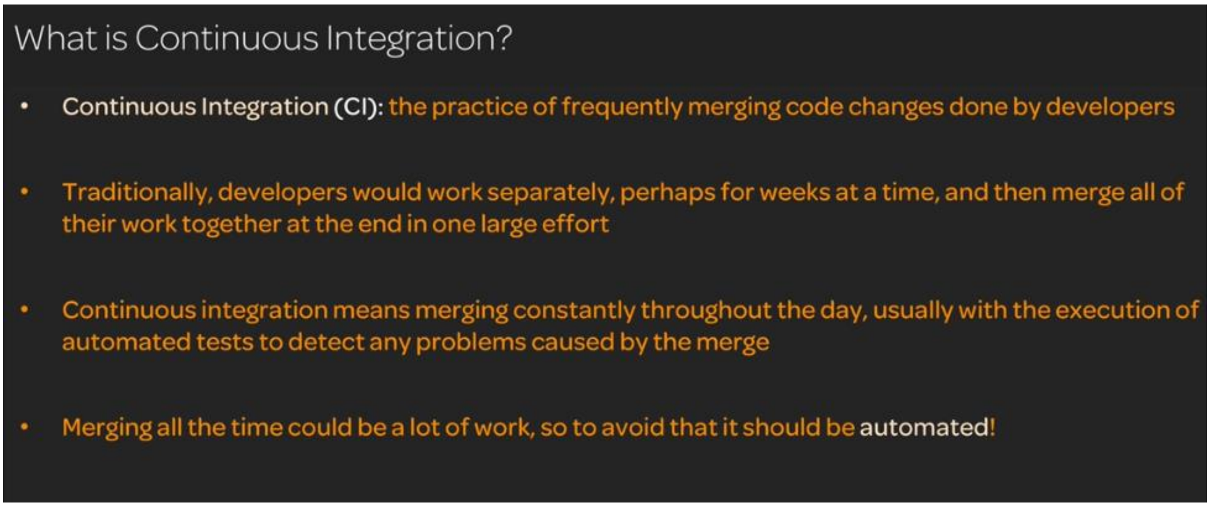
Build automation is **consistent** – The build happens the same way every time, removing problems and confusion that can happen with manual builds.

Build automation is **repeatable** – The build can be done multiple times with the same result. Any version of the source code can always be transformed into deployable code in a consistent way.

Build automation is **portable** – The build can be done the same way on any machine. Anyone on the team can build on their machine, as well as on a shared build server. Building code doesn't depend on specific people or machines.

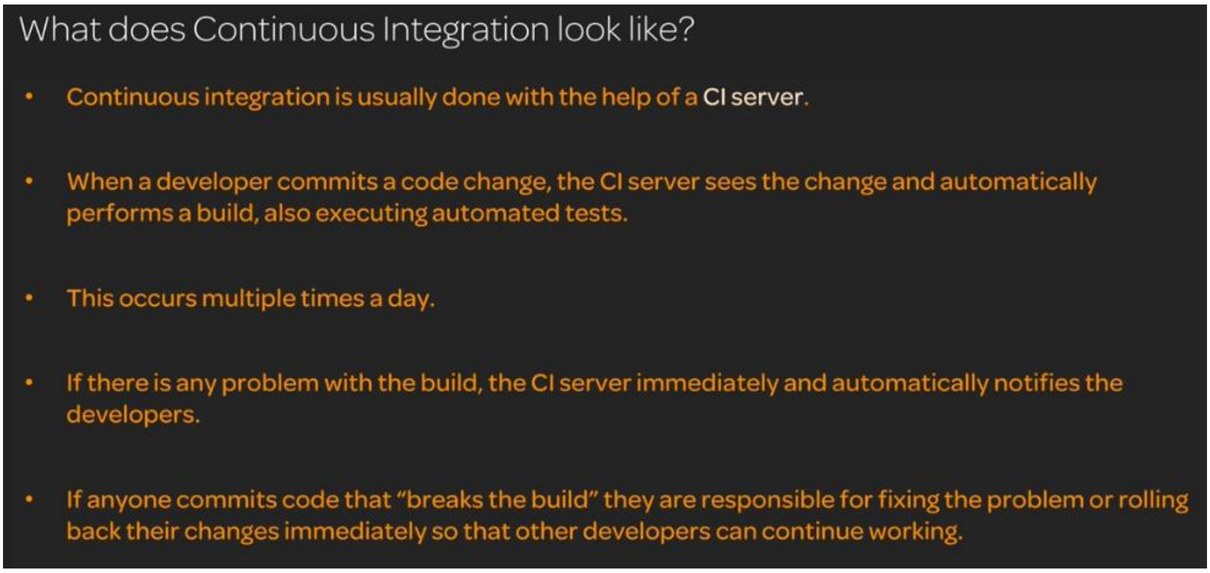
Build automation is more **reliable** – There will be fewer problems caused by bad builds.

Continuous Integration



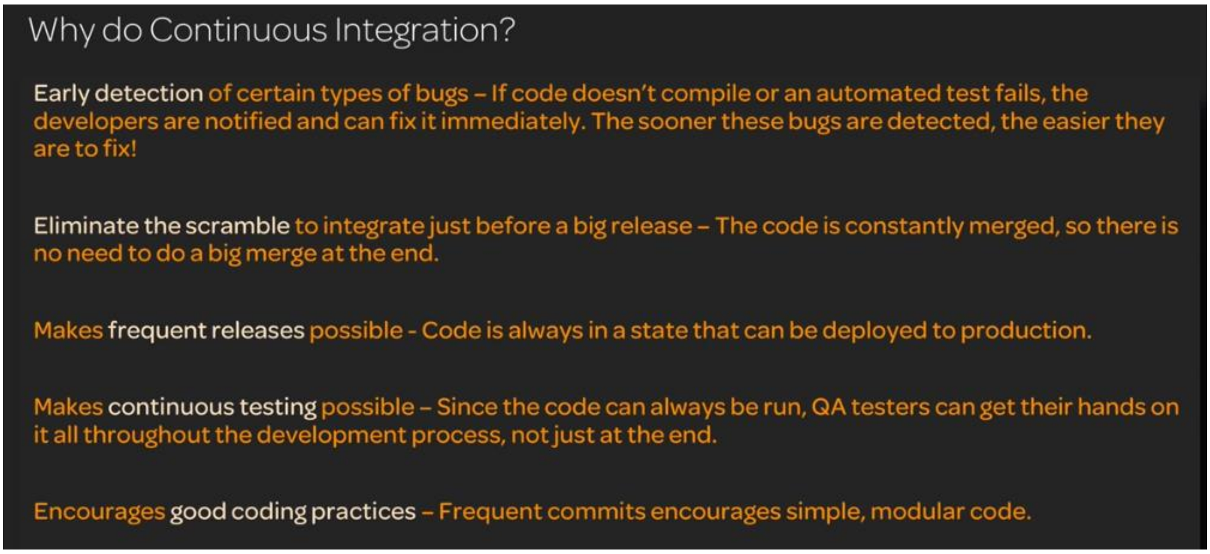
What is Continuous Integration?

- **Continuous Integration (CI):** the practice of frequently merging code changes done by developers
- Traditionally, developers would work separately, perhaps for weeks at a time, and then merge all of their work together at the end in one large effort
- Continuous integration means merging constantly throughout the day, usually with the execution of automated tests to detect any problems caused by the merge
- Merging all the time could be a lot of work, so to avoid that it should be **automated**!



What does Continuous Integration look like?

- Continuous integration is usually done with the help of a **CI server**.
- When a developer commits a code change, the CI server sees the change and automatically performs a build, also executing automated tests.
- This occurs multiple times a day.
- If there is any problem with the build, the CI server immediately and automatically notifies the developers.
- If anyone commits code that "breaks the build" they are responsible for fixing the problem or rolling back their changes immediately so that other developers can continue working.



Why do Continuous Integration?

Early detection of certain types of bugs – If code doesn't compile or an automated test fails, the developers are notified and can fix it immediately. The sooner these bugs are detected, the easier they are to fix!

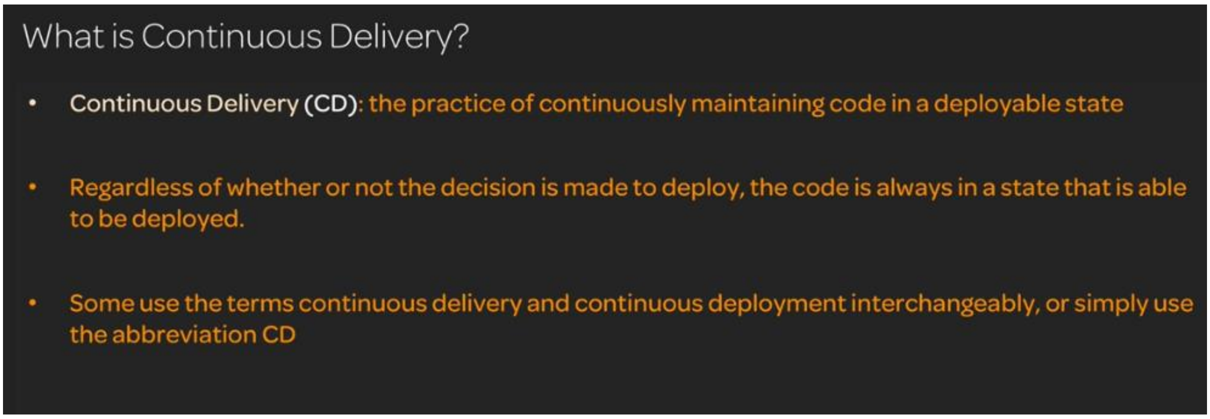
Eliminate the scramble to integrate just before a big release – The code is constantly merged, so there is no need to do a big merge at the end.

Makes **frequent releases** possible - Code is always in a state that can be deployed to production.

Makes **continuous testing** possible – Since the code can always be run, QA testers can get their hands on it all throughout the development process, not just at the end.

Encourages **good coding practices** – Frequent commits encourages simple, modular code.

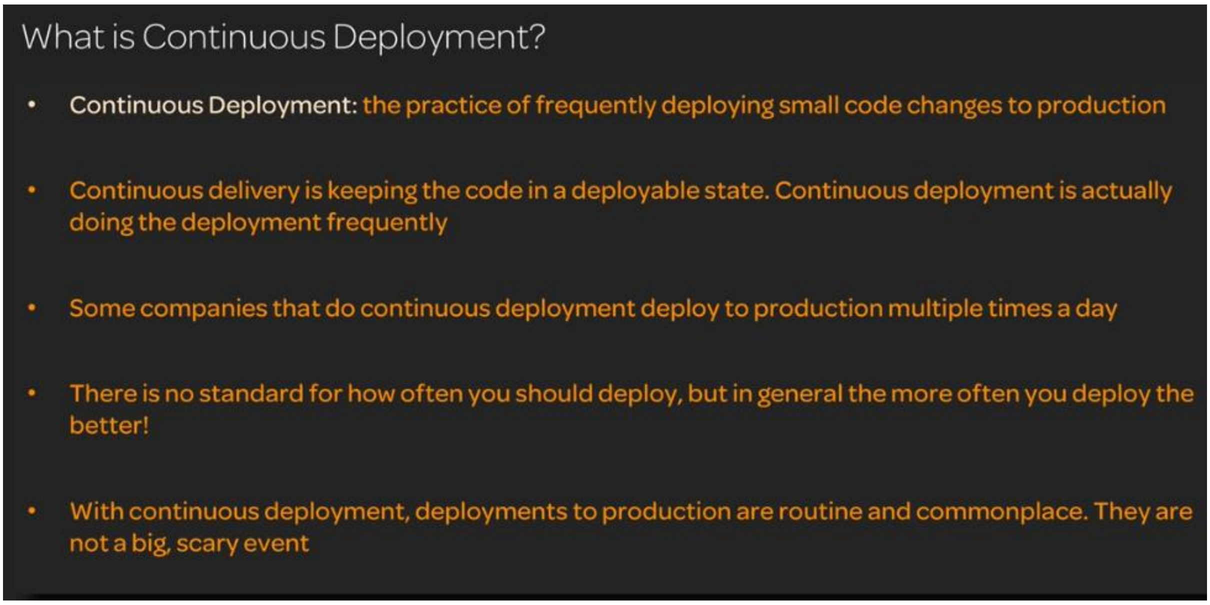
Continuous Delivery



What is Continuous Delivery?

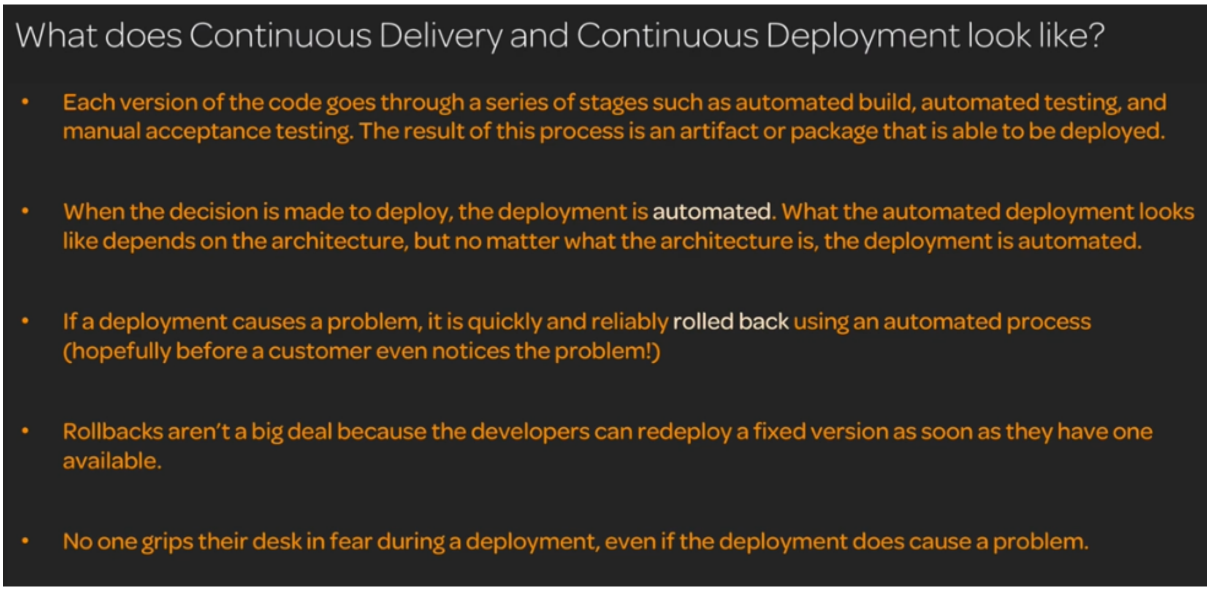
- **Continuous Delivery (CD):** the practice of continuously maintaining code in a deployable state
- Regardless of whether or not the decision is made to deploy, the code is always in a state that is able to be deployed.
- Some use the terms continuous delivery and continuous deployment interchangeably, or simply use the abbreviation CD

Continuous Deployment



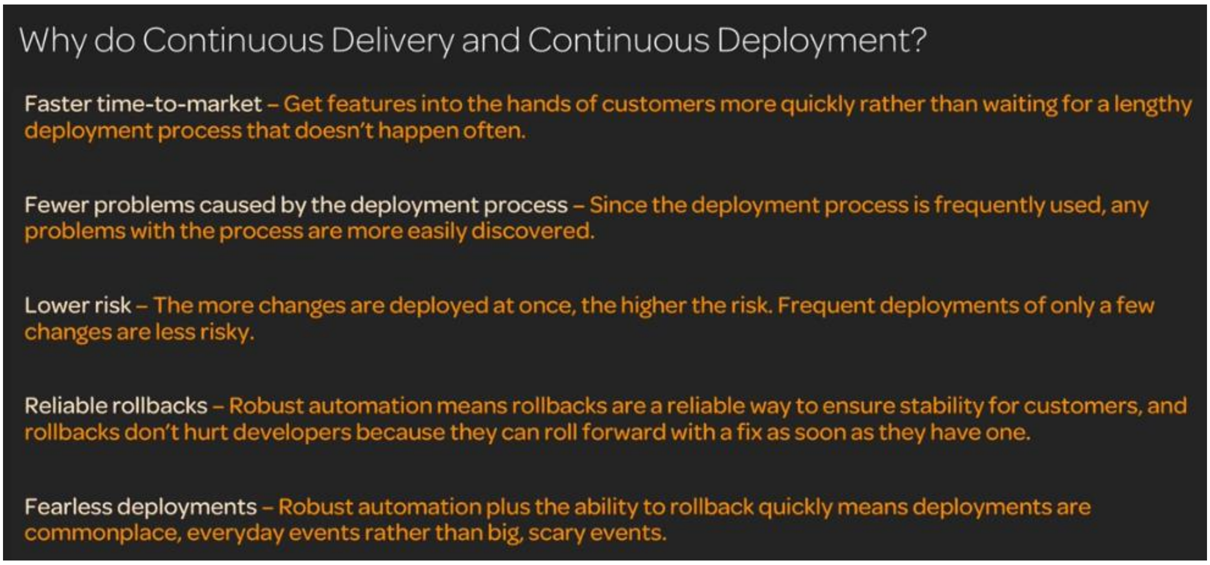
What is Continuous Deployment?

- **Continuous Deployment:** the practice of frequently deploying small code changes to production
- Continuous delivery is keeping the code in a deployable state. Continuous deployment is actually doing the deployment frequently
- Some companies that do continuous deployment deploy to production multiple times a day
- There is no standard for how often you should deploy, but in general the more often you deploy the better!
- With continuous deployment, deployments to production are routine and commonplace. They are not a big, scary event



What does Continuous Delivery and Continuous Deployment look like?

- Each version of the code goes through a series of stages such as automated build, automated testing, and manual acceptance testing. The result of this process is an artifact or package that is able to be deployed.
- When the decision is made to deploy, the deployment is automated. What the automated deployment looks like depends on the architecture, but no matter what the architecture is, the deployment is automated.
- If a deployment causes a problem, it is quickly and reliably rolled back using an automated process (hopefully before a customer even notices the problem!)
- Rollbacks aren't a big deal because the developers can redeploy a fixed version as soon as they have one available.
- No one grips their desk in fear during a deployment, even if the deployment does cause a problem.



Why do Continuous Delivery and Continuous Deployment?

Faster time-to-market – Get features into the hands of customers more quickly rather than waiting for a lengthy deployment process that doesn't happen often.

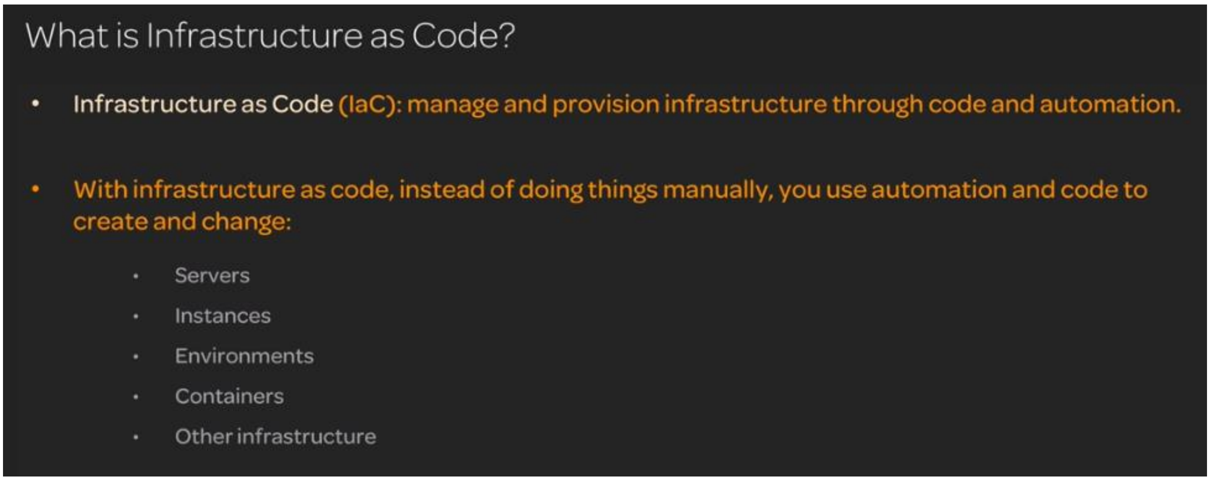
Fewer problems caused by the deployment process – Since the deployment process is frequently used, any problems with the process are more easily discovered.

Lower risk – The more changes are deployed at once, the higher the risk. Frequent deployments of only a few changes are less risky.

Reliable rollbacks – Robust automation means rollbacks are a reliable way to ensure stability for customers, and rollbacks don't hurt developers because they can roll forward with a fix as soon as they have one.

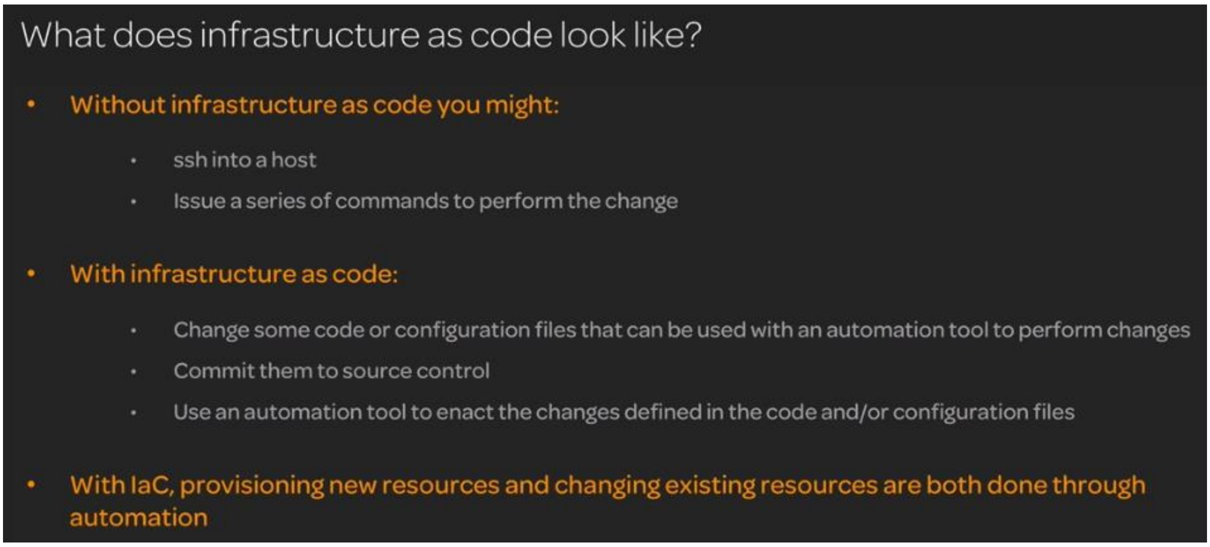
Fearless deployments – Robust automation plus the ability to rollback quickly means deployments are commonplace, everyday events rather than big, scary events.

Infrastructure as Code



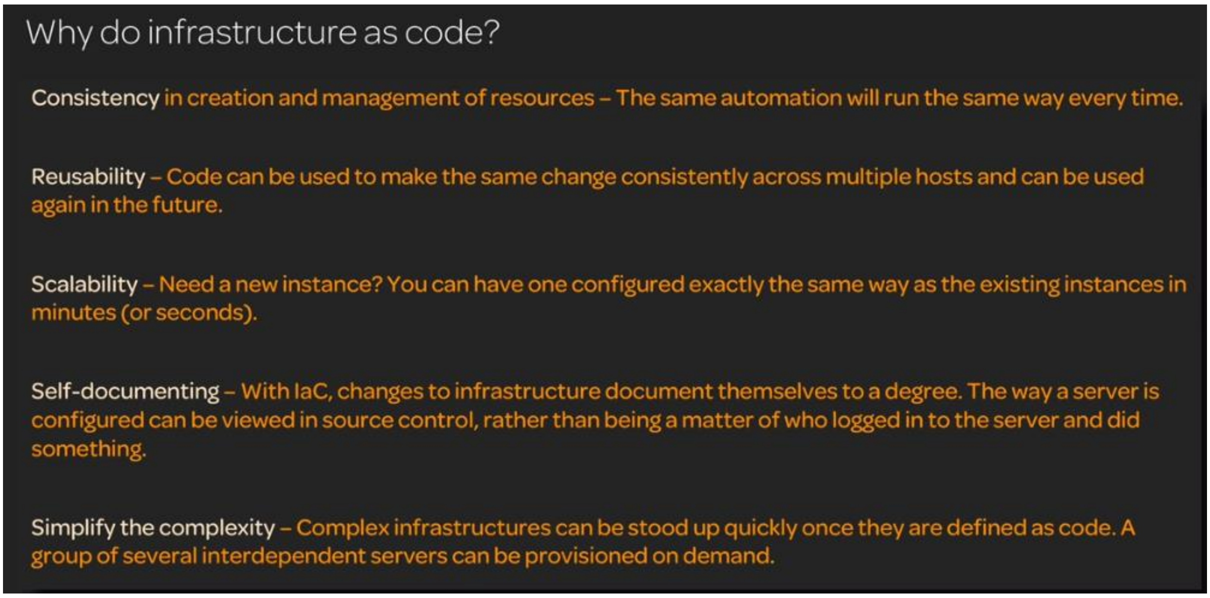
What is Infrastructure as Code?

- Infrastructure as Code (IaC): manage and provision infrastructure through code and automation.
- With infrastructure as code, instead of doing things manually, you use automation and code to create and change:
  - Servers
  - Instances
  - Environments
  - Containers
  - Other infrastructure



What does infrastructure as code look like?

- Without infrastructure as code you might:
  - ssh into a host
  - Issue a series of commands to perform the change
- With infrastructure as code:
  - Change some code or configuration files that can be used with an automation tool to perform changes
  - Commit them to source control
  - Use an automation tool to enact the changes defined in the code and/or configuration files
- With IaC, provisioning new resources and changing existing resources are both done through automation



Why do infrastructure as code?

Consistency in creation and management of resources – The same automation will run the same way every time.

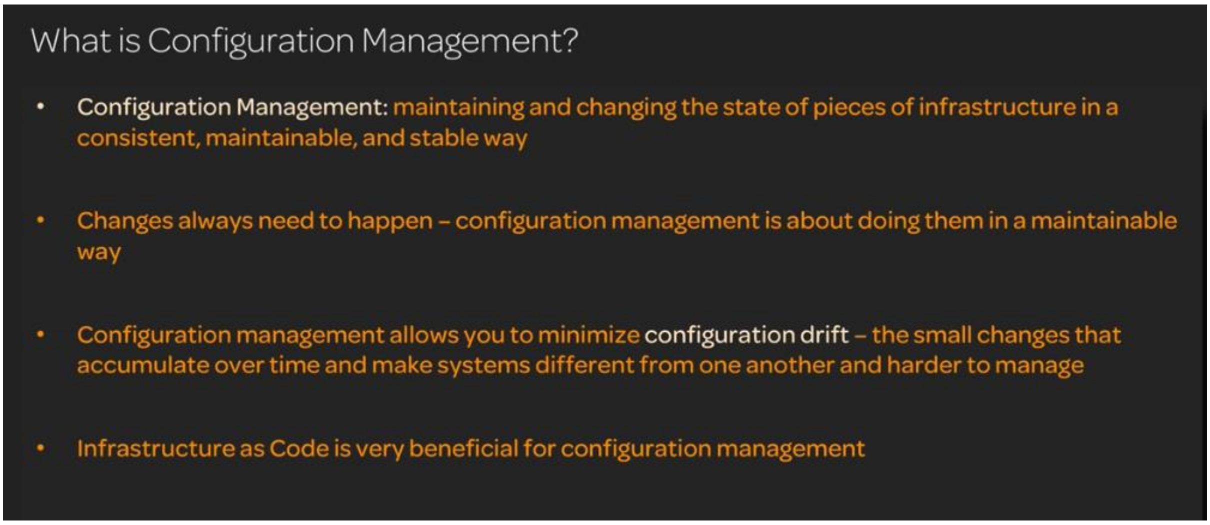
Reusability – Code can be used to make the same change consistently across multiple hosts and can be used again in the future.

Scalability – Need a new instance? You can have one configured exactly the same way as the existing instances in minutes (or seconds).

Self-documenting – With IaC, changes to infrastructure document themselves to a degree. The way a server is configured can be viewed in source control, rather than being a matter of who logged in to the server and did something.

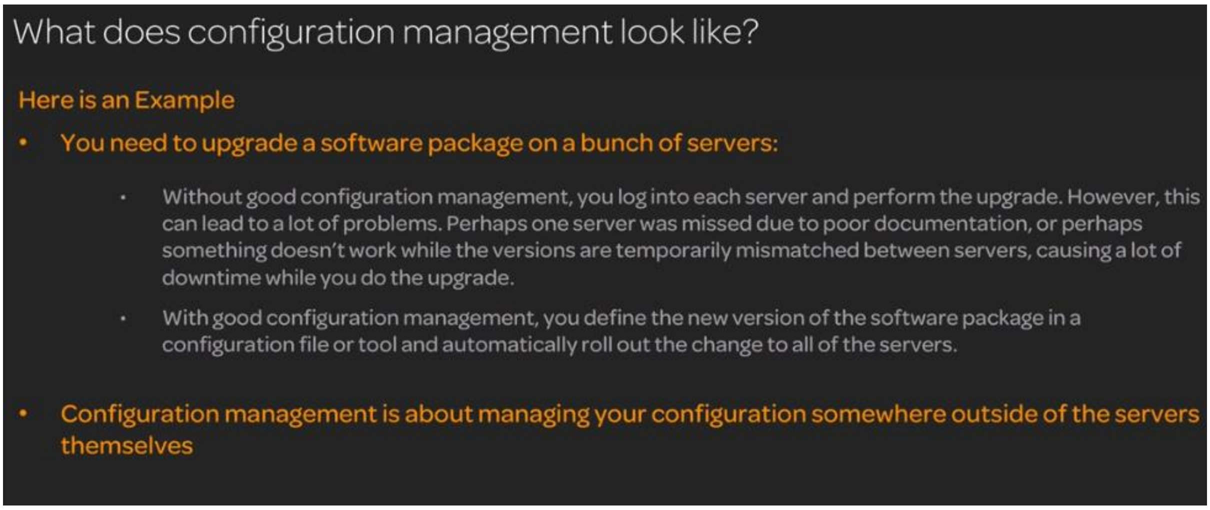
Simplify the complexity – Complex infrastructures can be stood up quickly once they are defined as code. A group of several interdependent servers can be provisioned on demand.

Configuration Management



What is Configuration Management?

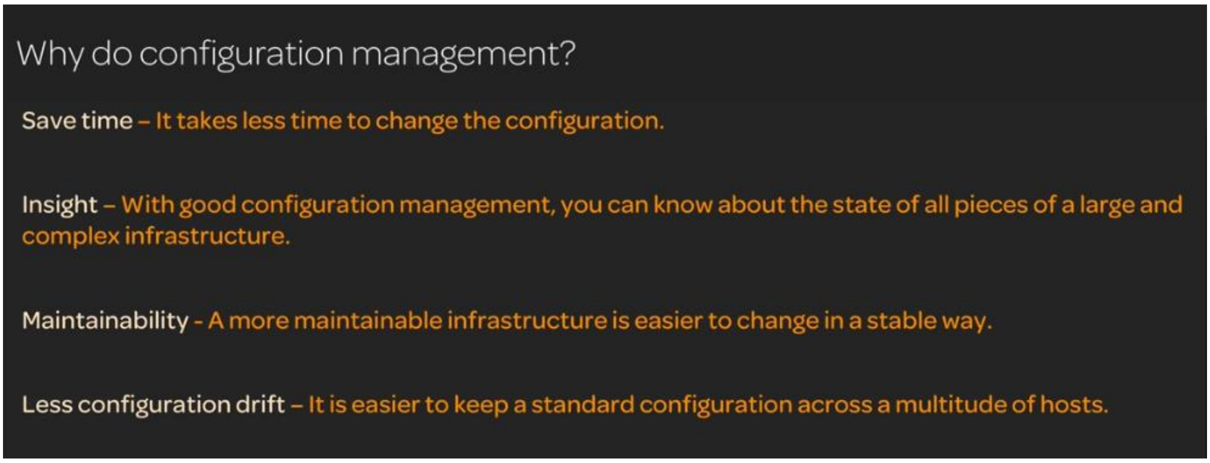
- Configuration Management: maintaining and changing the state of pieces of infrastructure in a consistent, maintainable, and stable way
- Changes always need to happen – configuration management is about doing them in a maintainable way
- Configuration management allows you to minimize configuration drift – the small changes that accumulate over time and make systems different from one another and harder to manage
- Infrastructure as Code is very beneficial for configuration management



What does configuration management look like?

Here is an Example

- You need to upgrade a software package on a bunch of servers:
  - Without good configuration management, you log into each server and perform the upgrade. However, this can lead to a lot of problems. Perhaps one server was missed due to poor documentation, or perhaps something doesn't work while the versions are temporarily mismatched between servers, causing a lot of downtime while you do the upgrade.
  - With good configuration management, you define the new version of the software package in a configuration file or tool and automatically roll out the change to all of the servers.
- Configuration management is about managing your configuration somewhere outside of the servers themselves



Why do configuration management?

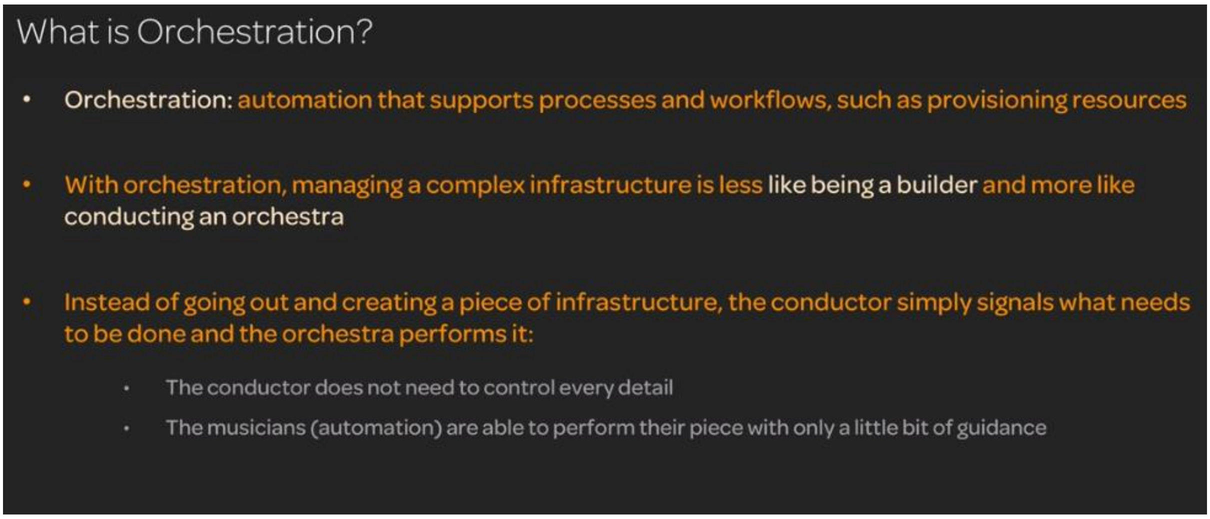
Save time – It takes less time to change the configuration.

Insight – With good configuration management, you can know about the state of all pieces of a large and complex infrastructure.

Maintainability - A more maintainable infrastructure is easier to change in a stable way.

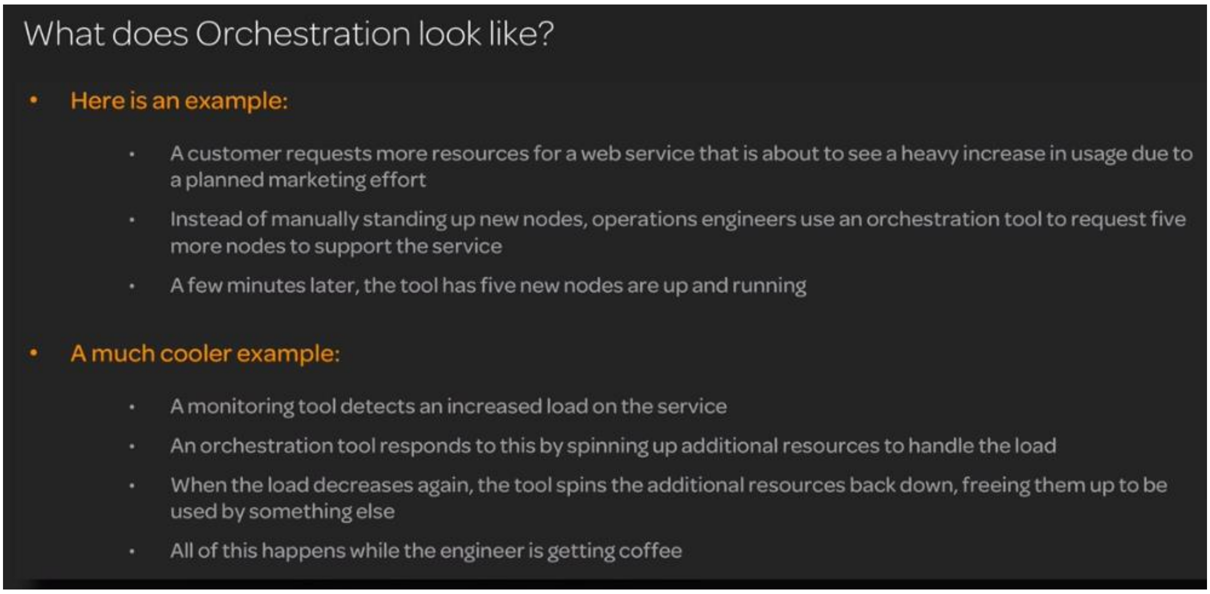
Less configuration drift – It is easier to keep a standard configuration across a multitude of hosts.

Orchestration



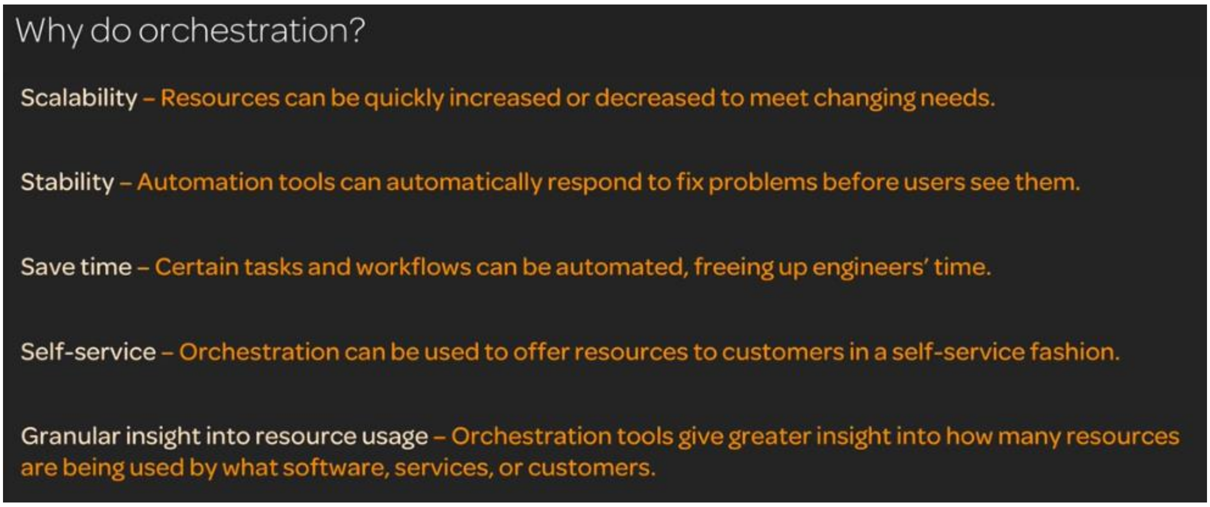
What is Orchestration?

- Orchestration: automation that supports processes and workflows, such as provisioning resources
- With orchestration, managing a complex infrastructure is less like being a builder and more like conducting an orchestra
- Instead of going out and creating a piece of infrastructure, the conductor simply signals what needs to be done and the orchestra performs it:
  - The conductor does not need to control every detail
  - The musicians (automation) are able to perform their piece with only a little bit of guidance



What does Orchestration look like?

- Here is an example:
  - A customer requests more resources for a web service that is about to see a heavy increase in usage due to a planned marketing effort
  - Instead of manually standing up new nodes, operations engineers use an orchestration tool to request five more nodes to support the service
  - A few minutes later, the tool has five new nodes are up and running
- A much cooler example:
  - A monitoring tool detects an increased load on the service
  - An orchestration tool responds to this by spinning up additional resources to handle the load
  - When the load decreases again, the tool spins the additional resources back down, freeing them up to be used by something else
  - All of this happens while the engineer is getting coffee



Why do orchestration?

Scalability – Resources can be quickly increased or decreased to meet changing needs.

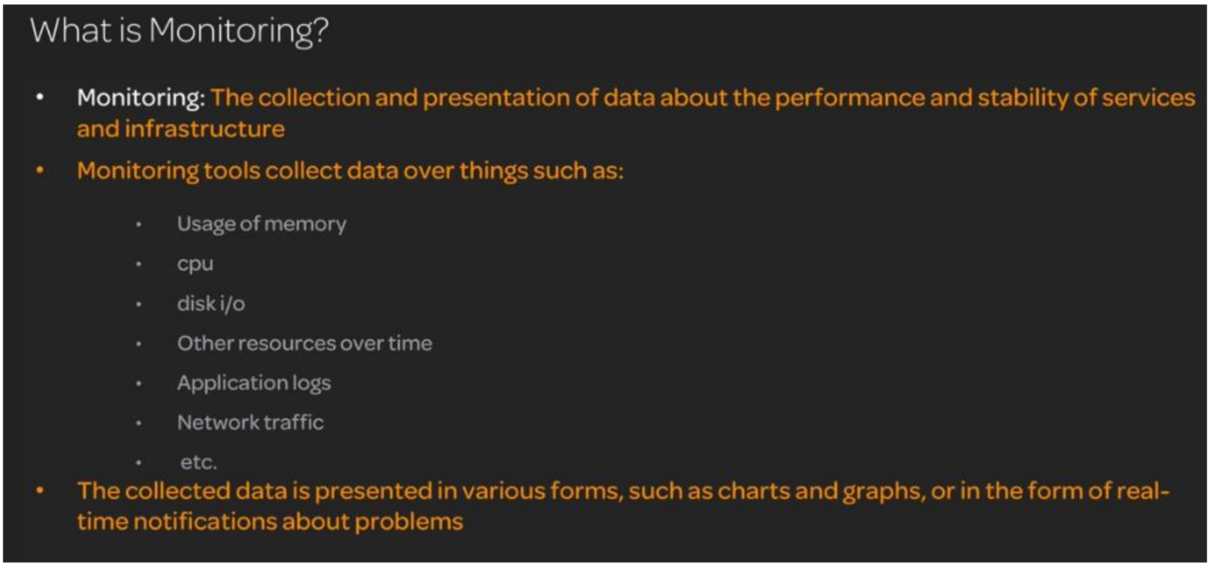
Stability – Automation tools can automatically respond to fix problems before users see them.

Save time – Certain tasks and workflows can be automated, freeing up engineers' time.

Self-service – Orchestration can be used to offer resources to customers in a self-service fashion.

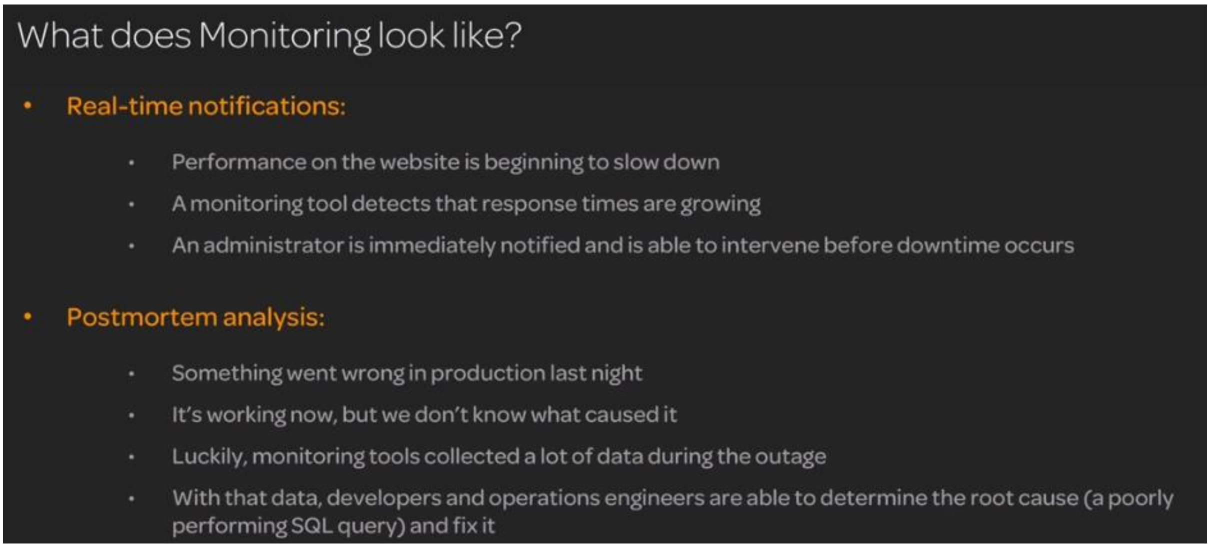
Granular insight into resource usage – Orchestration tools give greater insight into how many resources are being used by what software, services, or customers.

Monitoring



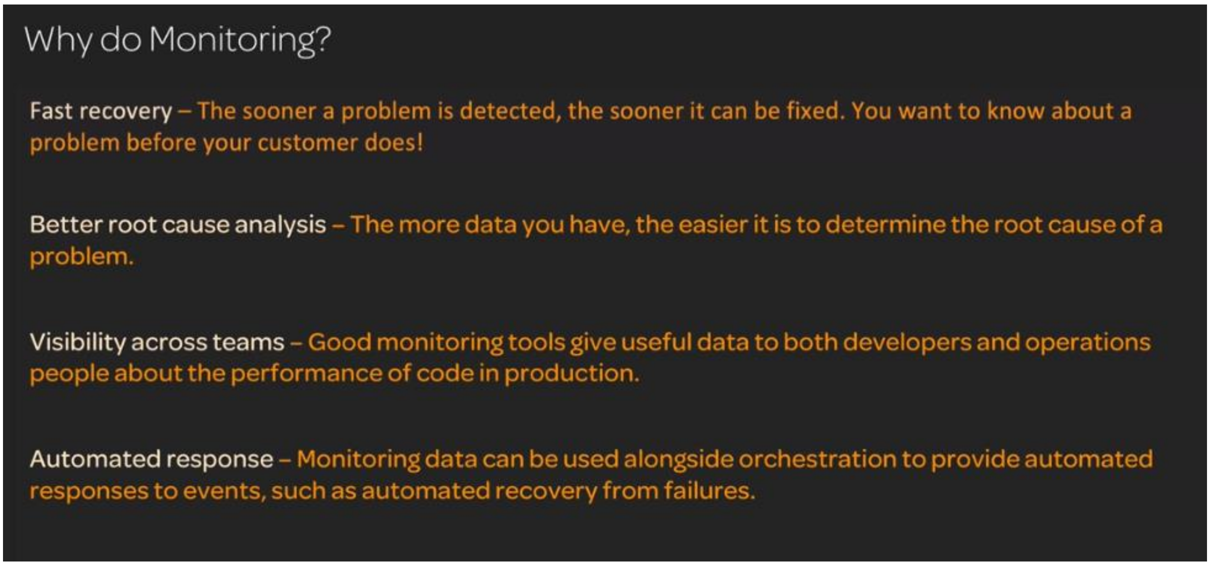
What is Monitoring?

- Monitoring: The collection and presentation of data about the performance and stability of services and infrastructure
- Monitoring tools collect data over things such as:
  - Usage of memory
  - cpu
  - disk i/o
  - Other resources over time
  - Application logs
  - Network traffic
  - etc.
- The collected data is presented in various forms, such as charts and graphs, or in the form of real-time notifications about problems



What does Monitoring look like?

- Real-time notifications:
  - Performance on the website is beginning to slow down
  - A monitoring tool detects that response times are growing
  - An administrator is immediately notified and is able to intervene before downtime occurs
- Postmortem analysis:
  - Something went wrong in production last night
  - It's working now, but we don't know what caused it
  - Luckily, monitoring tools collected a lot of data during the outage
  - With that data, developers and operations engineers are able to determine the root cause (a poorly performing SQL query) and fix it



Why do Monitoring?

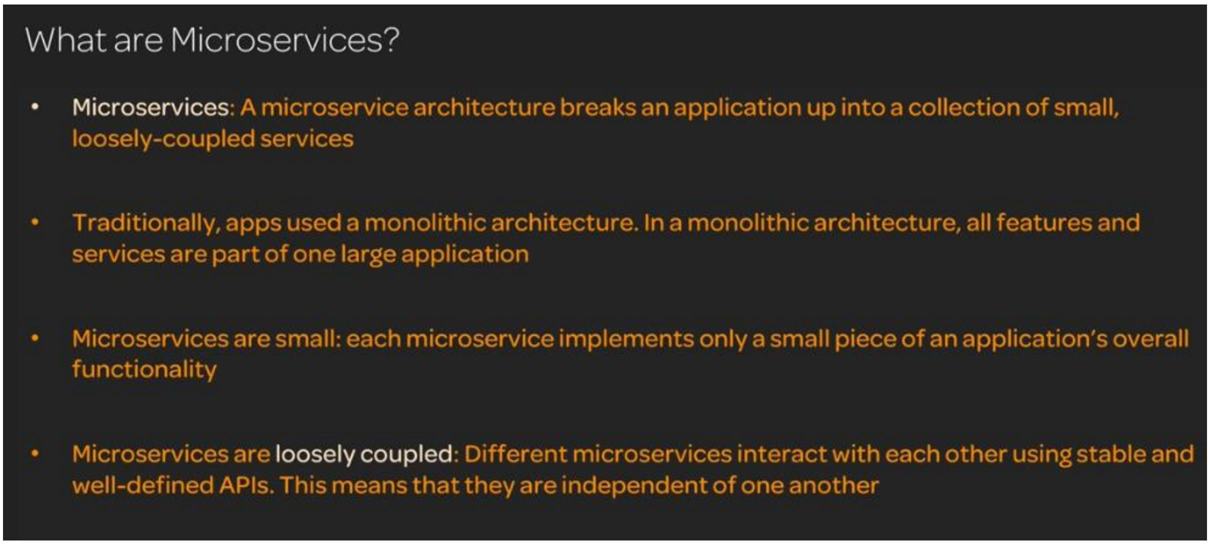
Fast recovery – The sooner a problem is detected, the sooner it can be fixed. You want to know about a problem before your customer does!

Better root cause analysis – The more data you have, the easier it is to determine the root cause of a problem.

Visibility across teams – Good monitoring tools give useful data to both developers and operations people about the performance of code in production.

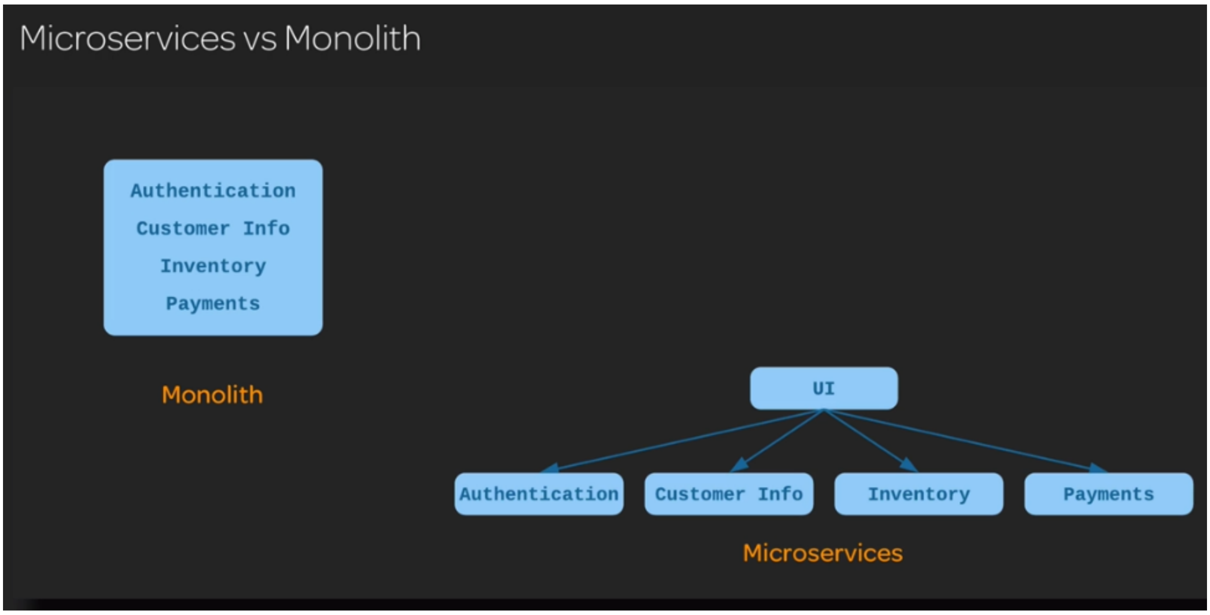
Automated response – Monitoring data can be used alongside orchestration to provide automated responses to events, such as automated recovery from failures.

Micro Services



What are Microservices?

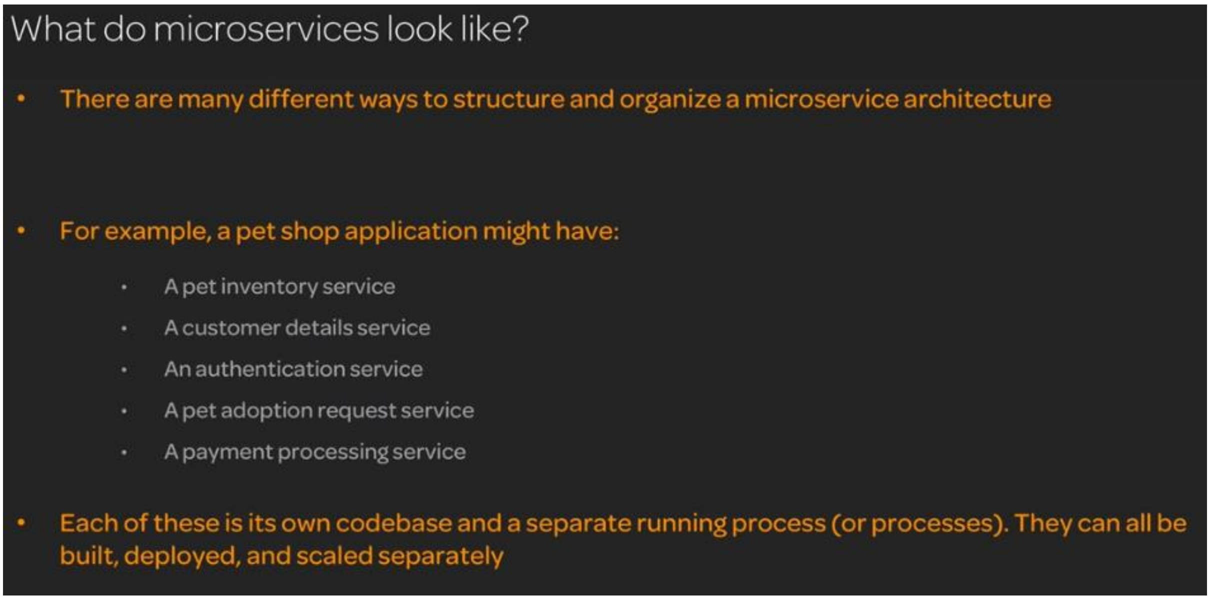
- Microservices: A microservice architecture breaks an application up into a collection of small, loosely-coupled services
- Traditionally, apps used a monolithic architecture. In a monolithic architecture, all features and services are part of one large application
- Microservices are small: each microservice implements only a small piece of an application's overall functionality
- Microservices are loosely coupled: Different microservices interact with each other using stable and well-defined APIs. This means that they are independent of one another



Microservices vs Monolith

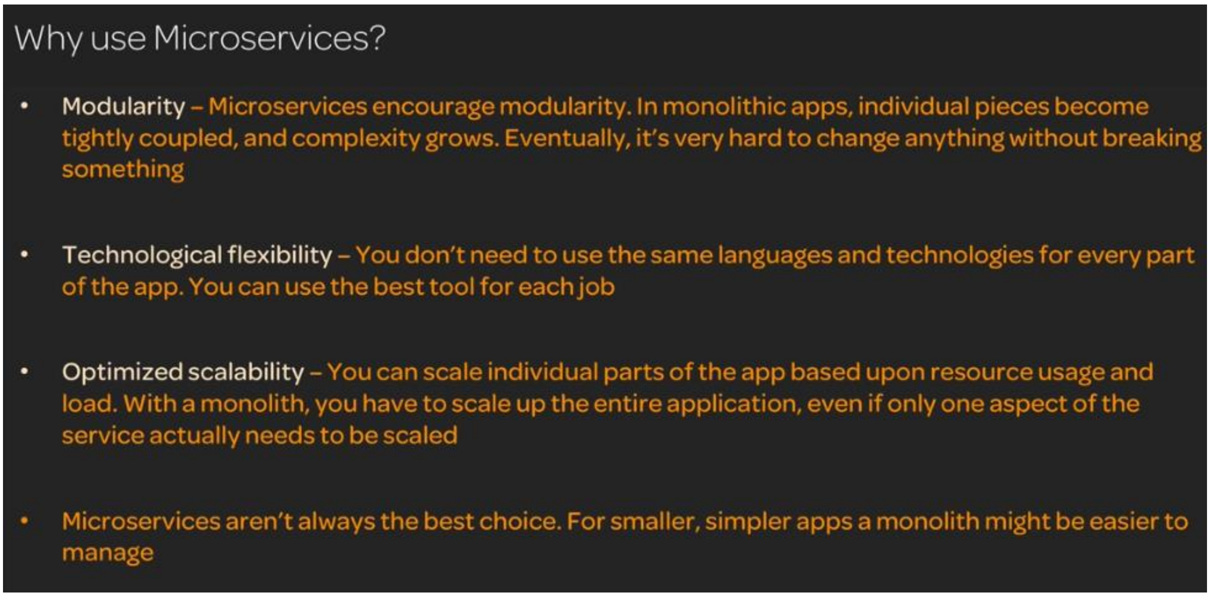
Monolith

Microservices



What do microservices look like?

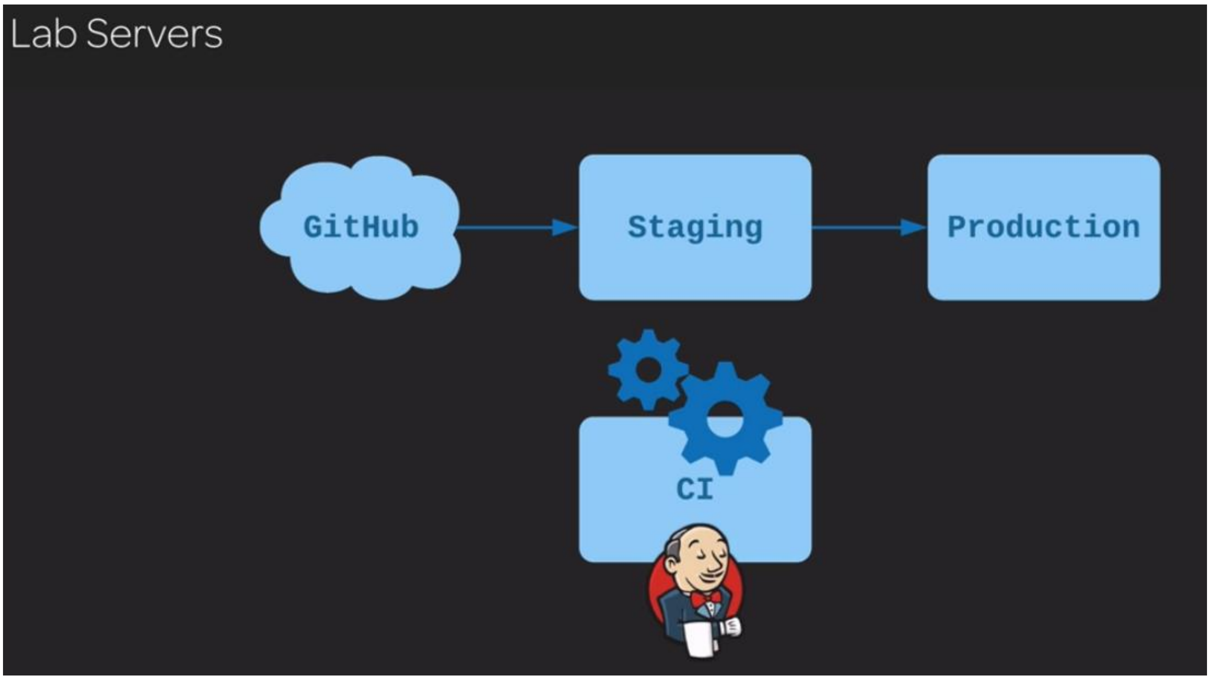
- There are many different ways to structure and organize a microservice architecture
- For example, a pet shop application might have:
  - A pet inventory service
  - A customer details service
  - An authentication service
  - A pet adoption request service
  - A payment processing service
- Each of these is its own codebase and a separate running process (or processes). They can all be built, deployed, and scaled separately



Why use Microservices?

- Modularity – Microservices encourage modularity. In monolithic apps, individual pieces become tightly coupled, and complexity grows. Eventually, it's very hard to change anything without breaking something
- Technological flexibility – You don't need to use the same languages and technologies for every part of the app. You can use the best tool for each job
- Optimized scalability – You can scale individual parts of the app based upon resource usage and load. With a monolith, you have to scale up the entire application, even if only one aspect of the service actually needs to be scaled
- Microservices aren't always the best choice. For smaller, simpler apps a monolith might be easier to manage

Lab



Lab Servers

GitHub

Staging

Production

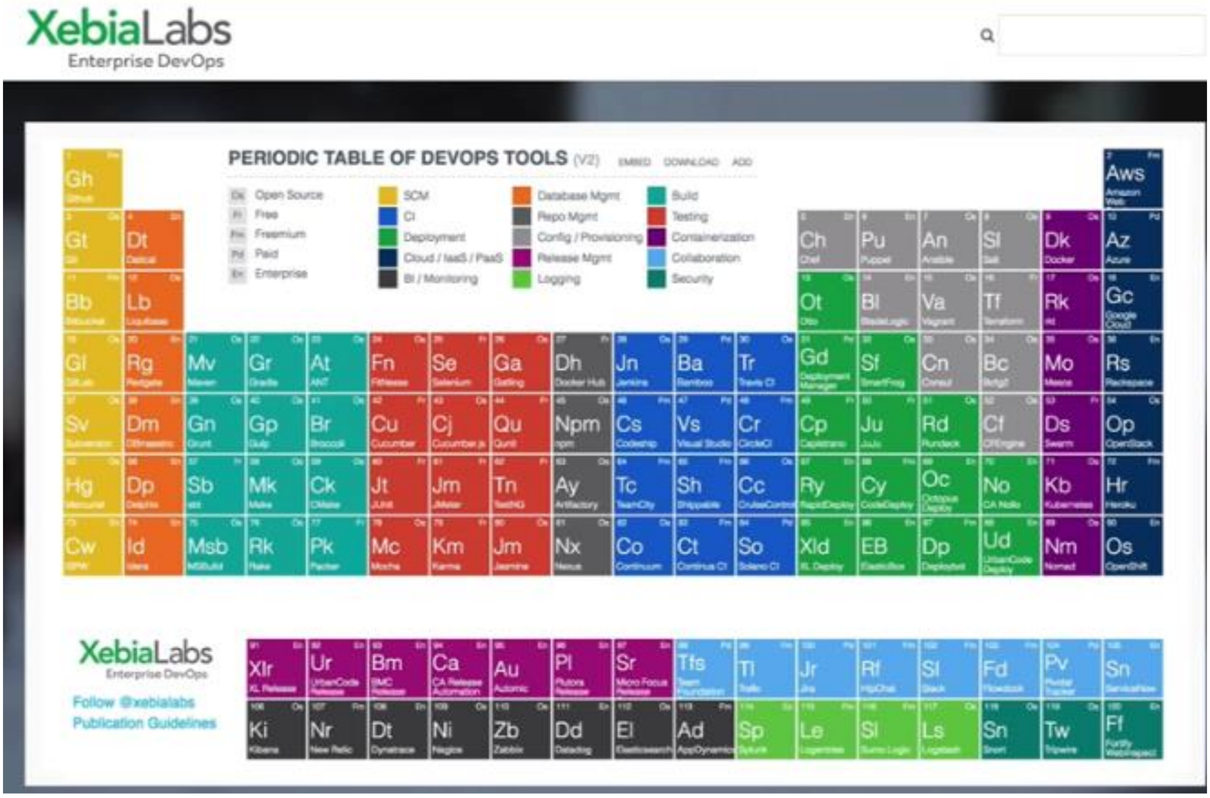
CI

https://github.com/linuxacademy/devops-essentials-sample-app

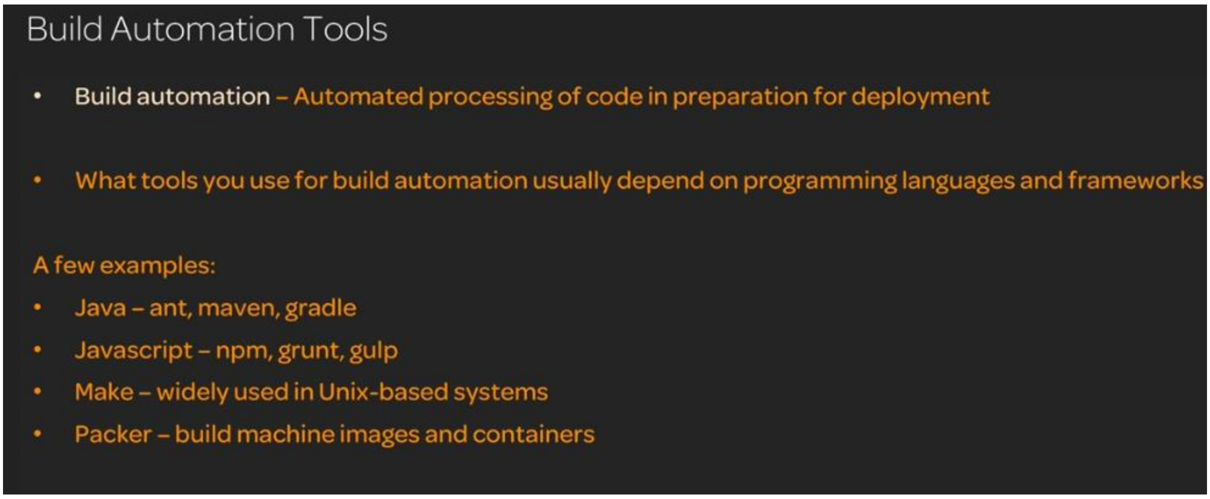
https://github.com/whboyd/devops-essentials-sample-app

#Chapter 4

DevOps Tools



Build Automation Tools



Build Automation Tools

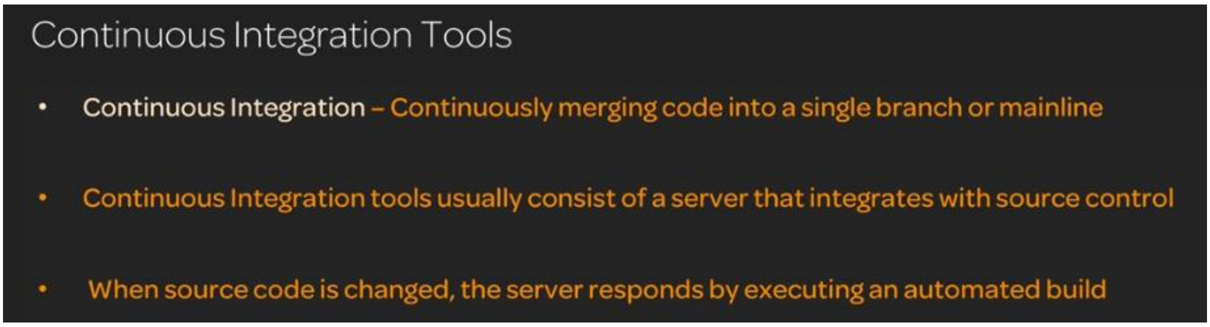
- Build automation – Automated processing of code in preparation for deployment

- What tools you use for build automation usually depend on programming languages and frameworks

A few examples:

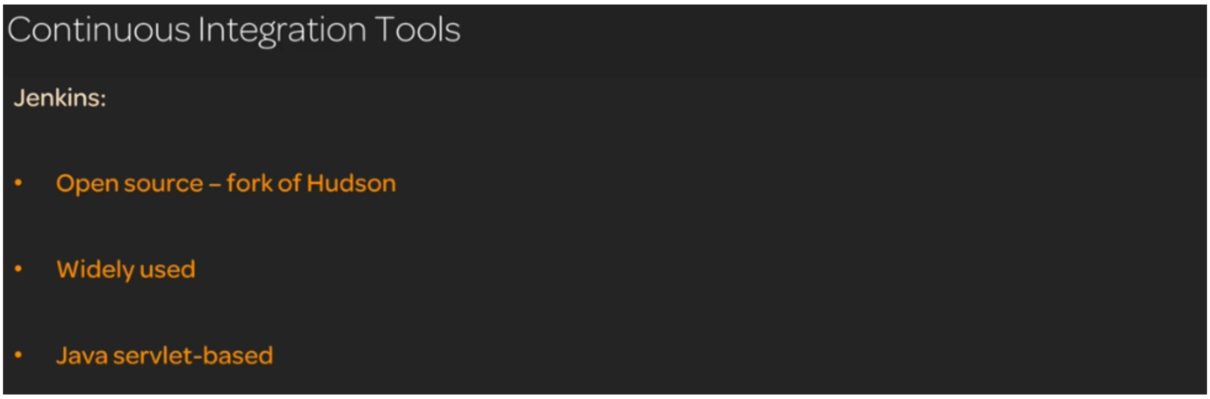
- Java – ant, maven, gradle
- Javascript – npm, grunt, gulp
- Make – widely used in Unix-based systems
- Packer – build machine images and containers

Continuous integration Tools



Continuous Integration Tools

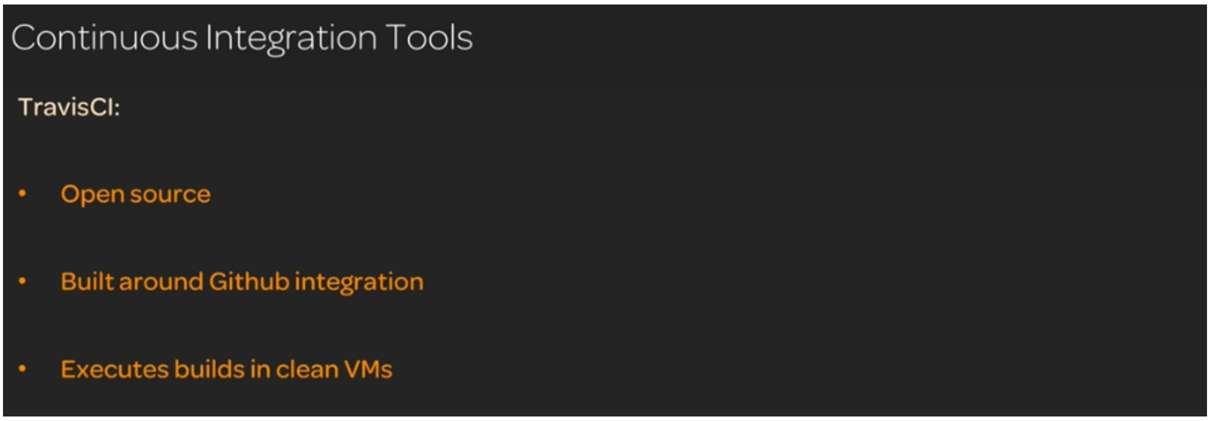
- Continuous Integration – Continuously merging code into a single branch or mainline
- Continuous Integration tools usually consist of a server that integrates with source control
- When source code is changed, the server responds by executing an automated build



Continuous Integration Tools

Jenkins:

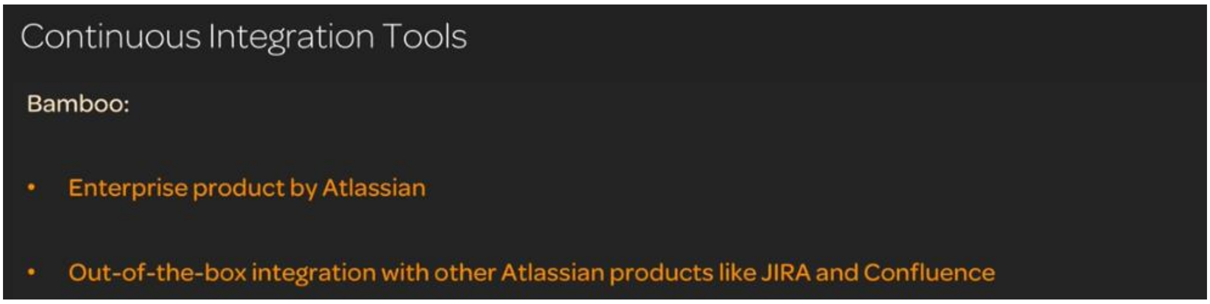
- Open source – fork of Hudson
- Widely used
- Java servlet-based



Continuous Integration Tools

TravisCI:

- Open source
- Built around Github integration
- Executes builds in clean VMs

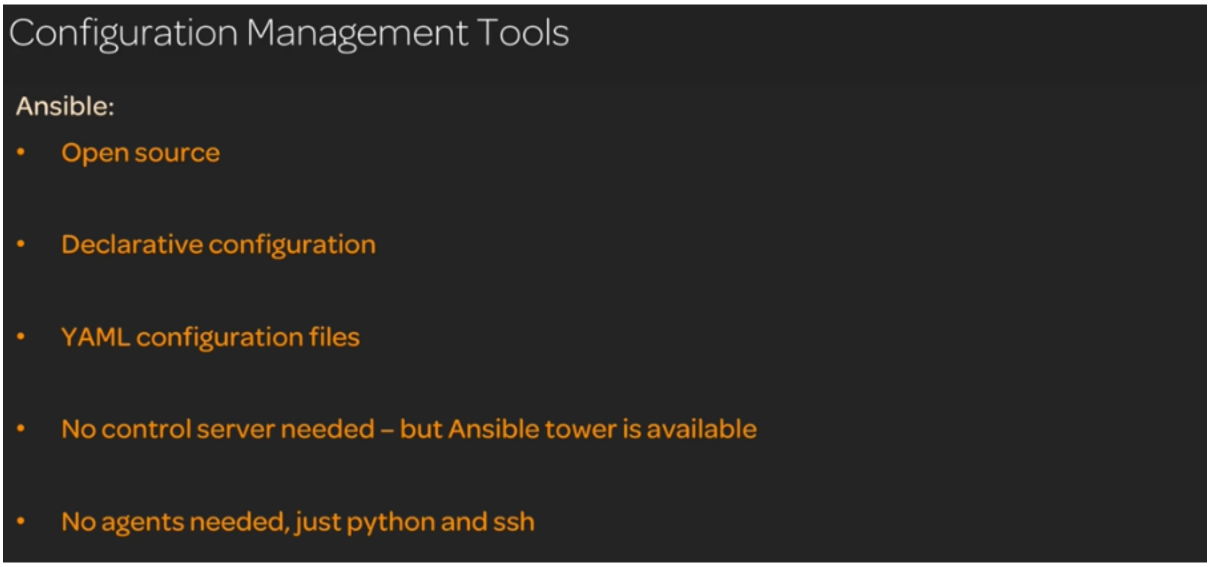


Continuous Integration Tools

Bamboo:

- Enterprise product by Atlassian
- Out-of-the-box integration with other Atlassian products like JIRA and Confluence

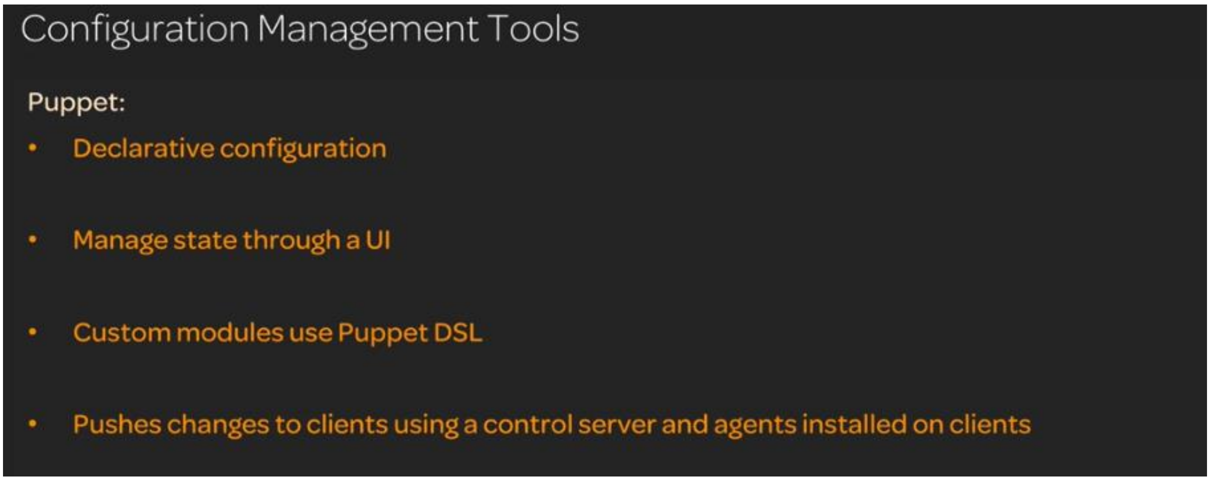
Configuration Management Tools



Configuration Management Tools

Ansible:

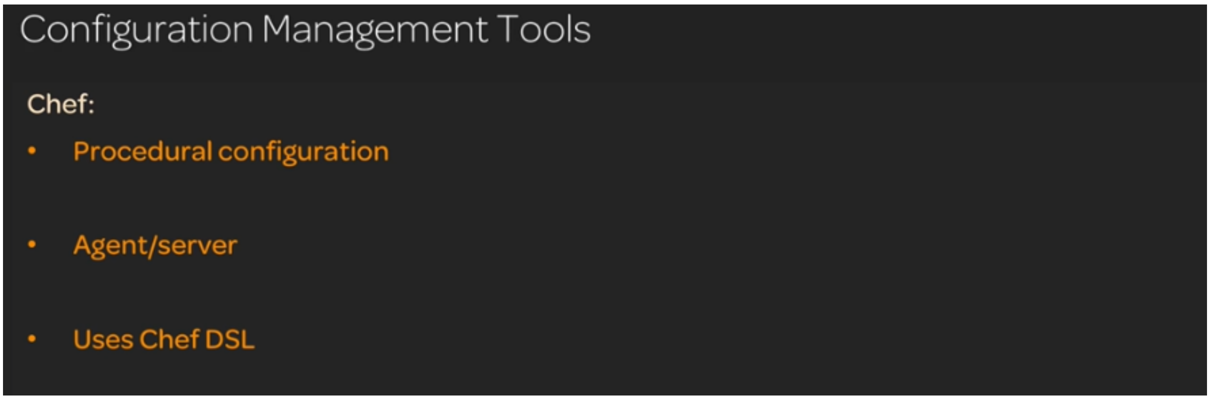
- Open source
- Declarative configuration
- YAML configuration files
- No control server needed – but Ansible tower is available
- No agents needed, just python and ssh



Configuration Management Tools

Puppet:

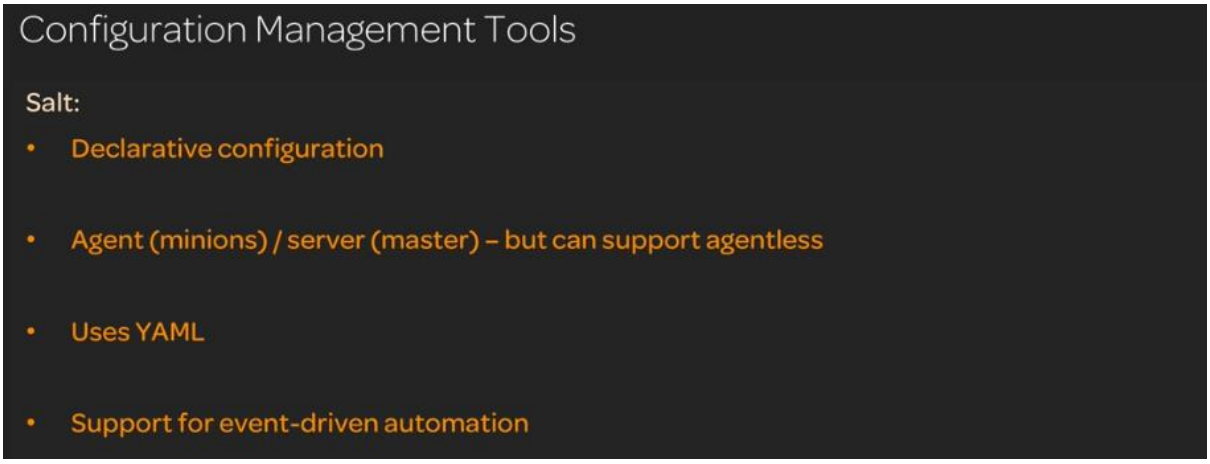
- Declarative configuration
- Manage state through a UI
- Custom modules use Puppet DSL
- Pushes changes to clients using a control server and agents installed on clients



Configuration Management Tools

Chef:

- Procedural configuration
- Agent/server
- Uses Chef DSL

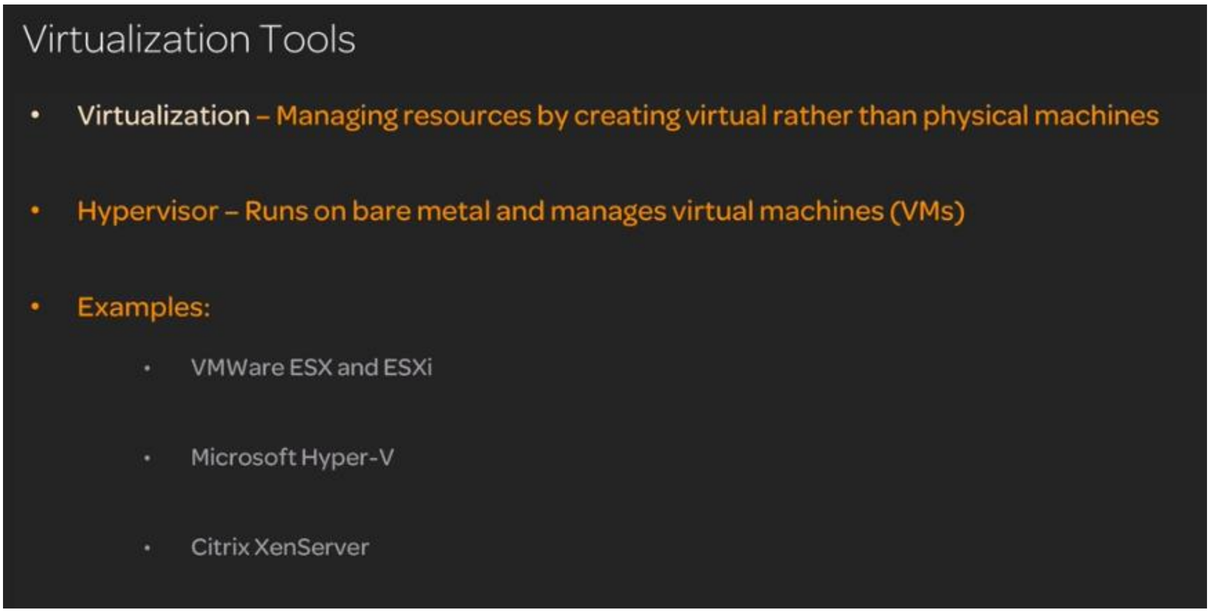


Configuration Management Tools

Salt:

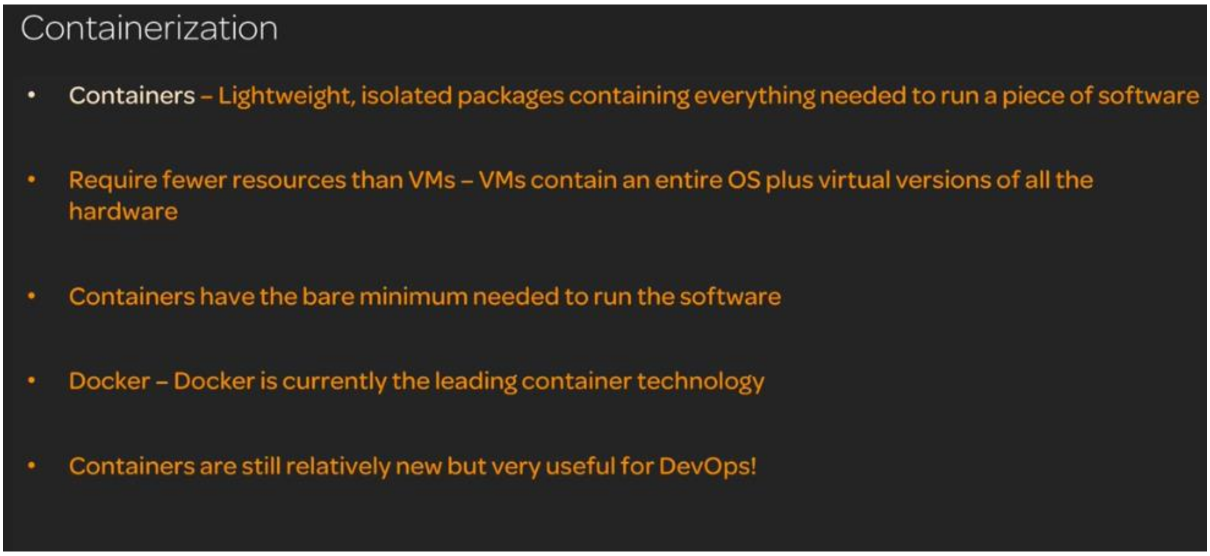
- Declarative configuration
- Agent (minions) / server (master) – but can support agentless
- Uses YAML
- Support for event-driven automation

Virtualization & Containerization Tools



Virtualization Tools

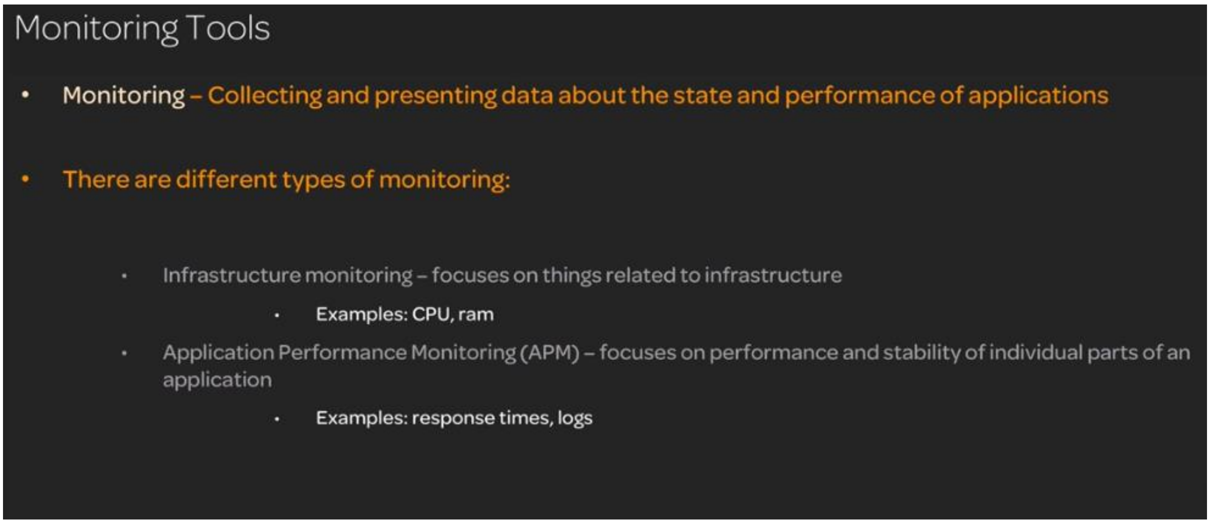
- Virtualization – Managing resources by creating virtual rather than physical machines
- Hypervisor – Runs on bare metal and manages virtual machines (VMs)
- Examples:
  - VMWare ESX and ESXi
  - Microsoft Hyper-V
  - Citrix XenServer



Containerization

- Containers – Lightweight, isolated packages containing everything needed to run a piece of software
- Require fewer resources than VMs – VMs contain an entire OS plus virtual versions of all the hardware
- Containers have the bare minimum needed to run the software
- Docker – Docker is currently the leading container technology
- Containers are still relatively new but very useful for DevOps!

Monitoring Tools

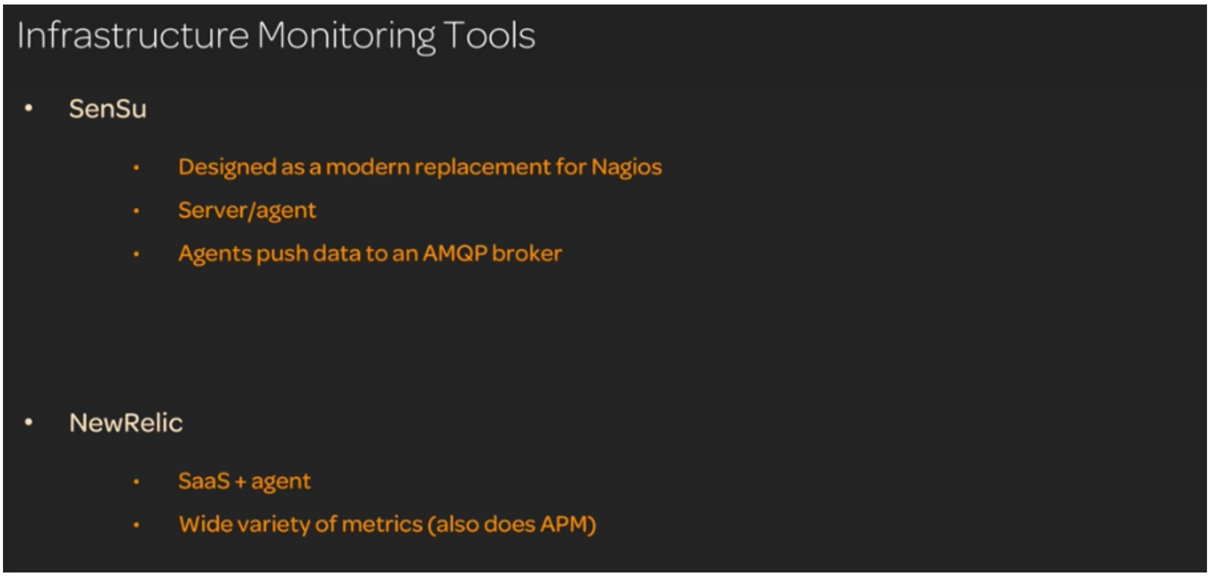


Monitoring Tools

- Monitoring – Collecting and presenting data about the state and performance of applications

- There are different types of monitoring:

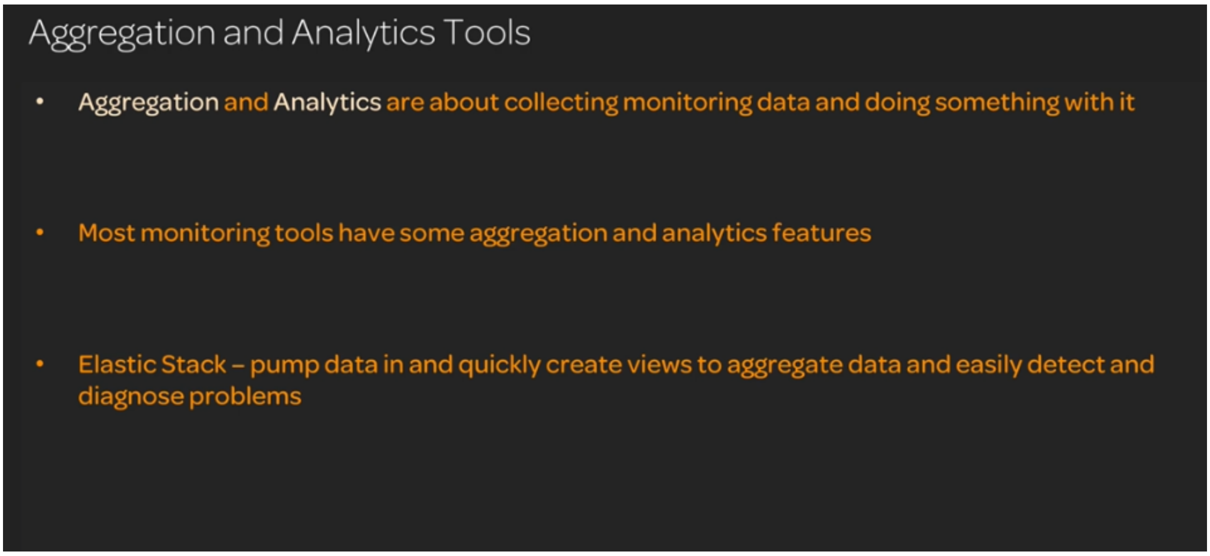
    - Infrastructure monitoring – focuses on things related to infrastructure
        - Examples: CPU, ram
    - Application Performance Monitoring (APM) – focuses on performance and stability of individual parts of an application
        - Examples: response times, logs



Infrastructure Monitoring Tools

- SenSu
    - Designed as a modern replacement for Nagios
    - Server/agent
    - Agents push data to an AMQP broker

- NewRelic
    - SaaS + agent
    - Wide variety of metrics (also does APM)

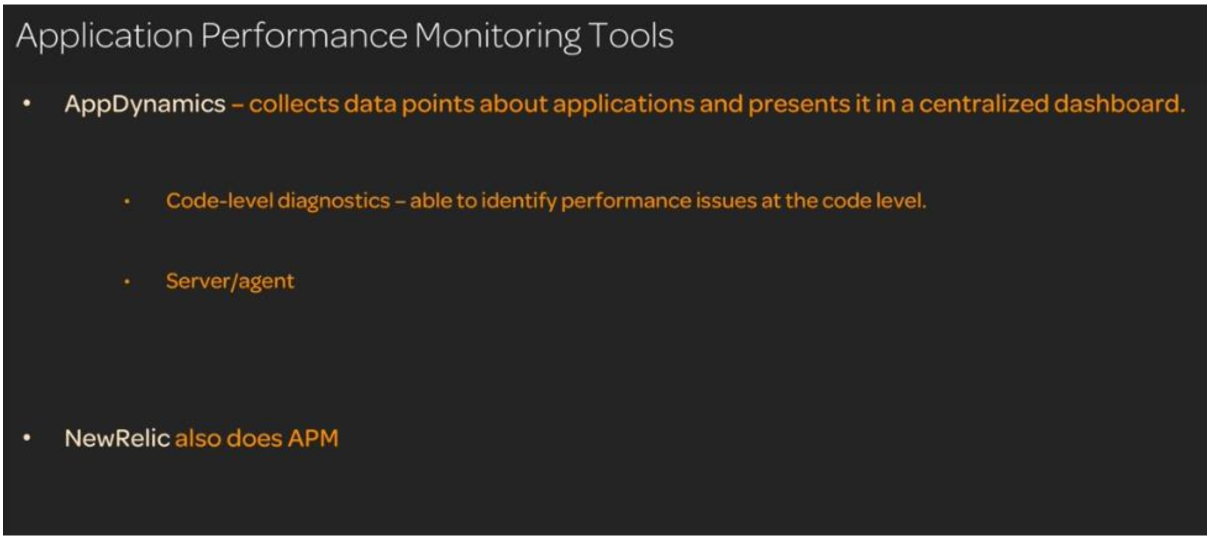


Aggregation and Analytics Tools

- Aggregation and Analytics are about collecting monitoring data and doing something with it

- Most monitoring tools have some aggregation and analytics features

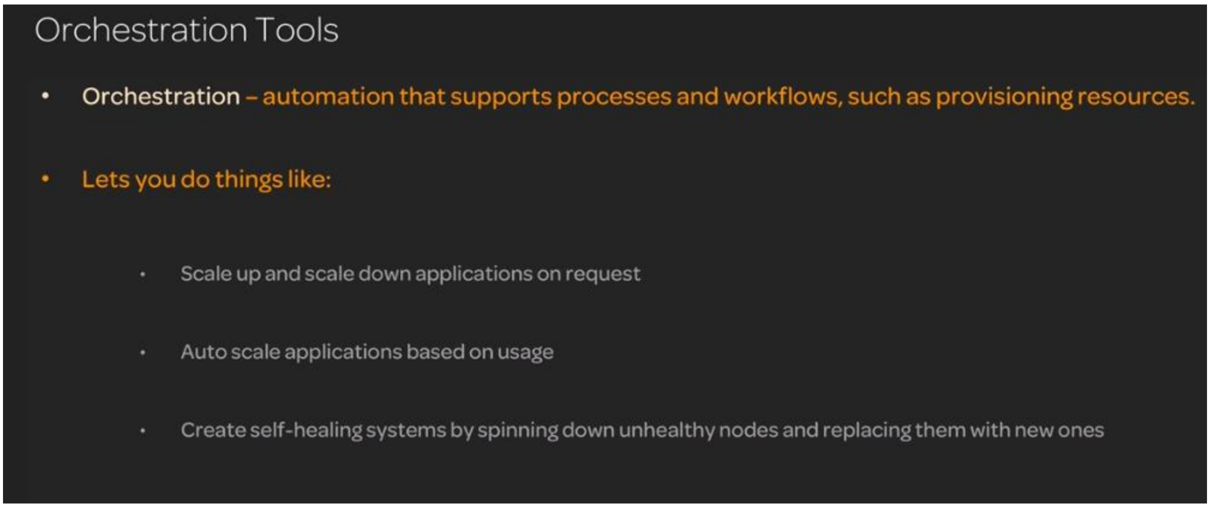
- Elastic Stack – pump data in and quickly create views to aggregate data and easily detect and diagnose problems



Application Performance Monitoring Tools

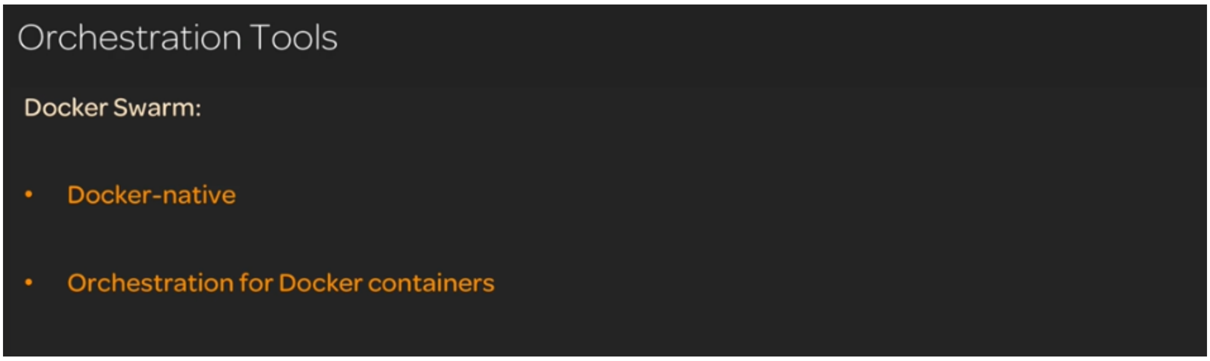
- AppDynamics – collects data points about applications and presents it in a centralized dashboard.
  - Code-level diagnostics – able to identify performance issues at the code level.
  - Server/agent
- NewRelic also does APM

Orchestration Tools



Orchestration Tools

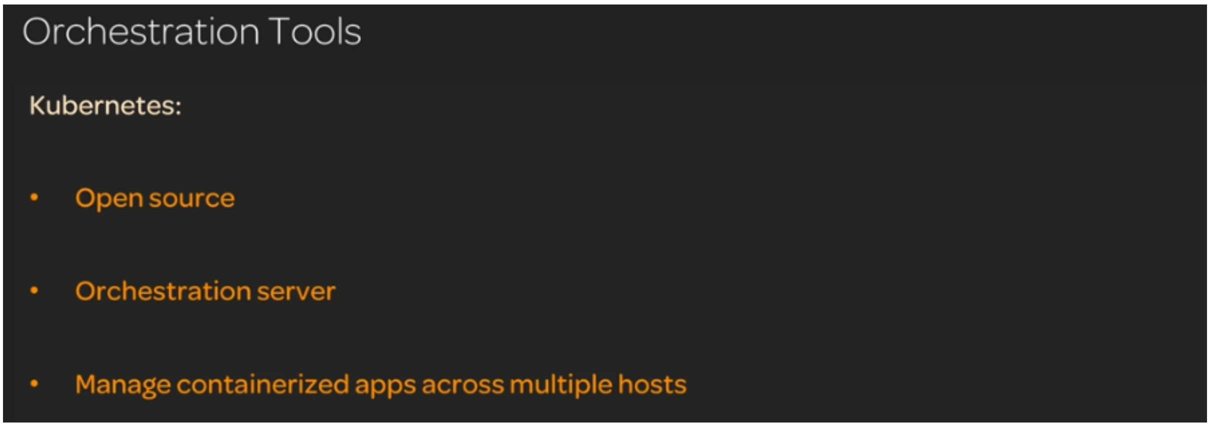
- Orchestration – automation that supports processes and workflows, such as provisioning resources.
- Lets you do things like:
  - Scale up and scale down applications on request
  - Auto scale applications based on usage
  - Create self-healing systems by spinning down unhealthy nodes and replacing them with new ones



Orchestration Tools

Docker Swarm:

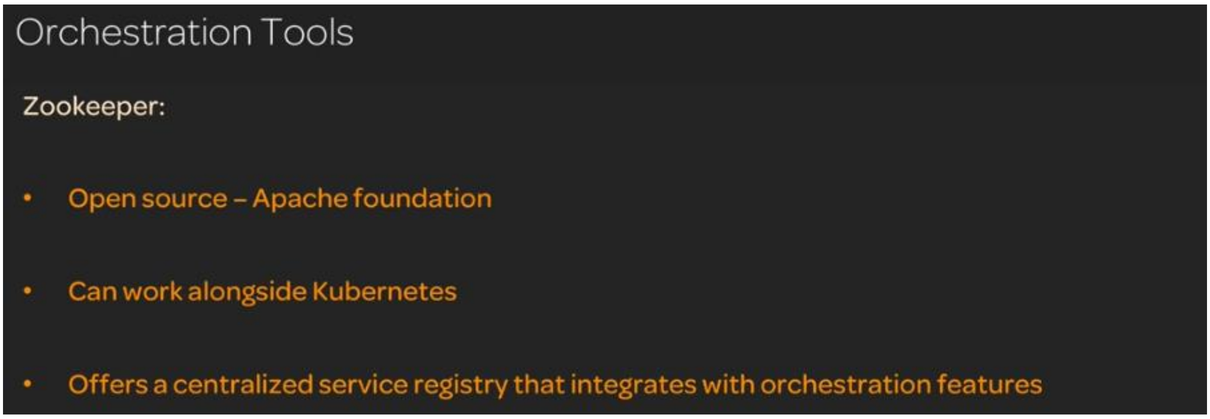
- Docker-native
- Orchestration for Docker containers



Orchestration Tools

Kubernetes:

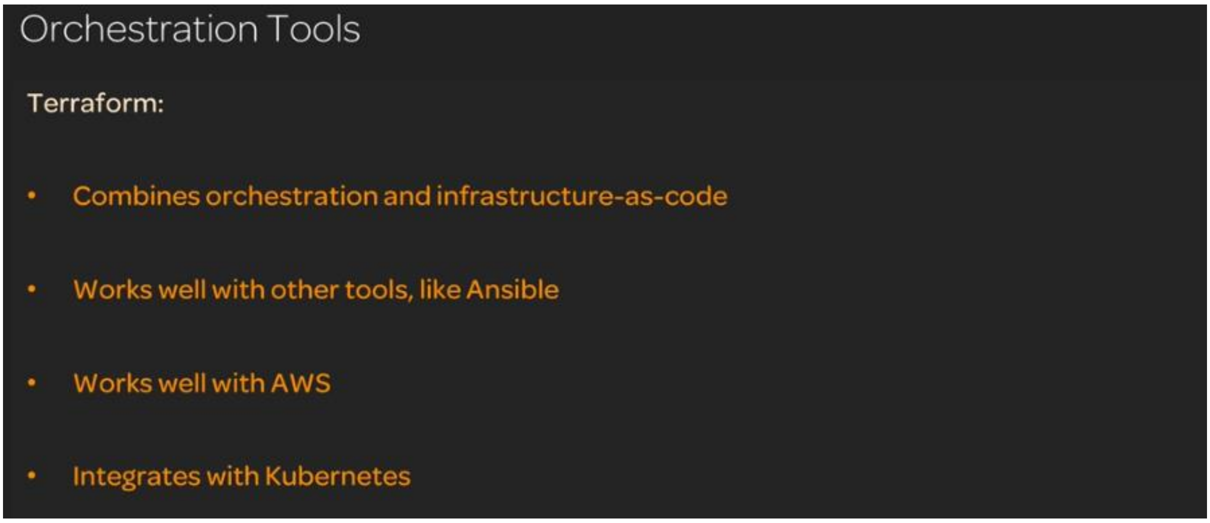
- Open source
- Orchestration server
- Manage containerized apps across multiple hosts



Orchestration Tools

Zookeeper:

- Open source – Apache foundation
- Can work alongside Kubernetes
- Offers a centralized service registry that integrates with orchestration features



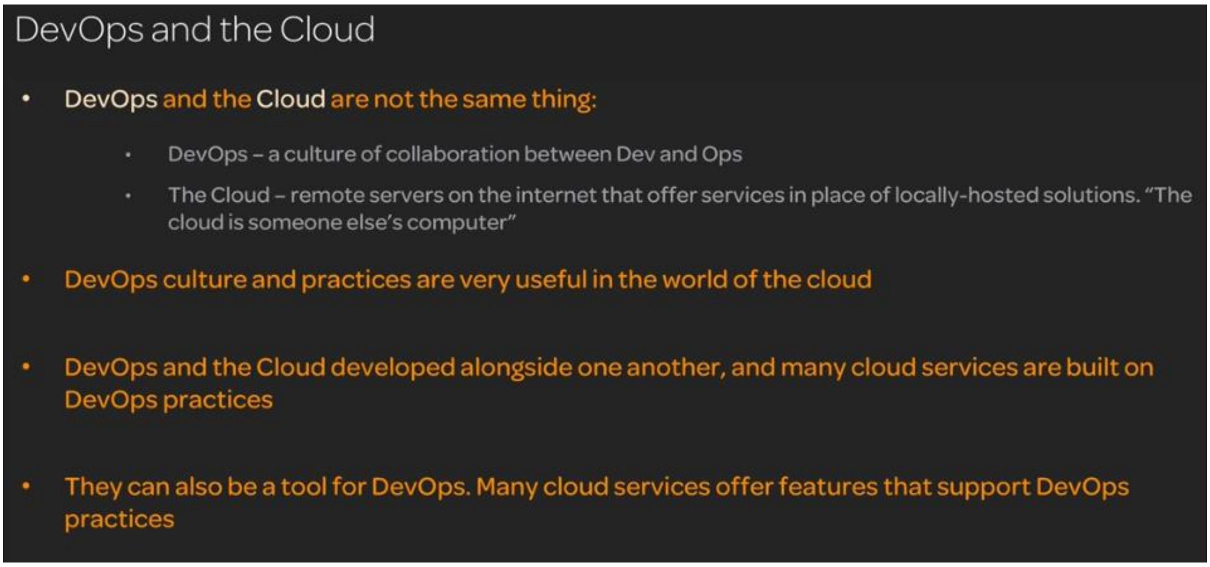
Orchestration Tools

Terraform:

- Combines orchestration and infrastructure-as-code
- Works well with other tools, like Ansible
- Works well with AWS
- Integrates with Kubernetes

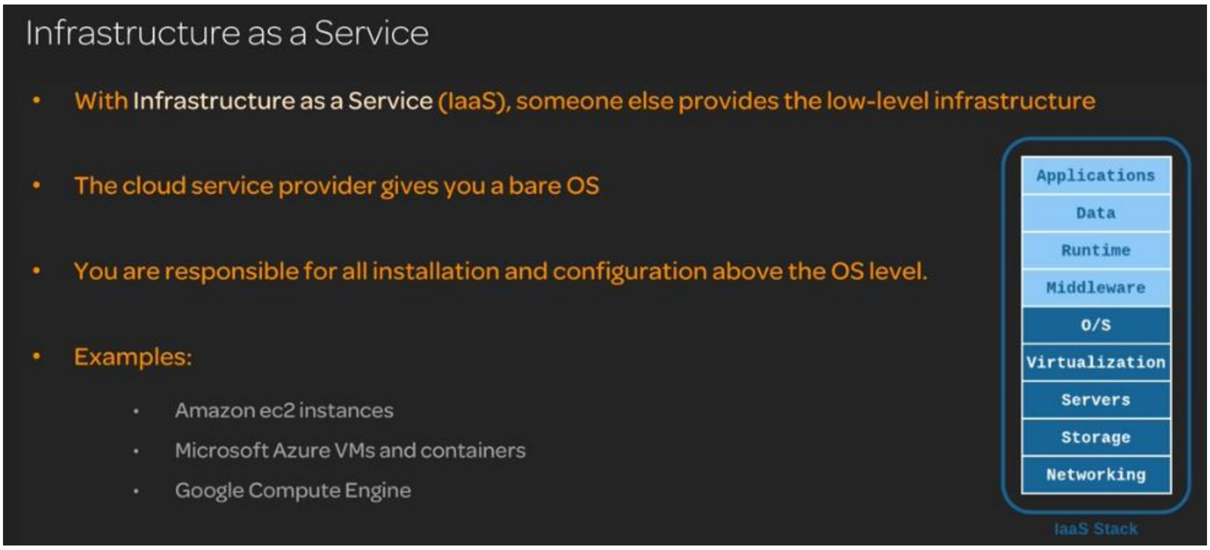
# Chapter 5

DevOps and the Cloud



DevOps and the Cloud

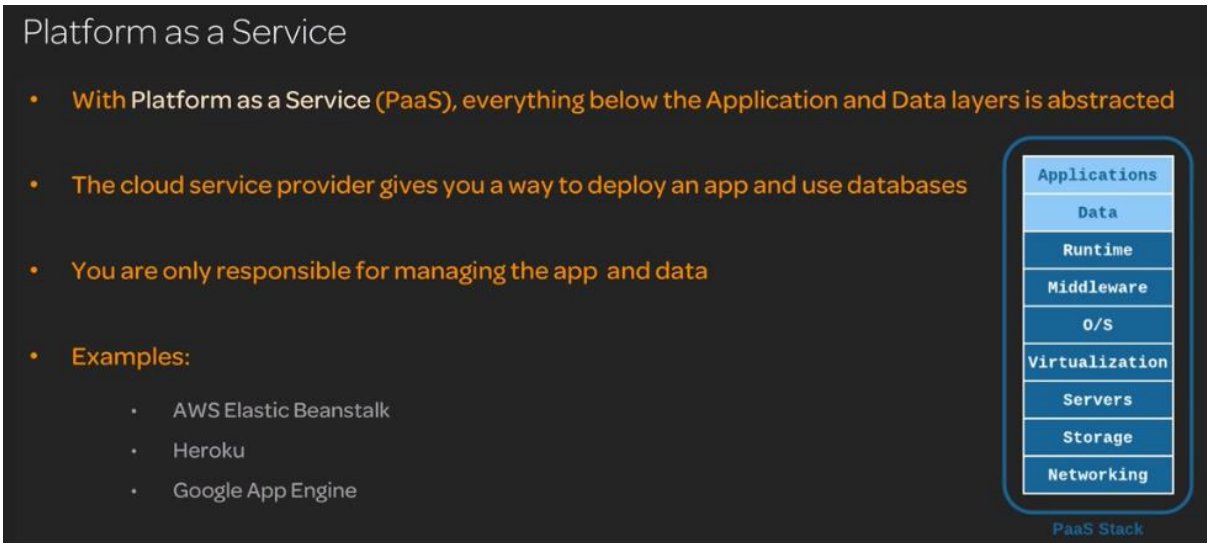
- DevOps and the Cloud are not the same thing:
  - DevOps – a culture of collaboration between Dev and Ops
  - The Cloud – remote servers on the internet that offer services in place of locally-hosted solutions. "The cloud is someone else's computer"
- DevOps culture and practices are very useful in the world of the cloud
- DevOps and the Cloud developed alongside one another, and many cloud services are built on DevOps practices
- They can also be a tool for DevOps. Many cloud services offer features that support DevOps practices



Infrastructure as a Service

- With Infrastructure as a Service (IaaS), someone else provides the low-level infrastructure
- The cloud service provider gives you a bare OS
- You are responsible for all installation and configuration above the OS level.
- Examples:
  - Amazon ec2 instances
  - Microsoft Azure VMs and containers
  - Google Compute Engine

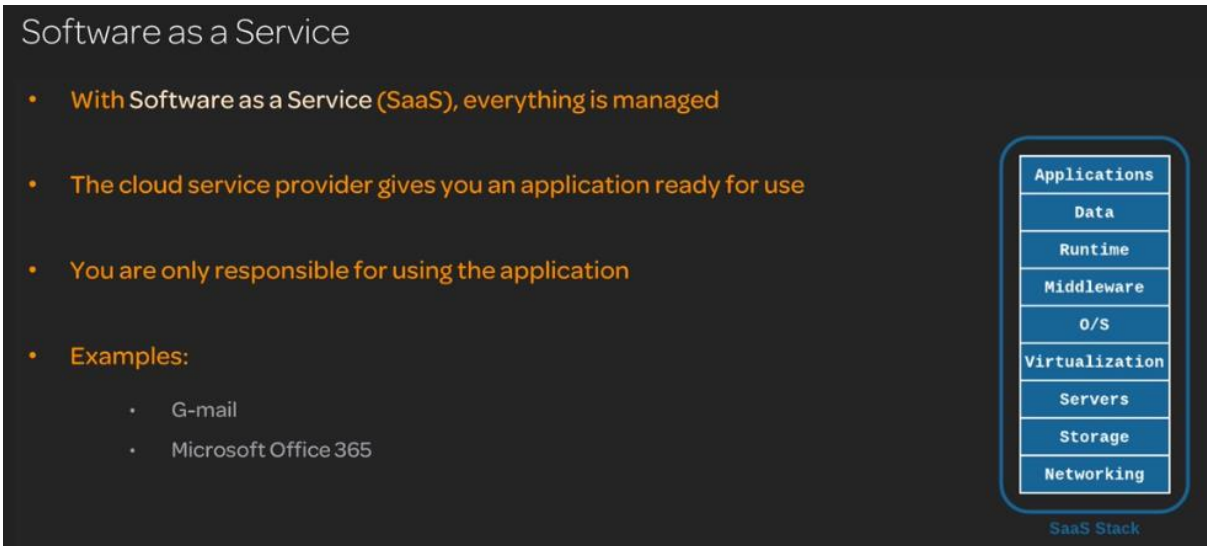
Applications
Data
Runtime
Middleware
O/S
Virtualization
Servers
Storage
Networking
IaaS Stack



Platform as a Service

- With Platform as a Service (PaaS), everything below the Application and Data layers is abstracted
- The cloud service provider gives you a way to deploy an app and use databases
- You are only responsible for managing the app and data
- Examples:
  - AWS Elastic Beanstalk
  - Heroku
  - Google App Engine

Applications
Data
Runtime
Middleware
O/S
Virtualization
Servers
Storage
Networking
PaaS Stack

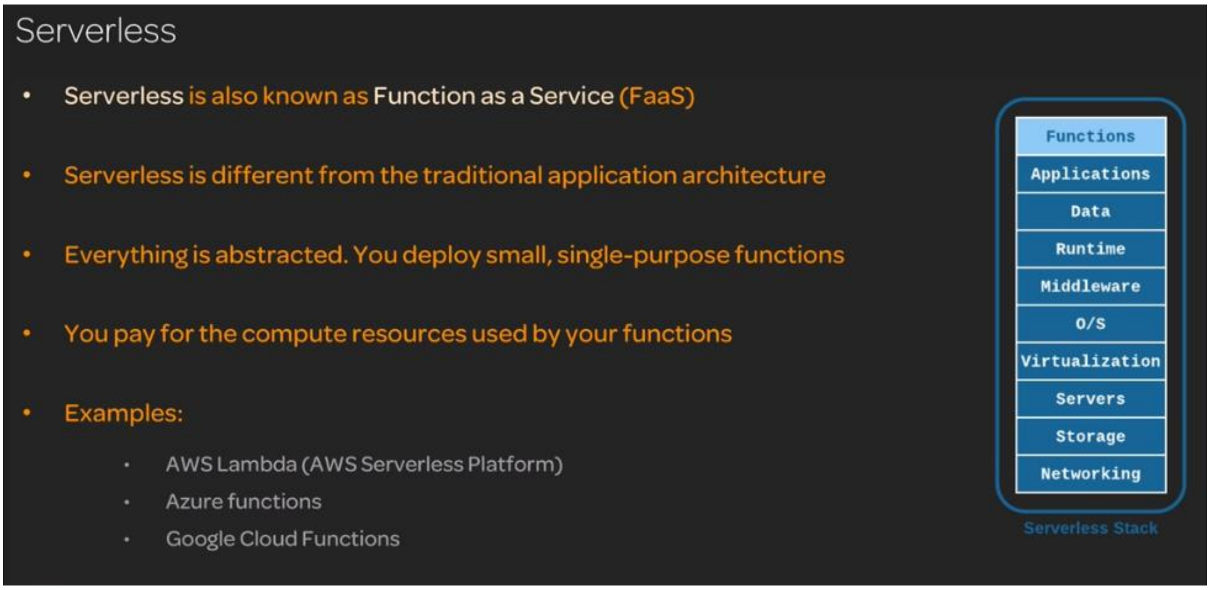


Software as a Service

- With Software as a Service (SaaS), everything is managed
- The cloud service provider gives you an application ready for use
- You are only responsible for using the application
- Examples:
  - G-mail
  - Microsoft Office 365

Applications
Data
Runtime
Middleware
O/S
Virtualization
Servers
Storage
Networking

SaaS Stack



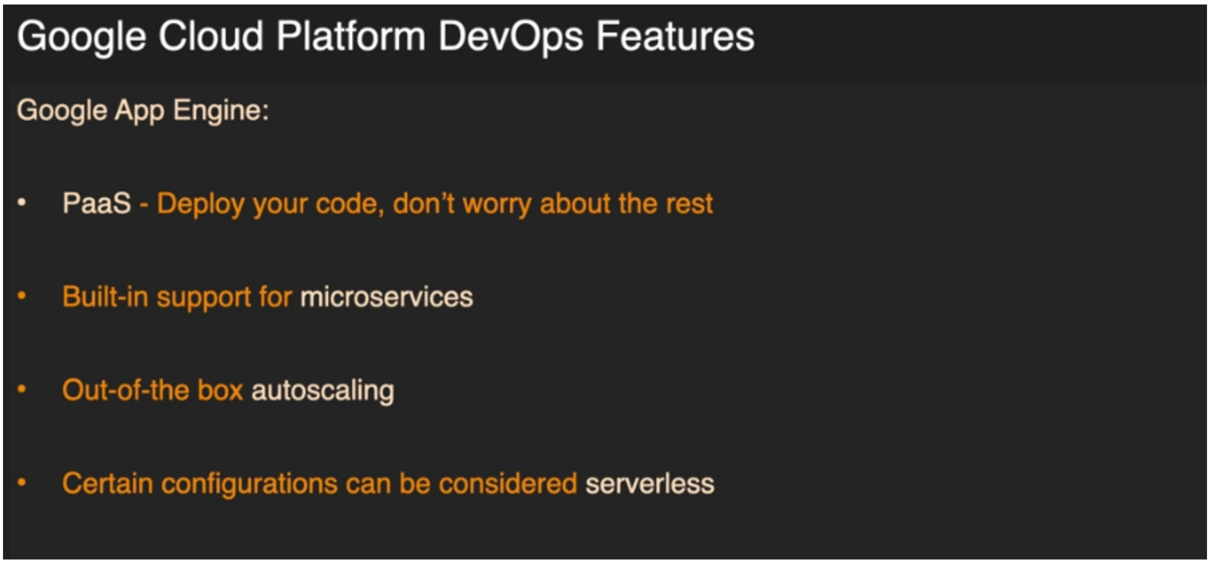
Serverless

- Serverless is also known as Function as a Service (FaaS)
- Serverless is different from the traditional application architecture
- Everything is abstracted. You deploy small, single-purpose functions
- You pay for the compute resources used by your functions
- Examples:
  - AWS Lambda (AWS Serverless Platform)
  - Azure functions
  - Google Cloud Functions

Functions
Applications
Data
Runtime
Middleware
O/S
Virtualization
Servers
Storage
Networking

Serverless Stack

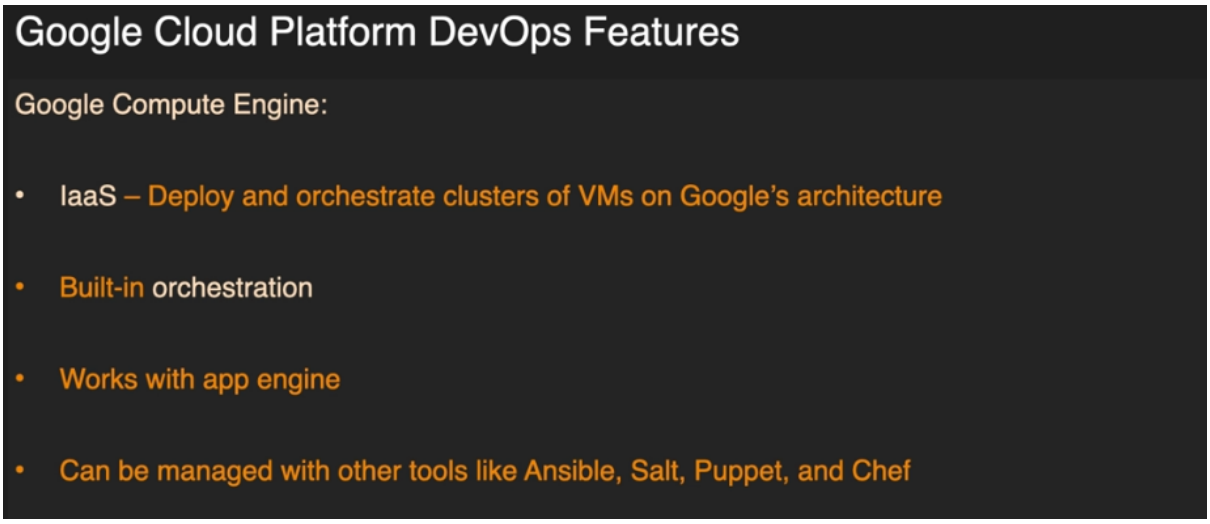
Google Cloud Platform DevOps Feature



Google Cloud Platform DevOps Features

Google App Engine:

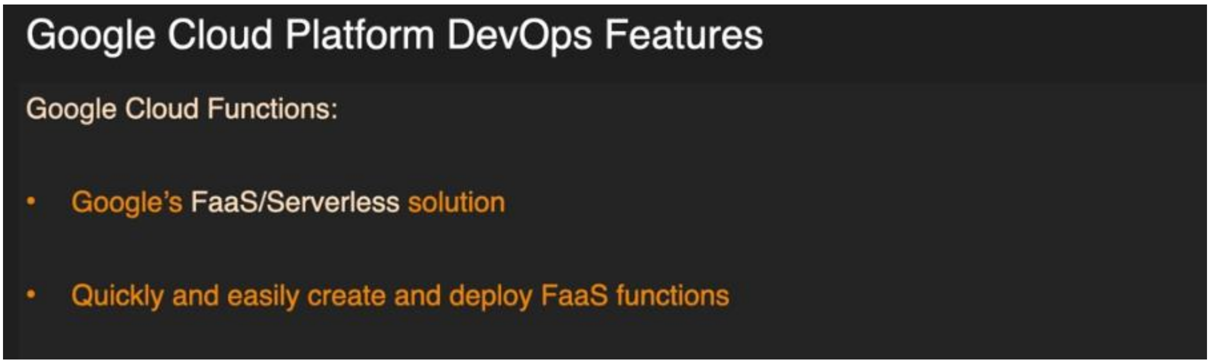
- PaaS - Deploy your code, don't worry about the rest
- Built-in support for microservices
- Out-of-the box autoscaling
- Certain configurations can be considered serverless



Google Cloud Platform DevOps Features

Google Compute Engine:

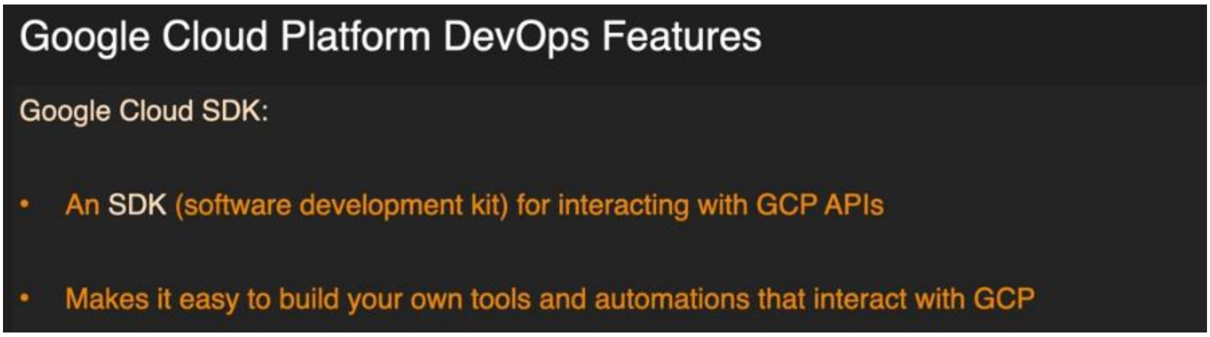
- IaaS – Deploy and orchestrate clusters of VMs on Google's architecture
- Built-in orchestration
- Works with app engine
- Can be managed with other tools like Ansible, Salt, Puppet, and Chef



Google Cloud Platform DevOps Features

Google Cloud Functions:

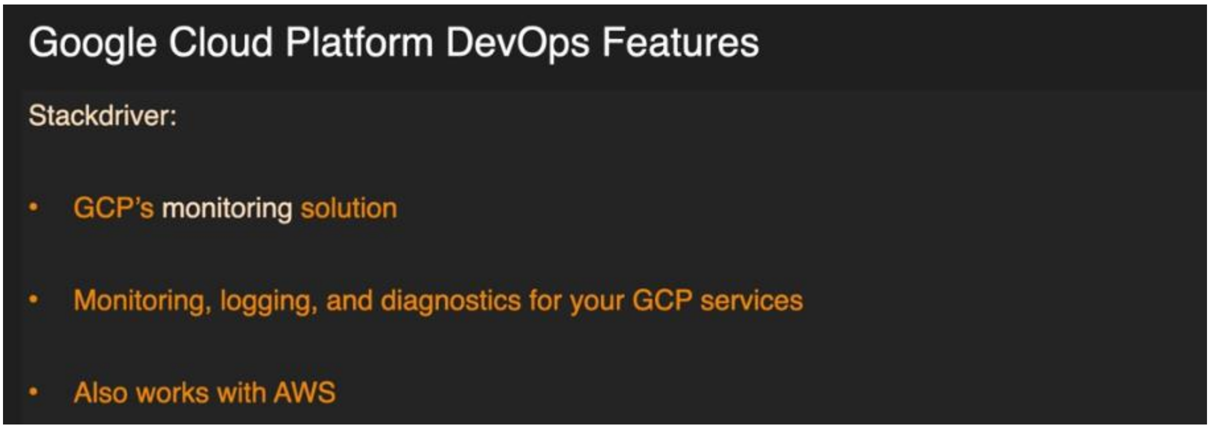
- Google's FaaS/Serverless solution
- Quickly and easily create and deploy FaaS functions



Google Cloud Platform DevOps Features

Google Cloud SDK:

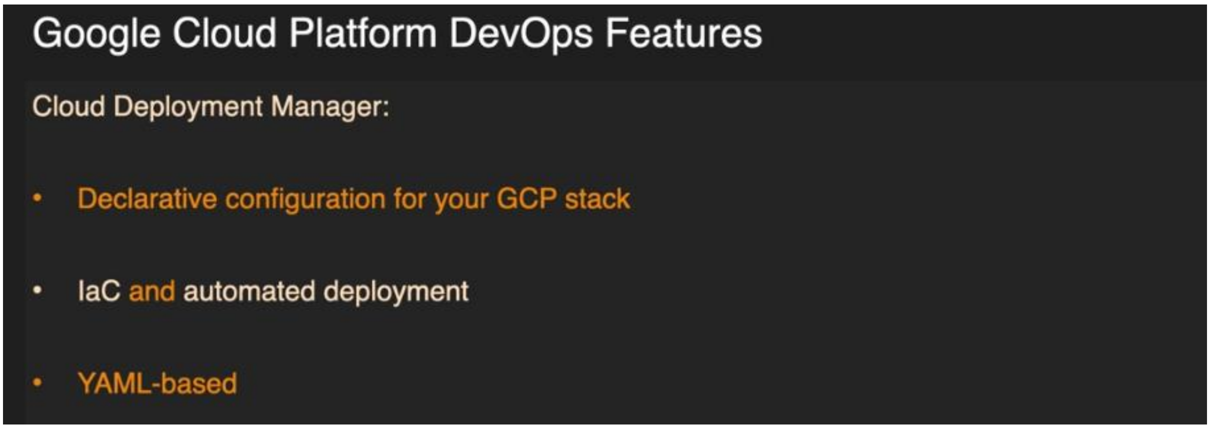
- An SDK (software development kit) for interacting with GCP APIs
- Makes it easy to build your own tools and automations that interact with GCP



Google Cloud Platform DevOps Features

Stackdriver:

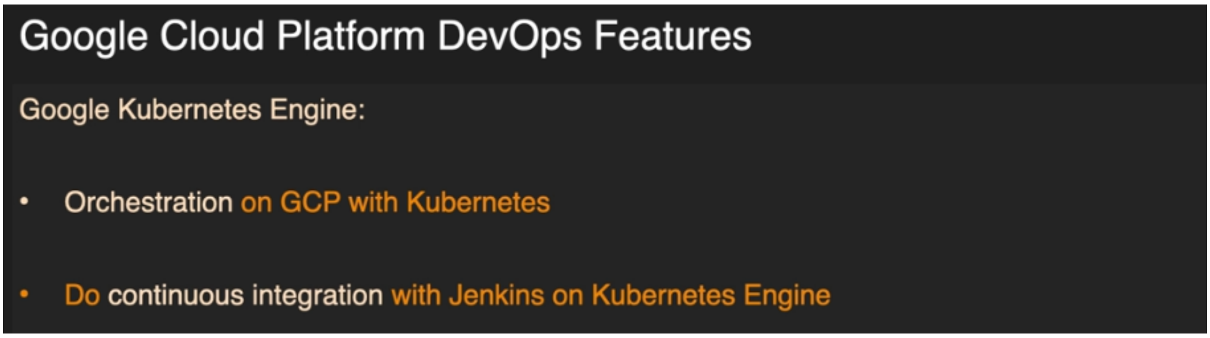
- GCP's monitoring solution
- Monitoring, logging, and diagnostics for your GCP services
- Also works with AWS



Google Cloud Platform DevOps Features

Cloud Deployment Manager:

- Declarative configuration for your GCP stack
- IaC and automated deployment
- YAML-based

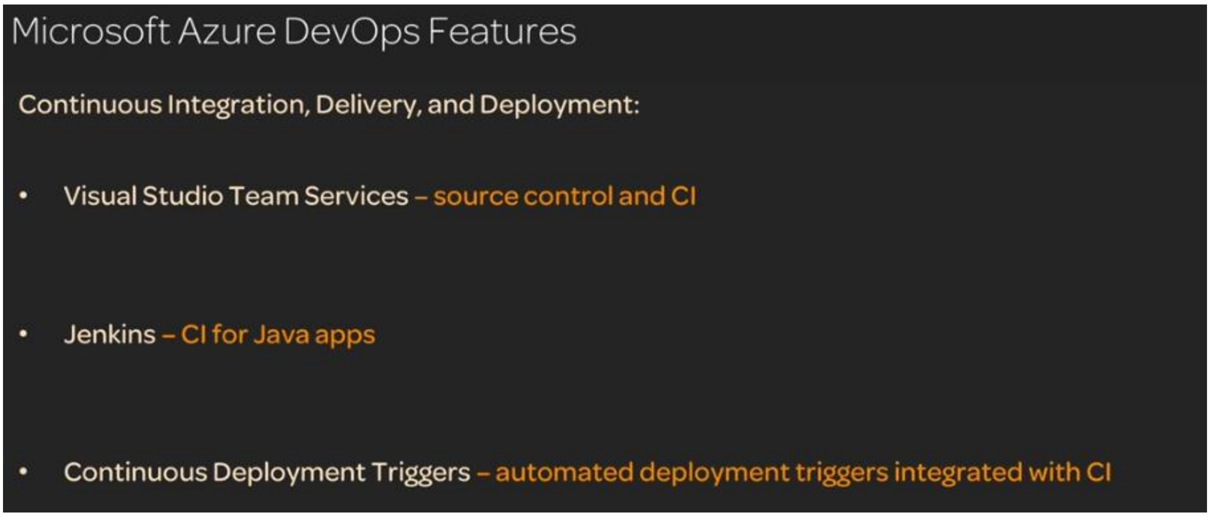


Google Cloud Platform DevOps Features

Google Kubernetes Engine:

- Orchestration on GCP with Kubernetes
- Do continuous integration with Jenkins on Kubernetes Engine

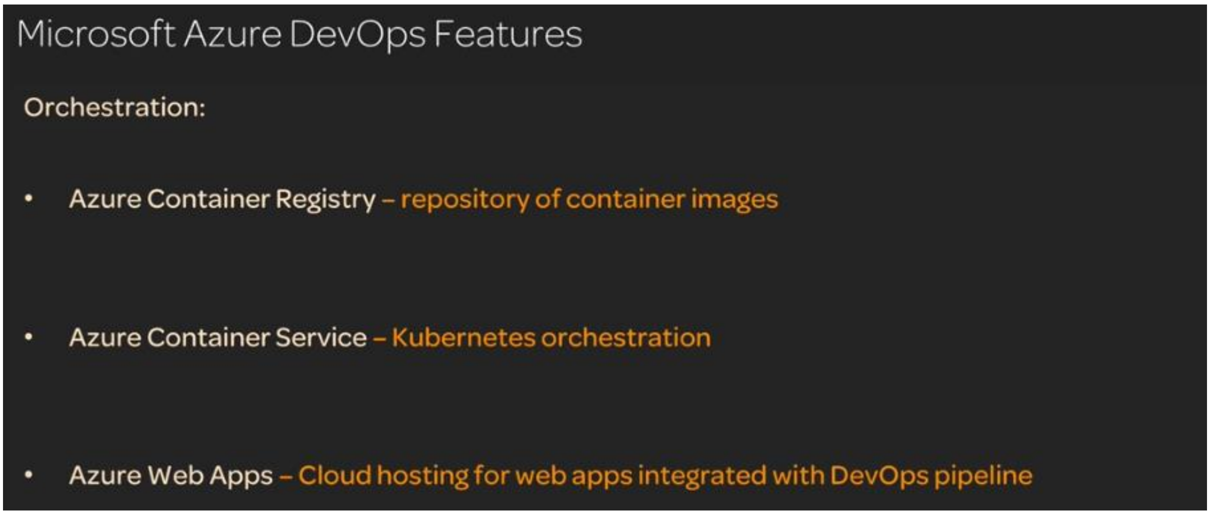
Microsoft Azure DevOps Feature



Microsoft Azure DevOps Features

Continuous Integration, Delivery, and Deployment:

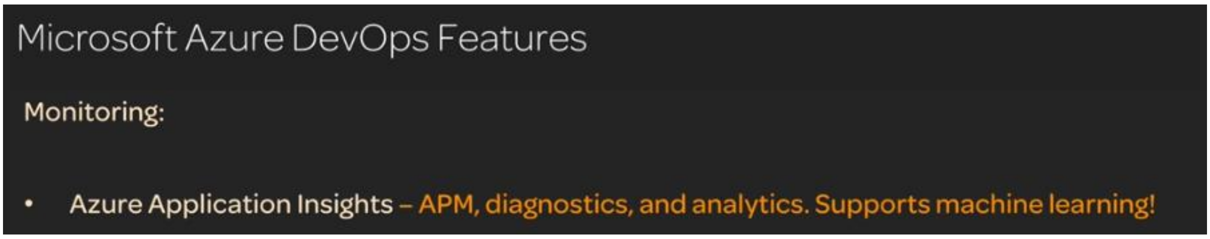
- Visual Studio Team Services – source control and CI
- Jenkins – CI for Java apps
- Continuous Deployment Triggers – automated deployment triggers integrated with CI



Microsoft Azure DevOps Features

Orchestration:

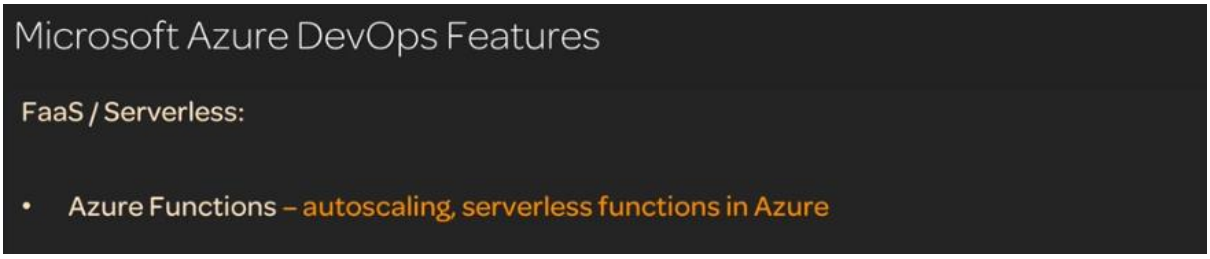
- Azure Container Registry – repository of container images
- Azure Container Service – Kubernetes orchestration
- Azure Web Apps – Cloud hosting for web apps integrated with DevOps pipeline



Microsoft Azure DevOps Features

Monitoring:

- Azure Application Insights – APM, diagnostics, and analytics. Supports machine learning!

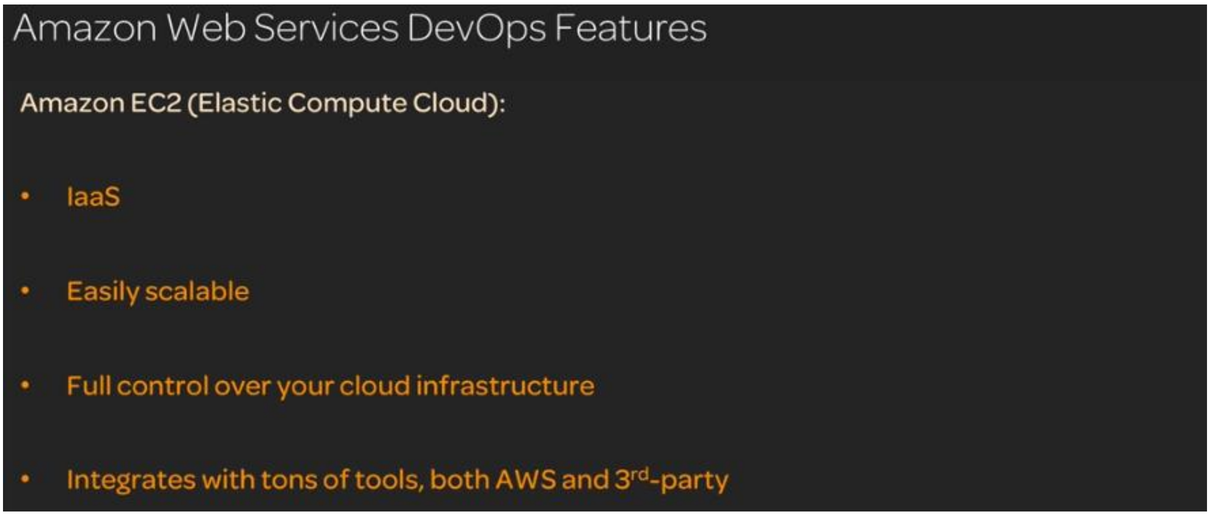


Microsoft Azure DevOps Features

FaaS / Serverless:

- Azure Functions – autoscaling, serverless functions in Azure

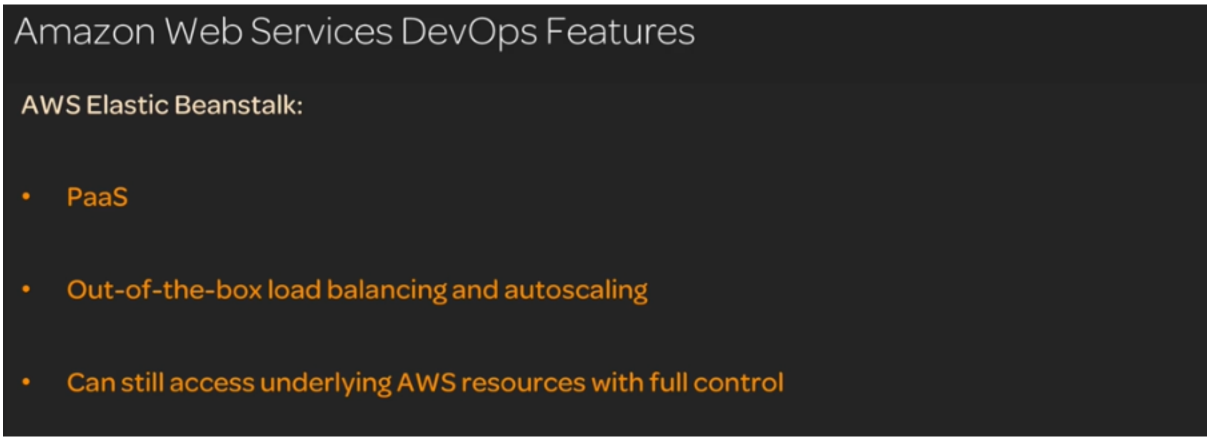
Amazon Web Services DevOps Features



Amazon Web Services DevOps Features

Amazon EC2 (Elastic Compute Cloud):

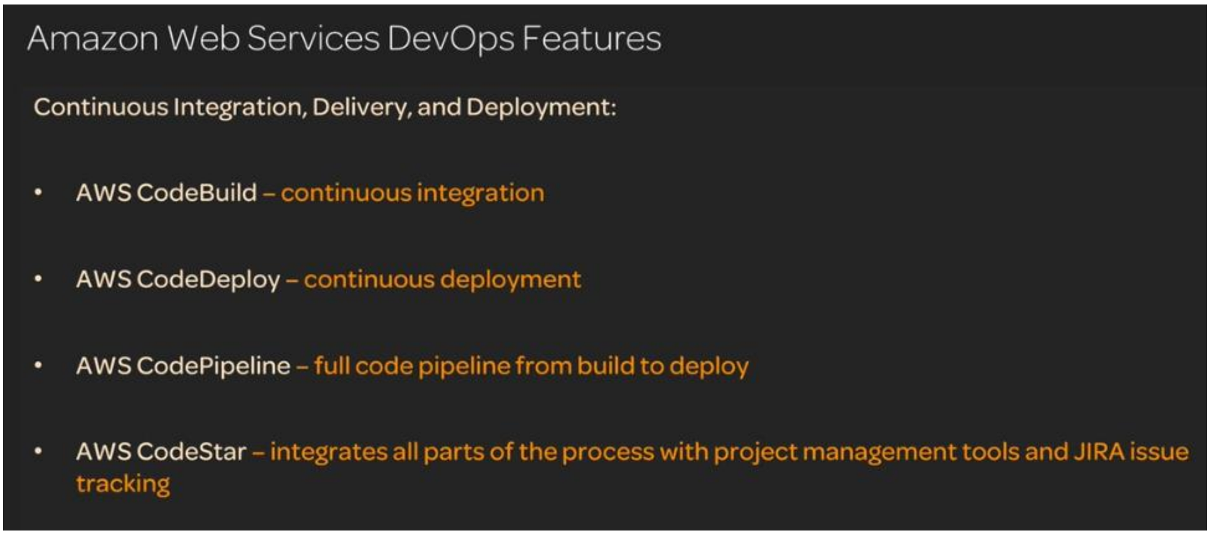
- IaaS
- Easily scalable
- Full control over your cloud infrastructure
- Integrates with tons of tools, both AWS and 3rd-party



Amazon Web Services DevOps Features

AWS Elastic Beanstalk:

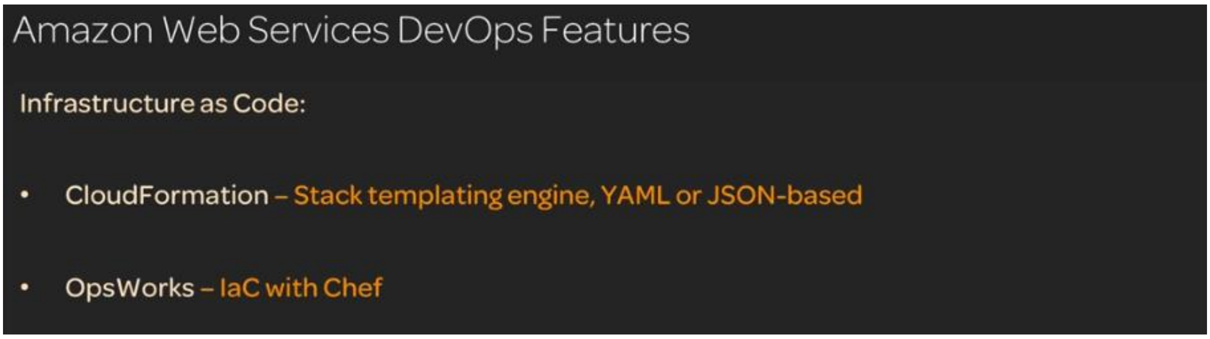
- PaaS
- Out-of-the-box load balancing and autoscaling
- Can still access underlying AWS resources with full control



Amazon Web Services DevOps Features

Continuous Integration, Delivery, and Deployment:

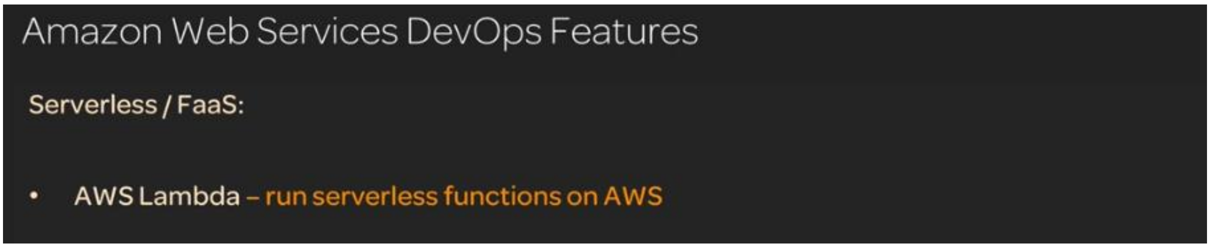
- AWS CodeBuild – continuous integration
- AWS CodeDeploy – continuous deployment
- AWS CodePipeline – full code pipeline from build to deploy
- AWS CodeStar – integrates all parts of the process with project management tools and JIRA issue tracking



Amazon Web Services DevOps Features

Infrastructure as Code:

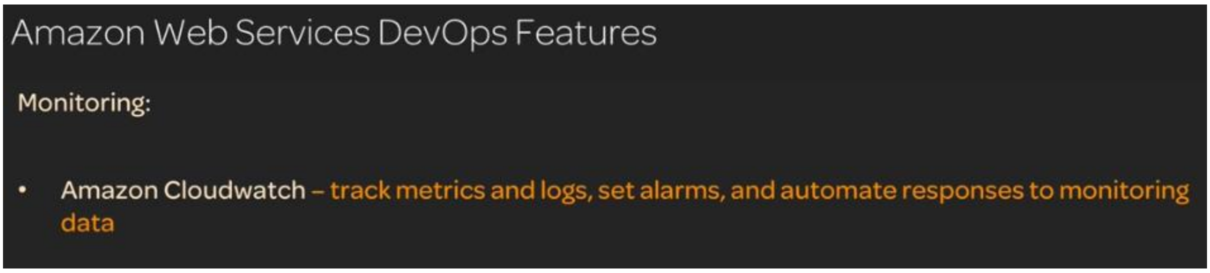
- CloudFormation – Stack templating engine, YAML or JSON-based
- OpsWorks – IaC with Chef



Amazon Web Services DevOps Features

Serverless / FaaS:

- AWS Lambda – run serverless functions on AWS



Amazon Web Services DevOps Features

Monitoring:

- Amazon Cloudwatch – track metrics and logs, set alarms, and automate responses to monitoring data