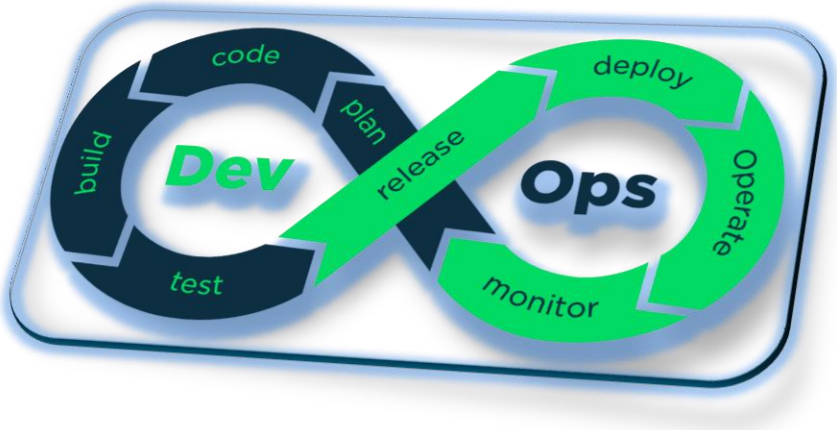
UNIVERSITY OF MPUMALANGA




# Advanced Application Development (ADPV400)

# Scope

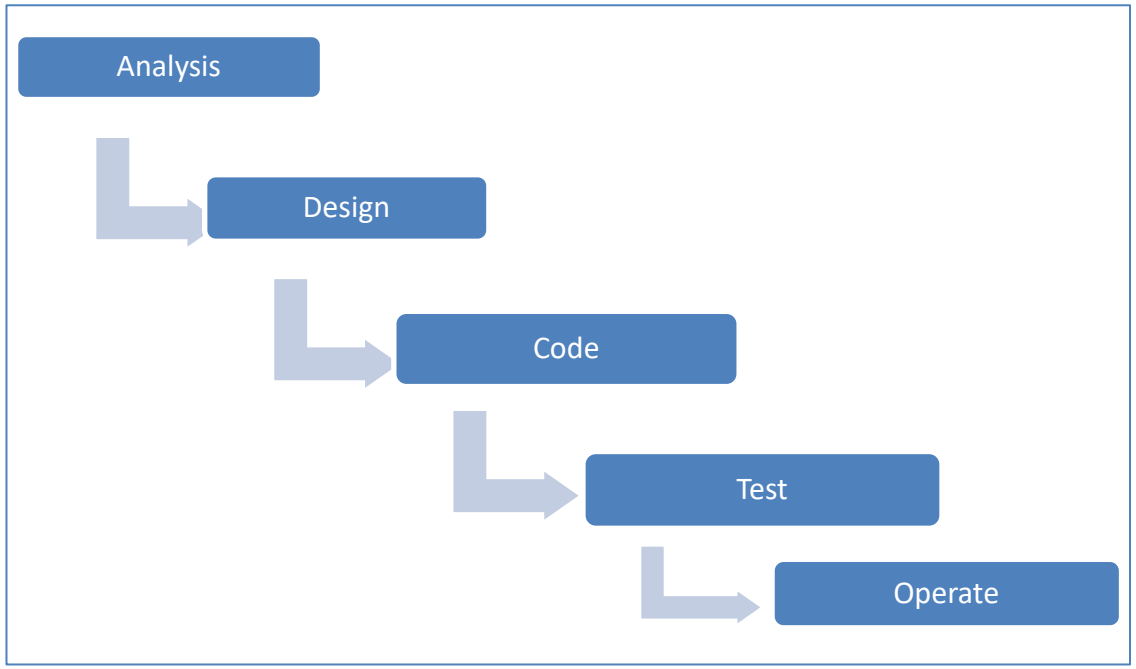
- Introducing DevOps
- How fast is fast?
- DevOps' Support for Agile Cycles
- The DevOps Process and Continuous Delivery vs ITIL
- Overview of DevOps process & Continuous Delivery

# What is DevOps?

# Introduction of DevOps

- **What is DevOps?**
  - The word “DevOps” consist of two words “Development” and “Operations”.
  - It is a practice that enforce collaborations between different discipline of software development.
  - It is a set of practices that enforce the integration of **people**, **process** and **product** to continuously deliver value to end users in a **minimal time**.
  - DevOps helps in optimizing a software delivery pipeline
- **Why DevOps**?
  - DevOps reduces the development lifecyle
  - Also ensures quick market penetration
  - Reduces deployment failures
  - Also improve communication and collaboration in software development

# Introduction of DevOps




- DevOps is a blend of software development and;
  - Software developers
  - Testers and
  - Quality assurance personnel
- Operations represents experts who put software into production and manage the production infrastructure, including
  - System administrators
  - Database administrators
  - Network technicians

# Introduction of DevOps

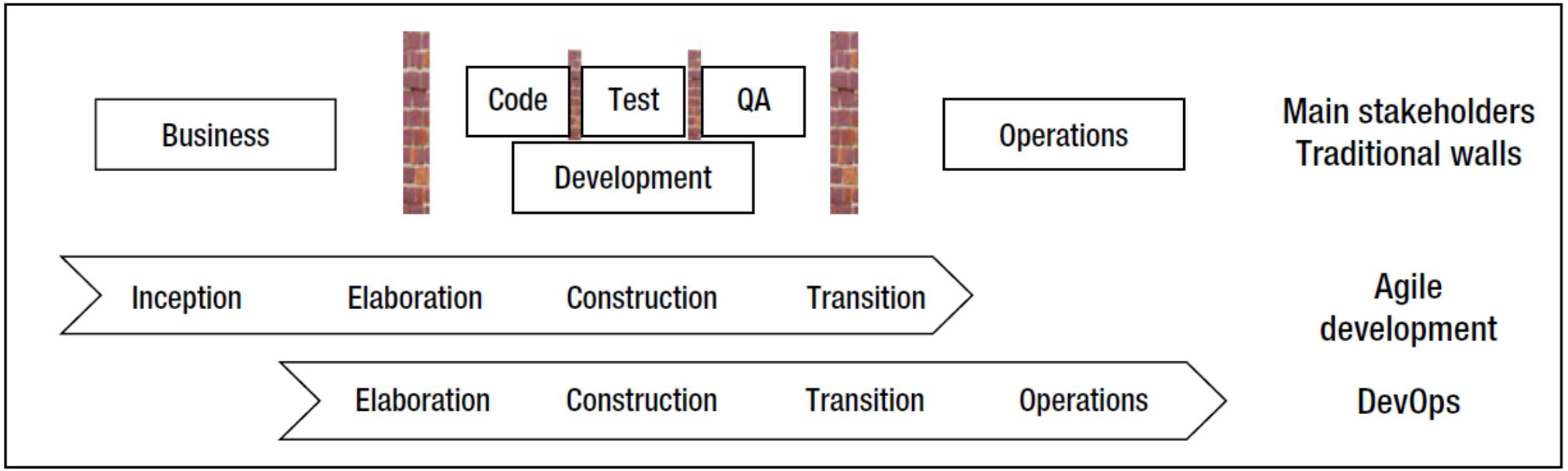
- **DevOps** has its roots in Agile software development principles.
- **Agile Manifesto:** *Individuals and interactions over processes and tools*
  *Working software over comprehensive documentation*
  *Customer collaboration over contract negotiation*
  *Responding to change over following a plan*

  That is, while there is value in the items on the right, we value the items on the left more.
- DevOps relates to the first principle, "Individuals and interactions over processes and tools."

# Introduction of DevOps

- **DevOps** tends to emphasize that interactions between individuals are very important, and;
  - That technology might possibly assist in making these interactions happen.
- **Example:** Choice of systems to be used for reporting bugs
  - Often, development teams and quality assurance teams use different systems
  - Operations team might have its own system too
- **A DevOps Engineer**, on the other hand, will recognize all three systems as being workflow systems with similar properties.
- Another core goal of **DevOps** is automation and Continuous Delivery
  - I.e., automating repetitive and tedious tasks leaves more time for human interaction, where true value can be created

# The Big Picture of DevOps

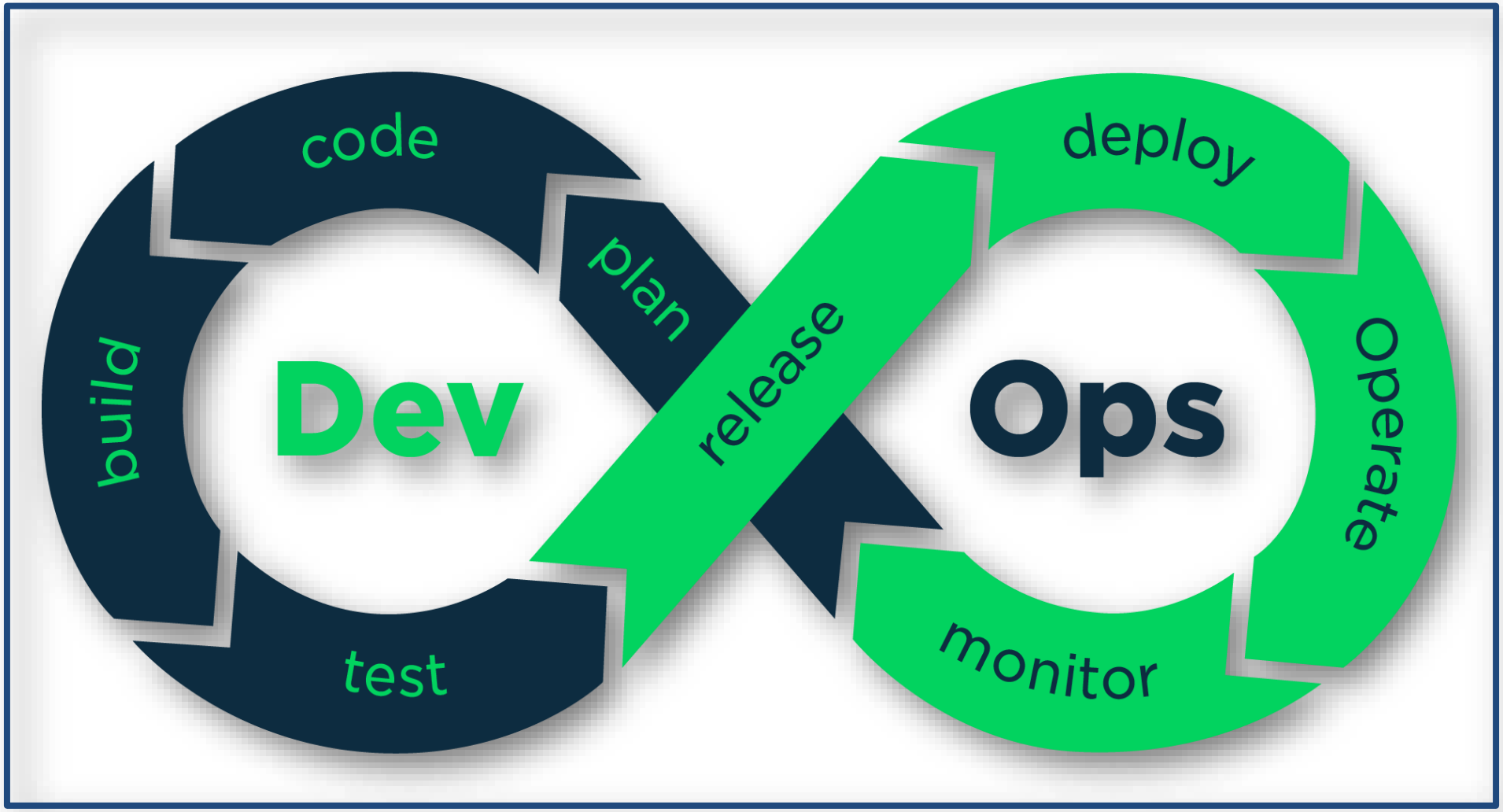


# How Fat is Fast?

# What does the practise of DevOps implies?

- It implies that DevOps is *goal oriented, which is reducing the time* and *increasing business value* between a commit & deployment.
  - DevOps engineers work on making enterprise processes faster, more efficient and more reliable.
  - Repetitive manual labour, which is error prone, is removed whenever possible.
- Always keep track of delivering increased business value, e.g.,
  - Increased communication between roles in the organization.
  - Delivering incremental improvements of code to the test environments quickly and efficiently.
    - In this way, involved stake holders, such as product owners and quality assurance teams, can follow the progress of the development process.

# DevOps Lifecycle



# DevOps toolchain

- Followers of DevOps practices often use certain DevOps-friendly tools.
- **Goal** – further streamline, shorten and automate the various stages of the software delivery workflow or pipeline.
- The tools also promote core DevOps principles of automation, collaboration and integration between development and operations teams.
- The following shows a sample of tools used at various DevOps lifecycle stages:
  - **Plan**: This phase helps define business value and requirements. Sample tools include Jira or Git to help track known issues and perform project management.
  - **Code**: This phase involves software design and the creation of software code. Sample tools include GitHub, GitLab, Bitbucket, or Stash.

# DevOps toolchain

- **Build**: In this phase, software builds and versions are managed and use automated tools to help compile and package code for future release to production. Sample tools include Docker, Puppet, Gradle or Maven.
- **Test**: This phase involves continuous testing to ensure optimal code quality. Sample tools include JUnit, Codeception, Selenium, Vagrant or BlazeMeter.
- **Deploy**: This phase can include tools that help manage, coordinate, schedule and automate product releases into production. Sample tools include Puppet, Jenkins, Kubernetes, OpenShift, OpenStack, Docker or Jira.
- **Operate**: This phase manages software during production. Sample tools include Puppet, PowerShell, Chef, Salt or Otter.
- **Monitor**: This phase involves identifying and collecting information about issues from a specific software release in production. Sample tools include New Relic, Datadog, Grafana, Wireshark, Splunk, Nagios, or Slack.

# DevOps Practices

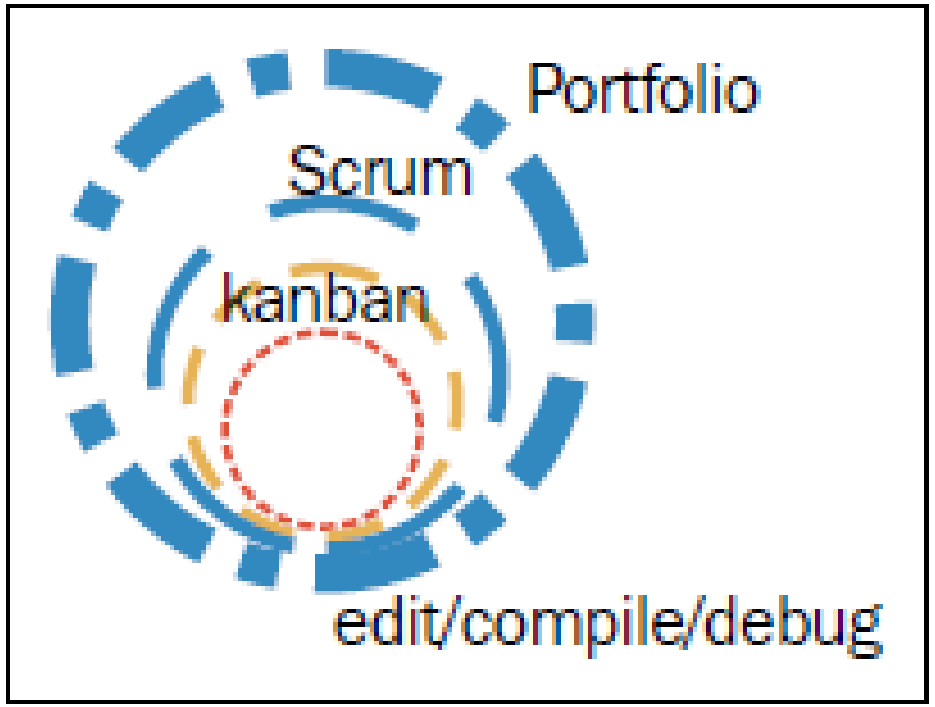
- **Continuous development** – This practice spans the planning and coding phases of the DevOps lifecycle. Version-control mechanisms might be involved.
- **Continuous testing** – This practice incorporates automated, prescheduled, continued code tests as application code is being written or updated. Such tests can speed the delivery of code to production.
- **Continuous integration (CI)** – Involves regularly merging code changes into a shared repository, followed by automated builds and tests to detect integration errors early.
- **Continuous delivery** – This practice automates the delivery of code changes, after testing, to a preproduction or staging environment.

# DevOps Practices

- **Continuous deployment (CD) –** This practice automates the release of new or changed code into production.
- **Continuous monitoring –** This practice involves ongoing monitoring of both the code in operation and the underlying infrastructure that supports it. A feedback loop that reports on bugs or issues then makes its way back to development.
- **Infrastructure as code –** This practice can be used during various DevOps phases to automate the provisioning of infrastructure required for a software release.

# DevOps' Support for Agile Cycles

# The Agile Wheel of Wheels




The Agile wheel of wheels

- DevOps should be able to support the following full delivery cycles of agile development:
  - Scrum cycle is 2 to 4 weeks
  - Kanban is 24 hours cycle
  - Program increments longer than 4 weeks

# How DevOps Benefits Agile Cycles

- DevOps' deployment systems make the deliveries at the end of Scrum cycles **faster** and **more efficient**.
  - It can take several days to deploy when done by hand.
- The Kanban cycle is 24 hours, deployment cycle should be shorter
  - A well-designed DevOps Continuous Delivery pipeline can take few hour or minutes based on the change requirement.

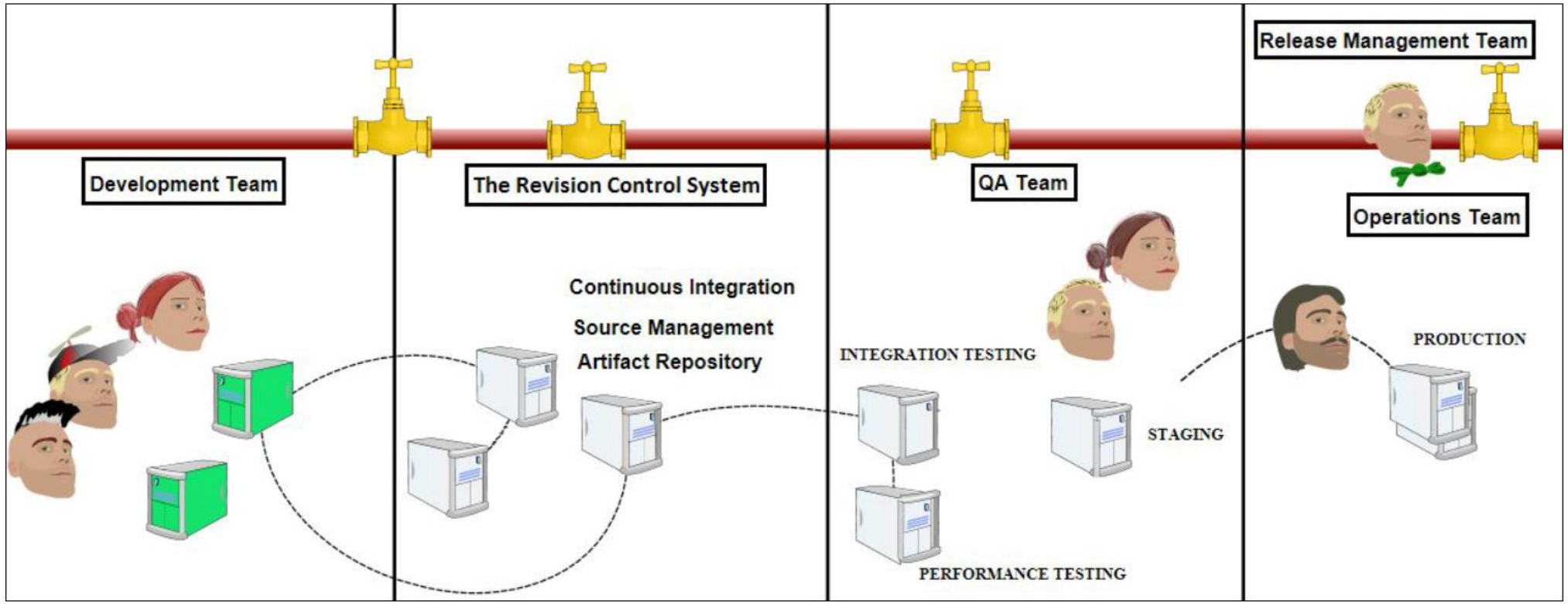
# The DevOps Process and Continuous Delivery vs ITIL

# DevOps and ITIL

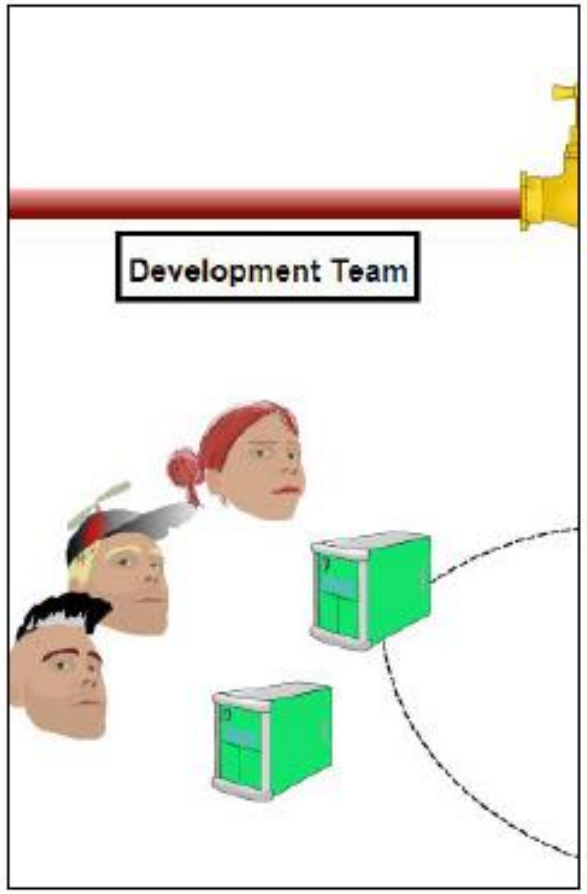
- DevOps fits well together with many frameworks for Agile or Lean enterprises, e.g., Scaled Agile Framework
- Information Technology Infrastructure Library (ITIL) is a large framework that formalizes many aspects of the software life cycle.
- DevOps and Continuous Delivery of changesets to production – be small and happen often.
  - ITIL would appear to hold the opposite view - this is not really true.
- Legacy systems are quite often monolithic - ITIL is needed to manage the complex changes associated with such systems.
- Note that many of the practices described in ITIL translate directly into corresponding DevOps practices.
  - E.g., Configuration management system, Configuration management database

# Overview of DevOps process & Continuous Delivery

- DevOps is used for large and complex processes in large context
- Example of continuous delivery pipeline in a large organization:

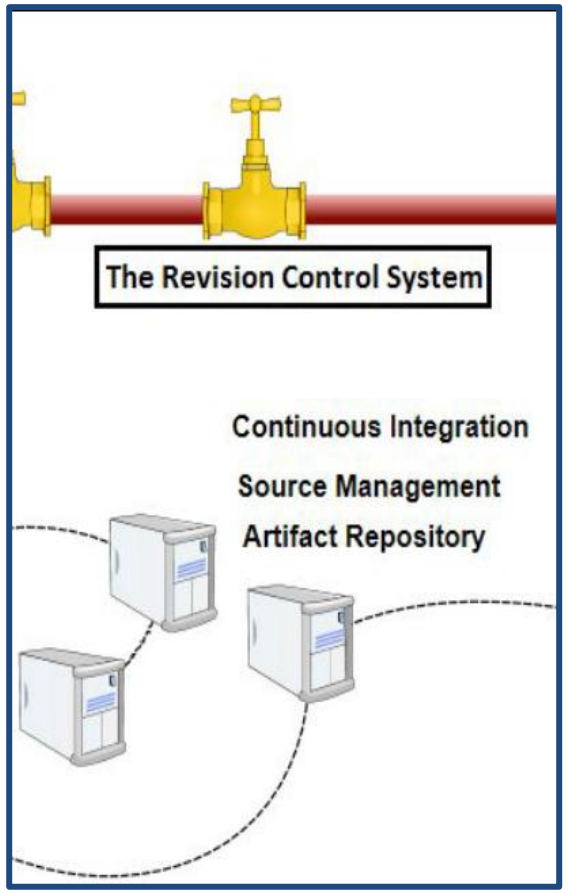


# The Developers



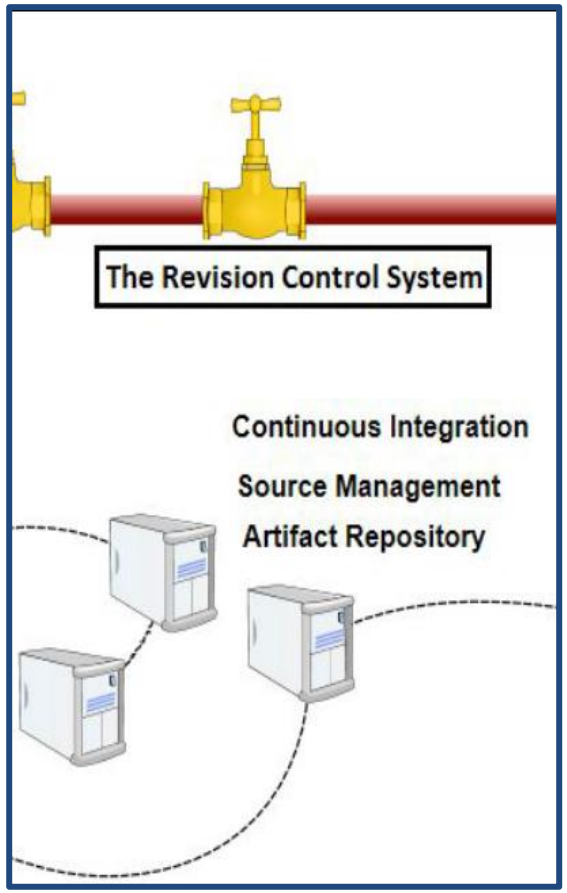

- Developers work on their workstations.
  - They develop code with many tools to be efficient.
- Use their workstations as a production line
  - Common to simulate or mock the parts of the production environments that are hard to replicate, e.g., external payment systems or phone hardware
- Developers also need some way to deploy their code in a production-like way
  - E.g., A virtual machine running on the developer's machine, a cloud instance running on AWS, or a Docker container

# The Revision Control System (RCS)



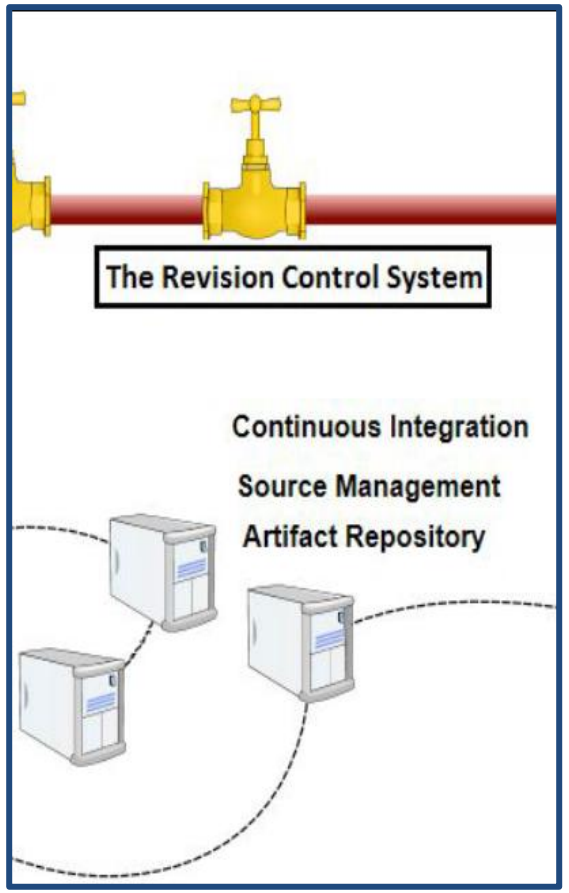

- The RCS is the heat of the development environment
- Stores code for an organization's software products
  - Common to also store infrastructure configuration files
  - Hardware designs may also be stored here.
- Many organization use Git for this part.

# Built Servers



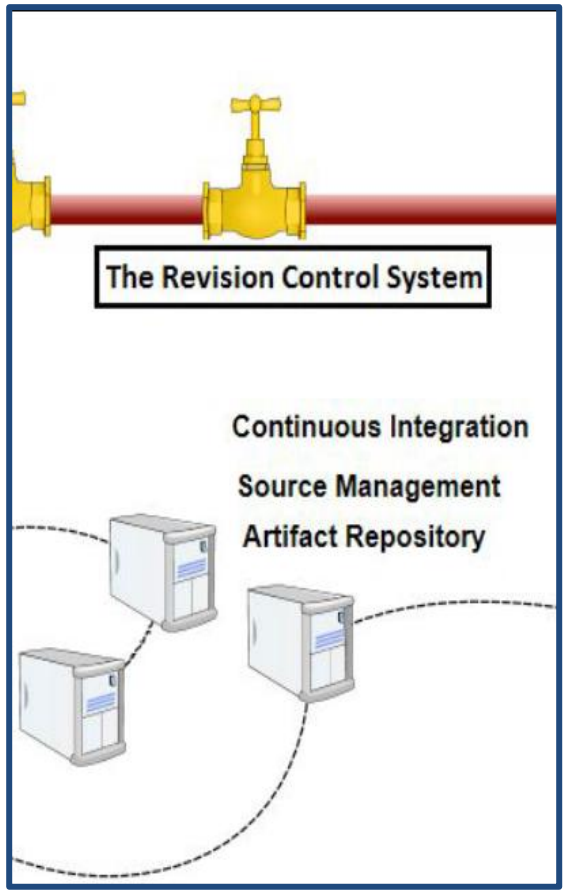

- The build server is responsible for building the source code at regular intervals or on different triggers.
- In most cases, the build server is automated to listen to changes made in the RCS
- When a change is noticed, the build server updates its local copy of the source from the revision control system
  - *This is followed by the source being built and tested for quality of changes*
  - This is what is called **Continuous Integration (CI)**.
  - *Jenkins will be explored, but there are many competitors.*
    - *Such as Buddy, Final Builder, CruiseControl, integrity, GoCD, UrbanCode (IBM)*

# The Artifact Repository



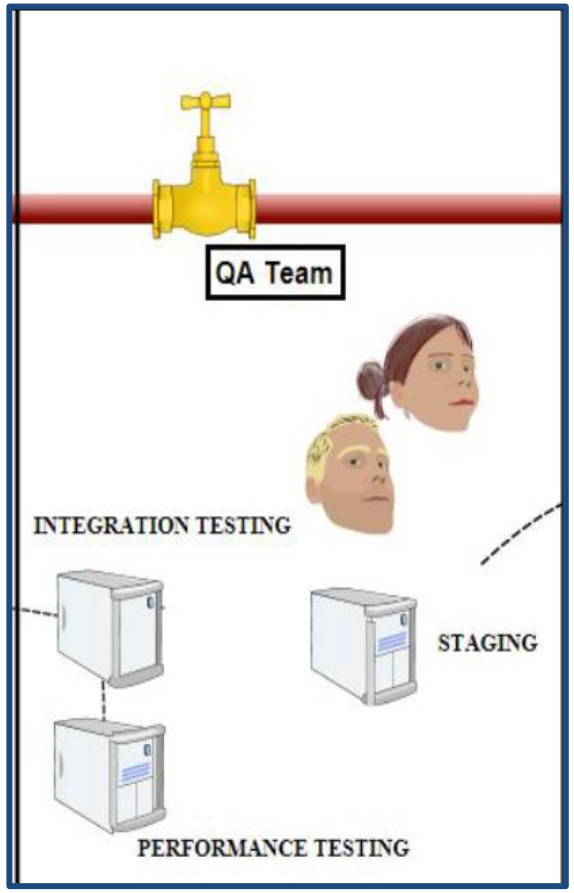

- Useful to store compiled binary artifacts after the build server has verified the quality of the code and compiled it into deliverables.
- Artifacts are accessed over the web as files
- For Java development, the most common choice is Sonatype Nexus.
  - Another option is Amazon S3

# Package Managers




- Package manager refers to commands that are employed to fetch and install the binary files.
- These package managers are useful for ensuring that installation & upgrading is made simple.

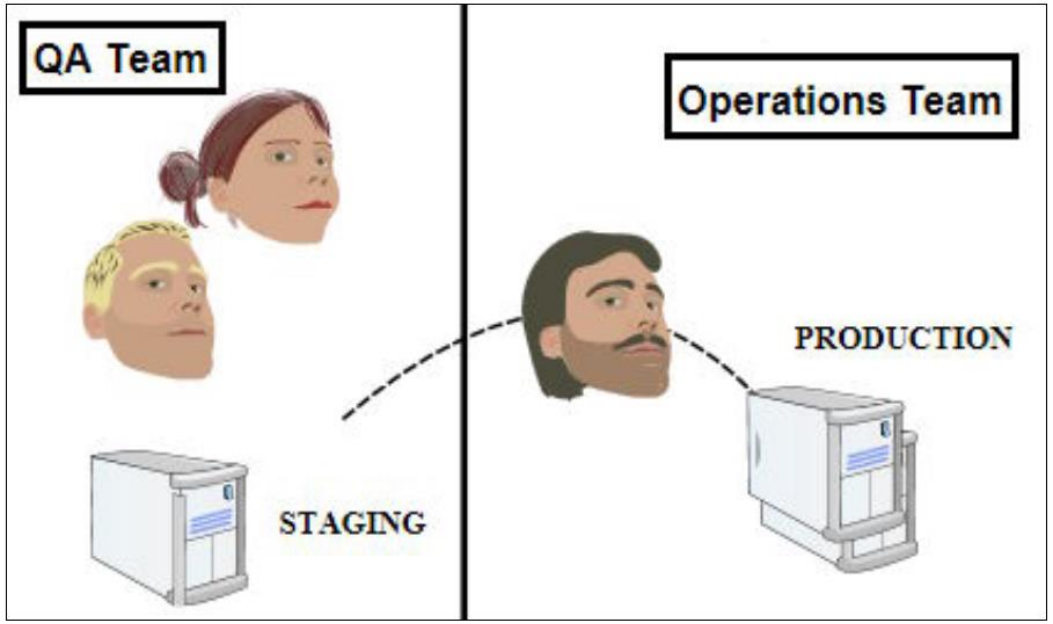
# Test Environments



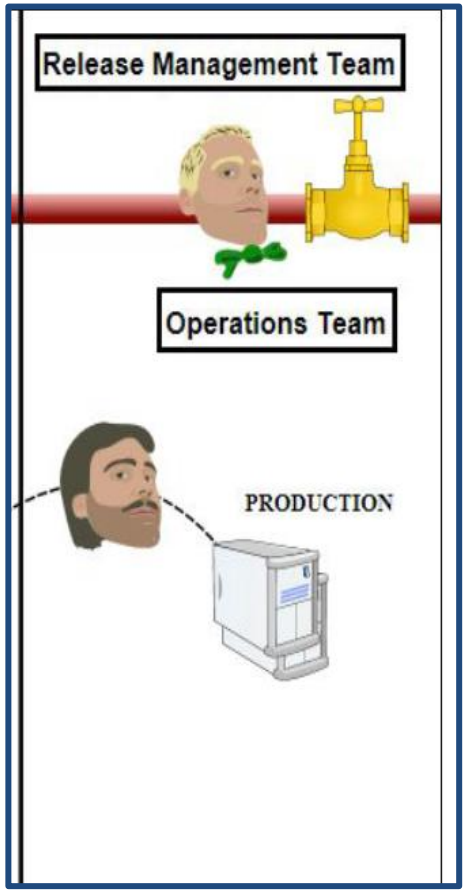

- The build server stores the artifacts in the binary repository.
  - They are then installed into the test environments.
- The figure shows the test systems in greater detail.
- Test environments should be production-like as is possible.
  - It is desirable that the binary repositories be installed and configured with the same methods to be used for installation in production servers

# Test Environment
# Staging/Production

- Staging environments are the last line of test environments.
  - They are interchangeable with production environments.
  - The new releases are installed on the staging servers, checked and swapped with the old production servers.
    - This is sometimes called the blue-green deployment strategy
- The figure from the larger Continuous Delivery image shows the final systems and roles involved.



# Release Management




- It is difficult to achieve an automated release process
  - This is because it is hard to reach the level of test automation needed in order to have complete confidence in automated deploys.
  - The pace of business growth does not always align with the pace of technical progress.
- Therefore, it is necessary to enable human intervention in the release process.
- A faucet is used in the figure to symbolize human interaction—in this case, by a dedicated release manager.

# Wrapping up – a complete example

- The development team has been given the responsibility to develop a change to the organization's system. The change revolves around adding new roles to the authentication system.
  - This seemingly simple task is hard in reality because many different systems will be affected by the change.
- It is decided that the change will be broken down into several smaller changes for simplicity reasons
- The first change, the addition of a new role to the authentication system, is developed locally on developer machines and given best-effort local testing.
- To really know if it works, the developer needs access to systems not available in his or her local environment; in this case, a Lightweight directory access protocol (LDAP) server containing user information and roles.
- The developer checks in the change to the organization's revision control system, a Git repository.

# Wrapping up – a complete example

- The build server picks up the change and initiates the build process. After unit testing, the change is deemed fit enough to be deployed to the binary repository, which is a Nexus installation.
- The configuration management system, Puppet, notices that there is a new version of the authentication component available. The integration test server is described as requiring the latest version to be installed, so Puppet goes ahead and installs the new component.
- The installation of the new component now triggers automated regression tests. When these have been finished successfully, manual tests by the quality assurance team commence.
- The quality assurance team gives the change its seal of approval. The change moves on to the staging server, where final acceptance testing commences.
- After the acceptance test phase is completed, the staging server is swapped into production, and the production server becomes the new staging server. This last step is managed by the organization's load-balancing server.

The process is then repeated as needed.