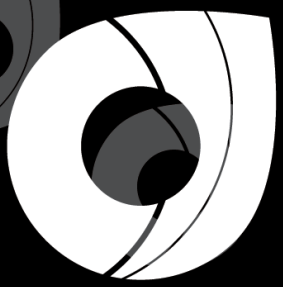
Corndel DevOps Engineering Programme

in association with Softwire

Module 6: DevOps Culture



Corndel Digital.

DB1S

Introduction

So far this course has focused on the tools and technologies that are central to the day-to-day work of a DevOps engineer. In this module we will take a step back, and look at the **workplace culture** and **business attitudes** that are needed to make the most of the DevOps skillset. DevOps is best practiced as part of a wider culture that values multi-disciplinary teams working quickly and efficiently to deliver user-focused software. Such teams must be able to deliver value quickly and efficiently, while also adapting to changes in business environment and priorities.

Achieving this is difficult, but is central to being a good DevOps engineer and spreading good DevOps practice across a company. In this chapter, we will discuss the approaches and attitudes that teams can take to make it easier to achieve these goals, as well as the broader business value delivered by DevOps practices and culture.

We'll look at how we can build **agile** teams that are able to effectively prioritise on the fly, keeping them focused on building and delivering the most important functionality in an environment of evolving requirements.

We'll look at how we can better understand our **users** and what they need from our software, and use this information to inform and drive the technical decisions we make when designing and implementing features and architecture. We'll discuss how this is an iterative process, and introduce the ways in which we can gather feedback on how our software is being used.

Finally, we'll discuss application and system **architecture**, looking at the ways we can deliver maximum functionality while keeping the system as simple and maintainable as possible by using common design patterns to solve industry-wide problems. We will review both technical and user-centric requirements, and see how both can deeply influence our decisions about how to structure our systems.

Table of Contents

Core Material

Introduction

Chapter 1:
DevOps Goals
- Development vs Operations
- Metrics
- Business Value Summary

Chapter 2:
Agile
- Waterfall Development
- The Agile Way
- Communication
- Agile, Culture & People Development
- Culture and Values

Chapter 3:
User Centric Design
- UI and UX
- Principles of User-Centric Design
- Learning about your users
- From User to Development

Chapter 4:
Architecture
- Building Clean Architecture
- User Requirements & Architecture

Chapter 5:
Problem Solving
- Affinity Mapping
- Impact Mapping
- Plan-Do-Check-Act

Additional Material

Agile Terminology

Remote Agile?

Agile at Scale

More on Accessibility

Architecture Patterns

Chapter 1

DevOps Goals

Throughout the course we have discussed, and will continue to discuss, the technologies and practices you'll use day-to-day as a DevOps engineer. We have discussed general purpose coding, automated testing, command line scripts and immutable infrastructure. We've seen how these technologies help with the stated aims of automating manual tasks and treating infrastructure as code.

Now, take a step back and consider not *how* DevOps engineers work, but *why* the role exists. What is the overall value of DevOps to a business, and how can it be measured? Traditional software engineering teams can report on user-facing feature delivery, but the impact of good DevOps can be harder to spot.

Development vs Operations

DevOps engineers straddle the roles of two traditionally separate teams, each with their own goals:

- **Software Development Teams** aim to deliver new features as quickly as possible, providing value to end users.
- **Operations Teams** aim to maintain service stability, which is best accomplished through limiting and carefully testing software deployments.

These goals are often in conflict. Development teams best demonstrate their value by pushing out feature after feature, with new deployments every day (or more frequently). That is often unacceptable to operations teams who may require a service outage to apply each update, and are often held responsible if a software deployment causes a user-facing issue.

By assuming both responsibilities, DevOps professionals look for sensible compromises that better deliver on overall business priorities, and use technology to eliminate or greatly reduce traditional trade-offs. This usually leads to an approach along these lines:

- Software deployments should be automated to the greatest extent possible.
- Software deployments should not routinely incur system downtime.
- Software deployments will sometimes break - it's unavoidable. Have processes in place to detect, fix or rollback broken deployments.
- Minimise total system downtime, rather than trying to avoid outages altogether.

Metrics

The goals defined above should be broadly agreeable, but how can we measure the impact of good DevOps practices? It's important to have good, reliable metrics to demonstrate value added.

There are a handful of common industry metrics widely used to report system performance and issues. These are not DevOps-specific, but are useful ways to track the impact of DevOps practices on a service or team. Let's run through some of the more popular metrics used.

Mean Time to Failure

Mean time to failure (**MTTF**) is the average time between system outages. A high MTTF implies a stable system that can relied upon by its users. This is obviously desirable, but it shouldn't be looked at in isolation.

Maximising MTTF at all costs is the traditional operations approach, and typically results in an extremely slow deployment pipeline with many manual approval and review steps. Feature delivery becomes slow, limiting the system's business value. Extremely rare failures, coupled with a slow, manual deployment process, risks a delayed response when a failure does occur.

Mean Time To Repair

Mean time to repair (**MTTR**) is the average time taken to repair a system after failure. It's usually considered alongside MTTF when evaluating overall system reliability, and is arguably the more important metric from a DevOps point of view.

You need good monitoring to detect failures and measure MTTR. An ideal monitoring setup also predicts when failure is likely, enabling preventative action. Failure prediction isn't always possible, but can be very useful in specific scenarios: for example, alerting you when a database server is running out of storage space.

Automated solutions minimise MTTR by removing slow human input and decision making. Software frameworks can help. For example, most web servers will automatically restart failing worker processes, and container orchestration tools use healthchecks to detect, kill and replace unresponsive containers. You can write your own automated repair systems to handle routine operations like server restarts or software rollbacks.

More involved repairs typically require human intervention. In this case, repairs are quickest in environments that leverage automated testing and automated deployments. Such systems allow engineers to develop, test and deploy fixes quickly and safely. Faced with similar circumstances, a traditional

operations environment with manual sign-off processes must either adhere to the slow process, or cut through "red tape" and deploy a fix without appropriate safety checks - an unhappy trade-off between speed and safety.

MTTR is a powerful metric of success in DevOps environments. A low MTTR shows that failures are detected and addressed quickly, and implies that the underlying development process is sufficiently flexible and well-resourced to develop and deploy fixes without delay.

Service Level Objectives

MTTF and MTTR are internal metrics used together to paint a meaningful picture of service reliability. A service should not fail too frequently, and it must recover quickly when it does. DevOps engineers help build the monitoring systems that report these metrics, and are also responsible for the processes that improve them.

A service level objective (**SLO**) is a user-facing target for service performance or reliability. For example, you might guarantee users of your web application that the app will be available 99% of the time. 99% uptime is a SLO. A SLO is a metric that feeds into a Service Level Agreement (**SLA**) - the SLA is a broader document that specifies what the consequences are for missing the SLO, or any other relevant business agreements.

SLO selection is a tricky process. A weak commitment will drive away users fearing unreliable service. An overly strict commitment will hamstring the development process, and may provide no user benefit over a slightly weaker SLO. DevOps engineers help define SLOs that meet user requirements, while also allowing development teams freedom to deploy new code without undue limitation.

DevOps engineers help agile teams meet SLOs by building robust infrastructure and tooling.

More Metrics

MTTF and MTTR, discussed in the main text, are just two metrics widely-used in the software industry. Here are a few others you might see:

- **MTBF** - mean time between failures
- **MTTR** - mean time to recover
- **MTTR** - mean time to resolve
- **MTTR** - mean time to respond
- **MTTA** - mean time to acknowledge

As always, these are broad industry-wide terms and specific businesses may have their own domain-specific interpretations. Always check with your team! In particular, be wary of MTTR. In this course we discuss *mean time to repair*, but there are several other definitions of the acronym listed above.

Business Value Summary

DevOps engineers help agile software teams optimise their tooling and workflows. They bring operational knowledge into the team, helping integrate historically conflicting development and operations priorities to improve efficiency.

By agreeing and implementing robust service metrics (e.g. MTTF, MTTR) DevOps engineers can monitor team efficiency and service performance, and use these metrics to inform future improvements. DevOps engineers help define and meet user-facing SLOs that fulfil user requirements while also enabling, rather than restricting, feature delivery.

Chapter 2

Agile

In this section we'll review how DevOps engineering has evolved as part of a wider move towards more flexible, iterative and multi-disciplinary approaches to software projects. These developments are broadly known as **Agile** and aim to address some of the key challenges faced by projects delivered along more traditional, linear methodologies such as **Waterfall**.

We will look into the traditional Waterfall delivery model, and contrast it with an Agile approach. We'll review some of the core ideas behind Agile, and see how they address some common challenges to project delivery. This discussion centres on the software industry, but many of the concepts are broadly relevant to all managed activities.

Agile teams aim to deliver quickly and frequently, prioritise effectively and adjust to changing requirements or new data. The best software teams are always looking out for ways to improve both their knowledge and their workflows, and encourage strong communication skills to help eliminate knowledge silo and improve requirements gathering. We'll discuss how DevOps engineering is central to delivering on these targets.

Waterfall Development

The **Waterfall methodology** is a way of planning and delivering a software engineering project. The approach dates back to the earliest large-scale software projects, which drew inspirations from similar physical engineering projects. A Waterfall project involves a series of well-planned and estimated phases of work, performed in sequence. A typical (simplified) set of steps is as follows:

1. Requirements Analysis
2. System Design
3. Implementation
4. Testing
5. Maintenance

Waterfall software development is fixed, rigid and predictable. The early phases invest a great deal of effort in making all design and functionality decisions. There is clear separation between the subsequent implementation, testing and deployment/ maintenance phases, which leads to a natural separation of development, testing and operations teams. Sharing of knowledge and expertise between the teams is often minimal.

The waterfall development workflow is frequently criticised in modern *agile* circles. We will review the common pitfalls of waterfall in a moment, but for now note that waterfall development provides a clear

specification and delivery estimate. These factors are of great value to business stakeholders. In addition, rigid adherence to a careful design is essential in safety-critical systems and highly-regulated environments, where the uncertainties of agile delivery (discussed next) are unacceptable.

Nevertheless, waterfall development suffers from some drawbacks that make it unpopular for many modern software projects.

1. **Slow to add value** - No software is delivered until the whole project is complete, which may take years. This means that user feedback can only be incorporated in the maintenance phase.
2. **Design inaccuracies** - It is extremely difficult to accurately design a software project from the outset, without the benefit of iterative feedback. Even an originally-accurate design may no longer meet user requirements when the project is finally delivered.
3. **Rigid** - Waterfall has little capacity for incorporating design or requirement changes, as designs are fixed early in the process.
4. **Delays propagate** - Key waterfall milestones are often fixed dates and can involve handover to other teams, leaving little capacity to absorb delays. In practice, implementation delays often shrink the testing phase to keep to overall delivery targets. This is far from ideal.
5. **Slower bug feedback** - Bugs detected during testing or maintenance can be expensive to fix as the original development team are no longer available.
6. **Poor knowledge sharing** - Development, testing and operations team interaction is often limited to formal handovers and bug reports. This restricts the sharing of project-specific expertise, and introduces overheads as teams fail to adopt or communicate processes that could improve workflows across team boundaries.

The DevOps role does not exist within a Waterfall project. DevOps is a contraction of *developer operations* - two separate project phases, owned by separate teams.

The Agile Way

Agile software development is a broad family of software development methodologies that share some core principles relating to flexibility, rapid delivery and shared ownership. Agile workflows have evolved to address some of the pain-points in the more traditional waterfall approach. Most importantly, Agile combines the traditionally separate development, testing and operations phases into one.

Agile development relies on the ability to release frequently, gather feedback, and re-prioritise based on that feedback. DevOps is central to this effort. DevOps engineers facilitate rapid, safe deployment through *automated testing*, *continuous integration* and *continuous delivery*. DevOps engineers support gathering feedback through *logging and monitoring* systems, and use *immutable infrastructure* tools (e.g. docker) to safely update single components of a system. DevOps engineers ensure the system remains automated, flexible and up-to-date; they deploy tooling and infrastructure that enables teams, rather than limits them.

The core principles of agile are summarised in the Agile Manifesto, repeated below:

- **Individuals and interactions** over processes and tools
    - Have a informal chat with a team member about some odd code behaviour, rather than immediately raising a bug report.
    - Discuss requirements with the customer, rather than asking for a functional specification.
- **Working software** over comprehensive documentation
    - While documentation is important, it should never be to the detriment of software quality.

Agile or not?

The word 'agile' can mean different things to different people. That includes within the software industry! It is perhaps unfortunate that software professionals have a habit of re-purposing existing everyday words. Nevertheless, you'll come across "agile" in various in the software industry:

1. The adjective 'agile'.
2. The Agile Manifesto and its core principles.
3. A rigidly-enforced workflow (e.g. scrum or kanban).
4. A marketing keyword used to sell a variety of courses, qualifications and tools.

We focus on the underlying ideas and motivations behind agile software development, rather than fixing to one of many possible specific definitions. Each software project should adopt the principles and workflows that work best for them, modify and iterating as necessary.

    - Good software is largely self-documenting, with intuitive design and built-in help tooling.
- **Customer collaboration** over contract negotiation
    - Collaboration gets the customer involved in the delivery process.
    - You gain valuable user insight, and the customer gains visibility of, and trust in, the development process.
- **Responding to change** over following a plan
    - Requirements change; inflexible projects risk releasing software that is not fit-for-purpose and can prevent you from quickly taking advantage of new opportunities or shifting priorities.

These key principles are terse, but important! Take a moment to think about their relevance to your own professional experience.

To apply these principles, some agile teams split a large software project into smaller pieces, often called tasks. Instead of planning everything up-front, the team repeatedly plan and re-plan their next set of deliverables, picking up tasks based on short term priorities and the available resources. These planning & delivery cycles are usually a couple of weeks in length, and are usually referred to as **sprints**.

An agile team typically owns all phases of software delivery, testing and maintenance. To enable this, teams are often multidisciplinary, combining technical roles (e.g. software engineers, DevOps engineers) with more customer and user-facing skills (e.g. product owner, scrum-master, user researcher).

Agile focuses on delivering a **minimal viable product (MVP)** as early as possible, and then iterating on it based on detailed monitoring and user feedback. Teams release software far earlier than is possible in a waterfall methodology. This has two main advantages. Firstly, releasing software demonstrates progress to business stakeholders and builds customer trust early in the delivery process. Secondly, incremental releases let developers gather critical user feedback and fine-tune the project plan as they go; this goes a long way to ensuring that the finished product truly meets user requirements.

Agile projects are not all perfect, however. There are some common difficulties:

- **No fixed scope** - Agile projects are usually described as *fixed time/cost*, rather than the *fixed scope* of waterfall. This can make it more difficult to 'sell' agile projects to budget holders at outset, and requires greater trust in the development team to deliver.
- **Risk of repeated work** - Changing requirements can sometimes require the team to re-work, or entirely scrap, existing code.

For many projects agile is a great fit but it might not always work for you. It is best to understand the concepts and cherry pick the parts that work well for your project.

Agile Ceremonies

Agile workflows typically involve several **ceremonies** (meetings) to facilitate and encourage knowledge sharing and communication. These are equally important for both technical and non-technical persons involved in delivery. Some key ceremonies are described below, and more details can be found in the additional material section (including practical tips on how to get the most out of them).

Standups

A **standup** is a very brief meeting, usually held daily, where each member of a software team summarises their recent work, shares their plans for the day and raises any blockers, bugs or dependencies they've discovered with the rest of the team. Standups are a central mechanism for knowledge sharing within an agile team. They are informal and usually internal.

In addition to routine standups, many teams will run more dedicated knowledge sharing sessions. Pair programming, mob programming and lunchtime presentations are common ways to share knowledge through a team.

Demonstrations (or Show and Tell)

A **demo**, also known as a **show and tell**, is a customer-facing meeting where a software team present their progress to project stakeholders. For an agile team, this will often be a guided tour through recent features deployed to live software. Demos are usually non-technical in their focus, although this can vary depending on the content of the sprint.

Sprint Planning

Sprint planning meetings define the priorities and goals for an upcoming sprint. They typically involve a product owner and scrum-master, but may include the whole Agile team or key external stakeholders. Sprint planning

will set goals, and review any upcoming milestones. Selected tasks are brought from the *backlog* into scope for the sprint, taking into account available development resources and current development priorities.

Retrospective ('Retro')

A **retro** is an internal meeting where the team reflects honestly about what went well, and what could be improved about the project and workflow. Most Agile teams hold a retro at the end of each sprint, and will focus on that sprint, although they may be less frequent. A retro should review positive achievements, and also areas where the project did not progress as smoothly as had hoped. Discussion should be constructive and blameless.

A retro should end with a clear set of actions, defined based on the discussion and ideas raised in the meeting. A retro often starts with a review of the previous retro's actions.

Communication

Communication, both within your team and externally, is a key skill for a DevOps engineer and anyone else involved in agile software delivery. You should be comfortable discussing your work with both technical and non-technical people at all levels within a business and through various media (face-to-face, phone call, email, message platform, video chat).

Internal team communication helps build strong professional relationships, and facilitates knowledge sharing. You'll gain expertise in new areas of the project (particularly if working in a multi-disciplinary team), and pick up valuable career-long skills. These communications may be informal, but that does not make them unimportant.

Agile and Software Quality

Some readers may wonder whether the flexibility and freedom of the agile process will reduce quality. In fact, the opposite is generally true. Agile ensures the product aligns closely with user needs, and develops an architecture that serves those needs. DevOps approaches let teams iterate rapidly. Features are never "set in stone" and strong tooling leverage automated testing and deployments to reduce the number of errors that slip through.

Agile workflows are often challenged in highly regulated industries (e.g. finance), where compliance is traditionally assessed with Waterfall development methodologies in mind. Specific DevOps processes such as continual deployment are a poor fit in such environments, but the broader DevOps and Agile movements can work well. DevOps practices enhance quality, improve traceability and minimise the opportunity for human error.

We'll come back to this topic in Module 8 when we discuss continuous delivery and continuous deployment pipelines.

Clear, professional communication with external project stakeholders helps build trust and rapport between the two parties. Strong relationships with project stakeholders will help you when you need them to take action (e.g. provide clarification on requirements, sign-off a project budget), and will make it easier to communicate progress updates.

Choosing a Communication Medium

Choosing the right communication medium can be as important as choosing what to say! There are many ways to communicate in the modern workplace:

1. Face-to-face
2. Email
3. Phone call
4. Instant message (e.g. Slack)
5. Video call

Each method has its own pros and cons. Here are some things to consider:

- **How urgent is it?** Emails are asynchronous and low-bandwidth - don't use them if you need a response immediately. On the other hand, don't interrupt someone for a phone call, video call or face-to-face meeting if you really don't need an answer to your question soon.
- **How easily explained are the issues?** Some matters can be put across much more clearly in carefully structured text. But if the recipient might not understand what you're saying - or if making sure they understand means writing an excessively long and time-consuming message - then a phone call, video call or face-to-face meeting is preferable.
- **Pictures or words?** - Some topics are better shared as pictures. This might include a team organisation chart, or an architecture diagram. Take some time to create clear illustrations, and you'll have more success sharing ideas. You can follow up with a meeting or video call to discuss feedback.
- **Does the information need to be recorded for posterity?** It almost always does! This doesn't mean you need to write it down in the first instance though - consider discussing in person and then writing it up afterwards.
- **How much back-and-forth will there be?** Long email trails debating an issue are rarely worthwhile. It's much better to speak directly to someone and come to an agreement than it is to let your discussion drag on in text form.
- **To what extent is having to think on the spot likely to be effective?** This is a counter-balance to the previous bullet - if you need to put some good hard thought into something, it might be better to do that offline than in a meeting. But don't forget that once all parties have done this, a face-to-face

meeting might be the best way to move to a conclusion.

- **Does this need to be phrased carefully?** Nuances of body-language and tone are important in all human communication - it's surprisingly easy to mis-read the tone of an email or instant message. Even phone calls can be tricky without visual cues. Sensitive topics are best discussed in-person or over a video call.
- **Do you need to see their reaction?** How someone reacts to a communication is important. Discussing in person or over a video call means you can see how they are reacting and tailor your approach. If you're negotiating it's almost always preferable to do so face-to-face.
- **How much effort is imparting and receiving the information?** It might be quicker all round to chat than to think about how to write something down, and it might be easier to listen to a brief explanation than read an email. On the other hand, don't dive into a conversation without thinking about what you're going to say in advance!
- **What do the individuals involved prefer?** Everyone has their own personal communication styles. This becomes most relevant when working with people regularly - some prefer a chat, others like not to be interrupted unless it's urgent. Find out how the people you communicate with most often prefer to interact, and fit that in with your own communication style.

Agile, Culture & People Development

Agile practices require development teams that are able and willing to take ownership of the whole product lifecycle. This is a marked change from waterfall methodologies, where individual teams are involved in only a single part of the process with well-defined boundaries.

Individuals working on agile projects should feel empowered and supported to take ownership of software features and have the initiative to drive them forwards. This supports the individual's technical development, as well as having a wider positive impact on their team.

Junior team members in particular can learn a great deal from the rapid agile delivery cycle. A developer can create a feature, test it and deploy it to production within a few days. They can see a feature move through the full delivery cycle, support it once deployed, and all the while applying any lessons learned to the next task. This kind of holistic view is very hard to acquire quickly on waterfall projects.

Agile team-members should keep up to date with the latest technologies and further their career development through training courses, certifications and attending conferences. They should be motivated to adapt ongoing work to take advantage of recent developments, rather than sticking to long-established processes. That is only possible where teams have a positive attitude towards ongoing learning, and actively seek out opportunities to develop new skills.

DevOps engineers can look to develop their skills by joining professional organisations, attending informal meetups and conferences. Major cloud infrastructure providers offer certification programmes for various cloud technologies; these can help develop and demonstrate technical ability, and many employers sponsor enrolment.

Agile puts a lot of stock in people (*people over process*), and businesses need to develop a culture that trusts their technical teams to work effectively. Individual motivation, training and adaptability are essential. Agile works best when practiced in a blameless culture of continual improvement, where the response to a bug is not "who caused this?" but instead "how can we learn from this, and improve our workflow in future?". The rapid delivery iteration provides ample opportunity to learn from occasional mistakes and oversights.

Agile processes require great communication and inter-personal skills, even in roles that are primarily technical. An open and positive culture, coupled with a healthy lifestyle and work-life balance can help all team members develop skills.

Zero-Blame Cultures

Sometimes, things go wrong in the DevOps workplace. A bug may get through testing, the test suite may be brittle and weak, or perhaps whoever created the VM didn't assign enough RAM and the process won't stop crashing. We're all human and make mistakes.

When something goes wrong it's very easy to blame the person that made the mistake and continue on, perhaps assigning the bug-creator to fix "their" issue. This is very common, and it's a deeply unhelpful way to address a mistake. This attitude does not promote learning from the experience, and the person responsible likely did not intend to make the mistake. Without learning from issues, there's every likelihood that someone else will make a similar mistake in future.

A better approach is to establish a zero-blame culture. At its core, a zero-blame culture assumes that everyone did their best with the situation, task and information they were presented with. Viewed in that light, we can ask: What went wrong? Was there a misunderstanding? Can we improve

communication channels in future? Can we improve our review process to catch that bug?

In short, this culture requires us to drop the idea of "people are to blame", instead seeing that "our system has flaws", and work constructively to address those flaws. This approach can be a great benefit to the team as a whole. People feel more able to talk about mistakes that might have slipped through. The team can identify and address flaws sooner, learn from them and improve the system as a result. Teams shift from finding problems and assigning blame to finding solutions and improving.

Zero-blame cultures are hugely important in DevOps. Mistakes will always happen; any system that assumes otherwise *will* fail sooner or later, and be less prepared to recover. That's poor design. We want to build systems where it's easier to not make mistakes, and where the system can tolerate mistakes when they do slip through.

DevOps engineers build systems that foster zero-blame cultures through increased sharing, constructive feedback and experimentation. DevOps tools empower teams, and help them feel that their environment is working with them, rather than against. This empowerment is essential to developing a zero-blame culture. When processes feels fixed and immovable, mistakes are bound to repeat and team members can feel powerless and apathetic. Below are a few specific ways good DevOps practices can help avoid this:

- **Write clear, easy to understand, maintainable code**. This facilitates future mistake-free changes.
- **Write unit tests**. Unit tests check and document code behaviour. They catch the small mistakes every programmer makes daily.
- **Pair programming and code review**. Code review catches mistakes quickly, and provides a great forum for constructive discussion and improvement.
- **Automation**. Automating manual process reduces the opportunity for human error in repetitive tasks.
- **User testing and feedback**. Catches mistakes arising from incomplete information or miscommunication.
- **Immutable infrastructure**. Immutable infrastructure protects against configuration mistakes by removing the need to make such changes in the first place.

These layers of defence guard against human mistakes. They reduce the chance of mistakes being made in the first place, while increasing the chance we discover mistakes before they reach our users. The result is a

great reduction in the number of issues seen by our users, and a learning process through which we continually improve the system.

Despite all these steps, mistakes will still happen! We need to continue to learn from them, work out how they slipped through the net and make sure the next one can't.

Culture and Values

Throughout this apprenticeship we highlight certain values that are central to a good DevOps workplace: positivity, proactivity and continual learning. We also emphasise some related values of wider societal and democratic significance. You can find these in the 'British Values' sidebar, and are relevant both in and out of the workplace.

British Values

The values below are the norms that shape our society and which are enshrined in law, through legislation such as the Equality Act 2010.

Democracy - Apprentices are encouraged to take part in democratic processes and understand how democracy influences all our lives, including through laws. In the workplace, Agile teams should give everyone an opportunity to have their voice heard in regular retrospectives, helping shape the team's environment and processes through democratic consensus.

The rule of law - Apprentices are encouraged to research health and safety laws which regulate industry or review the health and safety processes relevant to their work. Agile teams should adhere to all legislation relevant to their field of work, which may include topics of privacy, accountability and discriminatory bias in automated processes.

Individual liberty - Apprentices are encouraged to discuss the extent that this exists or is limited by regulation, as well as their liberty in a professional capacity. Agile teams should give everyone the liberty to pursue their work as they see fit, as far as is reasonable. For example, managers should be open to adjusting work schedules and trust individual approaches over rigid standard workflows.

Mutual respect and tolerance - Apprentices are encouraged to respect other people with particular regard to the protected characteristics of the Equality Act 2010. Agile teams are flexible and multi-disciplinary; they thrive on a breadth of experiences, backgrounds and opinions. Team diversity is a great strength, and is best shared in a culture of mutual respect through all Agile ceremonies.

Chapter 3

User Centric Design

In the previous section we introduced the agile workflow, and how it uses a rapid feedback cycle to incorporate feature requirements and user feedback into ongoing development.

Here we will dive deeper into **user requirements** and **user feedback**, how you can gather them, and how you can turn them into workable technical tasks that drive all feature development. We'll start by introducing **user interface (UI)** and **user experience (UX)** to lay the groundwork, then discuss how we use these concepts to drive development.

UI and UX

User interface (UI) is the graphical components that users see - the buttons, labels, shapes and colours on a web app, for example. UI is entirely concerned with *visual* appearance, rather than functionality.

User experience (UX) is a related, but very different, design concept. UX describes how users *interact* with your application. UX is about functionality - good UX provides the services that users want, with the minimum possible baggage.

Make sure you understand the difference between the two. UX describes how easily users can get what they want from your service. UI describes how good the interface looks. Good UX is essential. Good UI is important, but should be done to support great UX, not for its own sake.

Good UX, supported by UI, is critical for modern business success. Increasingly, customers interact with businesses solely via websites and apps; UX becomes a measure of a company's quality, and a unique UI design language can help develop and maintain a brand.

As well as making commercial sense, investment in great UX can be important for legal reasons. Many nations and business sectors require digital companies to provide adjustments for users with particular accessibility requirements.

Some simple UI errors, such as a font size that is too small, or a poorly-selected colour palette, may exclude large numbers of potential users.

Accessibility

Accessibility is about making digital experiences great for everyone, and is a cornerstone of good UX.

Many users have particular access needs that must be considered when designing services. Here are some common considerations, although this list is by no means exhaustive:

- Is my service navigable using a screen reader?
- Are my UI font sizes responsive to global scaling adjustments?
- Is my UI still clear to users with visual impairments, including colour-weakness and colour-blindness?

Addressing accessibility concerns need not detract from good UI and UX for the typical user. In reality, the opposite is true. For example, simple mobile-first designs are best practice in general, and facilitate easy navigation with screen readers. Proper use of semantic HTML tags not only helps from an accessibility point of view, but also benefits regular users using particular browser features or extensions (e.g. read-it-later services). Every platform will have specific frameworks to help with accessibility - make sure to use them.

Accessibility is broader than catering to specific disabilities. Here are a few more concerns that fall under the accessibility umbrella:

1. Does it work on a wide range of web browsers?

UI vs UX

Think about UX and UI when using the apps and websites you visit every day. This can help you identify common ways that designers get it wrong, but also highlight what works well.

Does it feel 'clunky' and hard to use? Is it difficult to find what you want? That's poor UX.

Does it simply look bad? That's bad UI.

2. Does it work on a web browser with JS disabled?
3. Does the UX remain strong on an intermittent internet connection?
4. Does my app work efficiently on low-end devices with limited CPU and memory?
5. Does the UI respond quickly, smoothly and as expected to all possible inputs?

Making sure your application works well under all circumstances, both personal and technical, will ensure your service is available to as many users as possible.

For further information on accessibility see Making Your Service Accessible - An Introduction on GOV.UK.

Principles of User-Centric Design

User-centric design is about building services based on user needs to provide the best possible UX. This might sound obvious, but is it really? Here are a few possible reasons for adding a new feature to a service:

1. "*A new API version exposed a neat feature, which we can probably make use of*" - This may be fun for developers, but does it address any user requirement? New technologies should be brought in to address requirements. User requirements should determine the technologies used, not the other way around.

2. "*Removing this user journey will reduce the back-office administrative load*" - In this case there are two user groups (your customers and office staff) with conflicting requirements. What can you do to align their interests? Is there some automation that would alleviate your back-office load, while keeping this user journey in place?

3. "*Switching libraries to avoid a security vulnerability*" - Security considerations are invisible to most users, and NFRs of this kind are rarely considered as part of user journeys. They are nevertheless important! If you need to make this kind of change, be sure to review any possible impacts on user experience.

4. "*The project lead thinks this feature is '*cool*'*" - Those involved in project delivery often have ideas on how to enhance a service, but whether these features are useful to end users is another matter. Such ideas should be fed into user research discussions, not added immediately.

5. "*This will increase revenue*" - Aligning business goals with great UX is a common challenge for modern tech companies, particularly those that must draw revenue from their digital offerings. Damaging UX to improve revenue is commonplace, particularly in modern public-facing digital services (e.g. intrusive advertising, pay-walling features) and can have a lasting impact on an organisation's reputation. Yet most private businesses must find a way to monetise digital services to continue providing user experiences at all!

These are just some suggestions of alternative drivers for feature delivery. They are not necessarily bad ideas, but UX changes based on anything other than sound knowledge of user needs are unlikely to make your service more appealing or intuitive to users. Other factors (in particular commercial decisions) may take priority, but you should always be aware of potential UX impacts.

To pursue user-centric design, you must first take the time to understand your users. There are many ways to approach this. For example, you might choose to run a discovery project, interview individuals or distribute surveys. We'll discuss some common **user research** strategies shortly.

User-centric design works best when coupled with agile delivery. The effect of each deployment can be monitored via analytics and advanced live testing approaches such as A/B testing. Does your feature improve UX? Could it work better? Is it being used in the way you expected? The answers to these questions should feed back into development, informing future sprint direction.

At every stage, user-centric design should be based on evidence about what works and what doesn't for users. This is an ideal, and it can be very difficult to achieve in practice.

Sometimes user-centric design is not possible or desirable. Users may be unavailable for testing, or perhaps the user needs are extremely simple and well understood. Sometimes a business will instinctively trust in-house judgement calls over an evidence-based approach. In extreme cases, it may be decided that UX is simply not a priority, although this is ultimately damaging to any digital service that operates in a marketplace.

Learning about your users

User research monitors user behaviour and gathers user feedback to define *user requirements*. User requirements direct future development efforts and inform architecture decisions. User research is central to a *data-driven approach* to delivering user-centric design.

DevOps engineers are unlikely to involve themselves directly in user research, but may work alongside user researchers and are well served by a basic understanding of the topic. In this section we'll discuss a few common techniques user to gather user requirements.

Interviews

User interviews are great for getting context and opinions, but are easily swayed by a small and/or biased sample. Careful planning is also needed to avoid introducing bias via leading questions.

Perhaps more worryingly, it's commonly understood that people aren't always good judges of their own actions! Many people will honestly say one thing in conversation, then act differently when the discussed situation arises. Here's an example of the US retailer JC Penney, whose user-research-based such as fair pricing strategy proved something of a failure.

Lab Test

A lab test involves recording users interact with your product, or a prototype of it. Lab tests let you see how a user will intuitively interact with a service, and highlight aspects of your UX that are intuitive and those that are not.

Similar to interviews, lab tests can be costly and logistically challenging to perform in large numbers.

Site Analytics

It's uncommon to rely solely on individual-based research (interviews and lab tests) to know how your digital service is performing, simply because those methods do not scale well. Almost every website and app will include an analytics package, such as Google Analytics or ChartBeat that can automatically log huge quantities of data about user behaviour.

Site analytics services can be a treasure-trove of information about how users interact with your site. Analytics data can identify popular pages, common paths through the site (**user journeys**), error scenarios, time-per-page and many other metrics. Some services can offer broad user geographic and demographic tracking, enabling you to build a clearer picture of your user base.

A/B Tests

An A/B test is a digital user-research tool used on live environments to test the impact of a new feature. It works by serving a subset of users an updated version of your software (typically a web app) and tracking a conversion rate, or other useful metric. A conversion rate is the proportion of users who "successfully" interact with your site (e.g. join a mailing list or make a purchase).

The rest of the user population are served the existing software version and act as a control group. A comparison of metrics between the groups can identify any UX impacts, positive or negative, associated with the new feature.

A/B testing is extremely powerful, as it lets developers test features "in the wild", and accurately measure their impact on user behaviour. A/B tests can generate huge quantities of data, enabling user researchers and data scientists to draw robust conclusions.

A/B testing should be used with care. Without additional controls, it is easy for A/B testing results to re-enforce existing biases in user populations, prioritising features that benefit the 'typical' user while potentially damaging the service for minority users. Data-driven decision making of this kind is best handled by professional data scientists with appropriate statistical backgrounds, and a good understanding of the diversity of the user population. You can read this article for more detail on some of the potential pitfalls.

Site Analytics

Site analytics is a popular topic, but not without controversy. Many tech-savvy users run tracking blocker browser extensions, or use DNS-level tools such as PiHole to block user tracking and analytics across a network. Apple's Safari browser limits the amount of fingerprinting data available to websites, as well as offering users the option to block analytics tracking entirely.

Such analytics tracking prevention may only be used by a minority of users, but it will bias analytics datasets and is becoming increasingly popular and topical.

Legal frameworks such as the EU GDPR place strict opt-in requirements on the storage and use of personally-identifiable information, and should be considered when implementing an analytics strategy.

From User to Development

We've discussed the goals of user-driven design and reviewed some of the ways user researchers collect the data needed to power this approach. But how do we turn user research and user needs into something a software developer can implement?

This is typically done through **user stories** - detailed descriptions of expected user interactions. User stories are built on two related ideas: **user journeys** and **personas**.

We've already mentioned user journeys. A user journey is a walk through your application - a series of related pages, buttons and inputs that a user will go through to perform a particular action. You can think of an individual user journey as a single slice through your user experience.

Personas

You can't write features for specific users. Your application may have thousands or millions of users. Instead, developers find it helpful to create **personas**. A personal is a fictional character that represents a certain type of user. A collection of personas, crafted with carefully-selected characteristics, can characterise a diverse user-base and ensure their interests and considered as part of the development cycle.

A typical project would create multiple personas, and use them to exemplify application requirements. Personas help you understand what your app does and how people expect to use it. They can be used to confirm that specific needs are accounted for and also track areas for improvement.

The value of using a persona is best demonstrated with an example. Consider the following hypothetical situation:

A web application you're working on has a well-understood user registration user journey. It involves creating an account online, clicking a link in a verification email then logging in.

That all sounds reasonable, but how would this particular journey play out when embarked upon by the following persona?

Gareth is a privacy-concerned tech-savvy user. He always runs his browser with javascript disabled, and is red-green colour-blind. His email client is configured to only render text, and does not external resources (e.g. stylesheets, images).

Gareth is clearly a challenging user, but none of the characteristics are uncommon.

Exercise

Take a moment to reflect on this user journey and the persona described.

1. What difficulties might this persona experience on your user registration journey? What features might be needed to deliver good UX to Gareth?

2. Create another persona with different characteristics. Be creative!

3. What challenges might your new persona experience on the user registration journey?

User Stories

From personas you can create user stories. User stories are the connection between user research and software development.

User stories are the translation of user needs into something a developer can act on. They are behaviour-driven and highly-descriptive accounts of user actions and expected software responses. User stories provide a concrete requirement for a developer to work with. They are also testable. A well-written user story will provide lots of context to help build a complete picture of the desired behaviour. Here's an example of a very simple story:

Given valid non-admin login credentials, Gareth can navigate to `/login` and login with username and password. He is then redirected to `/homepage`. On the homepage the navbar includes links to the account and subscription pages, but does not include any links to site admin functionality.

User stories are extremely helpful, but can take a long time to write and must be written by someone with good knowledge of the user needs. They are frequently very long, and can sometimes be too large to sensibly handle in a single developer task. In that case the development team can divide the story into technical sub-tasks for incremental or parallel delivery.

User stories should *not* suggest any particular implementation details - they focus on user interaction. Agile delivery works best when developers are given requirements to satisfy, and have the freedom to decide how to implement them.

Chapter 4

Architecture

This section takes a brief tour of application and systems architecture, and how you can design systems and applications that meet both technical and user requirements. This topic is extremely broad, and details of any given system will depend on its particular requirements, budget and scope. Nevertheless, there are some common principles and patterns to be aware of. This will help you to understand existing architectures, as well as propose new architectures for your own projects.

A word of warning: software architecture is a huge topic. A basic familiarity with some common architecture principles is helpful, but a comprehensive introduction to architecture is beyond the scope of this course. The additional materials can point you to additional resources.

Building Clean Architecture

It is common practice to split software architecture into multiple specific roles. This separation creates a helpful abstraction, defining different isolated components of a system so they can have their internal details designed and developed in parallel. Two common roles are described below:

1. **Application architecture** is designing the high-level components of a single application. For example, a single application might include an app for Android devices. This might include outlining the central user journeys, as well as describing the high-level relationships between certain classes and system frameworks.
2. **System architecture** designs the relationships between multiple applications, without overly concerning itself with the inner workings of each component. For example, while an application architecture might design an Android app, a systems architect would plan the interactions between that app and a backend API or database, as well as the interactions between that backend and other user-facing services (e.g. a web application and an iOS app).

In both cases, good, clean architecture reflects user requirements and technical requirements. Technical requirements include considerations such as security, scalability, uptime. For many kinds of technical architecture challenges there are known, best-practice solutions, and keeping up-to-date with industry-wide best practices is essential to avoid re-inventing the wheel!

Don't re-invent the wheel

Common architecture patterns exist to solve certain families of problems with related technical and user requirements. Most programming langauges have fully-featured open-source frameworks available for use that implement common patterns. Some standard problems for which out-of-the-box solutions or guidance are readily available include:

1. MVC web applications
2. single page web applications (SPAs)
3. user authentication and authorisation
4. data encryption
5. mobile application architecture
6. database architectures (both OLTP and OLAP)

Some system architectures are totally unique, but many software projects have similar considerations. Work with known patterns where possible, and you can leverage existing tools to help you along the way.

Architecture Diagrams

Architecture diagrams are a tool for designing, and communicating, application and system architecture. You'll find many, varying in style from cartoonish to schematics. They show the key components of a system and their interactions.

Universal modelling language (UML) is one attempt to define a precise standard for architecture diagrams, and it is used extensively in enterprise environments. It's less common elsewhere, but you're bound to come across it at some point.

You can visit the official website, or one of many cheat-sheets online to learn more.

Clean Code Principles

We have already talked about clean code principles and introduced concepts such as YAGNI and SOLID. Many of the same principles can scale up to application and system architecture where many of the same technical principles apply.

Single Responsibility

In day-to-day programming, the single-responsibility principle applies to classes and functions. They should do one thing, and have only one reason to change.

In application architecture, the same approach also to larger-scale components of a system. In reality these are still classes, but typically much higher level ones! This paradigm is enshrined in certain architectures, for example the MVC web application architecture uses the SRP to define the roles of Models, Views and Controllers.

At the system level, the SRP supports a micro-services architecture, where small, logically consistent chunks of functionality run as separate applications in a cluster.

Reducing Tight Coupling

In code, coupling between classes is reduced using interfaces, base classes and other abstractions, depending on the language in question.

At the application and system level, decoupling is usually focused around how to pass data or messages between different components. In applications architecture you'll encounter tools such as message busses and event handlers, which help decouple application components.

At the system level, there are message queues, message brokers and other "pub-sub" (publish-subscribe) tools than traffic data between applications.

Reducing State

Good programming practices make use of immutable objects and collections, as well as functional programming styles, to reduce the state stored in objects. This makes scaling, modifying and debugging code much easier.

At the application level, state cannot be avoided (an application with fixed data is rather uninteresting). Data is instead stored in central locations, keeping as much of the application stateless as possible. Backend data is kept in databases, rather than code, and state on web applications is often delegated to the client, using browser cookies and storage facilities.

At the system level, we can reduce state by adopting immutable infrastructure principles, such as container deployments.

Least Privilege

Least privilege involves granting a user (be that a function, class, app or human) the minimum privileges necessary to perform its function. In code you've used this paradigm to design classes and functions that encapsulate internal logic, exposing a clean API for client code.

When considering architecture, least privilege is analogous to security considerations. You'll use tools such as role-based access control (RBAC), secure VPNs and IP whitelists to control access to resources.

User Requirements & Architecture

User requirements can have a profound impact on architecture decisions, and are not restricted to the UI layer. User stories inform architecture decisions. Architecture should be designed to support known user stories, and provide a flexible framework for adding more at a later stage. User stories should not be forced retrospectively on an architecture that does not support them!

Architecture decisions can be prohibitively costly to reverse once implemented. Getting this right is important!

There are many domain-specific ways user stories might impact architecture. Consider the examples below, which might be applicable to a complex public-facing web app (e.g. an e-commerce platform):

1. **User authentication flow** - will you manage user credentials directly, or integration with third-party OAuth providers? Perhaps both?
2. **Service uptime** - What guarantees do you provide about service availability? Is it acceptable to have 5 minutes downtime while a production VM is upgraded?
3. **Data storage** - What regulatory requirements exist? Must data for specific users be stored in particular locations?
4. **User research strategies** - Will our architecture support A/B testing? What user analytics do we need, and what service or architecture best supports that?
5. **User feedback** - How will a user know a requested operation is successful? Do they need a quick in-browser response, or is a follow-up email notification acceptable? This has impacts on back-end data processing and messaging.

DevOps engineers commonly face unfamiliar and non-standard challenges that require creative, ambitious and/or exploratory solutions. This is somewhat inevitable - good DevOps practices automate routine and simple tasks, leaving only the complicated or challenging ones for human attention!

Chapter 5

Problem Solving

It's useful to have some standard problem-solving approaches in your toolkit to apply to unfamiliar technological and business problems. Even if you have a solution in mind, these can help ensure you've not missed anything and "sanity-check" your approach. A well-understood problem-solving methodology can also help demonstrate to external parties that your approach is well-considered and effective.

This section is not about small technical problem-solving (e.g. debugging), but rather 'bigger picture' challenges. For example, you may be asked to design a new service, improve team efficiency or propose ways to improve conversation rates on a web app. The tools discussed here can help.

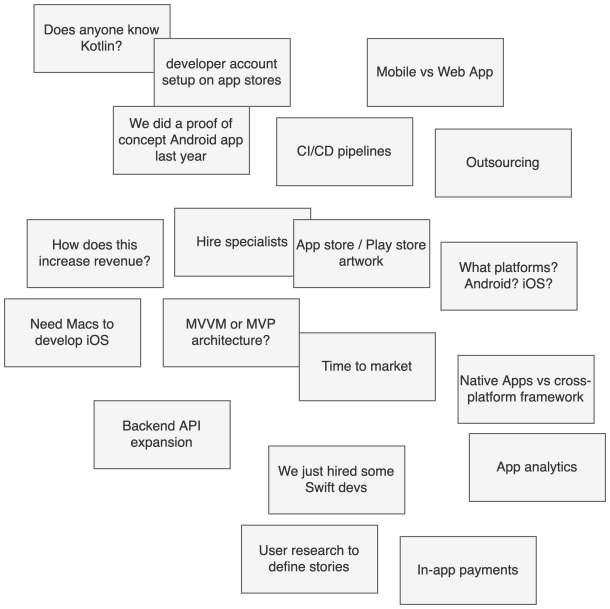
Affinity Mapping

Affinity mapping is one approach to exploring a problem space and gaining new insight. It's a team exercise that starts with collecting as many thoughts as possible on the topic (brainstorming), organically grouping them and drawing out meaningful relationships between groups. It's a way to gain fresh perspective on a challenge.

It's important to treat this as a group activity, and try to get as diverse a team as possible. Different viewpoints provide different insights, and affinity mapping helps you get them all in one place. Once you've got ideas on the table, try to assemble them into a a coherent story. In an office, you'd use a whiteboard or post-it notes to assemble and group ideas. Online you can use a digital whiteboard to similar effect.

Step 1: Gather Ideas

The first step is to gather as many ideas as possible! Have everyone grab a stack of post-it notes (or digital equivalent) and put on the board any ideas that come to mind. This should be unfiltered and organic. The image shown is an example - can you guess the topic?



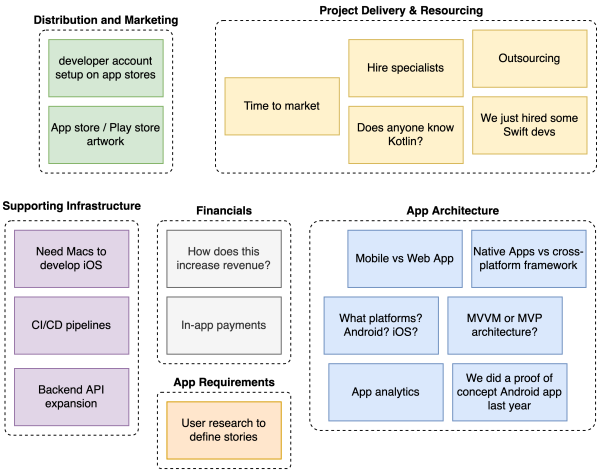
Affinity Mapping Ideas

Step 2: Group Ideas

Once everyone is done, you need to group ideas together. Take ideas one at a time, each time asking "Is this related to a previous idea, or is it something different?" Group ideas based on the team's answers. At this stage you don't need to know what each group represents.

In our example affinity mapping session, a fictional business is planning a mobile app. The group might start by identifying that the ideas "Hire specialists" and "We just hired some Swift devs" are related, but are not in the same group as "App analytics". Continue this process until every idea is grouped with others, or identified as a standalone idea.

Now you've organically grouped ideas together, it's time to identify what each group represents and give it a name. There's not always a single answer to this (indeed, different teams might assemble different groups entirely), but try to find a team consensus.



Affinity Mapping Groups

Step 3: Analyse

With groups defined, your team can consider the relationships between them. Tasks and ideas in one group might block progress in another area. In our example, "Resourcing" is likely a pre-requisite for "App Architecture" - it is, after all, advisable to have subject matter experts on hand before making technical architecture decisions!

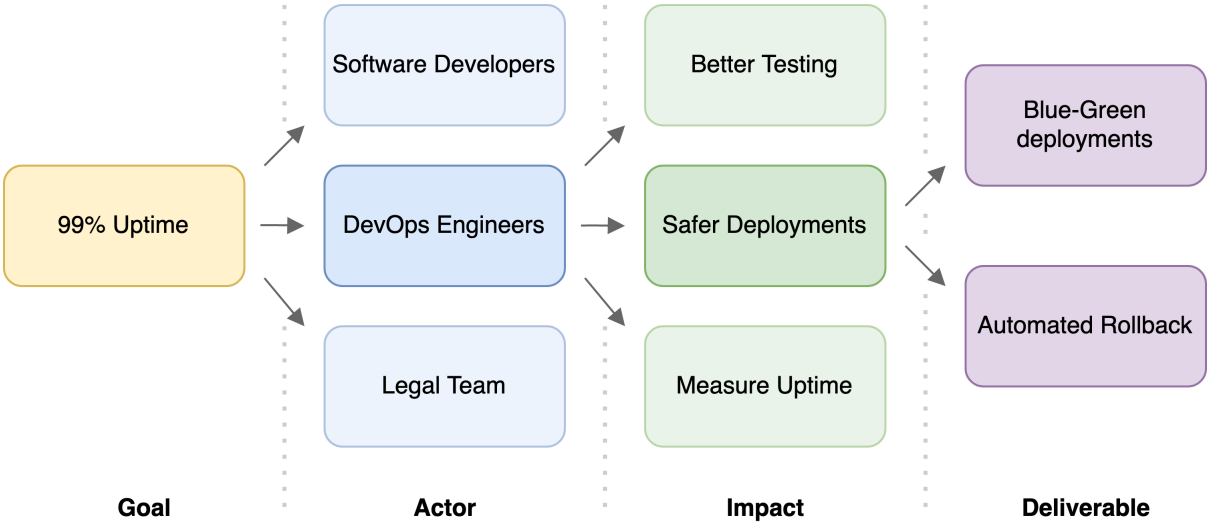
Ideas in each group can be prioritised, and converted into tasks for further development outside the affinity mapping session.

Remember, the goal of affinity mapping is to draw out as many ideas as possible, organise them into groups and use the groups to inform future direction. The diversity of tasks and groups created reflects the diversity of viewpoints represented in the session.

Impact Mapping

Impact mapping is a management tool and problem solving technique that clearly links deliverables (e.g. software features) back to driving business goals. The idea is that, with greater context, deliverables are more likely to accomplish their original goals and the business will invest less time and effort in projects poorly aligned to business goals.

The technique links **goals** to **deliverables** through **actors** and **impacts**. Actors are those who can help you reach your goal. They can produce certain impacts through concrete deliverables. In an agile setting, each deliverable can be mapped to one or more user stories.



Impact Mapping

Plan-Do-Check-Act

Plan-Do-Check-Act (PDCA) is an iterative problem-solving methodology that can be applied to a range of challenges. It emphasises planning, testing changes or new processes, and analysing the results. It's an evidence-based approach to solving problems, and is broadly applicable to many fields.

- **Plan**. Review your current position and compare to where you would like to be. Plan out the steps needed to get there, and how you will measure progress.
- **Do**. Test out your plan on a small scale, gathering any feedback or metrics necessary.
- **Check**. Review your progress using metrics defined in the planning stage. Did you meet your goals?
- **Act**. If successful, roll out the solution more widely.

The PDCA cycle should sound familiar! It can be compared to an agile sprint, and the scientific, evidence-based approach is similar to the data-driven, user-centric development we have advocated. It's a helpful framework that you can apply to solving both technical and non-technical challenges.

The methodology is also known as the Deming cycle, after William Deming, a management consultant to whom the method is widely attributed.

Chapter 6

Additional Material

Agile Terminology

There are many ways to approach agile development, and there are a huge number of terms commonly used to describe certain roles, events and structures within the workflow.

Flavours

There are two main 'flavours' of agile development: **kanban** and **scrum**. They share the same agile philosophy and values, but differ significantly in how they structure their workflows.

Scrum is largely what we have described in the core material. Large projects are split into a series of smaller sprints, each with well defined deliverables. Scrum teams will appoint a **scrum master** who is responsible for running regular sprint ceremonies.

Kanban teams are less sprint-oriented, and instead focus on continually delivering work and limiting work-in-progress. There are fewer structures and fixed ceremonies, compared to a scrum approach.

You can read more about these differing flavours of agile here.

Ceremonies

Agile ceremonies are meetings. Many different kinds of ceremonies are often included in agile workflows. You don't need to do all of these! Instead, pick the ones that make sense for your team. Remember, meetings are expensive and often dull - pick ones that add real value to your team, and look for ways to keep them quick and focused.

Here's a few of the common ones:

Ceremony	Purpose
Backlog refinement	Review the task backlog, and prioritise tasks for development.
Sprint planning	Plan goals and task assignment for an upcoming sprint.
Standup	Daily meeting where team members update each other on progress.
Demonstration (demo)	A demonstration of recent progress, usually involving any external customer. Frequently held at the end of each sprint.
Sprint review	Review performance of the recent sprint, often including aspects of sprint planning and backlog refinement.
Retrospective (retro)	An internal team meeting to reflect honestly about what we went, and what could be improved about the project and workflow.

How to get the most from Agile

It's important that Agile ceremonies are quick, focused and useful. You've probably been in a meeting where one participant derails discussion by bringing up a important, but out-of-scope topic, or where someone can't remember a key detail and has to look through emails while everyone waits! Agile tries to avoid this - each ceremony has a clear purpose, and, with a bit of preparation, you can keep them short and effective.

- **Standups**: Before the meeting, note down a few quick notes about what you worked on yesterday, and what your task assignments look like for today. This will help keep your update short and useful.

- **Retros**: Take a few minutes earlier in the day to reflect on the current sprint. Try to come up with a few positive and negative points, as well as some improvement ideas to discuss with the team. Double-check that your points are helpful and constructive.

- **Demo**: Decide in advance which features you're going to demo, and how. Do a test run to make sure it runs smoothly. Make sure everything you need (e.g. test data or credentials) is in place. A demo plagued with technical issues does little to build trust in the development process!

Remote Agile?

Remote working can be challenging for a variety of reasons. Team members may struggle to find appropriate work space, create a focused environment or balance work with childcare commitments. There may be technical barriers to remote work if infrastructure is poor. For some people it is simply challenging to be productive without colleagues, face-to-face conversations and typical office amenities.

These challenges affect all teams, whether following agile workflows or not. For Agile teams, remote working has the potential to impact development, communication, user feedback and knowledge sharing. Remote working can also impact team morale, further damaging productivity if not managed well.

Fortunately, with good processes, and good DevOps, Agile teams can work extremely efficiently in a fully-remote setup. Here are some tips:

- Have access to reliable text chat and video-conferencing software
- Prefer voice and video chat to text communication
- Proactively seek out conversation as regularly as you would if sat next to your colleagues!
- Hold all agile ceremonies virtually (even if two or more team members are co-located)
- Support team-members setting up a productive home working environment
- Make the DevOps toolchain accessible from everywhere (i.e. not restricted to a network)

These recommendations aim to reenforce team social cohesion, while also ensuring that distance causes little disruption to day-to-day work.

Agile at Scale

Agile is usually envisaged and described on the scale of a small team, but how does it work at scale? This is a wide topic, more related to management and entrepreneurship than DevOps and programming. Here are a few discussions you might find interesting:

1. https://hbr.org/2018/05/agile-at-scale
2. https://www.atlassian.com/agile/agile-at-scale

More on Accessibility

You can think of accessibility as another requirement, alongside more technical non-functional requirements like security, or responsiveness. You can test accessibility and you can automate those tests using tools like axe.

There's a lot written about accessibility, and good UI and UX in general, online. The gov.uk accessibility blog is a great place to learn about the thought that goes into well-designed digital public services.

Architecture Patterns

Software architectures often fall into common patterns. These patterns have names like "client-server" and "micro-services", but there are many more! You can read more about software architecture, and some common architecture patterns, at the resources below:

- https://towardsdatascience.com/10-common-software-architectural-patterns-in-a-nutshell-a0b47a1e9013
- https://mitsloan.mit.edu/ideas-made-to-matter/agile-scale-explained
- https://netflixtechblog.com/