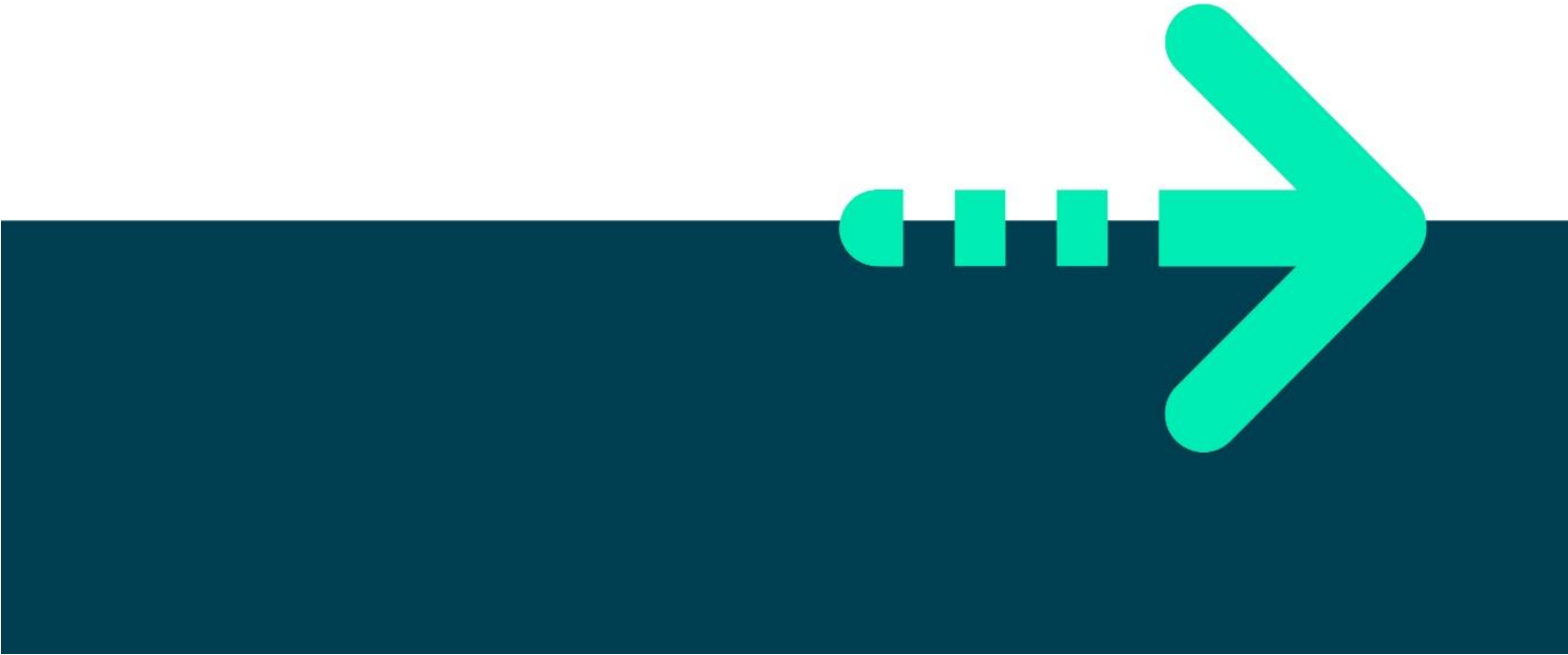
Dev Ops Culture & Methodologies

Module 1 - Introduction to DevOps

Learner Guide



CONTENTS

What is agile?

In a nutshell, agile is the most common software development method; it's a way of creating software. The simplest and oldest method is the waterfall method. However, the waterfall method had some disadvantages, which led most companies to adopt the new agile method. Both methods use the software development life cycle (SDLC) - but both implement it in different ways.

Software development life cycle

The SDLC is a sequence of necessary steps to create any piece of software. It's made up of six separate stages:

- Requirements
- Analysis
- Design
- Coding
- Testing
- Operations

Requirements

For a software product to be made, there must be a demand for it. The demand will come from stakeholders. We need to record all the requirements about what the end product will become.

Analysis

We need to analyse the requirements to create data models and business rules.

A data model is an entity which organises data. For instance, we could model a profile to hold data that includes a person's name, their age and their phone number.

A business rule is logic (usually about functionality of the product) which we need to write out. For example: 'user must be logged in and authenticated before they can add a friend' or 'mobile phone number can only be used for a single profile'.

Design

Once we have determined the functionality and models, we break the product down into programmable modules. Each module will have a specific purpose and will talk to other modules. By splitting functionality into smaller chunks, we can separate concerns and create reusable code.

Coding

This is where the development team take the designs and create a working piece of code. The code should be unit-tested and integration-tested to make sure that the delivered functionality is correct.

Testing

The testing team will take the working piece of code and perform extensive testing on it. This could be functional testing to find bugs, or it could be performance testing to see if the application is fit for purpose. Testing could also include backwards compatibility and system testing.

Operations

Once the application has been proved to work well in a staging environment, it is time for it to go "live". This live piece of functionality will continue to be maintained and patched in this stage until it is decommissioned.

Waterfall

The waterfall method (with respect to the SDLC) is quite straightforward. You analyse when the product will need to be completed, and work towards that goal by going through each step of the development life cycle, without going back. Like a waterfall, the water only travels in one direction.

The downside to this method is that, if the requirements were to change, you would have to restart the product from scratch. This makes the method inflexible.

Agile

Agile is a flexible software development method, and is one of the most common ways of developing software. It allows companies to adapt quickly to changes. This gives them a competitive edge, because they are able to deliver the most relevant product solutions.

A software development framework is said to be agile if it abides by the agile manifesto.

Agile Manifesto

The agile manifesto contains information on the four values and twelve principles which make a process agile.

Agile Values

Agile and all of its methodologies work against four core values in software development:

1. Individuals and Interactions over Processes and Tools. Ensure that we value the people building a product over standard processes and tools. The people will need to communicate and adapt to changes within a project.
2. Working Software over Comprehensive Documentation. Extensive documentation takes time away from software development. Keep documentation streamlined and simple to develop software at the rapid rate needed.
3. Customer Collaboration over Contract Negotiation. Work with the customer to make the most valuable product possible, rather than creating the bare minimum for a specification.
4. Responding to Change over Following a Plan. Allow customers to request changes, and embrace those changes within software development.

Note: whilst the values on the left are more valuable than the right, the right-sided values are still important!

Agile Principles

Agile also works with 12 supporting principles in conjunction with the four values.

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome requirement changes, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, as often as a couple of weeks to a couple of months. There is a preference for shorter timescales.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity; the art of maximising the amount of work *not* done is essential.

11. The best architectures, requirements, and designs emerge from self-organising teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly.

Empirical process control

Agile embraces empirical process control during a project.

Working with empirical process control means our work is based on facts, experience and evidence. We work using these, rather than using meticulous planning.

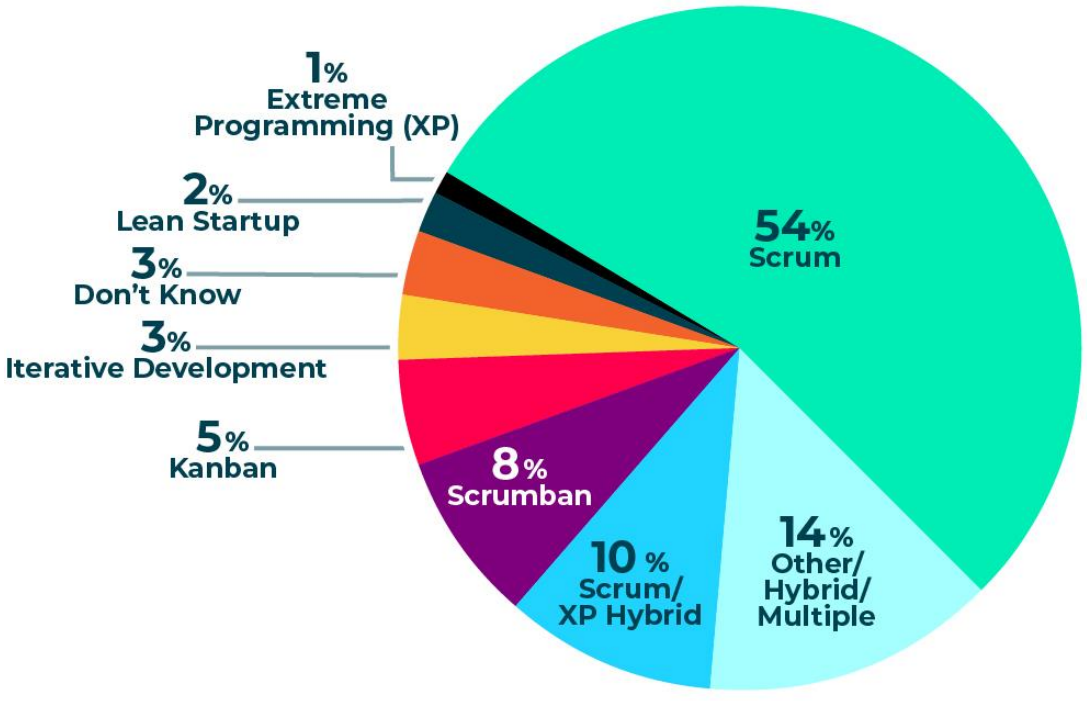
There are three main pillars to abide by for empirical process control:

- Transparency. Be transparent with any and all stakeholders
- Inspection. Transparency allows for better inspection of people, processes, tech and the product
- Adaptation. Based on findings in inspection, we adapt and continuously improve

Common agile development frameworks

There are many agile development frameworks, some of which even pre-date the agile manifesto but are currently considered 'agile' as they conform to the agile manifesto.

The State of Agile survey found that, of all frameworks, scrum was by far the most common:



13th State of Agile survey 2019

Agile Scrum

Scrum gets its name from rugby, where a team tries to go the distance as a unit, passing the ball back and forth.

It is:

- Lightweight
- Simple to understand
- Difficult to master

The scrum framework is repetitive and will produce an addition of code at the end of every sprint. The code will add functionality to what was there previously, therefore adding value. This additional code should aim to be 'shippable'; it shouldn't be half-broken, but ready to distribute.

What is a sprint?

A sprint is a time-boxed event in which work takes place.

This could be anywhere between one week and four weeks. Sprints will run one after another until the product no longer needs development.

During the sprint the developers will work from a 'sprint backlog', a small list of items which must be completed by the time the sprint is completed.

The 'sprint backlog' items are pulled from the 'product backlog', which is a list of all pieces of functionality which will go into the product eventually.

The pieces of functionality are prioritised within the team to determine what will be worked on during the next sprint.

Product backlog

The product backlog is a list of all features, enhancements and fixes to be made to the product in future releases.

The product owner handles the product backlog, including its content, availability, and ordering.

The development team will collaborate with the product owner, adding details where necessary.

Product backlog items include a description, order, estimate, and value.

Sprint backlog

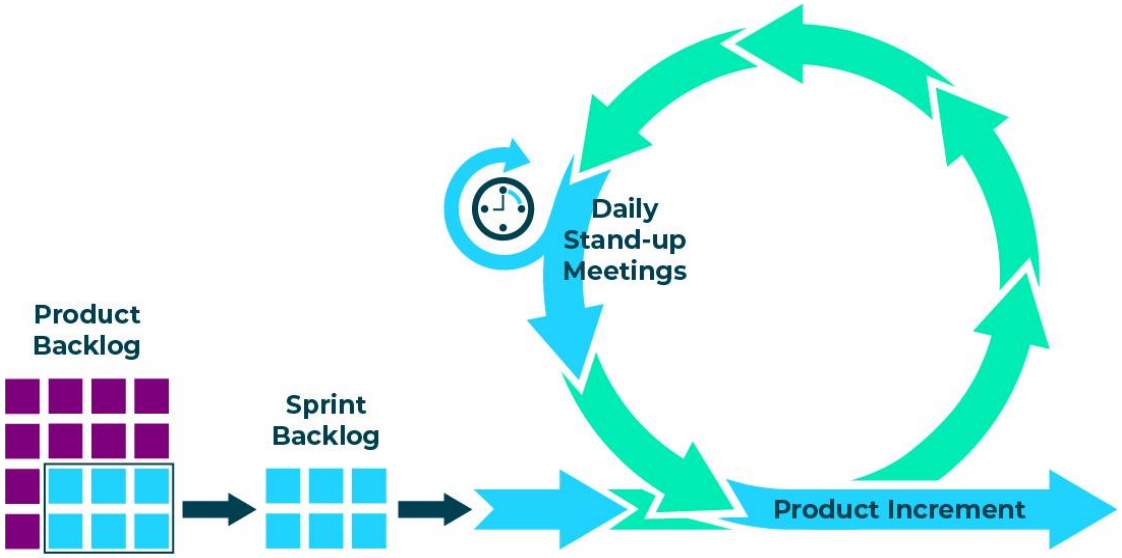
The sprint backlog is the set of product backlog items selected for inclusion in the sprint. It shows all the work that the development team think necessary to meet the sprint goal.

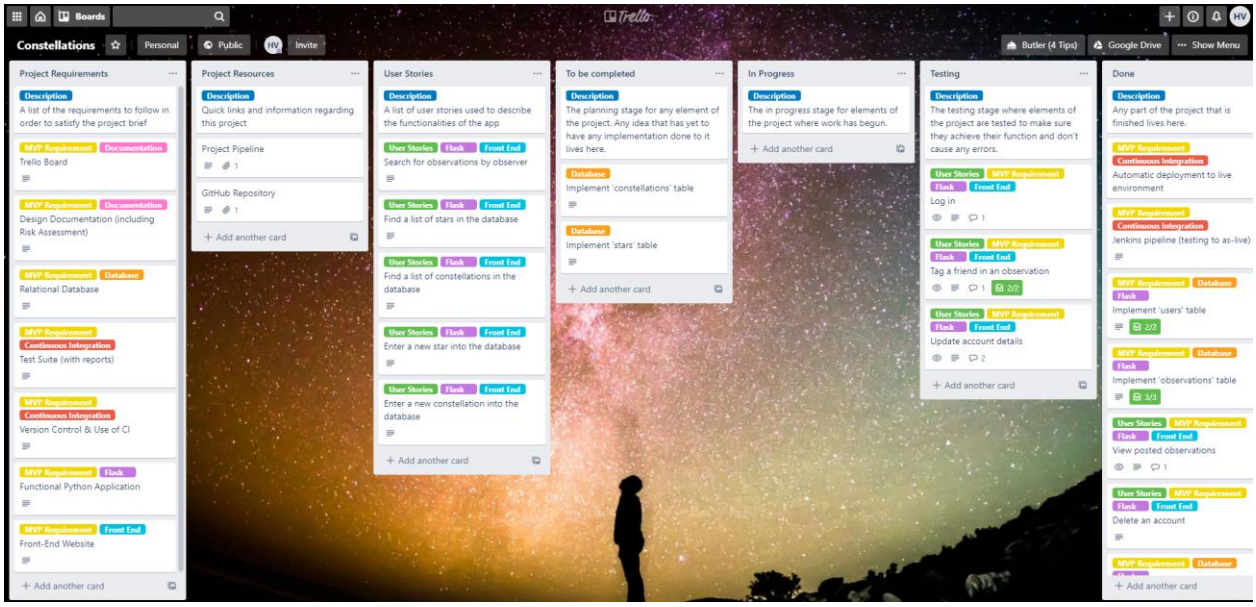
As we work on items, they move through the workflow. The workflow usually has at least three stages, 'To be completed', 'In progress' and 'Done'.

A sprint backlog is usually presented in a Kanban board, where tasks are written onto cards and placed into columns that correspond to the three stages mentioned previously (though more columns may be added!).

As we work on tasks, the cards move from one end of the board to the other. This movement helps the team to see the project's progress.

The Kanban board could be a physical whiteboard with sticky notes, or it could be part of a project-tracking software. Common project tracking tools include Trello, Jira, Asana and GitHub Project Boards, but there are loads out there.





To make the scrum framework run smoothly, a set of roles are given out which allocate certain responsibilities. These are roles and not necessarily jobs, meaning that someone can take more than one role. These roles are:

Project manager (PM)

PMs manage projects and will overview expenses. They will also try to reduce risk on the project.

Product owner (PO)

POs are the sole person responsible for managing the product backlog (i.e. the list of features to be worked on).

Business analyst (BA)

A BA supports the product owner by gathering requirements and providing guidance on what to build. They usually work across many products.

Development team

A multi-disciplinary development team usually consists of software architects, designers, programmers and testers.

Scrum master (SM)

The scrum master helps those outside the scrum team understand which interactions are beneficial. The scrum master supports the development team by removing impediments, facilitating meetings and coaching self-organisation.

Stakeholder

A stakeholder is anyone with an interest in or an influence on the product.

There are four different key meetings which take place in the sprint and are discussed in this module.

These meetings are:

- Sprint planning meeting
- Daily standup
- Sprint review meeting
- Sprint retrospective meeting

Sprint planning meeting

This meeting is held at the beginning of the sprint. The meeting will usually take the full day for a two-week sprint and is split into two sections (what and how).

The first half of the meeting is where the team decide what is going to be brought into the sprint backlog.

The second half of the meeting is where the team decide how the items they have brought in are going to be completed, usually by further breaking down the tasks into smaller pieces and adding technical details.

Daily standup

This is a daily 15-minute meeting. To keep conversation quick and on point, attendants must stand up during the meeting.

This is used to optimise communication across the team.

During the meeting the following points are discussed by every participant:

- What did I do yesterday that helped meet the sprint goal?
- What will I do today to help meet the sprint goal?
- Do I see any future impediments that prevents the team from meeting the sprint goal?

Sprint review meeting

A review is held at the end of the sprint to inspect the work done and adapt the product backlog. The development team, product owner and key stakeholders are all present at this meeting. In the sprint review meeting:

- The development team demonstrates what work was done, and answers any questions
- The product owner discusses the product backlog as it stands
- The entire group collaborates on what to do next, to provide valuable input to future sprint planning

Sprint retrospective meeting

The sprint retrospective meeting is an opportunity for the scrum team to inspect itself and create a plan for improvements to be made during the next sprint.

The goal of the retrospective is to:

- Inspect how the last sprint went with regards to people, relationships, process, and tools
- Identify what went well and potential improvements that could be made
- Create a plan for implementing improvements to the way the scrum team works

The most common type of product backlog item (PBI) is a user story.

However, there are multiple forms an item can take, which include:

- User stories
- Use cases
- Bugs, errors and fixes
- Constraints

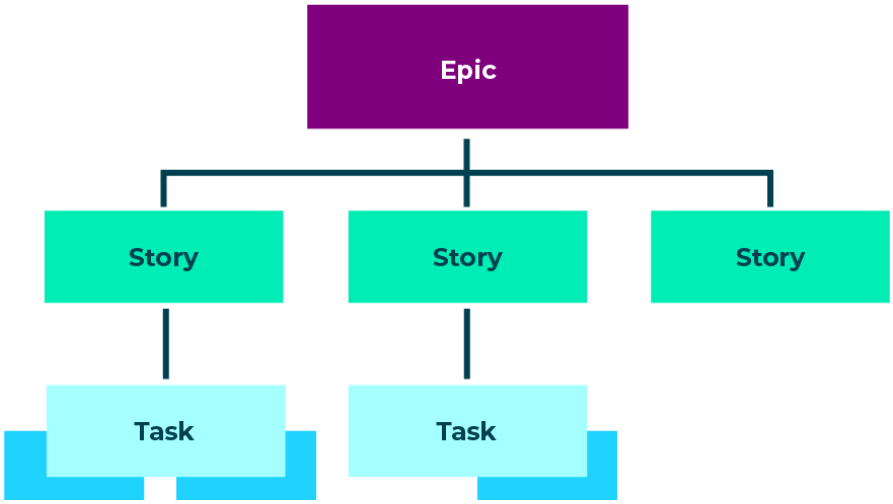
User story

A user story is an end goal expressed from the user's perspective:

- It focuses on what they want to do rather than how to do it
- These users can be external or internal
- They should be expressed in simple language that everyone can understand

An epic is made up of multiple user stories.

A user story is made up of multiple tasks.



Definition of ready (DoR)

Definition of ready (DoR) defines what a product backlog item needs before it can go into the sprint backlog.

A checklist of items could include:

- Technical details that have been discussed on the item and agreed
- Assigning a priority
- Timing estimates

Completing a product backlog item (PBI)

To complete a product or item of work, it must go through a series of checks to make sure it meets the standards set out by the team.

Definition of done (DoD)

Definition of done (DoD) defines what is needed before it can be regarded as complete. A definition of done can be applied to a feature, a sprint or a release.

A checklist of items which could be included for a feature could include:

- Unit testing written and passed
- Documentation updated
- Peer code review completed

Acceptance criteria

Acceptance criteria dictates the conditions for software to be considered done. It is a set of statements that usually have a pass / fail result for all requirements.

Acceptance criteria are attached to our user stories to understand what a feature needs. It is easier to understand the minimum viable product based on these criteria, and you can also derive tests from the criteria.

Planning and estimations

In scrum, story points are used to estimate the amount of work needed to complete a task. A story point is a unitless measurement; it is the relative difference which is important.

When assigning story points to a task, it is important to note that **time is not used to estimate work.** Time-based estimates don't work, because an hour's work will vary between developers. This means that time is a bad measurement for predicting the amount of work.

It is easier for developers to agree on relative amounts of work. For example, not everyone would agree on the time it would take to build a chair. However, it is easy to agree that the relative time to build two chairs is twice as long as one chair.

Planning poker

Going through multiple features one after another and discussing them individually can be a very monotonous task. To alleviate this, there are plenty of games used to encourage engagement of participants when estimating work.

Rules

1. The developer team is handed out a pack of cards - usually a modified fibonacci sequence of numbers[1].
2. Next someone will read out a user story.
3. Each developer puts down an estimate with the number of story points they think that work will take.
4. Those who chose outlying story points must talk about why they chose that number in particular.
5. If there was a difference in the card numbers, repeat from step 3.
6. If all cards are the same then move onto the next user story.

[1] A modified Fibonacci sequence (e.g. 1, 2, 3, 5, 8, 13, 20, 40 and 100) is often used because it's easier to agree on the difference in magnitude between 3, 5 and 8, for example, than between 3, 4 and 5. You can read more about it here.

Note: It is not a vote, the game continues until there is a unanimous decision.

Velocity

Velocity is the amount of work the team is able to get done in one sprint. It is based on experience from previous sprints (estimated for first few sprints) and can be measured in ideal days or story points. It is designed to help improve accuracy of estimations over the length of a project.

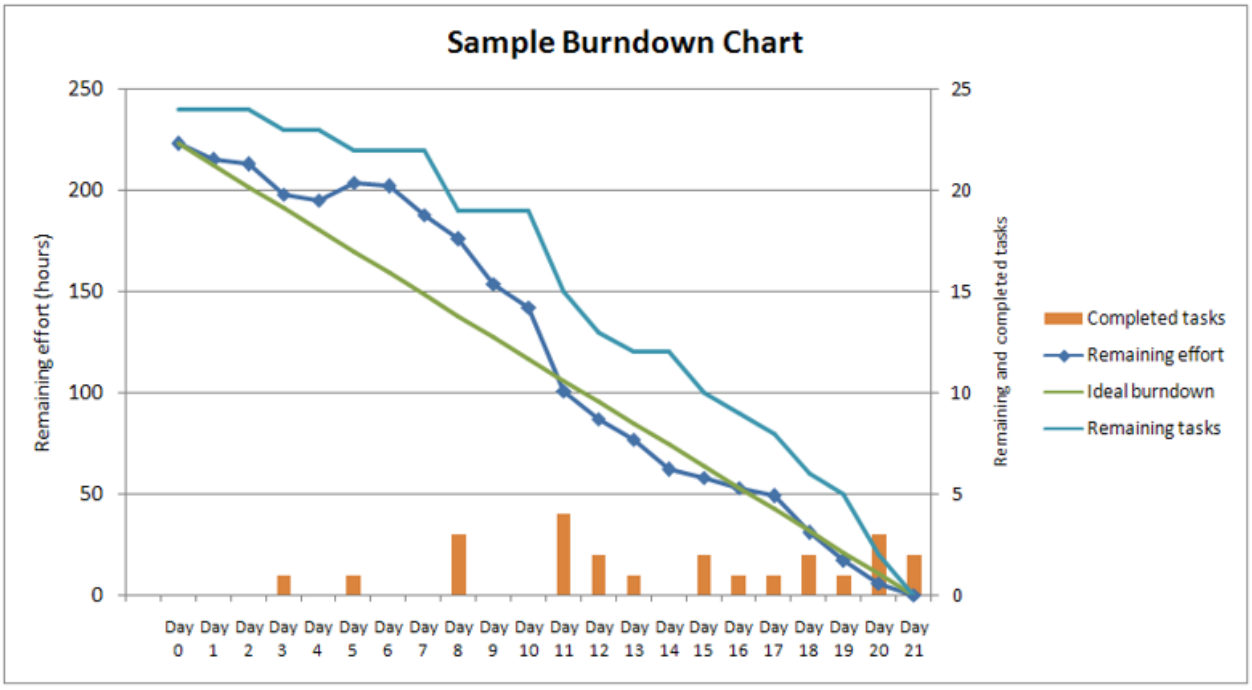
Velocity – how many story points the team can complete in one sprint.

Capacity – the size of the team, the length of the sprint, any absences within the team.

Capability – the team's knowledge and any impediments around this.

Burndown charts

As the tasks are completed throughout the sprint, the amount of work left will decrease. Progress can be presented in a graph known as a burndown chart. This chart is designed to show the work done by the whole team.



DevOps as a culture

The term DevOps is a difficult one to define, as different organisations have varying approaches to implementing DevOps. It is best described as a cultural approach to software development project structure, with a particular philosophy, designed to achieve the following:

- Increased collaboration
- Reduction in silos
- Shared responsibility
- Autonomous teams
- Increase in quality
- Valuing feedback
- Increase in automation

How things used to be done

Traditionally, software companies are structured in separate, stratified teams for development, quality assurance (sometimes known as testers), security, and operations.

These teams tend to have varying and sometimes conflicting goals and there is often poor communication between them. This regularly results in work that is out of sync with other parts of the organisation. These isolated teams are referred to as silos.

As a result, this structure regularly results in slower releases, wasted time (and therefore money), and blame cultures, where production problems become the fault of another team.

How DevOps changes things up

DevOps is based upon agile project management, an approach to projects that promotes:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

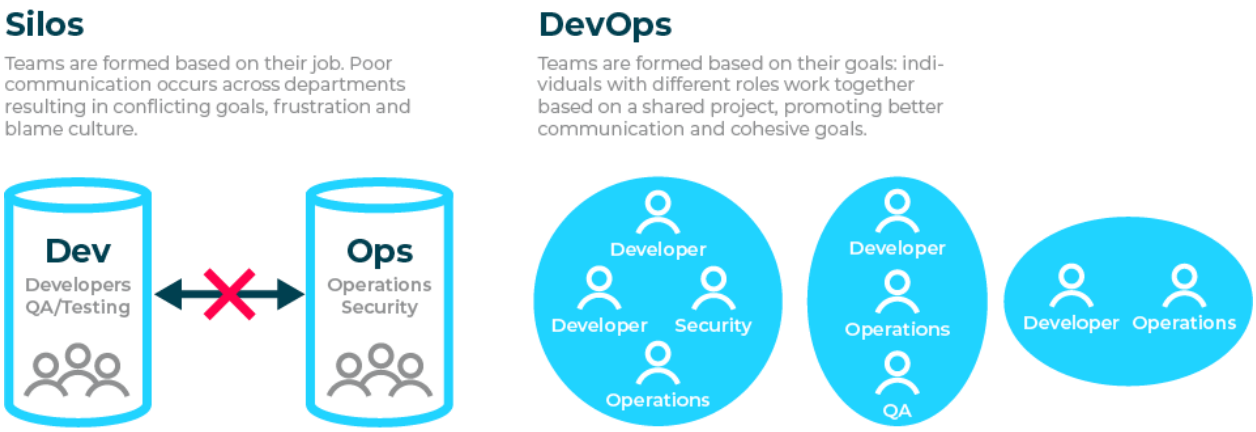
Agile encourages flexible teamwork - with the ability to fail (and recover) fast and celebrate achievements. It aims to promote a productive work culture and bridge the gap between developers and customers.

Agile development teams measure their progress by the amount of working software. Agile teams have product owners, developers, testers, and user experience (UX) people working together with the same goals in mind.

While agile focuses on bridging the gap between developers and customers. DevOps is focused on bridging the gap between developers and operations teams.

Within software companies, there has historically been friction between the developers (who are responsible for generating code and creating new software features) and operations teams (who are responsible for IT infrastructure, security, and application deployment).

Developers would regularly generate code that broke the applications they were working on. Then, operations teams would throw back to developers in frustration, and often without sufficient details of the problem. This then results in slower release times, team-members being unable to focus on their primary responsibilities, and general frustration within the organisation.



Adoption of the DevOps methodology requires the **dismantlement of silos.** Devs and ops are encouraged to break down their silos and start **collaborating**, as well as **share the responsibility** for maintaining the system on which the software is running, and prepare the software to run on the system with better **feedback and delivery automation.**

It is the role of a DevOps engineer to create an infrastructure that encourages collaboration. This is done through **automated, continual testing and integration of new code**. This means that developers can focus on developing - and operation teams can focus on maintaining their IT infrastructure.

Software organisations may find the transition to a DevOps method jarring. Silos become dismantled and teams reorganised. Staff may find responsibilities – once defined by their job description – to be less specific due to the emphasis on shared responsibility. This may result in staff feeling lost in the new paradigm.

With the right support during the adoption of DevOps, individuals should adapt quickly. Then, staff can start reaping the benefits of

- increased collaboration within their organisation
- and increased productivity from tedious tasks being fully automated

There are two key elements to the DevOps approach that ensure that the method works: **automation** and **measurement**.

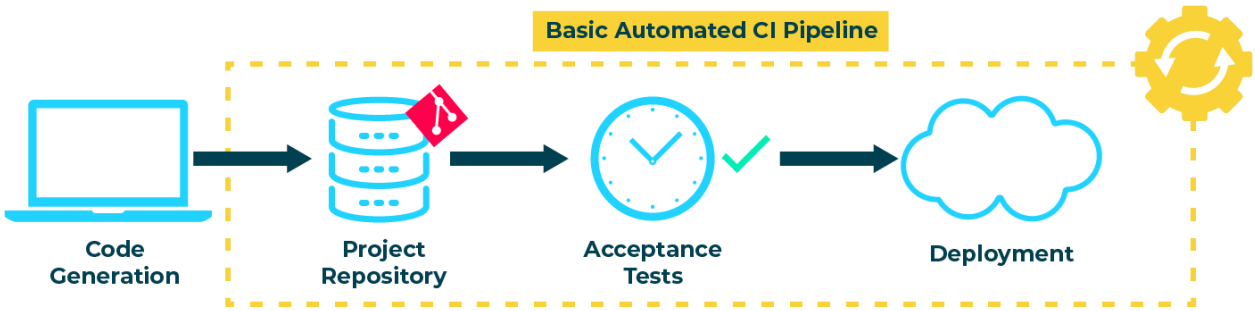
Automation

A DevOps culture encourages automation in as many areas of the production pipeline as possible. As a rule of thumb: if a machine **could** do it, a machine **should** be doing it.

Working manually allows the potential for human error and slower development/deployment times. People are not best suited to performing repetitive tasks - we tend to get bored and distracted.

Machines, on the other hand, never tire nor become bored with their work. Automation provides a level of consistency, predictability, scalability, and quality for this reason.

Eliminating the human error in a production pipeline allows teams to spot issues in the production workflow. This enables issues to be fixed and streamlined to provide maximum output.



Automation allows for faster production times from development to delivery and allows developers to focus the majority of their time on generating new features. The result is an efficient pipeline and happier developers, as they get to focus on the creative aspect of their work.

Some automation techniques include:

Continuous integration (CI)

- When code is committed to a repository, it is automatically built and subjected to acceptance tests
- Test failures result in the code being prevented from integrating with the repository. Developers are immediately notified of a test failure so they can fix issues as quickly as possible

Continuous deployment/delivery (CD)

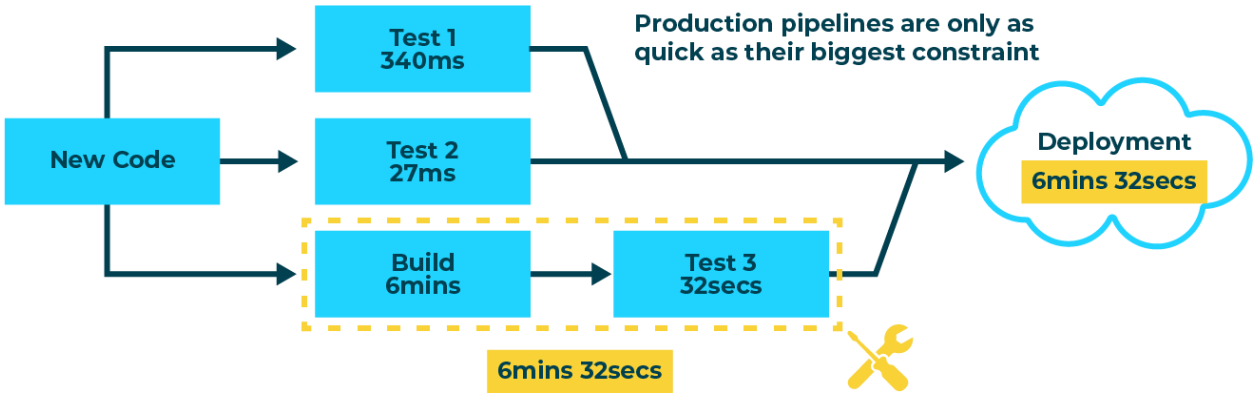
- As new code passes acceptance tests, it is automatically integrated into a deployment environment
- Being able to choose a version to deploy with one push of a button requires a fair amount of automation

Infrastructure as code (IaC)

- To deploy and redeploy production environments (i.e. where your application lives at each stage in the production pipeline) quickly and easily, IaC is used to specify the configuration of a computer environment with easy-to-write/read config files
- Having environment infrastructure declared in code allows for infrastructure to be created or modified using version control
- IaC allows environments to be easily replicated, so they stay consistent across the pipeline. This is important for testing environments where you want to replicate the deployment environment as closely as possible

Measurement

Measurement is central to ensuring that a production pipeline is working efficiently. After all, how can you know something is working better if you can't measure it?



Accurate and precise measurements allow us to *pinpoint constraints* in the pipeline and fix or improve them faster.

Measurements are also important from a cultural standpoint, as they can inform teams when they're working more productively and what can be done to improve.

The types of metrics we work to measure include:

Frequency of deployments

- DevOps pipelines encourage frequent, smaller updates to software. Charting the frequency of deployments is a good indicator of the effectiveness of a pipeline
- Upon adoption of the DevOps methodology, deployment frequency should tend upwards until it reaches a natural plateau, though fluctuation is normal

Mean time to recovery (MTTR)

- This refers to the average time it takes to solve problems that impact the end-user. Common problems include outages, security issues, and severe bugs
- This is a more worthwhile metric than charting the frequency of failures. DevOps is more interested in the speed at which problems are solved, rather than minimising problems altogether. Failure is inevitable: what matters is how well we respond to it

Mean time to discovery (MTTD)

- This refers to how quickly problems are discovered. The faster problems are identified, the faster they can be fixed. This time is measured from the point of integration into production to the point the problem is identified. Naturally, faster MTTDs are more desirable
- This metric should also indicate whether discovery is made by the customer or the automated systems, with the latter being more desirable

System availability

- In almost all cases, we want our systems to be available at all times to customers. Knowing the availability of our systems allows us to pinpoint which parts of our infrastructure need attention

Service performance

- Measuring this metric allows us to see whether our services are running within the desired thresholds
- Metrics may include response times per request, CPU load, or how long it takes for a website to load

Continuous integration

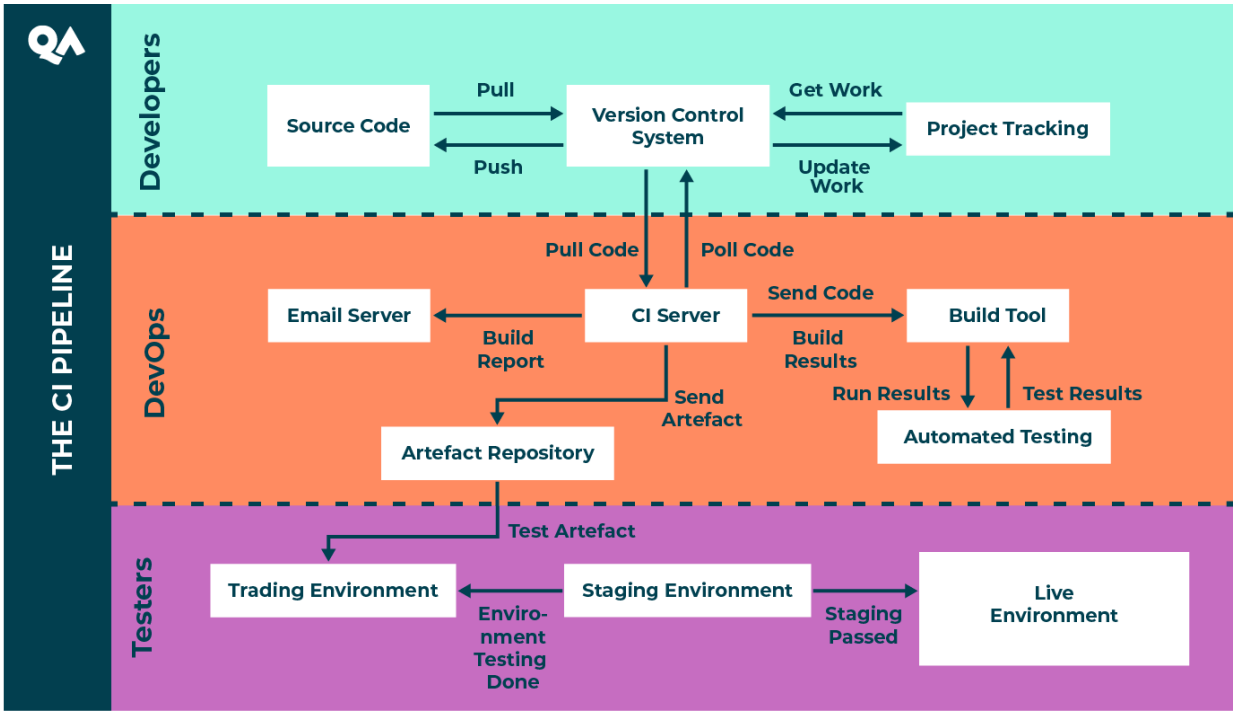
Continuous integration (CI) is the automated integration of code from many contributors into a single software project.

The purpose of the CI pipeline is to allow developers to integrate newly-generated code easily and frequently. This is achieved through the use of automated testing

tools to check the correctness of code before full integration. Additional checks that may also be performed include syntax style, code quality, etc.

At the heart of the process is the version control system (VCS) and the CI server:

- The VCS is designed to track changes to code over time as contributors add new features to the application. This system allows for cohesive collaboration, and the ability to easily revert an application to a previous stable state if new code breaks something
- The CI server handles all the automated building, testing, and deployment of code as it is pushed to the VCS



This pipeline diagram illustrates the many parts that make up the CI pipeline.

What CI does

CI allows developers to generate and implement new functionality with ease and speed by automating the integration process. Building and testing of new code is handled through automation, allowing developers to focus their efforts on the creation of new features.

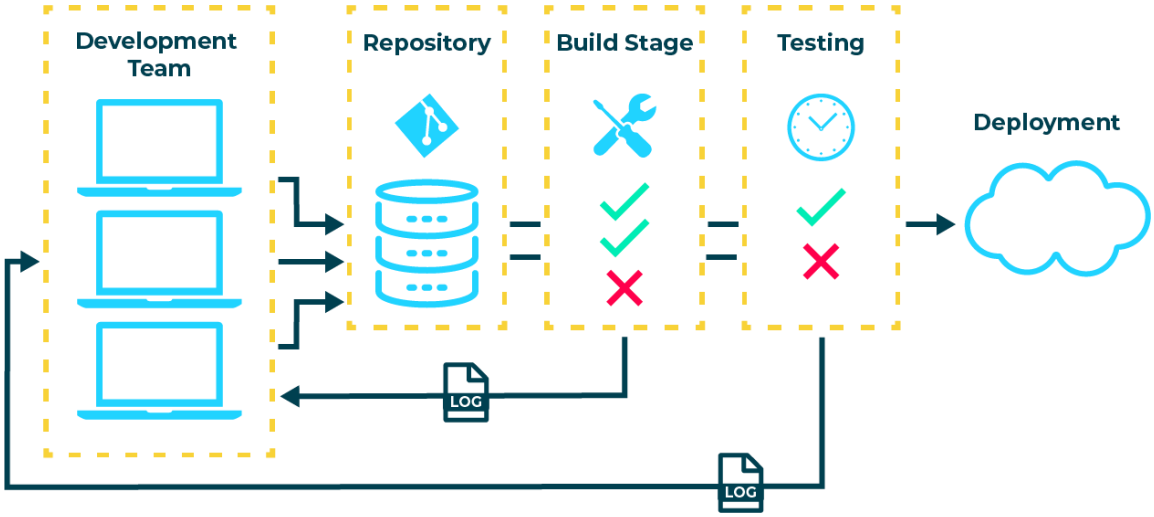
Developers work on features independently and in parallel with each other, without having to worry about clashes within their teams. Tracking and merging code additions allows for smoother integration of new features without fear of irreparably damaging the source code.

A continuous integration pipeline should:

- Maintain a single source code repository for a project

- Have a ‘master’ branch that should always be ready to deploy
- Keep all team members informed of every update to the source code
- Automate build processes
- Automate testing of new builds
- Inform developers of test failures with detailed logs
- Encourage smaller, frequent deployments of code

How CI works

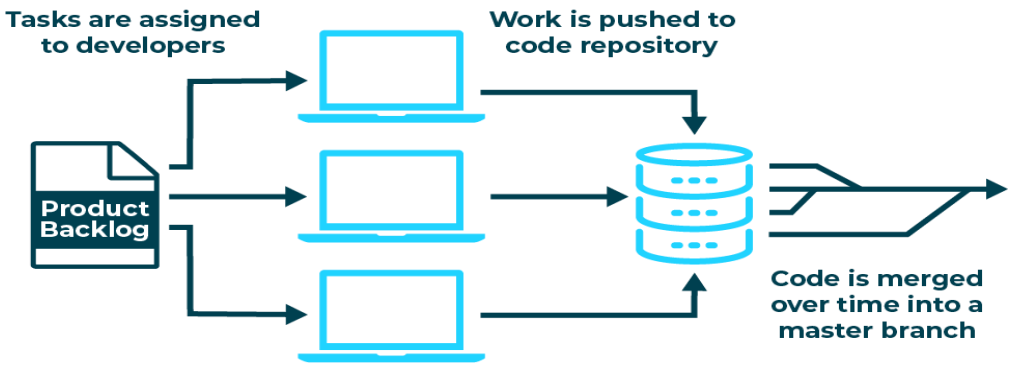


To achieve this pipeline, CI leverages many software tools to handle the automated building, testing, and deployment processes. The main steps in the CI pipeline include:

Code generation

Typically, a CI pipeline would be utilised within an agile workflow. As such, a list of tasks would be compiled into the backlog for developers to refer to. These tasks are divided up to the developer team(s).

These tasks are then worked on in parallel and independently between the developers. Once a task is completed, they will push it to the code repository to be integrated into the CI pipeline.

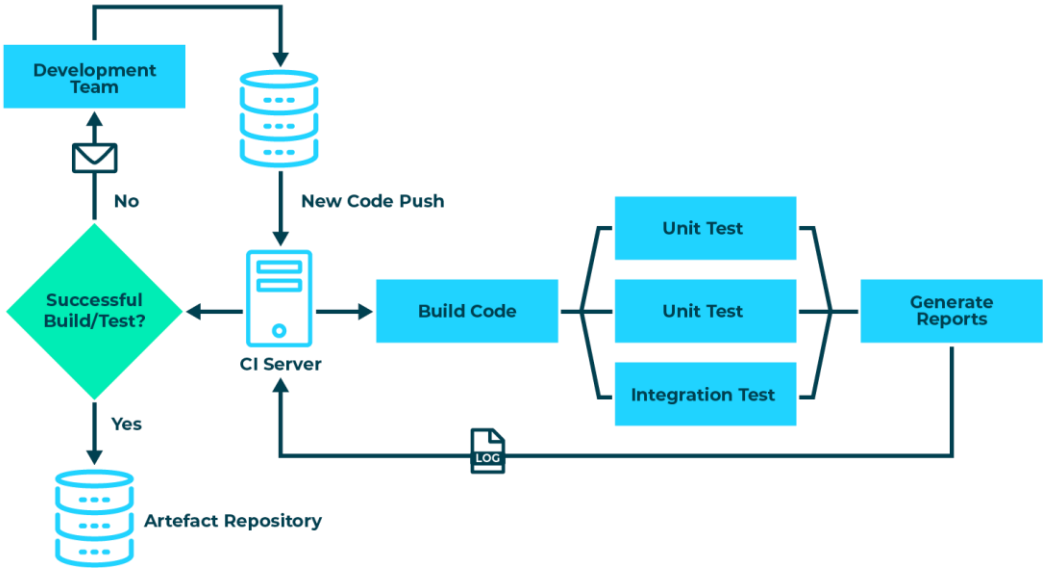


Code repository

The code repository (often referred to as the repo) is where all the code is stored. It is managed using the VCS, which handles the integration of new code into the existing project. It checks for compatibility issues between the new and existing code and alerts the developer, making sure that a code push doesn't result in lost data.

Building and testing

Once the push has been made, the VCS will trigger the CI server: a tool for automating integration and deployment processes. The CI server will build the application with the new code and perform acceptance tests to check the code works correctly.



If the new code won't build or it fails any tests, integration is halted and the developer is informed of where the code failed, along with detailed reports. This means bugs are caught long before they are integrated into the application and the developer is able to solve the problem quickly and efficiently.

These reports should contain such information as logs, error messages, build labels, build times, etc.

Successful builds that pass their acceptance tests get added to the artefact repository, ready to either be tested in a testing environment or sent straight to the staging/live environment to be served to the customer.

Benefits

There are a range of benefits to employing a CI pipeline to a production workflow that aren't limited to that of the developer and operations teams, but often positively affect the entire organisation.

This may be through:

- cost-effectiveness,
- facilitating better planning,
- and greater transparency and understanding of underlying processes,

allowing the company to better execute marketing strategies.

Some of the main benefits include:

Scaling

CI cuts out a large chunk of overhead occupied by manual code building/testing, slow communication channels both within and outside of the team, and highly stratified team structures.

Freeing up this time and energy in the production workflow frees up resources that can go towards scaling up the development team, code generation, code integration, and more.

Overly-planned release schedules can be abandoned for more frequent updates, scaling team productivity.

Feedback loop

Frequent and gradual feature updates allow for far more opportunities for business feedback. Teams can test new design ideas, new features, and get feedback about them faster, allowing for an agile approach to product development.

It also allows for more client/customer feedback, as development teams can now show off new product features to the people they're being designed for, allowing them to adjust their product accordingly.

Incidentally, bugs and other issues can be rapidly fixed due to CI minimising the hassle that comes with redeployment.

Communication

Leveraging VCS enhances communication between the teams, as all changes are easily trackable. Teams and individuals are therefore more aware of each other's progress.

Stronger communication means teams/individuals avoid stepping on each other's toes and impeding each other's work.

Greater awareness of progress also aids transparency of work across the organisation. Other non-technical teams are much more able to review and understand what development teams are working on and how much they are achieving.

Challenges

While CI comes with a wide array of benefits, adopting the approach is not without its challenges.

Installation and adoption

The greatest hurdle for using CI is its initial adoption and technical configuration.

If there isn't an existing solution in place, the installation of the new pipeline is likely to be a long and involved process that has the potential to waste time, effort and money, should it be approached without enough planning.

Hence, before installing a CI pipeline considerations need to be made about the existing engineering solution and how it will need to be built together.

A clear design approach with explicit and thoroughly thought-out goals will facilitate a smoother adoption process with as little additional cost and effort required.

Learning curve

CI pipelines make use of many different and relatively new technologies that teams may not have any prior experience with, resulting in an initially high learning curve.

The main technologies required are version control systems, new hosting infrastructure, and container orchestration.

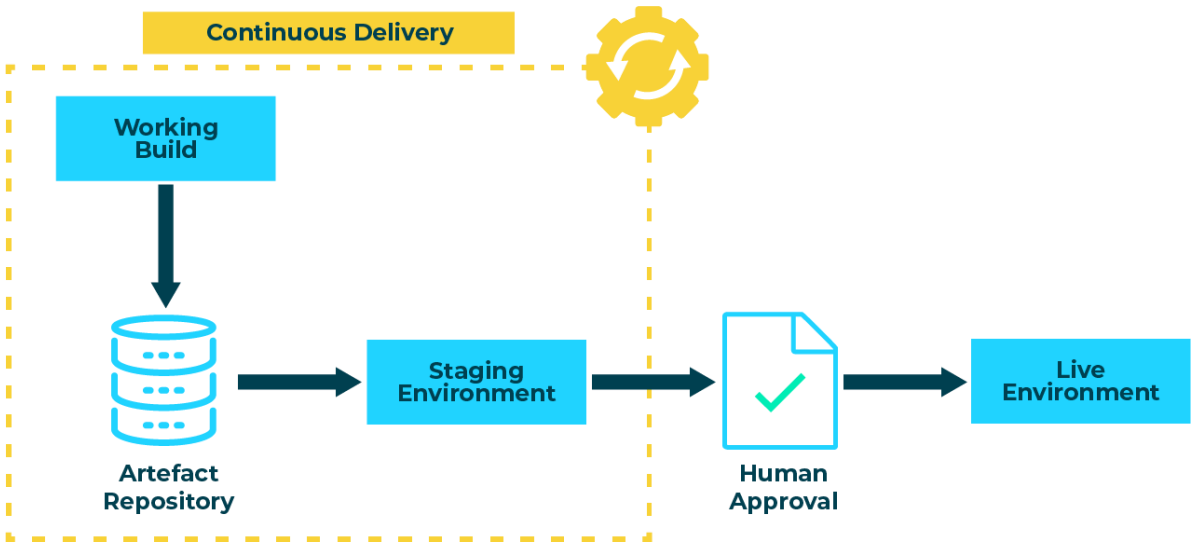
Not only is the technology likely to be different, but the workflow itself may also take some getting used to. Some members of the team may find their main responsibilities (e.g. testing) have been largely automated, ultimately requiring them to adopt and adjust to a new set of responsibilities.

CD in the Enterprise

Two extensions of CI that are commonly employed are **continuous delivery** and **continuous deployment** (both of which are confusingly shortened to CD). They're quite similar to both CI and each other, but it's worth knowing the difference.

Ultimately, they are two similar practices that extend CI's scope by automating the delivery of code to the end user on a regular basis.

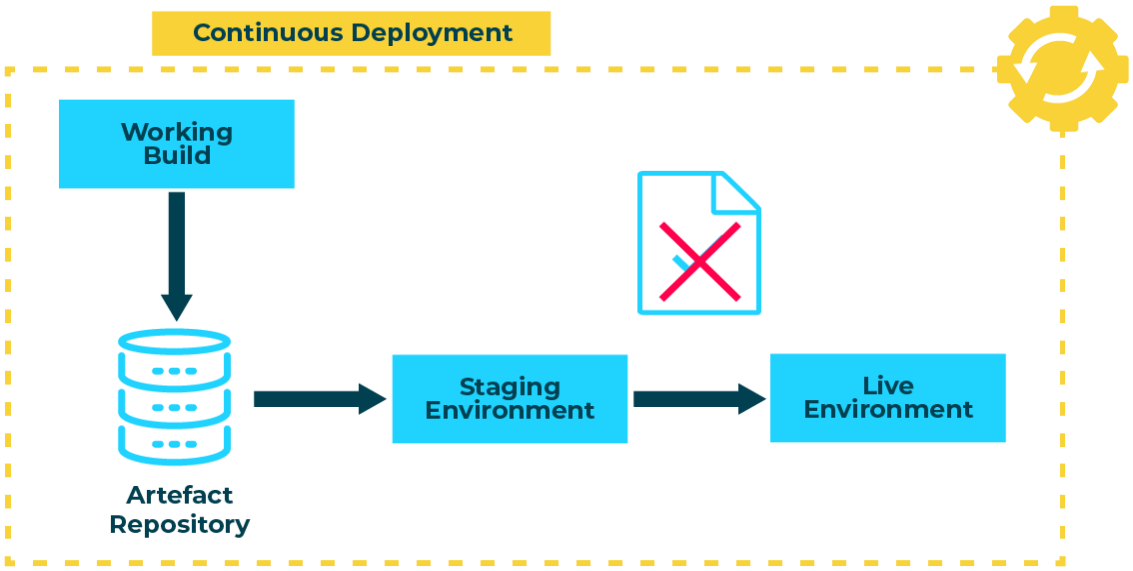
Continuous delivery



Continuous delivery is an extension of the CI philosophy that seeks to deliver new features to customers on a regular basis. So, while CI integrates code regularly, continuous delivery also **delivers** that code regularly.

It requires an automated process in place so that new releases only require approval via a click of a button to go live. This approach encourages organisations to deploy their applications as early as possible in smaller batches.

Continuous deployment



Continuous deployment further extends the continuous delivery practice so that feature releases are entirely automated. No human interaction is required for new application features to end up in the customer's hands.

Continuous deployment workflows have the benefit of eliminating the idea of a 'release date', allowing developers to simply focus on creating new ideas without time constraints.

Environments and containers

An environment is a computer system for code to run in. This may refer to your local computer, a virtual machine, a cloud network, etc. As code is developed as part of a production pipeline, it will migrate to different environments before being fully deployed.

Code will almost always have external dependencies that need to exist in the environment, e.g. code packages, environment variables, and configuration files. Therefore, we need a robust method for migrating code from environment to environment so that the code runs reliably on any system.

Environments

A production pipeline will contain several environments that your code will pass through before being fully deployed. Each of these environments is separate from one another. The typical environments you will encounter in a pipeline are:

- Development
- Testing

- Staging
- Production

Each has a separate computer infrastructure associated with it. As a DevOps engineer, it is your job to architect these environments so that your app runs correctly on each.

Development

The development environment refers to the environment in which developers are creating new code. Physically, this environment most likely resides on the individual developer's computer, but may also exist as a virtual machine in the cloud. It is from here that developers will generate new code and commit it to a code repository.

Nothing developers do in this environment should impact what the end-user sees. This is a safe space to create new features that will regularly break the code. It is important to use the feature-branch model of the version control system to isolate these new features from the working code until they have been rigorously tested.

Testing

The testing environment is where both automated and non-automated tests are carried out on newly developed code. Typically, tests should be as automated as possible, so that code is added to the code repository and immediately subjected to unit and integration tests. Should the code fail any of these tests, the developer will be notified with detailed error logs so they can attempt to fix the issue.

The exact nature of this environment will change per production pipeline based on the type of application that is being created, and there may be multiple test environments.

Staging

The staging environment is where code is tested in an environment designed to be as identical to the production environment as possible, but is not accessible to end-users. It is also sometimes called an ‘as-live environment’ for this reason.

It is here that code is tested as if it were being delivered as a final product to the end-user. As such, it is the last stage in testing code and fixing any bugs before it is served to users. Performance and load testing (essentially testing how the code performs under pressure from increased usage/network traffic) is also performed here. This is also where new features are demonstrated to the client, if you have one.

Production

This is the environment that the app will run from in once it is considered ready for deployment to the end-user. Ideal production environments will result in zero downtime for the user as updates are carried out. This can be approached in several ways that will be discussed at a later point in the course.

It's increasingly common for code updates to be rolled out gradually to some groups of users first rather than to all users at once. This allows you to catch any bugs you may have missed in the staging environment before they affect all users of your service.

Containers

Containers are lightweight software packages that are designed to contain all the necessary dependencies for code to run correctly, including:

- The source code
- The application runtime
- System tools
- System libraries
- Settings

In essence, they are miniature virtual environments that you can put your code into so it can run anywhere. Consider them to be neat little packages containing your code and everything it needs to run smoothly. Because all your code's dependencies are wrapped up within its container, it will reliably run on any operating system with any configuration. This reduces the headache of migrating your code to different environments massively.

Infrastructure consistency

A production pipeline will require the migration of source code to various environments before being fully deployed to the end-user.

If these environments are too inconsistent, code could begin to work unreliably. It is part of a DevOps engineer's job to maintain consistent infrastructure across environments.

This module discusses some of the tools DevOps engineers employ to maintain consistency across infrastructures.

Scripts

All cloud infrastructure can be created from the command line.

This is no doubt the fastest method for spinning up a single VM in your cloud network, as it often only requires one or two commands.

Using a script, you can write a sequential list of commands that you can run as many times as you want.

2. vim deploy-vm-2.sh

```
#!/bin/bash
groupName=chaperooNew
vmName=chaperooVmNew
image=UbuntuLTS
userName=ubuntu
cloudInit=run-app.txt
storageAccountName=testdiagnostics1234
dataDisk=newChaperooDisk

az vm create \
  --resource-group $groupName \
  --name $vmName \
  --custom-data $cloudInit \
  --boot-diagnostics-storage testdiagnostics1234 \
  --attach-os-disk $dataDisk \
  --os-type linux

az vm boot-diagnostics enable --storage $storageAccountName  --ids $(az vm list -g $groupName --query "[].id" -o tsv)

az vm open-port -g $groupName -n $vmName --port 80 --priority 1010

~
-- INSERT --
```

The benefit of writing scripts is the amount of control and configuration you can achieve.

The downside is that it can be a slow process and somewhat inflexible compared to other approaches, as well as becoming unwieldy with larger, more complex infrastructures.

Infrastructure as code

Infrastructure as code (IaC) refers to the process of using code to describe cloud infrastructure, including virtual machines, virtual networks, auto-scaling sets, database platforms, and more. The idea is that you can write an entire cloud environment for your applications to live in using code.

The benefit of this is replicability: you need only write your code once, but you can create as many replicas of this infrastructure as you need.

The IaC approach differs from writing scripts in that it is a declarative approach.

This means that rather than having to write the necessary commands for deployment in the order each resource needs to be deployed, you are simply declaring all the resources that are needed and your IaC tool will handle the necessary sequence of events.

This makes IaC scripts much easier to write and read, saving you a lot of time, especially when dealing with large cloud environments.

Common IaC tools include:

- Terraform

- AWS CloudFormation
- Azure Resource Manager (ARM)
- Chef

Configuration management software

Configuration management software is used to automatically provision and configure computer systems. With a simple script, you can spin up a virtual machine and automatically install all the necessary software you want on the machine.

It is particularly useful for keeping computer environments in an *idempotent* state, which is a fancy word that (at least in a software development context) means the system or service will always perform the same way every time we use it.

Computer infrastructure experiences configuration drift over time, meaning the computer environment changes gradually as a result of the sheer number of processes that are occurring at any given moment. Eventually, this can cause code to run erratically.

Configuration management tools keep these systems idempotent by running checks on an existing computer environment to make sure they match with their desired state; if it doesn't, it will run the necessary commands to effectively pull the configuration back to its desired state.

The most common configuration management tool you'll encounter is Ansible.

Container orchestration tools

Containers are lightweight environment packages that store code along with all its necessary dependencies. Container orchestration tools allow multiple containers to work together as a network, so that you can have multiple containerised services interacting with each other.

These orchestration tools can also be used to:

- Easily replicate and scale out containers based on incoming traffic
- Gradually swap out containers running old code with new, updated code, so that the end user never experiences any service outage
- Manage and migrate containers across multiple machines
- Monitor the health of each container

Common container orchestration tools include:

- Kubernetes
- Docker Swarm
- Amazon Elastic Kubernetes Service (EKS)
- Azure Kubernetes Service (AKS)

- Google Kubernetes Engine (GKE)

What is The Cloud?

The terms ‘cloud computing’ and ‘the cloud’ have been used to describe all kinds of different technology. Are we talking about distributed computing? Networked services? Virtualised servers or hosted services?

The actual definition of cloud computing as reported by NIST (National Institute of Standards and Technology) is:

> *Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g. networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.*

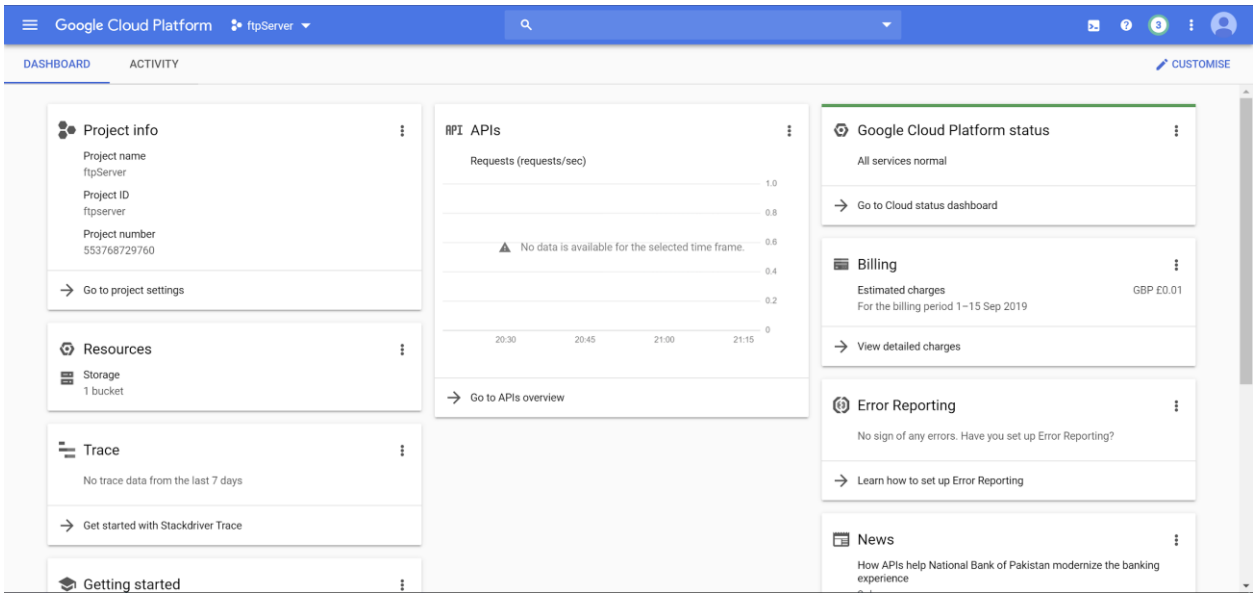
This document is designed to guide you through some of the key concepts behind cloud computing.

On-demand self service

All cloud providers have a dashboard through which the customers can control the services that the provider offers, hence the name on-demand self-service.

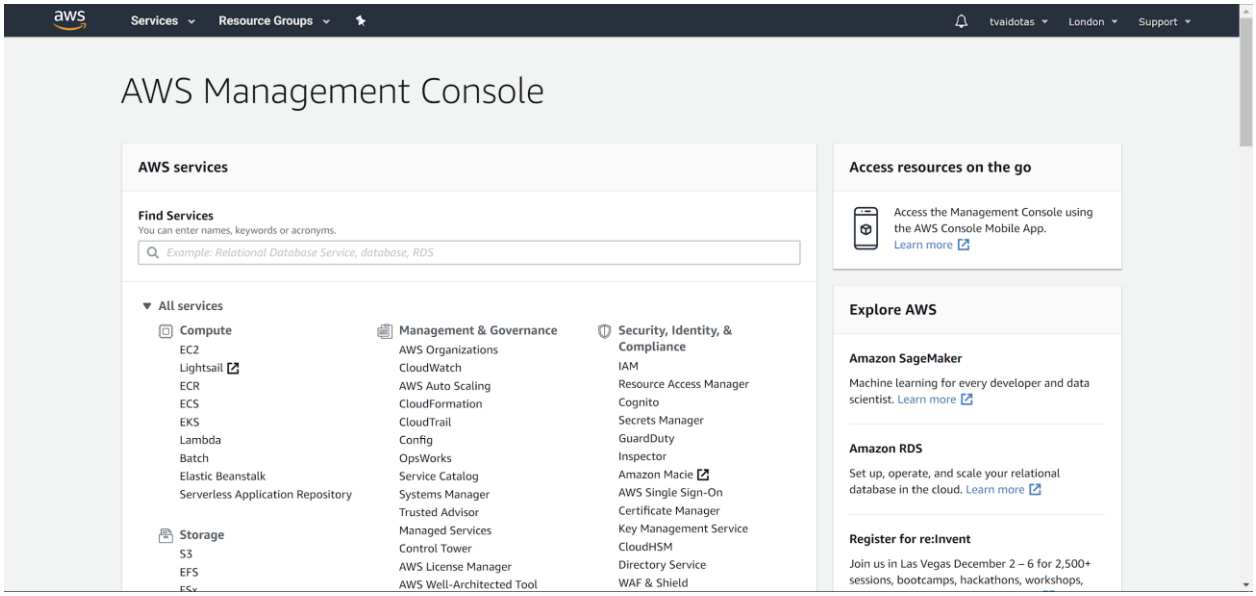
Google Cloud Platform (GCP)

This is how it looks for Google Cloud:



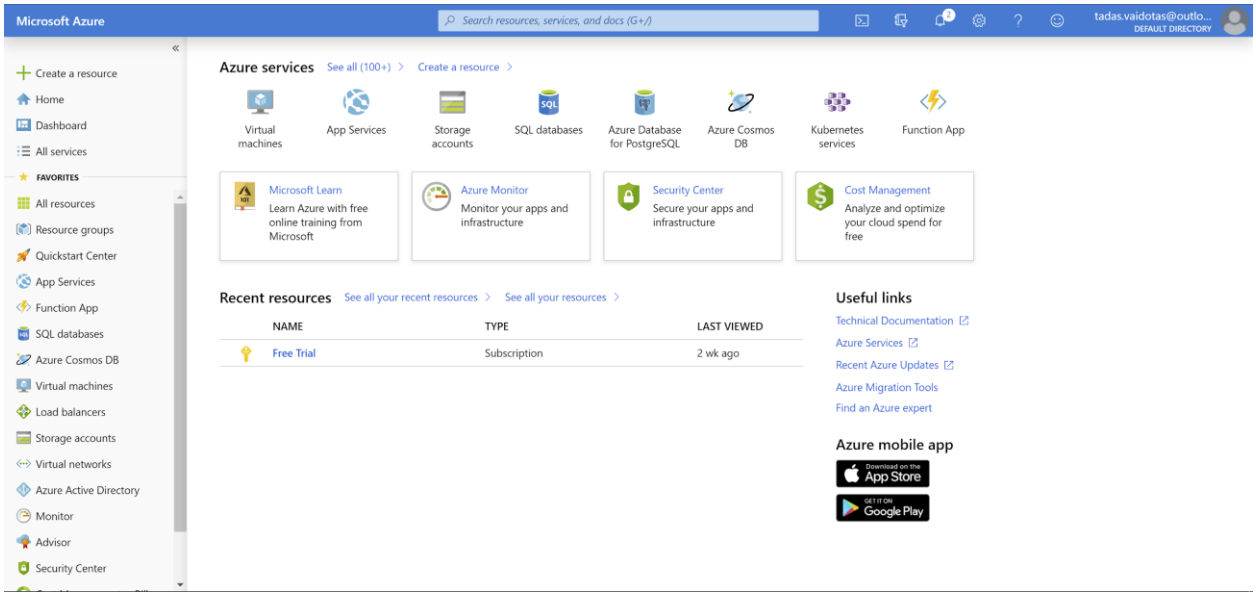
Amazon Web Services (AWS)

This is how it looks for AWS:



Microsoft Azure (AZ)

This is how it looks for Azure:



Consumption based pricing

Consumption based pricing is a service and payment scheme where the customer pays according to the amount of resources used. This is similar to how you would pay utilities companies for electricity/gas/water based on how much you've used it.

The prices may have flat rates per resource or they could be varied rates for different components.

Broad network access

This relates to access over a network via standard mechanisms, which would generally be taken to mean standard protocols like HTTP or TCP.

Services should be accessible to a variety of clients running on various hardware (phones, laptops, desktops). In other words, if it's only accessible using a proprietary protocol or data format from a custom client, it's probably not cloud computing.

Notice that network-accessible is not the same thing as internet-accessible; there is no such thing as a private cloud on a public network.

Resource pooling

Cloud services are provided to multiple tenants (users, applications) by a pool of interchangeable resources. If each tenant needs its own, specific, customised resources, then it's not cloud computing.

Providing on-demand resources with utility pricing can only make economic sense if the resources come from a shared pool. These resources are dynamically assigned and reassigned in order to get optimal use of out of them.

- Storage, processing, memory, etc.
- Location independent
- The customer generally does not know, or need to know the exact physical location of the resources
- For regulatory and architectural reasons, the customer is generally able to specify a general location; such as the country.
    - e.g. can this data be stored outside of the EU?
    - Can the application function well if the web-server is in the EU and the data it uses is in Australia?

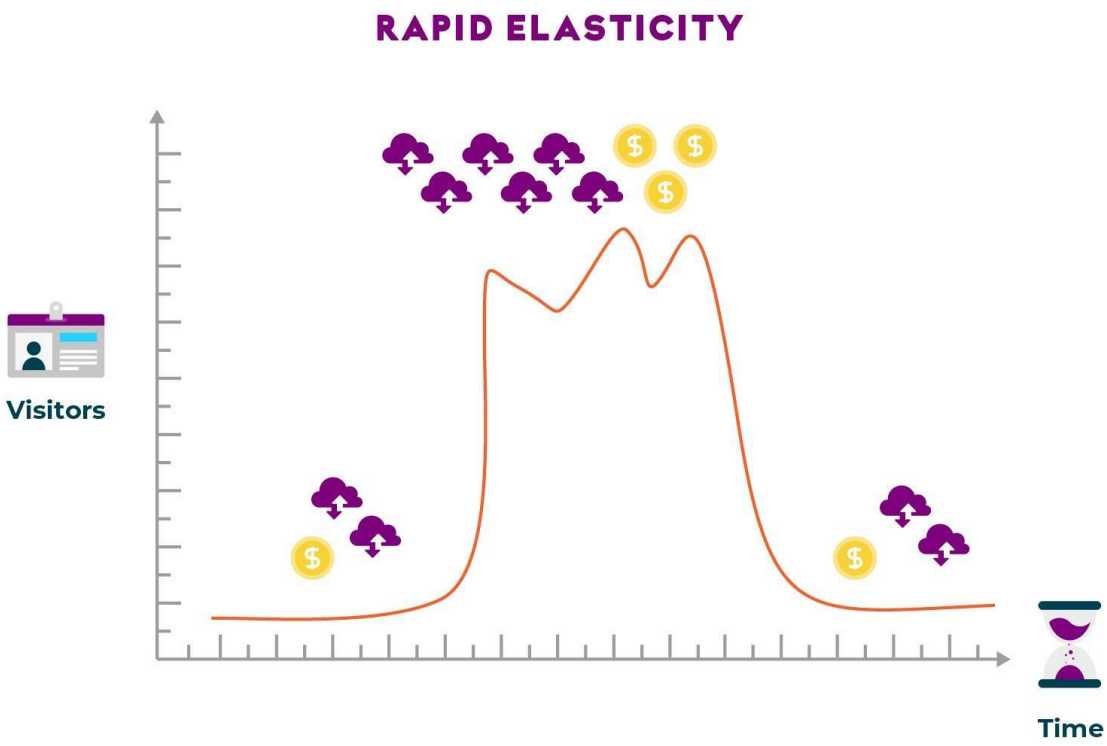
Rapid Elasticity

Elasticity is a fundamental property of the cloud. It is the ability to use exactly the resources you need, without either under-provisioning or excessively over-provisioning, and is one of the key benefits of cloud services.

It allows the customer to dynamically scale computer resources based on demand. There are two ways to scale resources:

- Horizontally (in/out) - adding/removing instances
- Vertically (up/down) - increasing/decreasing resources (e.g. CPU/RAM) for instances

Typically, elasticity refers to scaling horizontally, as it is typically much easier and faster to provision more instances than provide more resources to a machine. Increasing or decreasing resources allocated to a virtual machine instance often requires you to stop the machine for a while and start it again.



Measured services

If resources are being dynamically provisioned, it's essential that the customer should be able to monitor the performance and usage of those resources in real time.

Most cloud resources are offered on a pay-per-use basis, and the customer must be able to monitor their usage in order to control their costs.

Fault tolerance

Fault tolerance exists not only in the cloud but also in the self-hosted environment. What this is referring to is the ability for your application to function even if part of your infrastructure fails.

In the cloud, you have auto-scaling as well as multiple geographical zones to help you aid with fault tolerance. In the self-hosted domain, you would need to configure the infrastructure in order for it to function in case of failure

or maintenance. This could be done with build and orchestration tools that would monitor your resources. These tools check whether your resources are alive and responding or dead and new ones need to be created.

Economies of scale

Cost advantages experienced by companies when the level of output increases are known as economies of scale. This advantage comes from the relationship between per-unit cost and the quantity produced.

Greater quantity produced lowers the per-unit cost. Increase in the output reduces the average costs; this also falls under the economies of scale. The increase is brought by synergies of efficiency and operation.

Economies of scale can be implemented at any stage of the production process. The 'production process' refers to all the activities related to the commodity without the final buyer's involvement.

There are numerous ways that economies of scale can be implemented in the company, such as hiring more experienced marketing employees and automating human labour with machines.

Capital expenditure

Capital expenditure (CapEx) are funds that companies use to upgrade, maintain and acquire physical assets like buildings, technology and equipment.

In short, CapEx is the list of expenses that the company shows on the balance sheet marked as investment, rather than on the income statement labelled as expenditure. It is typically used for new projects.

A company's CapEx can tell you how much they are investing in new or existing assets for growing or maintaining the company.

Operational expenditure (OpEx) is short-term expenses that keep the business running. This may include maintenance costs and utilities: any costs that you pay for over time. CapEx is not OpEx, and should not be treated as such.

Because you don't pay for the physical hardware in The Cloud, moving your infrastructure to the cloud means your CapEx gets transformed into OpEx.

Infrastructure as a service (IaaS), platform as a service (PaaS) and software as a service (SaaS)

The main differences between the three models can be summed up here:

- IaaS gives maximum flexibility for hosting custom-built applications or acting as a data centre for data storage
- PaaS built on top of IaaS reduces the need for administering the system. This frees up time: by not needing to manage infrastructure, time can then be used for development of the application
- SaaS is an out-of-the-box solution ready to be used that meets a specific business use, and is built on top of IaaS and PaaS

Here's a diagram of the different responsibilities between each model:

When picking which model to use, think about how much control you might require for the given task.

This will make it easier to decide which model to select.

Infrastructure as a service (IaaS)

Infrastructure as a service (IaaS) is a computing infrastructure that can be run and managed over the internet. You pay for what you use, and IaaS allows quick scaling up or down of the infrastructure.

One of the most worthwhile reasons that companies migrate to IaaS is that the burden of buying and managing physical infrastructure is eliminated. Rather than installing physical servers, companies can run the infrastructure they need from a cloud provider.

Each service component is sold as a separate service altogether, which allows you to use the specific resources that you require for exactly as long as necessary.

There are three main *IaaS* providers:

- Google Cloud Platform
- Amazon Web Services
- Microsoft Azure

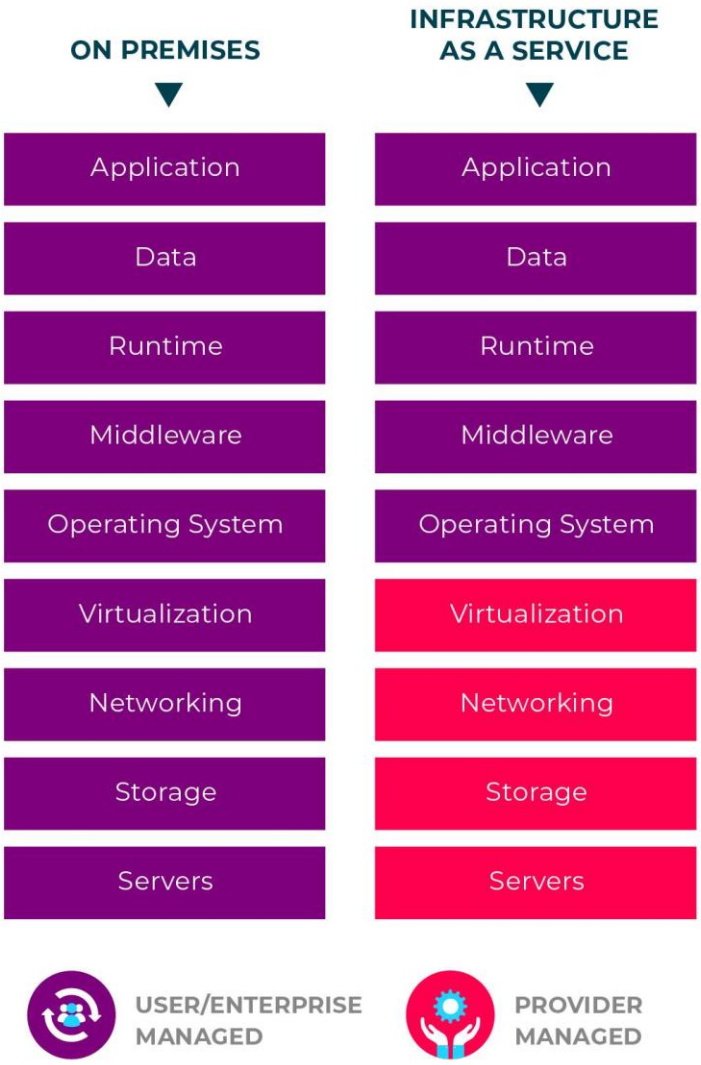
IaaS is typically aimed at developers. For instance, a user would have root access and/or administrator privileges on any provisioned virtual machines (VMs). This means that configuring the operating system (OS), installing applications, etc., becomes possible.

Unlike in traditional systems architecture, the cloud provider takes the responsibility for the infrastructure. However, you are still responsible for:

- Software
    - *Installation*
    - *Configuration*
    - *Management*
- Middleware
- Applications

There is no possibility to have access to the physical resources.

In the following picture, we can see the responsibilities of the vendor and client when comparing IaaS against on-premises hosting:



Note that this diagram does not include the actual physical premises involved.

Business scenarios

Here are some typical business case scenarios where IaaS is applicable:

- Testing and development work. Where environments used for testing can be run and disposed of quickly. Additionally, these environments can be scaled up or down, saving you money
- Website hosting. Hosting a website through IaaS could be less expensive than doing it the traditional way
- Storage, backup and recovery. This takes away the need for data storage to be managed and secured by skilled staff. It also handles unpredictable demand and growing storage needs automatically or manually, and simplifies the management and planning of backup and recovery systems
- Web apps. With IaaS, web apps have the support, storage, servers and networking provided to deploy them. This allows for quick deployment with scalable structure based on demand, which is especially useful when demand is unpredictable
- Performance computing. Performance computing, which requires millions of variables or calculations, is supported, such as for weather and climate simulation
- Big data. Big data is a popular term in the modern IT industry and has enjoyed significant media coverage. It refers to data sets that contain potentially valuable patterns, trends or have any associations. Mining these data sets in order to locate these hidden patterns is computationally intensive, and can be done through IaaS

Advantages

Here are some of the main advantages of IaaS:

- No capital expense and reduction in costs. There are no upfront costs as you pay for what you use, an attractive solution for start-ups
- Business continuity and disaster recovery. Typically in order to achieve high availability, disaster recovery and business continuity requires a significant investment in both technology and staff. With the right Service Level Agreement (SLA) on IaaS you could reduce the costs while achieving the previously mentioned functionality
- Rapid innovation. The infrastructure is ready in minutes or hours for you to launch a product, compared to days or weeks when done internally
- Rapid response. You are able to respond quickly to demand with scaling up and down, such as scaling up to accommodate holiday spikes, and scaling down to save money
- More focus on business. Time saved on not requiring to care for the IT infrastructure can be spent on other business needs
- Stability, reliability and supportability can be achieved with the right SLA with the provider. Things like software updates, hardware, troubleshooting or equipment problems could be taken care of by the provider, freeing up the time to focus on the business
- Security through the service provider can be better in most cases than what you may be able to achieve in-house, though this will depend on the SLA you choose

Platform as a service (PaaS)

Platform as a service (PaaS) is an environment in the cloud with all the resources you need for development or deployment of cloud-based applications.

The resources are purchased on a pay-as-you go basis, where access to them is granted through a secure internet connection.

PaaS is mainly targeted at developers.

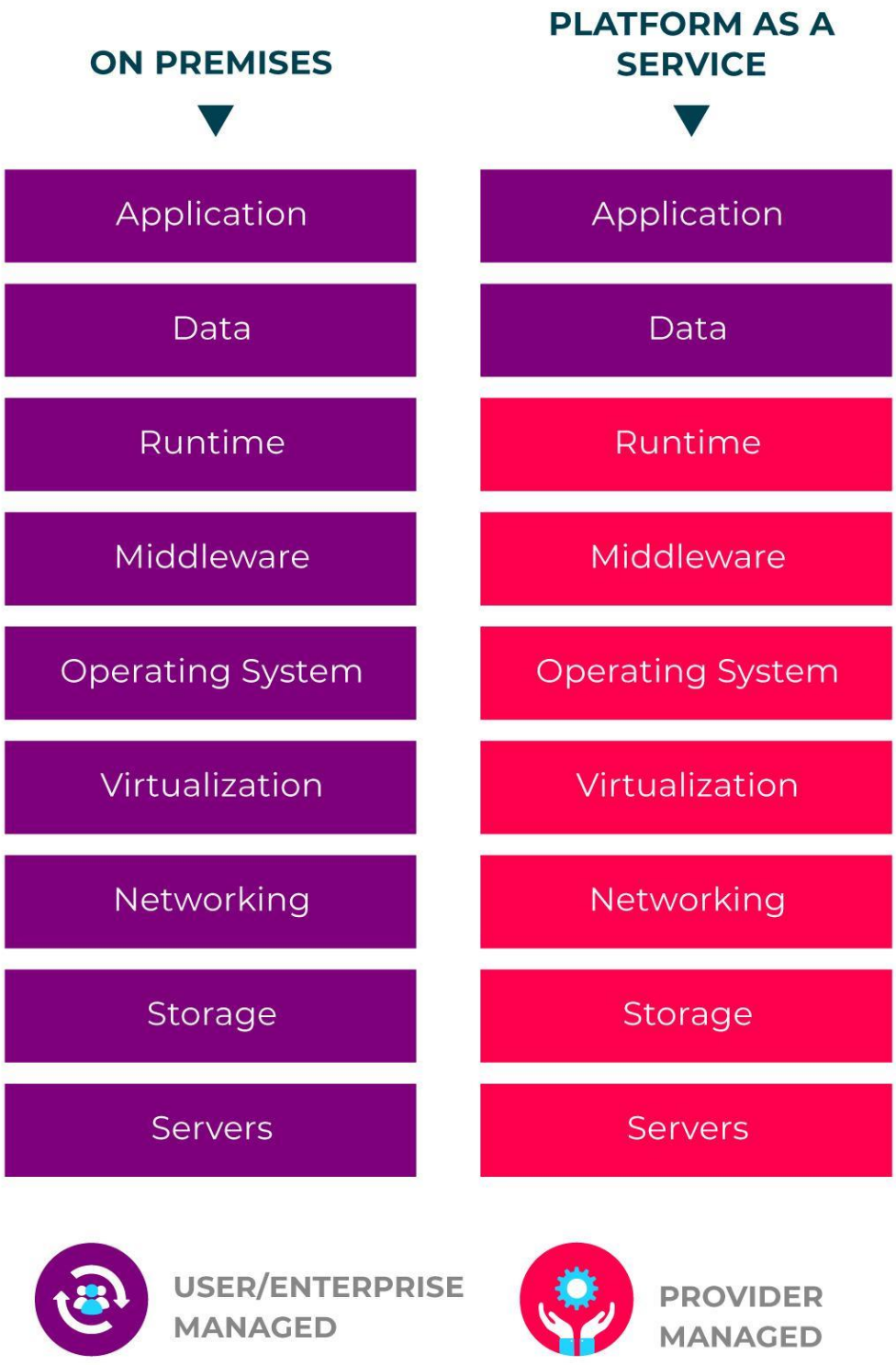
PaaS is a stage above infrastructure as a service (IaaS). It includes all *IaaS* services, with the addition of:

- middleware
- development tools
- business intelligence services
- database management systems

With *PaaS* you would avoid the hassle of paying for and managing:

- software licenses
- application infrastructure
- middleware
- container orchestration
- development tools

The applications and services that are being developed remain the responsibility of the user, while everything else is the responsibility of the cloud provider.


ON PREMISES
PLATFORM AS A SERVICE
Application
Data
Runtime
Middleware
Operating System
Virtualization
Networking
Storage
Servers
Application
Data
Runtime
Middleware
Operating System
Virtualization
Networking
Storage
Servers
USER/ENTERPRISE MANAGED
PROVIDER MANAGED

Business scenarios

Typical use case scenarios for *PaaS* are:

- Development framework. A framework is provided for the development of cloud applications. Applications can be created through the built-in software components, with features such as scalability, multi-tenancy, and high-availability included and managed by PaaS
- Business intelligence and analytics. Tools are provided for data mining, finding patterns, investment returns and for many more business-related needs
- Additional services. There are services that provide advice in regards to networking, security, scheduling, etc.

Advantages

As PaaS is built on top of *IaaS* it offers all of the advantages of IaaS, as well as several of its own:

- Improving coding efficiency through the use of in-built tools
- Providing development options for multiple platforms
- Sophisticated tools for development, business intelligence and analytics through the pay-as-you go model allows for even the smallest companies to utilise them and therefore save time
- Remote teamwork is available, as all the resources are accessible over the internet
- All the capabilities which exist in a typical development environment life cycle are included, namely building, testing, and deployment

Software as a service (SaaS)

Software as a service (SaaS) allows everyone to connect to cloud-based applications over the internet.

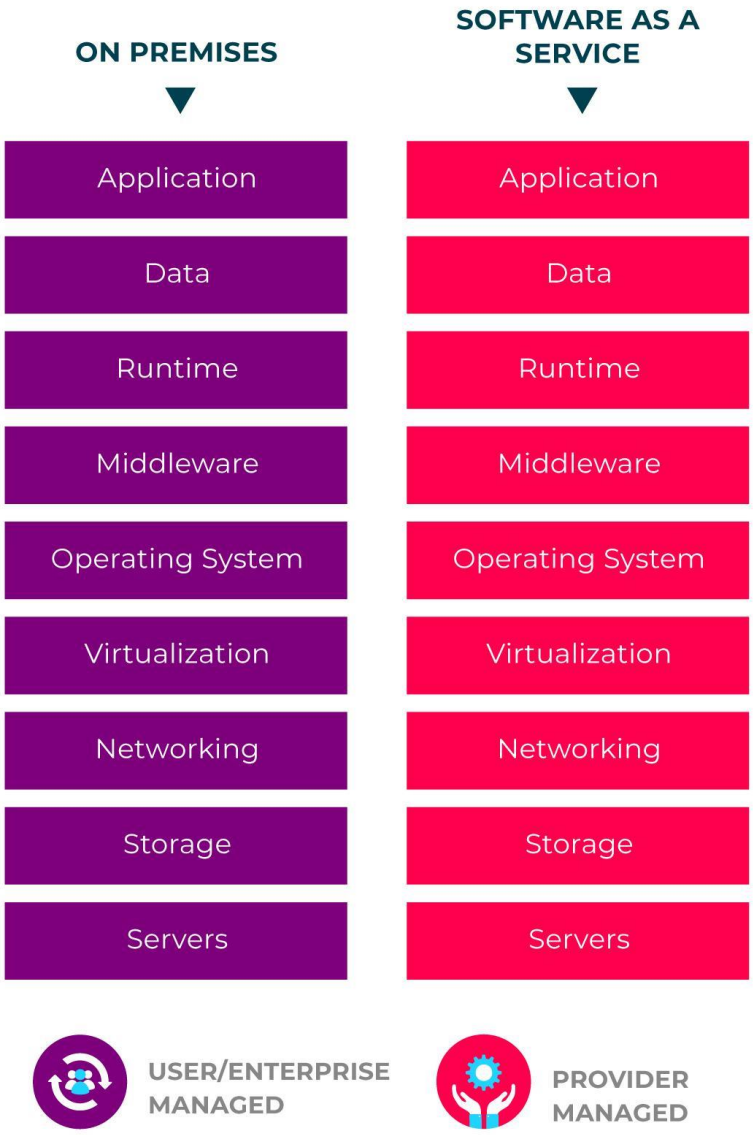
Common examples of SaaS applications would be:

- Gmail
- Netflix
- Office 365

Much like the other cloud service models, *SaaS* is a pay-as-you-go service which allows you to obtain a complete software solution from a chosen cloud provider.

In simple words, you are able to rent an application, which you or your users are then able to access through the internet. Everything from infrastructure to data is stored in the cloud provider's data centre.

The cloud service provider is responsible for managing all aspects of the service that is provisioned, including availability, hardware, software, security, data, etc.



Business case

One of the most applicable examples is the email service you use. It is likely that this application is a SaaS.

The service enables you to log on and access your emails from anywhere in the world, whether it be from a browser, mobile app, tablet, or any other Internet-enabled device. The software itself is stored by the provider in a data centre alongside your data.

The actual place it is stored is not important, and you as the user need not be concerned; the important thing is that the application runs and can be used from anywhere.

This example refers more to a personal use case.

A business case is similar, where you gain access to services that allow your team to collaborate, use emails, send messages, store data, manage calendar events, etc.

A more advanced application for business purposes would likely include services for customer relationship management (CRM) and enterprise resource planning (ERP).

Here, the business case is more likely to be a subscription-based service, or based on some level of usage.

Advantages

Here are some of the advantages of SaaS:

- There is no need to purchase or maintain:
    - hardware
    - software
    - middleware
        - applications like *ERP* or *CRM* become more available, as savings gained from not needing to use in-house software/hardware/middleware can be used for these services instead
- automatic scaling ensures that, based on your usage, you only ever spend for what you use
- the applications used may not need installing, as the majority of them can be accessed through a web browser
- mobile device support
- no data is lost if the user's personal machine fails, as everything is kept in the cloud and made readily available over the Internet

Source control

Source control is known by a few different names, such as version control and revision control. It is the practice of tracking and managing changes to code.

Source control is vital for managing software development projects. Being able to track changes to code allows developers to:

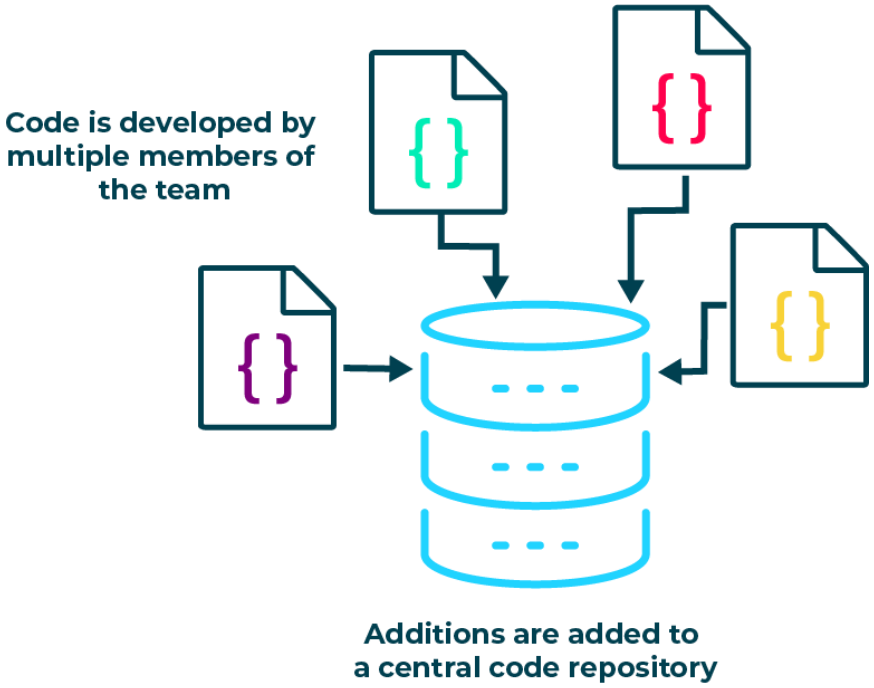
- centralise all code changes and additions to one code repository
- allow for simple and effective collaboration within development teams
- control the integration of new code into the codebase
- track changes from the entire team over the full lifetime of the project
- revert code back to previous versions

This module covers the benefits of using source control for software development, how we make use of source control using source control management systems, and finally some of the most common SCM tools that are out there.

Source control management

We use source control management (SCM) systems to implement source control practices. These provide the ability to:

- Store code in a central repository
- Track changes over time
- Create code branches so that additions are made in isolation from stable code
- Merge new code into a stable release branch, known as the master branch
- Integrate with CI/CD automation tools (such as Jenkins and CircleCI) so that code will be built and tested as it is generated and pushed to the repository



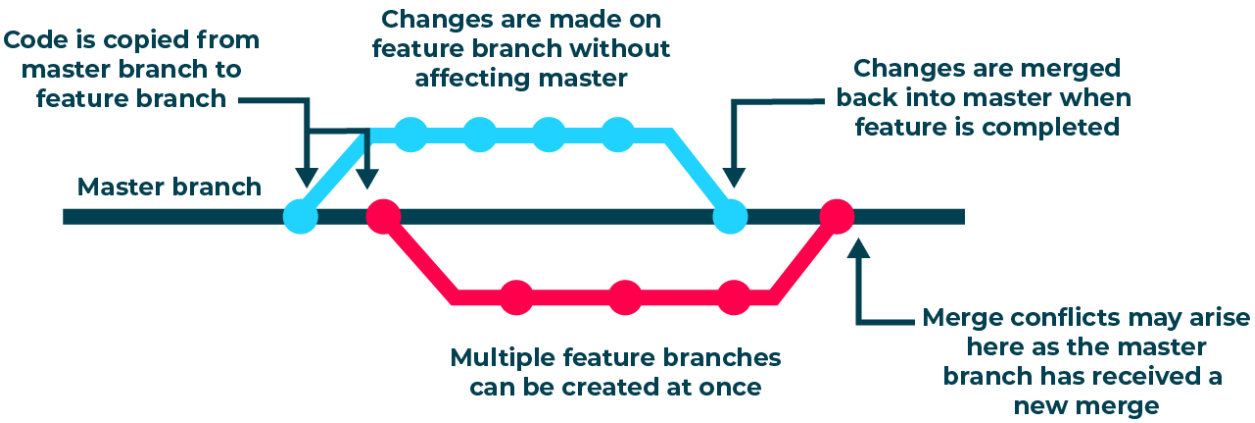
Repositories

To keep code content in one place, SCM systems store code in a repository. This essentially functions as a big bucket for all your code to live in.

Having one place where your code resides means that versioning stays consistent, regardless of who's working on it.

This is vital when developers are working on the same project in tandem with one another. If there was no repository, there would be no cohesion between each member's varying versions as they made changes to the code base.

Branching



Without source control, having multiple developers developing on the same code base at the same time will result in code conflicts and breakages to functionality, as two developers may need to work on the same parts of code to create their particular features.

Branching allows for each developer to work on new features in isolation by creating a version of the source code that only they are working on.

This new version is called a feature branch, while the source code lives on the master branch.

Once changes have been made and the feature implemented, the feature branch code is merged back into the source code on the master branch.

The SCM system will automatically detect the differences between the feature and master branches and alert the developer if there are any conflicts.

Conflicts may arise if changes have been made to the master branch after the feature branch has been created. The SCM will clearly show the changes that need to be made before merging can occur.

Code tracking

Code tracking allows development teams to keep track of all changes made to a project over time. This allows for greater organisation, as all additions to code are fully documented and attributed to the developer who made them.

Problems can then be solved with ease as code contributors can be consulted directly regarding their code. If a contributor is no longer a part of the team and can't be consulted directly, their changes are still fully documented, allowing the team to track their contributions despite their absence.

If a feature causes a bug in the software after it is merged to master, code tracking makes it easy to revert the master branch to a previous stable state until fixes can be made.

SCM tools

Some common SCM tools are:

- Git
- Mercurial
- Subversion (often abbreviated to SVN)
- CVS
- Perforce

Git is by far the most common SCM system used in software development. Git is a free and open source version control system.

Its ubiquity is largely due to how easy it is to learn and its tiny footprint, which is a result of being written in low-level C code.

It is a decentralised SCM tool, meaning its operations are largely performed on your local computer and requires no communication with an external server, resulting in very fast performance.

Repository hosting services

There are several web-based version control repository hosting services out there, most of which either utilise or are wholly based upon Git, such as:

- GitHub
- GitLab
- Bitbucket
- SourceForge
- Launchpad

Many cloud providers have their own code repository offerings which offer simple and powerful integration with other cloud services:

- AWS CodeCommit
- Azure Repos (as part of Azure DevOps)
- Google Cloud Source Repositories

Git basics

Git is used for tracking changes in source code during software development.

It is designed for coordinating work amongst programmers, but it can be used to track changes in any set of files.

Basic workflow

The basic workflow for using Git includes staging, committing and pushing changes.

Before a change can be committed it must be staged and to apply your changes for everyone else on the team, the changes must be pushed to the remote repository.

Common commands and concepts

Cloning a repository

To download a remote repository, you can use the git clone command and provide the URL of the remote repository.

The clone command is very useful because it configures the local repository for you, the remote repository is automatically configured for when you need to push your new changes to it.

Command	Function
`git clone [REPO_URL]`	Clone (download) a remote repository to the current working directory

Staging a change

Staging is the step that you must take before committing a change.

Staging is a feature in Git that enables the developer to choose what changes are actually going to get committed to the repository when the commit is made. There are a few ways you stage files:

Command	Function
`git add --all`	Stage all files
`git add .`	Stage all files (only if you are at the root of your project)
`git add file_1.txt file_2.txt`	Stage specific files
`git add *.txt`	Stage all files with a specific ending

Username and email in Git Config (git config)

Before you can commit changes to the repository you need to have your username and email configured.

This can either be set in the scope of the repository you downloaded or set globally, so that it does not need to be configured for any other repository that you clone.

The information that you enter will get tied to the commit and this task doesn't need to be repeated every time you want to make a commit.

Command	Function
`git config --global user.name [YOUR_USERNAME]`	Set login username globally
`git config --global user.email [YOUR_EMAIL]`	Set login email globally
`git config user.name [YOUR_USERNAME]`	Set login username for local repository
`git config user.email [YOUR_EMAIL]`	Set login email for local repository

Local repository status (git status)

Knowing the current state of your local repository is very useful so that you can understand what commands to run.

For instance, how can you know what files have been staged and not staged?

Command	Function
`git status`	View staged files

Committing a change (git commit)

When you make a commit to a Git repository, you are effectively 'saving' the changes that you have staged to the repository.

Unlike saving files in most other programs, Git also requires a message to be saved against the commit along with some basic information about the user from the Git config shown above.

What you put in this message is important, so that you understand what it is that you changed on that particular commit.

Commits can be reverted, so it helps when there is a concise message about what was implemented or removed.

Command	Function
`git commit -m "initial commit"`	Commit new changes

Pushing changes

To apply the changes that you have made in a remote repository, you must push them to that remote repository.

Only changes that have been committed will be pushed to the remote repository.

We can use git push and provide the remote repository (default is origin) and remote branch to push our changes.

Command	Function
`git push -u [REMOTE] [REMOTE_BRANCH]`	Push committed changes to remote repository
`git push -u origin master`	Push committed changes to the master branch of the origin

Retrieving remote changes

Because Git is a collaborative tool, other people could have made changes to code in the remote repository, these changes will need to be pulled down to avoid conflicts in the code.

Command	Function
`git pull [REMOTE] [REMOTE_BRANCH]`	Pull new changes from a remote repository
`git pull origin master`	Pull new changes from the master branch of the origin

Git branching

Branching in Git helps us to define workflows that make sure the code that is being delivered is in the best state possible, minimising risks of any errors or crashes.

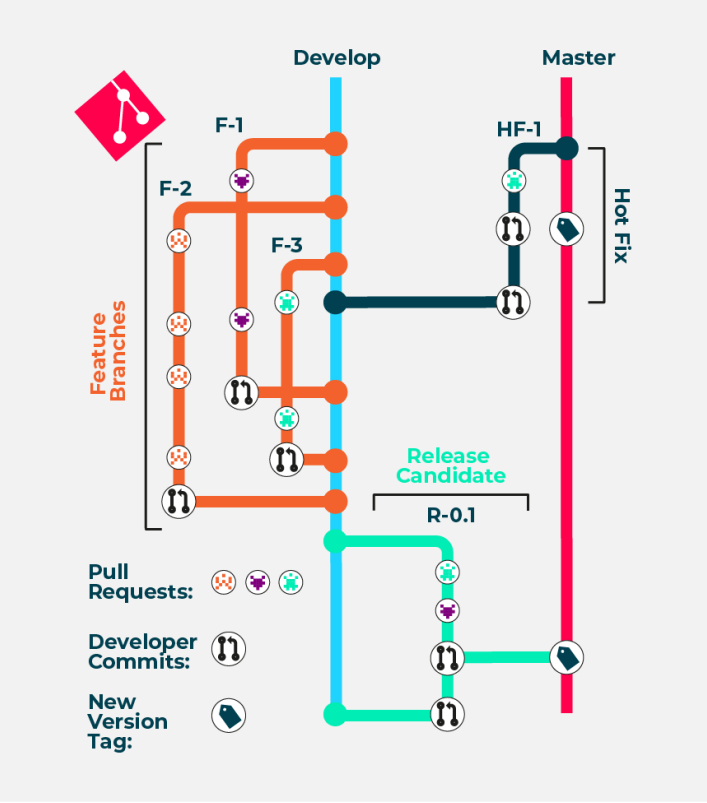
With version control systems, like Git, we can separate the codebase onto many different branches.

This feature can be utilised for isolating the development and testing of new features from working code that is running on a production environment.

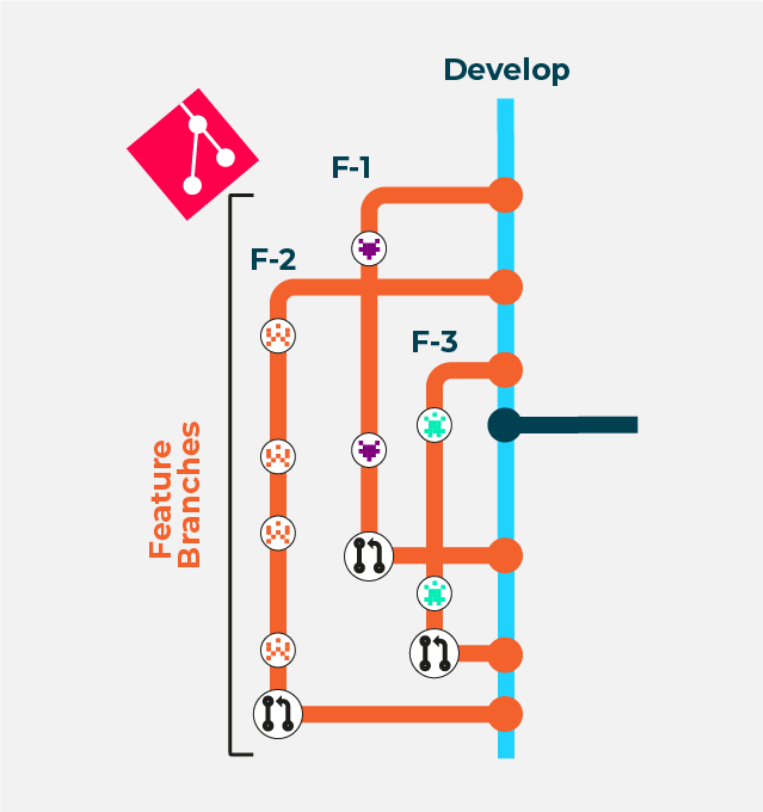
Git branching workflow example

This is an example of a workflow using Git with some notes below which explain in more detail the processes.

The two main branches that exist are the develop and master branches.



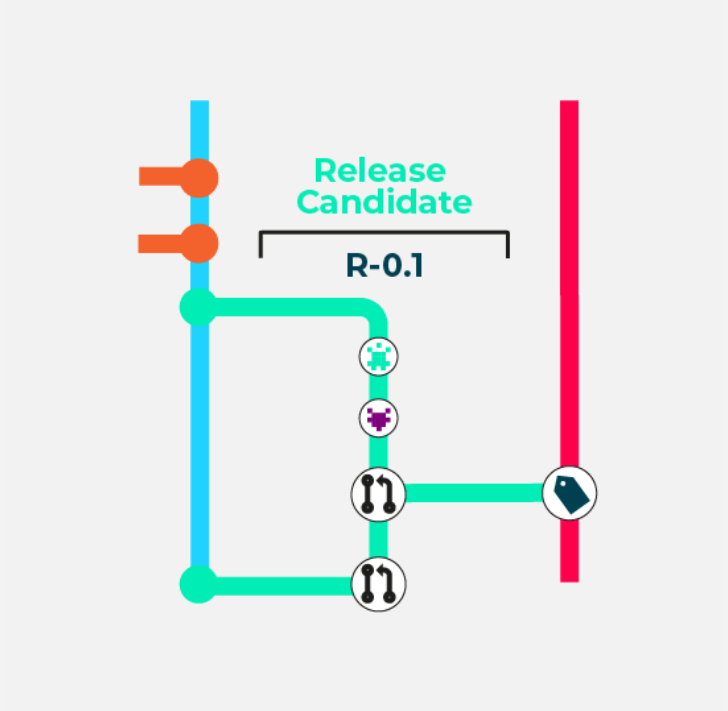
New application features



Conventionally, new features are to be created in **feature branches** from the **develop** branch, and, once the feature has been developed, the code can be reviewed by a peer in a **pull request** and deployed to a test environment for **integration** or **user acceptance testing**.

If all testing and reviews pass, the **pull request** can be approved and merged into the **develop** branch.

Releases



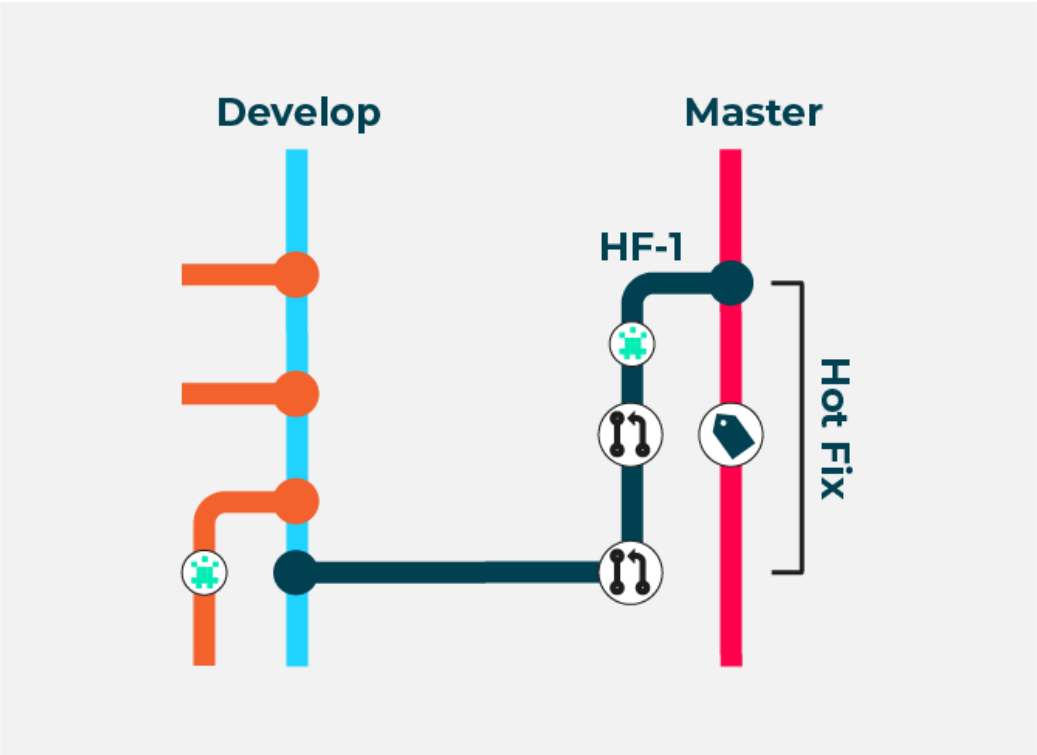
When there is a release approaching, a release candidate branch can be made.

On this new branch, further testing of all the new features working together can take place, and more candidates can be made on this branch to amend any issues.

Once testing has passed and the release has been signed off by the individual accountable for releases, the release candidate branch can be merged into master for the release to be deployed to a production environment.

Once merged into the master branch, the code can be tagged or marked as a release on the Git service that you are using.

Changes must also be merged back into the develop branch, so that any changes that were made to the release candidate will also be included in future releases.



Hotfixes

Hotfixes are changes to code made straight to the master branch. They occur when a bug has been found in the released code and requires a quick fix.

One of the main reasons for designing a workflow like this is to prevent hotfixes, as bugs should be caught in the testing stages, and so should be avoided wherever possible. However, problem code will slip through the cracks eventually, thus hotfixes are inevitable.

A hotfix can be conducted by creating a hotfix branch from the master branch and applying the changes on that branch. Before merging back into the master branch all the changes should, of course, be tested and reviewed to avoid even more hotfixes!

Once merged into the master branch, the changes must also be merged back into the develop branch; this will keep any important changes the hotfix made, and the code in the master branch should be tagged.

Creating and deleting branches

Command	Function
`git branch`	Show all branches in current repository
`git branch [NEW_BRANCH_NAME]`	Create a new branch
`git checkout [BRANCH_NAME]`	Switch to a branch
`git checkout -b [BRANCH_NAME]`	Switch to a branch; create branch if it doesn't exist
`git branch -d [BRANCH_NAME]`	Delete a branch
`git push --delete origin [BRANCH_NAME]`	Delete a branch in your remote repository

When you run a command to create a new branch, it will be branched off of the branch you are currently working on. For example, if you want to create a new feature branch off of the develop branch, you should run git checkout develop to switch to develop first before creating the feature branch with git branch feature.

When you have finished working on your branch, and your code has been merged to master, it is good practice to delete the branch. You will also need to do this on your Git service, which you're likely using (such as GitHub). This is usually done when closing a pull request.

Merging branches

Joining the history of two or more branches through git merge incorporates changes into the current branch. This command is used by git pull to incorporate changes from a different repository, as well as to merge changes from one branch into another.

If we assume that we have a history like this, and the current branch is master:

```
      E---F---G issue
     /
A---B---C---D master
```

Executing git merge issue would add changes E, F and G into master and result in a new commit.

This new commit would end up in the following history.

```
      E---F---G issue
     /         \
A---B---C---D---H master
```

Merge conflicts

Merge conflicts happen when more than one person has edited a file, and the line numbers that were edited are the same. It can also happen if someone deleted a file that another person was working on.

Say the file hello.py in this repository as this code in it at commit B in our repo's history:

```
hello = "Hello?"

print(hello)
```

Next, let's say that hello.py was changed at commit D on the master branch and separately at commit G on the issue branch like so:

app.py at commit D	app.py at commit G

`hello = "Hello World!"`
 	
`print(hello)`
	`user = "Harry"`
	`hello = f"Hello ${user}"`
	
	`print(hello)`

When the issue branch is merged into the master branch at commit H, the developer will be informed of a merge conflict and will have to specify which lines of code they want to keep after the merge takes place.

This conflict only affects the person performing the merge, and the rest of the team wouldn't be affected by it.

If a merge conflict happens, *Git* will automatically halt the merge process and mark the file, or files, that are conflicting. It is then up to the developer to resolve them.

Reverting

Reverting in Git is a forward-moving way of undoing changes on a repository.

Reviewing the history of a repository

The following commands will allow you to view the commit history of a repository from the command line.

Command	Function
`git log`	Shows the commit history of the current branch
`git log [BRANCH_NAME]`	Passing an argument through allows you to display the commit history of a specific branch
`git log --oneline`	Shows a simplified view of the commit history
`git log --branches=*`	Shows the commit history for all branches in the current repository

When you run `git log`, output should look similar to this:

```
commit 0cf7b874ada29ad4907476afda1bb845a3be39ab (HEAD -> master, origin/master, origin/HEAD)
Author: htr-volker <htr.volker@gmail.com>
Date:   Thu Jul 23 10:05:35 2020 +0100

    another new commit

commit 53e1dbbdc34d79c599c4925a0dc6aa4fb9d5a35d
Author: htr-volker <htr.volker@gmail.com>
Date:   Thu Jul 23 09:54:41 2020 +0100

    little comment addition

commit c263f9e4ce03319e4139ec510f59cfe86b39b418
Merge: 6031e56 e057e3f
Author: htr-volker <57865118+htr-volker@users.noreply.github.com>
Date:   Thu May 14 10:18:39 2020 +0100

    Merge pull request #8 from htr-volker/readme-cleanup

    Update README.md
```

Reverting to a previous commit

Occasionally, we may commit a change we didn't mean to. Thankfully, we can undo commits in Git. There are two commands you can use to revert a repository to a previous state:

Command	Function
`git revert [COMMIT]`	Creates new commit that reverts changes to the code its state at the specified commit
`git revert HEAD`	Creates a new commit that reverts the repository back to the state of the previous commit
`git reset --hard [COMMIT]`	Reverts changes to the code to the state at the specified commit and discards the history after that point

To specify which commit we want to revert/reset back to, we need the commit's hash code. We can view this by running `git log --oneline`.

Let's say your commit history looks like this:

```
875f31e (HEAD -> master) fourth commit
 483856a third commit
 2dd011d second commit
 bcabb84 first commit
```

If we execute:

```
$ git revert bcabb84
```

Git will create a new commit that will undo all of the changes performed in the second, third and fourth commit.

If we just wanted to go back one commit stage, we could instead enter:

```
$ git revert HEAD
```

The history will still contain the entire commit history, but it's changes will be undone. Using revert allows us to use the same branch and is considered the better solution for reverting.

Reverting with reset

Instead of using git revert in the situation above, we could have used:

```
$ git reset --hard bcabb84
```

This command will return the state to the selected commit (bcabb84 first commit).

The difference between this and git revert is that now the Git history will no longer contain the fourth commit, and work would continue as if the fourth commit never happened.

However, not having the commit reflected in the commit history can cause complications when working with a shared remote repository.

If the reset happened to a commit that is already shared with others, and we tried to push some changes afterwards, Git would throw an error. This is because it would think that our local Git history isn't up to date. In these scenarios, it's more appropriate to use the revert strategy.

Using revert to go back to the latest commit

If you have made a lot of changes and just want to back to the latest commit that was made then you can use reset without specifying a SHA1:

```
git reset --hard
```

Forking

In layman's terms, forking is creating a copy of an existing repository under your account.

Forking a repository allows you to freely experiment with the existing code base without the fear of breaking the project.

Forking also gives you the ability to contribute to a project by creating additional functionality, without having to be made a collaborator by the original developer.

Proposing a change

Imagine you are using someone's project and you find a bug. Usually, you would raise an issue for this; however, forking means you would be able to attempt a fix yourself.

The steps would be:

1. Fork the repository.
2. Create a new branch.
3. Fix the issue.
4. Submit a pull request to the owner of the original project.

If the owner approves your fix, your work should then be merged into the original repository.

Your idea

If a project is under public license that allows you to freely use it, you could use the project as a starting point for your own project.

For example, if you found a web application that you liked, and it was under the public license, you could fork the project and add any additional functionality. This would then be yours to freely use or distribute.

Pull requests

Pull requests let you tell others about changes you've pushed to a GitHub repository.

Once a pull request is sent, interested parties can review the set of changes, discuss potential modifications, and even push follow-up commits if necessary.

Collaborative tool

Pull requests are commonly used by teams and organisations collaborating using the shared repository model.

This is where everyone shares a single repository, and separate (topic) branches are used to develop features and isolate changes.

Many open-source projects on Github use pull requests to manage changes from contributors, as they are useful in providing a way to notify project maintainers about any changes made.

Once the project maintainer has been notified of a change via a pull request, it opens the door for code review and general discussion about a set of changes.

Once a pull request has been opened, it can be reviewed by other collaborators and merged into the base branch you dictated earlier.

Creating a pull request

There are two main flows when dealing with pull requests (PRs):

1) PR from a branch within a repository

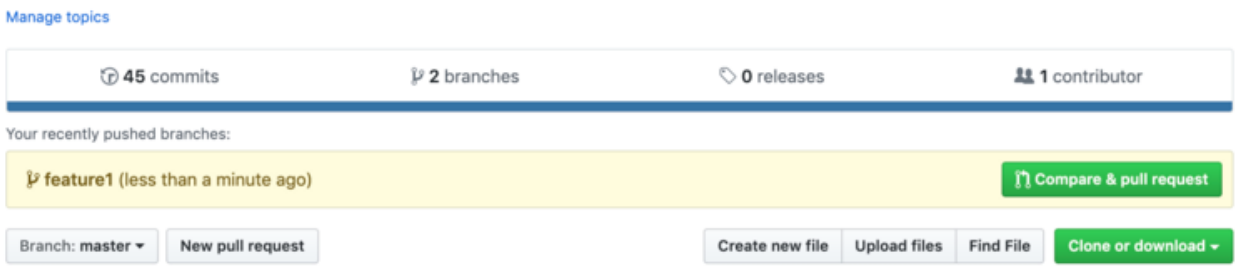
Create a feature branch and push it to VCS

1. Clone a repository down using `git clone [URL to repository]` and `cd` into it.
2. Create and switch to a new (feature) branch using `git checkout -b [new branch name].`
3. Make changes on your feature branch, and then use `git add` and `git commit` to stage your changes. You will need to configure your VCS email and username at this point, using `git config` (if you haven't already).
4. Push the feature branch up to git using `git push origin [new branch name].`
5. Your new branch should now be reflected in your VCS.

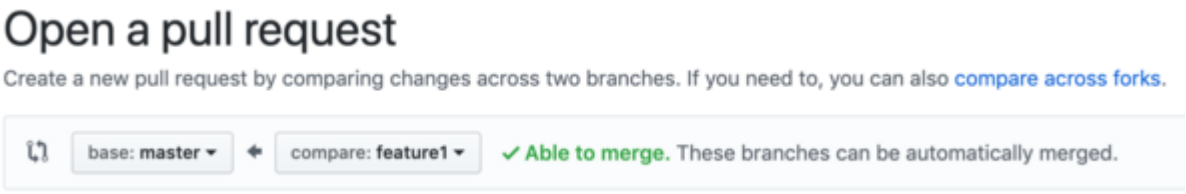
Create a pull request

This is done on your VCS GUI (in this example, we are using GitHub):

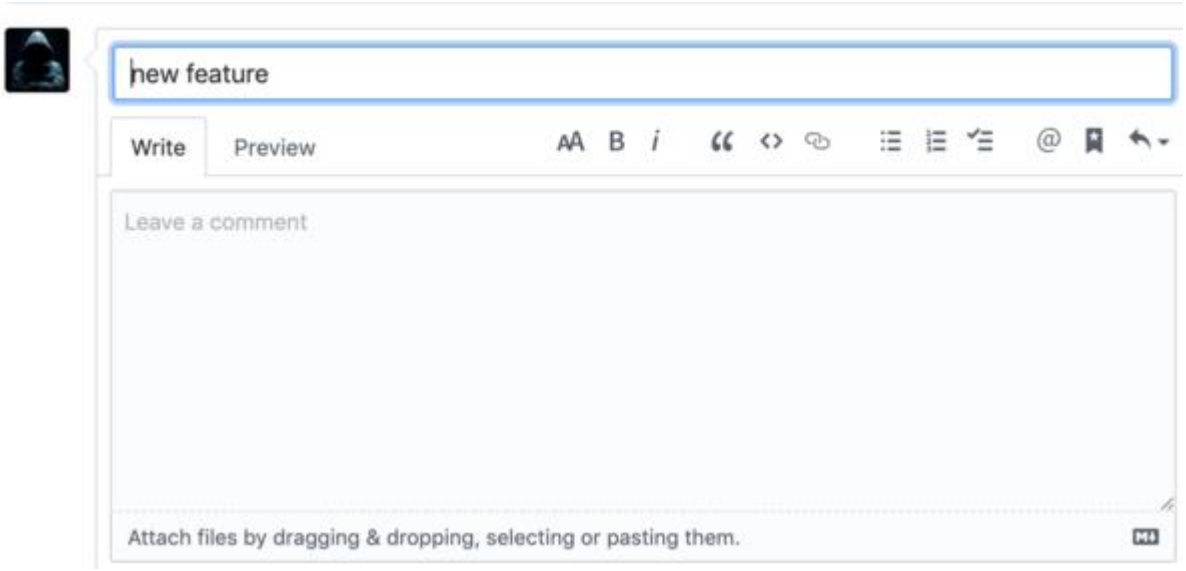
1. Go to the repository you're working with and click on the 'Compare and pull request' button:



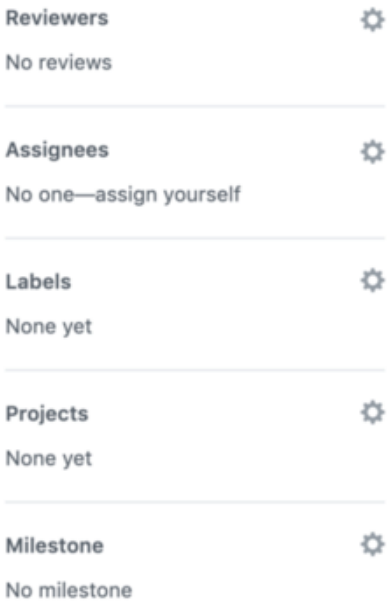
2. You will need to choose which branch you want the changes to eventually be implemented on (`base`), and which branch the proposed changes are currently on (`compare`):



3. At this point, you can give your pull request a title and add some comments, for context:



4. You can configure additional options on the right-hand side, such as assigning a reviewer, adding a label, etc:



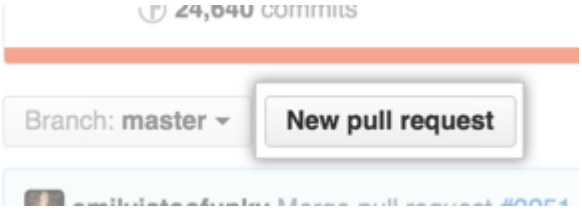
5. Click Create pull request:

2) PR from a forked repository

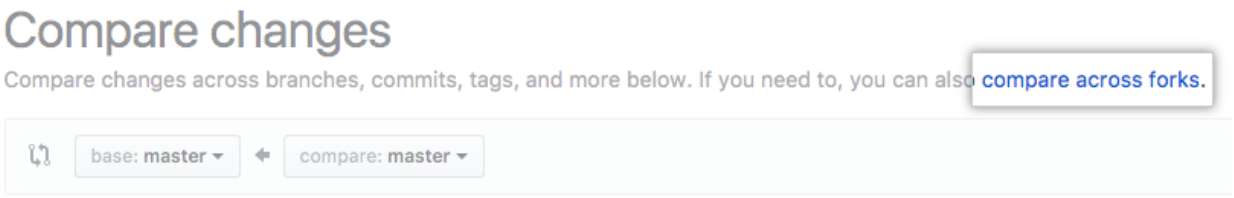
Create a pull request

This is also done on your VCS GUI:

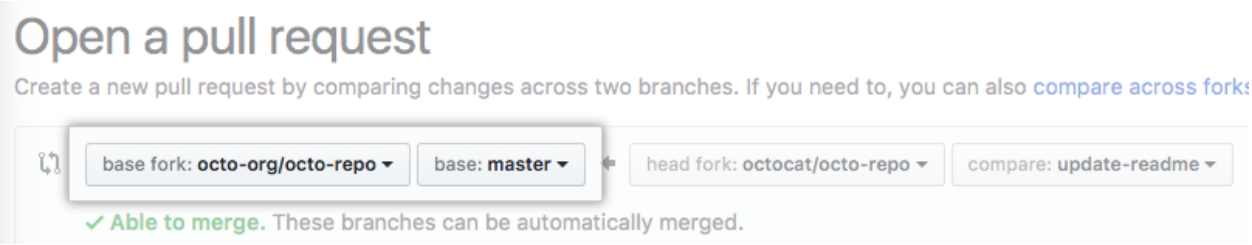
1. Ensure you have a forked repository, and have made some changes to it.
2. Navigate to the repository from which you created your fork.
3. Click on 'New pull request':



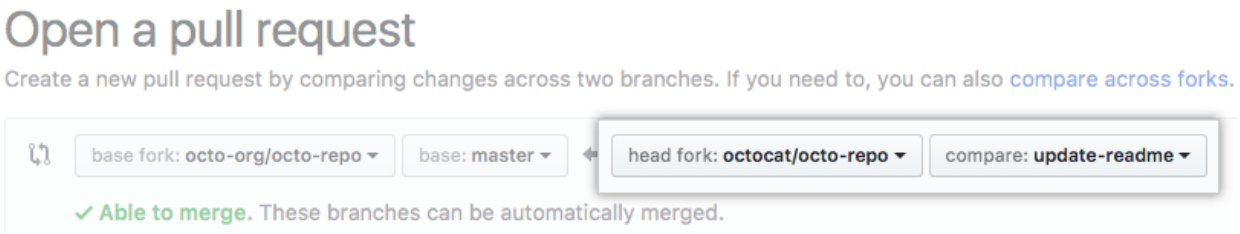
4. Amend your `base` and `compare` to reflect to correct branches.
5. Click on 'compare across forks':



6. Confirm that the base fork is the repository you'd like to merge changes into. Use the `base` drop-down menu to select the branch of the repository you'd like to merge changes into:



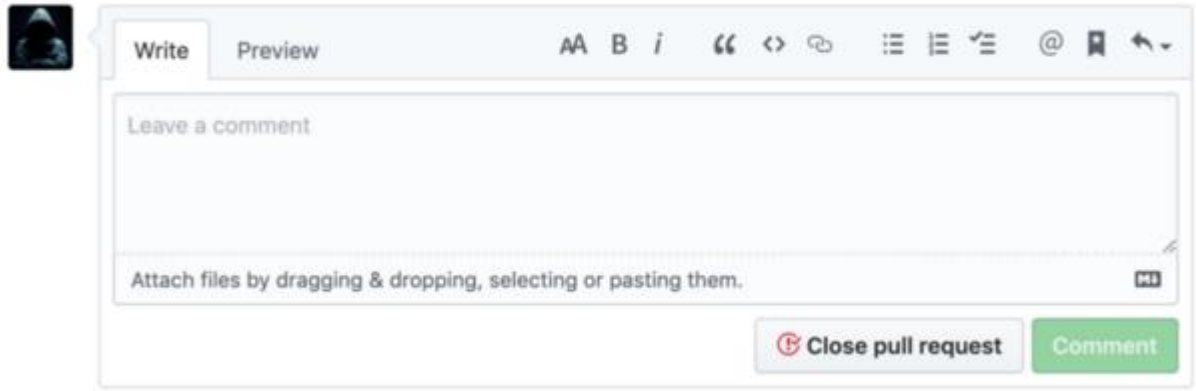
7. Use the head fork drop-down menu to select your forked repository, then use the compare branch drop-down menu to select the branch you made your changes in:



8. Insert a title and description for your pull request.
9. Configure any other options you want on the right-hand side.
10. Click 'Create Pull Request'.

Closing a pull request

You can simply click on the ‘Close’ button on the pull request page to close it:

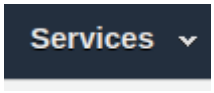


You will be given the option to delete the branch directly at this point, should you wish to do so.

Security Groups

When using a virtual machine in the cloud, by default all the ports are closed. So, if we want to access our machine by SSH or through the browser, we need to open some ports. We do that with a security group.

In the AWS console go to Services.

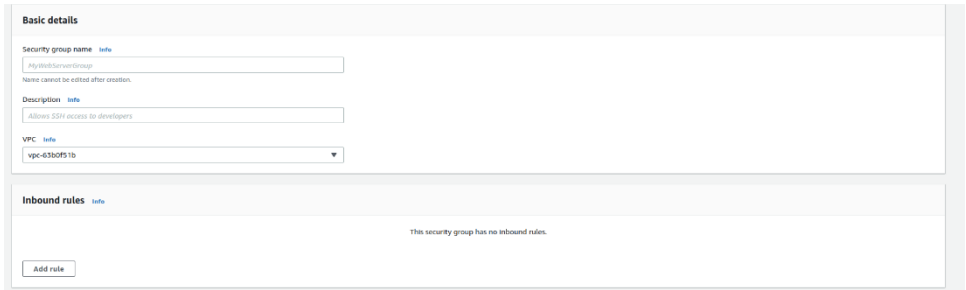


And select EC2.

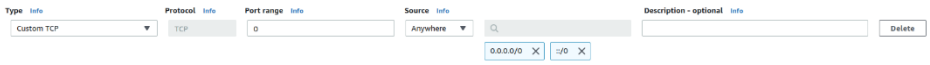
Now, on the left-hand side select Security Groups under Network and Security.

Now on the security group dashboard, select Create new security group.

Your page should look like this.

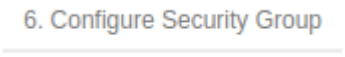


Now give the security group a name and description. Then under ‘inbound rules’, add a rule and enter the port you wish to open and ‘Anywhere’ in the source section.

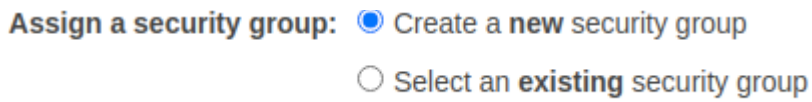


You can add as many rules as you like for different ports.

Now when creating a VM, under ‘Configure Security Group’:



Tick ‘Select an existing security group’



Then select the security group you just created.

You can then ‘Review and Launch’ the instance and it will have the ports you specified open.

Jenkins CI/CD

Installation

Jenkins is a self-contained, open-source automation server that can be used to automate all sorts of tasks related to building, testing, and delivering or deploying software.

Jenkins can be installed through native system packages, Docker, or even run standalone by any machine with a Java Runtime Environment (JRE) installed.

Install Script

Please note, if you are using the script then you will need to have sudo access on the machine you execute it on.

Simply put this script in a file on the machine you wish to install it on and execute it:

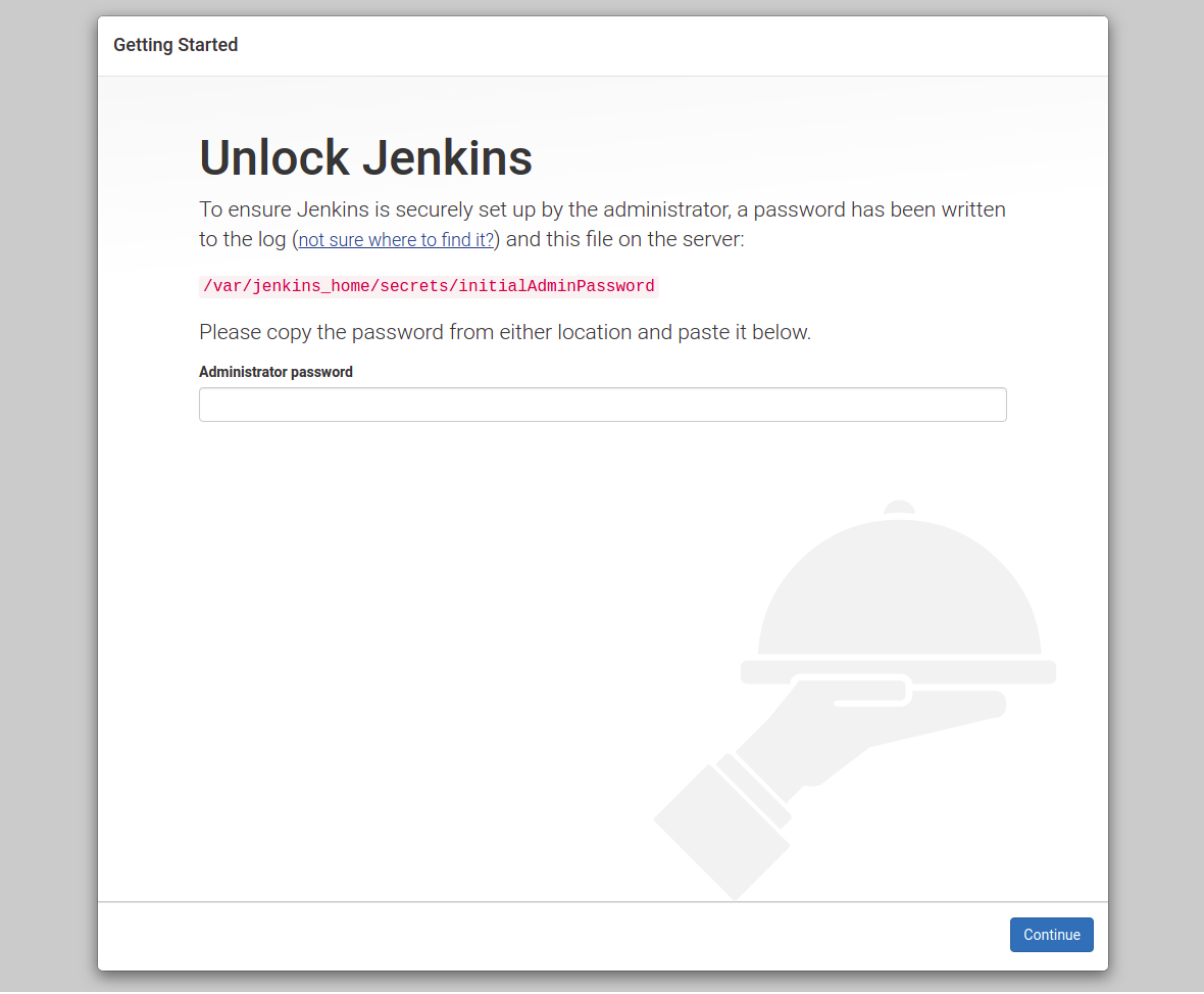
```
#!/bin/bash
if type apt > /dev/null; then
    pkg_mgr=apt
    java="openjdk-8-jre"
elif type yum /dev/null; then
    pkg_mgr=yum
    java="java"
fi
echo "updating and installing dependencies"
sudo ${pkg_mgr} update
sudo ${pkg_mgr} install -y ${java} wget git > /dev/null
echo "configuring jenkins user"
sudo useradd -m -s /bin/bash jenkins
echo "downloading latest jenkins WAR"
sudo su - jenkins -c "curl -L https://updates.jenkins-
ci.org/latest/jenkins.war --output jenkins.war"
echo "setting up jenkins service"
sudo tee /etc/systemd/system/jenkins.service << EOF > /dev/null
[Unit]
Description=Jenkins Server


[Service]
```

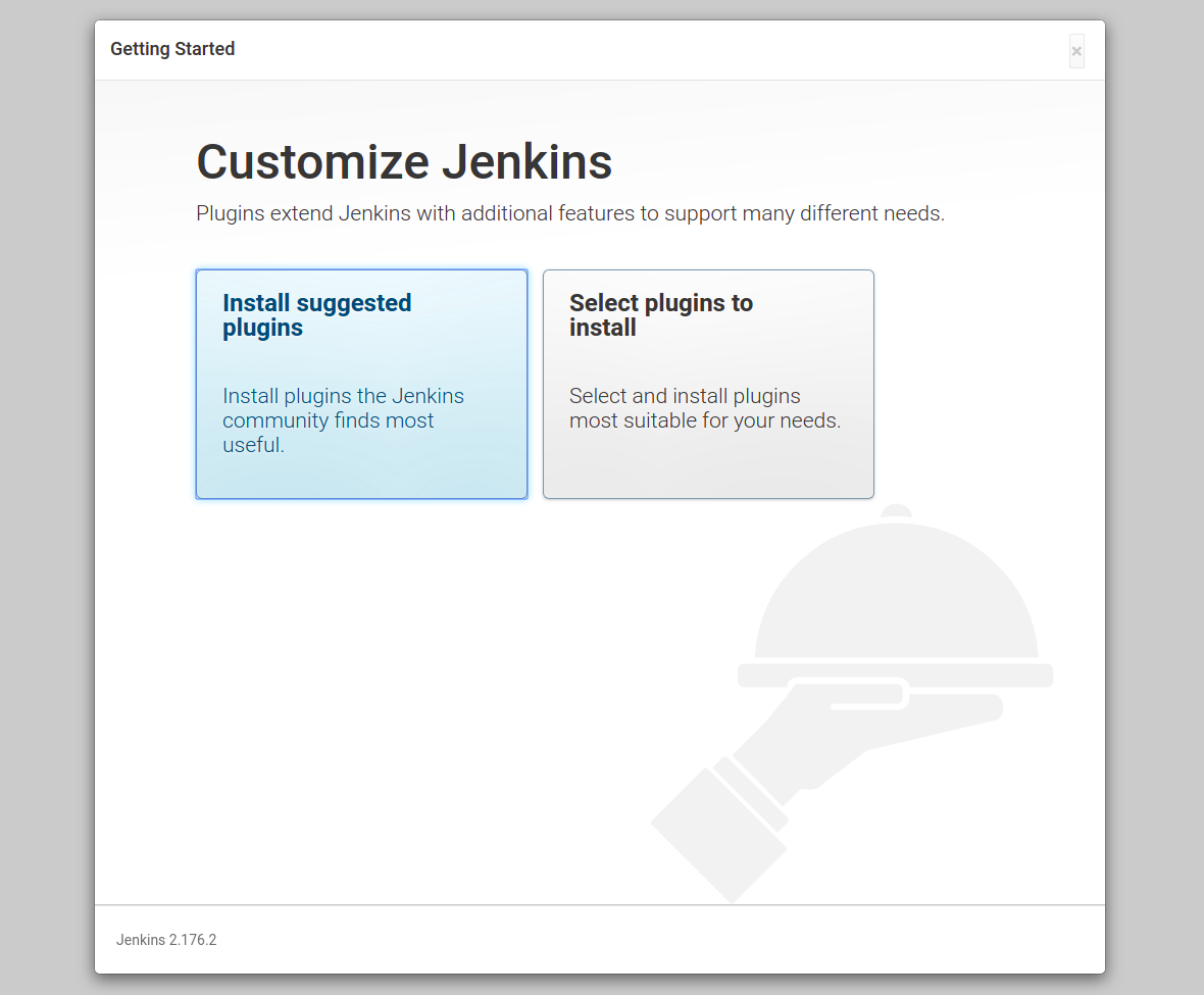
```
User=jenkins
WorkingDirectory=/home/jenkins
ExecStart=/usr/bin/java -jar /home/jenkins/jenkins.war

[Install]
WantedBy=multi-user.target
EOF
sudo systemctl daemon-reload
sudo systemctl enable jenkins
sudo systemctl restart jenkins
sudo su - jenkins << EOF
until [ -f .jenkins/secrets/initialAdminPassword ]; do
    sleep 1
    echo "waiting for initial admin password"
done
until [[ -n "\$(cat  .jenkins/secrets/initialAdminPassword)" ]]; do
    sleep 1
    echo "waiting for initial admin password"
done
echo "initial admin password: \$(cat
.jenkins/secrets/initialAdminPassword)"
EOF
```

Once you have ran the script, you should be able to go to port 8080 on your machine and the Jenkins app should appear.

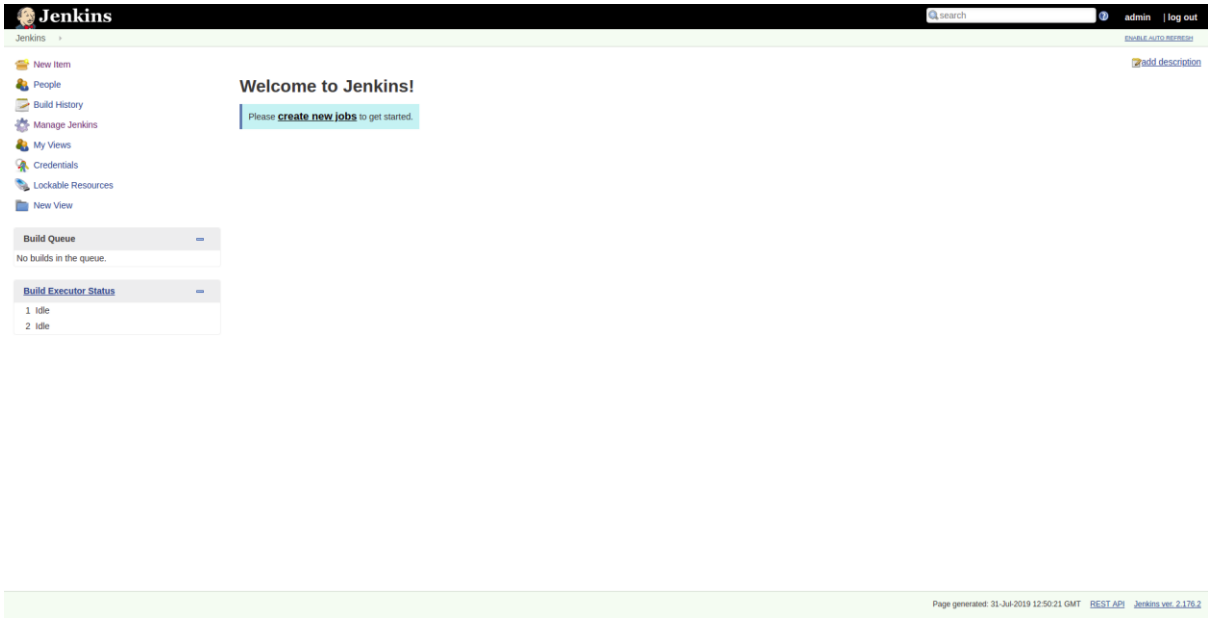


Enter the initial admin password and select Install suggested plugins



Follow the next steps, create a user and then select Save and Finish.

This should bring you to the Jenkins dashboard.



New Item	This is for creating new jobs in Jenkins. Jobs are essentially scripts that can be triggered.
People	The users that are registered to this instance of Jenkins.
Build History	A graph displaying the jobs that have been executed over time, through this instance of Jenkins.
Manage Jenkins	This is where to go for setting up plugins and other administrative settings for Jenkins.
My Views	You can customise how the jobs, and what jobs, are listed on the dashboard here.
Credentials	In the Jenkins jobs that you create, you may need to authenticate with external services, such as GitHub. Credentials for these external services can be stored securely here and accessed by jobs and plugins when they need them.

Lockable Resources	This plugin allows defining lockable resources (such as printers, phones, computers, etc.) that can be used by builds. If a build requires a resource that is already locked, it will wait for the resource to be free. You can define a lock-priority globally or on a per-job basis.

Jobs

A Jenkins project (job) is a repeatable build job, which contains steps and post-build actions.

A job can really do anything, it just depends what you configure it to do.

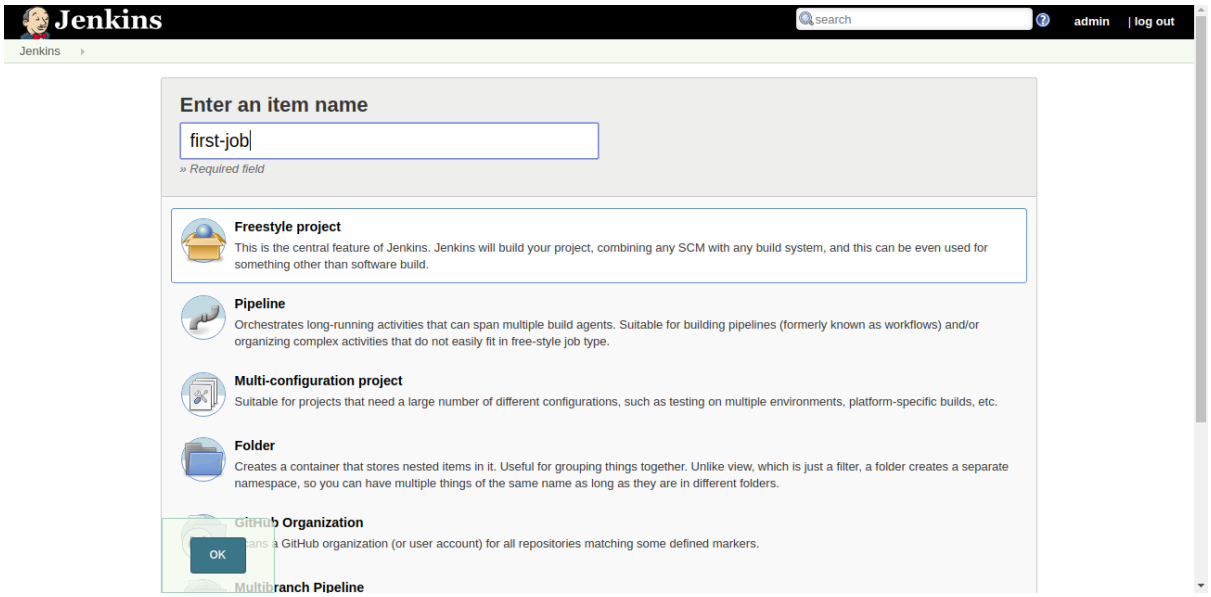
An example of what a job can be used to do is - automatically build a project and deploy it on a server, to be accessed over the internet.

Create a Job

To create a new job, you can navigate to the New Item link on the Jenkins dashboard.

This will then present you with some options for what type of job to create.

Go ahead and name your job first-job, select Freestyle Project and then select OK.



Workspaces

A workspace in Jenkins is basically just a folder that is on the host machine.

Jenkins will run every single job in its own workspace.

This is to keep jobs separate from each other, which avoids conflicts with files and configurations that the jobs may be using.

Help!

There are many configurations to choose from when setting up a Jenkins job; however, there is usually a "?" symbol on the right side of the page, which gives a comprehensive explanation of what the option does.

Job Configuration

This is where you can set up what your job is going to do.

There are many options, but this module will just focus on the General Settings.

General Settings

The settings and options throughout all job configurations will depend on what plugins are installed; this module aims to explain the ones that are suggested when you go through the Jenkins setup.

Setting Name	Description
Description	This is just an open text field to fill in information about the job. You'll likely want to put some instructions about your job here, or some information about its current state; for example, if it's a work in progress.
Discard Old Builds	Disk space doesn't grow on trees. If your job uses up quite a bit of space, then, after you have built it many times, considerable disk space might be used by old builds that no longer serve any purpose.
GitHub Project	Jenkins jobs are commonly associated with some kind of source code repository, like GitHub. You can add a link to that repository here, so it's easier to navigate to from the job's page.
Parameters	To make your job more generic and able to accept different configurations, you can pass it parameters.
Disable Project	Prevention is better than cure. If there are issues with the job's current configuration and you would like to make sure that it doesn't get executed, then you can check this option.

Concurrent Builds	Use this option carefully. There are many cases where your job will not be able to run at the same time as another instance of it, which is why this is disabled by default.

Build History & Console Output

The build history is simply a list of all the individual builds that have run.

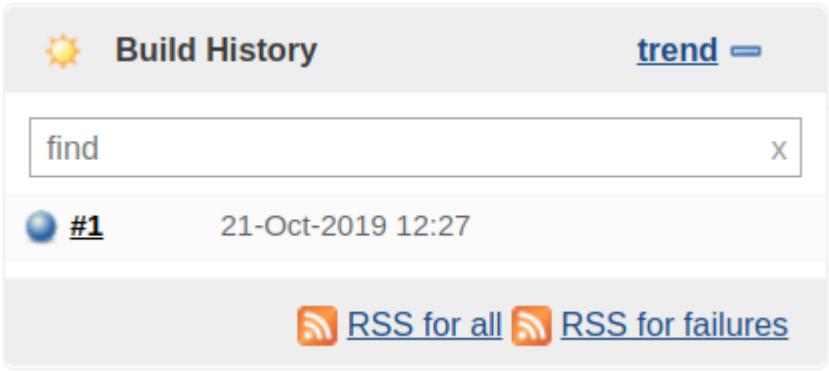
These are especially useful for checking the *Console Output*, to acces this you can either:

- Click the blue or red 'bulb' on the job to go straight to the Console Output
- Click the number and then once you have been sent to the build page, select the *Console Output* link

Builds

A build is a result of an execution of a project (job) in Jenkins.

Builds for projects can be seen on the Build History section of the project dashboard:



Builds are named by the order that they were executed, so the first build ever executed will be called #1 and the second #2, etc.

Build Status

The status of the build can have a few different values:

SUCCESS: The build succeeded.

If all the build steps complete successfully, this will be the build status.

FAILURE: The build failed.

If any of the steps exit with a non-zero status (if they throw an error), then the build status will go to failed.

ABORTED: The build aborted before it finished.

This exit status is more uncommon; it must be set either by yourself or plugins that are being used in the project.

Build Steps

Build steps are effectively where you configure what your job is going to do. Depending on your situation, this could accomplish many different tasks:

Java

If you have a Java project that you are wanting to automate builds for, you may be running commands or using plugins for running Maven, Gradle or Ant.

Docker

Docker images can be built for your project.

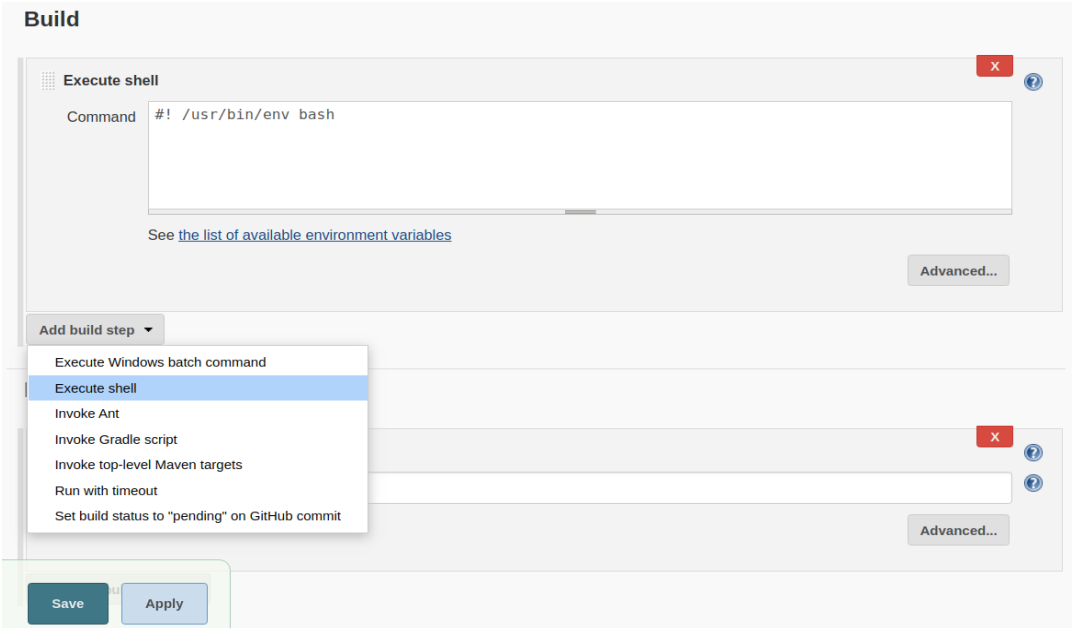
NodeJS

Installing dependencies for your node application before packaging it up for deployment at a later date.

Any Other Platforms

With the combination of shell scripting and plugins that are available for Jenkins, the sky truly is the limit with what you can automate in a Jenkins' build.

We can add build steps by selecting the Add build step drop-down button and then selecting the type of build step that we would like to use:



The type of build steps that are available will depend on the Jenkins' Plugins that you have installed.

For the example shown above, we can see that some plugins must be installed for Gradle, Maven and GitHub.

Most of the time you can get what you need out of a build step from shell scripting.

It's preferable to separate your shell scripts into different build steps, to keep you project more maintainable.

For example, you could have separate build steps for compiling, testing and deployment.

Console Output

The console output is likely one of the main parts of a build that you will be checking for information and debugging purposes.

This section includes the output for any shell scripts and plugins that have been executed in the build step for a project

The console output can be navigated to quickly by selecting the blue or red bulb () on the build history.

Here is a very simple console output example:


```
Started by user admin
Running as SYSTEM
Building in workspace /var/jenkins_home/workspace/test
Finished: SUCCESS
```

Post-build Actions

Post-build actions are plugins that can be run after you execute all the build steps.

You can add a post-build action by selecting the Add post-build action drop-down button and then selecting a post-build action from the list.

These actions can be set to run, or not run, depending on the success of the build steps.

Even if the build failed, you could still run post-build actions as an immediate response to the build failing.

For instance, you may want to send an email notification on a build failure.

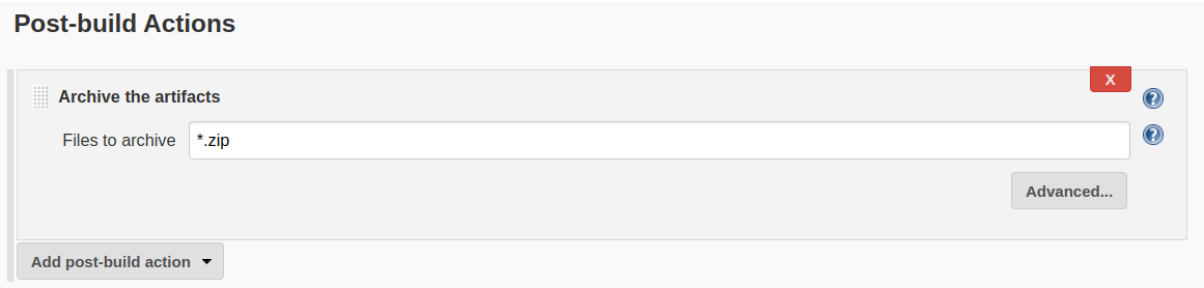
If the build succeeds, you might want to store the built application files as an artifact for access later on.

Artifacts

Build artifacts are immutable files that are created as result of the build.

You have complete control over which files you would like to store as an artifact, and this can be configured in the Post-build Actions section of the project configuration.

The example here will archive any files and store them as artifacts for the project:



Once the job has executed, the artifacts are then available on the job dashboard:

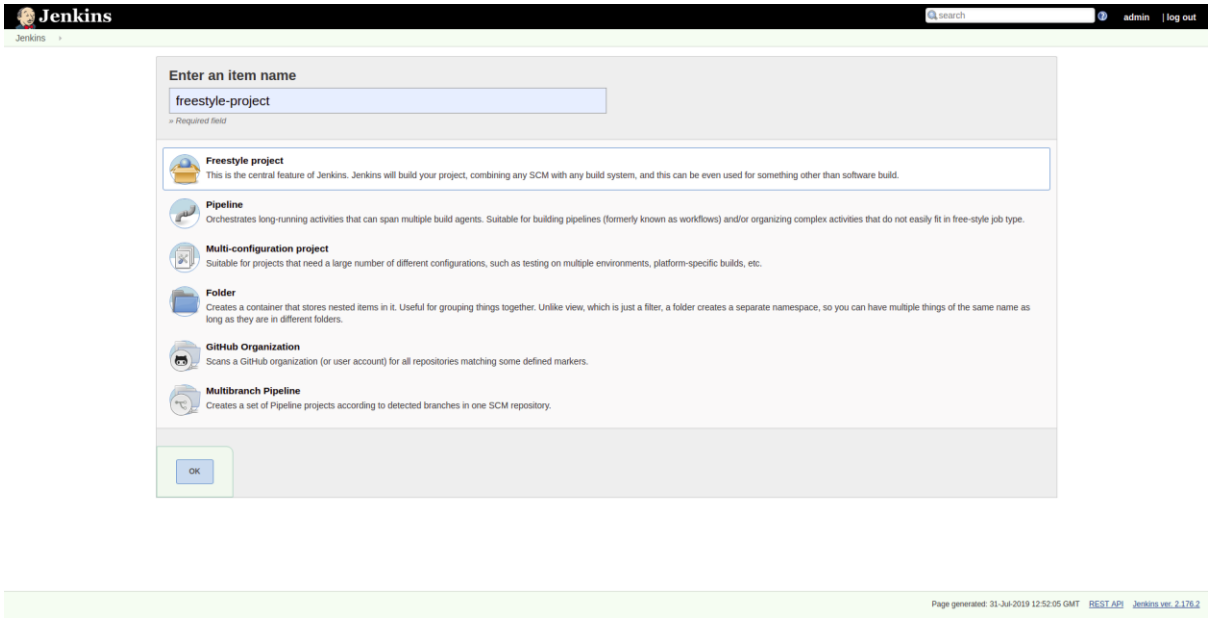
Freestyle Job

Freestyle projects in Jenkins are a type of job you can create for almost any automated task.

These are the best place to start for building any sort of general purpose automation in Jenkins.

Create a Freestyle Project

Go ahead and select New Item from the Jenkins dashboard, and then create a new Freestyle Project called freestyle-project:



Configuration

Once you have created a new job, you will then be redirected to a configuration page for that job.

Source Code Management

This section is used for configuring a source code repository to download.

The job will download the repository you provide into the jobs workspace.

Build Triggers

The most simple way to run a Jenkins job is by pressing the build button for the job.

However, jobs can be triggered in many ways - we will usually try to avoid doing this manually.

Option	Description
Build Periodically	You can create a schedule here for the job; for example, having it build every hour or at 6:15 PM every Thursday.
GitHub Hook	This is where GitHub can send a HTTP POST request; for example, a web hook to your Jenkins server to trigger a build of the job. This must be configured in GitHub and your Jenkins instance must be accessible from the internet for this to work.

Poll SCM	This feature can be used if your Jenkins instance is not accessible on the internet. Jenkins will check, using a schedule that you define, whether a change has been made on the configured SCM repository. As soon as a change has been detected, the job will be executed.

Build Environment

Various options that can be configured for when the job runs.

Option	Description
Delete Workspace Before Build Starts	You will likely want this option checked. The folder where the job runs on the host machine's file system will be deleted before building again.
Secret Texts & Files	You may securely use secret texts and files that you have configured in the Credentials section here in the job. These secrets will also be hidden in the Jenkins logs as well.
Build	This is likely where you will spend most of your time on a Jenkins job. The most common build step here is Execute shell; other options are available, depending on what plugins you have installed. Exactly what your job accomplishes is configured here.

Build

The part of the job that is 'executed'. Functional steps of the job are configured here.

Option	Description
Build	Select your build step. This is likely where you will spend most of your time on a Jenkins job. The most common build step here is Execute shell; other options are available, depending on what plugins you have installed. Exactly what your job accomplishes is configured here.

Post-build Actions

Events to trigger once a build has been completed.

Option	Description
Add Post-build Actions	Select your post-build action. You may want to configure your job to react depending on how the build went. For instance, you could be notified by email if the job failed. You could also publish a report if the job completed successfully. Like with most of the options, the sky is the limit; it all depends on what plugins are installed.