Introduction to Jenkins

Module 1: CI/CD Overview

Software Development

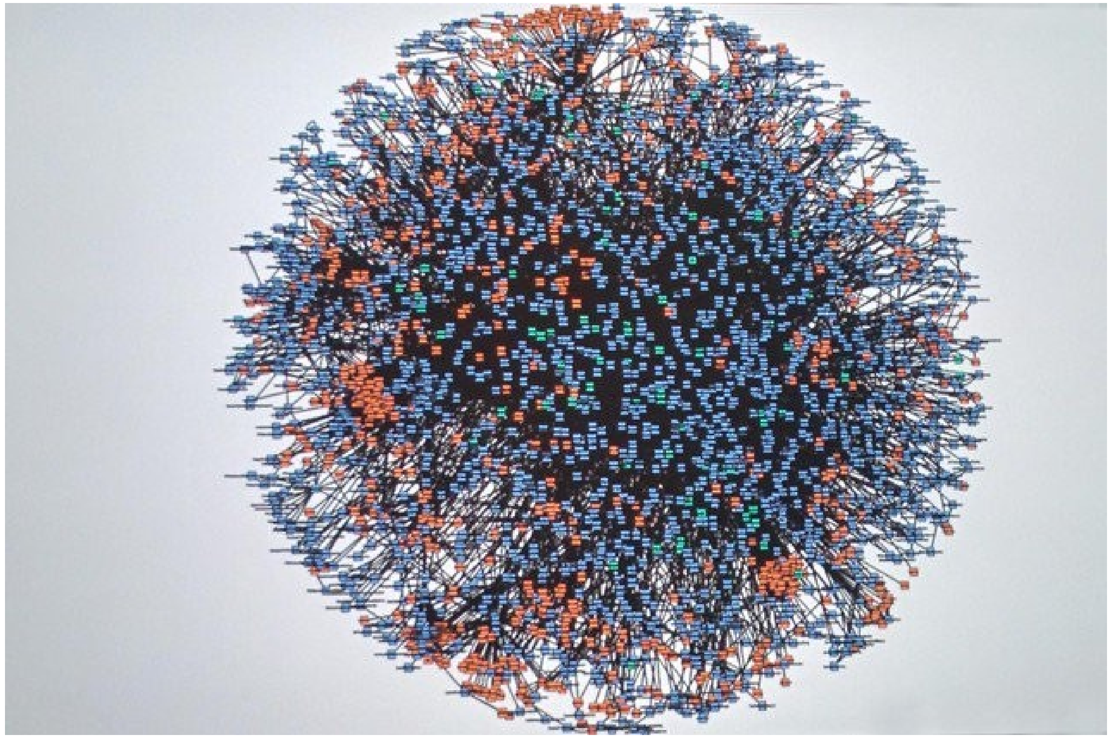
- Software development has gone through a series of evolutionary steps
- 1950s – 1980s
  - Dominated by mainframes
  - Batch programming and procedural languages
  - Monolithic applications
  - Software development was usually a series of manual steps
  - Applications were relatively small in terms of the amount of code

Software Development

- 1990s – 2010s
  - Cheaper hardware allowed distributed computing
  - Networking and Internet based applications started to be developed and deployed
  - Real time and Object-Oriented programming went mainstream
  - Adoption of virtualization to make more effective use of hardware
  - Business drivers
    - *Be more responsive to market and customers*
    - *Reduce delivery times*
    - *Be able to scale operations and code base*
  - Agile methodologies and automation tools become important for developers in responding to business drivers

Software Development

- 2010s – Now
  - Massive increases in the complexity of application and operational codebase
    - *Petabytes of streaming data*
    - *Billions of transactions*
    - *Mission critical fault tolerance*
  - Rise of microservices
    - *Results in a “Death Star” architecture*
    - *Not manageable by previous operations techniques*
    - *Image: Amazon in 2008 – each node represents a running code module*
  - DevOps becomes mainstream
    - *Infrastructure as code (IaC) to support virtualization*
    - *Automation support through the development and deployment lifecycle*
  - Pipelines and automation tools are required for these large and complex applications



Moore’s Law

- Historically, advances in hardware capabilities tend to be exponential.
  - Coupled with similar decrease in costs
- This is referred to as Moore’s law
  - Originally formulated as a measure of the amount of computation power of a chip in terms of “transistors”
  - The original formulation is no longer valid because of changes in chip and CPU architecture
  - But the term Moore’s law is now generally used to describe increasing compute and storage capabilities coupled with dropping costs

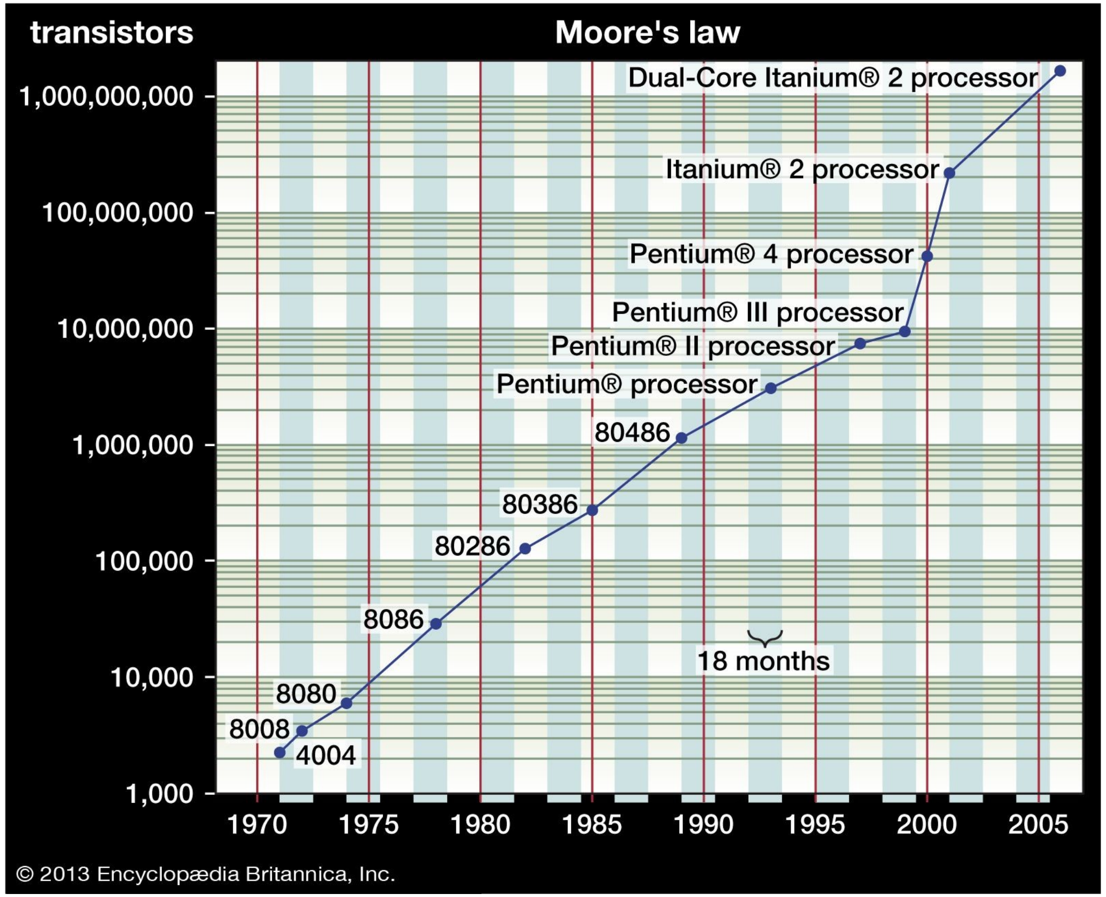


Image Credit: https://www.britannica.com/technology/Moores-law

Hardware Capabilities

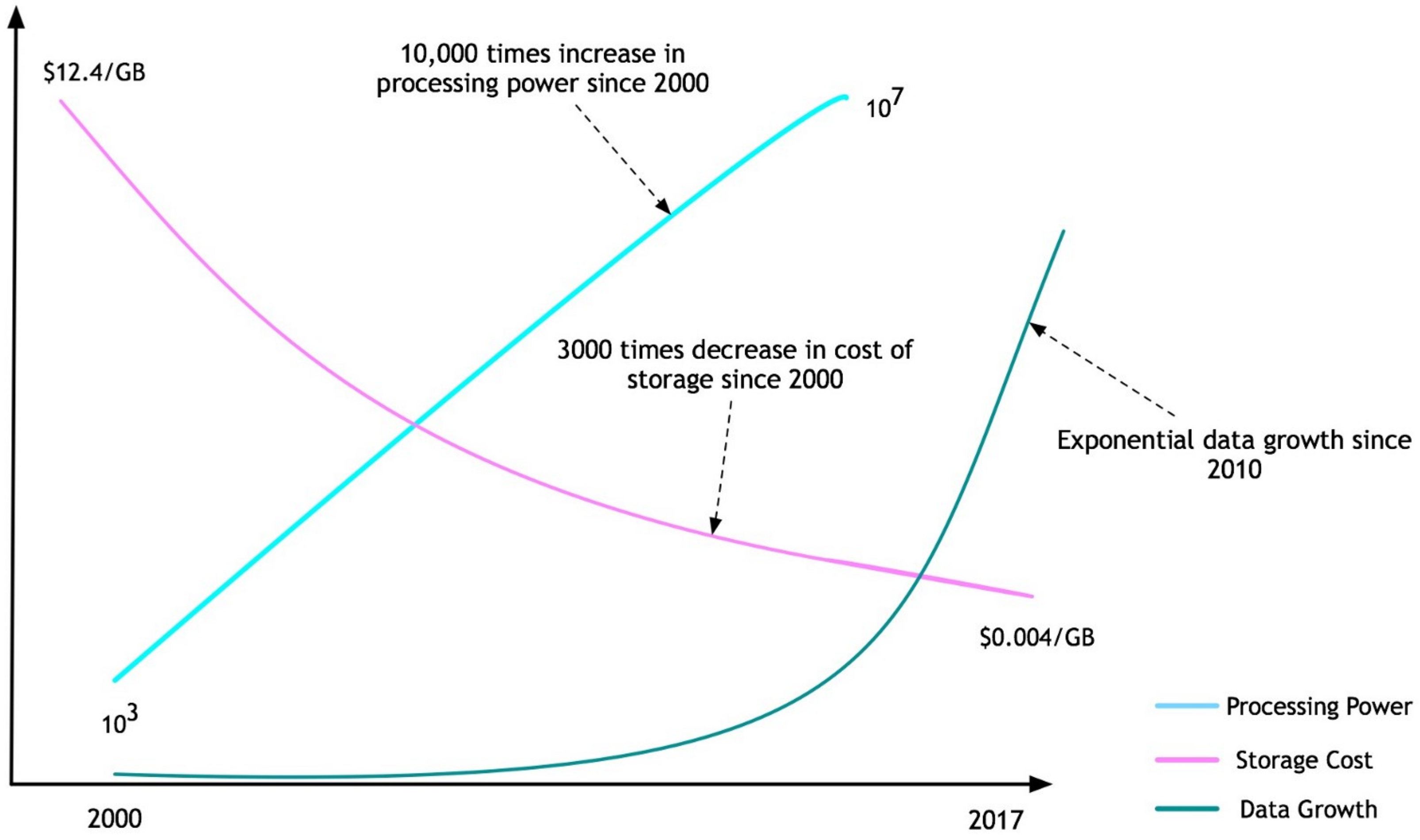
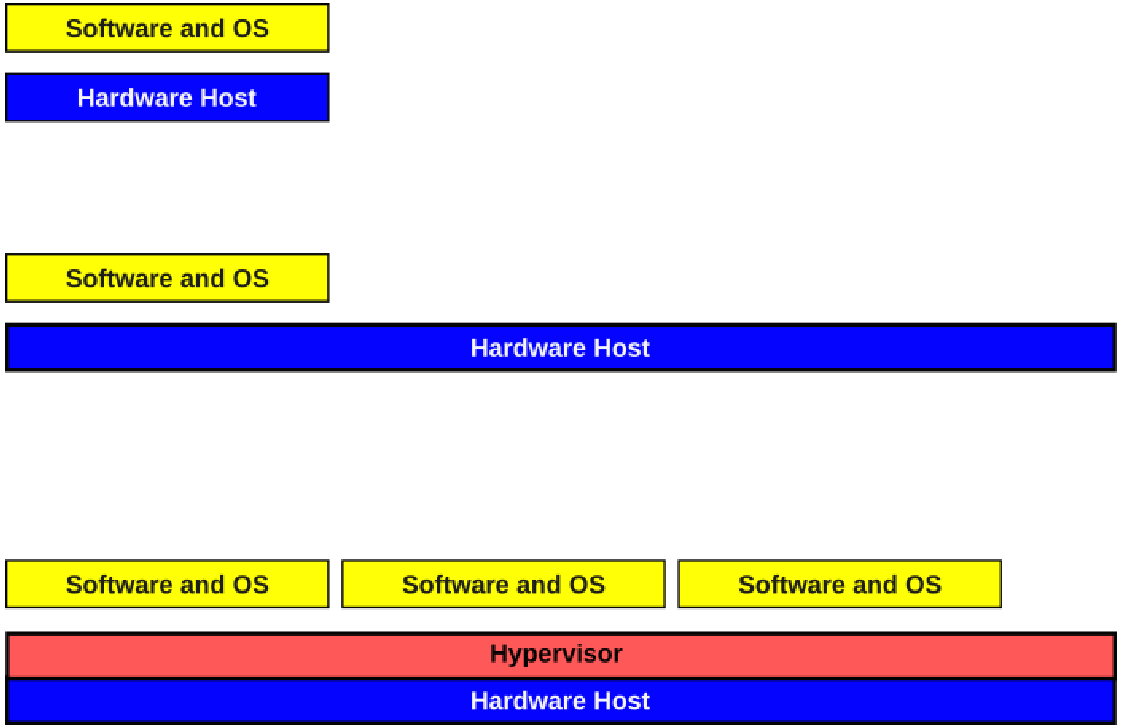


Image Credit: https://blog.govnet.co.uk/technology/reflection-on-the-maturity-of-bim-and-digital-twins

Hardware Outstrips Software

- Historically, the limit of computing was hardware capabilities
  - With the increases in hardware capabilities, the situation reversed
  - The model of running and developing a single application in an OS running on a hardware platform didn't scale well
- Hardware capabilities were under utilized
  - Adding virtualization allowed for multiple OS installations (VMs) to use the same hardware
  - The problem became how to effectively utilize these new hardware capabilities



Containers

- An alternative to VMs was development of Linux containers
  - These allowed very lightweight self-contained applications to be independently deployed
  - Like VMs, these are much smaller without the overhead of an OS and libraries
  - Containers only contain what is necessary to run the application
  - Containers don't require infrastructure beyond the container engine

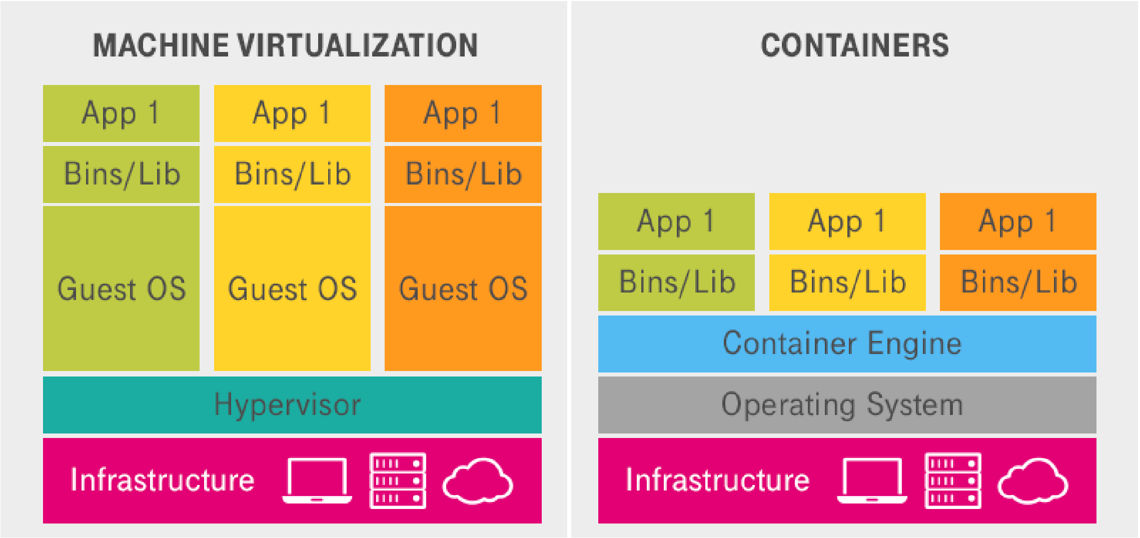


Image Credit: https://www.open-telekom-cloud.com/en/blog/cloud-computing/container-vs-vm

Microservices

- Microservices are a software architectural pattern
  - Enables software operations to use compute resources effectively and to operate in a more robust and reliable manner
  - Focus is on issues around scaling in both the development and operations
- Deploying microservices requires
  - Supporting technologies in operational environments like Kubernetes, Kafka, Docker
  - Analysis and design techniques for building a component based software architecture
  - Code and application design techniques to make code “microservices ready”
  - Production techniques to support the successful deployment of a microservice
- An integral part of microservices development is the use of CI/CD

Agile – DevOps - CI/CD

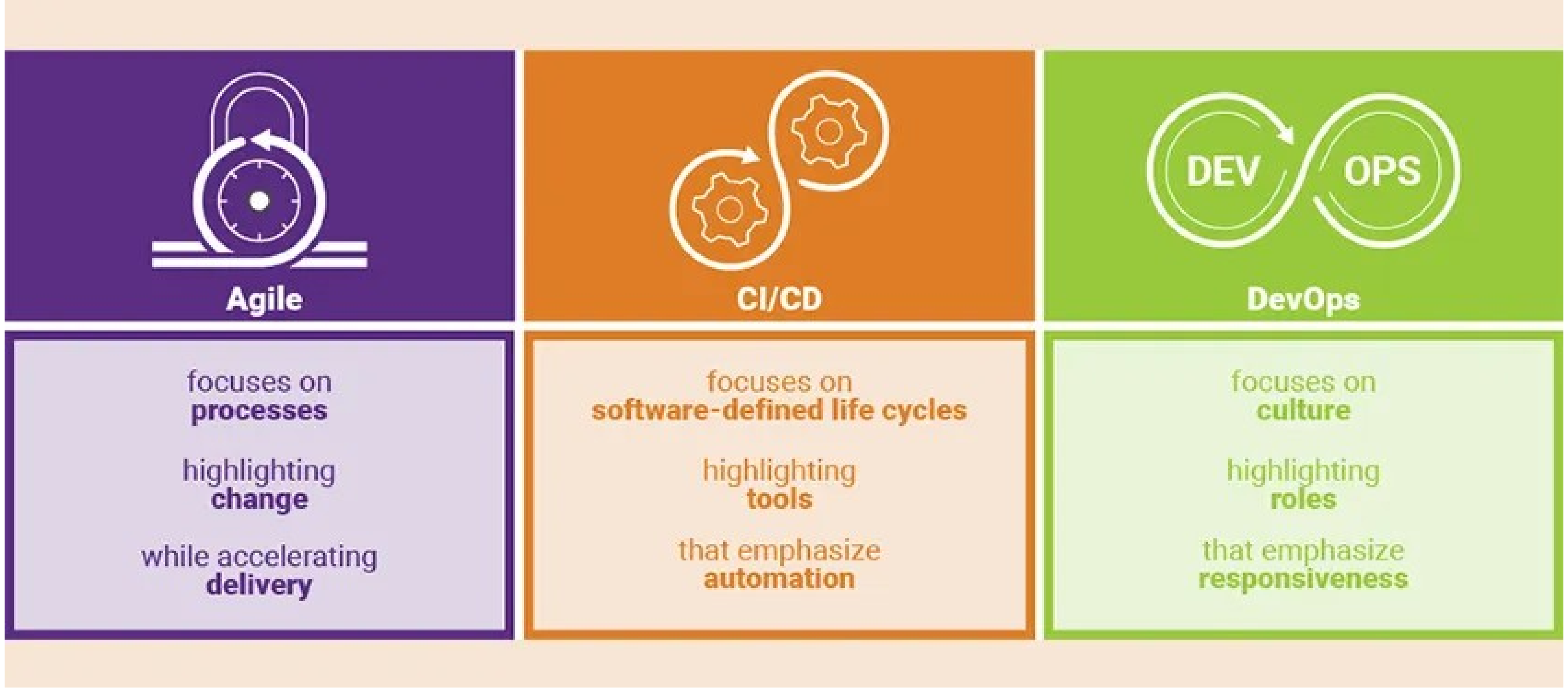
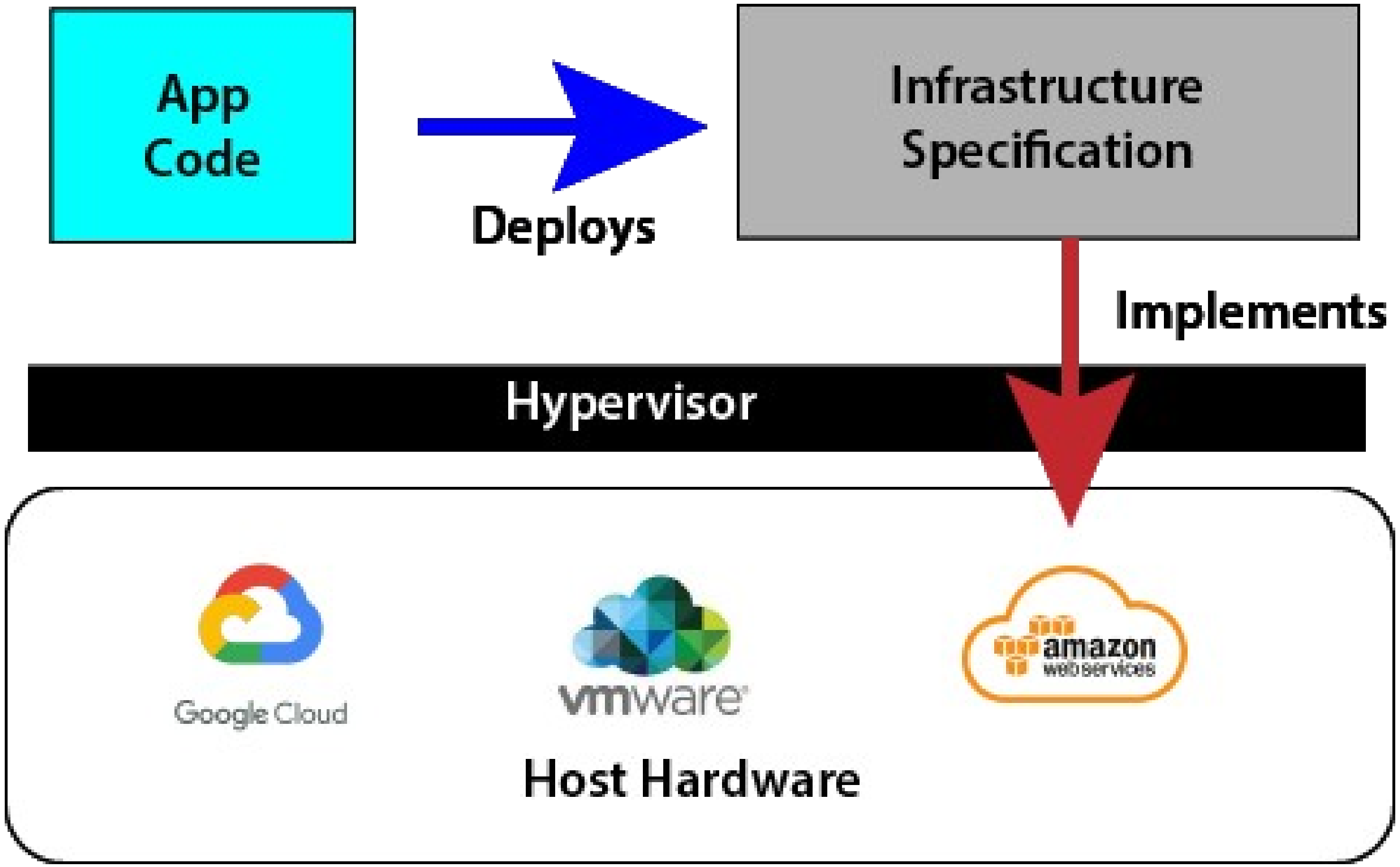


Image Credit: https://medium.com/@vidhatanandV/achieving-value-with-progressive-delivery-438507cfbc6b

Infrastructure as Code

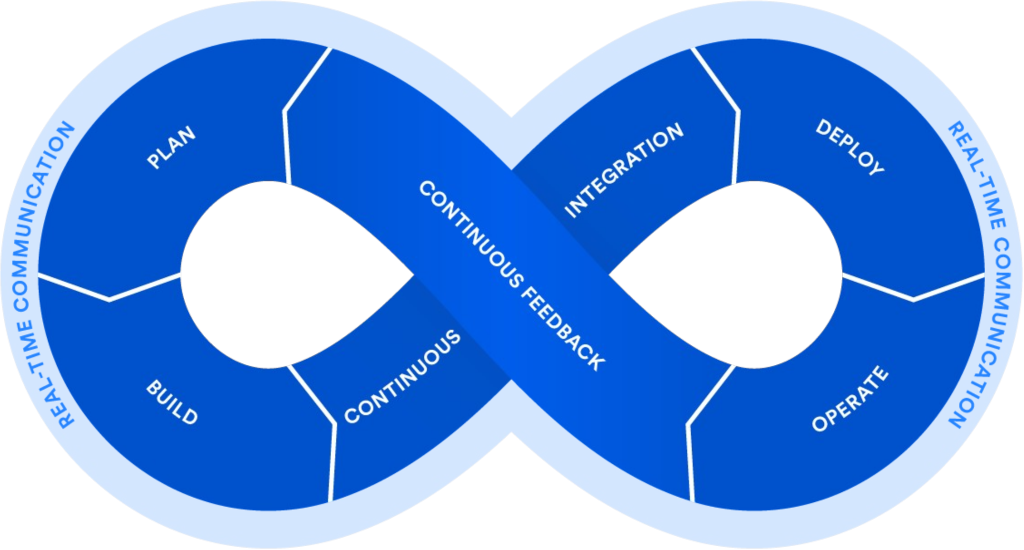
- Virtual machines directed the hypervisor to allocate hardware from the host systems
  - The host hardware was presented to the VM as virtual devices
  - Allowed AWS and others to implement VMs and other virtual hardware “in the cloud”
    - *Referred to as “Infrastructure as a Service” (IaaS)*
- This meant that provisioning an operational environment
  - Did not mean working with hardware directly
  - Instead, a specification or set of instructions to the hypervisor is written in some IaC language
    - *For example, Terraform is one of the most commonly used tools for cloud infrastructure*
  - The spec tells the hypervisor what virtual hardware needs to be created and deployed
  - The hypervisor does the hardware allocation that maps to the virtual device in the VM
- This written description of the virtual environment is “infrastructure as code”

Infrastructure as Code



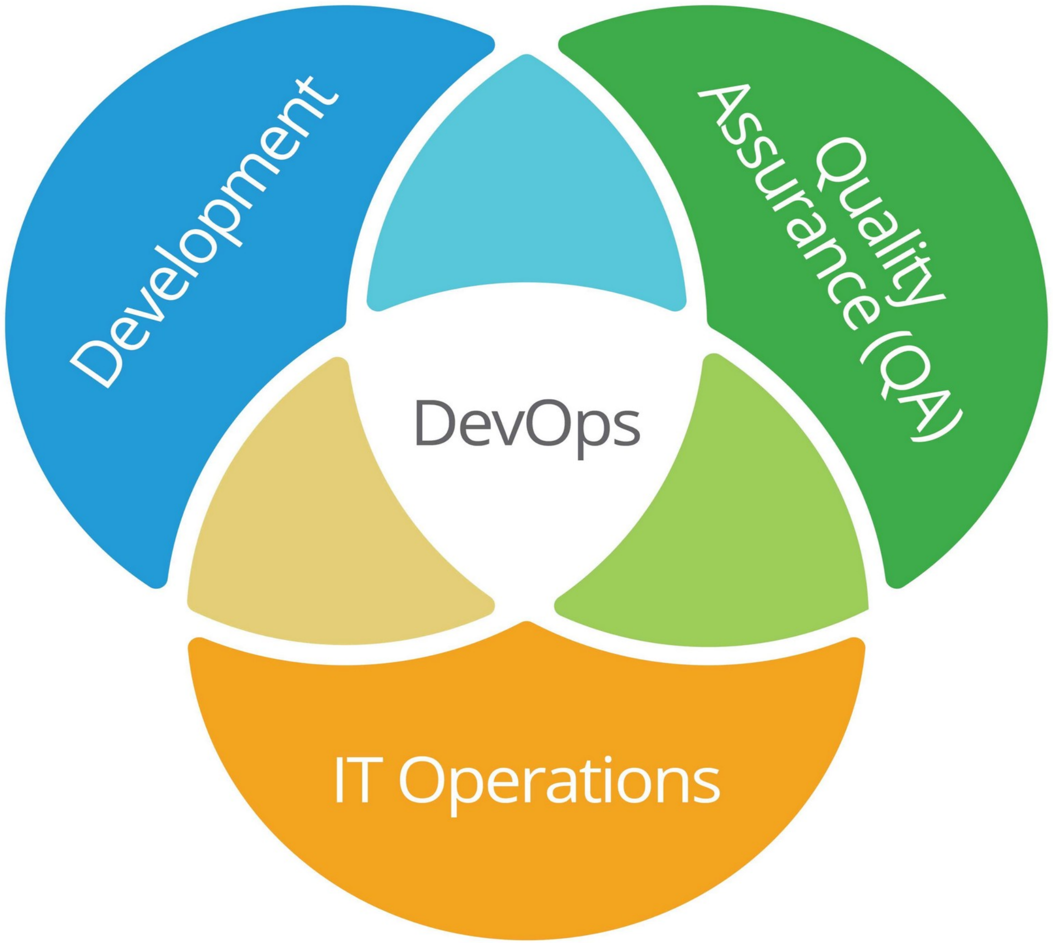
DevOps

- Driven by virtualization and IaC
  - Dev and Ops had been two separate worlds
  - Dev used automated tools
    - *Build tools, SCMs, etc.*
  - Ops was manual and bare metal
    - *Physically installing software, running cables, etc.*
- Virtualization turned it all into code
  - Now the same tools can be used in the entire life cycle of a software product
  - Opportunity for full process automation support
  - It allowed the integration of the product support phase with development



The Goal of DevOps

- De-siloize the three areas involved in software development
- To get everyone using the same sorts of tools, practices and automation processes
- Operations infrastructure is now IaC
  - The same processes are used to manage both application code and infrastructure code
- Allows for integration of the processes used in development with operations
  - Able to respond to real time feedback on how the applications running in the Ops environment



Defining CI/CD

- CI/CD is not a methodology
  - Continuous Integration / Continuous Delivery (Deployment)
  - It is not Agile or DevOps, although both rely on and use it extensively
- CI/CD is a process automation applied to software engineering
  - Not a development or process methodology
  - Similar to other kinds of automation
  - Improves process efficiency and effectiveness
- CI/CD is process agnostic
  - Can be used anywhere a software engineering process is well defined
  - Using CI/CD with bad processes makes them worse

A fool with a tool is still a fool

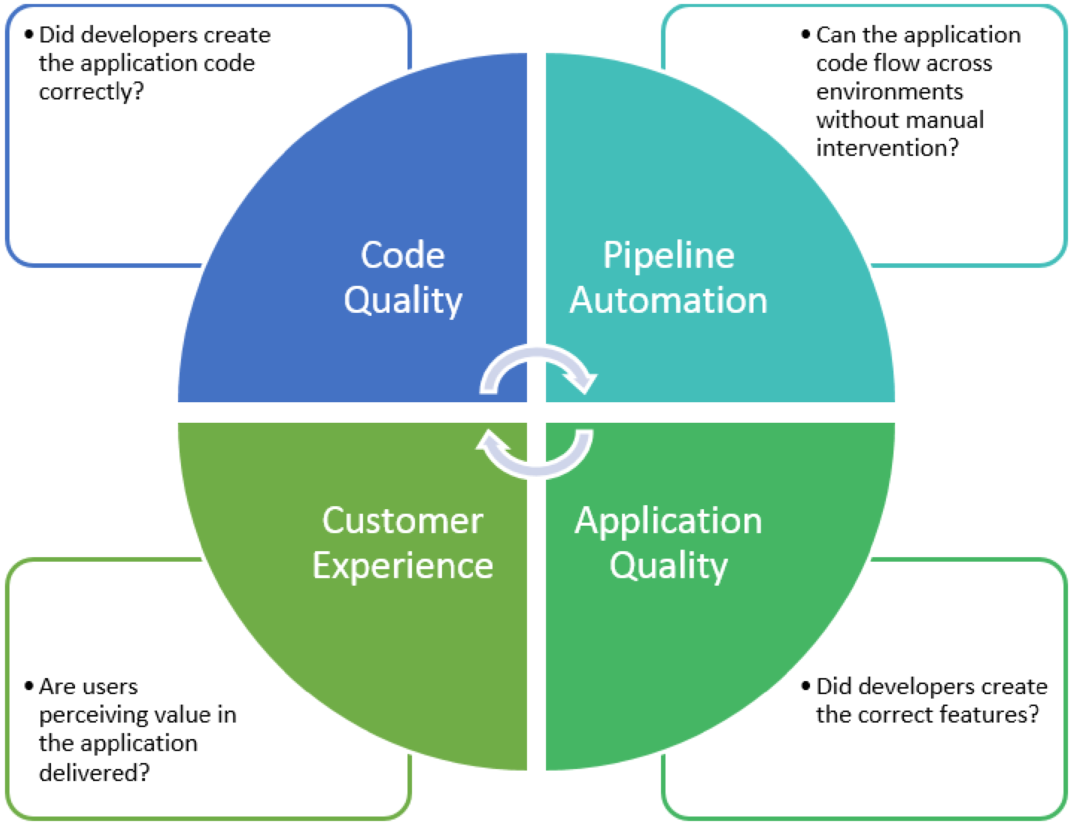
Martin Fowler

A computer lets you make more mistakes faster than any invention in human history – with the possible exceptions of handguns and tequila

Mitch Ratcliffe

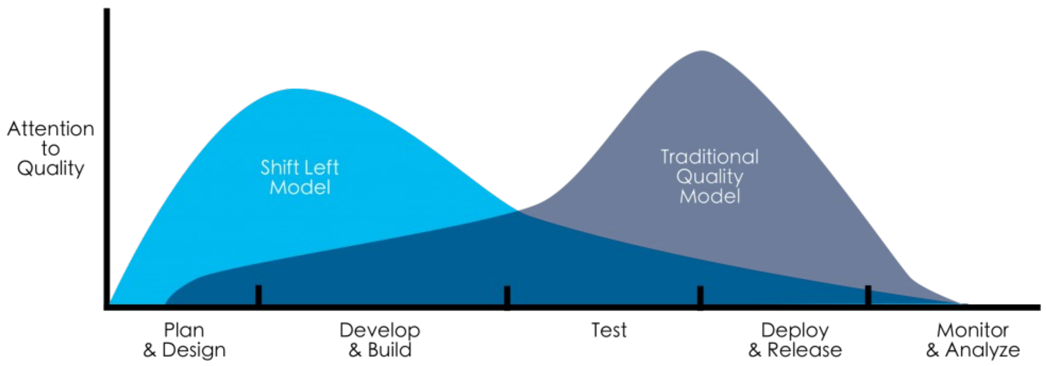
Continuous Testing

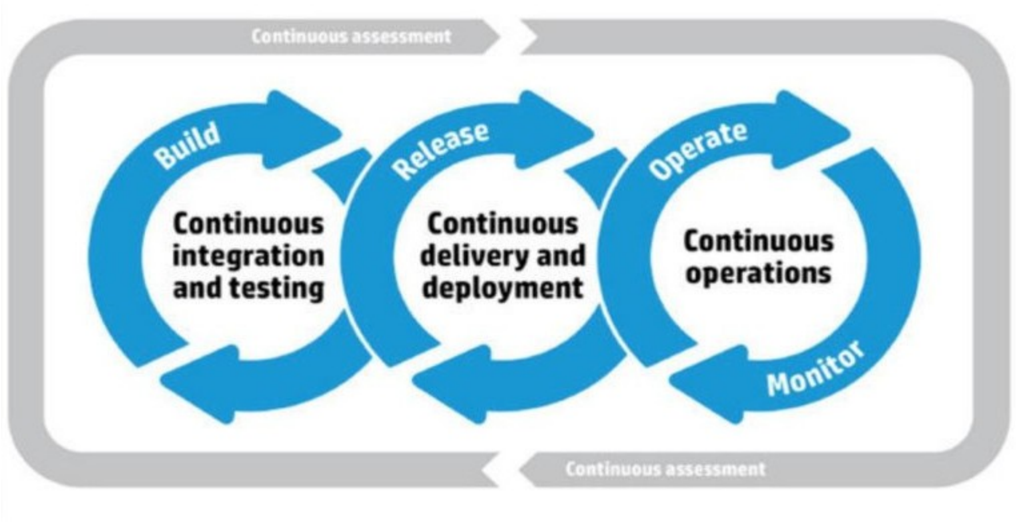
- Does not replace human based testing
  - Techniques like pair programming and code reviews are still critical to software QA
- Automated testing can now be built into the entire development and operations cycle
- Creates “quality gates”
  - These are automated tests that must pass for the pipeline to continue
  - Development pipelines abort when tests fail
  - Identifies problems early in production so they can be remediated sooner
  - Adding continuous security testing and security planning is called DevSecOps



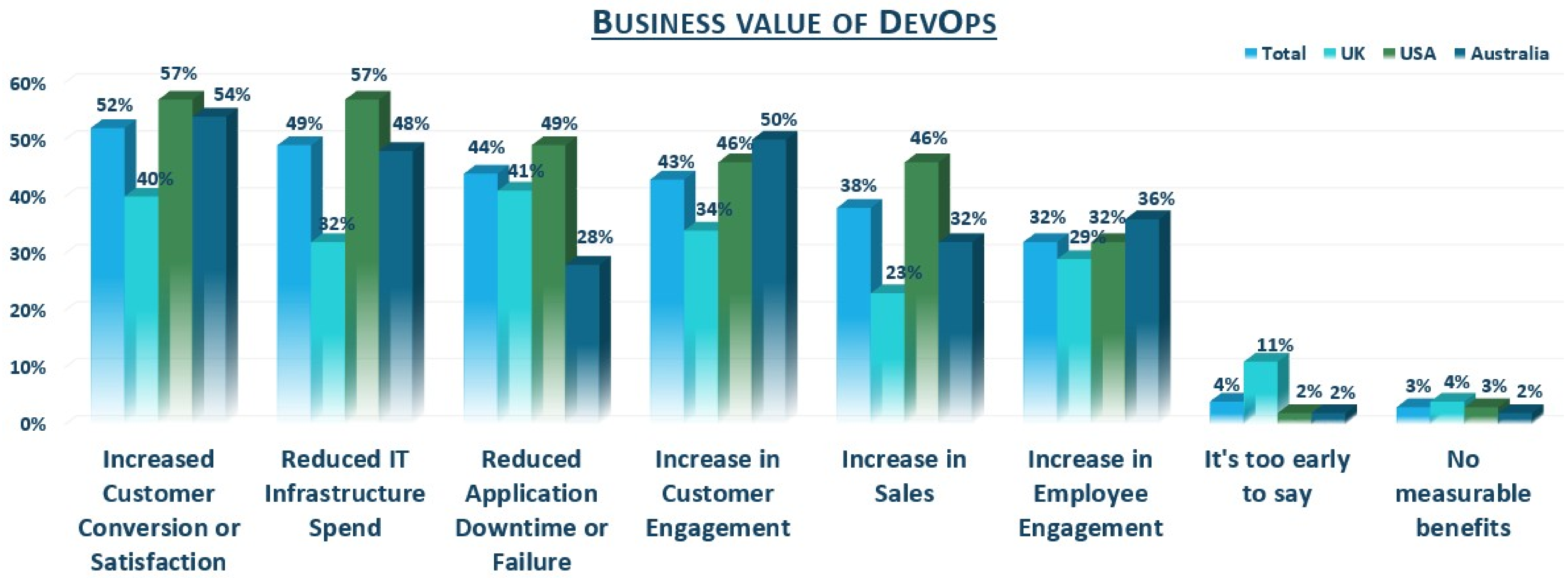
Continuous Testing

- Continuous Testing
  - Every artifact is tested as it is created
  - Shift Left Model
  - Test early, test often
- CI/CD also adds
  - Automated testing at every stage
- CT is triggered by events in the CI/CD process
  - Checking in code => automated unit testing
  - Build => integration testing

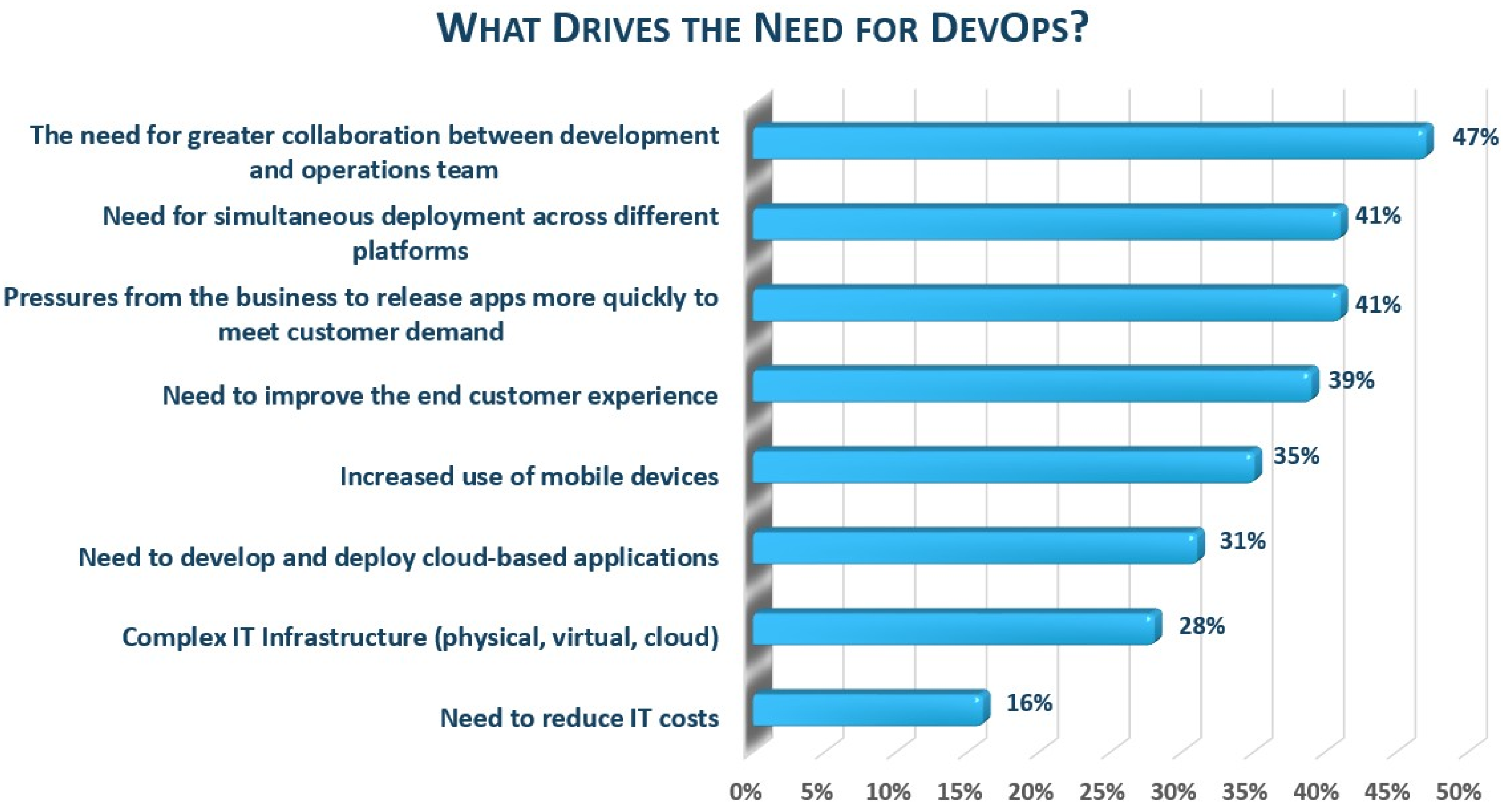




DevOps Cost Benefit



DevOps Cost Benefit



DevOps Cost Benefit

Functions	Previous Time Frame	Present Time Frame	DevOps Benefit
Project initiation	10 days	2 days	80% faster
Overall time to development	55 days	3 days	94% faster
Build verification test availability	18 hours	< 1 hour	94% faster
Overall time to production	3 days	2 days	33% faster
Time between releases	12 months	3 months	75% faster

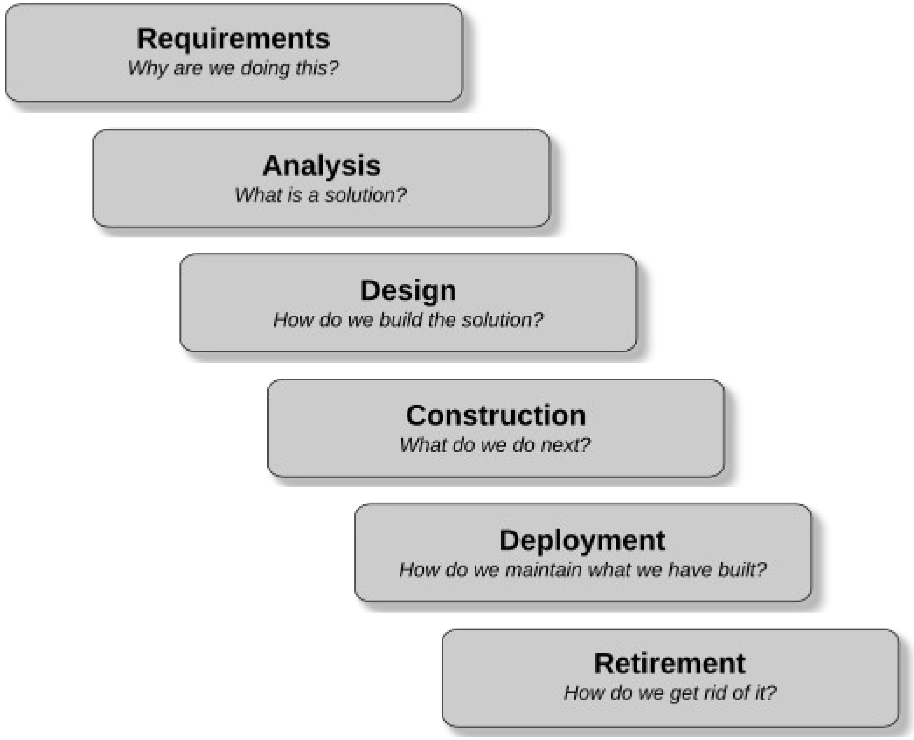
DevOps, clearly an extension of lean and agile principles, was as much, in IBM, born of necessity to respond to a pervasive industry mandate to "do more with less" and has evolved to "quality software faster."

- Kristof Kloeckner,
General Manager,
IBM Software Group – Rational

https://www.compaid.co.in/Articles/DevOps-Value-and-Cost-Savings

The Engineering Cycle

- The Engineering Cycle
  - A set of logical steps that are needed whenever we build something or deliver a service
    - *Each step builds on the previous one*
    - *How we apply this cycle is a “process type”*
- Waterfall process types
  - Completes each step in the process fully before moving on to the next step
    - *Also called a “predictive” process because given the full set of requirements and technical constraints, we can accurately predict what the final product will should be*
    - *Common in engineering systems and high risk systems like nuclear reactor control software or airline navigation software*
    - *Or where requirements and technology don’t change over the lifetime of the project, like building a bridge for example*



In Real Life

- For a lot of application areas, the waterfall doesn’t work
  - There is often too much uncertainty and variation at each stage of the engineering cycle
  - Results in a lot of re-work as we respond to variance, unplanned changes or newly discovered facts in any of the engineering cycle stages
- These problems are not just software related
  - They are common across a variety of industries
- Various alternatives to the waterfall approach were experimented with in multiple industries
  - Collectively, these are referred to as adaptive methodologies
  - They continuously adapt the engineering process and stages to accommodate uncertainty and variance during the project
    - *Essentially incorporating risk management into a production process*
  - Most notable of these is Scrum

Scrum

- Scrum was not originally designed for software development
  - It was originally developed for industrial manufacturing in the 1950s-1980s
  - Major influences on Scrum were “Lean manufacturing” and the Toyota production system
- In the 1980s, there was a major crisis in software development
  - It was being developed in a big-bang approach using waterfall methodologies
  - Siloed teams (design, development, testing) with one-time hand offs of artifacts
- However, this problem had been identified decades earlier
  - For example, NATO software engineering conference in 1968 was held to address the high failure rate of these big-bang projects
    - *The conclusion of this and other similar conferences was that an incremental and iterative approach, like Kaizen (continuous improvement) and lean approaches to product development were needed*
    - *They were looking to the adaptive methodologies used in manufacturing as a possible model for software development*

Scrum

- In the 1980s and 1990s
  - Companies experimented with using what they called adaptive software development
  - IBM, DuPont, and others experimented with iterative prototyping and empirical process control like the Spiral methodology
- Characterized by
  - Use of successive prototypes to get feedback on requirements, design and performance
  - Short iterations (one month or less)
  - Cross-functional teams
  - Daily meetings for synchronization
  - A prioritized feature list

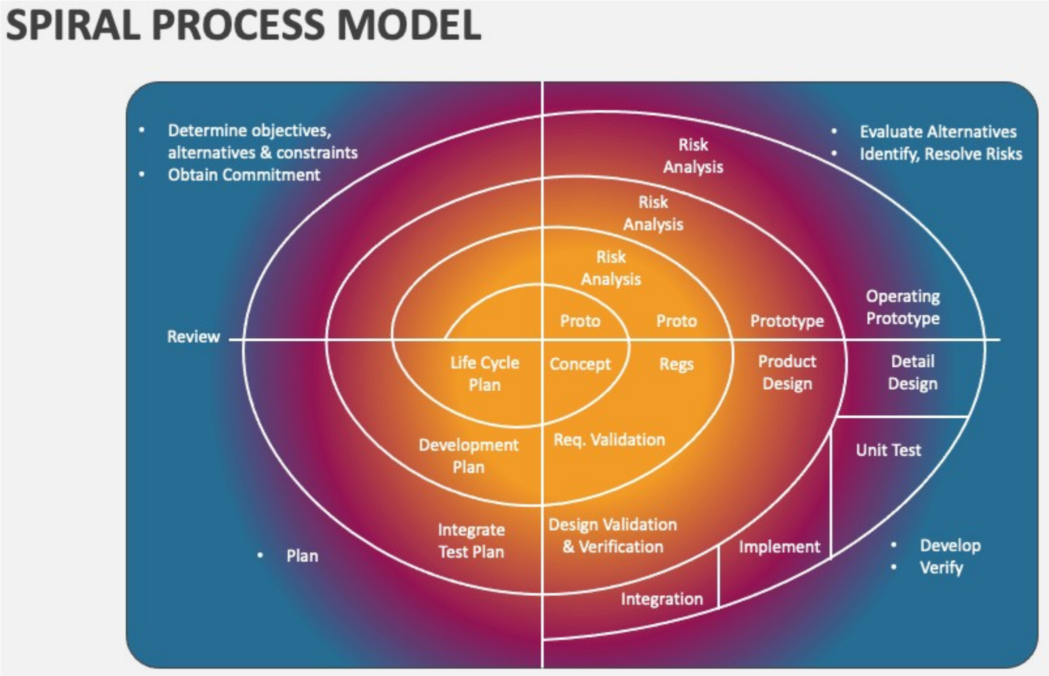
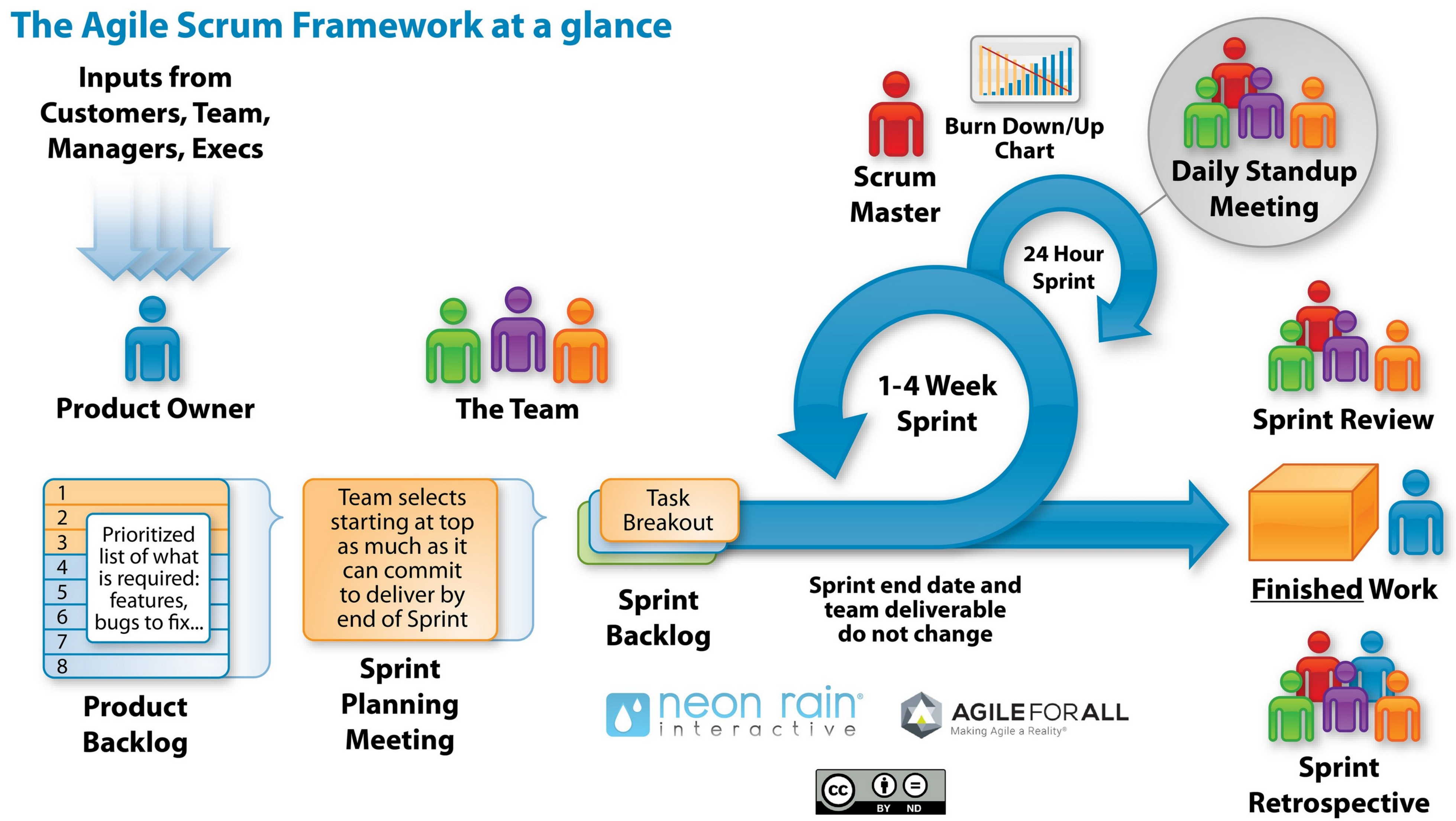


Image Credit: https://www.collidu.com/presentation-spiral-process-model

Frameworks and Methodologies

- A common mistake is to consider Scrum an Agile methodology
  - It is not a methodology, it is a process framework
  - Failure to have a methodology defined while using Scrum negates the value of using it
- Scrum is process-focused
  - Scrum focuses on how to organize the team and the work and manage the process
- Agile methodologies are practice-focused
  - Managing how the actual development work is done at the code and design levels
- The two complement each other
  - Scrum without a software engineering methodology risks low quality software being built
  - A software engineering methodology without Scrum risks lack of direction and effectiveness

Scrum at a Glance



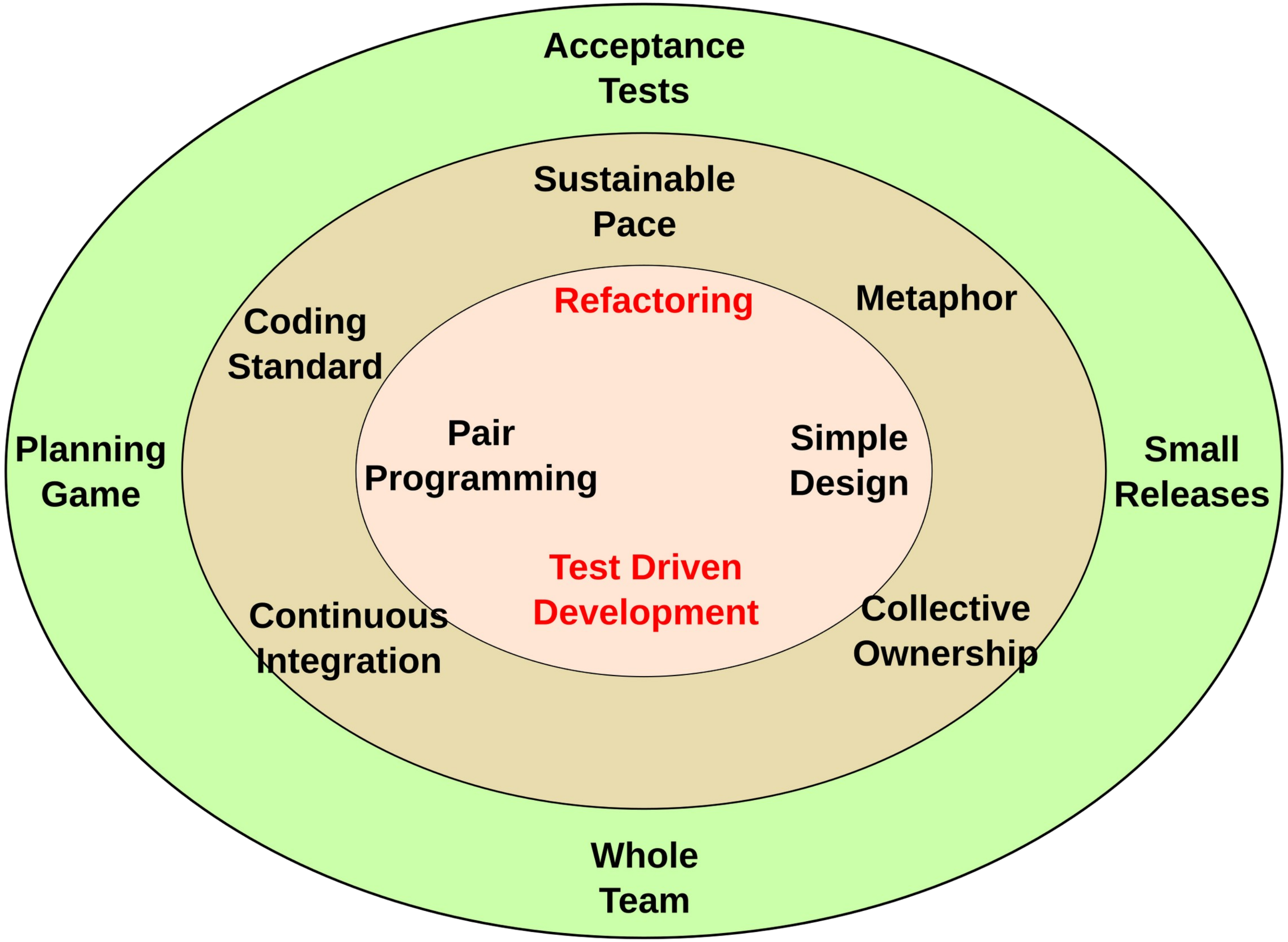
Scrum and Agile

- The term “Agile” was adopted in 2001
  - Defined by owners of adaptive methodologies that shared a similar approach to development
  - Derived many of their ideas from their use of Scrum ideas in their methodologies
    - *For example: Extreme Programming, Feature Driven Development*
- Many of the features of Scrum form the foundation of the Agile Manifesto and the Agile Principles
  - Note that Scrum is NOT an Agile methodology, but its concepts were shared among Agile methodologies and were highly influential
  - Specifically:
    - *Individuals and interactions*
    - *Working software prototypes*
    - *Customer collaboration and feedback loops*
    - *Responding to change in a planned systematic way*

Extreme Programming (XP)

- Example of an Agile methodology
- Developed in the late 1990s by Kent Beck
  - One of the original Agile methodologies
  - Designed to improve software quality and responsiveness to changing customer requirements through frequent releases, continuous feedback, and disciplined technical practices
- Core ideas
  - XP focuses on adaptability, collaboration, and technical excellence
  - Pushes core agile principles to the “extreme”
  - Ensuring that teams can deliver value rapidly and sustainably even in high-uncertainty environments
  - Very dependent on automation, typical of most Agile methodologies
    - *For example, automated unit testing, automated builds, integration testing and refactoring*

Extreme Programming (XP)



Extreme Programming (XP)

- Key characteristics:
  - Highly iterative and incremental: frequent releases and feedback loops
  - Human-centered: collaboration and trust are central
  - Quality-driven: testing, integration, and refactoring prevent defects
  - Adaptable: embraces change as part of the process, not a disruption
- But to meet these goals, automation is essential
  - Automated unit testing
  - Automated build management
  - Automated deployments
  - Automated feedback from prototypes

Three Drivers of CI/CD

- The need to develop and deploy large numbers of microservice components
  - These needed to be done via an automated development pipeline
  - Traditional app development wasn’t getting the job done
  - Mission critical nature of the components required built-in quality control and testing
- The need to automate a number of phases of Agile development
  - Working prototypes need be regularly produced with short turnaround times
  - Continuous testing during development to reduce rework
- Infrastructure as code
  - IaC code is now needed to define the environment for building application code
  - The same sort of development requirements as app code now apply to operations and IaC

Benefits of CI/CD

- Smaller code changes
  - Simpler (more atomic) and have fewer unintended consequences
- Mean time to resolution (MTTR) is shorter
  - Smaller code changes and quicker fault isolation
- Testability improves due to smaller, specific changes
  - These smaller changes allow more accurate positive and negative tests
- Elapsed time to detect and correct production issues is shorter
- The backlog of non-critical defects is lower
  - Defects are often fixed before other feature pressures arise
- The product improves rapidly through fast feature introduction and fast turn-around on feature changes

Benefits of CI/CD

- CI/CD product feature velocity is high
  - The high velocity improves the time spent investigating and patching defects
- Feature toggles and blue-green deployment strategies
  - Enable seamless targeted introduction of new production features
- Upgrades introduce smaller units of change and are less disruptive
- End-user involvement and feedback during continuous development leads to usability improvements
  - Can add new requirements based on customer's needs on a regular basis

Challenges for CI/CD

- Organization silos and corporate culture
  - Lack of communication between development, QA and operations
- Failure to automate testing or to do continuous testing
  - QA starts lagging behind development requiring rework to fix buggy code
- Legacy systems integration
  - Automated tools may not be available for legacy systems
  - E.g. Unit testing frameworks for COBOL code
- Complexity and size of applications
  - Trying to apply CI/CD to too big a “chunk” of development
  - Especially when introducing CI/CD improvements

Pipelines

- A pipeline is a series of automated steps that take a software component from coding all the way to the operational environment

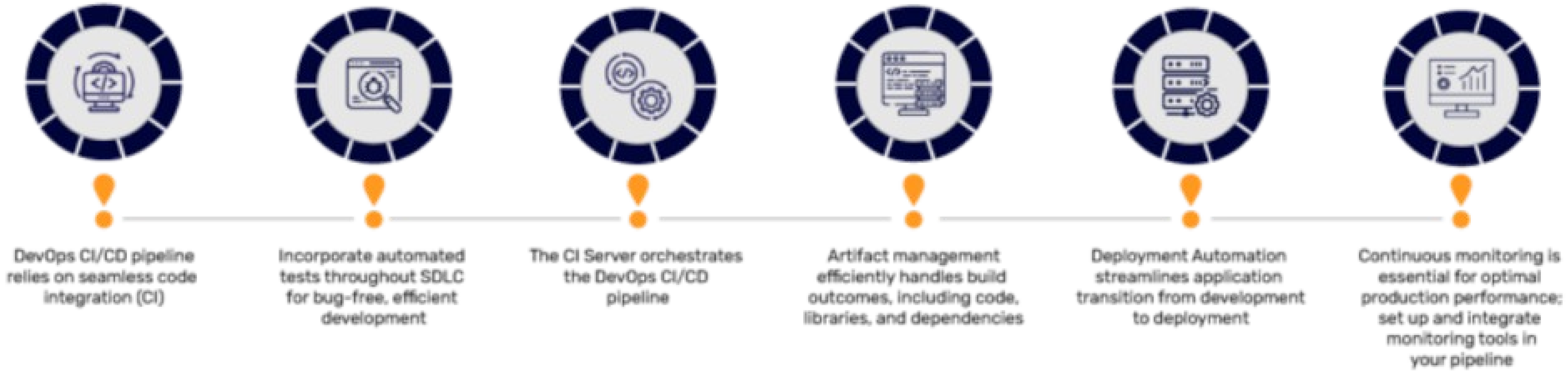


Image Credit: https://www.mindbowser.com/devops-ci-cd-pipeline-stages/

Automation Tools Drive Stages

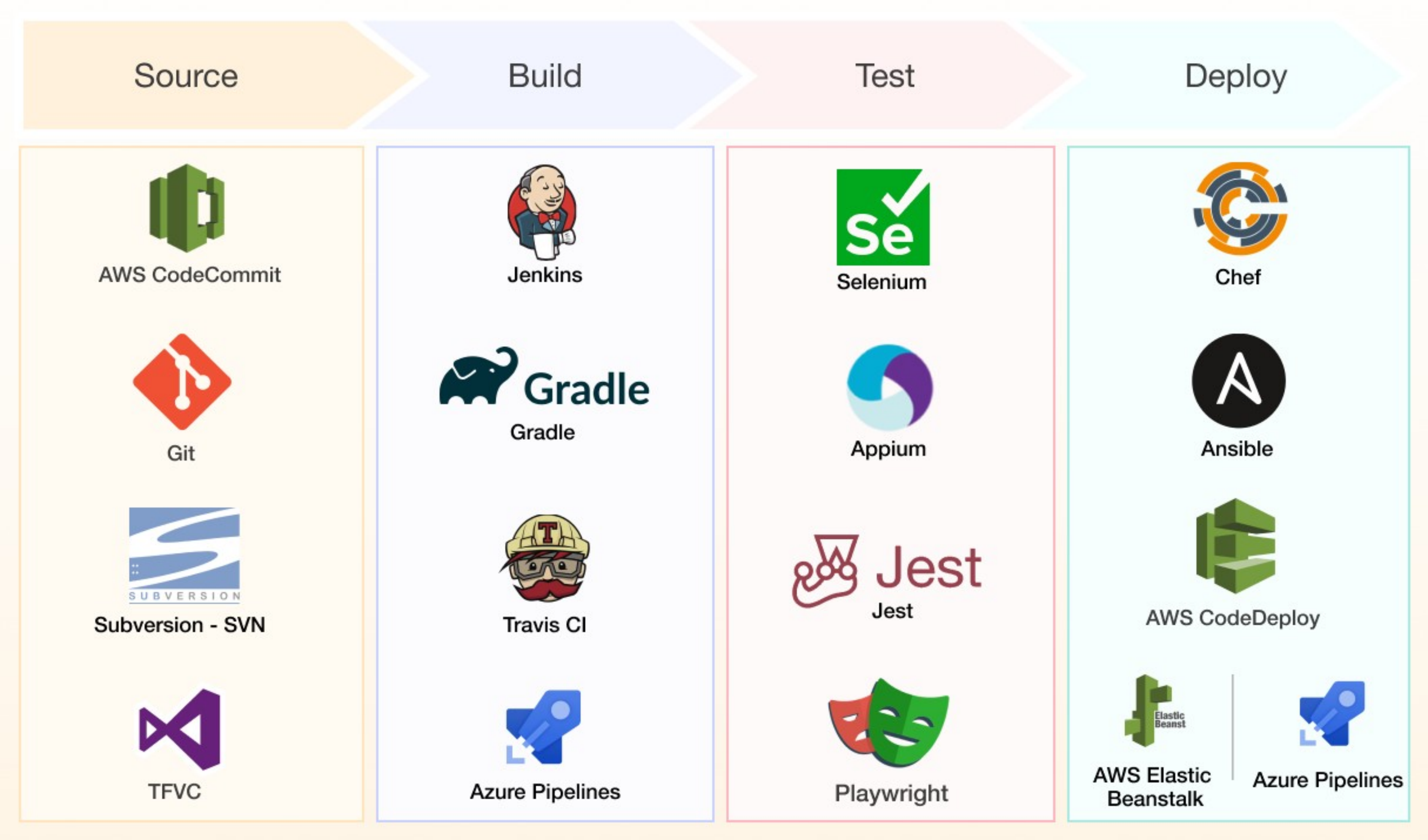
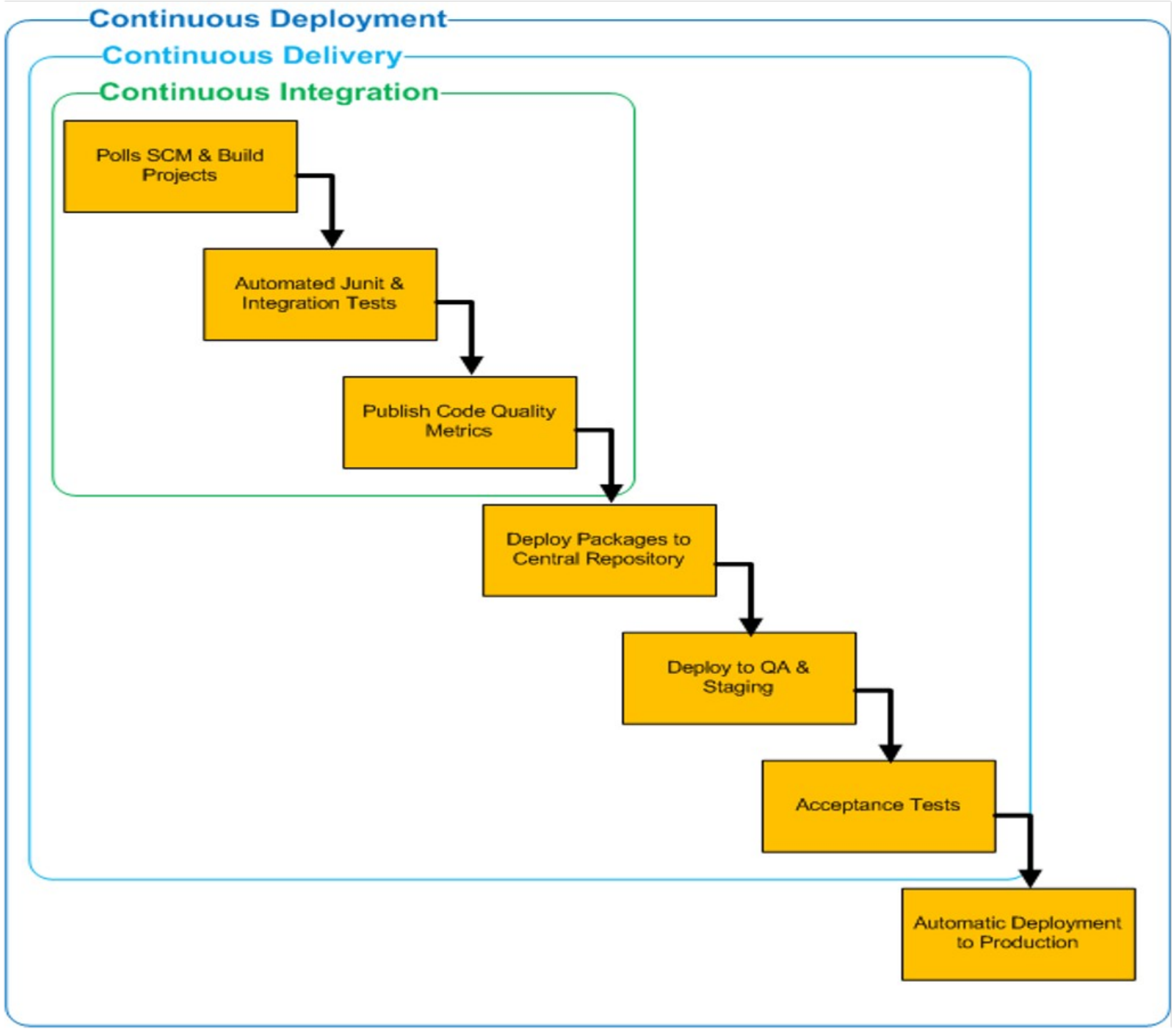


Image Credit: https://www.simform.com/blog/scalable-ci-cd-pipeline-examples/

Continuous Integration, Delivery and Deployment



Continuous Integration (CI)

- CI is the stage where members of a team integrate their work frequently
  - Usually each person integrates at least daily, leading to multiple builds per day
- Each integration is verified by an automated build
  - Automated builds that are successful trigger automated integration testing
    - *Intended to detect integration errors as quickly as possible*
    - *Goal is to merge and test the code continuously to catch issues early by automating the integration process*
- A CI project must have a reliable, repeatable, and automated build process involving no human intervention
  - CI Server (orchestration tool) is responsible for performing the integration tasks
  - Automatic unit testing, static analysis and failing fast are core to CI

Continuous Integration Practices

- Single source repository for all developers
- Build automation
  - Every change to the integration branch should trigger a new build
  - Keep the builds fast and trackable
  - Make the builds self-testing
- Test the builds in production-like environment
  - Keep all verified releases in artifacts repository and available to everyone
- Publish coding metrics

Continuous Delivery (CD)

- CD is a natural extension of CI
  - Every change to the system that has passed all the relevant automated tests should be ready to deploy in production
  - Team should be able to release any version “at the push of a button”
  - Keeps all verified releases in artifacts repository and available to everyone
- But the deployment into the production environment is not automatic
  - The goal of CD is to put business owners in the control of scheduling of the software releases
  - The decision to release is a governance decision, not a technical one
  - Users are notified the release is available but it is deployed only with the user’s approval

Continuous Deployment (also CD)

- Continuous Deployment adds automatic deployment to end users in the Continuous Delivery process
  - Continuous Deployment automatically deploys every successful build directly into production
  - Deploying the build to production as soon as it passes the automated and UAT tests
- Continuous Deployment is not appropriate for many business scenarios
  - Business Owners prefer more predictable release cycles as opposed to arbitrary deployments

CI/CD as a General Pipeline Pattern

- A pipeline is an automated, ordered set of stages that transform an input artifact into a validated, deployable output
- This idea shows up everywhere:
  - Software delivery (CI/CD)
  - Data engineering (ETL/ELT)
  - Machine learning (MLOps)
  - Infrastructure (IaC pipelines)

CI/CD Pipelines (Baseline Reference)

- Source → Build → Test → Package → Deploy → Monitor
- Example tools
  - GitHub Actions / GitLab CI / Jenkins
  - Maven / Gradle / npm
  - Docker / Helm
  - Kubernetes / VM / PaaS

CI/CD Pipelines (Baseline Reference)

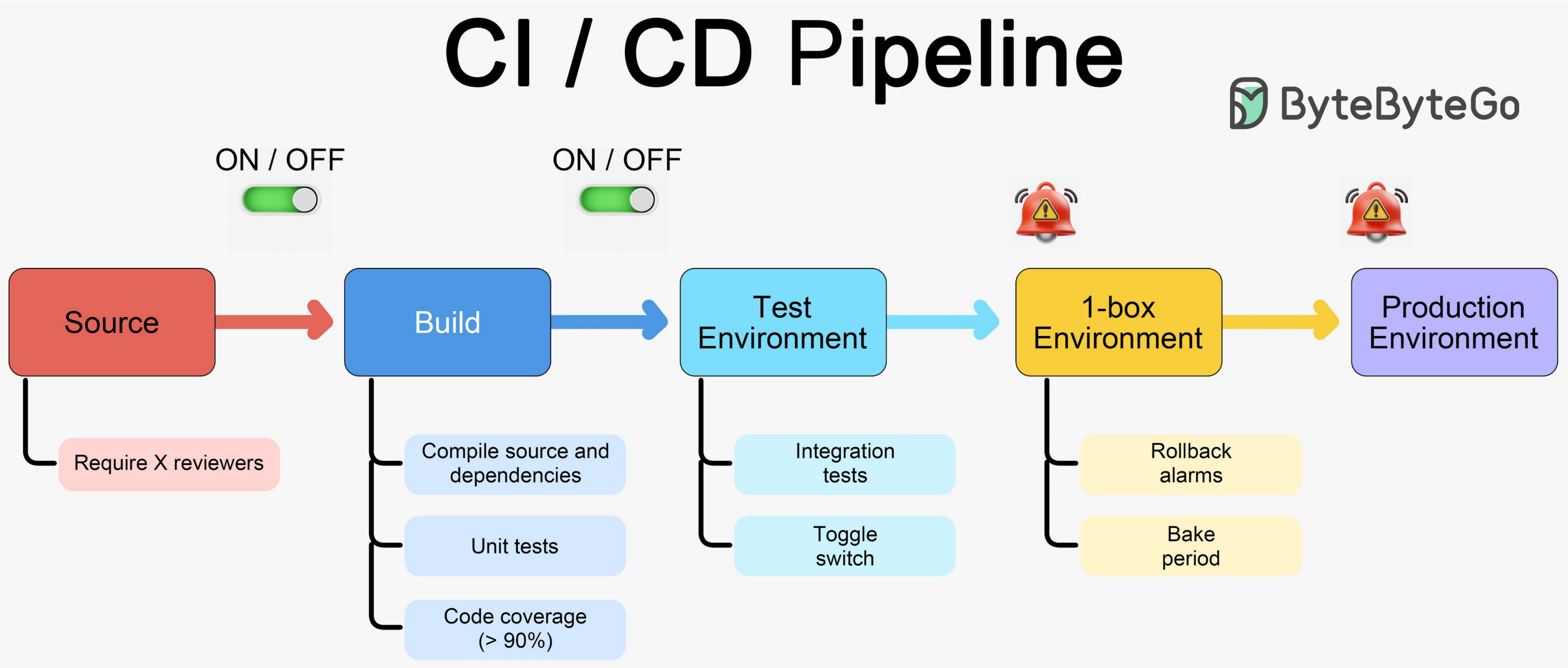


Image Credit: https://blog.bytebytego.com/p/a-crash-course-in-cicd

Data Pipelines (Data Engineering)

- Ingest → Validate → Transform → Load → Quality Check → Publish
  - Data pipelines apply the same CI/CD logic, but the “artifact” is data, not code
- Example tools
  - Source: Kafka topic / API / database
  - Transform: Spark / Flink / dbt
  - Load: Data warehouse (BigQuery, Snowflake)
  - Validation: Great Expectations
  - Orchestration: Airflow / Dagster
- CI/CD benefits
  - Versioned SQL and transformation logic
  - Automated schema validation
  - Test datasets before production loads
  - Promotion of pipelines from dev into prod

Data Pipelines (Data Engineering)

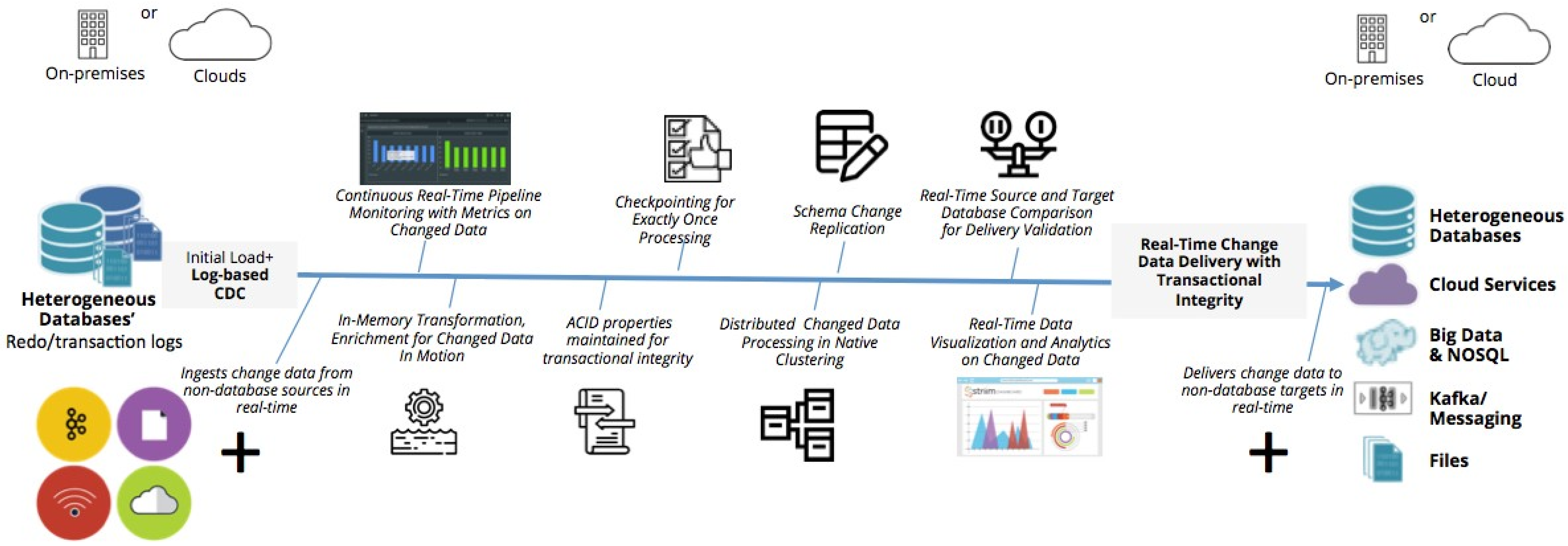


Image Credit: https://www.striim.com/blog/what-is-a-data-pipeline-and-must-have-features-of-modern-data-pipelines/

MLOps Pipelines (Machine Learning)

- Data → Feature Engineering → Train → Evaluate → Register → Deploy → Monitor
  - MLOps pipelines extend CI/CD to building, training and developing machine learning models
- Example tools
  - Training: TensorFlow / PyTorch
  - Tracking: MLflow
  - Model Registry: MLflow / SageMaker
  - Deployment: REST endpoint / batch job
  - Monitoring: Drift detection, accuracy decay

MLOps Pipelines (Machine Learning)

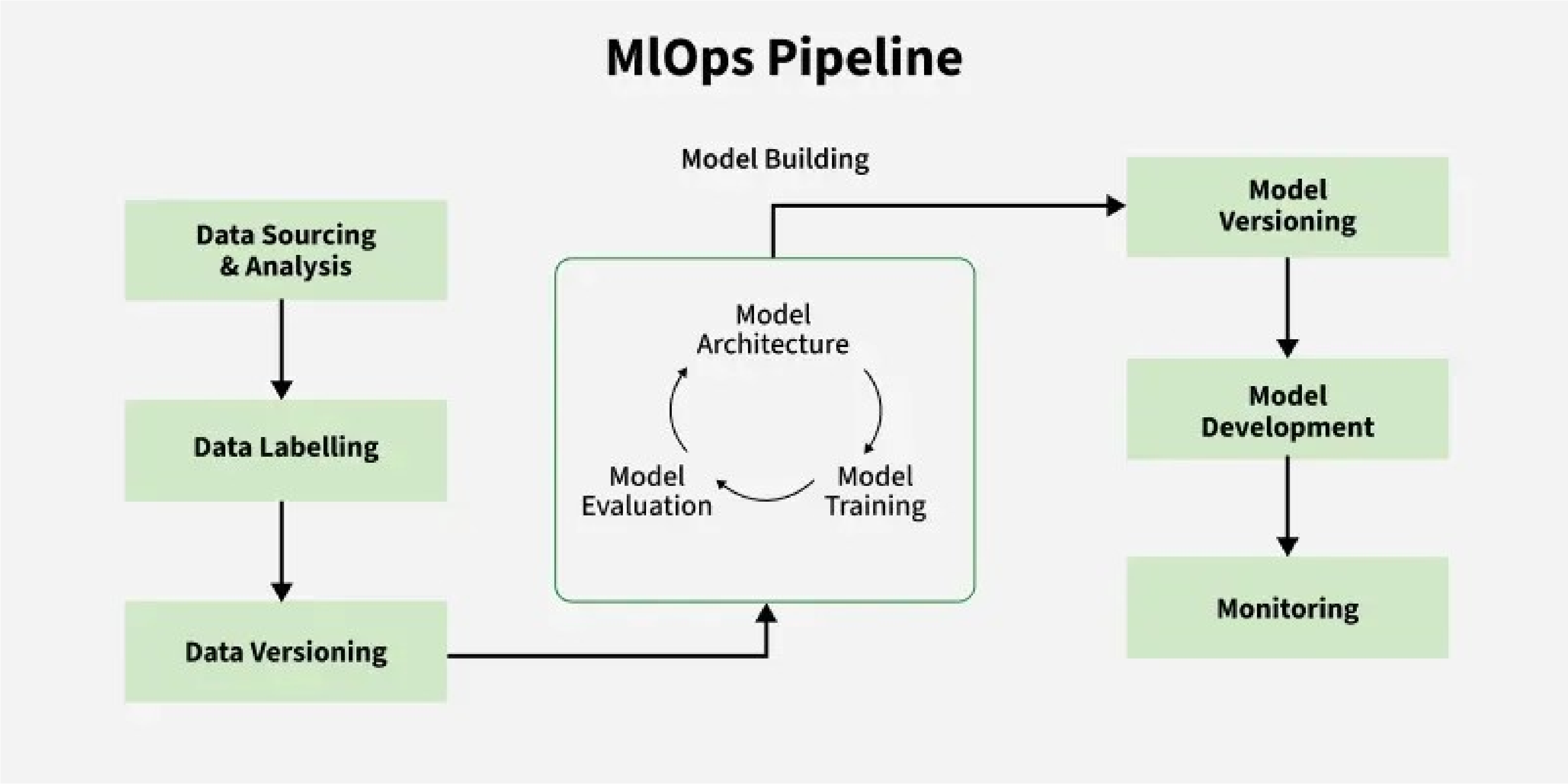


Image Credit: https://www.geeksforgeeks.org/machine-learning/mlops-pipeline-implementing-efficient-machine-learning-operations/

Infrastructure Pipelines (IaC)

- Define → Validate → Plan → Apply → Verify
  - Infrastructure is treated as a versioned artifact, validated and promoted like code
- Example tools
  - Terraform / Pulumi
  - Automated security scans
  - Environment promotion (dev → test → prod)

Infrastructure Pipelines (IaC)

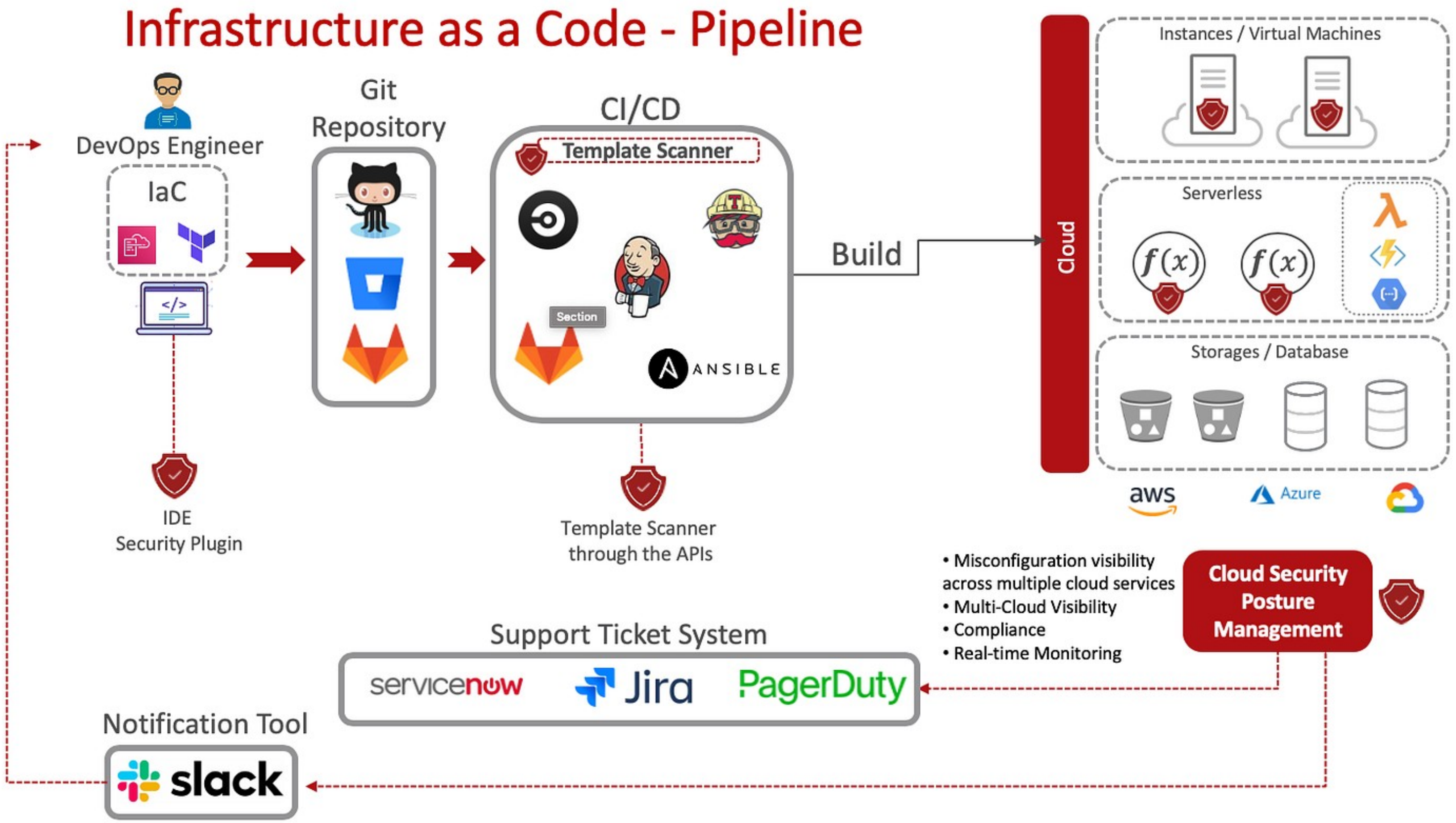


Image Credit: https://medium.com/swlh/putting-security-into-the-iac-pipeline-4de98f88ad24

Tool Chains

- Generally some form of repository tool is used for the CI environment
  - Usually git based like GitHub or GitLab
- Various packaging and build tools are used throughout the process
  - These are generally dependent on the programming development environments
  - For Java, we usually see Maven and Gradle for example
- Automated testing tools are used throughout the pipeline
  - Unit testing, Cucumber/Behave integration testing
  - Code quality tools like SonarQube
- The whole process is managed by an orchestration tool
  - Commonly Jenkins is used as a standalone tool
  - GitLab and GitHub have orchestration capabilities that are often used

Questions