A Practical Guide to

Testing in DevOps

KATRINA CLOKIE

A Practical Guide to Testing in DevOps

Katrina Clokie

This book is for sale at http://leanpub.com/testingindevops

This version was published on 2017-08-01

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

For my Dad

Contents

Preface

This book is for testers who want to understand DevOps and what it means for their role. It's also for people in other roles who want to know more about how testing fits into a DevOps model. I hope to encourage people to explore the opportunities for testing in DevOps, and to become curious about how they might approach quality in a different way.

The following pages give examples of test practices that rely on the development and operations teams working together in a DevOps culture. I've completed a year of research to source material for this book to supplement my own expertise. My goal was to provide a consolidated reference with a practical focus. Some of the ideas presented in this book are simply an introduction. The references section provides detail that I hope readers will explore further, diving into areas that interest them.

There are a few people to thank for their help to make this book possible.

Two people provided the impetus to start writing. My friend Lisa Grant set herself brave and humorous New Year's Resolutions that inspired me to make my own. The words of my former manager Noel Dykes challenged me to set a bold goal.

I have been fortunate to have three dependable reviewers with me on this journey. David Greenlees, always the quickest to read through the newest chapter and able to pinpoint gaps in my thinking. Shirley Tricker, who would ask questions that opened up a new lens on each passage. Sarah Burgess, with her thorough red pen to extract the simplest meaning from my words.

The images and colour palette of the book have been created by the talented Natasha Nath who took my hand-drawn sketches and converted them into professional graphics.

Many examples through the book are pulled from experience reports. Thank you for being transparent so that others can learn from your path, especially those at King, Capital One, The Guardian, Etsy, Spotify, and PagerDuty who feature in the industry examples section.

Thank you to my own organisation, the Bank of New Zealand, including the delivery team who provided information about their A/B testing, those in leadership who endorsed the inclusion of this example in the book, and my colleagues in the practices team who have provided encouragement.

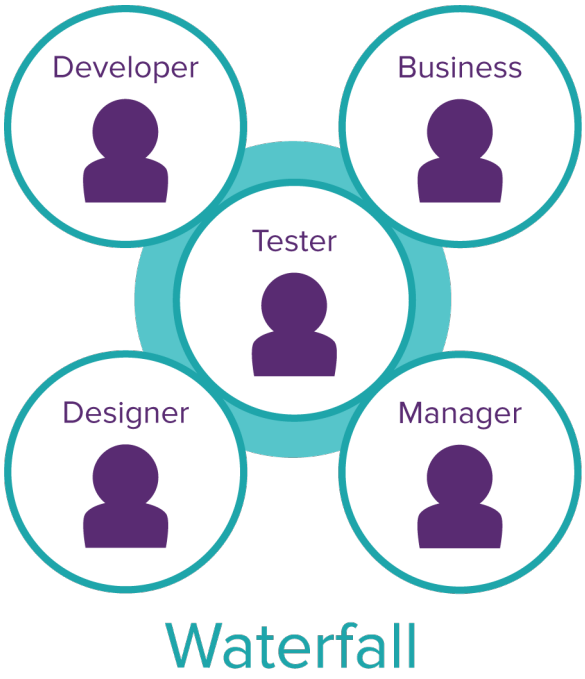
This book is dedicated to my Dad. The publication date would have been his 65th birthday. In his absence, my Mum and my sister provide daily examples of determination and energy. Thank you for driving me to keep up! Finally, thank you to my husband Richard for your support and wisdom.

Katrina Clokie
Wellington, New Zealand
1st August 2017

Background

DevOps is the latest evolution in a history of changes to software development. Waterfall to Agile, Agile to DevOps, each step has created corresponding change in the role and activities of the tester. The steps may not be distinct in every organisation, but for simplicity let's consider them separately.

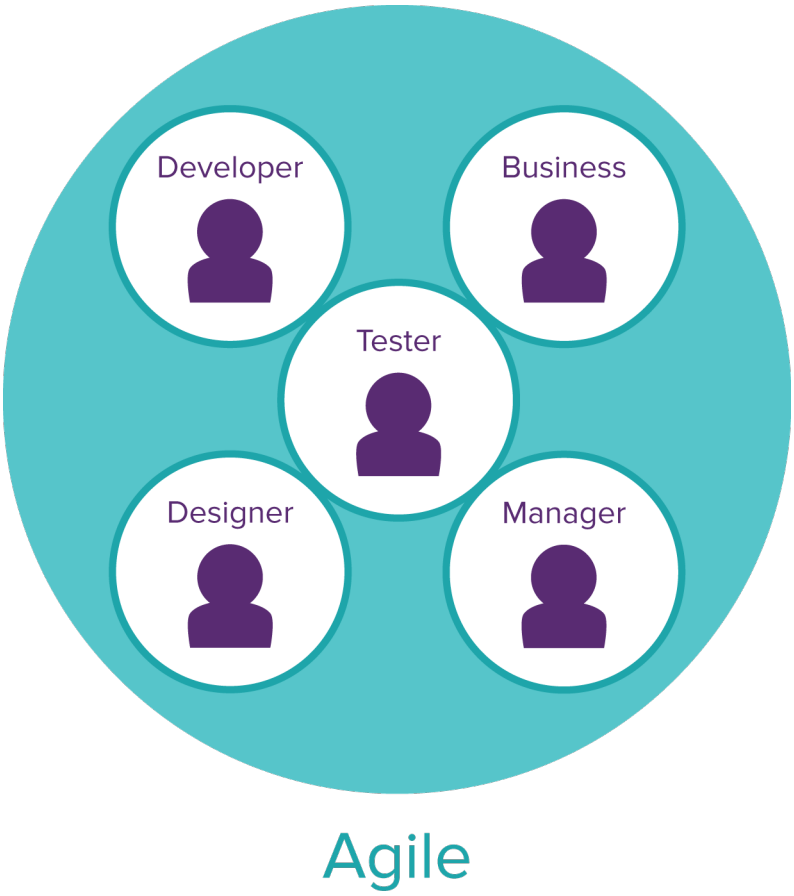
In a waterfall environment, testers have sole ownership of test strategy and execution. We revel in thinking up our own test ideas, independently exploring the software, locating and then reporting problems that we discover. Though we work with developers, business analysts, designers, test managers, and others, there is comfort in knowing that testing as an activity belongs to us. Testing is our role and our identity in the team.



In an agile environment, testing becomes an activity that the development team own together. The team think of test ideas. The team explore the software. With a variety of perspectives, the team find a variety of problems to discuss and resolve together.

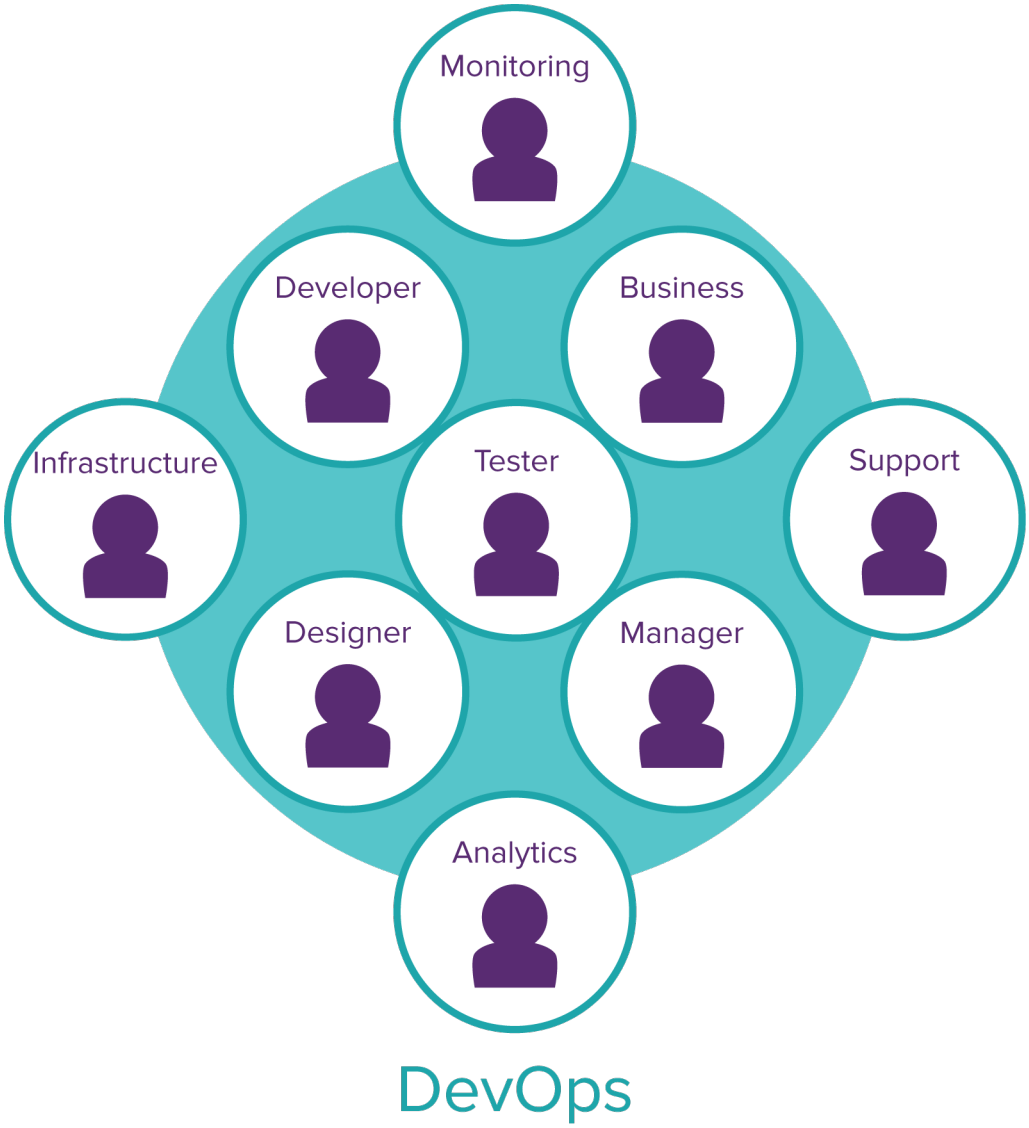
The relationship between testing and the business changes too. Discussions about requirements became more collaborative through Behaviour Driven Development, the importance of acceptance testing is diluted by day-to-day business involvement in the development process, and fast feedback on iterative releases tempers the need for exhaustive pre-release checks. All these activities influence testing.

Agile challenges our definition of testing and ownership of testing as an activity. It changes the scope and pace of testing, engulfing the whole development team.



DevOps takes this challenge further.

DevOps pushes the development team to actively engage more widely within their organisation, beyond the roles that are involved in development to those that support it: operations infrastructure and monitoring, customer support, and analytics. This change broadens the network of people who collaborate, which will further extend the boundaries and nature of testing.



Understanding the skills, practices and tools available in areas that have historically been peripheral to the development team creates opportunities to push test activities outwards. Examples include on-demand cloud-based infrastructure that enables testing in a production-like environment, feedback from A/B test experiments provided by customer metrics, or beta testing groups that offer rapid customer feedback.

On the flipside, information into the development team also changes. Problems that are discovered in real-time monitoring dashboards, customer support tickets, or unusual analytics reports are received directly for prioritisation and resolution.

The other facet of DevOps is speed. The aim of fostering a more collaborative culture is to achieve rapid and reliable releases. The pace of DevOps can be particularly challenging to testers who may struggle to find the right balance of investigating new features without impeding the release.

Automation is usually touted as the solution to speeding up testing, particularly by those who are driving change. Smart use of tools is usually the best way to go faster, though this need not be through automating testing within the development team. Improving production monitoring and alerting, coupled with a rapid automated deploy and rollback, could be just as effective.

Several definitions of DevOps exist in the software community. Some are tool-focused definitions that have strayed from the original intentions of the movement. To help you read this book, here are the definitions we will use. This section also explains DevOps, its relationship to continuous delivery and agile, and where testing fits.

What is DevOps?

The DevOps movement was born in 2009. The focus of the movement is to bridge the gap between development and operations, to reduce the risk in deployment of software.

> "... there's still a gaping hole in what Chris Read calls 'the last mile'. The thing is, 'dev complete' is a long long way from 'live, in production, stable, making money'.
>
> The problem is that it's typically the sysadmins who are responsible for getting the software out live - but often they don't know, trust, like, or even work in the same city as the developers. How on earth are we expected to deliver good software that way?"
>
> What is this DevOps thing anyway?[1]
>
> *Stephen Nelson-Smith*

The original DevOps material focused on empowering people to work across disciplines to create a more reliable release process. The emphasis was on creating trust, both between people and in their processes and tools.

Releasing without fear encourages people to release more often. John Allspaw and Paul Hammond highlighted this benefit at VelocityConf in 2009 where they spoke of their 10 deploys per day through Dev and Ops cooperation at Flickr[2]. At the time, this was an impressive release cadence.

DevOps hit the mainstream in 2013 when The Phoenix Project[3] was published. Written by Gene Kim, Kevin Behr, and George Stafford, the book is a novel that illustrates DevOps principles through fictional characters. Widely read, and with largely positive reviews, this book is how many people have become familiar with the term DevOps.

I read widely for a definition of DevOps that resonated. The one that I liked best came from the collaborative input of many people and captured the broad scope of DevOps. Wikipedia defines DevOps as:

[1] http://www.jedi.be/blog/2010/02/12/what-is-this-devops-thing-anyway/

[2] http://www.slideshare.net/jallspaw/10-deploys-per-day-dev-and-ops-cooperation-at-flickr/10-Ops_job_is_NOT_to

[3] https://www.amazon.com/Phoenix-Project-DevOps-Helping-Business/dp/0988262592/

> **DevOps** (a clipped compound of development and operations) is a term used to refer to a set of practices that emphasizes the collaboration and communication of both software developers and other information-technology (IT) professionals while automating the process of software delivery and infrastructure changes. It aims at establishing a culture and environment where building, testing, and releasing software can happen rapidly, frequently, and more reliably.
>
> Wikipedia[4]

DevOps and continuous delivery

DevOps and continuous delivery are movements that developed in parallel, with closely aligned timelines and similar goals. This can create some confusion about the difference between the two.

In 2009, the same year that DevOps originated, Timothy Fitz coined the term continuous deployment.

> "Continuous Deployment is simple: just ship your code to customers as often as possible."
>
> Continuous Deployment[5]
>
> *Timothy Fritz*

This was followed a little over a year later by the publication of a book titled Continuous Delivery[6]. This term from Jez Humble and David Farley held a slightly different intent and became more prevalent.

> "Implementing continuous delivery means making sure your software is always production ready throughout its entire lifecycle - that any build could potentially be released to users at the touch of a button using a fully automated process in a matter of seconds or minutes."
>
> Continuous Delivery vs Continuous Deployment[7]
>
> *Jez Humble*

Continuous delivery has a smaller scope than DevOps. It focuses on the technical practices that allow a development team to move quickly. It includes coding practices, source control management, and concepts such as feature flagging, which allows code to be deployed to production without being available to users. Jez Humble summarises what is necessary to achieve continuous delivery as:

[4] https://en.wikipedia.org/wiki/DevOps
[5] http://timothyfitz.com/2009/02/08/continuous-deployment/
[6] https://www.amazon.com/Continuous-Delivery-Deployment-Automation-Addison-Wesley/dp/0321601912
[7] https://continuousdelivery.com/2010/08/continuous-delivery-vs-continuous-deployment/

> "... ensuring our code is always in a deployable state, even in the face of teams of thousands of developers making changes on a daily basis. We thus completely eliminate the integration, testing and hardening phases that traditionally followed "dev complete", as well as code freezes."
>
> Continuous Delivery[8]
>
> *Jez Humble*

It is possible for the development teams in an organisation to adopt continuous delivery without any change in their operations team. This limited scope may be a sensible first step in improving the release process, but eventually it will create strain due to dependencies between teams that operate with a different rhythm. DevOps is the broader culture change that addresses this tension.

DevOps and Agile

Agile, at its genesis, was a manifesto with 12 accompanying principles. Its goal was to challenge the mindset of people involved in software development. There was no implementation detail, though adopting an 'agile' way of thinking would lead to implementation change within an organisation.

DevOps is more directive. It focuses on the relationship between development and operations. DevOps has a single desired outcome - to improve the reliability and frequency of releases.

Without agile, DevOps may never have emerged. When people within the development team start to think differently and work collaboratively under agile, they are able to regularly create smaller pieces of useful software. This causes tension with other areas of the organisation who have been accustomed to a slower process. DevOps addresses this tension by encouraging those who develop the software to work closely with those who support it.

But can you do DevOps without doing agile?

Yes. DevOps can strengthen relationships between people who are using any methodology. Improving reliability of releases could mean reducing the number of support patches for a product or decreasing the outage window that users experience. More frequent releases could be a target. In a non-agile environment, a release might be monthly or yearly. DevOps doesn't necessarily mean releasing every minute or hour.

However, DevOps is a harder path to follow without agile. People from different disciplines may not be comfortable working closely together. Agile provides the groundwork, so this book targets people who are looking to apply DevOps in an agile context.

DevOps and testing

Testing is conspicuous by its absence from most literature about DevOps. By nature, the focus is on developers and operations. Understandably this tends to make testers nervous about their role.

[8] https://continuousdelivery.com/

The people involved in shaping these communities fail to highlight testing because they are not from specialist testing backgrounds. This does not mean that testing isn't happening, but rather that it is a pervasive activity throughout development, that isn't considered notable.

In reference to the DevOps tool chain, Dan Ashby writes:

> "... you can see why people struggle to understand where testing fits in a model that doesn't mention it at all. For me, testing fits at each and every single point in this model."
>
> Continuous Testing in DevOps[9]
>
> *Dan Ashby*

Dan visualises this as:

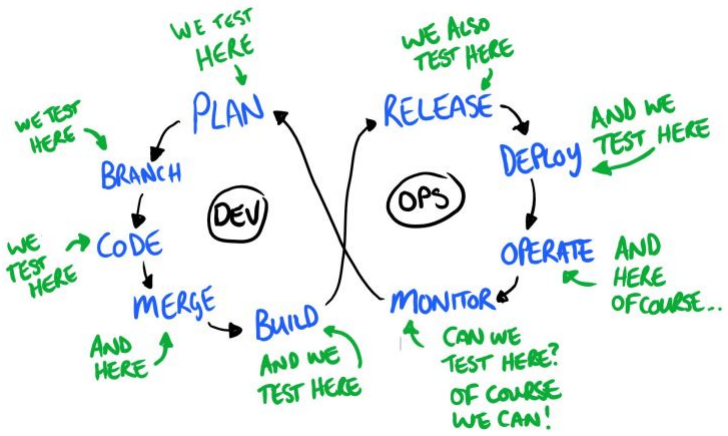


Image credit: Dan Ashby, Continuous Testing in DevOps[10]

In their book on continuous delivery, Jez Humble and Dave Farley summarise the place of testing in a similar vein:

> "Testing is a cross functional activity that involves the whole team, and should be done continuously from the beginning of the project."
>
> Continuous Delivery[11]
>
> *Jez Humble & Dave Farley*

[9] https://danashby.co.uk/2016/10/19/continuous-testing-in-devops/
[10] https://danashby.co.uk/2016/10/19/continuous-testing-in-devops/
[11] https://www.amazon.com/Continuous-Delivery-Deployment-Automation-Addison-Wesley/dp/0321601912

The difficulty for testers is identifying what their role is, when testing is expected to be a part of everything. What exactly is your responsibility? How should you try to contribute?

Beyond the testing role, there are questions for anyone involved in testing as an activity. Is there benefit in distinguishing testing tasks from other types of work in the development team? What weighting should you give to testing?

TESTING IN A DEVOPS CULTURE

How you experience DevOps culture will depend on your existing test approach, the vision for DevOps in your organisation, and how you start to collaborate beyond the development team. This section discusses these three aspects.

Before changing to a DevOps approach, understand the context in which you currently operate. Talk with your team about your existing approach to testing and the potential that they see for improvement. Compare your practices to others in the industry and use this to start discussions about areas of difference. You may not standardise with others, but it can be useful to talk about adopting new ideas in your organisation.

Understand your leaders' vision for DevOps. It is important to understand differences in three areas:

- how the vision is expressed at all levels of the organisational hierarchy
- how the pieces come together to achieve a single goal
- where your own contributions fit.

Understand how you can influence culture through the collaboration that you foster across the organisation. Work to identify and connect with new people beyond the development team. Build relationships that invite collaboration from others in your discipline.

Establishing your context

Before introducing testing in DevOps it's a good idea to reflect on your current approach and check that there's a shared vision for the future. This section explains how to run a test strategy retrospective. It also encourages you to learn more about the specific goals of DevOps for your organisation.

Start with a test strategy retrospective

A test strategy retrospective can help an established agile team to check that everyone is on the same page by:

- determining whether everyone in the team knows what the test strategy is,
- agreeing on which aspects of the strategy have been implemented.

Even without a *documented* test strategy, people will have their own ideas about what types of testing activities should or should not be occurring. It is worth periodically checking that everyone is on the same page.

You may argue that strategy is one aspect of testing that remains the responsibility of the tester. You might think the specialist tester is the only person in a cross-functional team able to define a test strategy, given that testing is their area of skill. However the method by which a strategy is decided and shared is important. A tester who does not make the process of how they arrived at their strategic decisions clear to their team is adopting a high level of risk by taking ownership of choices that may not be theirs to make.

During testing, a specialist tester will be making conscious decisions that alter the test strategy of the team. They will be actively thinking about the trade-off in adopting one practice over another, the implications to test coverage and the impact on the overall quality of the product. But they are often thinking and deciding as an individual without sharing their strategy changes with the team. Others may be making decisions that also alter the test strategy, but less consciously.

It can be the case in an agile team that testing is considered open to all, yet strategic thought about testing is not. This means that people testing may be adopting a practice without understanding why. A test strategy retrospective helps the team understand the test strategy they are using and any decisions that underpin it.

Preparation

A test strategy retrospective is designed to be engaging and interactive; to get people who are not testers to think about what types of testing are happening and why. It should take approximately one hour.

The specialist tester should facilitate this retrospective but not participate in creating the visualisation. This prevents the team from being led by the opinion of the facilitator, and ensures that everyone has the opportunity to contribute.

To run a test strategy retrospective you will need:

- one hour where the whole team is available
- sticky notes in four different colours
- a large surface to apply the sticky notes to.

A large boardroom table is ideal, as it allows the team to gather around all four sides. A large, empty wall is a good second choice.

Creating a visualisation

Start the retrospective by clearly stating the purpose to those gathered. Explain that you are going to visualise your test strategy and foster a shared understanding in the team about what testing is happening.

Take two sticky notes, one labelled IDEA and the other labelled PRODUCTION. These are placed at the top left corner and top right corner of the surface, creating a timeline that reflects the process of software development from an idea to a deployed application.

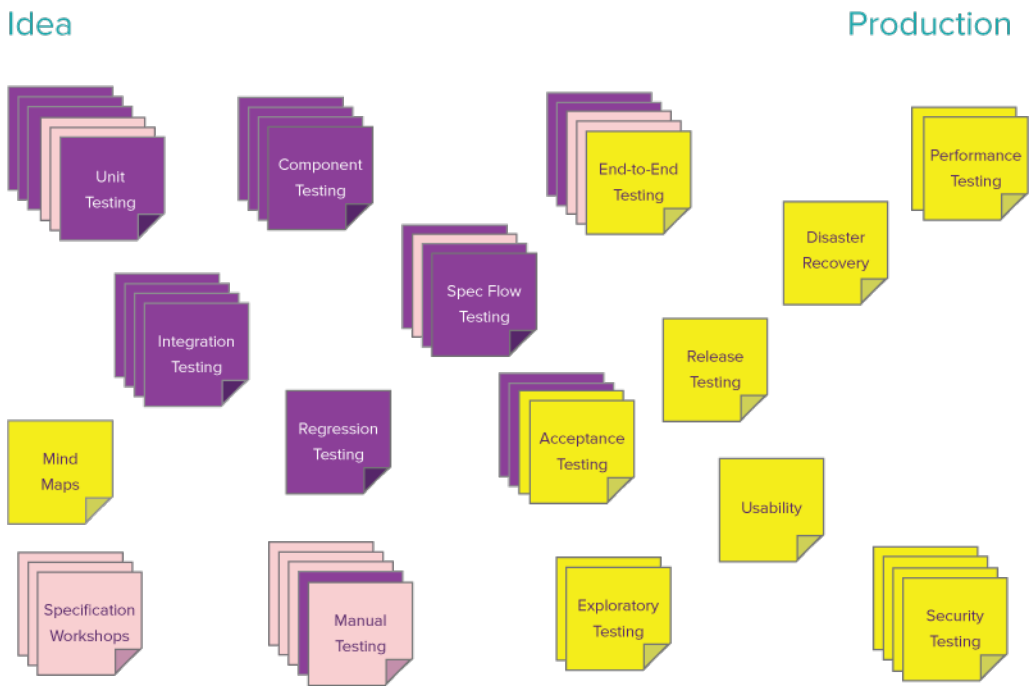
In this example, coloured sticky notes are used to mean:

- Purple - in our test strategy and we're doing it
- Pink - in our test strategy and we're not doing it
- Yellow - not in our test strategy, but should be

Each individual should put their sticky notes onto the timeline at the point they think the test activity will occur. At the end of the five minutes there will be a haphazard display of sticky notes.

Ask the team to work together to group the sticky notes that have testing activities with the same name, and agree on the placement of activities within the timeline. Where different names have been used to refer to the same concept, keep these items separate. Once the team are happy with their visualisation, or the conversation starts to circle, call a halt.

Here is an example:



Leading a discussion on strategy

If you've reached this point of the retrospective after ending a circular thread of conversation then that may be the first place to start a discussion. But there are a number of other questions to ask of this visualisation.

- Are there groupings that include different coloured sticky notes? Why?
- Have people used different terminology to refer to the same type of test activity? Why?
- Are there activities in the test strategy that aren't being implemented? Why?
- What activities are missing from the strategy? Do we want to include these?
- Are any activities missing from the visualisation altogether? What are they?

These questions uncover misunderstandings about the current state of testing. They also highlight the decisions that shaped the current strategy. The visualisation is a product of the whole team and they are more likely to be invested in it.

When I did this with a team, the delivery team had some interesting conversations.

End-to-end testing appeared in all three colours of sticky note: some people thought it was in the test strategy and being done, some people thought it was in the test strategy but not being done, and some people thought it was not in the strategy at all. The team were working in a complex architecture that involved a number of internal and third party systems. The different coloured notes indicated disagreement about what end-to-end meant for their environment.

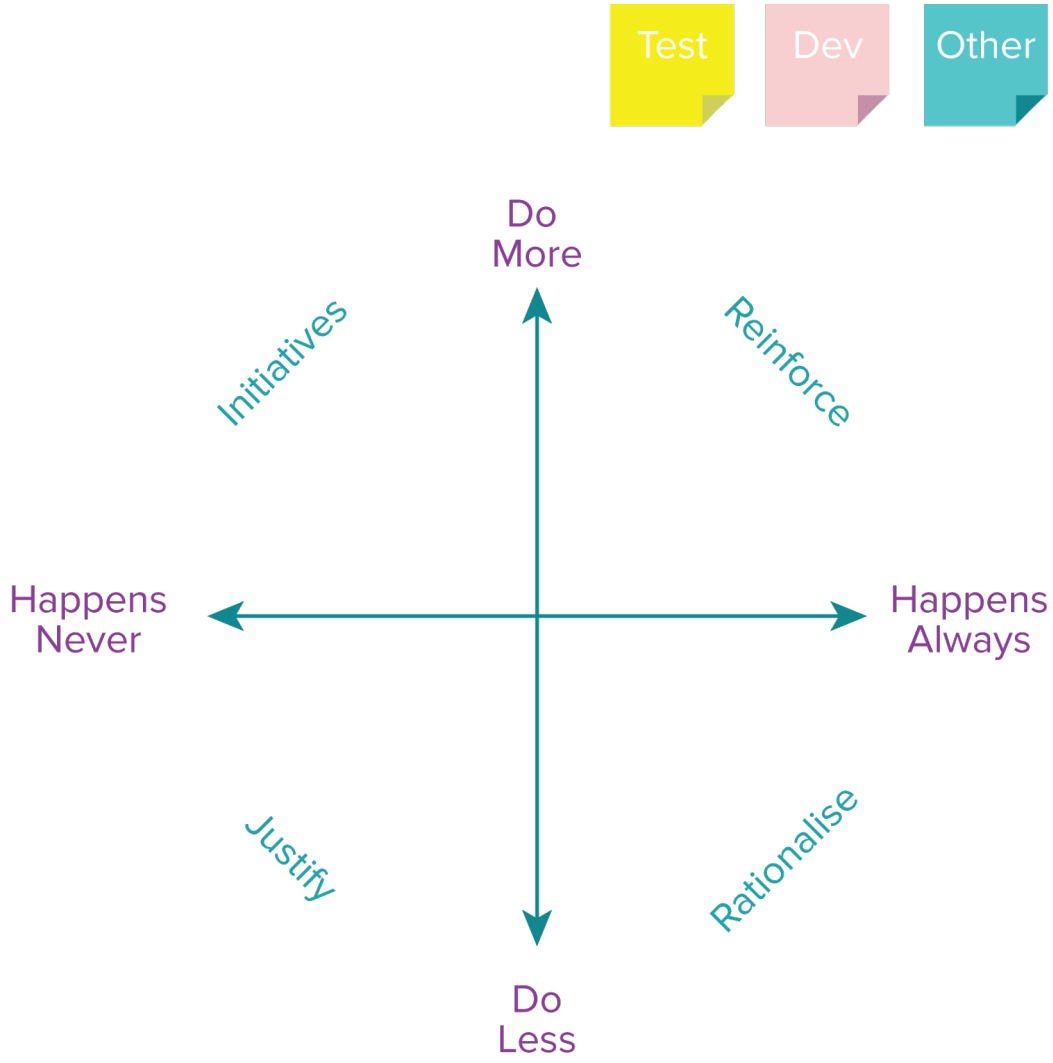
There were separate sticky notes for ‘manual testing’, ‘exploratory testing’, and ‘acceptance testing’. When pressed, the team realised that they were using these terms interchangeably, which was creating confusion. They decided to settle on ‘exploratory testing’.

While the majority felt unit testing was being done, there were also people who disagreed. Worryingly, these people were developers! The retrospective revealed that one piece of the architecture was unsuited to unit testing, something that the business people and testers in the team were unaware of.

An alternative approach

Sean Cresswell took the test strategy retrospective visualisation and developed an alternative approach.

Sean asked each person to write a set of sticky notes that each named one test activity. The colour of each sticky note showed who performed the activity: tester, developer, or other roles. The results were then placed on a matrix to show how often the activity happened compared to how often it should be happening.



Here is an example of a test strategy retrospective visualisation using this approach:

Image credit: Sean Cresswell, Twitter[12]

This format will drive a slightly different conversation. Sean's approach compares current testing to ideal, rather than current testing to the previously set strategy. There is less focus on the sequencing of activities, so it is more difficult to spot gaps in the strategy. There is more focus on who is completing activities, which may undermine the team ownership of testing.

On the other hand, this approach may offer an opportunity to recognise cross-discipline contributions. Gary Miller had this experience:

> "Immediately I saw something that I didn't expect to see – a sea of orange colored stickies (the color assigned to development). I was surprised to see the high level of

[12] https://twitter.com/seancresswell/status/499394371372843008

> contribution to testing by the developers and that was challenging my assumptions in a good way. I made sure to highlight and acknowledge their contribution to testing on the team, as I felt this was something they don't get enough credit for. If they hadn't plotted their contributions, I might have never been aware of them."
>
> A Test Strategy Retrospective Experience[13]
>
> *Gary Miller*

In a cross-functional team where anyone can perform testing, it is important to have a shared understanding of both the practical approach to testing tasks and the underlying test strategy. Agreement about the test activities means any person who picks up a testing task understands the wider context in which they operate. This helps them understand the boundaries of their task: what they should test and what sits within another activity.

[13] https://softwaretestkitchen.com/2015/01/27/a-test-strategy-retrospective-experience/

Agile testing assessment

DevOps is more than just agile testing, but involves many of the same activities and approaches. If you're not comfortable testing in an agile environment, you may struggle to leap straight into a DevOps culture.

Why?

People who work in agile are familiar with testing in a collaborative manner. Everyone is encouraged to develop an understanding of the requirements and what testing is required. Everyone is supported to perform testing, through exploration or by executing test automation. Everyone responds to problems rapidly, whether reported by an individual or a continuous integration server. These things are not left to the tester alone.

If these ideas do not resonate for your development team, there may be an interim step before moving to DevOps.

Karen Greaves has compiled a list of ten questions that can help you to assess your existing agile testing practices. Use these to see if you are ready for DevOps.

How agile is your testing?

Score one point for each statement that is TRUE

1. The whole team is clear on what should be tested for each story and feature before any coding starts.
2. Before you discuss the solution, you make sure you understand the who and why behind any requirement.
3. You ask, and answer the question "How will we test that?" when discussing a user story.
4. Everyone on the team knows how to run the automated tests and read the results.
5. You discuss what you will automate at which level so that you don't duplicate tests between the unit, component and UI levels.
6. Your test scripts are version controlled and labelled along with the source code, since tests are part of the working software.
7. You don't have a bug database because you fix bugs as soon as you find them, instead of logging them.
8. When your continuous integration server fails, it is addressed and returned to a working state within an hour.
9. When observing your standup meeting an outsider would not be able to tell who is a developer and who is a tester.
10. Your team has a way to measure quality, which you use to identify if your testing process is working for you.

Assessment - How agile is your testing?[a]

Karen Greaves

[a] http://www.growingagile.co.za/2015/10/assessment-how-agile-is-your-testing/

If your team scored less than five, you may find these books useful:

Agile Testing: A Practical Guide for Testers and Agile Teams[14] *Lisa Crispin & Janet Gregory*

More Agile Testing: Learning Journeys for the Whole Team[15] *Lisa Crispin & Janet Gregory*

Growing Agile: A Coach's Guide to Agile Testing[16] *Samantha Laing & Karen Greaves*

[14] https://www.amazon.com/Agile-Testing-Practical-Guide-Testers/dp/0321534468

[15] https://www.amazon.com/More-Agile-Testing-Addison-Wesley-Signature/dp/0321967054

[16] https://leanpub.com/AgileTesting

DevOps in your organisation

DevOps has a broad scope and can look different in different organisations. The first step in developing an effective test approach is to understand the context in which you're working. You need to understand what DevOps means in your situation and become confident articulating your goals.

Start by identifying the people who are involved in delivering your product, both in and beyond your development team. Include developers, business analysts, designers, product owners, operations engineers, agile coaches, delivery managers, and any other relevant roles in your organisation.

DevOps affects all these roles. Once you've thought about who these people are, try to find out what DevOps means to them.

People who occupy different roles will hold different perspectives. A management explanation will probably focus on outcome, e.g. "the work we are doing to move to a daily release". By contrast, a developer will probably focus on implementation changes towards that goal, e.g. "the work we are doing to switch our branching model to trunk-based development".

The variety in response is a good thing. Managers should provide a clear vision and rationale for change but avoid dictating the specifics. Those involved in implementing the vision should focus on the next incremental improvement. By understanding these different perspectives, you develop a rounded understanding of DevOps in your organisation, from day-to-day changes to the overarching goal.

When speaking to managers, find out which driver for DevOps is most important to them: speed to market, product quality, or stability of the production environment. Knowing their motivation can help with prioritising the new development practices that are explained in the following chapters.

Getting to know people with different roles is also an opportunity to determine their appetite for doing things differently. Those who are open and enthusiastic are likely to be positive allies for change.

Collaboration beyond development

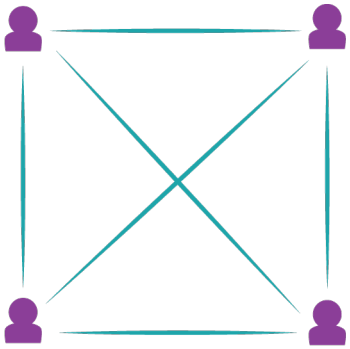
Agile development teams are usually relatively small. The Scrum Guide historically cited an ideal team size as 7 plus or minus 2, which has become a prevalent phrasing across the industry. The latest version of the guide[17] extends this range slightly to 6 plus or minus 3.

The two pizza rule is another variant on the same theme:

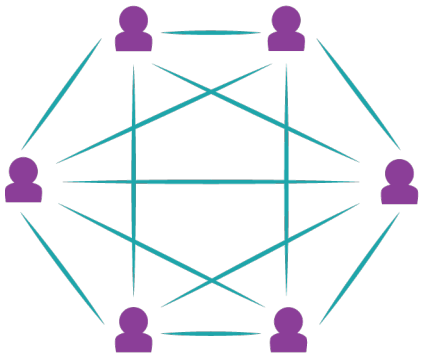
> "The two pizza rule is a guideline for limiting the number of attendees at a meeting. According to the rule, a meeting should never have so many attendees that they could not all be fed with two pizzas. Generally, this limits the number of attendees at a meeting to less than eight. The two pizza rule is often credited to Jeff Bezos, founder and CEO of Amazon."
>
> Two Pizza Rule[18]
>
> *Ivy Wigmore*

The primary reason to have a small team is to make it simple to collaborate with each other. Adding an extra person has a significant impact on the number of communication paths used by the team. Steve McConnell explained the effect of team size on communication paths in Less is More: Jumpstarting Productivity with Small Teams[19] using an illustration similar to this one:

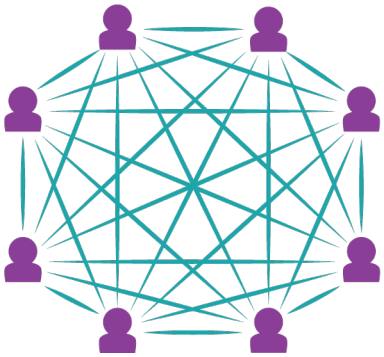
[17] http://www.scrumguides.org/docs/scrumguide/v1/scrum-guide-us.pdf
[18] http://whatis.techtarget.com/definition/two-pizza-rule
[19] http://www.stevemcconnell.com/articles/art06.htm



4 people
6 communication paths



6 people
15 communication paths



8 people
28 communication paths

The culture change required for DevOps could mean an additional operations specialist on your team. More often though, collaboration with people outside the development team will be expected.

The first step in creating collaboration beyond development is to trailblaze the connections to people beyond your team. The following sections explain what it means to blaze a trail, offer specific examples of how you might create communication paths for testing, explain why you would want to foster different connections, and share some thoughts on resistance that you might encounter in

building relationships across your organisation.

Blazing a trail

When I'm blazing a trail I'm building a path to someone new in the organisation. Building a communication path is not just about talking to people, though that is a good start. A path evolves by adopting progressively collaborative activities.

Identify a person you would like to build a relationship with. Find out their name and where they work in the organisation. Then initiate a connection to that person using your normal channels. Contact them online or schedule a face-to-face meeting to introduce yourself. Explain your role and why you believe a path to their team could be useful.

Start sharing information through informal conversation to build a one-to-one path.

A word of caution. If you've initiated the connection it can be tempting to flood it with your material. Remember that the purpose of these trails is not simply to promote the ideas and agenda of your development team. Collaboration goes both ways. As you begin telling someone what you're doing, make sure to ask questions about what they're doing too. Invite them to sessions in your team focused on delivery of software, then request to attend similar sessions in their team focused on its support.

Initially the paths that we create are fragile as they are formed between two individuals. If someone at either end were to leave, the path would be completely severed. The next steps in trailblazing mature the connection by involving others, working towards a collaborative exchange between disciplines.

Invite more people to use your path by extending the audience for your exchange of information. Once you've built a relationship with one person in a team and shared something of use to them, run a session that shares this same information to their colleagues. When they've shared something of use to you, ask them to run a session for your colleagues.

Put markers in your path so that people can find their own way. This helps people who are current team members, those who will join the team at a later date, and interested parties from other teams. Record the knowledge being shared so that it can be accessed later on. It may be as simple as saving Powerpoint presentations to a shared folder, or you could video knowledge sharing sessions to capture both the visual and verbal elements. Those with more time may be able to create learning resources that are specifically designed for on-demand consumption e.g. a learning pathway, an online training course, or a book.

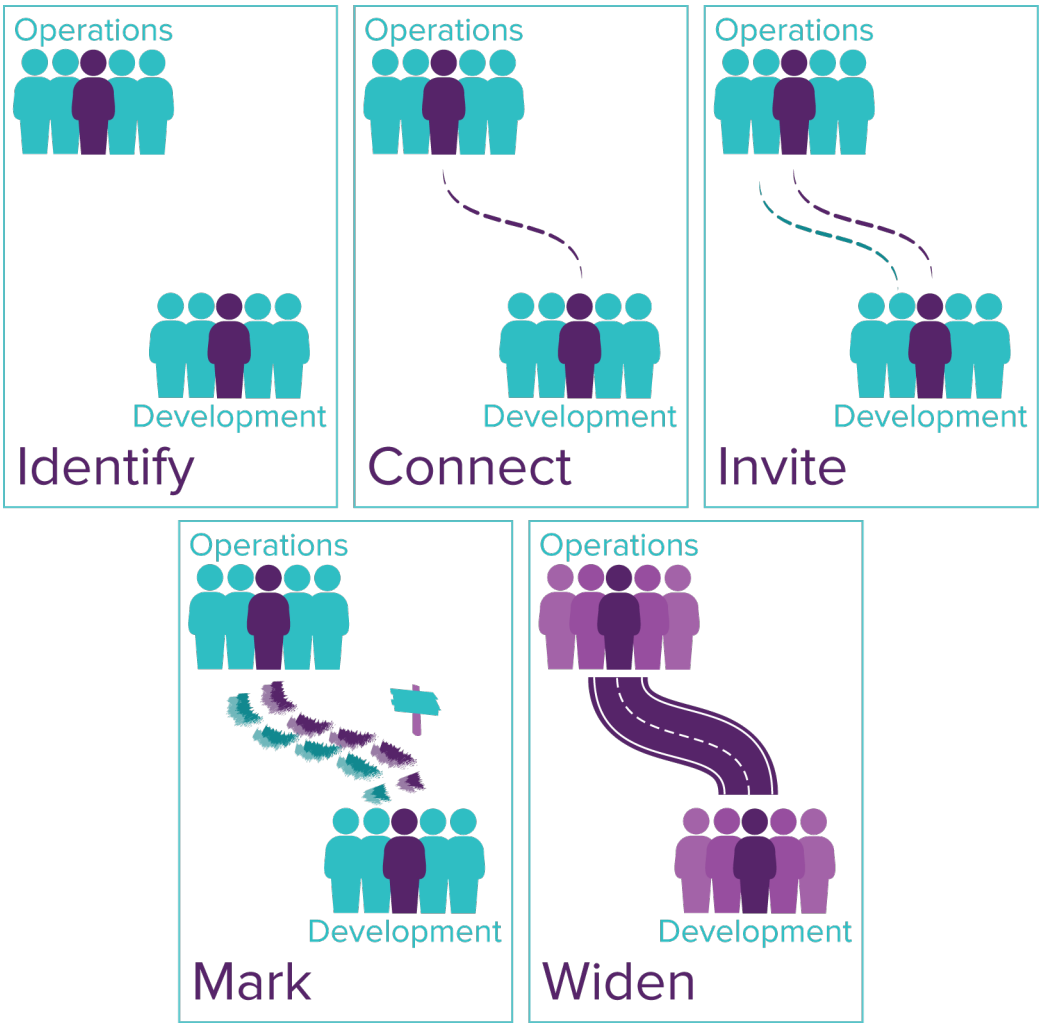
Widen the path by inviting different perspectives to contribute to it. Within your organisation this might include:

- inviting someone else to present their knowledge,
- involving those from other disciplines in peer review and debrief activities,

- cross-discipline pairing or inter-team collaboration on tasks through mobbing.

You could also widen the trail further by inviting people to attend industry presentations or conferences in areas outside of their usual discipline.

The steps to blaze a trail can be summarised as identify, connect, invite, mark and widen, as shown in the illustration below:



Paths for testing

How do you take the theory of communication paths and apply it to build cross-discipline inter-team relationships of your own that aid testing?

Identify

Start by thinking of people you know who work in operations, support, business intelligence, analytics, the call centre, etc. Anyone who works outside of your development team but has some connection with what you deliver. This independent brainstorming activity can be a good indicator of your existing network. If you struggle to think of names then you have a lot of work to do, or a very poor memory!

Ask your development team and your manager to help you extend your list. In a large organisation, you could also spend some time browsing the corporate directory or structure diagrams to identify people with job titles of potential interest. Try to cast the net widely. There'll be a natural filter as you start talking to people and determine which paths offer the most opportunity.

Connect

Once you have identified a group of people, try to connect with each person individually.

If you're lucky, your list will include people that you already have a good connection with. Perhaps they are a former colleague, a friend, or someone you interact with outside of work through shared activities. If this is the case you may be able to skip or minimise this step in building your working relationship with them.

If you have not had any direct interaction with person that you identify, it may be enough to start by asking them a few general questions by email. What do you do day-to-day? What sort of tools do you use? Who do you interact with most often? What do you know about my role? How do you think that I could help you?

It can feel strange to try to build a relationship from nothing. Instead, you might look for a catalyst to create a connection. Ioana Serban, a Test Engineer, spoke about her experience in building a relationship with an Operations Engineer by asking problem-solving questions to diagnose a deployment failure:

> How does the deployment process work? Where is the monitoring? What are the alerts? How do I get to the logs? What went wrong and how did you fix it?
>
> Taking control of your test environments[20]
>
> *Ioana Serban*

Questions can also be the basis of a connection. In another presentation, Ioana talks about the "Three Little Questions" that she uses for building individual relationships between testing and operations:

1. What are you working on?
2. Want to see something cool?

[20] https://www.youtube.com/watch?v=ufZ4tDSgvv8

3. Can you pair with me on this?

TestOps: Chasing the White Whale[21]

Ioana Serban

Asking questions is a proactive way to go about connecting with people across your organisation. It can also be an intimidating prospect. As an alternative, or in addition to, you can create opportunities for people from outside your team to connect with you.

One means of doing this is being conscious of describing your activities with sufficient detail, so that others can understand how they might offer their assistance to help you complete a task. One of my frustrations in agile environments is the laziness that can emerge in stand-up meetings. People give updates that create an illusion of information being exchanged when, in fact, there is very little transparency about what people are really doing.

Imagine that you're a tester in an agile stand-up meeting. You have all gathered around your visual management board, each person is sharing an update and it's your turn to speak.

"Yesterday I tested story 19574 and didn't find any bugs." you say, moving that story to Done.

"Today I'm just going to test story 19572." you continue, moving that story to Doing.

"There's nothing blocking me at the moment, so that's it." you finish.

Sound familiar?

Now consider whether, based on the information you've just provided, anyone listening actually understands what you've done and what you're planning to do.

If you were to ask them, they might say that you're testing. You are the tester. You just told them you were testing. They're not stupid.

But what exactly does that mean? Have you been confirming acceptance criteria are met? Have you been investigating how this story handles error scenarios? Have you tested from the perspective of each of the business user personas? Have you tried some simple penetration tests?

We can make our testing more visible by being transparent in our stand-ups. Start making a conscious effort to be more specific. Take the opportunity in your stand-up to verbally create an image in someone's mind of the activities that you're undertaking. Without this detail there's a good chance that what you do is a little bit of a mystery. And if they don't really understand what you're doing, then it may be difficult for them to identify opportunities to collaborate with you.

If the person that you are trying to connect with does not attend your stand-up meeting, it is still useful to get into the habit of describing your day-to-day work. Regularly practicing how to explain what you are doing can make it much easier to repeat this information to a person beyond your development team.

[21] https://www.youtube.com/watch?v=I6A07ESTc_k

In a similar vein, visualising information and putting it on display can encourage people to connect. Records that mark shared understanding within a particular group can invite interaction from those outside the group who become curious about what is on display.

Maaret Pyhäjärvi writes about how she has altered her behaviour in meetings from capturing a written record to drawing visual images, and the positive impact this has had:

> "The transformation of how I am in meetings has been quite significant. From the person on the computer making detailed notes (and often ending up as the meeting secretary), I've moved my role to be the person on the whiteboard, making conceptual notes. And where in my written detailed notes I never got feedback if we understood things the same way, my notes on the whiteboard invite corrections, additions and depth. The visual nature of the boxes and arrows helps us discuss things, remember things and go deeper into the things.
>
> My phone is full of pictures that remind me of discussions in the past, and they work better than any of the detailed notes I used to have."
>
> Step up to the whiteboard[22]
>
> *Maaret Pyhäjärvi*

As Maaret describes, the process of creating a visual diagram together can forge connections between people. Rather than keeping these artifacts in a personal phone, Christina Ohanian is an advocate for establishing a "creative wall" to display them, which further extends the opportunities for people to engage:

> "Pin up your thoughts. I truly believe in creative walls. For me, it's a great conversation starter, it's a great way to inspire thinking … what you'll realise is that people start walking past and going "Oh, I didn't think about that"."
>
> Unleash your creativity and become a better tester[23]
>
> *Christina Ohanian*

There are a number of options for displaying test-specific information as a lure for conversation with people from outside the development team. At a granular level, you can visualise your test ideas using timelines, buckets, or Venn diagrams. At a higher level, you can visualise coverage using a visual test model or flows through the system using state transition diagrams. You can also evolve your visualisations over time and connect with people as they comment on how it changes.

For more information on this topic, I have published a number of illustrated blog posts that give practical examples of applying visualisation when testing a product:

[22] http://visible-quality.blogspot.co.nz/2017/01/step-up-to-whiteboard.html
[23] https://www.youtube.com/watch?v=m-noJQvR4gk

- Visual Test Ideas[24]
- Visual Test Models & State Transition Diagrams[25]
- Evolution of a Model[26]
- Three examples of context-driven testing using visual test coverage modelling[27]

If your development team and operations team don't always get the opportunity to work closely together day-to-day, a shared project can create an environment for connections. One example from my own experience was a project that I led to improve our test automation by implementing Selenium Grid.

This piece of work required changes to the test automation code, which the testers completed. It also required changes to our test environments. Instead of a dedicated Jenkins slave to execute our tests through a local browser, we built a cloud-based Selenium Grid that would allow us to execute tests in parallel across multiple remote browsers. The operations team were pulled into the project as they could provision and configure the new environments we needed.

The project had a relatively tight timeframe, as our existing solution was not meeting the growing demands of our development teams. Leveraging the skills of the operations team meant that the we could complete the work quickly. Having a reason to collaborate across disciplines was a catalyst for establishing new relationships across the organisation.

Invite

Invitation is extending a connection beyond a one-to-one interaction. Invitation brings in people from the same discipline at each end of the path. For example, if a tester and an operations engineer connect, the next step is to involve other testers and operations engineers in the same exchange of information.

A consistent theme in many presentations about building cross-discipline relationships is caring. Showing a genuine interest and appreciation for the work of others. In my experience, caring is key to organically engaging a group of people.

Carol Brands gave an example of invitation by creating a successful relationship with customer support during a talk at CAST 2016:

> "I asked [customer] support for their help. Not just their help in understanding the program but also their opinion. They know our customers really well and their opinions really matter to me. I make sure to show them that [their opinions] matter by asking for [them] regularly."

[24] http://katrinatester.blogspot.co.nz/2014/11/visual-test-ideas.html
[25] http://katrinatester.blogspot.co.nz/2015/01/visual-test-models-state-transition.html
[26] http://katrinatester.blogspot.co.nz/2014/04/evolution-of-model.html
[27] http://katrinatester.blogspot.co.nz/2014/10/three-examples-of-context-driven.html

> Babble & Dabble: Creating bonds across disciplines[28]
>
> *Carol Brands*

Ioana Serban also talks directly about the activities that make up caring: visiting people in their own space, offering small tokens of gratitude, and being interested in their lives. She clearly sees a link between these actions and her ability to collaborate successfully:

> "I used to spend time in the Operations Engineers room. I used to feed them candy. I'd ask them how their day went. I'd ask them to show me cool stuff."
>
> Taking control of your test environments[29]
>
> *Ioana Serban*

Invitation may also happen formally. Asking people to attend a knowledge sharing presentation, or a Lean Coffee[30] discussion, or a series of lightning talks[31]. The purpose of these sessions is to deliver material that has emerged through connection to a wider audience.

Kate Falanga, in her role as a Director of Quality Assurance, speaks about formal knowledge sharing between teams to create connections between disciplines. She emphasises that these sessions are two-way:

> "I've had people from different kinds of groups in our organisation speak at our team meetings. I want to know what [they] do and I want to know how [they] want to interact with us. Having that shared roadshow of speaking at different team meetings is very helpful."
>
> Testing is your brand: Sell it![32]
>
> *Kate Falanga*

Mark

Marking the path is creating a record of conversations that preserve the information exchanged for future use. Generally, markers will be specific to your organisation and the teams that are at each end of the path.

You can mark a path with any repeatable recorded communication: written, verbal or visual.

Most organisations capture in traditional written medium: slide sets, documents and wiki-based notes. I created a general resource for building relationships from testing to other teams, the Testing

[28] https://www.youtube.com/watch?v=RgcNgabDN_c
[29] https://www.youtube.com/watch?v=ufZ4tDSgvv8
[30] http://leancoffee.org/
[31] http://katrinatester.blogspot.co.nz/2016/04/lightning-talks-for-knowledge-sharing.html
[32] https://www.youtube.com/watch?v=TEln1YTrjrg

for Non-Testers Pathway[33], which has practical information grouped by questions that people might have when learning about testing:

1. Why non-testers should be involved in testing
2. How to make an application testable
3. How to decide what to test
4. How to document your test ideas
5. What to think about while testing
6. How to debrief
7. Where to automate

Kate Falanga has spoken about how her organisation are Teaching Testing to Non-Testers[34] through a workshop. The marker is delivered verbally, but via a standard set of materials and exercises that support a consistent conversation.

Perhaps the most common visual marker between teams in DevOps is an automated deployment pipeline. A pipeline is a repeatable, recorded communication of automated feedback. It includes test and deployment scripts, from development and operations respectively, along with a pipeline to illustrate the process by which the scripts run. It provides structure for a lot of information exchange.

Widen

Widen the path through activities that engage multiple contributors at each end. This phase of trailblazing is about inviting different perspectives to help create a rounded view.

One approach is to adopt practices that encourage different individual connections between disciplines. In the analogy of the path, this would be a new person at each end of the trail collaborating together. You might achieve this through peer review, debrief, pairing, or rotation.

Peer review & debrief

Peer review and debrief are similar activities, but they are different. In this context I consider peer review to be something that happens prior to an activity taking place, and debrief to be something that happens after. In a testing context, this means that you might ask someone to peer review your test ideas or plan, then to help you debrief once you have test results. Both activities are common in agile development teams. Moving to a DevOps culture might involve a wider range of people in peer review and debrief.

Asking a person from the operations team, the analytics team, or a support channel to peer review or debrief testing will give a fresh perspective. They have deep domain knowledge, but from an

[33] http://katrinatester.blogspot.com.au/2015/11/testing-for-non-testers-pathway.html
[34] https://www.youtube.com/watch?v=KUiBGtHWdeU

entirely different angle to the development team. Their focus will be different. The opportunity to involve these people earlier in testing can improve the way that the software operates in production.

The reverse also applies. Ask those in the operations team, analytics team, or support channels whether they have work that they would like to discuss with the development team in a peer review or debrief context. Bring that perspective into operations.

Peer review and debrief activities don't need to be onerous. I suggest setting an expectation with the wider team that these are short, time-boxed activities, with the group determining a duration that they feel is appropriate. Whoever is initiating the exercise would spend the first half of that period sharing what they plan to do, or what they have done. The second half of that period should be an open discussion about areas that may have been missed or received insufficient attention, along with general questions.

A development team won't have *everything* peer reviewed or debriefed with an operations audience, nor vice versa. These are useful practices to apply to work that you think may suit it, and occasionally to work where you don't expect feedback simply to test that your assumptions are accurate.

Pairing

Pairing is a practice where two people work together on the same thing at the same time and place, continuously exchanging ideas[35]. When paired, two people use a single machine or device. One has the keyboard, though it may pass back and forth in a session, while the other suggests ideas, pays attention and takes notes, listens, asks questions, grabs reference material, etc.[36]

The pair should tackle a single task, so that they have a shared and specific goal in mind. Though the pair will work together, one person must own the responsibility for getting the task done. The person with ownership of the task may do some preparation, but pairing may be more successful if there is flexibility. During a pairing session the participants should talk constantly to create shared understanding of what they're doing and, more importantly, why they are doing it.

Pairing is high creativity, as it forces people to explain their thinking and react to the thoughts of their partner. It brings people together to learn and resolve problems. It's an embodiment of the old proverb "two heads are better than one".

Pairing is high productivity, as it forces people to maintain their focus on a task. When people are working together, it makes their colleagues less willing to interrupt them. Pairing encourages dogged pursuit of insights.

Pairing is common within agile development teams but may be utilised less beyond the team boundary. Perhaps this is a missed opportunity to bring a different perspective to the development process. Imagine working on a story that changes a workflow within the product, and pairing with a person from the support team to test the logging output for the new code. Imagine working on a story that changes the load on a web service through the introduction of pagination in the user

[35] http://www.exampler.com/testing-com/test-patterns/patterns/XT-Pattern-jb.pdf

[36] http://www.testingeducation.org/a/pairs.pdf

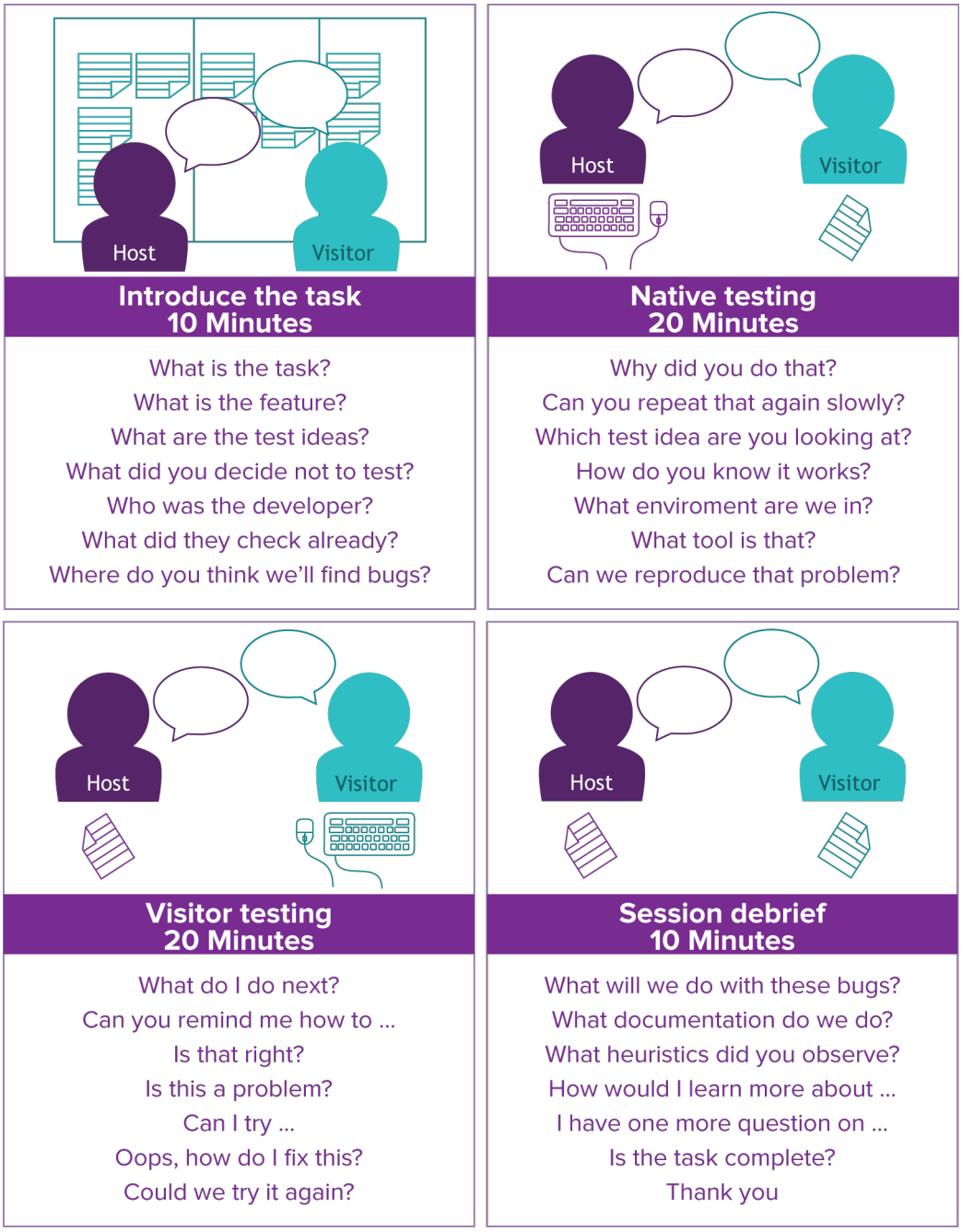
interface, and pairing with a person from the operations team to test the performance of the new code. A DevOps culture can make this reality.

If pairing is new to one or both participants, having an agreed structure to a session may be useful while learning. It can be easy for a pairing session to turn into a demonstration, where one person holds continued ownership of the keyboard and direction of the session, which can be frustrating for the other participant. A structure that I have used successfully with inexperienced pairs is a one hour session broken into the following broad sections:

- 10 minutes – Discuss the context, the story and the task for the session.
- 20 minutes – Host driving, visitor suggesting ideas, asking questions and taking notes.
- 20 minutes – Visitor testing, host providing support, asking questions and taking notes.
- 10 minutes – Debrief to analyse and document outcomes of the session.

This sample structure formed the basis of a pairing experiment for sharing knowledge between agile teams[37] that I facilitated. To support the participants and emphasise the importance of communication, I produced and distributed the following graphic that included potential questions to ask in each phase:

[37] http://katrinatester.blogspot.co.nz/2015/06/a-pairing-experiment-for-sharing.html



People who are unfamiliar with pairing may be a little hesitant to try it. But if you've created and fostered a connection with someone, they'll probably be willing to spend an hour to give something new a try. If it works, you've found a new way to share ideas across disciplines. If it doesn't, the cost is limited.

Job rotation

A number of organisations widen the path between development teams and support by operating a rotation for those in development to work in support for a short period of time. This gives the development team firsthand perspective on how the product they develop is being used, the problems that people are encountering, and the ease with which their issues can be diagnosed then resolved.

The frequency and duration of these rotations vary.

It might be as simple as rotating ownership of first level support for one night each week so that a wider audience can experience the joy of being woken in the night to diagnose a problem, determine its urgency, then identify the appropriate people to fix it.

Automattic, the company who develop WordPress, run "All Hands Support". They rotate close to 300 non-support employees through their support team each year, where individual employees put their work aside for a week and help out in Support[38].

Similarly, Etsy run a support rotation program:

> "Since 2010, Etsy employees have participated in "support rotations" every quarter, where they spend about two hours replying to support requests from our members. Even our CEO participates. What started as a way to help our Member Operations team during their busiest time of year has evolved into a program that facilitates cross-team communication, builds company-wide empathy, and provides no shortage of user insights or fun!"
>
> We Invite Everyone at Etsy to Do an Engineering Rotation: Here's why[39]
>
> *Dan Miller*

Etsy have also implemented a reciprocal program that invites people from across their organisation to be part of their engineering team.

> "… any Etsy employee can sign up for an "engineering rotation" to get a crash course in how Etsy codes, and ultimately work with an engineer to write and deploy the code that adds their photo to our about page."
>
> We Invite Everyone at Etsy to Do an Engineering Rotation: Here's why[40]
>
> *Dan Miller*

The Etsy program is split into three parts: homework; an in-person class; then hands-on deployment. As with a development team member rotating into support, inviting other people into development builds empathy and cross-team collaboration. It's a hands-on way to widen the path between disciplines.

[38] https://jeremey.blog/all-hands-support-automattic/
[39] https://codeascraft.com/2014/12/22/engineering-rotation/
[40] https://codeascraft.com/2014/12/22/engineering-rotation/

Dojos

The practices above create opportunities for different individual connections between disciplines. Another approach is to adopt practices that encourage team collaboration. Instead of encouraging a variety of one-to-one connections, create a many-to-many collaborative environment. You might achieve this through use of coding dojos.

A dojo in the context of software refers to an environment where people can learn and practice their development skills together. The participants of a dojo may be from a single team, a single discipline, or a wider audience.

I have experienced coding dojos as part of a development team, where the developers and testers worked together to write new production code and accompanying automated tests for a story. I have facilitated a coding dojo for a group of testers from across different development teams, where the objective was to agree standards for implementing a new test automation framework. I have found dojos to be an excellent format for focused collaborative work.

To run a dojo, you bring everyone together for a set period of time: usually 90 - 120 minutes. There is a single computer that is connected to a projector, and a whiteboard for use during problem solving. The computer is operated by a pair with the remaining participants contributing verbally. The two people at the computer will rotate during the session, so that everyone present has the opportunity to drive.

There is also a sensei - a teacher or facilitator - who runs the dojo. Their role might include setting the topic of the dojo, timekeeping during the session, and asking questions when the group appears to have run off track. They should avoid providing answers; solutions should come from the participants themselves.

From my own experience in this facilitation role, the absence of a single authority can help the group to work together – although I find it quite challenging to keep quiet at times. It was really interesting to see the approach taken, which differed from how I thought the participants might tackle the problem. I also learned a lot more about the personalities and social dynamics within the team by watching the way they interacted.

The mechanics of a dojo are illustrated and explained clearly in this short video: How do you put on a coding dojo?[41].

As with the other practices in this section, a DevOps environment provides the opportunity to utilise this approach with a wider group. Instead of involving operations solely in the definition or review of the solution, why not co-create the solution with their active involvement?

This may not be a practice that is utilised day-to-day, but in situations where close collaboration would be beneficial it may be worth exploring. Is the development team working on a story that directly impacts the mechanics of collecting analytics events or altering the structure of the underlying database? Be mindful of opportunities where involving a wider group can provide essential fast feedback.

[41] https://www.youtube.com/watch?v=gav9fLVkZQc

When widening the audience, remember to foster contribution from those who might find the format intimidating:

> "Some other considerations when running a dojo: there will likely be different skill levels involved. Be sure to encourage people who want contribute – designers, PMs, or anyone in between. It's scary to put yourself out there in front of people whose opinions you respect; a little bit of reassurance will go a long way towards making people comfortable. Negative feedback and cutting down should be actively discouraged as they do nothing in a dojo but make people feel alienated, unwelcome and stupid."
>
> The Art of the Dojo[42]
>
> *Bucky Schwarz*

Industry connections

Another way to widen a path between disciplines within your organisation may be to invite the perspectives of people from outside your organisation. You can do this by sending your people out to industry presentations or conferences in areas outside their usual discipline, or by inviting people into your organisation as guest speakers.

If you are connected within your profession, locally or internationally, then you are likely to be aware of organisations who are successfully implementing some aspect of DevOps. The people who work in these organisations are often happy to share their experiences, both the successes and the learnings. There is little to lose in politely approaching someone to ask if they would be willing to present to you or your team.

I have experience in arranging internal conferences, within my organisation, that utilised external speakers. In our first iteration I asked people from around my country. In the second iteration I was able to make arrangements for international speakers to participate. Hosting a conference of our own meant that all our people to had the same learning experience, so conversations carried from the event back into our day-to-day work.

[42] https://codeascraft.com/2015/02/17/the-art-of-the-dojo/

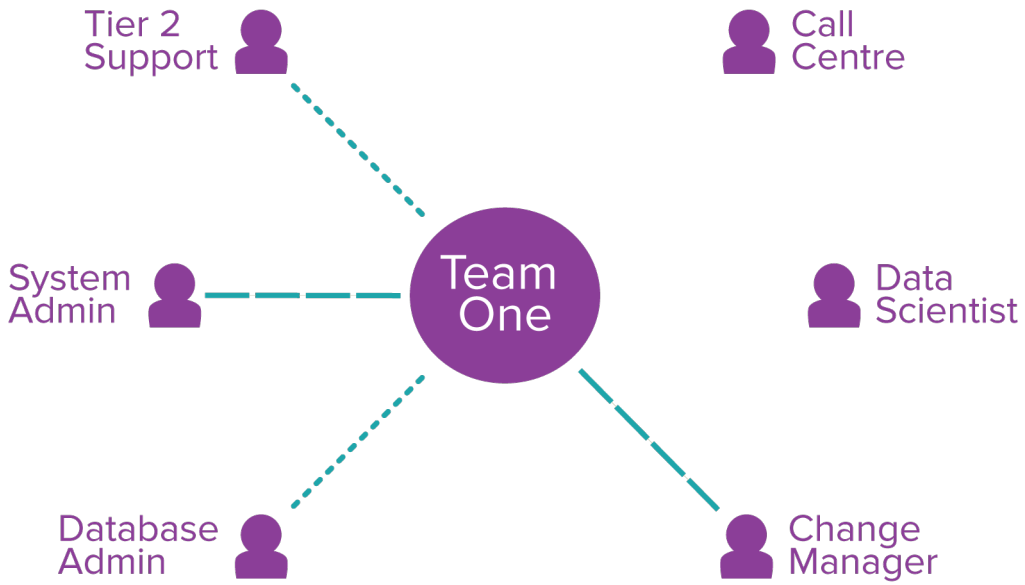
Choose your map

As with any software development, the paths that you cultivate through a DevOps culture will have a direct influence on your software. This phenomenon is known as Conway's Law:

> "Any organization that designs a system … will produce a design whose structure is a copy of the organization's communication structure."
>
> Conway's Law[43]
>
> *Mel Conway*

To illustrate this, imagine two teams operating in a DevOps environment who have built connections beyond their development team of varied collaborative strength.

Team One have built strong connections with a System Administrator in the Operations team and a Change Manager in the Communications team. They have some relationship with a Database Administrator in the Database team and Tier 2 Support in the Operations team who are the escalation point for call centre problems. They have no relationship with their call centre directly or with anyone in the analytics team.



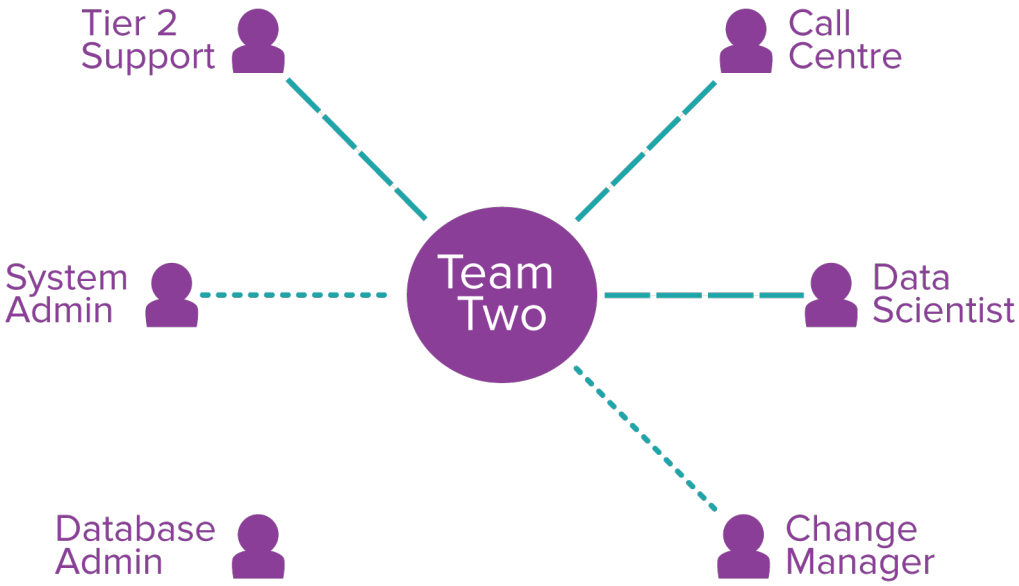
This ecosystem means that Team One are really good at delivering software quickly that runs efficiently. They have worked with their System Administrator to create reliable test environments and they have a strong understanding of how their software impacts their hardware. They are proud that they have never caused a hardware outage through unexpected changes to memory, CPU or disk use. They work alongside their Change Manager to create accurate and thorough user

[43] http://www.melconway.com/Home/Conways_Law.html

communication when they deliver new functionality, which means that their customers have a good understanding of changes that are made.

Though the software is built the right way and users understand changes being made, this team are not always building the right thing. The lack of relationship with the call centre and analytics, along with a limited relationship to Tier 2 Support who run production monitoring, means that Team One don't track how their changes have been adopted. There have been a few cases where they've released a feature built well that the customers did not need, and chose to avoid.

Contrast this to Team Two who have built strong connections with a Data Scientist in their analytics team, a Customer Service Representative in their Call Centre, and Tier 2 Support. They have some relationship with their System Administrator and Change Manager, but so little as to be none with their Database Administrator.



This ecosystem means that Team Two are really good at building the right thing. They collaborate closely with the people who have deep understanding of customer behaviour in their software to understand how call centre volumes or page views should shift as a result of their development work, create testing personas to explore the software as they create it, then track the outcome post-release to confirm that they have achieved their goal.

Yet they lack in the area where Team One were strong and sometimes fail to build the right way. Their system administrator and database administrator are prone to frustration when changes from Team Two introduce a memory leak or create referential integrity problems.

Both Team One and Team Two have adopted DevOps with very different outcomes.

Though these fictitious teams portray situations with intentional hyperbole, there are shades of difference in every DevOps environment. The outcome you will achieve is so dependent on the specific communication ecosystem that your team cultivates. With an understanding of what DevOps means in your organisation, you can work out which connections are most important for

you.

Resistance

You might have a goal in mind when you start to build relationships that support DevOps in your organisation. The reality may look quite different, because each path starts and ends with people.

You may remember a news story that published photos of an uncontacted tribe from the Peruvian-Brazilian border. The images showed men "[s]kin painted bright red, heads partially shaved, arrows drawn back in the longbows and aimed square at the aircraft buzzing overhead." The Daily Mail[44] The response of this tribe to intrusion was aggression. They didn't want to connect with the outside world.

Resistance in a professional context is unlikely to be quite this obvious. But it happens and there are a lot of reasons that can cause someone to react in a negative way.

What might you do when you encounter resistance?

Think about the positive relationships you have established. There might be someone in your existing network who can approach the connection you are trying to build from a different perspective. They might be able to have a discussion on your behalf or facilitate your initial interactions. Think about how you can use the people around you to understand and resolve the resistance that you see.

Think about what you're asking of the person who is blocking your path or reacting negatively to your approach. Collaboration is a continuum. If you haven't built the foundation of a path and you attempt to jump straight into an intensely collaborative practice that may cause a negative response. As an example, if you started with an attempt to run a dojo or pairing activities, you might take a step back to encourage knowledge sharing first. Give people the opportunity to adjust, learn and grow.

Think about your workload and the number of activities that you're juggling. A negative response may be a result of a person feeling undervalued. Are you trying to do too many things at once, to the detriment of doing them well? Is the resistance a reflection of their perception that you don't care about them? You need to give an equal amount of time and energy to your end of the relationship.

These are some common reasons that people might resist, but this isn't an exhaustive list. If you haven't found the answer, spend some time reflecting on why you might be encountering resistance. If you can't see any reason, ask a few trusted colleagues for their perspectives.

Beyond interpersonal relationships, you may encounter resistance because of organisational processes and tools that fail to support a new way of working. In my experience these are easier to workaround than the resistance that comes from people, as there is generally a methodical approach to resolution. Collaboratively identifying a new process or replacement tool is a lot less threatening than discussing the behaviour of an individual.

[44] http://www.dailymail.co.uk/sciencetech/article-1022822/Incredible-pictures-Earths-uncontacted-tribes-firing-bows-arrows.html

If you are unsuccessful, resistance may end up altering the shape of your map. If your options are an easy path through an open field or a difficult path with a lot of resistance through a dense jungle, the easier journey is more appealing. Taking the easy path may mean that you build unexpected connections. Working around resistance will result in a slightly different outcome, but an outcome nonetheless.

The other point to remember when encountering resistance is that the paths you build will always be in flux. People change. Roles change. The way that we work together is constantly evolving. Some paths will become overgrown with time, while others will emerge. Resistance may shape our way today, but tomorrow there could be a different path.

TESTING IN DEVELOPMENT

DevOps affects both the development and operations team. Building relationships will create opportunities for learning in both areas.

When the development team involves the operations team earlier they have more opportunity to address operational risks. Common operational risks include software that won't deploy correctly, won't run efficiently, or that users don't find valuable. These are risks that the development team otherwise address in isolation. An inclusive approach could change the solution that the development team choose.

Feedback in development

This section describes the feedback that your development team uses for making decisions about the quality of your product, and explains how your development practices might change in a DevOps environment.

Automation

Unit tests, integration tests, and end-to-end tests are all examples of the usual automated tests that a development team might create. They focus on confirming behaviour of the software at different levels of abstraction - from individual components through to the entire solution.

In some development teams "test automation" is a term that refers only to checking functionality. However test automation can also include non-functional behaviour like accessibility, security, and performance. For example, you could check that images in the product have an alternative text for screen readers, that text fields escape scripted input appropriately, or that page load time remains consistent. Non-functional automated tests have similar benefits and limitations to functional ones. They give information about one very specific aspect of the system but cannot provide deep or thoughtful insight.

Machine learning, an emerging trend in both product development and test automation, is starting to improve the information automation can give you. It offers a more robust approach as it is driven by a goal rather than specific scripted steps towards a known outcome. Imagine that we want to check whether a user can be added to our system correctly. A machine learning solution could learn how to achieve that outcome and adapt if the implementation changed. A test script with specific steps would fail when the implementation changed.

Beyond the product, automation can be extended to testing the platform. A deployment pipeline combines test automation with the deployment process. This includes automated scripts that are specific to deploying the product to an environment and verifying that the deploy has completed as expected. A later section of this book shares practices around testing infrastructure as code as a separate concern to product development.

Exploration

> "We always find the most serious bugs when we go off the script"
>
> Explore It! Reduce risk and increase confidence with exploratory testing[45]
>
> *Elisabeth Hendrickson*

[45] https://pragprog.com/book/ehxta/explore-it

Exploration is the counterpart to automation. Where automated feedback comes from tools, the feedback from exploration comes from people. What type of information can these people provide, and how can we collect it?

Testing that happens during development is best spent exploring: searching for the unknowns of the system, adopting different personas to offer varied subjective feedback, attempting to subvert security measures through malicious activity, or exploring the performance boundaries of the application. Focus on areas where the human brain is required to think: generate ideas, process results, and form opinions.

The results of exploration during development are most likely to be captured by conversation. In an environment where the development team collaborate closely, problems and questions can be raised directly. An interesting discovery can lead to an interesting discussion.

Test practices in development

Changing to a DevOps approach doesn't change the methods you use to gather feedback - there is still automation and exploration. However DevOps expands the scope of each by pushing testing into new areas and speeding up feedback cycles.

A closer relationship with the operations team will build their skills and concerns into the deployment pipeline. The final step of the pipeline might move closer to deployment, or "shift right", to include execution of test automation on the production environment. Instead of deploying code to dedicated test environments, the pipeline may change to build new environments on-demand.

In order to support the practices of testing in production - A/B testing, beta testing, and monitoring as testing - the development team will change their approach. For example, they might start to use feature toggles. These changes in approach will alter the testing effort in both development and operations.

The number of people who perform testing may grow, from the development team to a wider audience. This could happen through a bug bash or an internal beta release. Early feedback might also be sought from people outside the organisation who are potential or existing users of the product, and crowdsourced testing can also reach a wide audience for general feedback. A user-experience session could collect deep feedback from an individual who represents a particular user group.

This section explains how DevOps can create change in the test practices of the development team through the deployment pipeline, feature toggles, bug bashes, and crowd testing.

Deployment pipeline

Automated feedback can be rolled into the deployment pipeline that takes code from commit to production. The pipeline will then include all of your automated checking, along with build and deployment scripts that move the code through different environments.

> "At an abstract level, a deployment pipeline is an automated manifestation of your process for getting software from version control into the hands of your users."
>
> Continuous Delivery: Anatomy of the Deployment Pipeline[46]
>
> *Jez Humble & David Farley*

[46] http://www.informit.com/articles/article.aspx?p=1621865&seqNum=2

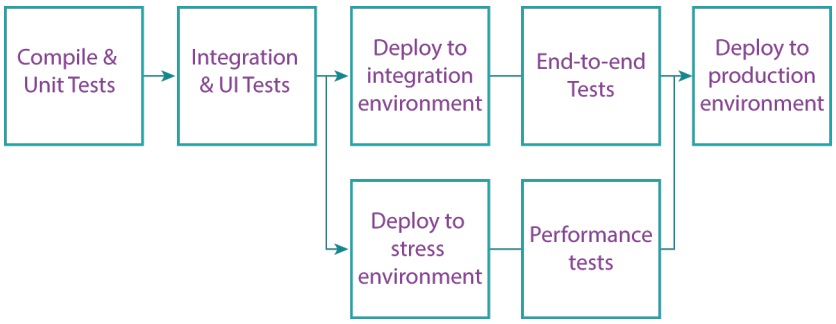
An pipeline is a communication path marked by a tool. The development team can use the pipeline to check behaviour of their software, while the operations team can use the pipeline to check the behaviour of their deployment. Combining automation into a single tool means that each team has visibility and understanding of the other team's activities.

What goes into a pipeline?

Traditionally a pipeline begins when a development team member commits a change to the application source code, and ends with a deploy to production. Between these two points the pipeline may include:

- static analysis of code quality
- building the source code into a product that can be deployed
- unit, integration, and end-to-end test automation
- functional and non-functional test automation
- deployment scripts for different environments

An example pipeline is illustrated below:



Deployment pipelines are well-known and have been part of the agile testing approach for many years. The abstract from Jez Humble and David Farley's book Continuous Delivery: Anatomy of the Deployment Pipeline[47] is a useful starting point if the concept is new to you.

There is a trend towards automated deployment pipelines where the process between code commit and production deploy is entirely automated. This approach will squeeze exploration to either end of the pipeline. The development team may explore the application, for example by holding a bug bash or by engaging a wider group through crowdsourcing. Once the software is in production, your organisation may adopt A/B testing or beta testing.

There are also development teams who add an exploration step to their pipeline. The pipeline might run static analysis and unit tests on the code, create a build that is deployed to a test environment, run functional test automation against that deployment, and then stop. At that point a development team member has the opportunity to explore the product in that test environment.

[47] http://www.informit.com/articles/article.aspx?p=1621865&seqNum=2

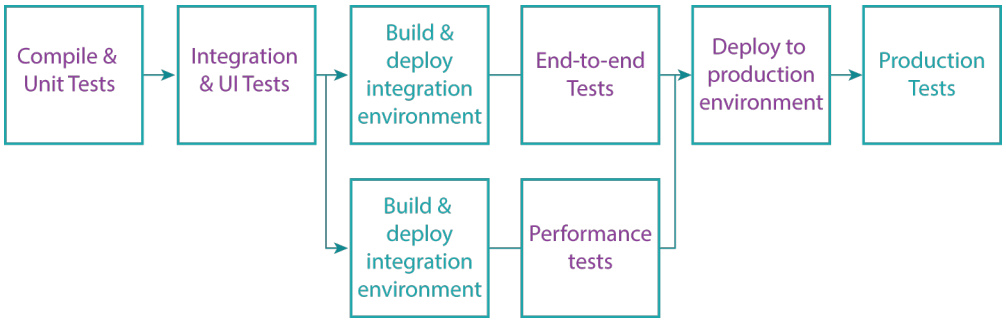
What goes into a pipeline will depend on the overall development approach of your team. The pipeline is a way to visualise what you are doing, which can help you think about how you work and identify areas for improvement.

In a DevOps-oriented organisation your pipeline may start to evolve in two key areas: infrastructure as code, and test automation in production.

Instead of deployment scripts that install code on dedicated test environments, the pipeline can include on-demand creation of infrastructure. As the operations team gain more experience in development practices, like programming and source control management, they can start to write infrastructure as code. Removing dependencies on specific environments can help the development team scale their pipeline to run faster and in parallel. These ideas are explained in more detail in the chapter on test environments.

The end of the pipeline may shift so that the final step is to run automation tests in the production environment. Extending the pipeline into production means that your development team can check the behaviour and performance of the application after each change that you apply. Running test automation in production also gives the operations team fast feedback about the implementation of monitoring, alerting, analytics and logging. These ideas are explained in more detail in the chapter on testing in production.

An example of how a pipeline might evolve is illustrated below:



How do you test a pipeline?

While the deployment pipeline will execute test automation, it also needs to be tested itself.

A pipeline will generally be created by developers who bring a builder's mindset to the task. They seek to create something that works, and will check that the steps of your pipeline can execute linearly and successfully. A testing mindset is then required to think about how your pipeline might fail.

A good starting point is to establish what might happen if your pipeline were to stop at any given step. Ask what state the product and platform will be in at each point. Determine whether the pipeline failing between steps would create an impact on people outside of your development team.

Then question the steps themselves. Ask what sequence of commands are being executed and what opportunities there are for these commands to fail. How could you recover or rollback if something

went wrong? If your pipeline includes rollback scripts, test they work as expected before you need to use them.

Testing the pipeline becomes more important when it includes steps that target the production environment. You will want to be very careful about what exactly is executed in production and how often. Think about the opportunities for failure and attempt to mitigate them.

Feature toggles

A feature toggle is a configuration option that defines whether or not a feature within your software should be executed. You might also hear this concept called feature flags, flippers, switches, feature bits, or latent code.

By wrapping new features in this type of configuration, code can be released into production without being available to all users. When the toggle is off, the code within that toggle is not being exercised. Feature toggles can also be used to hide code that is untested or incomplete.

Why would you want to put untested or incomplete code into production?

Imagine that your development team has been given a new feature to deliver. You estimate it might take a couple of weeks to complete, but your organisation is on a daily release cycle. Your team creates a feature branch to work on, which they merge back to the main repository at the end of the fortnight. Two weeks worth of code is then released into production at once. When running daily release cycles, this amount of code is relatively large and could increase risk.

An alternative is to make your new feature part of the daily release. Instead of writing code and only releasing it at the end of two weeks, you release incrementally each day. Your feature might still take two weeks to complete, but your code is disabled by a feature toggle. Because your code is constantly pushed to production each release is small and easier to manage.

Feature toggles reduce risk in making new features available to users. They'll be granted access to new functionality only once it has been present in production for a period of time. The development team and business users can test toggled code to discover problems in the live environment. If something does go wrong when the feature is enabled for users, it can simply be switched off again.

> "While everyone committing to trunk takes a couple of weeks to get used to, in the mid-to-long-term it represents a serious competitive advantage. Just ask Google[48], where 15,000 engineers commit to a single trunk and over 50 percent of the codebase changes every week. Facebook, Amazon, Netflix, Etsy, and IMVU all have similarly rapid commit cycles."
>
> Configuration Flags: A Love Story[49]
>
> *Noah Sussman*

Understanding implementation

Before you determine how to test a toggle, spend some time understanding how it has been implemented. Ask questions about the type of toggle, what values it can take, the number of times within the code that the toggle is queried, and the mechanism for changing its value.

Pete Hodgson identifies four types of toggles:

[48] https://twitter.com/hogfish/status/344567813345779712

[49] https://www.stickyminds.com/article/configuration-flags-love-story?page=0%2C1

- *Release toggles*: allow incomplete and un-tested codepaths to be shipped to production
- *Experiment toggles*: used to perform multivariate or A/B testing
- *Ops toggles*: used to control operational aspects of our system's behavior e.g. intentionally disable non-critical features in periods of heavy use
- *Permissioning toggles*: used to change the features or product experience that certain users receive e.g. activating premium features to reward customer loyalty

Feature Toggles[50]

Pete Hodgson

Feature toggles as described above are an example of release toggles. Each type of toggle will impact the system for a different audience. Pete also identifies differences in the longevity of the toggle (how long it is present in the software), and its dynamism (how often configuration is changed to switch the toggle on or off).

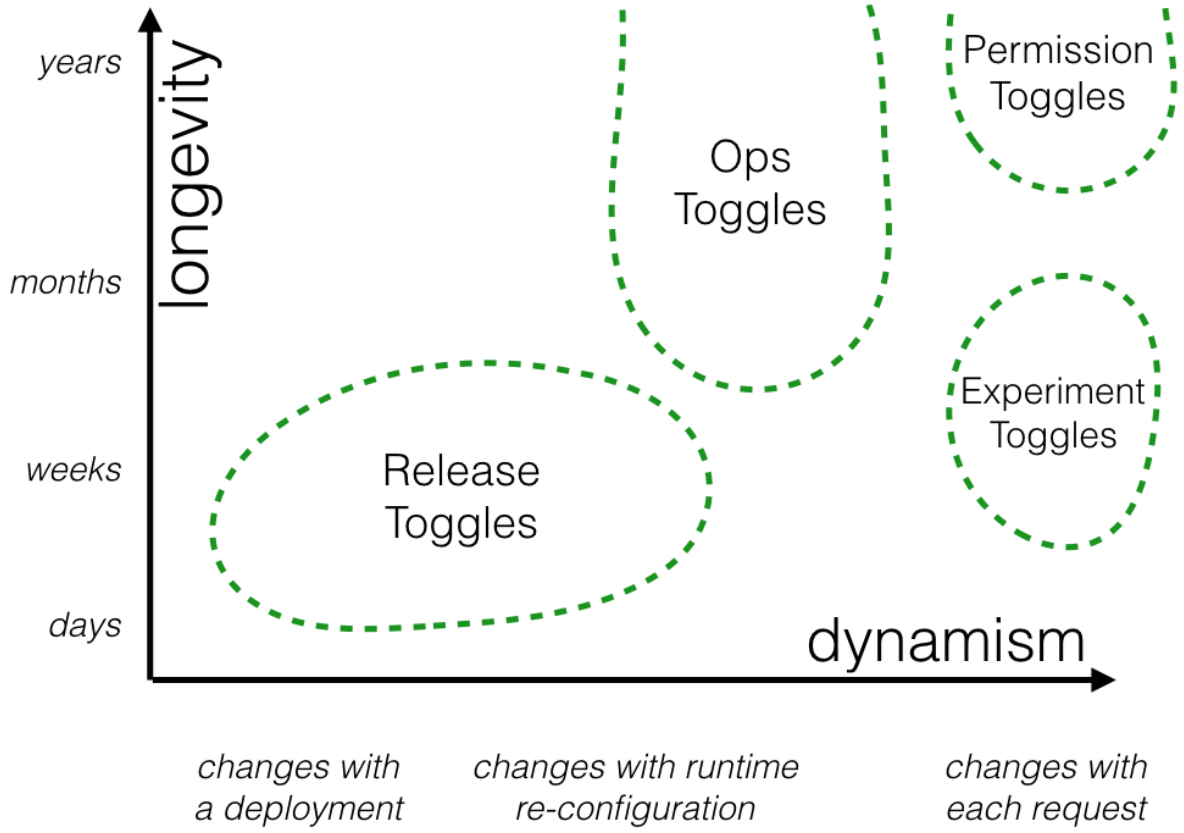


Image credit: Pete Hodgson, Feature Toggles[51]

[50] https://martinfowler.com/articles/feature-toggles.html
[51] https://martinfowler.com/articles/feature-toggles.html

A release toggle that exists for a short period of time, generally with a single change from off to on when the feature is ready for use, can have quite a different risk profile to a permissioning toggle that exists permanently and could change frequently. Understand who will use your toggle, and how.

The type of value that the toggle can hold can also alter the test strategy. A simple boolean toggle may require less testing than a list that can contain multiple values. Consider how your toggle can be set. Abhishek Tiwari lists a number of different toggle states:

> "Normally features toggles can have boolean states - ON or OFF. But depending on your requirements a feature toggle can be more than boolean states - lists, arrays, strings, conditions, logical statements, integers or any other value. State of toggles is either set statically using application configuration based toggles or dynamically using rules based toggles."
>
> Decoupling Deployment and Release - Feature Toggles[52]
>
> *Abhishek Tiwari*

Toggles should be implemented in the simplest way possible. Martin Fowler uses the phrase "toggle tests" to refer to the conditional logic by which toggles are coded, saying:

> "Toggle tests should only appear at the minimum amount of toggle points to ensure the new feature is properly hidden. [...] Don't try to protect every code path in the new feature code with a toggle, focus on just the entry points that would lead users there and toggle those entry points. If you find that creating, maintaining, or removing the toggles takes significant time, then that's a sign that you have too many toggle tests."
>
> Feature Toggle[53]
>
> *Martin Fowler*

Ask the developer if you can sit alongside them to walk through where their toggle tests are in the code. It should quickly be apparent whether they've tested at *entry* points or at *every* point. If the solution feels complex, the first task may be to simplify the implementation.

Finally, ask how a toggle is applied. Is the configuration read at build or runtime? How simple is it to switch between the different options? Will you have access and permissions to make the configuration changes in your own test environment? Will you be able to mimic the production process?

[52] https://abhishek-tiwari.com/post/decoupling-deployment-and-release-feature-toggles
[53] https://martinfowler.com/bliki/FeatureToggle.html

Testing a toggle

Most material about feature toggles is written from a development perspective, with a suggested approach to testing like Pete Hodgson describes below:

> "We can see that with a single toggle in play this introduces a requirement to double up on at least some of our testing. With multiple toggles in play we have a combinatoric explosion of possible toggle states. Validating behavior for each of these states would be a monumental task. This can lead to some healthy skepticism towards Feature Toggling from folks with a testing focus.
>
> Happily, the situation isn't as bad as some testers might initially imagine. While a feature-toggled release candidate does need testing with a few toggle configurations, it is not necessary to test *every* possible combination. Most feature toggles will not interact with each other, and most releases will not involve a change to the configuration of more than one feature toggle.
>
> So, which feature toggle configurations should a team test? It's most important to test the toggle configuration which you expect to become live in production, which means the current production toggle configuration plus any toggles which you intend to release flipped On. It's also wise to test the fall-back configuration where those toggles you intend to release are also flipped Off. To avoid any surprise regressions in a future release many teams also perform some tests with all toggles flipped On. Note that this advice only makes sense if you stick to a convention of toggle semantics where existing or legacy behavior is enabled when a feature is Off and new or future behavior is enabled when a feature is On."
>
> Feature Toggles[54]
>
> *Pete Hodgson*

This can be summarised as three sets of tests that use the current production configuration plus:

1. Any toggles you intend to release set to ON
2. Any toggles you intend to release set to OFF
3. All toggles set to ON

This is a solid starting point for static, boolean, and release toggles that are relatively independent to features created by other development teams.

If your situation doesn't fit that context, you may want to think a little bit more about your testing strategy:

[54] https://martinfowler.com/articles/feature-toggles.html

- If your toggle is set dynamically by a rule, you may want to test users who move into or out of the targeted group.
- If your toggle is for an experiment, you may want to test consistency across user sessions, to ensure individual users are given the correct version at every login.
- If your toggle is for operations or permissions purposes, you may want to explore toggle combinations to see how the new configuration interacts with existing options.

It can feel overwhelming to think of all the testing that is possible around feature toggles. Think independently, then share your ideas with your team. If you miss something, your team can help to fill in the gaps. If you come up with too much, they can help filter your list. It's generally better to collaborate, than to assume that the tests you have scoped will suffice.

If something fails you can always switch the toggle off. This safety net can be useful to remember when scoping test ideas. For release toggles this is particularly relevant, as the user is likely to have had a limited exposure to new functionality. If a problem appears in a feature that very few people have seen, then hardly anyone will have seen the problem.

On the other hand, for toggles that have been on for some time it may introduce risk for a feature to appear and then disappear e.g. there may be reputational damage in switching off premium features for loyal users.

Beyond testing how the software behaves, you should also think about testing the toggles themselves.

Can you easily determine the current toggle configuration? Is there a command to dump this from the live system? If you rely on reading the configuration file, how can you be sure that this is reflective of what is currently in use?

Is there audit information associated with the current toggle configuration? Can you see who was responsible for changing the state of each toggle and when that occurred? Is there a historic record of all state changes through time? What other information is associated with each of these audit records, or should be?

Are there active toggles that should be removed? If a toggle doesn't change state, whether that means it is always on or always off, then is it still required? One way to reduce combinatorial test explosion around toggles is to keep the smallest set possible. Part of testing is to challenge what currently exists; ask questions if you're not sure.

Finally, is the toggle needed at all? If the development work for a feature can be coded and released in small pieces where the user interface changes happen last, then a toggle might not be required as a mechanism to hide the functionality. Part of testing the toggle is to question the validity of implementing a toggle in the first place.

Jim Bird believes that feature toggles are one of the worst kinds of technical debt[55]. He explains how feature toggles can make it increasingly difficult to support and debug a system. With a large number of configuration options it can become difficult to understand and replicate problems found in the production environment.

[55] http://swreflections.blogspot.co.nz/2014/08/feature-toggles-are-one-of-worst-kinds.html

Bug bash

A bug bash is an activity where all the people involved in the delivery of software put aside their usual work and participate in a dedicated testing session. It involves development, management, design, operations, and any others who are willing to contribute to discovering problems in the product prior to its release. The term originated in Ron Patton's book titled Software Testing[56] first published in 2000.

There are many situations where a bug bash may be useful. They may be adopted as a final pre-production testing phase, or as a replacement for testing as a specialist discipline. They may be adopted by organisations who wish to release a high quality product, or as an in-house exploration of the production platform already in use by customers. The choice to use a bug bash depends on the type of feedback the team are seeking and the wider development process they are operating within.

Stephen Janaway writes about his Experiences of a Testing Bug Bash[57] with a mobile development team who didn't have any dedicated testers. They accumulated a number of stories that required testing, which they cleared by shifting the focus of all team members to testing at the same time.

Stephen took ownership of scheduling and preparing the bug bash. He planned "two 45 minute sessions with a tea break in between". In advance of the session he worked alongside the lead developer in the team to prepare testing charters for the stories that required testing.

> "Using charters enabled us to define the boundaries of the testing; ensuring there was not too much overlap. It also meant that people who were not so familiar with the story were able to quickly learn about it, and areas to test, without needing to spend time reading stories and Epics in JIRA."
>
> Experiences of a Testing Bug Bash[58]
>
> *Stephen Janaway*

Stephen also stocked the room with resources, including the stationery items that you'd expect to find in a workplace as well as some common testing resources including Elisabeth Hendrickson's Test Heuristics Cheat Sheet[59]. These were intended to provide inspiration if the non-testers were struggling to think of ideas while exploring the software.

The session was successful and Stephen's preparation, particularly in creating charters, was well received. He concluded that "it really helped everyone to understand the feature, to explore it and to test together, giving multiple viewpoints on potential issues."

[56] https://www.amazon.com/Software-Testing-2nd-Ron-Patton/dp/0672327988
[57] http://stephenjanaway.co.uk/stephenjanaway/experiences/experiences-testing-bug-bash/
[58] http://stephenjanaway.co.uk/stephenjanaway/experiences/experiences-testing-bug-bash/
[59] http://testobsessed.com/wp-content/uploads/2011/04/testheuristicscheatsheetv1.pdf

Theresa Neate writes about her experiences from a different context in Bug Bash: The Game[60]. Theresa organises her team to participate in bug bash competitions as a way to encourage shared ownership of product quality:

> "I run a bug bash as a game and competition with the highest point earners winning a famous Melbourne coffee, sponsored by me, the tester. I encourage playfulness and count on the competitive nature of my colleagues to make it fun and rewarding to find new bugs. As a team member I have no ego about anything I may have missed during testing. I encourage team ownership of our product; and want our team to use the bug bash to discover, remedy and learn from what WE may have missed."
>
> Bug Bash: The Game[61]
>
> *Theresa Neate*

Theresa leads the bug bash with an entirely different approach to Stephen. The participants only receive a broad testing mission rather than a specific test charter. They are grouped into pairs and have just under an hour for their time-boxed testing session that includes test setup, test execution, and defect reporting.

Her experience report emphasises the competitive spirit of the bug bash. Pairs are awarded points for the bugs that they discover, with more points for higher severity problems. The Product Owner is tasked with determining allocation of points and declaring the winner at the end of the event.

Though fiercely contested, the bug bash is ultimately a fun and light-hearted way for the team to work together. Theresa sees the benefits of a bug bash as going beyond the problems it discovers, to also having a positive effect on team morale and an improved collective confidence in the product that has been developed.

Scott Berkun offers a few other practical considerations in How to run a bug bash[62]. He suggests scheduling the event in advance to avoid creating panic, to make sure the code is stable during the bug bash (to help isolate problems), and to provide an illustration of what a good bug report looks like. The last point is important for setting direction as you would rather the team discover a handful of useful problems than raise a plethora of trivial ones.

In some DevOps environments, a bug bash may bring together the development and operations teams to focus on quality, particularly where there are no dedicated testers. It might be useful to introduce a holistic consideration of quality where the team are generally focused in small pieces of work. Or it could simply be framed as a fun team-building activity, a competition to see who can uncover the strangest quirks for a token reward.

In other DevOps environments a bug bash may not be a sensible choice. If the team are releasing multiple times per day there may not be a window of opportunity to explore a fixed release, or

[60] http://www.testingtrapezemagazine.com/wp-content/uploads/2016/12/TestingTrapeze-2016-December-v2.pdf
[61] http://www.testingtrapezemagazine.com/wp-content/uploads/2016/12/TestingTrapeze-2016-December-v2.pdf
[62] http://scottberkun.com/2008/how-to-run-a-bug-bash/

value to doing so, if users are testing in production. If there is a beta testing program there may be reduced value in running a bug bash alongside it. As with adoption of any practice, consider the whole development approach.

Crowdsourced testing

Crowdsourced testing gathers people from all walks of life, potentially in different geographies, to test a product. It can be used while the product is still in development to invite early perspectives of potential and existing users without releasing to production. It can also be used once the product has been released to gather a wider variety of feedback.

Most of the literature about crowdsourced testing comes from organisations who are looking to sell it as a service. This can make it difficult to determine how effective crowdsourced testing actually is. The advantages and disadvantages of the approach are well agreed across the industry:

Advantages	Disadvantages
Variety of test environments inherent in a crowd	Confidential aspects of the software may be leaked to competitors
May be cost effective where payment is only provided for valid bug reports	Payment model may result in a large number of trivial problems reported
Unbiased perspective from people who are external to the organisation	Communication with a large group of crowdsourced testers may be difficult
Diverse opinions from multiple locales	Management challenge to coordinate timezones, languages, etc.
Can be fast when engaging a lot of people for a short period	

For organisations who are starting the DevOps journey, or those who are more focused on product quality than speed to market, crowdsourced testing may be an alternative to beta testing. Both provide the same type of feedback, but with slightly different risks and costs.

TESTING IN PRODUCTION

Every test strategy is a balance between the need to release quickly and the need to test thoroughly. Testing in production is a way to tilt the balance towards speed. By releasing software earlier and adopting practices that allow the user to test, the development team can mitigate risk once the software is in operation.

Testing in production means that some decisions that would normally be made by the development team can be deferred. Decisions can be driven by data from actual users interacting with production software.

There are a number of different practices that fall under the banner of testing in production: A/B testing, beta testing, and monitoring as testing. The success of each relies on established communication paths between development and operations.

Should your new design use a button or a link as a call to action? Try an A/B test. Does your software work across different browsers and platforms? Try a beta test. Can you keep a quick response time with many concurrent users? Try monitoring as testing.

For testing in production to be effective, the automated feedback available from production environments must be robust in the areas of interest for testing. Information provided by production monitoring and alerting, analytics events, and logging must be sufficient for the team to have visibility of both user and system behaviour.

You need to think carefully about what information is required to determine the result of a test in production. It's difficult to test adoption of a feature without appropriate analytics being captured based on the user's decision. It's difficult to track performance of the software without monitoring the relevant system measures including CPU utilisation, memory use, response times, etc.

The merits of testing in production can only be fully realised when the wider IT organisation are working together to create experiments, identify relevant measurements, collect results, and use those to determine their collective next steps.

Feedback in production

The production environment can provide a rich array of feedback about the use of your software through monitoring and alerting, analytics events, logging, and customer feedback. These sources are not new, but the culture change that occurs with a move to DevOps offers the opportunity for a conscious shift in test strategy to take advantage of this data. The development team needs to:

- become more aware of the feedback created in operations,
- understand what information it provides, and
- look to improve and utilise it.

Monitoring and alerting

Once in production, the fastest automated feedback about software and the environment that it operates in comes from monitoring. People in the operations team can create dashboards for key information, and configure alerts for unexpected events.

Monitoring and alerting are formally defined as:

> "*Monitoring* is the process of maintaining surveillance over the existence and magnitude of state change and data flow in a system. Monitoring aims to identify faults and assist in their subsequent elimination. The techniques used in monitoring of information systems intersect the fields of real-time processing, statistics, and data analysis. A set of software components used for data collection, their processing, and presentation is called a monitoring system.
>
> *Alerting* is the capability of a monitoring system to detect and notify the operators about meaningful events that denote a grave change of state. The notification is referred to as an alert and is a simple message that may take multiple forms: email, SMS, instant message (IM), or a phone call. The alert is passed on to the appropriate recipient, that is, a party responsible for dealing with the event. The alert is often logged in the form of a ticket in an Issue Tracking System (ITS), also referred to simply as ticketing system."
>
> Effective Monitoring & Alerting[63]
>
> *Slawek Ligus*

[63] https://www.safaribooksonline.com/library/view/effective-monitoring-and/9781449333515/ch01.html

Effective monitoring can discover production issues quickly, which can enable more effective testing in production. For example, if you release a new version of an application and instantly observe performance degradation, you can immediately respond. In organisations that value rapid release, this process may be preferable to a stress test before release.

Monitoring information can also help you prioritise test exploration for future development activities, where you focus on areas of high use or previous concern. This information can come from people in operations as well as from the dashboard. People will usually remember the richest information about trends over time and be able to provide stories to accompany the numbers.

As an example, I worked on software for an online marketplace. The software actively monitored the error codes returned to users. The graph showing the number of errors over time was constantly on display. Development teams could react to an unexpected spike in errors as an indicator that a release had introduced a problem.

When I joined the organisation the default state of this graph was a constant trickle of errors. There weren't many, but they had become ingrained and weren't seen as a high priority to fix. A few people from the operations and development team understood the root causes of the problems and advocated for them to be resolved. They remembered when the errors had started, understood the different situations that generated an error, and knew who would be best to fix them. It was their persuasive action, rather than the dashboard alone, that returned the default state to an empty graph.

This activity had a flow-on impact to exploratory testing of future work, as it raised awareness of monitoring for these types of errors prior to a release to production.

Testing monitoring and alerting

Monitoring and alerting provide feedback as users interact with the production environment, which make them a source of information when adopting practices of testing in production. However, monitoring and alerting are also a testable component of the system in their own right. So, how can you test them?

When configuring a new monitor, ideally you will test that it works by causing an issue that the monitor should detect.

If you cannot trigger a monitor through a real environment, you may be able to pass mocked data to the monitoring system to simulate the problem. The ability to have full control over the data that triggers a problem can make it easier to distinguish the test from real day-to-day system activity.

The ability to control test data can also help where monitoring configuration is complex. A single system observation might be able to trigger multiple monitoring thresholds, with each causing a different set of actions to occur. Instead of a simple binary state, the monitor may be at varying levels of severity with different people notified depending on state. For particularly important measurements, it may be beneficial to test the rules that have been defined are sensible and work as expected.

If you have configured a monitor that generates an alert, you may wish to test that the alert is delivered successfully. Be aware who will see the alerts generated by your testing, as you won't want to cause alarm about the state of your production environment!

As well as testing triggering, it is also sensible to test how the monitor and alert are cleared. Are the alerts marked as resolved? Are they archived or removed? Is an associated notification generated? What information is recorded when a problem is resolved automatically? If the monitor has a graphical dashboard view, does this update as you expect?

Finally, take a step back from the individual measures and look at the big picture. How noisy are the alerts collectively? They should cover important situations without creating repeated false alarms. What is the process when an alert is received? If an alert fires correctly, but the required response isn't clear, then the monitoring solution as a whole has failed.

Analytics events

Analytics refers to the data we collect about our users' activity with our product. In a web application this might be page views. In a telephone network it might be call durations. In a banking product it might be transaction volumes. Organisations collect all sorts of data to learn about their users' behaviour and respond to it.

Analytics data can be collected directly by measuring actions that users are consciously undertaking, such as using a website, a telephone, and a banking application. Analytics data can also be captured indirectly through sensors and tracking functions. A fitness tracking watch, a home weather station, or a ventilation system capture information through their environment.

The scope of analytics can include both information about user activities and information about their systems or platforms. A website might count which pages are viewed, but also track which browser is being used. A fitness tracking watch might measure heart rate, but also track the devices that data is synced to.

Capturing analytics events for large systems will create a lot of information, from which the term 'big data' has emerged. Extracting value from the vast volume of events generated can be complex. The people who interact with these data sets generally have a broad set of skills across computing, statistics, machine learning, and analysis. They might be referred to as Data Scientists, or Business Intelligence, or simply the Analytics Team.

There are a variety of ways to track users directly. Small web applications may simply use Google Analytics to capture fundamental analytics for their site. Larger applications may generate analytics events as the software is used, storing them in a database.

When a development team is updating or extending software, they should think about the impact on analytics events. Be mindful of what is recorded, and work closely with the analytics team to determine what's most appropriate to track. This can give you better insights once the code is in production. It's important to test that there are no blind spots in your analytics solution.

Testing analytics

Testing of analytics will generally focus on the design of information rather than the specific behaviour of individual pieces of data. It's trivial to see a number increment. What is interesting to test is the holistic view of how people are tracked through various workflows.

As an example, imagine a payments workflow in an online banking application. A user can transfer money between their accounts using a 'quick transfer' function or the traditional 'payments' function. Each workflow is present on both the web and mobile application.

It is relatively simple to make a transfer and see that the total number of transfers increment. The data collection will generally work correctly. It is more interesting to test how the analytics increment differently in each scenario and question whether the design of the data collection is correct. Should

you record ‘quick transfer’ and ‘payments’ actions as separate events? Should you record web and mobile actions separately?

Beta and A/B experiments are supported by analytics information. Fast feedback may appear through monitoring, but it is often through deeper analysis of analytics data that final decisions are made.

Logging

Log files record transactional and status information, along with errors and warnings that are generated by unexpected activity. They will be generated by:

- your own software,
- the third party applications and libraries that it integrates with, and
- the platform on which it operates.

When problems are encountered in production, log files often provide the low-level, detailed information to diagnose the root cause.

It is easy to take log files for granted, particularly if you've never worked in a support role that relies on the information that they contain. The development team should work to understand what types of information are valuable to log, and how to categorise messages appropriately as errors, warnings, information, or debug.

Testing logging

A bug in development can be a good opportunity to test log files. Any problems should create appropriate log messages. If you cannot determine the bug from log files alone, this may indicate that your logging is either insufficient or so verbose that it is impossible to locate the problem! If you can determine the bug, it may be worth checking that someone from your support team is able to reach the same conclusion from the log file.

Analysis of production log files might not happen much beyond incidents, but it can reveal interesting user behaviour and hidden issues in the application. One example that I have seen was a production log that contained a lot of similar error messages. It turned out that users had become accustomed to clicking a button twice, because a bug in the application prevented the first click working. The first click was generating an exception that was only visible in the log.

Matthew Skelton recommends using "logging as a channel to make distributed systems more testable[64]". He suggests concentrating testing of log files on these three points:

1. Events that we expect to occur appear correctly in the log stream
2. Transaction identifiers (aka correlation IDs) flow through the log stream as expected
3. Events are logged at the appropriate level (Info, Error, Debug, etc.) if we're using configurable log levels

Why and how to test logging[65]

Matthew Skelton

[64] http://www.slideshare.net/SkeltonThatcher/why-and-how-to-test-logging-devops-showcase-north-feb-2016-matthew-skelton
[65] https://www.infoq.com/articles/why-test-logging

The second point is particularly relevant in complex architectures where a single user action is processed by multiple systems. There must be a way to associate messages from different log files into an aggregated workflow. Without this connected view, it can be difficult to detect anomalies.

Managing log files is important too. Consider policies for logging rotation and retention. Very large log files become unwieldy to search and transfer. Keeping a large number of log files can consume excessive disk space. Ensuring that log files are archived when they reach a certain size and deleted when they reach a certain age can mitigate these problems. It is always a good idea to check that archived logs can be restored correctly. Your log management solution may also require testing, depending on your implementation.

Customer feedback

Your software might offer opportunities for users to provide feedback. This may be a ‘Contact Us’ email link at the bottom of a web page, an embedded feedback form, or an integrated chat application that connects users with your support team in real-time. Your organisation might offer support to your users, via a call centre or online ticketing system.

Each of these channels provides feedback. The people who use your software will be discovering the unexpected every day. These channels will capture information born from frustration and confusion. There will be problems that your development team had never pondered. Occasionally they’ll record praise, satisfaction, or suggestions for improvement.

Analytics provide quantitative feedback. They tell us what people are doing and the impact of their actions on the system. Customer channels provide qualitative user feedback. We learn how people are feeling, their reactions to a product, their reasons for performing a task via a particular path through the product. User feedback can provide a richer perspective than analytics, but one that can be difficult to process.

Customer feedback is most successful when people in the development team are exposed to it and make a conscious effort to make use of it. The very presence of feedback is a kind of feedback. If a member of the public felt strongly enough to comment, give them the consideration they deserve.

On the flipside, feedback is a gift not an obligation. Feedback from users doesn’t always need to create change. Sometimes it’s an emotional reaction, rather than a genuine problem with the software. Feedback may go no further than raising awareness in the development team.

Test practices in production

In a traditional project the development team may test in production after a large release to gain confidence that their changes have been applied successfully. It is a final check in a linear process, and rarely generates new work, with some exceptions for critical issues.

In a DevOps world, testing in production is a way to generate constant input to the development team. It forms part of the delivery cycle, providing feedback that shapes future iterations of the software.

This section explains three common practices for testing in production:

Name	Description
A/B testing	Use a controlled experiment to present two versions of your software to your users to see which one they prefer
Beta testing	Release a new version of your software to a subset of users to determine its readiness for a wider audience
Monitoring as testing	Use production monitoring to identify when problems happen and to confirm when they are fixed

A/B testing

A/B testing is when you create two different variants of your software, then present each user with one of the variants in a controlled experiment. The results show which variant users prefer overall: A or B.

For example, imagine a subscription button for the online edition of a major newspaper. An A/B test might compare a new version of the button, with different phrasing on the label, to the original version. During the A/B test, the newspaper will track how many subscriptions are generated by each option. Once a statistically significant sample of users have subscribed, the results are analysed. If the new version attracts more subscriptions it will be adopted. If not, the existing version will be retained. Sometimes there is no clear winner, in which case either option is equally appropriate.

With A/B testing there's an assumption that both options of software have been *verified* prior to the experiment. You are not expecting the users to discover bugs. You use A/B tests to *validate* your solution: have you built the right thing?

Testing the A/B experiment

Though an A/B testing experiment is likely to be designed outside the development team, by data scientists, marketing or product management, it's important for you to understand:

- What are we measuring?
- How long will the experiment run for?
- Who will be included in each audience?
- What will each experience be like for our users?

If you think about testing within the development team, what differentiates an A/B testing release from a typical release to production? Both variants of the software are expected to work well. There is unlikely to be change to how you verify the software prior to release. The difference is testing the experiment that you will run to give you confidence in the results that it provides.

Check there is at least one specific measurement that the experiment aims to influence. How is the measurement expected to change? What impact would the desired change have on the wider organisation? There may be a financial benefit, which could be direct, like increasing sales, or indirect, like reducing call center volumes. You may look for non-financial benefits too, like increasing the amount of time that a user spends reading information on your website.

Ask how long the experiment will run for. It can be tempting to choose the winner of an experiment before there is a statistically significant outcome, particularly if early results reflect your own expectations. Make sure you understand the amount of time or number of interactions that are required for the result to be robust.

Who will see each variant? How can you be sure that the results are due to your software and not the people using it? Probabilistic equivalence means that we can compare the effect of a variation in a way that assumes all other factors remain constant. In A/B testing we can create probabilistic equivalence by using randomisation in selection of the groups involved in our experiment.

This academic paper excerpt explains why randomisation is important for measuring the impact of different advertisements on the purchasing habits of consumers:

> "What makes randomization so powerful is that it works on all consumer characteristics at the same time – gender, search habits, online shopping preferences, etc.. It even works on characteristics that are unobserved or that the experimenter doesn't realize are the related to the outcome of interest. When the samples are large enough and have been truly randomized, any difference in purchases between the conditions cannot be explained by differences in the characteristics of consumers between the conditions — they have to have been caused by the ad. Probabilistic equivalence allows us to compare conditions *as if consumers were in two worlds at once*."
>
> A Comparison of Approaches to Advertising Measurement:Evidence from Big Field Experiments at Facebook[66]
>
> *Brett Gordon, Florian Zettelmeyer, Neha Bhargava, Dan Chapsky*

Consider the experience for users in the experiment by testing that your software is consistent for individuals. If someone is part of the control group, they should see the control software in every interaction they have during the experiment. If someone is part of the group that sees the new design, it should be applied to every appropriate part of the software and not just intermittently.

Limitations of A/B testing

A/B testing may become addictive to your organisation. There's the well-known example of how Google used A/B testing to determine which shade of blue to use on their toolbar. They tested 41 gradients to learn the colour people preferred, which was seen by some as a trivial attribute to experiment with.

Marissa Mayer, then VP of Search Products & User Experience at Google, justified the decision to A/B test in a New York Times article:

> "As trivial as color choices might seem, clicks are a key part of Google's revenue stream, and anything that enhances clicks means more money."
>
> Putting a bolder face on Google[67]
>
> *Laura M. Holson*

[66] https://www.kellogg.northwestern.edu/faculty/gordon_b/files/kellogg_fb_whitepaper.pdf
[67] http://www.nytimes.com/2009/03/01/business/01marissa.html

Douglas Bowman, a former designer at Google, wrote about the negative impact of that same A/B test in the context of the wider attitude to design decision making:

> "With every new design decision, critics cry foul. Without conviction, doubt creeps in. Instincts fail. "Is this the right move?" When a company is filled with engineers, it turns to engineering to solve problems. Reduce each decision to a simple logic problem. Remove all subjectivity and just look at the data. Data in your favor? Ok, launch it. Data shows negative effects? Back to the drawing board. And that data eventually becomes a crutch for every decision, paralyzing the company and preventing it from making any daring design decisions."
>
> Goodbye, Google[68]
>
> *Douglas Bowman*

A/B testing should not become the only way to make a decision. If you start to defer every decision to your user, you remove autonomy from the development team. Customer behaviour can provide insight into incremental improvement, but people may react poorly to a more significant innovation that challenges them. There is a famous quote often misattributed to Henry Ford: "if I had asked people what they wanted, they would have said faster horses."

[68] http://stopdesign.com/archive/2009/03/20/goodbye-google.html

Beta testing

Beta testing is when you release a new version of your software to a subset of users to determine its readiness for a wider audience. The users who participate in a beta test may be internal to your organisation, a wider group that includes friends and family, or a subset of the general public. Ideally the test audience will be representative of the entire production user group.

In contrast to A/B testing, the primary focus of beta testing is to verify your product. You are expecting users to discover bugs. The problems they find may differ to those discovered by the development team due to software interaction habits of the individual, the specific data that they store in the software, the hardware on which they operate, their geographical location, their physical abilities, and any number of other variants. Beta testing provides a diversity of perspective that is not possible within the development team alone.

The adoption of beta testing in your organisation may be driven by the limitations of testing within your development team. Your software may support a specialist profession that prevents those without this knowledge from exercising its full functionality, may run on a wide variety of platforms that cannot all be replicated by your own test environments, or may give a more realistic view of performance measurements than your internal benchmarking.

Beta testing might also be driven by product strategy. If you want to be the first to market, then beta testing can help put software in front of people earlier. The theory is that though it may not be perfect, your software is seen as innovative and exciting. A strategy that differentiates your organisation from its competitors.

Testing alongside beta testing

At first glance, it may seem that beta testing means outsourcing all of your testing to your users. While it certainly shifts how some risks are addressed, beta testing also offers an opportunity for the development team to do some testing of their own in the production environment.

Users will often be asked to give direct feedback during a beta test, but they'll also be providing feedback through their actions. A beta release is an opportunity to test alarms, monitoring, analytics events, and logging. Unusual automated feedback may be how you discover bugs beyond what users have visibility of.

Generally users will focus on the functional aspects of the application. You may not receive feedback on the non-functional aspects of the application from users, except in situations that cause pain. For example, extremely poor responsiveness or an exceptionally non-intuitive user interface. A beta test environment can be a good place for the development team to perform their own usability, accessibility, performance, or penetration testing.

Users given access to beta software are unlikely to be malicious, their testing is likely to focus on their ability to complete actions successfully. They may not trigger error conditions or explore areas of functionality outside their normal activities. The development team can complement beta testing by focusing their efforts on negative or unusual scenarios.

It may also be valuable to have a person from the development team sit alongside a user and observe their behaviour, logging any problems on their behalf. This paired exploration of the system may help to improve both the identification and reporting of bugs.

To get the most value from beta testing, look for ways that the development team can add to the feedback that users provide.

Limitations of beta testing

Brian Marick cautioned against an overreliance on beta testing in his 1997 paper titled Classic Testing Mistakes[69]. He listed five problems with beta testing that can be summarised as:

1. If beta testing is optional, the users who self-select will not be representative of all users. There will be a skew towards early adopters of new technology, who will have different expectations to a pragmatist who waits to adopt until after the technology is proven.
2. Beta testers may only do a quick tour of the product rather than a deep exploration, which will leave problems undiscovered.
3. Usability problems may not be reported, as beta testers could be uncertain about whether these are important.
4. Beta testers may not report all the bugs that they observe, especially if they're not sure what they did to cause the problem.
5. The bug reports of beta testers are often difficult to interpret and it may require a lot of effort to identify the root cause.

Running the beta test program with users selected by the organisation, and having testers from the development team in the beta testing phase, can help you avoid these problems.

Beta testing may not suit DevOps, as it best suits organisations with relatively infrequent releases. If small pieces of work are being released hundreds of times per day, there is little value in each of these pieces going through beta. Problems may as well be identified and reported by the whole user base.

If DevOps is a goal, beta testing may introduce more risk into the release than the organisation has previously felt comfortable with. It can force a change in test strategy, reducing the test effort within the development team, in the understanding that the beta test group will cover some areas. It can force a change in development and operations practices to support multiple versions of the product running concurrently. Setting up beta testing introduces a lot of the scaffolding - feature toggles, analytics, monitoring - for other practices of testing in production that can be reused as DevOps in the organisation matures.

Beta testing will introduce a different degree of risk in different organisations. In some cases a beta test will mean releasing the code that the developer just finished writing. In other cases it will mean releasing well-tested code for feedback from a broader set of people.

[69] http://www.exampler.com/testing-com/writings/classic/mistakes.html

Google are well known for running lengthy beta programs. Gmail, was labelled as beta for five years from 2004 to 2009. In this case, beta became a signal for continuous improvement of the product rather than a meaningful and distinct testing activity. Even co-founder Larry Page once admitted that using a beta label for years on end is "arbitrary" and has more to do with "messaging and branding" than a precise reflection of a technical stage of development[70].

Tesla have received some criticism for their decision to beta test the autopilot feature of their vehicles. An article that was published after one of their vehicles was involved in a non-fatal accident read:

> "Whether or not Tesla's autopilot malfunctioned, the news reinforces the question of whether car companies should release critical safety features to be beta-tested by consumers. The autopilot software, which Tesla is testing in a "public beta", has been installed in all 70,000 of its cars since October 2014.
>
> "It is just not acceptable," says prominent analyst Ferdinand Dudenhöffer, head of the Centre of Automotive Research at the University of Duisburg-Essen in Germany. "Cars are not toys and to launch a beta version of software with safety risks is not the way a company should care about the life of human beings. Just putting a sticker on it saying 'customer is responsible' is a nightmare"."
>
> Should Tesla be 'beta testing' autopilot if there is a chance someone might die?[71]
>
> *Olivia Solon*

All of these limitations are opportunities for discussion when implementing beta testing in your organisation. You will want to consider the audience, scope, and risk in beta testing for your context.

[70] http://www.slate.com/articles/news_and_politics/recycled/2009/07/why_did_it_take_google_so_long_to_take_gmail_out_of_beta.html
[71] https://www.theguardian.com/technology/2016/jul/06/tesla-autopilot-fatal-crash-public-beta-testing

Monitoring as testing

You can use production monitoring to identify when problems happen and to confirm when they are fixed, as an alternative to testing before release. This practice is called 'monitoring as testing'.

Using monitoring as testing is a response to the pace of delivery in a DevOps environment. There is a degree of risk in frequent releases that is mitigated by having fast feedback in production. Monitoring as testing is reactive rather than proactive: problems are discovered after they have happened. They must be fixed, and the fix deployed, as fast as possible. The test to determine whether the problem is resolved comes from the monitor returning to normal after the release.

Where did the idea of monitoring as testing originate?

Ed Keyes delivered a short talk at the Google Test Automation Conference in 2007 to argue that "Sufficiently advanced monitoring is indistinguishable from testing"[72]. He advocated for monitoring that included functional verification and log message analysis, used to detect problems related to two types of input:

1. User events. Treat every interaction as a test case that "you didn't have to think of and maybe wouldn't have".
2. Automated tests. Run targeted, scripted actions against the production environment regularly.

Ed saw pre-release testing and post-release monitoring as unnecessary duplication of effort. In an environment where short-term production problems are acceptable to users, why invest in prevention? Instead, focus on generating or observing events that provide rapid detection.

At DevOps Days in 2015, Leon Fayer presented a more moderate stance in his presentation titled Testing and monitoring in production[73]. Rather than monitoring *as* testing he advocated for monitoring *and* testing. Leon positioned monitoring as a component of test strategy to help discover "unknown unknowns".

Merging ideas

Testing and monitoring have traditionally served different purposes. The focus of testing is generally to identify problems before the user experiences them in production. The focus of monitoring is to identify any remaining problems after they have happened in production.

Differences between the two can be summarised as:

[72] https://www.youtube.com/watch?v=uSo8i1N18oc
[73] https://www.youtube.com/watch?v=VIhIm5kEWFg

Testing	Monitoring
Active examination	Passive observation
Tests focus on product	Monitors focus on performance
Problems found before release	Problems found after they happen
Problems identified immediately	Problems identified by trend
Test environment	Production environment

Software Testing and/or Software Monitoring: Differences and Commonalities[74]

Antonia Bertolino

Monitoring as testing blurs these differences, pulling ideas from testing into creating and testing monitors.

Instead of waiting passively for a user to interact with a feature, run test automation to generate events in production. You may discover and resolve a bug in production before a user experiences it. These changes make monitoring more active, specific and immediate.

Seth Eliot distinguishes two types of data-driven validation, effectively the same as monitoring as testing, in his TestBash 2013 presentation: passive and active. He explains the differences between the two:

> "When I say *passive* validation people think it must be a negative thing. It's not. It's not a negative term at all. Passive validation is very powerful. But why is it called passive validation? There's data out there, either we instrumented for it or it's ambient data, but there's data out there that we're consuming and making quality decisions based on that data. So it's passive, we're not actually executing anything other than analysis of the data. We're not executing anything against the system under test.
>
> So *active* validation is when we execute things against the system under test. And when do we execute things against the system under test? We do that when we run test cases, before we release, right? So, active validation is using something called synthetic transactions, and synthetic transactions look a lot like test cases. Test cases are a subset of synthetic transactions. Synthetic transactions are 'I make up some data, I make up some workflow or scenario, and I execute it against the product to see the results'. That's what I mean by active validation."
>
> Testing in Production[75]
>
> *Seth Eliot*

Seth talks about a test strategy that uses both types of validation in production. The two strategies complement each other. Passive validation gives insight into how users behave, by monitoring

[74] http://www.slideshare.net/ProjectLearnPAd/cadiz-bertolinolp
[75] https://vimeo.com/64786696

information for real interactions. Active validation will generate observable data by executing known repeatable scenarios.

One of the examples that Seth gave of using passive validation was mining a Twitter stream for information about the Xbox Kinect product. They created a dashboard that graphed whether people were expressing positive, negative, or neutral sentiment about the product, along with common phrases that were appearing in tweets. They used this information for assessing quality. When they released a new version of the product, "if you saw negative sentiment shoot up, or you saw the phrase 'bug' or 'piece of garbage' shoot up, then you know you have a problem[76]".

In a separate article, Seth gives an example of active validation in Microsoft Exchange. Seth highlights the difference between verifying in a lab versus production. In the lab, the focus is on a pass or fail for behaviour of a feature. In production, the focus is on the overall measurements of the system. He says:

> "Microsoft Exchange … re-engineered their 70,000 automated test cases they used to run a lab so that they could run them in production with their new hosted Microsoft Exchange Online cloud service. While test cases in the lab give us the traditional pass/fail signal, we can take a different approach in production, instead looking at the success rate over thousands of continuous runs. Success or failure is measured by availability and performance KPIs. For a scenario did we meet the "five nines" (99.999%) availability? Or did we complete the task in less than 2 seconds 99.9% of the time? The signal is still the same, it is still the data coming out of our system under test, but we are triggering that signal with our testing."
>
> The Future of Software Testing, Part Two – TestOps[77]
>
> *Seth Eliot*

Limitations of monitoring as testing

Passive validation means that the development team may need to handle sensitive user information. Active validation may change data in the production environment - you must take care not to annoy the user.

Poorly chosen data for active validation may be obvious. A longstanding example is Amazon, who create test items for sale in their production systems. A search for "test ASIN", where ASIN is the Amazon Standard Identification Number assigned to all items for sale on the Amazon site, returns a number of items that are "for sale". Seth Eliot claims:

> "This is [testing in production] done poorly as it diminishes the perceived quality of the website, and exposes customers to risk – a $99,999 charge (or even $200 one) for a bogus item would not be a customer satisfying experience."

[76] https://vimeo.com/64786696

[77] https://www.ministryoftesting.com/2013/06/the-future-of-software-testing-part-two-testops-3/

> Testing in Production (TiP) – It Really Happens – Examples from Facebook, Amazon, Google, and Microsoft[78]
>
> *Seth Eliot*

Poorly chosen data for active validation can also be more insidious. A famous example of active validation having unexpected consequences is Harvard Business School Publishing (HBSP). In 1999, HBSP created a new website along with "an aggressive testing strategy". This meant that they would run active validation on their production site.

> "This aggressive testing strategy ensured that the site would function as intended for years to come. That is, until that one day in 2002. On that day, one of the test cases failed: the "Single Result Search".
>
> The "Single Result Search" test case was part of a trio of cases designed to test the system's search logic. Like the "Zero Result Search" case, which had the tester enter a term like "asdfasdf" to produce no results, and the "Many Results Search" case, which had the tester enter a term like "management" to produce pages of results, the "Single Result Search" case had the tester enter a term – specifically, "monkey" – to verify that the system would return exactly one result."
>
> I've got the monkey now[79]
>
> *Alex Papadimoulis*

For three years the search term "monkey" returned a single result, a book titled "Who's Got The Monkey?". Until the day that the test case failed as it returned a second result, an update to the wildly popular original that was titled "Who's Got The Monkey Now?".

Initially the test team assumed that they simply needed to find a new search term for their "Single Result Search". Then they did a little bit of research to understand the best selling sales figures that had created a compelling case for a new book.

Their testing strategy was searching for "monkey" and then:

> "they'd add that single result – Who's Got the Monkey? – to their shopping cart, create an new account, submit the order, and then fulfill it. Of course, they didn't actually fulfill it – everyone knew that orders for "Mr. Test Test" and "123 Test St." were not to be filled. That is, everyone except the marketing department."
>
> I've got the monkey now[80]
>
> *Alex Papadimoulis*

[78] https://blogs.msdn.microsoft.com/seliot/2011/06/07/testing-in-production-tip-it-really-happensexamples-from-facebook-amazon-google-and-microsoft/

[79] http://thedailywtf.com/articles/Ive-Got-The-Monkey-Now

[80] http://thedailywtf.com/articles/Ive-Got-The-Monkey-Now

The active validation strategy for testing in production had polluted the sales figures used by the marketing department.

There are also limitations in adopting monitoring as testing in isolation; using monitoring *instead* of testing, rather than *as well as* testing. Jim Bird doesn't pull any punches in his description of how stupid he believes this idea to be:

> "If you're in an environment where you depend on developers to do most or all of the testing, and then tell them they should put monitoring in front of testing, then you aren't going to get any testing done. How is this possibly supposed to be a good thing?"
>
> Monitoring Sucks. But Monitoring as Testing Sucks a Lot More[81]
>
> *Jim Bird*

Yet, this does work in some environments. Facebook is a well known example of an organisation that emphasises testing in production (including monitoring as testing) above pre-release testing. People from within the organisation acknowledge:

> "This process works for Facebook partly because Facebook does not, by and large, need to produce particularly high-quality software."
>
> Is it true that Facebook has no testers?[82]
>
> *Evan Priestley*

This sentiment is the crux of most techniques that shift mitigation of risk to the user. What are their expectations? Will they accept and embrace a product of a poorer quality? Testing in production will make the quality of user experience slightly unpredictable for some percentage of your user base. You need to think carefully about how much this will matter to them.

[81] http://swreflections.blogspot.co.nz/2012/07/monitoring-sucks-but-monitoring-as.html
[82] https://www.quora.com/Is-it-true-that-Facebook-has-no-testers

Exposure control

Exposure control is controlling the introduction of new software to users to limit the exposure of the organisation that has written the software. Exposure control is a phrase coined by Seth Eliot and Ken Johnston in 2009 at Microsoft[83].

You may be familiar with exposure control through the names of specific practices that reduce risk when releasing software, for example canary release, staged rollout, dogfooding, and dark launching. This section will explain each practice in more detail.

Canary release

The phrase "canary release" has its origin in mining. Historically, miners would take canaries with them when they worked underground to provide a warning of toxic gases. Due to their smaller size, the birds were more sensitive to the toxic gases that can appear in a mine than their human companions. If the canary died, this was a signal to the miners to evacuate immediately.

Similarly in software development, a canary release is small and intended to catch problems. A 'dead canary' would be a signal to abandon a wider scale release.

> "Canary release is a technique to reduce the risk of introducing a new software version in production by slowly rolling out the change to a small subset of users before rolling it out to the entire infrastructure and making it available to everybody."
>
> Canary Release[84]
>
> *Danilo Sato*

A canary release can be created by limiting the infrastructure that runs new code. Facebook take this approach by applying releases to a subset of servers:

> "Prior to rolling out a full update, the new code first gets pushed to … a small number of public Facebook servers. This stage of the testing process exposes the update to many random Facebook users, but still just a fraction of the site's total audience, and it gives Facebook's engineers an opportunity to see how the update will perform in production."
>
> Exclusive: a behind-the-scenes look at Facebook release engineering[85]
>
> *Ryan Paul*

[83] https://blogs.msdn.microsoft.com/seliot/2010/12/13/exposure-control-software-services-peep-show/
[84] https://martinfowler.com/bliki/CanaryRelease.html
[85] https://arstechnica.com/business/2012/04/exclusive-a-behind-the-scenes-look-at-facebook-release-engineering/2/

Timothy Fitz writes about continuous deployment where not only is there an automated canary, there is also an automated rollback procedure based on monitoring results from the canary release:

> "A symlink is switched on a small subset of the machines throwing the code live to its first few customers. A minute later the push script again samples data across the cluster and if there has been a statistically significant regression then the revision is automatically rolled back. If not, then it gets pushed to 100% of the cluster and monitored in the same way for another five minutes. The code is now live and fully pushed. This whole process is simple enough that it's implemented by a handful of shell scripts."
>
> Continuous Deployment at IMVU: Doing the impossible fifty times a day.[86]
>
> *Timothy Fitz*

The advantage of an infrastructure-driven canary is that infrastructure change is contained. If a canary dies, its impact is restricted to just a few servers and therefore users.

Cloud Foundry use an open-source tool chain called BOSH for releases, which uses automatic canaries by default. The tool applies changes to one server first, then verifies the outcome of the changes before continuing. Cornelia Davis shares her experience of a canary release that failed:

> "Our production deployment runs multiple instances of the marketplace services broker. Enter the sacrificial canary. When BOSH started the upgrade it only took down one of the brokers, upgraded the bits and tried to restart it. In this case the air in the coal mine was toxic and our little bird did not survive. As a result, we sent no additional souls in, leaving the other service brokers fully functional.
>
> We fixed our problem, pushed the "go" button again and this time the canary came up singing, BOSH upgraded the remaining service brokers and our production deploy was complete."
>
> Canaries are great![87]
>
> *Cornelia Davis*

A canary release reduces the risk associated with releasing a new version of software by limiting its audience. However, it also creates overhead in managing two different environments where users are experiencing different versions of the software. This is a challenge both for operations in their monitoring of the system and for development when they create the new version of the software.

Data management is especially challenging, here's an example from the development team at Facebook:

[86] http://timothyfitz.com/2009/02/10/continuous-deployment-at-imvu-doing-the-impossible-fifty-times-a-day/
[87] https://content.pivotal.io/blog/canaries-are-great

> Given Facebook's release system, all code had to be written with the idea that it would be executing in an environment with previous releases as well. This meant it was more difficult to do things like change data stores. For example, to move a field from one MySQL table to another would have taken five steps:
>
> 1. Introduce double writing. Both the old and new table would need to be written to on any update.
> 2. Push new release.
> 3. Run a migration to copy all the old data over to the new format.
> 4. Switch reads from the old system to the new system, remove old read/write code.
> 5. Push new release.
>
> How Do Facebook And Google Manage Software Releases Without Causing Major Problems?[88]
>
> *Justin Mitchell*

Staged rollout

A staged rollout is a canary release with a different focus. Instead of creating a canary by limiting changes to infrastructure, the rollout intentionally limits the number of users with access to the new code. Both methods allow a smaller initial release that can help to discover problems.

A staged rollout is still a phased movement of users from one version of software to another. Google offer a staged rollout for their Android mobile applications:

> "When you have a new version of your APK that you want to gradually deploy, you may choose to release it as a "staged rollout" version. If you do this, Google Play automatically deploys it to a fraction of the app's users; you can choose that fraction, from a number of allowable values. If the "rollout" APK doesn't have any issues (such as crashes, etc.), you might increase the fraction of users who receive that version; when you are ready, you can deploy that APK as the new production version."
>
> Google Play Developer API[89]

The staged rollout comes with its own limitations. If your organisation develops software that is installed on a user's own device, for example installing an Android application on your mobile, then you have less control over when the upgrade to the new version happens[90]. The timing of the change depends on when the user accepts the updated version. If a large number of people skip the update, then you have fewer users trying the staged release.

[88] http://www.forbes.com/sites/quora/2013/08/12/how-do-facebook-and-google-manage-software-releases-without-causing-major-problems/#5a2c78c1c0cb

[89] https://developers.google.com/android-publisher/tracks

[90] https://martinfowler.com/bliki/CanaryRelease.html

Another drawback of staged rollouts is that you may no longer control the A/B split for an experiment[91]. If you are gathering opinions on two different variants of new software, but the variants are only released to a subset of the user base, then the experiment may be skewed.

Dogfooding

Dogfooding is where the people who write the software are the first to use it:

> "Dogfooding means using the software you make, often in beta form, to work out the kinks. The first recorded usage was in 1988, when Microsoft executive Paul Maritz was desperate for customers to try a new product and e-mailed a colleague, "We are going to have to eat our own dogfood and test the product ourselves." They created an internal server called "\dogfood" and sent it out to staff."
>
> Dogfooding[92]
>
> *Nora Caplan-Bricker*

Dogfooding is the same as an employee-only beta release or having an environment dedicated to internal users. It limits exposure by using employees to discover problems.

The main limitation of dogfooding is that employees aren't always representative of general users. For example, the developers of a social media platform will understand expert features in a way that members of the general public may not. On the flipside, it may be that the users of the software are expert in comparison. For example, the developers of an investment banking platform will understand less about the financial domain than specialist users. Either situation will restrict the relevance of dogfooding.

Another possible disadvantage of dogfooding is a misalignment in the perception of quality. If employees use the pre-release version of the software and have a different expectation of quality compared to users, there are two possible outcomes. Employee feedback may become polluted with problems that are low priority to fix, because the employees have higher expectations than the users themselves. Or the feedback may be sparse, because the employees have lower expectations and are willing to accept problems that users would complain about.

Dark launching

Dark launching is a slightly different variant of exposure control. Instead of creating a small release to validate new software, it implements the new code but in a way that it is not visible to the user. This allows new code to be executed and monitored alongside the old code.

Facebook coined the term "dark launch" in the release of the Facebook Chat functionality in 2008:

[91] https://blog.twitch.tv/a-b-testing-using-googles-staged-rollouts-ea860727f8b2#.8posd2h13
[92] https://newrepublic.com/article/115349/dogfooding-tech-slang-working-out-glitches

> "The secret for going from zero to seventy million users overnight is to avoid doing it all in one fell swoop. We chose to simulate the impact of many real users hitting many machines by means of a "dark launch" period in which Facebook pages would make connections to the chat servers, query for presence information and simulate message sends without a single UI element drawn on the page. With the "dark launch" bugs fixed, we hope that you enjoy Facebook Chat now that the UI lights have been turned on."
>
> Facebook Chat[93]
>
> *Eugene Letuchy*

A dark launch pushes traditional testing into the production environment. It is the development team who retain ownership of exploring the new code and identifying potential problems. Dare Obasanjo writes that the "main problem with a dark launch is that it ignores the fact that users often use social features a lot differently than is expected by site developers[94]".

A dark launch is likely to be paired with a staged rollout when the user-facing piece of the feature is switched on. This approach allows the first users to uncover behaviour that the development team did not anticipate, and lets the development team react to any problems that may appear as a result.

[93] https://www.facebook.com/note.php?note_id=14218138919

[94] http://www.25hoursaday.com/weblog/2008/06/19/DarkLaunchesGradualRampsAndIsolationTestingTheScalabilityOfNewFeaturesOnYourWebSite.aspx

TESTING IN DEVOPS ENVIRONMENTS

Within the last 20 years the infrastructure landscape has changed dramatically. Environments created on virtual infrastructure, using automation, have created new challenges and opportunities for testing. This section introduces the key pieces of platform evolution then discusses their impact.

Platform evolution

In a DevOps organisation the number and types of environments available to a development team are likely to change dramatically. This reflects the changes that DevOps drives in production infrastructure. Environments built with configuration management tools that execute automated, scripted actions replace the actions of a system administrator who manually completes these steps. Hardware requirements shift with the adoption of container-based architecture and cloud systems. These changes increase expectations that a development team member can create a production-like environment on their own machine for their personal use.

This section explains some key ways in which platforms have changed in recent years: infrastructure as code, configuration management, containers, and cloud.

Infrastructure as code

Historically, configuring a server to run a software product was a manual process. A person would install the operating system, prerequisite applications and libraries, then the product itself. Then someone might tune the parameters of the server, define the settings of the product, and create any automated jobs required to maintain a healthy server. This process could take a long time, be prone to errors, and was not easily repeatable.

Infrastructure as code changes this manual process into an automated one. It provides a way to define a server using code - a set of instructions specifying what needs to be installed and how to configure it. Knowledge that was previously held by an individual, and not always understood by others, becomes transparent.

You can imagine the commands that were typed in by a person being replaced by a set of scripted steps. The real strength in infrastructure as code comes in its ability to scale. Imagine having to configure ten servers, or a hundred. Then imagine a move from physical servers into cloud-based computing and configuring not just hundreds of servers, but thousands. Infrastructure as code makes this possible.

As well as making configuration visible, repeatable, and scalable, infrastructure as code has other benefits. Systems are more reliable because each piece of architecture is created in an automated way. It allows for faster recovery from failure as scripted configuration can be used to re-create or alter environments quickly. Environment changes are less risky as new configuration can be tested using a production-like setup before applying that exact modification to production through a scripted deployment.

> “It also means that the developer can do it all. Whereas in the past all this orchestration and automation belonged to the sysadmin, now the developer at the very least has

> to advise, but in many cases fully take over the process. Making the relationship to application code and the infrastructure it runs on even tighter, almost indistinguishable. Now the developer can not only write the applications code, they can write the infrastructure that the code runs on."
>
> Meet Infrastructure as Code[95]
>
> *Chris Riley*

Beyond the scripts themselves, infrastructure as code introduces development practices like version control into the operations space. Using version control means that all configuration is stored together in one place.

> "The ability to use version control on your infrastructure code implies that you can easily track all the changes in your infrastructure environment. Therefore, you have an easier option of rolling back configuration changes to a previous working configuration in case of a problem."
>
> Infrastructure as Code: A Reason to Smile[96]
>
> *Jafari Sitakange*

Configuration management

Configuration management refers to the process of systematically handling changes to a system in a way that it maintains integrity over time[97]. Infrastructure as code is a key component of configuration management, but the latter term encompasses a wider set of activities.

> "Configuration Management (CM) ensures that the current design and build state of the system is known, good & trusted; and doesn't rely on the tacit knowledge of the development team."
>
> 5 things about configuration management your boss needs to know[98]

Puppet, Chef, Ansible and SaltStack are popular configuration management tools. Each tool has its own advantages and disadvantages, but their underlying purpose is similar:

> "Puppet, Chef, Ansible and SaltStack present different paths to achieve a common goal of managing large-scale server infrastructure efficiently, with minimal input from developers and sysadmins. All four configuration management tools are designed to

[95] https://devops.com/meet-infrastructure-code/
[96] https://www.thoughtworks.com/insights/blog/infrastructure-code-reason-smile
[97] https://www.digitalocean.com/community/tutorials/an-introduction-to-configuration-management
[98] https://www.upguard.com/blog/5-configuration-management-boss

> reduce the complexity of configuring distributed infrastructure resources, enabling speed, and ensuring reliability and compliance."
>
> Puppet vs. Chef vs. Ansible vs. Saltstack[99]
>
> *Ali Raza*

Wikipedia has an up-to-date comparison of notable free and open source configuration management software[100] with links to further reading.

Installing Apache: A comparison of Puppet, Chef, Ansible, and SaltStack

Apache is an extremely popular open-source web server. To illustrate the concepts of infrastructure as code and configuration management, this section shares some basic information about how to install apache via each of the top four configuration management tools: Puppet, Chef, Ansible, and SaltStack.

Configuration management could be owned by any member of the development team. These specific examples are to help understanding and introduce some key terminology in each toolset.

Puppet

A Puppet script is called a manifest. Manifests are a set of declarations written in Ruby syntax and saved with the .pp file extension. A collection of manifests, along with any associated files, can be grouped into a module. Modules help organise Puppet code, similar to how you might organise files into folders within a directory.

The contents of a simple Puppet manifest to install apache could read:

```
# execute 'apt-get update'
exec { 'apt-update':                    # exec resource named 'apt-update'
  command => '/usr/bin/apt-get update'  # command this resource will run
}

# install apache2 package
package { 'apache2':
  require => Exec['apt-update'],        # require 'apt-update' before installing
  ensure => installed,
}

# ensure apache2 service is running
```

[99] http://www.intigua.com/blog/puppet-vs.-chef-vs.-ansible-vs.-saltstack

[100] https://en.wikipedia.org/wiki/Comparison_of_open-source_configuration_management_software

```
service { 'apache2':
  ensure => running,
}
```

Getting Started With Puppet Code: Manifests and Modules[101] *Mitchell Anicas*

Puppet scripts are generally executed in a 'master and agent' architecture. One way to think of this is that the master issues instructions to the agent then, once the agent installation is complete, the agent becomes a new server in your architecture.

Chef

A Chef script is called a recipe. Recipes are a set of declarations written in Ruby syntax and saved with the .rb file extension. A collection of recipes, along with any associated files, can be grouped into a cookbook. There are parallels between the terminology of Puppet and Chef: a Puppet manifest is similar to a Chef recipe, and a Puppet module is similar to a Chef cookbook.

The contents of a simple Chef recipe to install apache could read:

```
name "webserver"
description "Systems that serve HTTP and HTTPS"
run_list(
  "recipe[apache2]",
  "recipe[apache2::mod_ssl]"
)
default_attributes(
  "apache" => {
    "listen" => ["*:80", "*:443"]
  }
)
```

apache2 Cookbook[102]

Chef scripts can be executed in a client-server mode, or in a stand-alone mode known as chef-solo, which sends commands via ssh.

Ansible

An Ansible script is called a playbook. Playbooks are a set of declarations written in YAML syntax and saved with a .yml file extension. Unlike the other tools, there isn't a concept to group playbooks together.

[101] https://www.digitalocean.com/community/tutorials/getting-started-with-puppet-code-manifests-and-modules
[102] https://supermarket.chef.io/cookbooks/apache2

Ansible does have modules that can be used to implement common instructions in a playbook. Modules can be written in any language - bash, C++, clojure, Python, Ruby, whatever you want is fine[103].

> Ansible ships with a number of modules (called the 'module library'). Users can also write their own modules. These modules can control system resources, like services, packages, or files (anything really), or handle executing system commands.
>
> About Modules[104]

The contents of a simple Ansible playbook to install apache could read:

```
1  ---
2  - hosts: apache
3    sudo: yes
4    tasks:
5      - name: install apache2
6        apt: name=apache2 update_cache=yes state=latest
```

How To Configure Apache Using Ansible on Ubuntu 14.04[105] *Stephen Rees-Carter*

> Ansible is installed on a single host, which can even be your local machine, and uses SSH to communicate with each remote host. This allows it to be incredibly fast at configuring new servers, as there are no prerequisite packages to be installed on each new server.
>
> How To Configure Apache Using Ansible on Ubuntu 14.04[106]
>
> *Stephen Rees-Carter*

SaltStack

SaltStack is a Python-based configuration management tool that allows users to maintain various types of modules. Modules manage all the execution and state management of the platforms that they are building. There are six types of module, each with a different purpose: execution modules, state modules, grains, rendered modules, returners, and runners.

This example uses a state module, which would be saved with an .sls file extension. SaltStack allows the user to choose the declarative language that a module is written in. This example uses YAML, which is the default.

The contents of the SaltStack module could read:

[103] http://docs.ansible.com/ansible/dev_guide/developing_modules_general.html
[104] http://docs.ansible.com/ansible/modules.html
[105] https://www.digitalocean.com/community/tutorials/how-to-configure-apache-using-ansible-on-ubuntu-14-04
[106] https://www.digitalocean.com/community/tutorials/how-to-configure-apache-using-ansible-on-ubuntu-14-04

```
apache:                 # ID declaration
  pkg:                  # state declaration
    - installed         # function declaration
```

States Tutorial, Part 1 - Basic Usage[107]

SaltStack executes modules through a ‘master and minion’ server relationship, which is the SaltStack terminology for a client-server architecture.

[107] https://docs.saltstack.com/en/latest/topics/tutorials/states_pt1.html

Containers

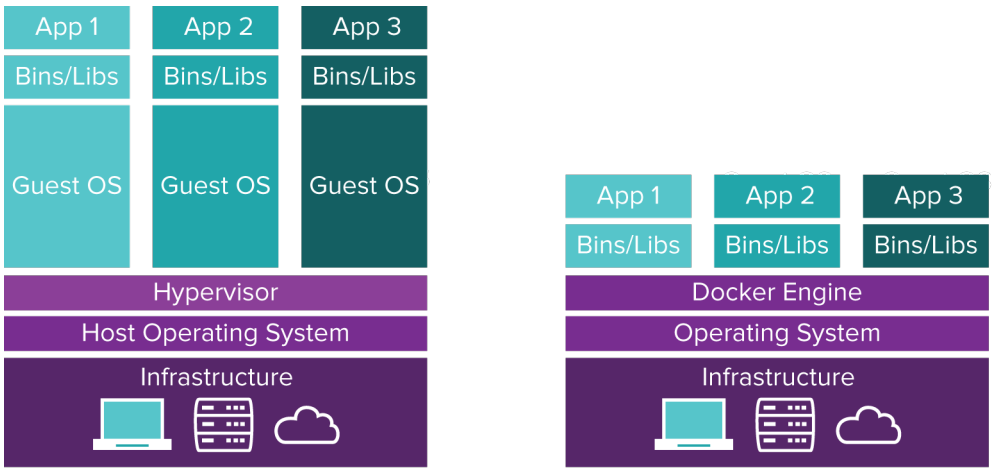
Containers are a recent type of virtualisation, a practice that has been part of system administration since the early 1960s. To understand containers it is important to know a little about the types of virtual machines.

A system virtual machine (VM) is a way of running multiple parallel machines on the same physical hardware. These could be running different operating systems or multiple instances of the same operating system. The easiest way to understand this is at the level of a personal computer. Imagine a tester with an Apple laptop who needs to test that their product works on Windows. Rather than locating a colleague with a Windows machine, they could install a VM to run Windows. The physical hardware doesn't change, but now a single laptop can run two different operating systems along with the software that is specific to each.

A process virtual machine allows any type of computer to run a specific program. The most well-known example is the Java Virtual Machine (JVM) for universal execution of Java applications. If you are running Java, you will have a JVM as part of this installation.

Another variant of virtualisation has emerged within the last 20 years. Operating-system-level virtualisation is a server virtualization method in which the kernel of an operating system allows the existence of multiple isolated user-space instances, instead of just one[108]. This type of virtualisation is widely known as containers, with Docker being the most popular tool to create and administer containers.

The simplest way to see the difference between a traditional system VM and a container is through a diagram:



> "Virtual machines (VMs) are an abstraction of physical hardware turning one server into many servers. The hypervisor allows multiple VMs to run on a single machine.

[108] https://en.wikipedia.org/wiki/Operating-system-level_virtualization

> Each VM includes a full copy of an operating system, one or more apps, necessary binaries and libraries - taking up tens of GBs. VMs can also be slow to boot.
>
> Containers are an abstraction at the app layer that packages code and dependencies together. Multiple containers can run on the same machine and share the OS kernel with other containers, each running as isolated processes in user space. Containers take up less space than VMs (container images are typically tens of MBs in size), and start almost instantly."
>
> What is a Container[109]

Docker was only launched in 2013, but it has been adopted with remarkable enthusiasm. This is partly driven by its close alignment to the DevOps movement. Docker gives development teams access to an unprecedented number of consistent environments, while allowing the operations team to manage integrity of system configuration. Docker also integrates with many popular DevOps tools.

> "Docker has been designed in a way that it can be incorporated into most DevOps applications, including Puppet, Chef, Vagrant, and Ansible, or it can be used on its own to manage development environments. The primary selling point is that it simplifies many of the tasks typically done by these other applications. Specifically, Docker makes it possible to set up local development environments that are exactly like a live server, run multiple development environments from the same host that each have unique software, operating systems, and configurations, test projects on new or different servers, and allow anyone to work on the same project with the exact same settings, regardless of the local host environment. Finally, Docker can eliminate the need for a development team to have the same versions of everything installed on their local machine."
>
> Docker through hype[110]
>
> *Ben Lloyd Pearson*

[109] https://www.docker.com/what-container
[110] https://opensource.com/business/14/7/docker-through-hype

Cloud

Cloud computing shares hardware between a number of users through virtualisation and the internet. There are different types of cloud that an organisation might choose to use: public, private, or hybrid.

A public cloud is a collection of servers that are maintained by an external company. A well-known public cloud provider is Amazon Web Services (AWS). They offer on-demand cloud computing services from data centres that are located in 16 different geographical regions. Most public cloud providers will incorporate redundancy at multiple layers of their infrastructure - network, storage, and geography - which makes them a robust choice for platform services.

A private cloud is maintained within an organisation. Large enterprises may have their own data centres that they can provision as a private cloud. The organisation retains total control of their infrastructure and the data that it contains. This is generally seen as positive from a security perspective, but may come with a high cost.

A hybrid cloud is a solution that combines both public and private. In an organisation that delivers multiple products, some may be suited to a public cloud while others may not. For example, a bank could place their public-facing website on a public cloud and their authenticated online banking channels on a private cloud. The authenticated channels require a higher level of security and control, which may drive a decision to keep hosting in-house.

How does cloud computing relate to DevOps?

The configuration management tools that can install and configure infrastructure as code - Puppet, Chef, Ansible, SaltStack - can be used in cloud environments. Cloud offers a lot of flexibility in storage and services; it is easier to scale up and down with cloud than when provisioning against local physical servers. Cloud is also a popular choice with IT management:

> "Cloud is a direct response to the need for agility. Initially, people saw cloud primarily as a way to save money and transfer CapEx to OpEx. They've since come to realize that its real value lies in reducing waste that impedes speed and defocuses effort. Very few companies would identify data center operations as part of their core value proposition. Cloud services let IT departments shift their focus away from commodity work such as provisioning hardware or patching operating systems, and spend it instead on adding business-specific value."
>
> Cloud and DevOps: A marriage made in heaven[111]
>
> *Jeff Sussna*

[111] https://www.infoq.com/articles/cloud-and-devops

Test practices for environments

Any member of a development team can execute an automated, repeatable process for creating and configuring an environment. Where the environment footprint is lightweight, maybe through the use of containers or cloud-based components, they can repeat the process on their local machine and create their own copy of production. It becomes possible for every team member to have an identical yet distinct environment on which they can perform their work.

Environments become a commodity. In a popular DevOps analogy, each server becomes part of a herd rather than a pet:

> "If you view a server (whether metal, virtualized, or containerized) as inherently something that can be destroyed and replaced at any time, then it's a member of the herd. If, however, you view a server (or a pair of servers attempting to appear as a single unit) as indispensable, then it's a pet."
>
> The History of Pets vs Cattle and How to Use the Analogy Properly[112]
>
> *Randy Bias*

The ease of creating and destroying environments completely shifts the mentality around their use. This section explains three key areas where environments are driving change for testing:

- managing the environments that are available for testing
- testing whether the underlying infrastructure is reliable
- destructive testing.

Environment management

The path to production is often paved by multiple test environments including development, test, integration, stress, UAT, and staging. Each environment has a different set of users who interact with it for their own purpose.

Environments crafted by hand are usually dedicated and persistent. They are built and then remain, with different teams negotiating a schedule for their use. There are often differences among hand-crafted servers. They might be connected to different third party systems, use different networking configuration, or have a different amount of disk space.

[112] http://cloudscaling.com/blog/cloud-computing/the-history-of-pets-vs-cattle/

If servers are a commodity, it may not be necessary to have dedicated test environments. Instead, different teams can create and destroy their own environments as required. This can enable many people to work in parallel if the organisation has sufficient capacity in either physical hardware or cloud storage to accommodate a large number of virtual environments.

Test automation can be run in a freshly created environment, that can be removed after test execution. Where tests fail, in addition to capturing screenshots and snippets of log files, you can snapshot a complete record of the state of the environment that may help identify the problem.

Problems in the product that are caused by inconsistent environment configuration are eliminated, relegating the phrase "it works on my machine" to history. Every environment should be identical.

The introduction of more environments can have a negative impact. As more environments are introduced, feedback from testing activities can be more difficult to manage. The development team will need to be sure that the environment where a problem is discovered has not been modified. They need to trust that the platform on which the problems are discovered is reflective of the ultimate destination for the code: the production environment.

Because of this, Dave Nolan is an advocate for *reducing* the number of test environments in the development process. He argues that code should pass straight from the development environment to the production environment. In his talk at a London Continuous Delivery event in 2015 he spoke about four activities that people claim require a staging environment, then refuted each with practices that could enable this activity to occur in production.

Reason for staging	**Practices to adopt in production instead of a using a separate environment**
Testing sign-off	Implement code behind a feature flag then test in production
Data integrity	Use scripts to filter, partition, and clean production data so that test data doesn't pollute customer experience
Business confidence	Improve logging and monitoring in production so that correct behaviour is evident on release
Integration	Implement change with exposure control practices to mitigate integration risk

Ref: Say "NO!" to a Staging environment[113] *Dave Nolan*

The number of environments in use might also have an impact on test data management. If the number of environments increases, can every platform provide consistent test data? If the number of environments decreases, can multiple people use these environments for testing without impacting each other?

[113] https://vimeo.com/148161104

Infrastructure testing

When infrastructure becomes code, it can have the same types of test automation as the product itself. Static analysis, unit tests, and acceptance tests can all be applied to infrastructure. In his talk at QCon London, Matt Long gave the following examples of different types of test in an infrastructure context:

> "a unit test can assert that deployment scripts work as expected, an integration test can validate whether operating system services are running correctly, and a functional acceptance test could assert that a system user can SSH into a provisioning instance and perform an arbitrary operation."
>
> Is it Possible to Test Programmable Infrastructure? Matt Long at QCon London Made the Case for "Yes"[114]
>
> *Daniel Bryant*

While it's possible to perform different types of testing against infrastructure, you may not want to. In the same presentation, Matt Long spoke of the challenges of unit testing infrastructure as code. He said "this is often difficult due to lack of tooling, and also typically offers a poor return on investment; as much as the code is declarative, the tests simply echo back that declaration.[115]"

When looking at testing with a broader scope, there are many tools that can help verify that infrastructure has been installed and configured correctly. Test Kitchen is a popular choice as a framework that brings together all these moving parts:

> "Test Kitchen is a tool for automated testing of configuration management code. It automates the process of spinning up test VMs, running your configuration management tool against those VMs, executing verification tests against those VMs, and then tearing down the test VMs.
>
> What's makes Test Kitchen so powerful and useful is its modular design:
>
> - Test Kitchen supports provisioning VMs in a variety of infrastructure environments, including Vagrant, VMware vSphere, Amazon EC2, Docker, and more.
> - Test Kitchen supports a variety of provisioners, including Chef, Puppet, Ansible, and Salt.
> - Through its Busser framework, Test Kitchen supports a variety of test execution engines, including ServerSpec, RSpec, Bash Automated Testing System (BATS), Cucumber, and more. You can write tests in whatever language suits your needs or skill-set!"

[114] https://www.infoq.com/news/2017/03/testing-infrastructure
[115] https://www.infoq.com/news/2017/03/testing-infrastructure

> Testing Ansible Roles with Test Kitchen[116]
>
> *Dan Tehranian*

The most popular test execution engine is currently ServerSpec. Matt Long has spoken about this tool, saying:

> "Integration testing can be conducted with ServerSpec, a Ruby/RSpec-based infrastructure testing framework, which can SSH into compute instances and assert that OS packages are installed, services are running, and ports are listening. Long stated that ServerSpec code is often very readable, by sysadmins and testers alike, and the community associated with the framework is large and helpful."
>
> Is it Possible to Test Programmable Infrastructure? Matt Long at QCon London Made the Case for "Yes"[117]
>
> *Daniel Bryant*

Infrastructure testing is a relatively new area. Many of the tools mentioned have emerged within the last five years. As well as fragmentation in tools, there are differing opinions in approach. Just as 'monitoring as testing' is challenging the scope of test automation required pre-release, ongoing monitoring of infrastructure can supplement automated infrastructure testing.

In the example above, ServerSpec is used to check that services are running and ports are listening. These are classic examples of monitors that can be configured against a platform. There is limited value in repeating the same check in two places: if you are already monitoring something, you probably won't gain anything from testing the same thing with your automated infrastructure test.

You could create a dashboard to show the number of environments and their status. Information displayed that could come from testing or monitoring includes:

- status of system installation including OS and basic packages
- status of core system processes
- status of interfaces to other systems
- historical trends for environment health

[116] https://dantehranian.wordpress.com/2015/06/18/testing-ansible-roles-with-test-kitchen/
[117] https://www.infoq.com/news/2017/03/testing-infrastructure

Destructive testing

Once environments are no longer precious, destructive testing becomes easier. This testing might include failover in an architecture with redundancy, recovery after failure of a service, or backup and restore of a file system. Kill a network interface, stop database processes, flood a web services port, or restart the environment unexpectedly. How does your product cope with events like these? Security testers can perform malicious test activities. Performance testers can explore the resilience of the system under load.

These activities can be contained, both from a platform and a people perspective. You can explore a variety of behaviours where aspects of the environment fail without consequences. This type of testing can be difficult where environments are static, shared by multiple people, or configured by hand.

Netflix pioneered intentional introduction of failure to improve system resilience. They developed a series of tools, branded the Netflix Simian Army, to increase the regularity of different types of infrastructure failures in their environments. This encouraged development teams to remain mindful of these failure conditions and implement solutions that were resilient to them. In their words:

> "We have found that the best defense against major unexpected failures is to fail often."
>
> Chaos Monkey Released Into The Wild[118]
>
> *Cory Bennett & Ariel Tseitlin*

First introduced in 2011, the Netflix Simian Army introduced a new lexicon and toolset for simulating a variety of infrastructure failures specific to a cloud-based AWS infrastructure:

- **Chaos Monkey** randomly disables production instances.
- **Latency Monkey** induces artificial delays in RESTful client-server communication layer to simulate service degradation and measures if upstream services respond appropriately.
- **Conformity Monkey** finds instances that don't adhere to best-practices and shuts them down.
- **Doctor Monkey** taps into health checks that run on each instance as well as monitors other external signs of health (e.g. CPU load) to detect and remove unhealthy instances.
- **Janitor Monkey** searches for unused resources and disposes of them.
- **Security Monkey** finds security violations or vulnerabilities, such as improperly configured AWS security groups, and terminates the offending instances. It also ensures that all our SSL and DRM certificates are valid and are not coming up for renewal.

[118] http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html

- **10-18 Monkey** (short for Localization-Internationalization, or l10n-i18n) detects configuration and run time problems in instances serving customers in multiple geographic regions, using different languages and character sets.
- **Chaos Gorilla** similar to Chaos Monkey, but simulates an outage of an entire Amazon availability zone.

Netflix Simian Army[119]

Yury Izrailevsky & Ariel Tseitlin

Netflix released the code for the Simian Army as an open source github project[120], which opened its development to the wider software community. The full set of tools is still in active development with over 45 contributors to the repository in its lifetime. In addition a newer, separate version of Chaos Monkey[121] has been released.

The Simian Army is run in the Netflix production environment during business hours, which allows staff to take immediate action to resolve problems that occur:

> "Netflix lets the Simian Army roam free causing random chaos, but only Monday through Friday between the hours of 9am and 3pm when managers and developers are present to address any urgent situations that might be caused inadvertently. In some cases the tools take automated corrective action, and in some they simply generate an alert and escalate the issue to the appropriate group or individual."
>
> Netflix, the Simian Army, and the culture of freedom and responsibility[122]
>
> *Tony Bradley*

These same tools also run regularly in the development environments. One of the merits of cloud-based infrastructure is the ability to create development environments that mimic the production architecture. The constant instability of infrastructure helps to create better quality code:

> "Thus, while writing code, Netflix developers are constantly operating in an environment of unreliable services and unexpected outages. This chaos not only gives developers a unique opportunity to test their software in unexpected failure conditions, but incentivizes them to build fault-tolerant systems to make their day-to-day job as developers less frustrating. This is DevOps at its finest: altering the development process and using automation to set up a system where the behavioral economics favors producing a desirable level of software quality. In response to creating software in this type of environment, Netflix developers will design their systems to be modular, testable, and highly resilient against back-end service outages from the start."

[119] http://techblog.netflix.com/2011/07/netflix-simian-army.html
[120] https://github.com/Netflix/SimianArmy
[121] https://github.com/Netflix/chaosmonkey
[122] https://devops.com/netflix-the-simian-army-and-the-culture-of-freedom-and-responsibility/

> DevOps Case Study: Netflix and the Chaos Monkey[123]
>
> *Aaron Cois*

Netflix operates on a scale that is difficult to emulate in other organisations. However the idea of introducing failure has propagated, albeit on a smaller scale. From DevOps Days in London in 2013:

> "While most companies aren't able to inject failures on a scale that Netflix does, that doesn't mean that others aren't embracing this approach. There was a presentation on how Pager Duty has "Failure Friday", where they introduce failures into their system. This allows them to test not only how resilient their software is but also how good their monitoring, logging and processes are."
>
> Failure is inevitable but you don't need to be afraid[124]
>
> *Jonathan Thorpe*

Failure Friday is explained in more detail as an industry example in the following section.

[123] https://insights.sei.cmu.edu/devops/2015/04/devops-case-study-netflix-and-the-chaos-monkey.html
[124] https://blog.microfocus.com/failure-is-inevitable-but-you-dont-need-to-be-afraid/

INDUSTRY EXAMPLES

This section shares experiences from a variety of organisations to demonstrate how the practices and ideas from previous chapters are being applied within the industry.

Each example begins with an overview of the organisation that includes their purpose, products, and scale. This is followed information about their release process and cadence where this is available. Finally, there is a detailed description of the relevant DevOps processes the organisation has put in place.

King: AI in testing

King is a social games company. They have over 200 game titles, developed and supported by over 1,800 employees who are spread across 13 geographical locations[125]. Their games cover web, native mobile applications (iOS, Android, Windows Phone), Facebook, and Windows 10. King were acquired by Blizzard in early 2016.

This industry example focuses on testing within the Candy Crush Saga suite of games. In 2013, at the height of its popularity, Candy Crush Saga was played 700 million times per day on smartphones and tablets[126].

Release information

Release cycle	Every 2 weeks
Automated release process	No
Duration of release process	3 days
Release branch	Release candidate

Candy Crush Soda Saga delivery pipeline[127] *Yassal Sundman*

Test automation with an AI bot

The Candy Crush games that King create are level-based, with updates to the game delivering additional levels. Though it is important for King to regularly release new content, it's more important for them to release quality content. King employee Alexander Andelkovic explains that people usually don't go back to play a level that they've already passed so "getting it right the first time is better for the player experience"[128].

Their focus isn't on pushing testing to the right, into their production system. Instead they work to improve what is tested prior to release, and investigate how information from production can help them do this.

Each level of Candy Crush has a success rate based on player data. The Business Performance Unit (BPU) at King analyse the success rate of all players in all levels, and make recommendations to the level designers based on their observations. Level designers use this information to set and tweak the difficulty of their levels so that they are not too hard or too easy.

The feedback loop for a new level of Candy Crush has historically been slow. When designers are creating a new level, the testers play it, creating success rate data for the BPU to analyse. The level

[125] https://www.youtube.com/watch?v=wHlD99vDy0s
[126] https://www.theguardian.com/technology/appsblog/2013/sep/10/candy-crush-saga-king-interview
[127] https://techblog.king.com/candy-crush-soda-saga-delivery-pipeline/
[128] https://www.youtube.com/watch?v=wHlD99vDy0s

is released after testing, but the tester's success rate isn't always reflective of the success rate for players in production. Occasionally a new level requires a second release to tweak its difficulty.

Alexander Andelkovic spoke at CAST2016[129] about how King have started to shorten this feedback loop by incorporating artificial intelligence into their test automation. They've developed a bot that uses an advanced neural network to perform strategic gameplay in the same manner as a real Candy Crush player. Their bot has iterated from a simple success test, to a Monte Carlo tree search, to a Neuro Evolution of Augmented Topologies (NEAT) algorithm. This builds an artificial neural network to successfully complete each level.

The results have been encouraging. The success rate for finishing a level using the bot is very close to the success rate of human players in production. The more iterations that the bot completes to learn about the level, the more accurate it becomes. The ability of the automated bot to emulate strategic thinking gives direct feedback to help the level designers tweak the difficulty of the level. The testers no longer have to manually test each version of the level to generate their own success rate data. The BPU are no longer required for analysis and recommendations, as the bot provides this automatically.

In addition to validating new levels, the bot can verify existing ones. The 2000+ existing levels of Candy Crush can be regression tested to ensure that their success rate remains consistent. The BPU continue to provide success rate information from production to keep the bot calibrated correctly.

This is a great illustration of how development and operations teams can work together to use production analytics to improve pre-release testing.

[129] https://www.youtube.com/watch?v=wHlD99vDy0s

Capital One: Hygieia delivery pipeline dashboard

Capital One is an American bank, a Fortune 500 company, and one of the nation's top 10 largest banks based on deposits[130]. They are most widely known for their credit card products, which were the origin of their business, but now operate with a diversified financial offering.

Capital One was founded in 1988, which makes them unusual among enterprise financial institutions. Topo Pal, a Director at Capital One, says "if you look at our nearest competitors, they are more than 100 years old, so we are kind of a start up in this industry"[131].

Hygieia dashboard for a delivery pipeline

Rob Alexander, the CIO of Capital One, sees an automated delivery pipeline as a key facet of their DevOps strategy:

> "We think that the move to DevOps is about automation of how we build software, from being able to develop software and push it, all the way to deployment in an automated way with all the testing and security and other things you need to do all along the way. That whole automated delivery pipeline is an important part of it."
>
> How Capital One became a leading Digital bank[132]
>
> *Peter High*

The engineering team at Capital One have taken that vision and created Hygieia, named after the Greek goddess of health. Hygieia is a single, configurable, easy to use dashboard to visualize near real-time status of the entire delivery pipeline[133].

The primary purpose of Hygieia is to aggregate. It doesn't introduce new data into the delivery pipeline, but rather pulls together information from the toolset that the organisation already uses. It includes information across the breadth of the development process: from code commit to production deploy. As Topo Pal states in an interview:

> "With the Hygieia dashboard, we were able to increase transparency in the pipeline. Now, everyone involved can see (a) what stories the teams are working on, how many are in progress, how many are done (b) what codes are being committed, how frequently, and who is committing code (c) how many builds are taking place, are builds failing, what broke the build etc. (d) static code analysis result of the code, security scan results of the code, unit test case results and test coverage, functional

[130] https://www.capitalone.com/about/

[131] https://www.youtube.com/watch?v=6Q0mtVnnthQ

[132] http://www.forbes.com/sites/peterhigh/2016/12/12/how-capital-one-became-a-leading-digital-bank/

[133] https://github.com/capitalone/Hygieia

> test case results etc. (e) various environments and code deployment statuses in those environments. (f) a deployment pipeline showing what version of software is deployed in which environment and what failed."
>
> DevOps Dashboard Hygieia Aggregates End-to-End View of the Delivery Pipeline[134]
>
> *Grischa Ekart*

A transparent, aggregated data set allows the development team to make collaborative decisions about improvements to their process. The goal at Capital One is improving the speed through the pipeline. Topo Pal explains how Hygieia enables this:

> "... you can actually see the stoppage time. So our theory is, instead of trying to understand how to speed up, try to reduce the wait time, so that we can increase the speed finally.
>
> Sometimes I have seen developers working hard to decrease their build time from 25 minutes to 15 minutes, where the test cases run for one hour. So do you spend the energy and time to decrease the build time or .... to speed up testing?"
>
> DOES16 San Francisco - DevOps at Capital One: Focusing on Pipeline and Measurement[135]
>
> *Topo Pal*

The pipeline includes build, test and deploy activities, any of which may be targets for speed improvements. From a testing perspective, the visibility of test automation within the wider development process is interesting. The organisation can understand what proportion of the pipeline is spent in testing activities, periodically question the relevance and necessity of those activities, and be comfortable in articulating the value of tests that are executed.

Hygieia is open source[136], under active development with multiple contributors.

> "I've read stories of banks embracing DevOps and leveraging open source code, but Hygieia is a concrete example of the change that's occurring in the most traditional of enterprise IT."
>
> A DevOps dashboard for all: Capital One's Hygieia project offers powerful open source resource[137]
>
> *Keith Townsend*

[134] https://www.infoq.com/news/2016/03/hygieia
[135] https://www.youtube.com/watch?v=6Q0mtVnnthQ
[136] https://github.com/capitalone/Hygieia
[137] http://www.techrepublic.com/article/a-devops-dashboard-for-all-capital-ones-hygieia-project-offers-powerful-open-source-resource/

The Guardian: Testing in production

The Guardian is a British national daily newspaper founded in 1821. In parallel to traditional print media, the newspaper runs three online editions: Guardian UK, Guardian US, and Guardian Australia.

This industry example focuses on testing of the applications that support their online content including their website, mobile apps, APIs, and editorial tools. At the end of 2016, The Guardian had approximately 155m unique monthly online readers[138]. The online content is designed to work in all situations[139], which means any browser on any operating system on any device.

Release information

Release cycle	Hundreds of releases daily
Automated release process	Yes
Duration of release process	Minutes
Release branch	Master

Shifting test automation to production

In mid-2016 Sally Goble, the Head of QA at the Guardian, opened a presentation by stating 9 million visitors a day, 400 releases a day, 4 automated tests across all products[140].

Four automated tests. Really? The answer is both yes and no.

Within the last decade, the Guardian deployment process has improved significantly:

> "In 2011 the Guardian deployed the software behind the website 25 times. It was a laborious and fairly manual process. In 2014 we made more than 24,000 highly automated production deployments."
>
> Delivering Continuous Delivery, continuously[141]
>
> *Phil Wills & Simon Hildrew*

This shift in pace created a huge amount of pressure on testing. The QA team had to adapt their approach and they ultimately chose to test less during development. Sally realised that it wasn't necessary for the software to be perfect and bug free[142] at the moment that it was released to the readers. "The risk of not getting software to our users is greater"[143]

[138] http://www.economist.com/news/business/21703264-newspaper-may-be-edging-towards-asking-readers-pay-its-content-guardians-losses
[139] https://www.theguardian.com/info/2015/sep/22/making-theguardiancom-work-best-for-you
[140] https://speakerdeck.com/sallygoble/what-really-happens-when-you-deliver-software-quickly-1
[141] https://www.theguardian.com/info/developer-blog/2015/jan/05/delivering-continuous-delivery-continuously
[142] https://www.theguardian.com/info/developer-blog/2016/dec/04/perfect-software-the-enemy-of-rapid-deployment
[143] https://vimeo.com/162635477

The Guardian team thought critically about what risks testing would traditionally address during development, then sought to find new ways to alleviate these later in the lifecycle. Most of the new ideas focused on quick identification of problems in production, as the ability to do rapid deploy and rollback reduced the impact of errors reaching production.

This meant moving test automation from pre-release to post-release. New automation has been created to run alongside operations:

> "Instead of focusing on lengthy validation in staging environments, our continuous deployment pipeline places greater emphasis on ensuring that new builds are really working in production. We believe that developers should derive confidence from knowing that their code has run successfully in the real world, rather than from observing green test cases in a sanitised and potentially unrepresentative environment. Therefore we minimised the amount of testing run pre-deployment, and extended our deployment pipeline to include feedback on tests run against the production site."
>
> Testing in Production: rethinking the conventional deployment pipeline[144]
>
> *Jacob Winch*

The Guardian implemented their production tests using prout, an in-house tool that answers the question "is your pull request out?"[145]. It is integrated with source control, which means that developers are informed that their changes are working in production within minutes of merging to master[146].

Tests that run in production can make use of different types of feedback: analytics events, monitoring, alerts, logging, and customer feedback. In one particular product, the QA team recognised the potential and decided to expand the scope of their production-facing test automation:

> "While this is a solid approach, in reality, a lot of the issues that are seen in production do not manifest on the initial deploy of the system, they occur when that particular build has been in production for some time. So rather than triggering a test run after a new release, then calling it a day, we wanted to continue to test on the production system. If we do this issues that manifest slowly, or are intermittent are detected as quickly as possible."
>
> Testing in Production: How we combined tests with monitoring[147]
>
> *Jonathan Hare-Winton & Sam Cutler*

They developed another in-house tool called Prodmon that combines testing in production and monitoring. Prodmon regularly runs a set of tests that checks the product at two different layers: the API and the user interface:

[144] https://www.theguardian.com/info/developer-blog/2016/dec/20/testing-in-production-rethinking-the-conventional-deployment-pipeline
[145] https://www.theguardian.com/info/developer-blog/2015/feb/03/prout-is-your-pull-request-out
[146] https://www.theguardian.com/info/developer-blog/2016/dec/20/testing-in-production-rethinking-the-conventional-deployment-pipeline
[147] https://www.theguardian.com/info/developer-blog/2016/dec/05/testing-in-production-how-we-combined-tests-with-monitoring

> "These behave like standard automated tests, using all the usual assertion techniques etc. Where these differ from a traditional tests however, is that they are not simply triggered by a new release. Prodmon runs continually, 24 hours per day, 7 days a week on our production systems. And rather than producing standard test reports, we alert on issues in the same way as our other monitoring systems, by triggering alarms to notify the team."
>
> Testing in Production: How we combined tests with monitoring[148]
>
> *Jonathan Hare-Winton & Sam Cutler*

The Guardian is an example of an organisation that challenges the need for test automation prior to release. Sally Goble has summarised their approach as "kill all the tests and be a company that fixes things quickly"[149]. Their aggressive release capability makes them an extremely responsive online news source - a great example of IT aligning to business drivers.

[148] https://www.theguardian.com/info/developer-blog/2016/dec/05/testing-in-production-how-we-combined-tests-with-monitoring
[149] https://vimeo.com/162635477

Bank of New Zealand: A/B testing

Bank of New Zealand (BNZ) is one of New Zealand's largest banks and has been operating continuously in the country since the 1860s[150]. BNZ offer a variety of financial products and services to retail, business, and institutional customers. In October 2016, BNZ had approximately 4500 full-time staff and 760 part-time staff[151].

This industry example focuses on the development of the 'New Internet Banking' (New IB) platform that is delivered to retail customers. This is a web-based application that is used by personal banking customers to manage their money.

Release information

Release cycle	Monthly
Automated release process	No
Duration of release process	Up to 5 days
Release branch	Release candidate

Experimenting on home loan call back

Using the New IB product, a customer can view the balance and details of their home loan. If a customer has a fixed rate home loan nearing the end of its term, a notification appears on their account. When the customer clicks to see details of their home loan, they are encouraged to arrange a call back from BNZ so that they can secure a new fixed rate home loan.

The call-back process is intended to give customers an easy way to get in touch with the bank, so they can get the best option for their home loan. Unfortunately the success rate for arranging a call back using New IB was around 5%. Based on this data the New IB team identified this was an opportunity for improvement. They decided to experiment with the wording in the dialog that prompts the customer to enter the call-back process. They created four variants of the dialog:

A) Control - currently in production

B) Time Focus - emphasised the impending date of fixed rate expiry

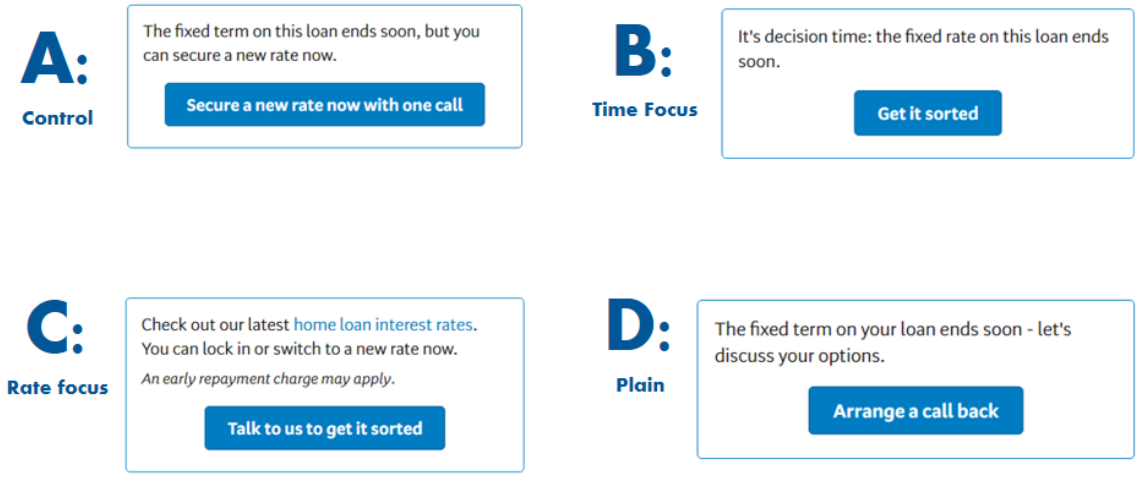
C) Rate Focus - emphasised the ability to lock in a new interest rate

D) Plain - gave a simple and clear prompt

These variants are shown below:

[150] https://www.bnzheritage.co.nz/archives/historic-timeline

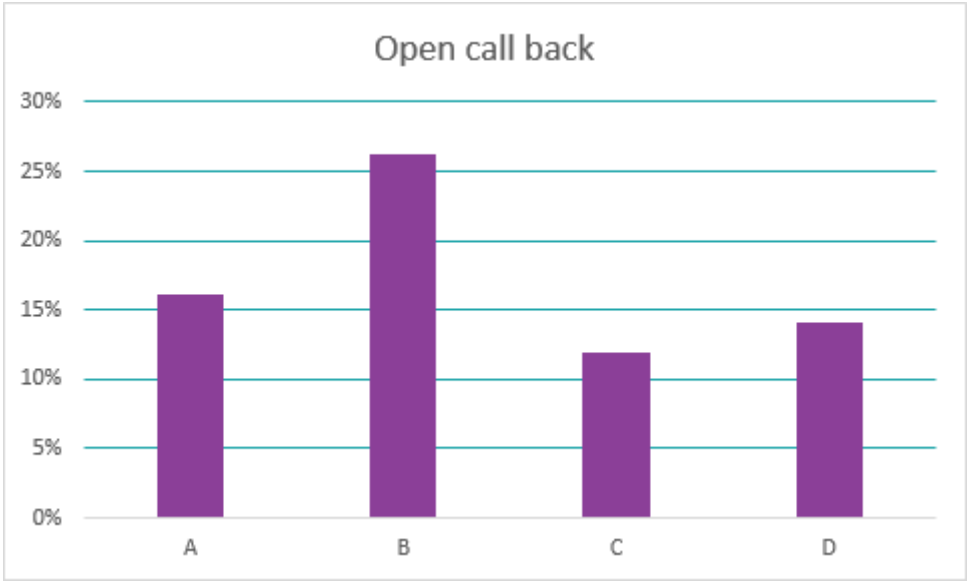
[151] http://www.stuff.co.nz/business/85563033/BNZ-head-office-staff-face-restructure



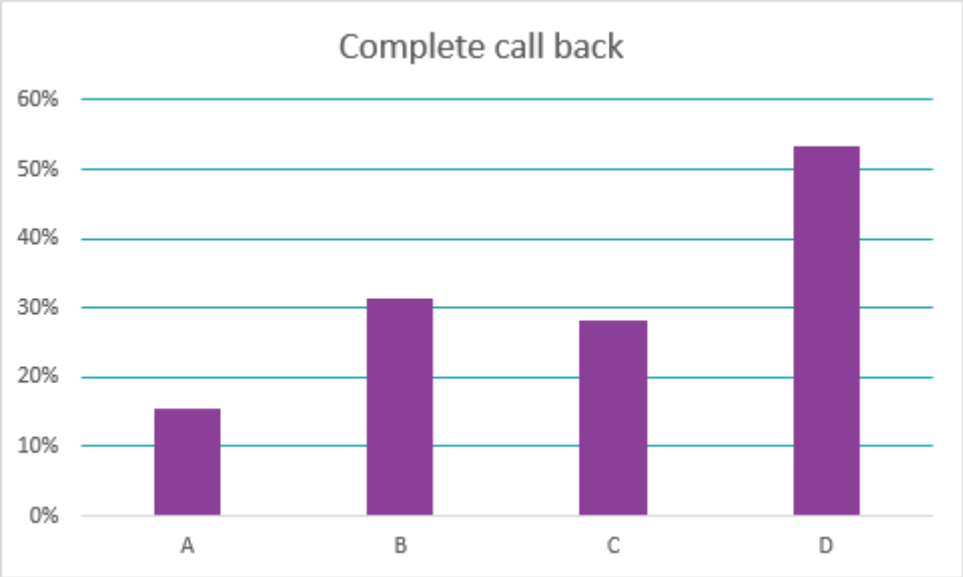
The experiment was pushed to the production platform and customer behaviour was monitored. Data was collected for two weeks and then analysed by the product owner, Luke Brown.

Luke examined two measurements; the percentage of customers who clicked the button and the percentage of customers who went on to complete the callback form.

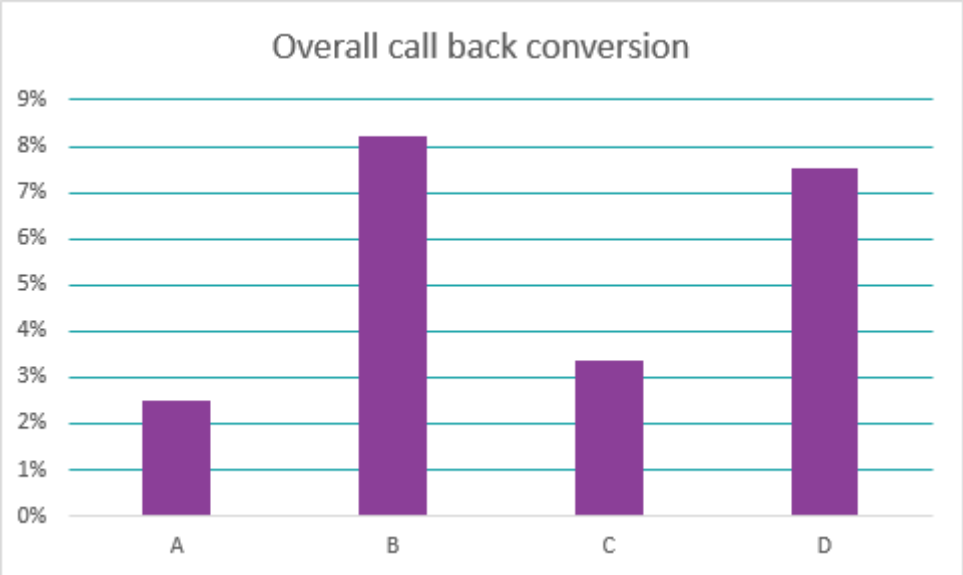
He saw that the time-focused prompt, Option B, was the winner for the first measurement, with almost double the response of other options:



The second measurement tracked the percentage of customers who went on to complete the call back form. The plain prompt, option D, was the winner for this measurement:



While Option B brought more customers into the flow, most failed to complete the call back form. By contrast, Option D attracted fewer people into the flow, but a much higher percentage arranged a call back. The overall result was very close between Option B and Option D:



Luke shared these results with the development team and discussed the outcome.

> "As a team we decided that B was a bit disingenuous (some labelled it clickbait!) because many of the customers would have clicked on 'Get it sorted' thinking they were able to refix their loan right then and there. While it did produce a slightly higher number of call backs, it would also be responsible for many more unsatisfied customers. Therefore we decided D was the winner.

> Interestingly, we just threw in Option D because we needed another variant to complete the experiment, and we thought that Option D seemed nice and dull. Our conclusion is that stating things clearly is the the way to go, so for future experiments we'll make sure at least one option simply states what's about to happen without any spin."
>
> *Luke Brown*

This example illustrates how the outcome of A/B testing can be a surprise to the team who implement different options!

Etsy: Monitoring as testing

Etsy is a marketplace where people around the world connect, both online and offline, to make, sell and buy unique goods[152]. They were founded in 2005 and have experienced rapid growth. In their 2014 IPO filing[153] Etsy said that they had 54 million members, 21.8 million app downloads, and an active marketplace with 1.4 million sellers and 19.8 million buyers in the previous year.

Release information

Release cycle	Multiple releases daily
Automated release process	Yes
Duration of release process	Minutes
Release branch	Master

Monitoring as testing

Etsy track everything: "number of logins, number of login errors, dollars of transactions through the site, bugs reported on the forum – you name it"[154]. Their monitoring solution can detect changes across hundreds of measurements.

Monitoring can tell you that something has happened, but it usually cannot tell you the root cause of the change. Mike Brittain, while a Lead Software Engineer at Etsy in 2010, used the example of a graph of PHP errors.

[152] https://www.etsy.com/nz/about/

[153] http://venturebeat.com/2015/03/04/etsy-files-to-go-public-with-100m-ipo/

[154] http://www.cio.com/article/2397663/developer/continuous-deployment-done-in-unique-fashion-at-etsy-com.html

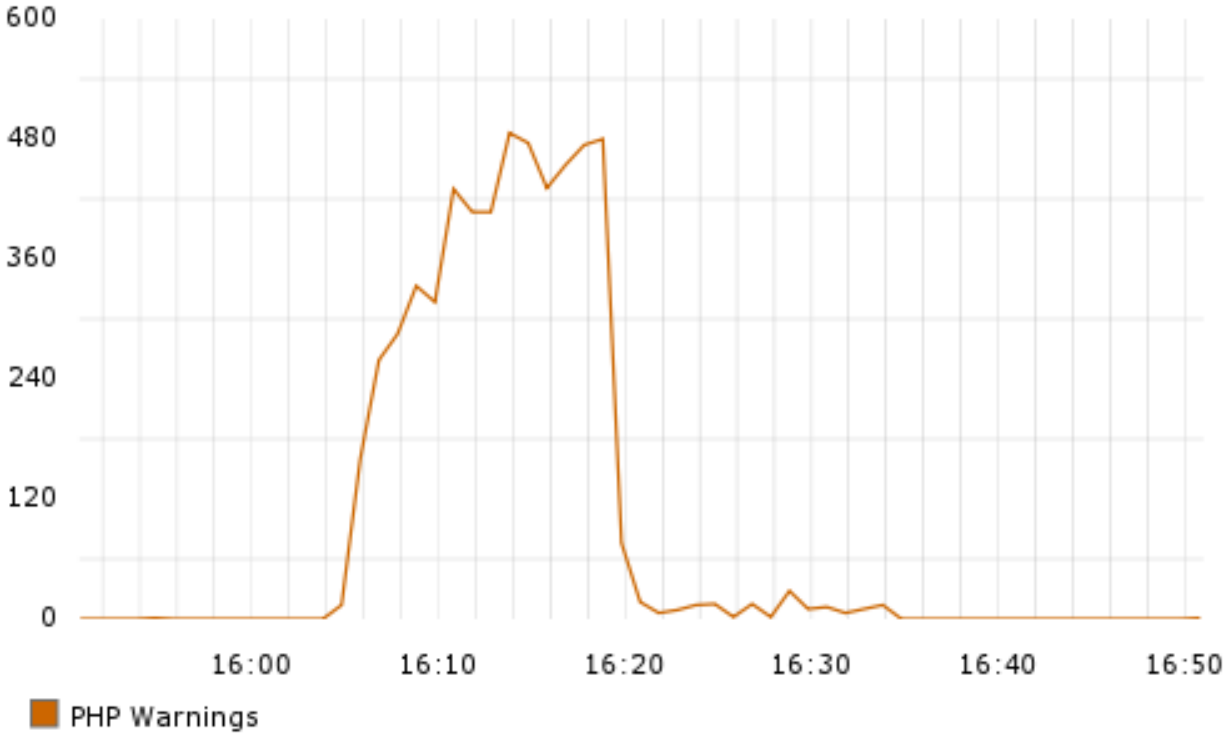


Image credit: Mike Brittain, Tracking Every Release[155]

> "Something obviously happened here… but what was it? We might correlate this sudden spike in PHP warnings with a drop in member logins or a drop in traffic on our web servers, but these point to effects and not to a root cause.
>
> At Etsy, we are releasing changes to code and application configs over 25 times a day. When the system metrics we monitor start to skew we need to be able to immediately identify whether this is a human-induced change (application code) or not (hardware failure, third-party APIs, etc.). We do this by tracking the time of every single change we ship to our production servers."
>
> Tracking Every Release[156]
>
> *Mike Brittain*

Having a vast array of monitors and tracking production deploys allows the Etsy team to quickly correlate the cause of an issue. This information is even more useful when combined into a single source. The Etsy team overlay deployment information on their monitors as vertical lines:

[155] https://codeascraft.com/2010/12/08/track-every-release/
[156] https://codeascraft.com/2010/12/08/track-every-release/

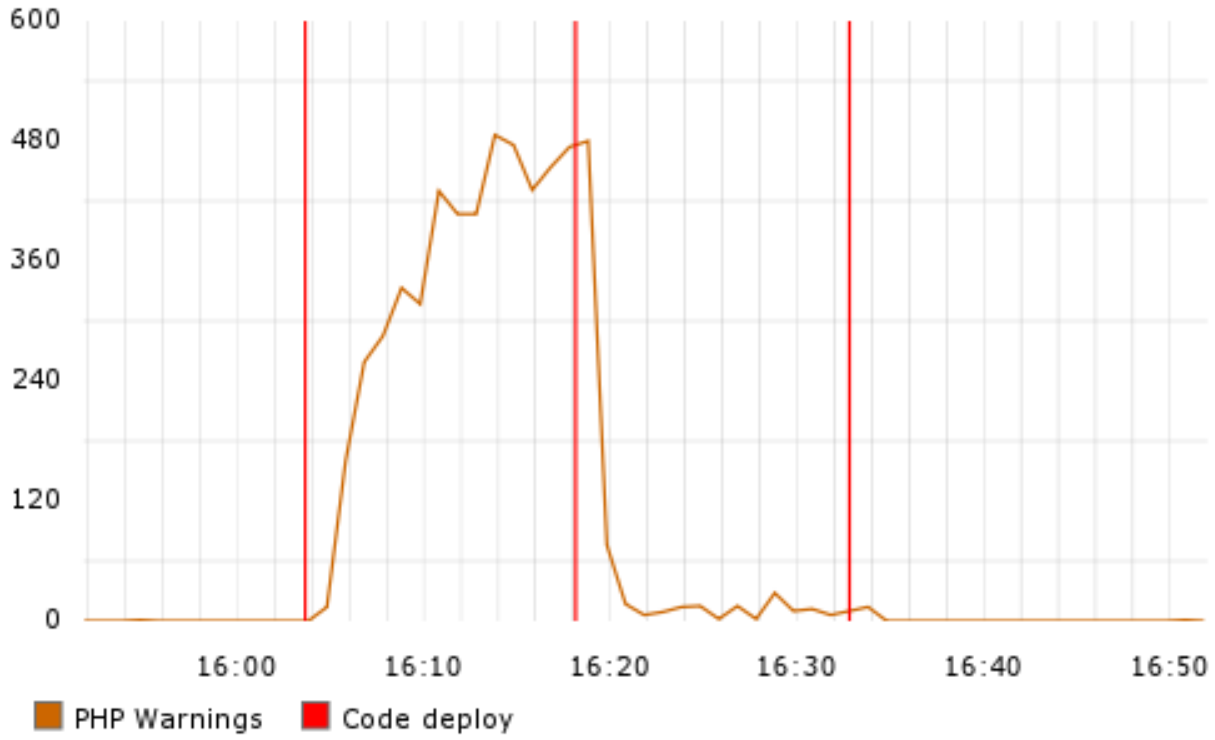


Image credit: Mike Brittain, Tracking Every Release[157]

Noah Sussman, interviewed in early 2012 while a Test Architect at Etsy, explained this approach:

> "Waving at a monitor, he continued, "We've averaged 26 deploys to production per day for the past month or so. Yeah, January is a pretty slow month. The vertical lines here are deploys. So basically, you watch the wall. If there's a sudden spike or valley immediately after a deploy, we've got a problem"."
>
> Continuous Deployment Done In Unique Fashion at Etsy.com[158]
>
> *Matthew Heusser*

That same year, Noah spoke at DevOps Day in Mountain View California. He summarised the way that Etsy use their monitoring at release with the following points:

- Etsy collects well over a quarter of a million metrics.
- Deciding which ones matter is a **human problem**.
- Everyone watches some subset of the graphs.

[157] https://codeascraft.com/2010/12/08/track-every-release/
[158] http://www.cio.com/article/2397663/developer/continuous-deployment-done-in-unique-fashion-at-etsy-com.html

- Human vision excels at anomaly detection.

How rapid release cycles alter QA and testing[159]

Noah Sussman

The monitoring solution isn't the entire test strategy, but it is an important piece. Noah summarises the wider view as:

> "We push small changesets to production dozens of times a day, hundreds of times a month. These changesets are supposed to have little-to-no impact on the behavior of the Web site. Automated tests are run before and after each deployment in order to verify that key features are indeed working as intended. This is in addition to constant, obsessive monitoring and manual testing. Finally, rapid deployment is part of our risk mitigation and resiliency strategy. It's fairly common to see bugs found and fixed within a few hours."
>
> Why does Etsy care so much about automated software testing?[160]
>
> *Noah Sussman*

[159] https://www.slideshare.net/noahsussman/continuous-improvement-devops-day-mountain-view-2012
[160] https://www.quora.com/Why-does-Etsy-care-so-much-about-automated-software-testing

Spotify: Using Docker

Spotify is a digital music service that gives you access to millions of songs[161]. Spotify began as a Swedish startup that launched in 2008. As of June 2016, Spotify has 100 million monthly active users[162], and as of March 2017, there are 50 million paying subscribers[163].

Spotify provide native applications for desktop, mobile, and e-readers. There is a web-based version of Spotify that does not require any download or installation of software. And they offer a service called Spotify Connect[164] that allows Spotify to play on external speakers, TVs, car stereos, smart watches, or any other receivers.

Implementing docker in a services architecture

Spotify were an early and prominent adopter of container technology in their production environment. Rohan Singh, an Infrastructure Engineer at Spotify, spoke about the situation that lead to the adoption of Docker at the end of 2013 saying:

> "We use a service oriented infrastructure, so we have tons of little services that each do one thing, and hopefully do it well. In reality, some services are huge and do more than one thing, and some services do things badly, but that's another story.
>
> To give you an idea, some of the services we have are a playlist service, or a search service, or a metadata service, or recommendations, or anything like that. Each one of these services is owned by a distinct squad, a group of developers, who are deploying it to a bunch of servers around the world at four different sites.
>
> We're at the point now where we have over 250 servers per Ops engineer, that's physical machines, and it's getting tough to handle."
>
> Docker at Spotify - Twitter University[165]
>
> *Rohan Singh*

Rohan went on to speak about how hardware could be poorly utilised, the difficulties in managing configuration across a vast number of physical machines, and how easy it is for machines that are meant to be identical to become inconsistent.

In a later presentation at DockerCon in 2014, Rohan talked about how Spotify adopted Docker as an alternative. Docker allows Spotify to create repeatable infrastructure - the same container can be used in development, test, and production. He claimed that it was straight-forward to create Docker files in a simple syntax. He also spoke of the ability to recover from failure:

[161] http://www.spotify.com
[162] http://www.telegraph.co.uk/technology/2016/06/20/spotify-crosses-100m-users/
[163] http://www.theverge.com/2017/3/2/14795274/spotify-streaming-50-million-paid-subscribers-milestone
[164] https://support.spotify.com/no/article/spotify-connect/
[165] https://www.youtube.com/watch?v=pts6F00GFuU

> "In terms of fault tolerance, even if your code is perfect and everything is going great, you never know when a deployment might fail. There might be a network issue. Some of the machines you're deploying to might just randomly go down. You might lose a data center that you're deploying to. We want to be able to recover from this to some degree.
>
> Docker gives us that to some extent. Instead of having to have a bunch of things work, really just one thing has to work with Docker deployments. You pull the image, you run it, it works. And if it doesn't work that's ok, you just try it again. You don't end up with a machine that's half-installed, half-configured."
>
> Docker at Spotify - DockerCon[166]
>
> *Rohan Singh*

Docker helped to improve utilisation and standardisation of hardware, but management of a large number of platforms was still a problem. Evgeny Goldin talked about this problem at DevCon in 2015 and shared how Spotify resolved it:

> "We had this problem with our code that we called NM problem. You have N services and you want to deploy them on M hosts. So how do you manage it?
>
> That's how Helios tool was born. We call it a docker orchestration framework. Basically, it solves one problem and solves it really well. Given a list of hosts and given a list of services to deploy, it just deploys them there and keeps them running. It's not trying to be over-sophisticated."
>
> CD @ Spotify: How Spotify does Continuous Delivery with Docker and Helios[167]
>
> *Evgeny Goldin*

Helios[168] is available as open-source software and is still under active development. Industry commentator Jonathan Vanian says "it seems like a lot of the ease, automation and stability of spinning up clusters of containers comes from the orchestration service that coordinates the endeavor."[169]

> "Containers and the Helios model has been used in production at Spotify since July 2014 and is a key pillar in their scalability strategy. "We're already at hundreds of machines in production," the team announced in their Helios README notes on GitHub. "But we're nowhere near the limit before the existing architecture would need to be revisited"."
>
> Containers in production: Case Studies, Part 1[170]
>
> *Mark Boyd*

---

[166] https://www.youtube.com/watch?v=Tlgoq9t95ew
[167] https://www.youtube.com/watch?v=5Ycb7jlZGkU
[168] https://github.com/spotify/helios
[169] https://gigaom.com/2015/02/22/how-spotify-is-ahead-of-the-pack-in-using-containers/
[170] https://thenewstack.io/containers-production-part-case-studies/

PagerDuty: Failure Fridays

PagerDuty provide enterprise-grade incident management[171]. Their niche has historically been orchestrating the multiple parts of an incident response. PagerDuty sits alongside the monitoring system, determining who to notify about problems, and delivering notifications.

PagerDuty have over 8000 customers across 80 countries[172]. In 2013 their fault tolerant architecture included:

> "... 3 data centers and 2 cloud providers to ensure all phone, SMS, push notifications and email alerts are delivered. Critical paths in our code have backups; and then our code has backups for the backups. This triple redundancy ensures that no one ever sleeps through an alert, misses a text message or doesn't receive an email."
>
> Failure Friday: How we ensure PagerDuty is always reliable[173]
>
> *Kenneth Rose*

Failure Friday

Due to the critical nature of their platform, PagerDuty have implemented architecture with redundancy throughout. Having multiple data centers, cloud providers, phone providers, email providers, and downstream providers allows multiple options for delivering the service.

PagerDuty's implementation means it can cope with switching between different platforms. But code for coping with failure is not executed as regularly as the rest of the application, which makes it a ripe location for undiscovered bugs.

Netflix solved this problem with the Simian Army, described in a previous chapter of this book. PagerDuty chose to adopt a different approach. Kenneth Rose of PagerDuty writes:

> "While some of these tools are freely available, they assume your application is deployed solely in AWS using auto-scaling groups, while ours is deployed across multiple providers. Not wanting to get bogged down trying to produce our own equivalent tools, we decided to invest in a straight-forward method. Simply, schedule a meeting and do it manually. We use common Linux commands for all our tests making it easy to get started."
>
> Failure Friday: How we ensure PagerDuty is always reliable[174]
>
> *Kenneth Rose*

[171] https://www.pagerduty.com/
[172] https://www.pagerduty.com/customers/
[173] https://www.pagerduty.com/blog/failure-friday-at-pagerduty/
[174] https://www.pagerduty.com/blog/failure-friday-at-pagerduty/

In the same vein as Netflix, PagerDuty decided to create failure in their production environments. Doug Barth of PagerDuty explains this by saying: "Since no test environment is ever true to production, we felt the best place to do that testing was in production itself"[175].

This approach creates risk because it may cause an unexpected problem. It's extremely important to get the business on-side, so they are aware and comfortable with the risk being introduced. Doug has explained Failure Friday to business people at PagerDuty by saying:

> "It's a lot better to respond to this failure in the middle of the day, when we're all sitting there ready to deal with it, than getting woken up in the middle of the night. The other benefit of us causing the failure is that we know exactly what we did to cause the failure, so we don't have to spend time trying to figure out why this is broken or what changed."
>
> Failure Friday | Injecting Failure at PagerDuty to Increase Reliability[176]
>
> *Doug Barth*

To prepare for Failure Friday, the team determine where they want to test that week. The type of work selected can be targeted to functionality or infrastructure. The idea is to create a negative impact on different aspects of the platform to test that there is no impact to customers when problems occur.

After selecting a task, the PagerDuty team then perform preparatory tasks that include:

- Schedule a one-hour meeting
- Notify everyone in the development team and the business of the scope of the session
- Ask the team whose service will be attacked to actively monitor their system dashboards
- Disable any cron jobs that are scheduled to run during the hour
- Disable configuration management so that it doesn't automatically revert infrastructure changes
- Create a dedicated chat channel and conference call for quick communication
- Keep PagerDuty alerts turned on to test their relevance and timeliness in relation to the failure

On the day, they introduce failure to the production environment:

> "Each attack we introduce against our service lasts five minutes. Between attacks we always bring the service back to a fully functional state and confirm everything is operating correctly before moving onto the next attack.
>
> During the attack, we check our dashboards to understand which metrics point to the issue and how that issue affects other systems. We also keep a note in our chat room of when we were paged and how helpful that page was.

[175] https://www.youtube.com/watch?v=KtX_Bx7HQsA
[176] https://www.youtube.com/watch?v=KtX_Bx7HQsA

> For each attack, we like to first start with attacking a single host. If that attack behaves as expected, we repeat the test against an entire data center. For services hosted only on AWS, we test that a service survives losing an entire availability zone."
>
> Failure Friday: How we ensure PagerDuty is always reliable[177]
>
> *Kenneth Rose*

This approach works well for PagerDuty, however they acknowledge that it may not work as well in other organisations. Tim Armandpour, while Vice President of Engineering at PagerDuty, was interviewed about Failure Friday:

> "Each business will have to build its own approach to Failure Fridays to be successful, he points out. There isn't a simple formula for everyone to follow. Some smaller businesses might have some employees dedicated to managing failure scenarios, while a bigger company might treat it as a more centralized fire-drill, he says. However you approach it, you want the goal to be staying two steps ahead, and being proactive instead of reactive – so you're never left struggling to fix a problem for a customer that could have been prevented.
>
> "Building that super strong culture where you're not panicking in moments of failure, which I think is fairly commonplace, you build a ton of trust and empathy inside your organization that I think is absolutely invaluable, especially as organizations grow and infrastructures get more complex," Armandpour says."
>
> Failure Fridays helps one company build a better customer experience[178]
>
> *Sarah K. White*

[177] https://www.pagerduty.com/blog/failure-friday-at-pagerduty/

[178] http://www.cio.com/article/3084262/relationship-building-networking/failure-fridays-helps-one-company-build-a-better-customer-experience.html

TEST STRATEGY IN DEVOPS

This book has explained a variety of practices that could be implemented as part of the test strategy in a DevOps organisation:

- Pairing
- Deployment pipelines
- Feature toggles
- Bug bash
- Crowdsourced testing
- A/B testing
- Beta testing
- Monitoring as testing
- Canary release
- Staged rollout
- Dogfooding
- Dark launching
- Environment management
- Infrastructure testing
- Destructive testing

When trying to establish a strategy for your own organisation, the number of options might be overwhelming. our organisation is unlikely to adopt every practice. Instead the development and operations team will choose a set of practices that suit them.

In an organisation that is aiming for a more regular release cadence, there is still a need to evaluate the product through testing. Testing is not disappearing. But if that goal is paired with more frequent releases, probably the most appealing aspect for management, then testing must be equally responsive.

Fast feedback depends on mature communication paths. In DevOps, there are paths beyond the development team that challenge when you learn about your products. If you can get rich and rapid feedback from the operations team, how does this change requirements for testing before you release?

Jerry Weinberg says that “quality is value to some person”. Ultimately the person who matters most in determining the value and quality of our product is your user. If you enable the user to test, then you receive an opinion of quality that comes directly from the most important stakeholder. This may be a challenging idea for a tester who has previously acted as the voice of the user in the development team.

On the flipside, you want to release a product that you have confidence in. It may damage the reputation of your organisation to consistently release software that has obvious problems, even to a subset of users. You want to build things right, so that user feedback tells you whether your product is valuable rather than pointing out problems. This is the place for automated pipelines to check the product using test automation.

Automation can be faster than repetitive manual checking, but it is also costly to develop and maintain. When speed to market is important, you may not want to invest as much time in writing pre-release test automation as you have in the past. A DevOps automation strategy can be more pragmatic. You need enough automation to help you make a release decision, but not so much that you are hampered by it.

If these are the overarching themes of a test strategy in DevOps, how can you determine the specifics for your organisation? This section shares ideas for discussing risk, rethinking the test pyramid, determining the balance of exploratory testing, questioning the changing role of the tester, and documenting a test strategy.

Risk workshop

It's common in software development to hear the phrase "All testing is risk-based". Despite this mantra, conversations that focus specifically on discussing risk may be relatively rare. We become complacent.

When we assume that we know what the risks are, and that we have the same opinion of risk as those around us, we create an opportunity for mistakes. When our assumptions are incorrect, we may make decisions about testing that confuse those around us. When we have different views on what risks are most important, we may argue about prioritisation of tasks.

The linear nature of waterfall development provided a consistent opportunity to have discussions about risk at the beginning of a project. In an iterative process these opportunities are not given, instead they must be taken. A regular risk workshop is a good opportunity to calibrate and confirm our understanding of why we're testing.

Appetite for risk

DevOps aims to create more frequent releases. Every time we release we might cause unexpected behaviour in the application, in the infrastructure, or with customer data. An increase in the number of releases will increase opportunities for problems.

That's not to say *there will be more* problems, just that it *will increase the opportunities for* problems.

On the flip side, the size of change means the problems should theoretically be smaller in nature than those that accompany big releases. When releasing in smaller batches, it's possible to instantly localise problems[179]. DevOps also aims to improve reliability of releases through automated deploy and rollback, aiding recovery from issues.

It's still important to gauge the appetite of your organisation for an increase in production issues. Nobody likes it when things go wrong, but will people accept it under the proviso that the system is "not wrong long[180]"? How critical is accuracy and availability in your software?

When asked directly, people tend to claim a conservative position. "We can't be wrong at all!" But when challenged, you'll find they are prepared to accept all sorts of issues that are in your production environment right now. Visual issues in a user interface. Accessibility standards that aren't met. Third party library versions that are out of support.

Instead of asking directly, start a risk workshop with an exercise focused on risk appetite. Gather all the people involved in a release decision in one location, then start the conversation about risk. Ask questions that reveal opinions on current state and future direction. I've previously used questions that are adapted from Adam Knight's The Risk Questionnaire[181]:

[179] http://www.startuplessonslearned.com/2009/02/work-in-small-batches.html
[180] https://vimeo.com/162635477
[181] http://www.a-sisyphean-task.com/2016/09/the-risk-questionnaire.html

- Where do you think [product] currently stands in its typical level of rigour in testing?
- Where do you think [product] should stand in its typical level of rigour in testing?

To make this an interactive exercise, you might ask people to give their answer by choosing a place to stand in the room: one wall of the room for low and the opposite wall for high. The results give a visual indicator of how people feel about the existing approach to testing and which direction they want to head towards. It highlights differences in opinion between individuals and roles, and provides a starting point for discussion.

When I ran this exercise with a team, I saw a correlation between the answer to these questions and risk appetite. People who indicated a preference for less rigorous testing generally had a higher tolerance for problems in production.

Identifying risk

Continuing the workshop session, ask the same audience to consider what risks are present in delivering and supporting their software then write out one risk per sticky note.

People may need to be reminded that the focus is on risk rather than testing activities as their tendency may be to express their ideas in the latter format. For example, instead of the activity 'cross-browser testing' a person could capture the risk that 'product may not work on different platforms'.

After five minutes of brainstorming, ask the attendees to share the risks that they have identified. As each risk is shared, other attendees should raise where they have captured a duplicate risk. For example, collect every sticky note with a risk similar to 'product may not work on different platforms' and combine them into a single group.

When I ran this exercise with a team we ended up with a list of 12 specific risks that spanned the broad categories of functionality, code merge, cross-browser compatibility, cross-platform compatibility, user experience, accessibility, security, performance, infrastructure, test data, confirmation bias and reputation.

Hopefully the diversity of perspectives present will generate a rounded view of risk. However, it's worth spending a little time to compare the list that your team generates against an external resource. I reference James Bach's Heuristic Risk-Based Testing[182] to prompt broader thinking.

Once aggregated, order the risks to create a prioritised risk backlog. The first risk in the stack should be the most important to mitigate, with following risks ordered all the way down to trivial matters.

Mitigating risk

Next, discuss how the team currently mitigate each of the risks, and how they could approach these risks in the future. This discussion might stay at a high level to identify only the type of activity that is needed, or it might delve into specifics.

[182] http://www.satisfice.com/articles/hrbt.pdf

People can become uncomfortable in conversations about mitigating risk; they may feel that talking about it makes it their responsibility. If you sense reluctance to definitively say that a risk will be mitigated by a particular type of activity, it may be worth clarifying. You are only discussing mitigating risk, not eliminating it altogether.

To explain the difference between mitigating and eliminating risk I use an example from one of my volunteering roles as a Brownie Leader. In one of the lodges where we hold our overnight camps there is a staircase to use the bathrooms that are located on a lower level. To mitigate the risk of a girl falling on the stairs at night, we leave the stairwell light switched on. This action doesn't mean that we will never have someone fall on the stairs, but it significantly reduces the likelihood. The risk is mitigated but not eliminated.

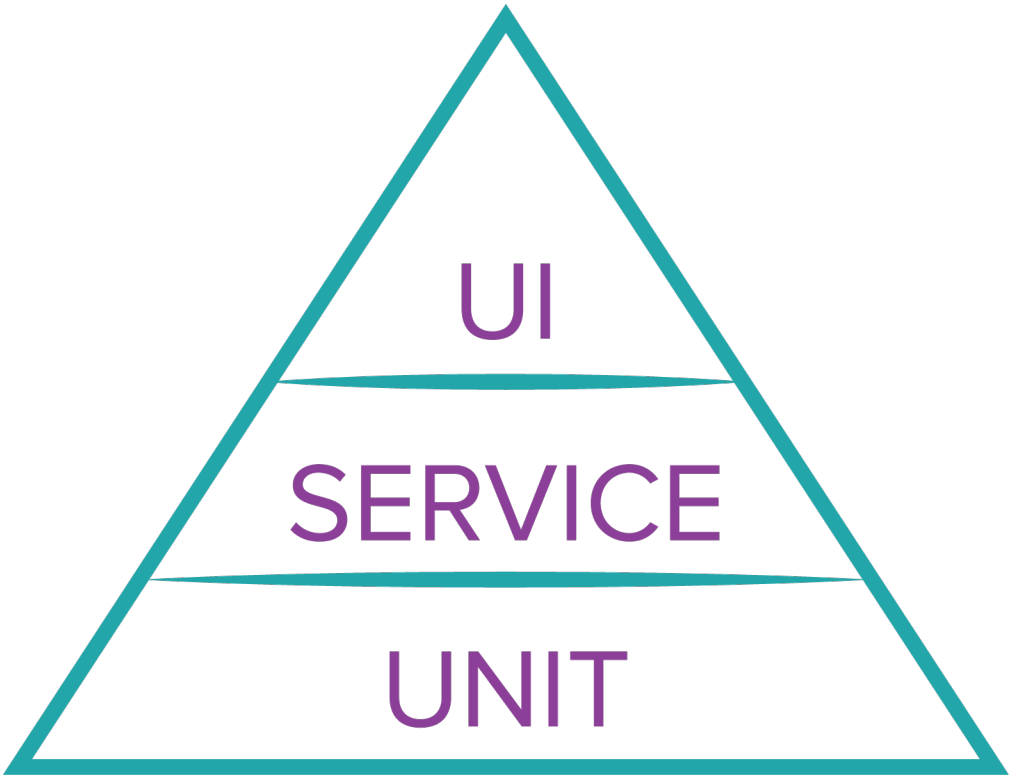
The language in this section of the workshop is intentionally broad. An activity that mitigates a risk may not be a testing activity. It could be a change in coding practices, design workshops, or production monitoring. Think widely about the options that DevOps provides.

An interesting question to ask in risk mitigation discussions within DevOps is "How late can we do that?". Challenge whether risks have to be mitigated prior to delivery or can be addressed in production. The more risks we defer, the faster we can deliver. However, the more risks we defer, the stronger the likelihood of a production problem. Look for the balance for your organisation.

A risk workshop can create a collective understanding of what risks are present and how we wish to mitigate them. The results of this activity feed into the creation of a test strategy.

Rethinking the test pyramid

A pervasive concept in agile test strategies is the testing pyramid. Originally developed by Mike Cohn in his 2009 book Succeeding with Agile[183] the pyramid contains three layers:



The testing pyramid encourages development teams to consider the proportion of test automation that they create at each layer. Unit tests that are granular, discrete checks of functionality should be added in larger numbers. There should be fewer user interface (UI) tests that interact with the entire system.

In recent years there has been growing discontent in the testing community about the broad and unthinking adoption of the test pyramid as the basis of a test strategy.

> "But does the Test Pyramid apply in all projects, or for all teams? Do all effective automated test suites follow the distribution described in the pyramid? Is it really the best way to approach testing on your project?"
>
> The short answer is no.
>
> A Test Pyramid Heresy[184]
>
> *John Ferguson Smart*

[183] https://www.amazon.com/gp/product/0321579364

[184] https://www.linkedin.com/pulse/test-pyramid-heresy-john-ferguson-smart

Many testers have reimagined the testing pyramid over the years. In the article referenced above, John Ferguson Smart goes on to argue for a model that considers whether tests are being written to discover, describe, or demonstrate[185]. At the Midlands Exploratory Workshop on Testing (MEWT) in 2015, Richard Bradshaw presented an alternative model[186] that included automation tools and skills. Marcel Gehlen has spent time considering how the original model can be interpreted in a more flexible way:

> "I do not need to slavishly stick to the layers the initial testing pyramids consist of and I don't need to focus entirely on automation anymore, yet the pyramid as a strong visualisation and framing device stays intact."
>
> Why I still like pyramids[187]
>
> *Marcel Gehlen*

More recently, in March 2017, Noah Sussman reimagined the test pyramid as a bug filter;

[185] https://www.linkedin.com/pulse/test-pyramid-heresy-john-ferguson-smart
[186] http://steveo1967.blogspot.co.nz/2015/10/mewt4-post-1-sigh-its-that-pyramid.html
[187] http://thatsthebuffettable.blogspot.co.nz/2016/03/why-i-still-like-pyramids.html

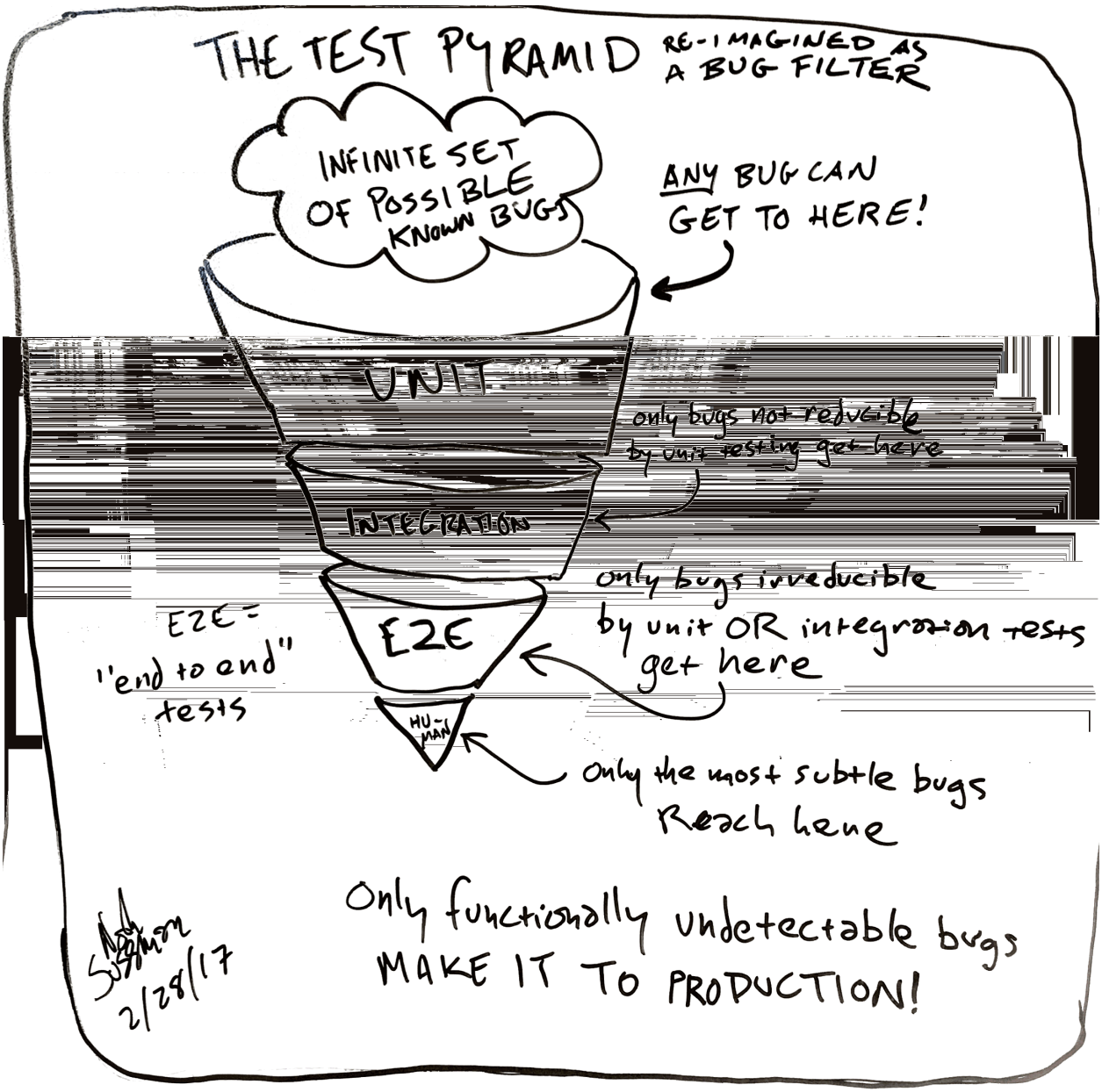


Image credit: Noah Sussman,

Abandoning the Pyramid Of Testing in favor of a Band-Pass Filter model of risk management[188]

Building on the bug filter

I like the idea of Noah's bug filter as it introduces the idea that bugs move between layers and that some may be missed. In this section I will discuss how the bug filter may be adapted in a DevOps

[188] http://infiniteundo.com/post/158179632683/abandoning-the-pyramid-of-testing-in-favor-of-a

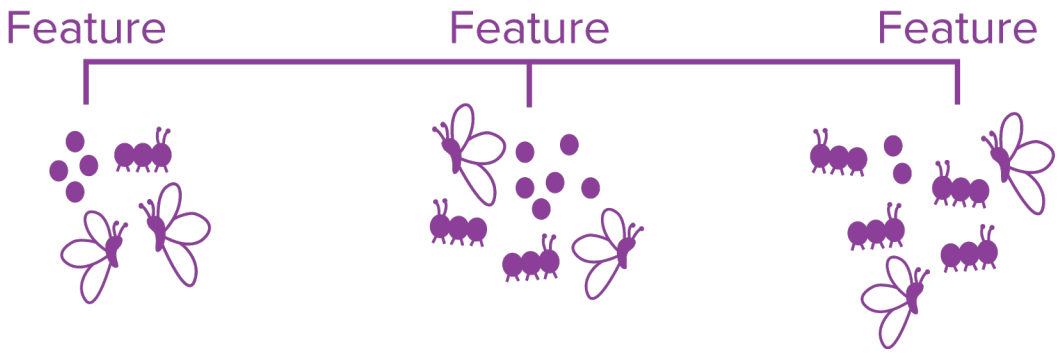
environment.

Think about all the bugs that might exist in your product. Different areas of functionality and different types of problems come to mind. Imagine a user registration function. There might be a bug where the form fails to accept a valid phone number and another that prevents the user from completing a successful registration.

Smaller bugs are those narrowly defined failures that are generally easier to identify and fix. They are also easier to test for. We can write targeted tests at the unit or service level against particular functions of our product to catch these small problems.

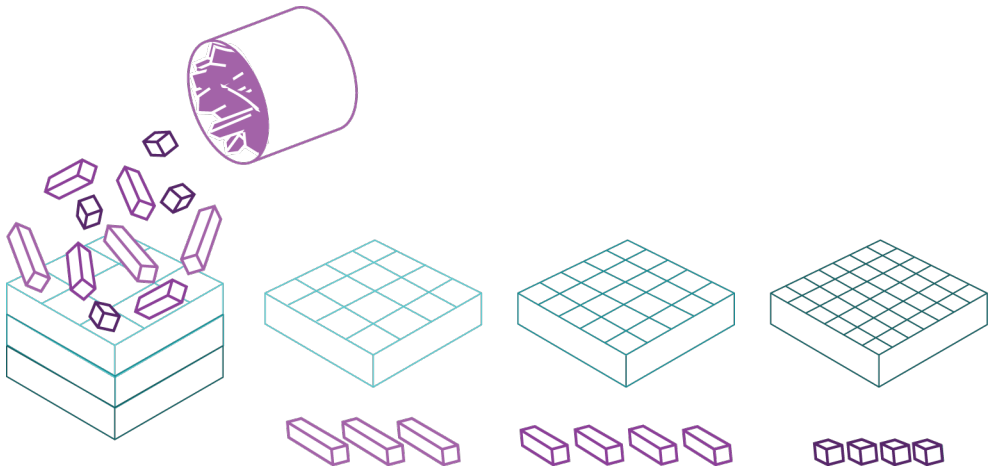
Where the potential cause of failure is unclear, we need tests that take a broad approach. End-to-end testing helps us catch larger problems. These problems might be caught at this stage because we didn't test for them earlier or might be introduced by the wider infrastructure, including platforms provided by other development teams.

Smaller and larger bugs are likely to exist in all areas of our software. You can imagine bugs of different sizes across the features:



If unit tests capture small bugs and the end-to-end tests capture the largest, then perhaps the filter as Noah describes it is upside down?

The sorting units for plastic building blocks work in the opposite way to Noah's bug filter. The top level has the widest gaps in the mesh. At each level of the box the gaps become smaller. When an unsorted collection of blocks is tipped into the top of the box and the box is shaken, the largest blocks stay at the top levels and the smallest blocks fall through to the bottom. When you take the levels apart, you can easily access the different sized blocks.



What is different between the sizes of plastic building blocks and the sizes of bugs in our product?

Plastic building blocks are constant. If I tip a small purple block into the top of the sorting unit, it will remain a small purple block throughout every layer of the unit until it is caught. The bugs in our product can evolve. If I miss a small bug at the unit test level it might become a large bug at the end-to-end test level. What was an incorrect field validation at unit test level, can lead to a larger problem when submitting the form during an end-to-end test.

What is different between a sorting unit for blocks and a filter for bugs?

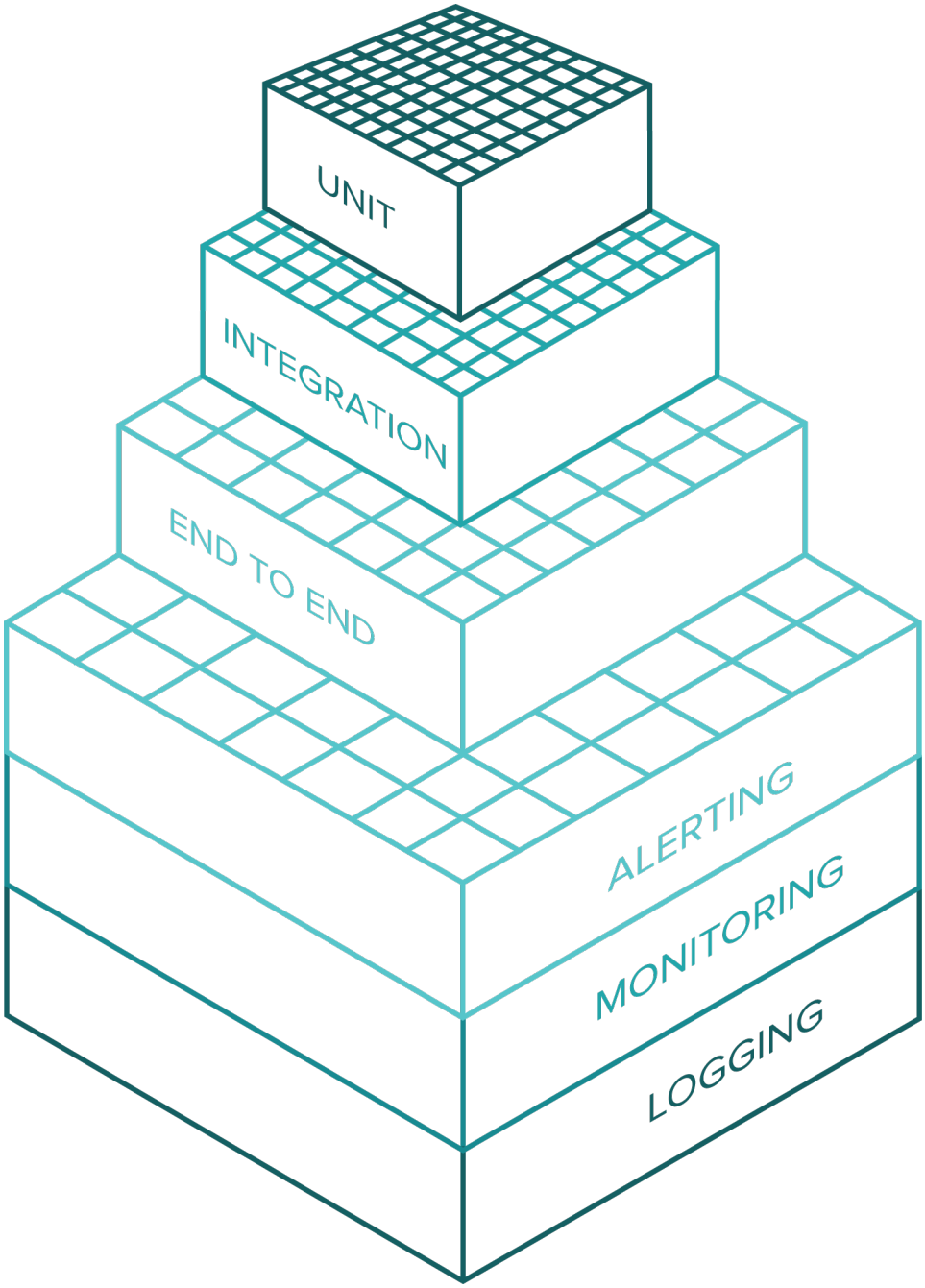
The sorting unit for blocks has the same area at each layer and is enclosed at the sides. Blocks cannot appear at a lower layer of the unit without having been tipped into the top. By contrast the area of the bug filter changes at each level. Unit tests focus solely on product code, but integration tests might include databases or external web-services. End-to-end tests cover an even larger architecture. Bugs can appear from these new systems without having passed through a previous filter.

Bugs can also appear when we enter production as a new set of perspectives are introduced to the software. Our users will interact with the software and have their own expectations about its behaviour. Bugs might appear because a user believes that there is a bug.

Test automation in a DevOps environment should not focus on the development of software in isolation. Likewise, a DevOps test strategy needs to consider a wider context that includes operations. Our model for understanding where and how to automate should be broader.

A DevOps bug filter

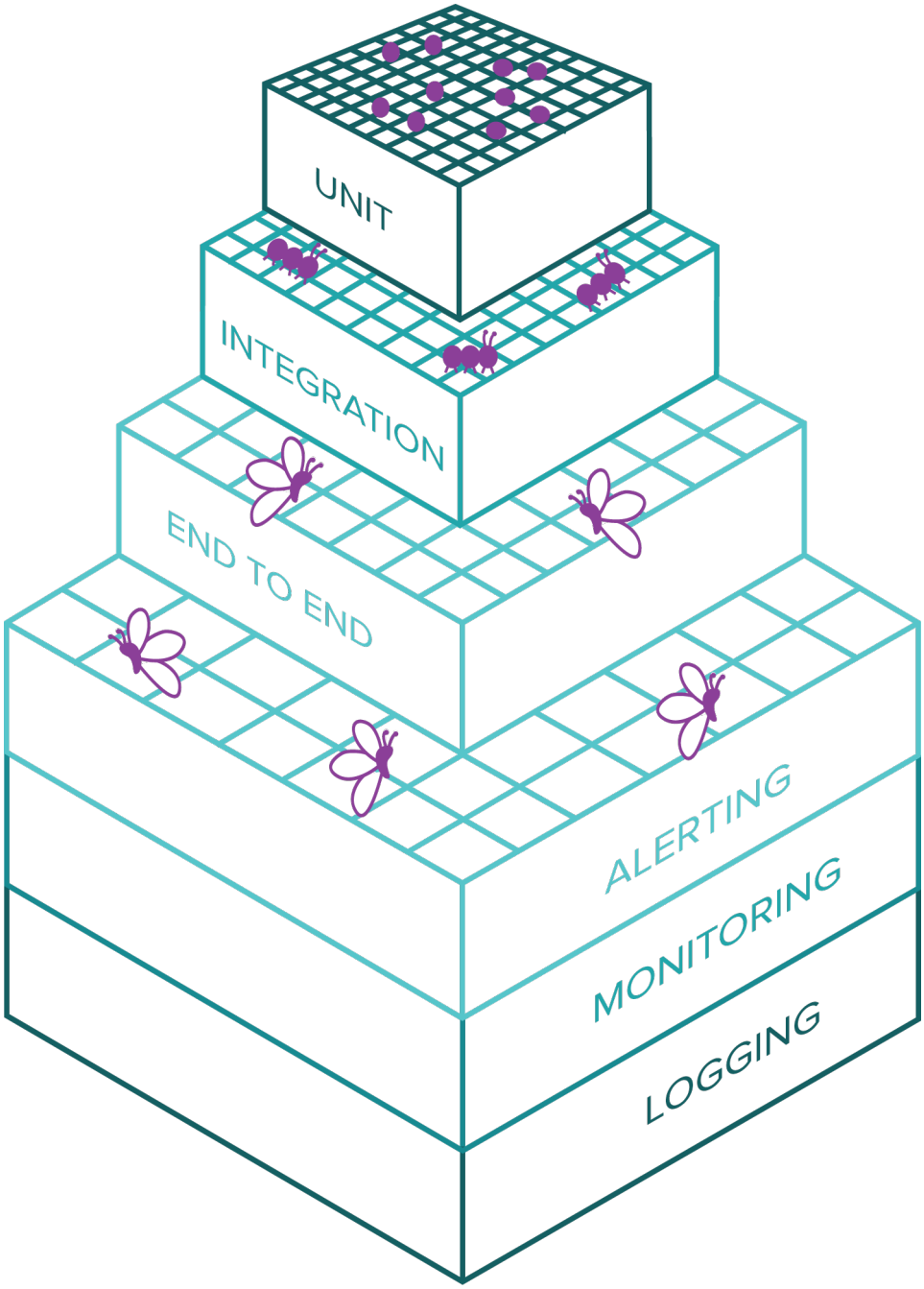
Here is the model that I imagine for automation in a DevOps environment:



There are six layers to the DevOps bug filter. The top three are for testing that occurs in the development environment: unit testing, integration testing, and end-to-end testing. The bottom three are for the information that is captured in production that can be used to detect bugs and determine product quality: alerting, monitoring, and logging.

Imagine the layers as different filters for bugs. Unit tests and logging have the finest grain mesh to catch the smallest bugs. Integration tests and monitoring have a slightly wider weave. End-to-end testing and alerting have the widest mesh to capture only the largest problems.

I imagine the bugs that drop through this filter as being butterflies in all stages of their lifecycle. Unit tests are going to capture the eggs — bugs before they develop into anything of consequence. Integration tests are going to capture the caterpillars. These may have arisen from a unit test egg that has hatched in the integrated environment, or may have crawled into our platform via a third party system. End-to-end tests capture the butterflies.



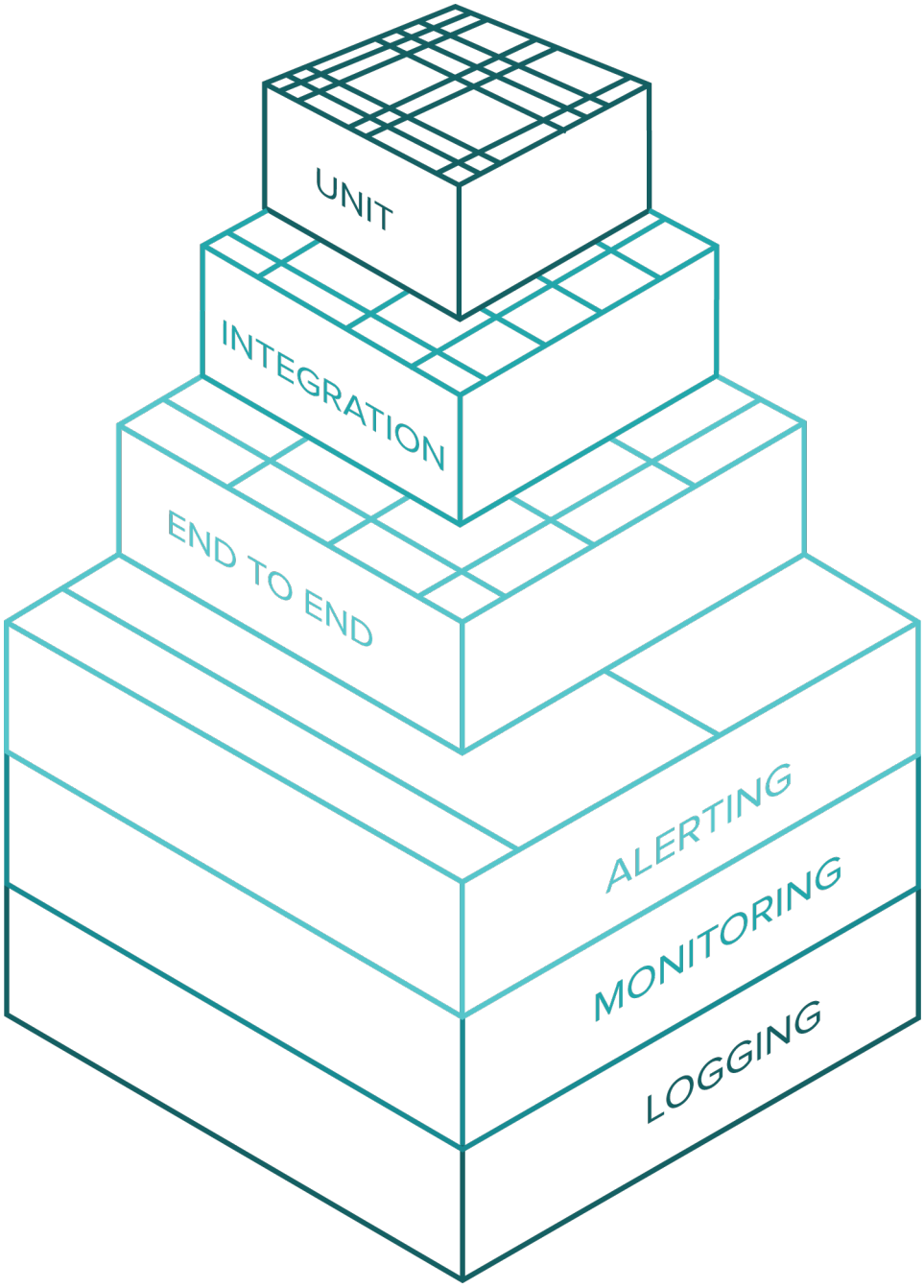
The top layers of the filter are not sealed. Even if we had 100% unit test coverage we couldn't have

prevented all the bugs from reaching lower layers of the filter. As we move through integration testing, end-to-end testing, and into the production environment, we change the infrastructure and systems that we are interacting with. Bugs can appear in these layers from unexpected origins, beyond our own application, which is why the filter trays grow in size.

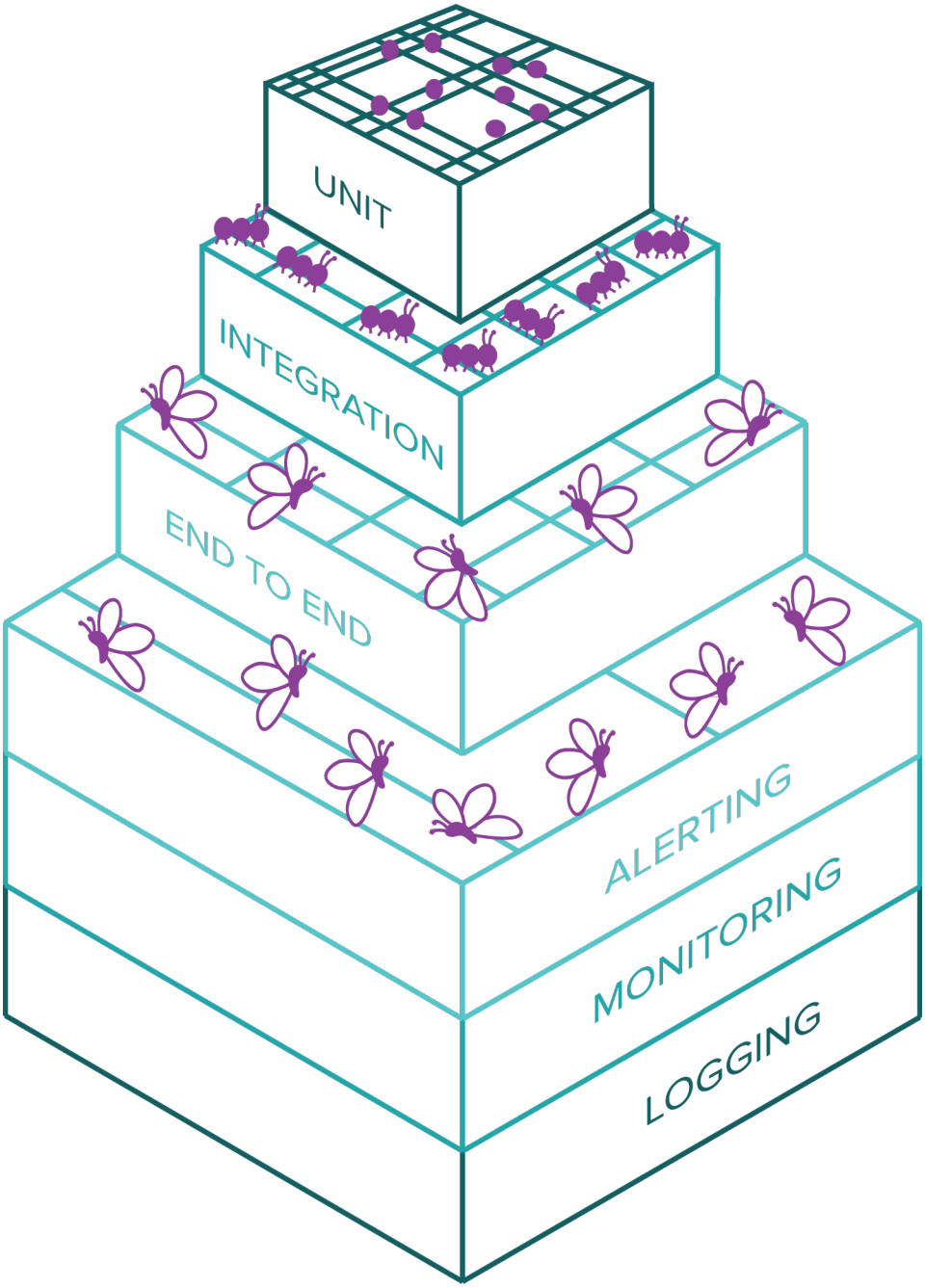
Once we reach production, the filter operates like the block filter mentioned above. The sides are enclosed. The top layer catches the largest problems, through to the bottom layer that catches the smallest. We don't have new problems emerging between the layers as at this point we are simply delving deeper into the detail of information of our production application.

The DevOps filter can help to drive conversations about a broader test strategy.

The tests that are implemented in your development team, and the data that is collected in the production environment, are the mesh in the DevOps filter. If your team have poor coverage, then the mesh is going to have gaps in it.



Where the mesh is incomplete, there are opportunities for bugs to slip through and become a larger problem. Missing coverage in the unit tests means more caterpillars in the integration test layer, and so on. More gaps may mean that more bugs are lurking in your software.

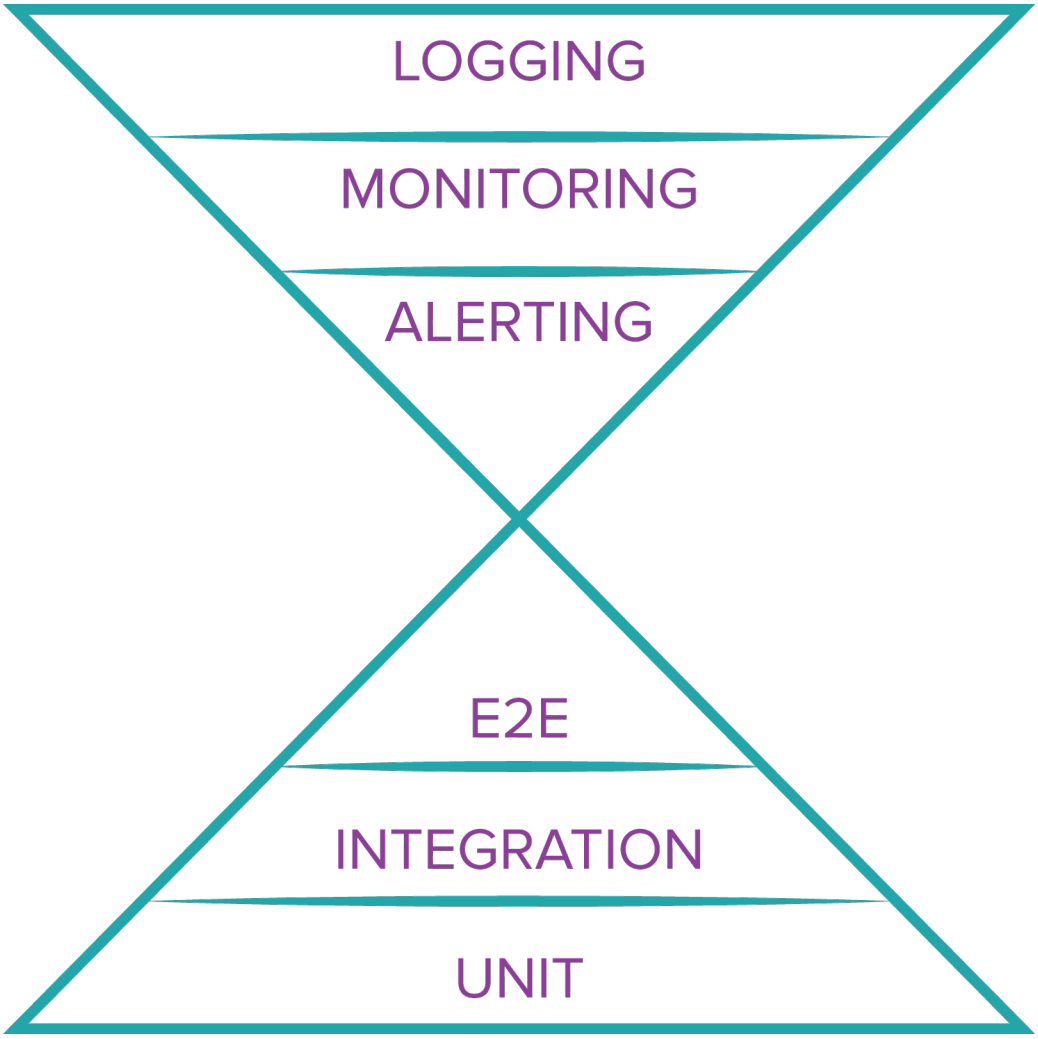


In some cases it is a strategic decision to catch a bug later in the process. Perhaps a particular integration service cannot be exposed independently or a unit test is difficult to isolate so is added to a later suite. It pays to discuss these strategic decisions to avoid misunderstanding.

A mesh might also have gaps when the development team decide that a test failure will not stop their release. It may be that it is appropriate to release to production with bugs in the software that are detected by alerting, monitoring, or logging. Part of DevOps is questioning the most appropriate place to mitigate risk and only adding the automation that is necessary.

A DevOps Hourglass

If the bug filter is too complicated to draw on a whiteboard, the starting point for conversation about strategy for testing in DevOps can be drawn as a simple hourglass:



Consider alerting, monitoring, and logging as a mirror of the original test pyramid. This is a quick way to expand the traditional conversation into the DevOps realm. Does your test strategy include the types of information operations can make available?

One final point on automation is that these conversations and models apply to functional and nonfunctional testing of the application. You can automate aspects of security, performance, usability, etc. by the same feedback mechanisms - utilising test automation in development and collecting metrics in production.

Finding balance in exploration

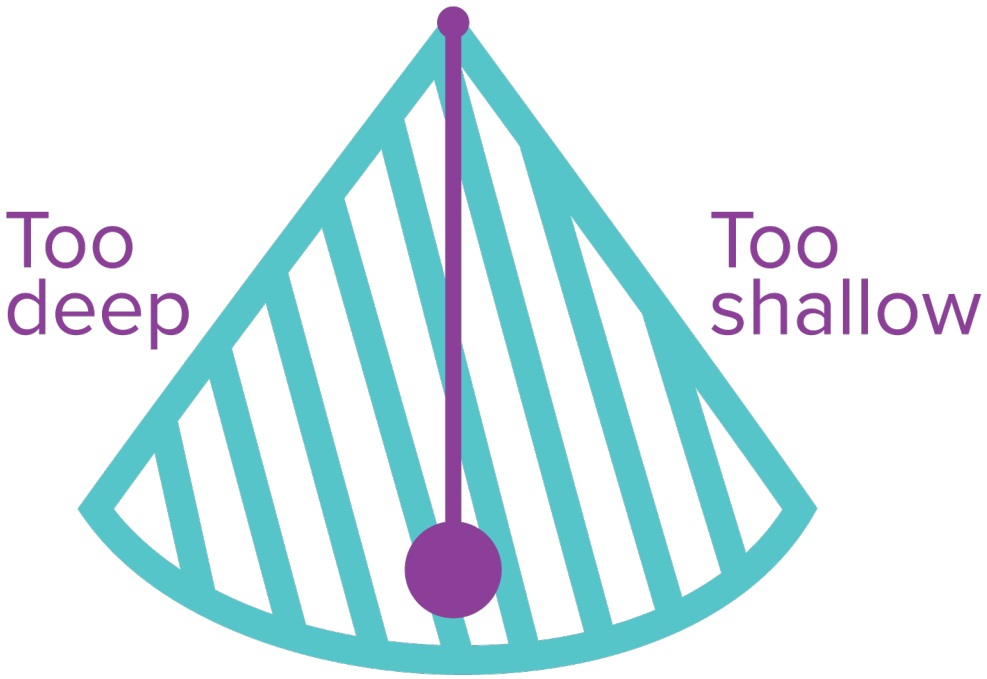
James Lyndsay has an excellent article on Why exploration has a place in any strategy[189]. He argues that exploratory testing is essential to discover and learn software functionality that has been delivered despite being outside the requirements.

In a DevOps environment it can be difficult to advocate for exploratory testing, particularly when there is a strong focus on speed of delivery. Exploratory testing tends to get squeezed to the edges of the development process. There may be exploration in the local development environment before code is merged into the main repository, then again post-release when the users are interacting with the software. There are fewer opportunities to explore within an automated pipeline process.

With limited time, how can a tester determine their strategy for exploration? How detailed should exploratory testing be?

The Testing Pendulum

Imagine a pendulum at rest. The space in which the pendulum can swing is our test approach. At the left apex we are going too deep in our testing and at the right apex we are staying too shallow. Initially, the testing pendulum sits directly in the middle of these two extremes on the spectrum:



When a tester starts in a new organisation or a new project, we apply the first movement to our testing pendulum. Lift the pendulum up to the highest point and letting go. The swing starts from

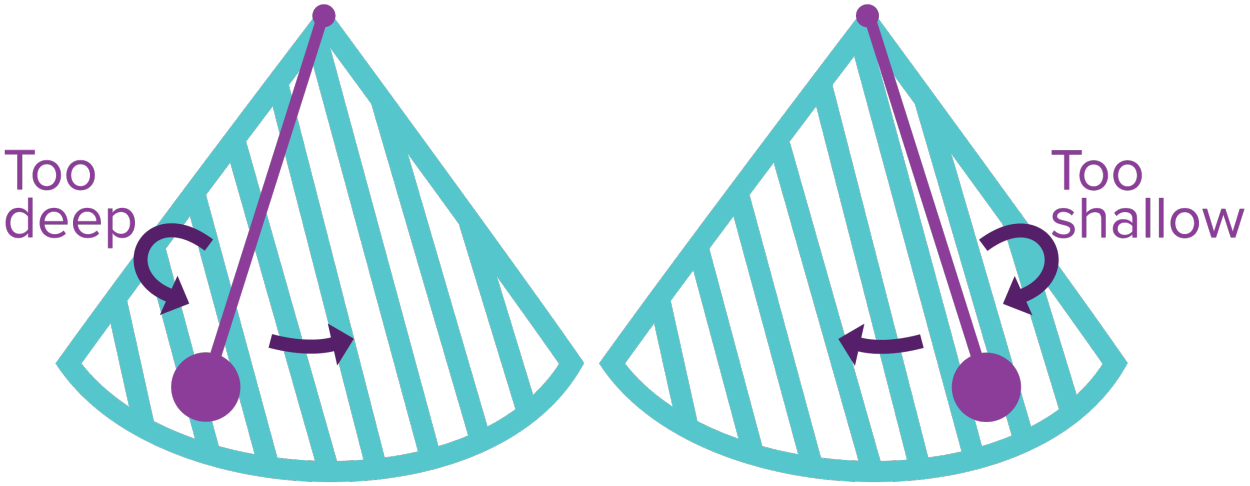
[189] http://www.workroom-productions.com/papers/Exploration%20and%20Strategy.pdf

the shallow side of the spectrum to reflect the limited knowledge we have as we enter a new situation. As our experience deepens, so too does our testing.

> "When given an initial push, [the pendulum] will swing back and forth at a constant amplitude. Real pendulums are subject to friction and air drag, so the amplitude of their swings declines."
>
> Wikipedia[190]

Similarly in testing, the pendulum will swing backwards and forwards. When our testing has become too intensive, we switch direction and swing back. When our testing has become ineffectual, we swing back again.

[190] https://en.wikipedia.org/wiki/Pendulum



The key skill for knowing how detailed testing should be is recognising when the pendulum is at the top of its swing. You need to be able to identify when you've gone too deep or stayed too shallow, so that you can adjust your approach appropriately.

We use indicators to help us determine the position of our pendulum in the testing spectrum. I see three categories of indicator: bug count, team feedback and management feedback.

Bug count

If you are not finding many bugs in your testing, but your customers are reporting a lot of problems in production, then your testing may be too shallow. On the flip side, if you raise a lot of bugs but not many are being fixed, or your customers are reporting zero problems in production, then your testing may be too deep.

The 'zero problems in production' measure may not apply in some industries. A web-based application may allow some user interface bugs to be released where the economics of fixing these does not make sense, but a company that produces medical hardware may seek to only release a product that they believe is perfect, no matter how long it takes to test. Apply the lens of your own organisation.

Team feedback

Regardless of the composition of your team and the development lifecycle that you follow, you are likely to receive feedback from those around you. Be open to those opinions and use them to adjust your behaviour.

If your colleagues frequently add scope to your testing, question whether you've spent enough time doing your testing, or perform testing themselves that you think is unnecessary, then your testing may be too shallow. On the flip side, if your colleagues frequently remove scope from your testing, question what else you could be doing with the time you spend testing, or do not perform any testing themselves, then your testing may be too deep.

A particularly useful indicator in a DevOps team is how much or little non-testers choose to test. You may see non-testers shirk their responsibility for quality, e.g. developers not writing unit tests or business people delegating acceptance testing to a tester. Be aware of how your own actions as a tester might contribute to a lack of shared ownership of the product.

Management feedback

If your testing pendulum is sitting at a point in the spectrum where your team are unhappy, it's likely that your manager will have a direct conversation with you about your test approach.

If you're testing too deeply, your manager will probably feel comfortable about telling you this directly. If your testing is too shallow, you might be asked for more detail about what you're testing, be questioned about bugs reported by users, or have to explain where you're spending your time.

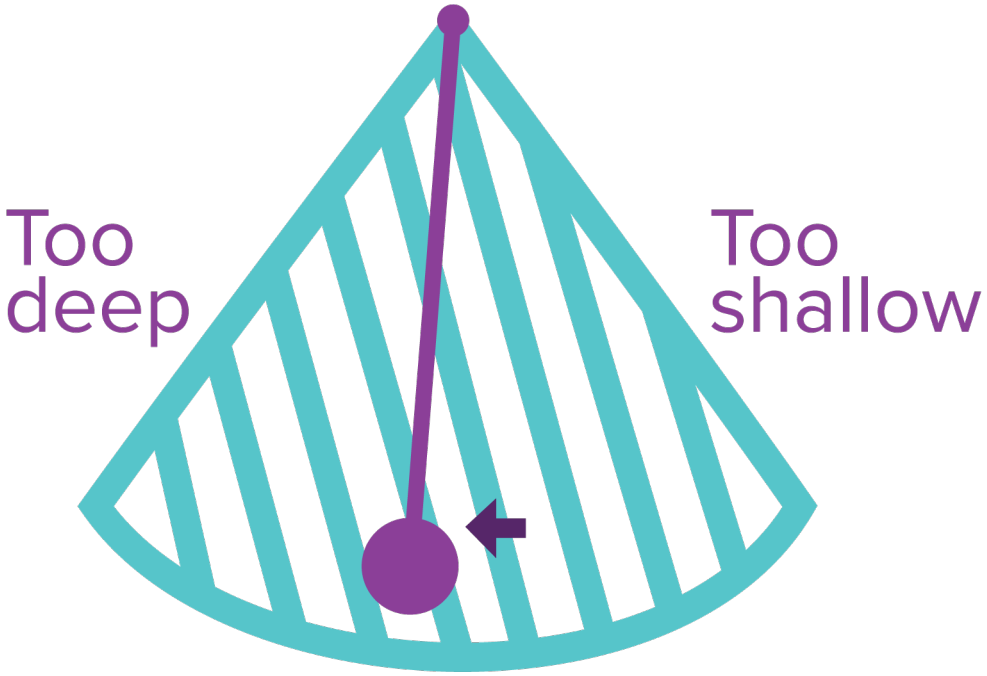
Indicators are heuristics, not rules. They include subjective language, i.e. "many", "not many", "frequently" or "a lot", which will mean different things in different situations. As always, apply the context of your organisation to your decision making.

The indicators that I've described can be summarised by opposing statements that represent the extremes of the pendulum swing:

Too Deep	Too Shallow
You raise a lot of bugs, not many are fixed	You don't find many bugs
Zero bugs in production	A lot of bugs in production
In peer review or debrief, your colleagues frequently remove scope from your testing	In peer review or debrief, your colleagues frequently add scope to your testing
Your team or peers question the opportunity cost of the time that you spend testing	Your team or peers question whether you have spent enough time testing
Your team or peers are not performing any testing themselves	Your team or peers are performing testing themselves that you think is unnecessary
Your manager tells you directly that you're testing too much	Your manager asks you for more detail about what you're testing

Eventually, most testers will end up at an equilibrium where the pendulum hangs at rest, halfway between the two extremes. The comfort of this can be problematic. Once we have confidence in our approach we can become blind to the need to adjust it.

Perhaps the state that we want to strive for lies slightly towards the 'too deep' end of the spectrum. In order to keep a pendulum positioned off-center, we have to regularly apply small amounts of pressure: a bump!



If you've been testing a product for a long time and wish to avoid becoming stale, give your testing a bump towards greater depth. Apply some different test heuristics. Explore using different customer personas. Alter your test data. Any variation that could provide you with a fresh crop of problems to explore the validity of with your team.

You can use the outcome of a bump to calibrate your test approach within a smaller range of pendulum movement. Which changes are too much? Which are welcome and should be permanently added to your testing repertoire? Continued small experiments help to determine that you are in the right place.

The testing pendulum is a useful analogy for describing how to find balance in exploration. When entering a new team or project, it can be used to illustrate how we experience large initial swings in the depth of our testing as we learn the indicators of our environment. For those who have been testing the same product for a while, it can remind us to continuously verify our approach through bumps.

Testing vs Tester

The "Test is Dead" rhetoric originated in 2011 when Alberto Savoia took the stage at the Google Test Automation Conference (GTAC) wearing a Grim Reaper costume[191]. This opinion has proliferated and in the DevOps community people make unequivocally negative statements about testing, like:

> "In the end, if there is a layer of people in the middle between development and operations, then you cannot perform CI and CD seamlessly and, therefore, cannot do DevOps. To run a proper DevOps operation, you cannot have QA at all."
>
> How DevOps is Killing QA[192]
>
> *Asaf Yigal*

Much of this attack is on the role of a tester rather than testing as an activity. I have not seen anyone advocate for software development that is absent from considerations about quality altogether.

The changing role of a tester

In 2011 Seth Eliot was a Senior Knowledge Engineer for Microsoft Test Excellence when he published an article titled "The Future Of Software Testing Part Two"[193]. The article focused on TestOps and the shift that he observed in the testing profession as a result.

Seth contrasted the way the testing happens in a traditional environment against the way that quality could be assessed in a DevOps environment. In a traditional environment quality is assessed during development, but in DevOps quality is assessed through user interactions with the production system. He summarised the differences in approach as:

Three Stages of Traditional Testing and TestOps

Input	Run tests	Production usage
Signal (Output)	Test results	Telemetry data
Observe	Pass/Fail	KPIs and patterns

In a TestOps environment Seth identified two key ways in which the day-to-day work of a tester might change:

1. The tester moves away from up-front testing and execution of functional test cases. This responsibility shifts towards the developer who "must use every available means to produce code free of low context bugs that can be found by running on his desktop"[194].

[191] https://www.youtube.com/watch?v=X1jWe5rOu3g
[192] https://devops.com/devops-killed-qa/
[193] https://dojo.ministryoftesting.com/lessons/the-future-of-software-testing-part-two
[194] https://dojo.ministryoftesting.com/lessons/the-future-of-software-testing-part-two

2. The tester creates tools that can run tests in the production environment, which may look a lot like monitors. "The value test brings is in moving these monitors from heartbeats and simple scenarios to high context user affecting scenarios"[195].

In early 2017 Kim Knup reflected on her experience as a Software Tester at Songkick. Though Kim had worked in agile environments previously, this role was the first time that she had been part of a team that truly used continuous delivery. This was a big culture shift, which Kim summarised as:

> "Forget about "Ready for test" columns. Everything is continuously being tested and not necessarily by you. Think about how you can integrate your tester mind-set to be heard."
>
> Joining a CD Team the experience report[196]
>
> *Kim Knup*

A lot of the advice that Kim offers in her article is focused on building relationships across the development and operations team, then using these connections to ask test-focused questions. Kim explained her involvement in planning discussions, pairing with developers, determining test coverage with business representatives, and being part of cross-functional team channels in instant messaging tools. As a tester at Songkick, Kim felt that she embraced "more of a coaching mindset" and became an advocate for quality.

She also echoed Seth's advice saying:

> "Get to know your production data really well, as a lot of testing will be done in the live environment. This has great advantages as the code is being tested where it is being used with real data but it can be daunting, as you could be affecting real reporting. Liaise with the relevant departments to help sort this and avoid any confusion."
>
> Joining a CD Team the experience report[197]
>
> *Kim Knup*

For Seth and Kim the tester role has evolved. In some organisations, rather than changing the role, there is a push towards eliminating the tester role entirely.

Heuristics for removing a tester

An organisation might consider removing a tester if they believe that testing activities can be conducted by other roles within the development team, or by the users themselves in the production environment. Though there are no definite rules for determining whether a tester can be removed, I see five areas to consider: context beyond the team, support for quality, team dynamic, measurement, and bias.

[195] https://dojo.ministryoftesting.com/lessons/the-future-of-software-testing-part-two
[196] https://punkmiktests.wordpress.com/2017/04/28/joining-a-cd-team-the-experience-report/
[197] https://punkmiktests.wordpress.com/2017/04/28/joining-a-cd-team-the-experience-report/

Context beyond the team

> *It would be interesting to understand their motivation behind asking that question.*
>
> Dan Billing

The wider context of the change you are making will have significant impact on how people feel about it. You should consider what your underlying reasons are, how people will feel about those reasons, and how far-reaching the implications of your change could be.

What's the context of the movement being made? In Dynamic Reteaming[198], Heidi Hefland talks about five different scenarios in which you may want to change a team:

- Company growth
- The nature of the work
- Learning and fulfilment
- Code health
- Liberate people from misery

Is one of these applicable? If not, can you succinctly explain what your reason is? How is this wider context being viewed by the team(s) involved? Are they enthusiastic, cautiously optimistic, or decidedly negative?

What is the composition of surrounding teams? Similar or different? How will that impact the outcome? If I'm part of the only team without a tester, or the only team with a single tester, what impact will that have?

If there are governance processes surrounding release, is the person who approves change to production supportive of the change to the team? Will that person still have confidence in what the team deliver?

Support for Quality

> *If you remove a piece from a Jenga puzzle, will it fall? The impact depends on what it supports.*
>
> Sean McErlean

The quality of your product doesn't come from testing alone. There are many activities that contribute to creating software that your customers enjoy. It's important to determine the depth and breadth of practices that contribute to quality within the team that you're looking to change.

What level of test automation is available to support the team? Is it widely understood, regularly executed and actively maintained?

[198] https://leanpub.com/dynamicreteaming

What other practices contribute to quality in the team? Pair programming? Code review? Code quality analysis? Continuous integration? Business acceptance? Production monitoring? To what extent are these practices embedded and embraced?

What does the tester do outside of testing? What sort of impact do you expect on the social interactions of the team? Agile rituals? How about depth of product knowledge? Customer focus? These things may not specifically be test activities or skills, but do impact quality of the product.

Team Dynamic

> *You never remove a tester though, you remove a person and that person is unique.*
>
> David Greenlees

Make the question personal. You want to consider the real people who will be impacted by the decision that you're making. There may be multiple facets to removing a tester: think about the team that they leave, the team that they move to, and the experience of the individual themselves.

How many testers do you have in the team? Are you removing the only tester?

Are you removing the tester role or testing as an activity? Are there, or will there be, others in the team with testing experience and knowledge, even if they are not testers? How will other people in the team feel about adopting testing activities? What support will they need?

Are you replacing the tester in the team with another person? What will their role be? How will the change impact specialist ratios within the team?

If the person is being moved to a new team, what opportunities exist for them there? How will their skills and experience match their new environment? What impact do you expect this change to have on the team that they are joining? How will the people in that team have to change their work to accommodate a new team member?

Measurement

> *How do you know what quality is now?*
>
> Keith Klain

As with any change, it's important to understand how you'll track the impact. Changing the way that a team approach testing could impact both the quality of software they create and how they feel about the work they do, in a positive or negative way.

What metrics help you determine the quality of your product? If quality decreases, how low is acceptable? What is your organisation's appetite for product risk? How many production bugs can the reputation of your organisation withstand?

What metrics help you determine the health of your team? If productivity or morale decrease, how low is acceptable? What is your organisation's appetite for people risk? What impact will the change have on happiness and retention of other roles?

Bias

> *How would someone else answer these questions? Remember the bias we have that impacts our answers.*
>
> Lanette Creamer

The final point to remember is that you don't want to consider these questions alone. A manager who sits outside a team may have quite different answers to a person who works within it. The tester who is being moved will have opinions. Whoever is ultimately responsible for the decision should also think about how other people would respond to the previous prompts.

Documenting a strategy

In the previous sections there are lots of ideas and practices. Aside from ongoing risk assessments and conversations, how do you gather the information about your test strategy in one place? Is there a test strategy document in DevOps?

Historically a documented test strategy was needed to inform those outside of testing about the scope and coverage of testing efforts. In a DevOps environment, "outside of testing" is probably an oxymoron. Everyone in the development and operations team is involved in testing to some degree.

There must be a shared understanding across all people involved about what the overarching test strategy is. My experience is that this understanding is more likely to come from conversation than a document. What ends up written down may not make much sense to those who weren't part of the conversations and may not age very well.

In DevOps, test strategy moves from being an exhaustive capture of an individual perspective to being a high-level capture of the collective perspective. Instead of a lengthy document, aim to create a single page, visual summary of the strategy conversations that you've had within your team. Post it on the walls of your office space.

One of the most attractive examples that I have seen is the visual test strategy that Jamie McIndoe and Jaume Durany developed at Fairfax Media in New Zealand. The visualisation proved more successful than we ever anticipated in communicating our test strategy and gaining engagement across our teams[199]

[199] http://assurity.co.nz/community/big-thoughts/a-visual-test-strategy/

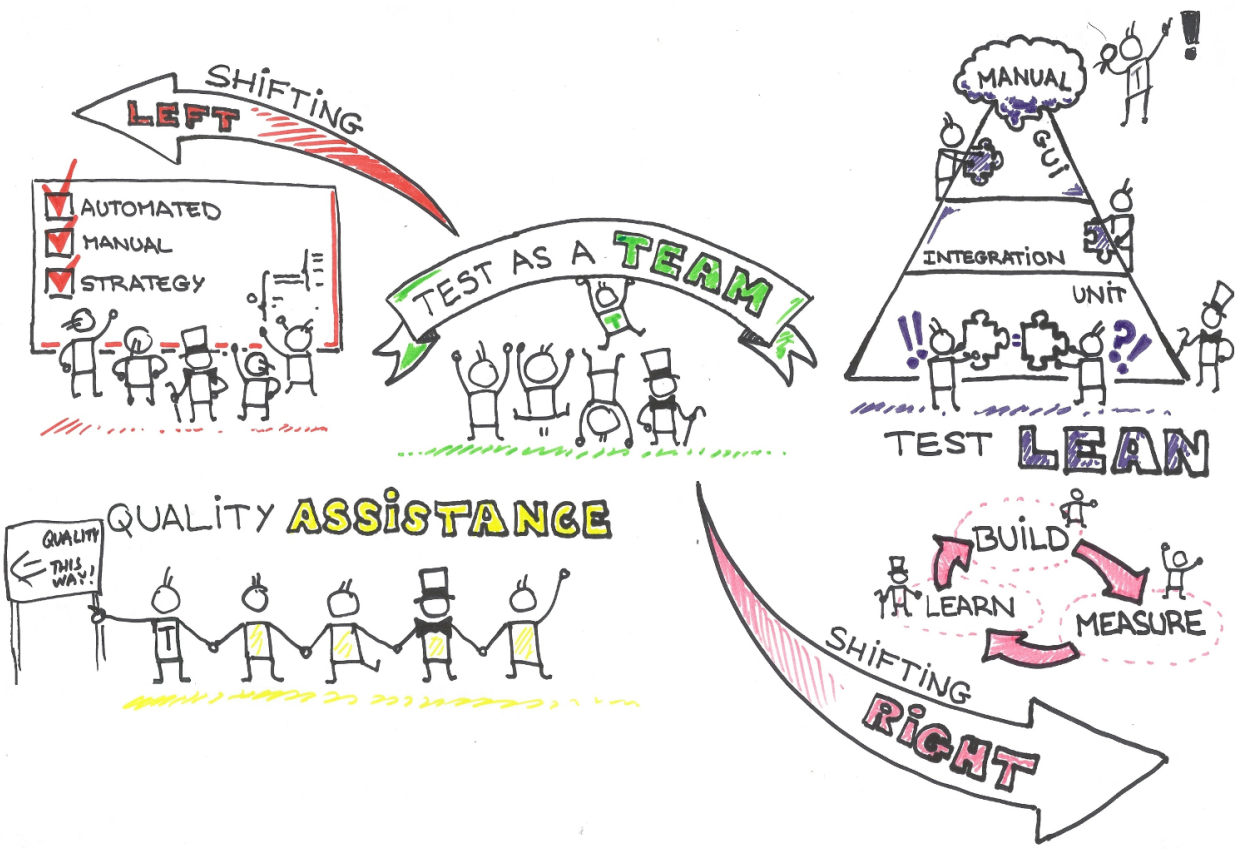


Image credit: Jamie McIndoe & Jaume Duraney, Testing Stuff - A One-Page Test Strategy[200]

In an environment with a wider set of stakeholders, including senior management or other development and operations teams within the organisation, a visual summary may not provide enough detail. If the visualisation represents a conversation that everyone was a part of, then anyone who is interested could ask for more information from any person in the team. You might also be surprised that in some cases it is enough for a manager to see evidence that a strategy is in place, they don't require the details.

[200] https://technology.fairfaxmedia.co.nz/testing-stuff-a-one-page-test-strategy/

Conclusion

DevOps presents a lot of opportunities to think differently about testing. Creating connections between the development and operations teams can influence test coverage, test strategy, and even the infrastructure where testing occurs. The degree of change in your organisation will depend on the business drivers for DevOps and the appetite for risk.

The ideas presented in this book are a starting point for your own conversations and further research. There is no best practice here, only guidance. With an understanding of the possibilities you can determine what will work for you.

DevOps is not a change to fear. Testing remains an integral part of the development process. But testers need to get involved. I hope that you will take inspiration from the experiences shared in this book.

References

Allspaw, J. 10+ deploys per day through Dev and Ops cooperation at Flickr.
Presented at Velocity Conference, San Jose, 2009.
http://www.slideshare.net/jallspaw/10-deploys-per-day-dev-and-ops-cooperation-at-flickr/10-Ops_-job_is_NOT_to

Andelkovic, A. How King Uses AI in Testing.
Presented at Conference of the Association for Software Testing (CAST), Vancouver, 2016.
https://www.youtube.com/watch?v=wHlD99vDy0s

Anicas, M. Getting Started With Puppet Code: Manifests and Modules.
https://www.digitalocean.com/community/tutorials/getting-started-with-puppet-code-manifests-and-modules

Ashby, D. Continuous Testing in DevOps.
https://danashby.co.uk/2016/10/19/continuous-testing-in-devops/

Bach, J. Heuristic Risk-Based Testing.
http://www.satisfice.com/articles/hrbt.pdf

Barth, D. Failure Friday | Injecting Failure at PagerDuty to Increase Reliability.
https://www.youtube.com/watch?v=KtX_Bx7HQsA

Bennett, C.; Tseitlin, A. Chaos Monkey Released Into The Wild.
http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html

Berkun, S. How to run a bug bash.
http://scottberkun.com/2008/how-to-run-a-bug-bash/

Bertolino, A. Software Testing and/or Sofware Monitoring: Differences and Commonalities.
Presented at Jornadas Sistedes, Cádiz, 2014.
https://www.slideshare.net/ProjectLearnPAd/cadiz-bertolinolp

Bias, R. The History of Pets vs Cattle and How to Use the Analogy Properly.
http://cloudscaling.com/blog/cloud-computing/the-history-of-pets-vs-cattle/

Bird, J. Feature Toggles are one of the worst kinds of Technical Debt.
http://swreflections.blogspot.co.nz/2014/08/feature-toggles-are-one-of-worst-kinds.html

Bird, J. Monitoring Sucks. But Monitoring as Testing Sucks a Lot More.
http://swreflections.blogspot.co.nz/2012/07/monitoring-sucks-but-monitoring-as.html

Bowman, D. Goodbye, Google.
http://stopdesign.com/archive/2009/03/20/goodbye-google.html

Boyd, M. Containers in Production: Case Studies, Part 1.
https://thenewstack.io/containers-production-part-case-studies/

Bradley, T. Netflix, the Simian Army, and the culture of freedom and responsibility.
https://devops.com/netflix-the-simian-army-and-the-culture-of-freedom-and-responsibility/

Brands, C.; Clokie, K. Babble & Dabble: Creating Bonds Across Disciplines.
Presented at Conference of the Association for Software Testing (CAST), Vancouver, 2016.
https://www.youtube.com/watch?v=RgcNgabDN_c

Brittain, M. Tracking Every Release.
https://codeascraft.com/2010/12/08/track-every-release/

Bryant, D. Is it Possible to Test Programmable Infrastructure? Matt Long at QCon London Made the Case for "Yes".
https://www.infoq.com/news/2017/03/testing-infrastructure

Caplan-Bricker, N. If You Want to Talk Like a Silicon Valley CEO, Learn This Phrase.
https://newrepublic.com/article/115349/dogfooding-tech-slang-working-out-glitches

Clokie, K. A pairing experiment for sharing knowledge between agile teams.
http://katrinatester.blogspot.co.nz/2015/06/a-pairing-experiment-for-sharing.html

Clokie, K. Evolution of a Model.
http://katrinatester.blogspot.co.nz/2014/04/evolution-of-model.html

Clokie, K. Lightning Talks for Knowledge Sharing.
http://katrinatester.blogspot.co.nz/2016/04/lightning-talks-for-knowledge-sharing.html

Clokie, K. Testing for Non-Testers Pathway.
http://katrinatester.blogspot.com.au/2015/11/testing-for-non-testers-pathway.html

Clokie, K. Three examples of context-driven testing using visual test coverage modelling.
http://katrinatester.blogspot.co.nz/2014/10/three-examples-of-context-driven.html

Clokie, K. Visual Test Ideas.
http://katrinatester.blogspot.co.nz/2014/11/visual-test-ideas.html

Clokie, K. Visual Test Models & State Transition Diagrams.
http://katrinatester.blogspot.co.nz/2015/01/visual-test-models-state-transition.html

Cohn, M. Succeeding with Agile: Software Development Using Scrum, 1st ed.; Addison-Wesley Professional: Boston, MA, 2009.

Cois, A. DevOps Case Study: Netflix and the Chaos Monkey.
https://insights.sei.cmu.edu/devops/2015/04/devops-case-study-netflix-and-the-chaos-monkey.html

Conway, M. Conway's Law.
http://www.melconway.com/Home/Conways_Law.html

Cresswell, S. Sean on Twitter.
https://twitter.com/seancresswell/status/499394371372843008

Crispin, L.; Gregory, J. Agile Testing: A Practical Guide for Testers and Agile Teams, 1st ed.; Addison-Wesley Professional: Boston, MA, 2009.

Crispin, L.; Gregory, J. More Agile Testing: Learning Journeys for the Whole Team, 1st ed.; Addison-Wesley Professional: Boston, MA, 2014.

Davis, C. Canaries are great!
https://content.pivotal.io/blog/canaries-are-great

Dredge, S. Candy Crush Saga: '70% of the people on the last level haven't paid anything'.
https://www.theguardian.com/technology/appsblog/2013/sep/10/candy-crush-saga-king-interview

Duvall, J. How All Hands Support Works at Automattic.
https://jeremey.blog/all-hands-support-automattic/

Edmunds, S. BNZ head office staff face restructure.
http://www.stuff.co.nz/business/85563033/BNZ-head-office-staff-face-restructure

Ekart, G. DevOps Dashboard Hygieia Aggregates End-to-End View of the Delivery Pipeline.
https://www.infoq.com/news/2016/03/hygieia

Eliot, S. Exposure Control: Software Services Peep Show.
https://blogs.msdn.microsoft.com/seliot/2010/12/13/exposure-control-software-services-peep-show/

Eliot, S. Testing in Production (TiP) – It Really Happens – Examples from Facebook, Amazon, Google, and Microsoft.
https://blogs.msdn.microsoft.com/seliot/2011/06/07/testing-in-production-tip-it-really-happensex-amples-from-facebook-amazon-google-and-microsoft/

Eliot, S. Testing In Production.
Presented at TestBash, UK, 2013.
https://vimeo.com/64786696

Eliot, S. The Future Of Software Testing Part Two.
https://dojo.ministryoftesting.com/lessons/the-future-of-software-testing-part-two

Falanga, K. Teaching Testing to Non-Testers.
Presented at Conference of the Association for Software Testing (CAST), Vancouver, 2016.
https://www.youtube.com/watch?v=KUiBGtHWdeU

Falanga, K. Testing Is Your Brand: Sell It!
Presented at Conference of the Association for Software Testing (CAST), Grand Rapids, 2015.
https://www.youtube.com/watch?v=TEln1YTrjrg

Fayer, L. Production Testing Through Monitoring.
Presented at DevOpsDay, Rockies, 2016.
https://www.youtube.com/watch?v=VIhIm5kEWFg

Ferguson Smart, J. A Test Pyramid Heresy.
https://www.linkedin.com/pulse/test-pyramid-heresy-john-ferguson-smart

Fitz, T. Continuous Deployment at IMVU: Doing the impossible fifty times a day.
http://timothyfitz.com/2009/02/10/continuous-deployment-at-imvu-doing-the-impossible-fifty-times-a-day/

Fitz, T. Continuous Deployment.
http://timothyfitz.com/2009/02/08/continuous-deployment/

Fowler, M. Feature Toggle.
https://martinfowler.com/bliki/FeatureToggle.html

Gartenberg, C. Spotify now has 50 million paid subscribers.
http://www.theverge.com/2017/3/2/14795274/spotify-streaming-50-million-paid-subscribers-milestone

Gehlen, M. Why I still like pyramids.
http://thatsthebuffettable.blogspot.co.nz/2016/03/why-i-still-like-pyramids.html

Goble, S. Perfect software: the enemy of rapid deployment?
https://www.theguardian.com/info/developer-blog/2016/dec/04/perfect-software-the-enemy-of-rapid-deployment

Goble, S. So What Do You Do If You Don't Do Testing?!
Presented at PIPELINE Conference, London, 2016.
https://vimeo.com/162635477

Goble, S. What really happens when you deliver software quickly?
Presented at Test Automation Day Conference, Rotterdam, 2016.
https://speakerdeck.com/sallygoble/what-really-happens-when-you-deliver-software-quickly-1

Goldin, E. How Spotify does Continuous Delivery with Docker and Helios.
https://www.youtube.com/watch?v=5Ycb7jlZGkU

Gordon, B.; Zettelmeyer, F.; Bhargava, N.; Chapsky, D. A Comparison of Approaches to Advertising Measurement: Evidence from Big Field Experiments at Facebook.
https://www.kellogg.northwestern.edu/faculty/gordon_b/files/kellogg_fb_whitepaper.pdf

Greaves, K. Assessment - How agile is your testing?
http://www.growingagile.co.za/2015/10/assessment-how-agile-is-your-testing/

Hanlon, M. Incredible pictures of one of Earth's last uncontacted tribes firing bows and arrows.
http://www.dailymail.co.uk/sciencetech/article-1022822/Incredible-pictures-Earths-uncontacted-tribes-firing-bows-arrows.html

Hare-Winton, J.; Cutler, S. Testing in Production: How we combined tests with monitoring.
https://www.theguardian.com/info/developer-blog/2016/dec/05/testing-in-production-how-we-combined-tests-with-monitoring

Heidi, E. An Introduction to Configuration Management.
https://www.digitalocean.com/community/tutorials/an-introduction-to-configuration-management

Hendrickson, E. Explore It! Reduce risk and increase confidence with exploratory testing, 1st ed.; The Pragmatic Programmers: Raleigh, NC, 2013.

Hendrickson, E. Test Heuristics Cheat Sheet. http://testobsessed.com/wp-content/uploads/2011/04/testheuristicscheatsheetv1.pdf

Heusser, M. Continuous Deployment Done In Unique Fashion at Etsy.com. http://www.cio.com/article/2397663/developer/continuous-deployment-done-in-unique-fashion-at-etsy-com.html

High, P. How Capital One Became A Leading Digital Bank. http://www.forbes.com/sites/peterhigh/2016/12/12/how-capital-one-became-a-leading-digital-bank/

Hodgson, P. Feature Toggles. https://martinfowler.com/articles/feature-toggles.html

Holson, L. Putting a bolder face on Google. http://www.nytimes.com/2009/03/01/business/01marissa.html

Humble, J. Continuous Delivery vs Continuous Deployment. https://continuousdelivery.com/2010/08/continuous-delivery-vs-continuous-deployment/

Humble, J. Continuous Delivery. https://continuousdelivery.com/

Humble, J. Continuous Delivery: Anatomy of the Deployment Pipeline. http://www.informit.com/articles/article.aspx?p=1621865&seqNum=2

Humble, J.; Farley, D. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, 1st ed.; Addison-Wesley Professional: Boston, MA, 2010.

Izrailevsky, Y.; Tseitlin, A. Netflix Simian Army. http://techblog.netflix.com/2011/07/netflix-simian-army.html

Janaway, S. Experiences of a Testing Bug Bash. http://stephenjanaway.co.uk/stephenjanaway/experiences/experiences-testing-bug-bash/

Kaner, C.; Bach, J. Exploratory Testing in Pairs. http://www.testingeducation.org/a/pairs.pdf

Keyes, E. Sufficiently Advanced Monitoring is Indistinguishable from Testing. Presented at Google Test Automation Conference (GTAC), New York, 2007. https://www.youtube.com/watch?v=uSo8i1N18oc

Kim, G.; Behr, K.; Spafford, G. The Phoenix Project: A Novel about IT, DevOps, and Helping Your Business Win, 1st ed.; IT Revolution Press: Portland, OR, 2013.

Knight, A. The Risk Questionnaire. http://www.a-sisyphean-task.com/2016/09/the-risk-questionnaire.html

Knup, K. Joining a CD Team the experience report. https://punkmiktests.wordpress.com/2017/04/28/joining-a-cd-team-the-experience-report/

Laing, S.; Greaves, K. Growing Agile: A Coach's Guide to Agile Testing, 1st ed.; LeanPub: Vancouver, BC, 2016.

Lapidos, J. Why Did It Take Google So Long To Take Gmail Out of "Beta"? http://www.slate.com/articles/news_and_politics/recycled/2009/07/why_did_it_take_google_so_long_to_take_gmail_out_of_beta.html

Letuchy, E. Facebook Chat. https://www.facebook.com/note.php?note_id=14218138919

Ligus, S. Effective Monitoring & Alerting. https://www.safaribooksonline.com/library/view/effective-monitoring-and/9781449333515/ch01.html

Lloyd Pearson, B. Who's using Docker? https://opensource.com/business/14/7/docker-through-hype

Lyndsay, J. Why exploration has a place in any strategy. http://www.workroom-productions.com/papers/Exploration%20and%20Strategy.pdf

Marick, B. Classic Testing Mistakes. http://www.exampler.com/testing-com/writings/classic/mistakes.html

Marick, B. Pair Testing. http://www.exampler.com/testing-com/test-patterns/patterns/XT-Pattern-jb.pdf

McConnell, S. Less is More: Jumpstarting Productivity with Small Teams. http://www.stevemcconnell.com/articles/art06.htm

McIndoe, J. A visual test strategy. http://assurity.co.nz/community/big-thoughts/a-visual-test-strategy/

McIndoe, J. Testing Stuff - A One-Page Test Strategy. https://technology.fairfaxmedia.co.nz/testing-stuff-a-one-page-test-strategy/

Miller, D. We Invite Everyone at Etsy to Do an Engineering Rotation: Here's why. https://codeascraft.com/2014/12/22/engineering-rotation/

Miller, G. A Test Strategy Retrospective Experience. https://softwaretestkitchen.com/2015/01/27/a-test-strategy-retrospective-experience/

Mitchell, J. How Do Facebook And Google Manage Software Releases Without Causing Major Problems? http://www.forbes.com/sites/quora/2013/08/12/how-do-facebook-and-google-manage-software-releases-without-causing-major-problems/#5a2c78c1c0cb

Murgia, M. Spotify crosses 100m users. http://www.telegraph.co.uk/technology/2016/06/20/spotify-crosses-100m-users/

Neate, T. Bug Bash: The Game. Testing Trapeze Magazine 2016, December, pg.3-8. http://www.testingtrapezemagazine.com/wp-content/uploads/2016/12/TestingTrapeze-2016-December-v2.pdf

Nelson-Smith, S. What Is This Devops Thing, Anyway?
http://www.jedi.be/blog/2010/02/12/what-is-this-devops-thing-anyway/

Nolan, D. Say "NO!" to a Staging environment.
Presented at London Continuous Delivery Lightning Talks, London, 2015.
https://vimeo.com/148161104

Novet, J. Etsy files to go public with $100M IPO.
http://venturebeat.com/2015/03/04/etsy-files-to-go-public-with-100m-ipo/

Obasanjo, D. Dark Launches, Gradual Ramps and Isolation: Testing the Scalability of New Features on your Web Site.
http://www.25hoursaday.com/weblog/2008/06/19/DarkLaunchesGradualRampsAndIsolationTestingTheScalabi

Ohanian, C. Unleash Your Creativity and Become a Better Tester.
Presented at European Testing Conference, Bucharest, 2016.
https://www.youtube.com/watch?v=m-noJQvR4gk

Painter, S. Making theguardian.com work best for you.
https://www.theguardian.com/info/2015/sep/22/making-theguardiancom-work-best-for-you

Pal, T. DevOps at Capital One: Focusing on Pipeline and Measurement.
Presented at DevOps Enterprise Summit (DOES16), San Francisco, 2016.
https://www.youtube.com/watch?v=6Q0mtVnnthQ

Papadimoulis, A. I've got the monkey now.
http://thedailywtf.com/articles/Ive-Got-The-Monkey-Now

Patton, R. Software Testing, 2nd ed.; Sams Publishing: Indianapolis, IN, 2005.

Paul, R. Exclusive: a behind-the-scenes look at Facebook release engineering.
https://arstechnica.com/business/2012/04/exclusive-a-behind-the-scenes-look-at-facebook-release-engineering/2/

Priestley, E. Is it true that Facebook has no testers?
https://www.quora.com/Is-it-true-that-Facebook-has-no-testers

Pyhäjärvi, M. Step up to the whiteboard.
http://visible-quality.blogspot.co.nz/2017/01/step-up-to-whiteboard.html

Raza, A. Puppet vs. Chef vs. Ansible vs. Saltstack.
http://www.intigua.com/blog/puppet-vs.-chef-vs.-ansible-vs.-saltstack

Rees-Carter, S. How To Configure Apache Using Ansible on Ubuntu 14.04.
https://www.digitalocean.com/community/tutorials/how-to-configure-apache-using-ansible-on-ubuntu-14-04

Ries, E. Work in small batches.
http://www.startuplessonslearned.com/2009/02/work-in-small-batches.html

Riley, C. Meet Infrastructure as Code.
https://devops.com/meet-infrastructure-code/

Rose, K. Failure Friday: How We Ensure PagerDuty is Always Reliable.
https://www.pagerduty.com/blog/failure-friday-at-pagerduty/

Sato, D. Canary Release.
https://martinfowler.com/bliki/CanaryRelease.html

Savoia, A. Test is Dead.
Presented at Google Test Automation Conference (GTAC), Mountain View, 2011.
https://www.youtube.com/watch?v=X1jWe5rOu3g

Schwaber, K.; Sutherland, J. The Definitive Guide to Scrum: The Rules of the Game, 1st ed.; ScrumInc: Cambridge, MA, 2013.

Schwarz, B. The Art of the Dojo.
https://codeascraft.com/2015/02/17/the-art-of-the-dojo/

Serban, I. Taking Control Of Your Test Environment.
Presented at Conference of the Association for Software Testing (CAST), Grand Rapids, 2015.
https://www.youtube.com/watch?v=ufZ4tDSgvv8

Serban, I. TestOps: Chasing the White Whale.
Presented at Codemotion Conference, Milan, 2016.
https://www.youtube.com/watch?v=I6A07ESTc_k

Shetzer Helfand, H. Dynamic Reteaming; LeanPub: Vancouver, BC, 2017.

Singh, R. Docker at Spotify - Twitter University.
https://www.youtube.com/watch?v=pts6F00GFuU

Singh, R. Docker at Spotify.
Presented at DockerCon, San Francisco, 2014.
https://www.youtube.com/watch?v=Tlgoq9t95ew

Sitakange, J. Infrastructure as Code: A Reason to Smile.
https://www.thoughtworks.com/insights/blog/infrastructure-code-reason-smile

Skelton, M. Why and How to Test Logging.
https://www.infoq.com/articles/why-test-logging

Skelton, M. Why and how to test logging.
Presented at DevOps Showcase North, Manchester, 2015.
http://www.slideshare.net/SkeltonThatcher/why-and-how-to-test-logging-devops-showcase-north-feb-2016-matthew-skelton

Solon, O. Should Tesla be ‘beta testing’ autopilot if there is a chance someone might die?
https://www.theguardian.com/technology/2016/jul/06/tesla-autopilot-fatal-crash-public-beta-testing

Stevenson, J. MEWT4 Post #1 - Sigh, It’s That Pyramid Again – Richard Bradshaw.
http://steveo1967.blogspot.co.nz/2015/10/mewt4-post-1-sigh-its-that-pyramid.html

Sundman, Y. Candy Crush Soda Saga delivery pipeline.
https://techblog.king.com/candy-crush-soda-saga-delivery-pipeline/

Sussman, N. Abandoning the Pyramid Of Testing in favor of a Band-Pass Filter model of risk management.
http://infiniteundo.com/post/158179632683/abandoning-the-pyramid-of-testing-in-favor-of-a

Sussman, N. Configuration Flags: A Love Story.
https://www.stickyminds.com/article/configuration-flags-love-story?page=0%2C1

Sussman, N. Continuous Improvement: How rapid release cycles alter QA and testing. Presented at DevOps Day, Mountain View, 2012.
https://www.slideshare.net/noahsussman/continuous-improvement-devops-day-mountain-view-2012

Sussman, N. Why does Etsy care so much about automated software testing?
https://www.quora.com/Why-does-Etsy-care-so-much-about-automated-software-testing

Sussna, J. Cloud and DevOps: A Marriage Made in Heaven.
https://www.infoq.com/articles/cloud-and-devops

Tehranian, D. Testing Ansible Roles with Test Kitchen.
https://dantehranian.wordpress.com/2015/06/18/testing-ansible-roles-with-test-kitchen/

Thorpe, J. Failure is inevitable but you don't need to be afraid.
https://blog.microfocus.com/failure-is-inevitable-but-you-dont-need-to-be-afraid/

Tiwari, A. Decoupling Deployment and Release - Feature Toggles.
https://abhishek-tiwari.com/post/decoupling-deployment-and-release-feature-toggles

Townsend, K. A DevOps dashboard for all: Capital One's Hygieia project offers powerful open source resource.
http://www.techrepublic.com/article/a-devops-dashboard-for-all-capital-ones-hygieia-project-offers-powerful-open-source-resource/

Tyley, R. Prout: is your pull request out?
https://www.theguardian.com/info/developer-blog/2015/feb/03/prout-is-your-pull-request-out

Vanian, J. How Spotify is ahead of the pack in using containers.
https://gigaom.com/2015/02/22/how-spotify-is-ahead-of-the-pack-in-using-containers/

Weber, B. A/B Testing using Google's Staged Rollouts.
https://blog.twitch.tv/a-b-testing-using-googles-staged-rollouts-ea860727f8b2#.8posd2h13

White, S K. Failure Fridays helps one company build a better customer experience.
http://www.cio.com/article/3084262/relationship-building-networking/failure-fridays-helps-one-company-build-a-better-customer-experience.html

Wigmore, I. What is two pizza rule?
http://whatis.techtarget.com/definition/two-pizza-rule

Wills, P.; Hildrew, S. Delivering Continuous Delivery, continuously.
https://www.theguardian.com/info/developer-blog/2015/jan/05/delivering-continuous-delivery-continuously

Winch, J. Testing in Production: rethinking the conventional deployment pipeline.
https://www.theguardian.com/info/developer-blog/2016/dec/20/testing-in-production-rethinking-the-conventional-deployment-pipeline

Yigal, A. How DevOps is Killing QA.
https://devops.com/devops-killed-qa/

5 Things About Configuration Management Your Boss Needs To Know.
https://www.upguard.com/blog/5-configuration-management-boss

About Capital One Financial Services: Company Overview, History, and Culture.
https://www.capitalone.com/about/

About Etsy.
https://www.etsy.com/nz/about/

About Modules.
http://docs.ansible.com/ansible/modules.html

apache2 Cookbook.
https://supermarket.chef.io/cookbooks/apache2

APKs and Tracks.
https://developers.google.com/android-publisher/tracks

Building A Simple Module.
http://docs.ansible.com/ansible/dev_guide/developing_modules_general.html

Connecting to a device using Spotify Connect.
https://support.spotify.com/no/article/spotify-connect/

GitHub - capitalone/Hygieia.
https://github.com/capitalone/Hygieia

GitHub - Netflix/chaosmonkey.
https://github.com/Netflix/chaosmonkey

GitHub - Netflix/SimianArmy.
https://github.com/Netflix/SimianArmy

GitHub - spotify/helios.
https://github.com/spotify/helios

Historic Timeline - BNZ Heritage.
https://www.bnzheritage.co.nz/archives/historic-timeline

How do you put on a coding dojo event?
https://www.youtube.com/watch?v=gav9fLVkZQc

Lean Coffee.
http://leancoffee.org/

Music for everyone - Spotify.
http://www.spotify.com

PagerDuty | The Incident Resolution Platform.
https://www.pagerduty.com/

PagerDuty Customer Case Studies.
https://www.pagerduty.com/customers/

States Tutorial, Part 1 - Basic Usage.
https://docs.saltstack.com/en/latest/topics/tutorials/states_pt1.html

The Guardian's losses mount.
https://www.economist.com/news/business/21703264-newspaper-may-be-edging-towards-asking-readers-pay-its-content-guardians-losses

What is a Container.
https://www.docker.com/what-container

Wikipedia - Comparison of open-source configuration management software.
https://en.wikipedia.org/wiki/Comparison_of_open-source_configuration_management_software

Wikipedia - DevOps.
https://en.wikipedia.org/wiki/DevOps

Wikipedia - Operating-system-level virtualization.
https://en.wikipedia.org/wiki/Operating-system-level_virtualization

Wikipedia - Pendulum.
https://en.wikipedia.org/wiki/Pendulum

About the Author

Katrina Clokie is a Test Practice Manager in Wellington, New Zealand. She is an active contributor to the international testing community as the founder and editor of Testing Trapeze magazine[201], a co-founder of WeTest New Zealand[202], an international conference speaker, frequent blogger and tweeter.

You can connect with Katrina via Twitter[203], Blogger[204], or LinkedIn[205].

[201] http://www.testingtrapezemagazine.com/
[202] http://wetest.co.nz
[203] https://twitter.com/katrina_tester
[204] http://katrinatester.blogspot.co.nz
[205] https://www.linkedin.com/in/katrina-clokie/