DevOps - introduction

Agenda

1. Why DevOps?
2. DevOps principles
3. DevOps concepts
4. DevOps practices
5. DevOps people
6. DevOps controls
7. Conclusion

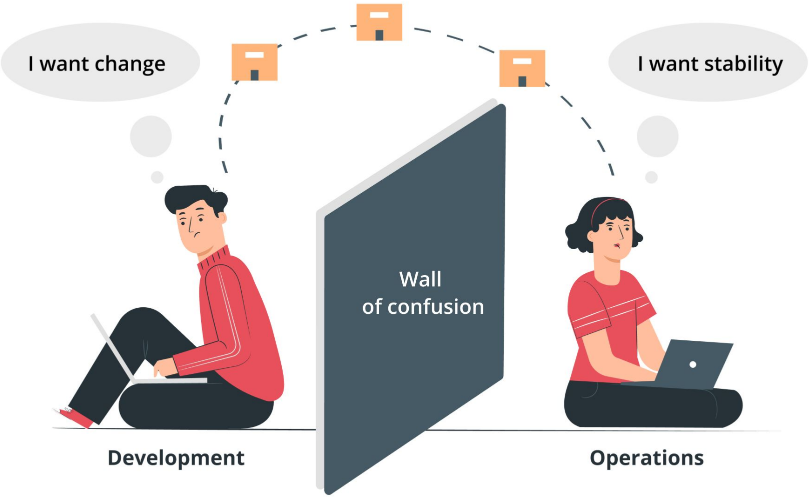
Why DevOps?

To reduce deployment lead time to minutes!

- The business demands faster and continuous delivery.
- Most organisations are not able to deploy production changes in minutes or hours, instead requiring weeks or months.
- Opposing goals between Development and Operations: Conflict between agile development (urgent projects) and stable operation (keep it running, don’t mess with the environment)

Typical problems between Dev and Ops

- Organizational silos
- Different mindsets
- Different tools
- Different environments
- Product back-logs
- Blame game
- Disintegrated processes
- Poor feedback loops

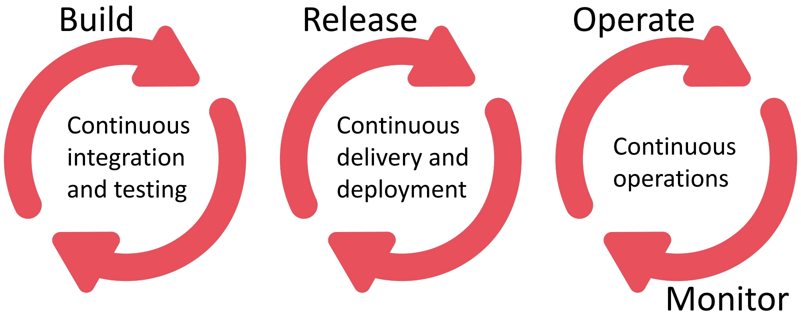


Leads to a downward spiral: Everybody gets a little busier, work takes a little more time, communications become a little slower, and work queues get a little longer. Work requires more communication, coordination, and approvals.

What is DevOps?

Tear down the wall between Development and Operations:

Continuous Delivery, Continuous Integration, Testing and Deployment



Build

Continuous integration and testing

Release

Continuous delivery and deployment

Operate

Continuous operations

Monitor

"Small teams independently implement their features, validate their correctness in production-like environments, and have their code deployed into production quickly, safely and securely."

The DevOps Handbook, by Gene Kim, Jez Humble, Patrick Debois and John Willis, 2016

What is DevOps?

“Continuous Delivery is about putting the release schedule in the hands of the business, not in the hands of IT. 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.

This in turn relies on comprehensive automation of the build, test and deployment process, and excellent collaboration between everyone involved in delivery – developers, testers, DBAs, systems administrators, users, and the business. In the world of Continuous Delivery, developers aren’t done with a feature when they hand some code over to testers, or when the feature is “QA passed”. They are done when it is working in production. That means no more testing or deployment phases, even within a sprint (if you’re using Scrum).”

Continuous Delivery, Reliable Software Releases through Build, Test, and Deployment Automation, by Jez Humble and David Farley, 2010

A brief history of DevOps

- DevOps relies on old and proven concepts such as lean, it service management, agile development, theory of constraints, resilience engineering, learning organisations, etc.
- At the 2008 Agile conference in Toronto, Canada, Patrick Debois and Andrew Schafer held a “birds of a feather” session on applying Agile principles to infrastructure as opposed to application code.
- Later, at the 2009 Velocity conference, John Allspaw and Paul Hammond gave the seminal “10 Deploys per Day: Dev and Ops Cooperation at Flickr” presentation.
- Patrick Debois was not there, but was so excited by Allspaw and Hammond’s idea that he created the first DevOps Days in Ghent, Belgium in 2009. There the term “DevOps” was coined.
- In 2010 Jez Humble and David Farley in their book “Continuous Delivery”, extended the concept to continuous delivery, which defined the role of a “deployment pipeline” to ensure that code and infrastructure are always in a deployable state, and that all code checked in to trunk can be safely deployed into production.
- In 2013 Gene Kim et. al. published their novel “The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win” that made DevOps commonly known in the industry.
- In 2016 Gene Kim, Jez Humble, Patrick Debois and John Willis published ”The DevOps Handbook”

Principles, Concepts, Practices and People

PRINCIPLES

1. Flow
2. Feedback
3. Continual learning and improvement

CONCEPTS

1. Product
2. Value stream
3. Loosely-coupled architecture
4. Autonomous teams

PRACTICES

1. Continuous integration
2. Fast and reliable automated testing
3. Continuous and automated deployment / provisioning
4. Measurement and feedback

PEOPLE

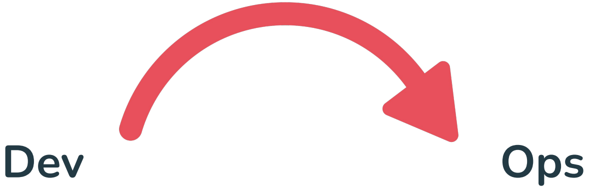
1. Customer centric
2. End-to-end responsibility
3. Collaboration
4. Learning and improvement
5. Experimentation and risk taking

DevOps Principles

The three DevOps Principles

1. Flow

Enable fast left-to-right flow of work from Development to Operations to the customer



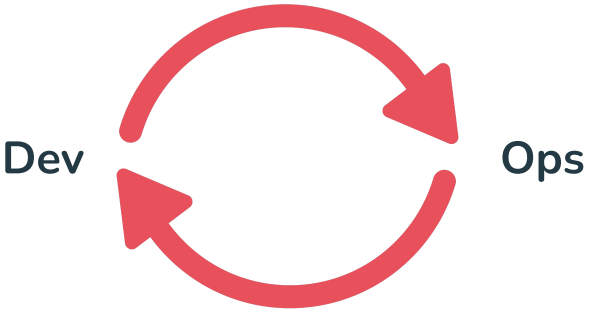
- Make work visible using visual boards
- Limit work in process (WIP)
- Reduce batch sizes
- Reduce the number of handovers
- Continually identify and elevate constraints
- Eliminate hardships and waste in the value stream

The three DevOps Principles

2. Feedback

Enable fast and constant flow of feedback from right to left at all value stream stages

- Establish fast feedback loops at every step of the process
- Establish pervasive production telemetry ensuring that problems are detected and corrected as they occur
- Keep pushing quality closer to the source
- Enable optimising for downstream work centers

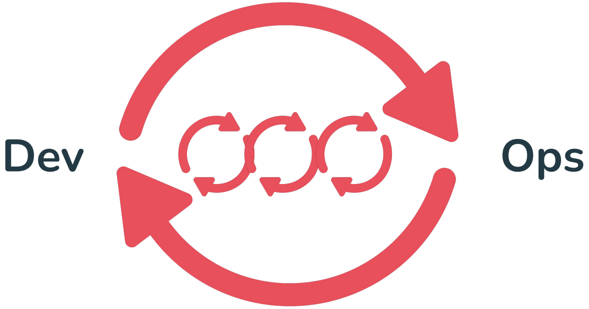


The three DevOps Principles

3. Continual learning & improvement

Enable a high-trust, experimenting and risk- taking culture as well as organisational learning, both from successes and failures

- Enabling organisational learning
- Institutionalise the improvement of daily work
- Transform local discoveries into global improvements
- Inject resilience patterns into daily work
- Encourage leaders to reinforce a learning culture
- Experiment, fail fast - *If it hurts, do it more often*



DevOps Concepts

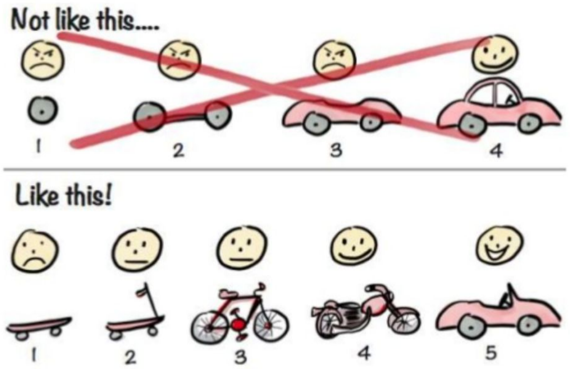
Product

Our application is our product. Strive towards delivering a minimum viable product (MVP), the smallest amount of functionality having value to the customer, as fast and reliably as it can.

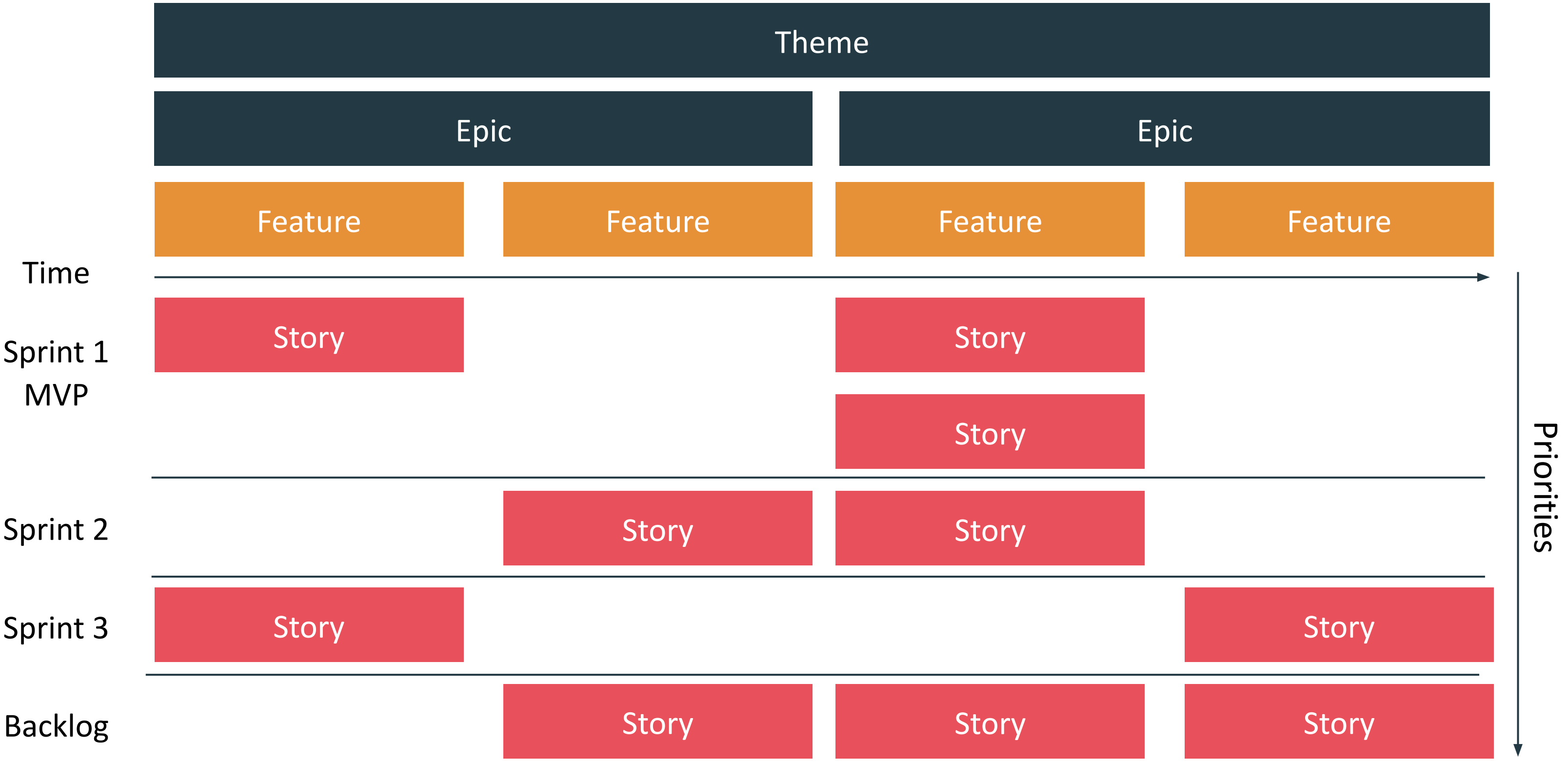
The minimum viable product (MVP) reflects the end-product in a minimal functional form. It is used to test new ideas and verify whether the hypothesis are correct.

"The minimum viable product (MVP) is that product which has just those features and no more that allows you to ship a product that early adopters see and, at least some of whom resonate with, pay you money for, and start to give you feedback on"

The Lean Startup, by Eric Ries, 2011



Product



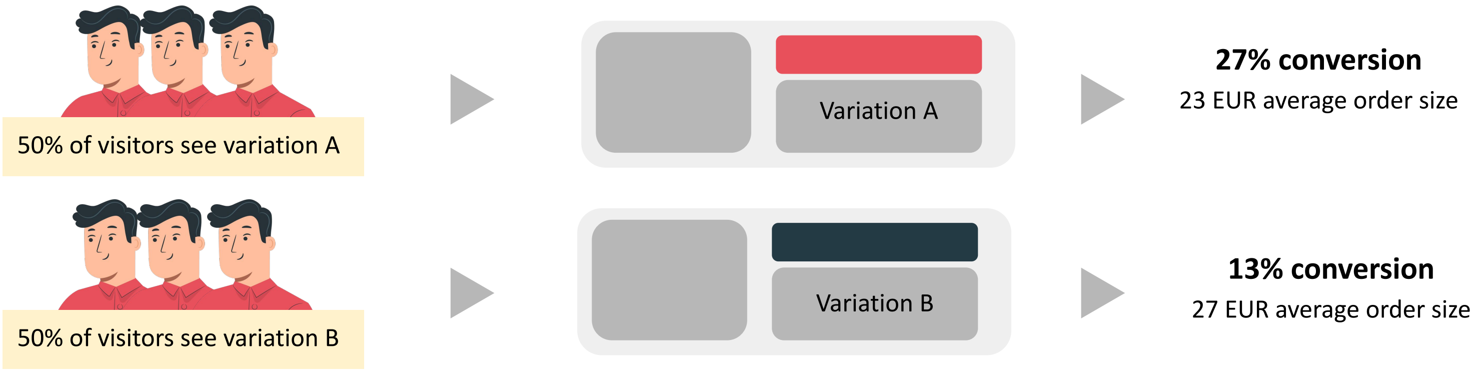
Time

	Theme			
	Epic		Epic	
	Feature	Feature	Feature	Feature
Sprint 1 MVP	Story		Story Story	
Sprint 2		Story	Story	
Sprint 3	Story			Story
Backlog		Story	Story	Story

Priorities

Product

- Create with the End in Mind: Understanding the real needs of customers.
- Before we build a feature, we should rigorously ask ourselves, “Should we build it, and why?”
- We should then perform the cheapest and fastest experiments possible to validate through user research whether the intended feature will actually achieve the desired outcomes.
- Think about each feature as a hypothesis and use production releases as experiments with real users to prove or disprove that hypothesis.
- We can use techniques such as hypothesis-driven development, customer acquisition funnels, and A/B testing.



50% of visitors see variation A

Variation A

27% conversion

23 EUR average order size

50% of visitors see variation B

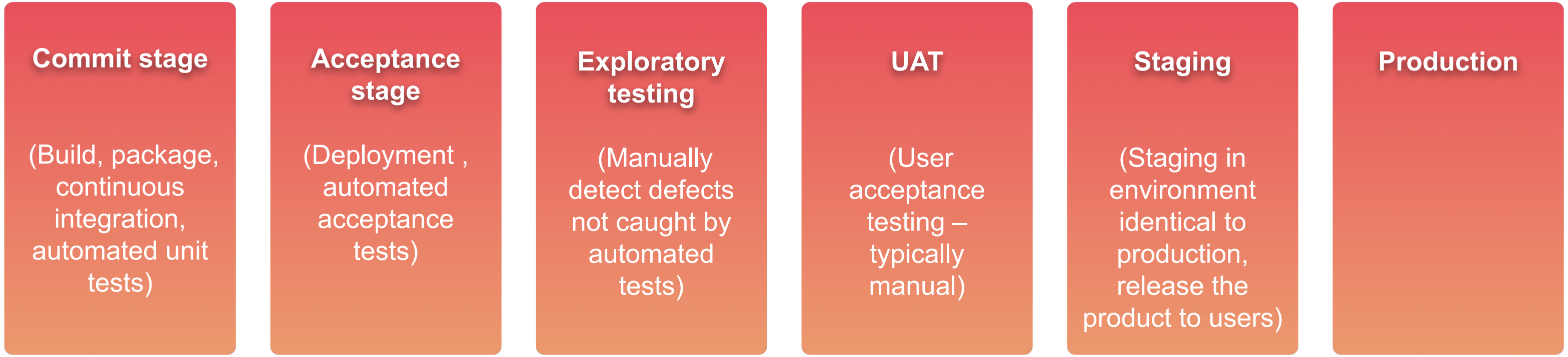
Variation B

13% conversion

27 EUR average order size

Value stream

The series of events that take a feature from code commit through to production.



Commit stage

(Build, package, continuous integration, automated unit tests)

Acceptance stage

(Deployment , automated acceptance tests)

Exploratory testing

(Manually detect defects not caught by automated tests)

UAT

(User acceptance testing – typically manual)

Staging

(Staging in environment identical to production, release the product to users)

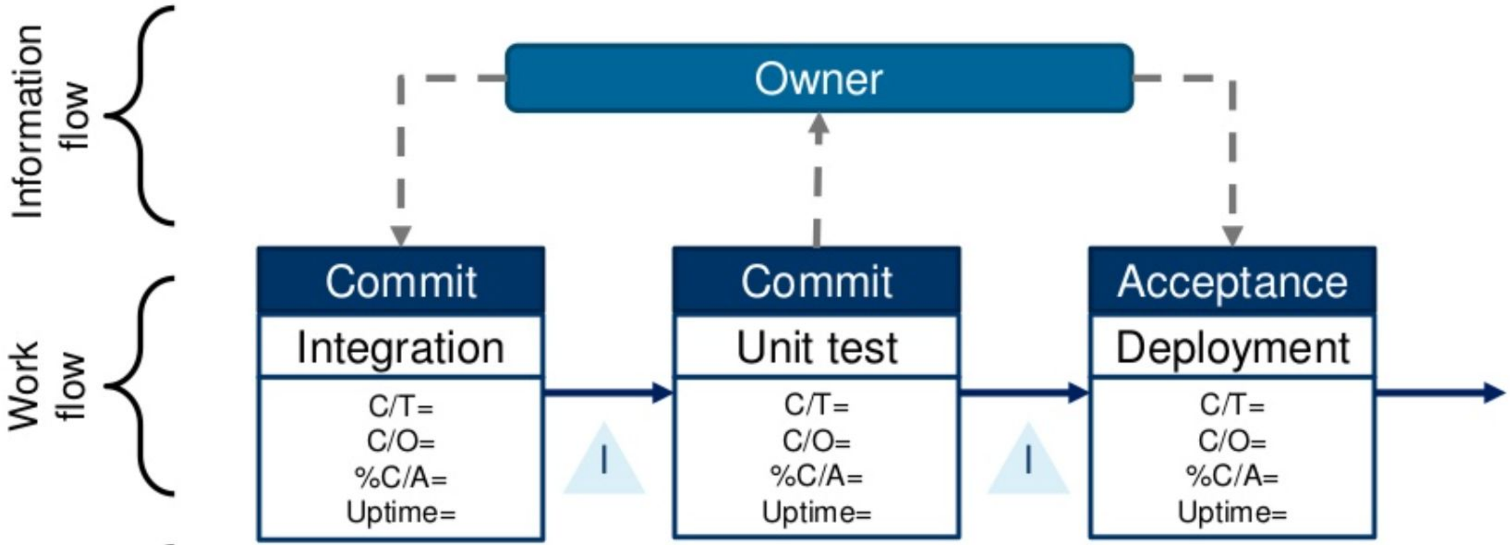
Production

The deployment pipeline, from Continuous Delivery

Use production-like environments at every stage of the value stream

Enable on-demand creation of Dev, Test, and Production environments. Environments must be created in an automated manner, ideally on demand from scripts and configuration information stored in version control, and entirely self-serviced, without any manual work required from Operations

Value stream mapping



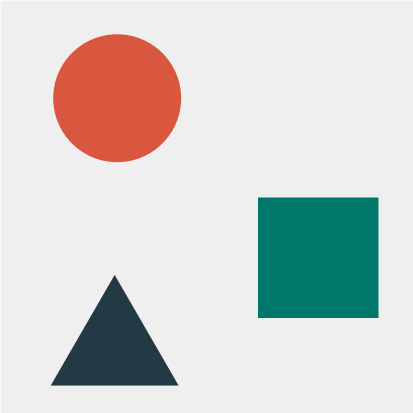
Loosely-coupled architecture

DevOps architecture principles

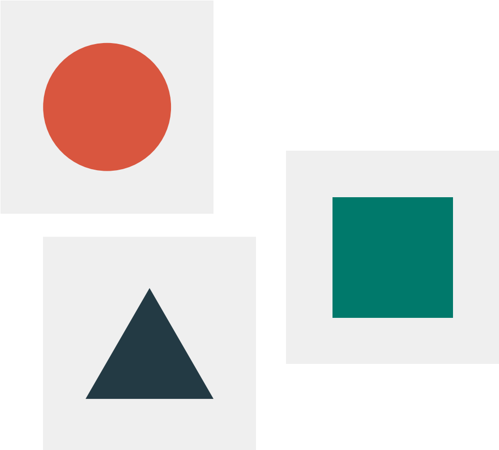
- **Componentization via Services:** It is independently deployable, cloud-ready, and scalable.
- **Organized Around Business Capabilities:** Organizations are organized around business capabilities (Micro Service Architectures - MSAs)
- **Products not Projects:** The characteristic appreciates “You build it, you run it.” This involves taking full responsibility.
- **Smart Endpoints and Dumb Pipes:** These are simple interfaces having no logic in between, such as the Enterprise Service Bus.
- **Decentralized Governance:** There is no standardization on a single technology platform.
- **Decentralized Data Management:** Transactions help with consistency but imposes temporal coupling.
- **Infrastructure Automation:** Automated tests and automated deployments.
- **Design for Failure Tolerance:** Any service call can fail due to unavailability of the supplier. Therefore, the client has to respond to this as gracefully as possible, detect the failures quickly, and restore the service.
- **Evolutionary Design:** The design supports independent replacement and upgradeability.

Loosely-coupled architecture

From monolith to microservices



Monolithic architecture



Microservices architecture

Loosely-coupled architecture

DevOps architecture archetypes

	Pros	Cons
Monolithic v1 (All functionality in one application)	• Simple at first • Low inter-process latencies • Single codebase, one deployment unit • Resource-efficient at small scale	• Coordination overhead increases as team grows • Poor enforcement of modularity • Poor scaling • All-or-nothing deploy (downtime failures) • Long build times
Monolithic v2 (Sets of monolithic tiers: “front end presentation”, “application server”, “database layer”)	• Simple at first • Join queries are easy • Single schema deployment • Resource-efficient at small scale	• Tendency for increased coupling over time • Poor scaling and redundancy (all or nothing, vertical only) • Difficult to tune properly • All-or-nothing schema management
Microservice (Modular, independent, graph relationship vs. tiers, isolated persistence)	• Each unit is simple • Independent scaling and performance • Independent testing and deployment • Can optimally tune performance (caching, replication, etc.)	• Many cooperating units • Many small repositories • Requires more sophistically tooling and dependency management • Network latencies

Infrastructure as Code

Infrastructure as code, also referred to as IaC, is a type of IT setup wherein developers or operations teams automatically manage and provision the technology stack for an application through software, rather than using a manual process to configure discrete hardware devices and operating systems. Infrastructure as code is sometimes referred to as programmable or software-defined infrastructure.

The concept of infrastructure as code is similar to programming scripts, which are used to automate IT processes. However, scripts are primarily used to automate a series of static steps that must be repeated numerous times across multiple servers. Infrastructure as code uses higher-level or descriptive language to code more versatile and adaptive provisioning and deployment processes. For example, infrastructure-as-code capabilities included with Ansible, an IT management and configuration tool, can install MySQL server, verify that MySQL is running properly, create a user account and password, set up a new database, and remove unneeded databases.

Infrastructure as Code

Application Deployment

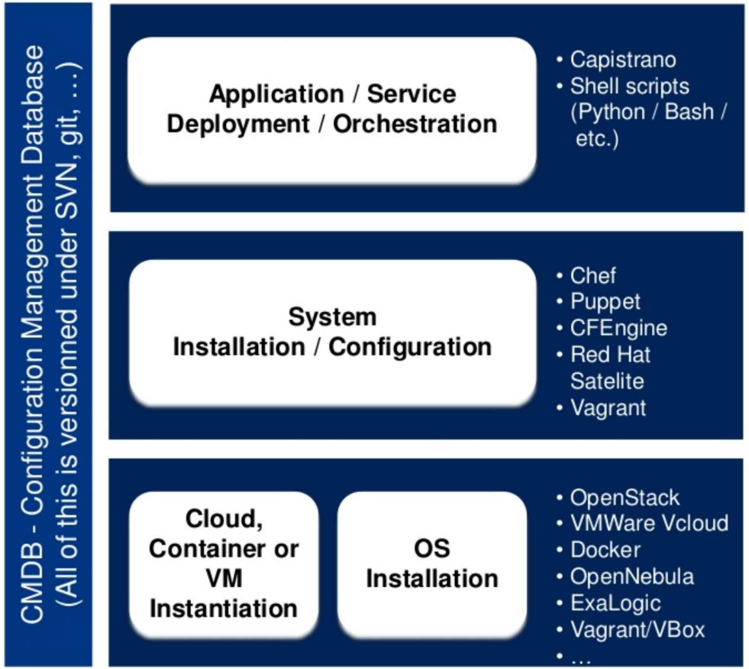
- Deploy application code … and rollback
- Configure resources (RDBMS, etc.)
- Start applications
- Join clusters

System Configuration

- JVM, app servers …
- Middlewares …
- Service configuration (logs, ports, user / groups, etc.
- Registration to supervision

Machine Installation

- Virtualization
- Self-Service environments



CMDB - Configuration Management Database
(All of this is versionned under SVN, git, …)

Application / Service Deployment / Orchestration

- Capistrano
- Shell scripts (Python / Bash / etc.)

System Installation / Configuration

- Chef
- Puppet
- CFEngine
- Red Hat Satelite
- Vagrant

Cloud, Container or VM Instantiation

OS Installation

- OpenStack
- VMWare Vcloud
- Docker
- OpenNebula
- ExaLogic
- Vagrant/VBox
- …

Benefits of infrastructure as Code

Software developers can use code to provision and deploy servers and applications, rather than rely on system administrators in a DevOps environment. A developer might write an infrastructure-as-code process to provision and deploy a new application for quality assurance or experimental deployment before operations takes over for live deployment in production.

With the infrastructure setup written as code, it can go through the same version control, automated testing, and other steps of a continuous integration and continuous delivery (CI/CD) pipeline that developers use for application code. An organization may choose to combine infrastructure as code with containers, which abstract the application from the infrastructure at the operating-system level. Because the OS and hardware infrastructure is provisioned automatically and the application is encapsulated atop it, these technologies prove complementary for diverse deployment targets, such as test, staging and production.

Despite its benefits, infrastructure as code poses potential disadvantages. It requires additional tools, such as a configuration management system, that could introduce learning curves and room for error. Any errors can proliferate quickly through servers, so it is essential to monitor Version control and perform comprehensive prerelease testing.

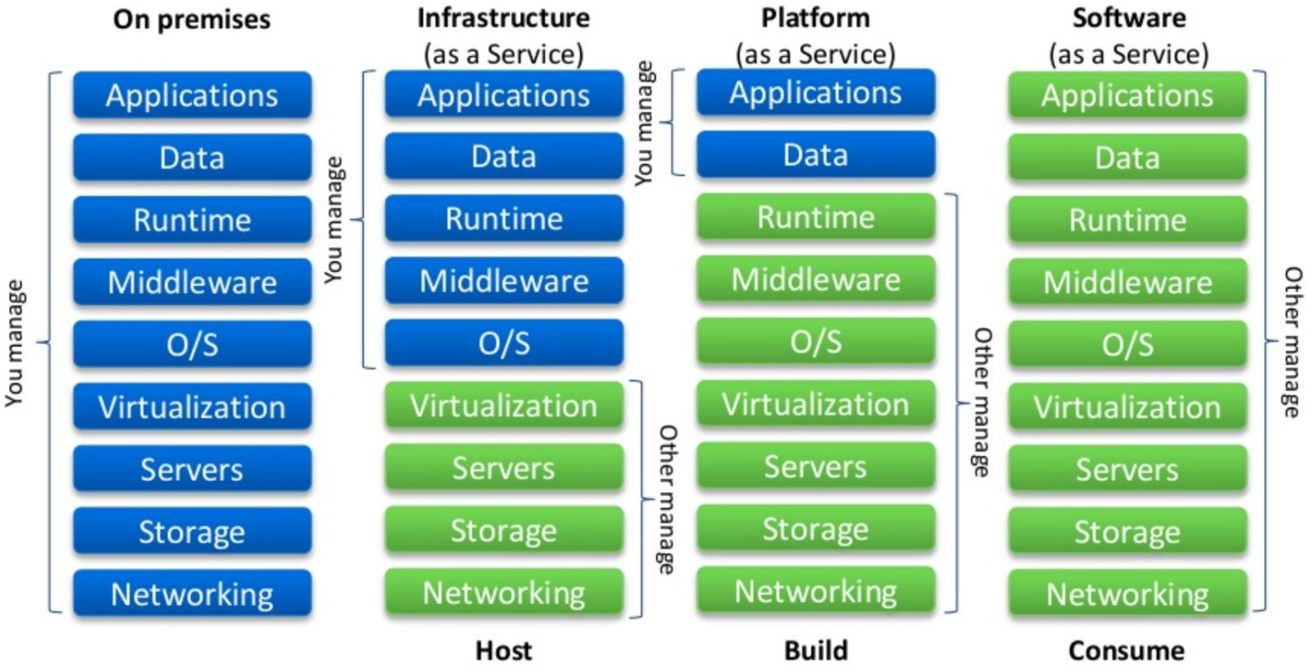
If administrators change server configurations outside of the set infrastructure-as-code template, there is potential for configuration drift. It's important to fully integrate infrastructure as code into systems administration, IT operations and DevOps practices with well-documented policies and procedures.

Loosely-coupled architecture

Good practices

- Automate building and configuring the environment using virtualized environments, environment creation processes that start from “bare metal”, “infrastructure as code” configuration management tools, automated operating system configuration tools, assembling an environment from a set of virtual images or containers, spinning up new environments in public cloud etc.
- Embrace ‘everything as code’ concepts. In the traditional operations space, more and more components can be defined with code, such as software-defined networks. Define more and more artefacts with code.
- Create our single repository of truth for the entire system. Put all application source files and configurations as well as all production artefacts in version control. Version control is for everyone in our value stream, including QA, Operations, Infosec, as well as developers
- Make infrastructure easier to rebuild than to repair. When we can quickly rebuild and re- create our applications and environments on demand, we can also quickly rebuild them instead of repairing them when things go wrong.
- Design a loosely-coupled architecture with well-defined interfaces that enforce how modules connect with each other to promote productivity and safety.

Loosely-coupled architecture



On premises	Infrastructure (as a Service)	Platform (as a Service)	Software (as a Service)
Applications	Applications	Applications	Applications
Data	Data	Data	Data
Runtime	Runtime	Runtime	Runtime
Middleware	Middleware	Middleware	Middleware
O/S	O/S	O/S	O/S
Virtualization	Virtualization	Virtualization	Virtualization
Servers	Servers	Servers	Servers
Storage	Storage	Storage	Storage
Networking	Networking	Networking	Networking
	Host	Build	Consume

On premises: You manage (Applications to Networking)

Infrastructure (as a Service): You manage (Applications to O/S); Other manage (Virtualization to Networking)

Platform (as a Service): You manage (Applications, Data); Other manage (Runtime to Networking)

Software (as a Service): Other manage (Applications to Networking)

Autonomous teams

"A team is a small number of people with complementary skills who are committed to a common purpose, set of performance goals, and approach for which they hold themselves mutually responsible."

The Wisdom of Teams, by Katzenbach & Smith, 1993

Cross-functional DevOps autonomous teams:

- consist of representatives from all disciplines fully accountable for developing and deploying an IT service
- are fully empowered and self-sufficient to design, build, test, deploy, and run the software
- operate autonomously and work independently from one another to deliver a continuous stream of software change
- have end-to-end responsibility. There is no handover or transfer of responsibility and accountability
- possess all the necessary expertise to take on the end-to-end responsibility
- needs team members with T-shaped profile and complementary skills
- provide support to products until end-of-life.
- are kept stable so they can keep iterating and improving and embed effective working habits.

Autonomous teams

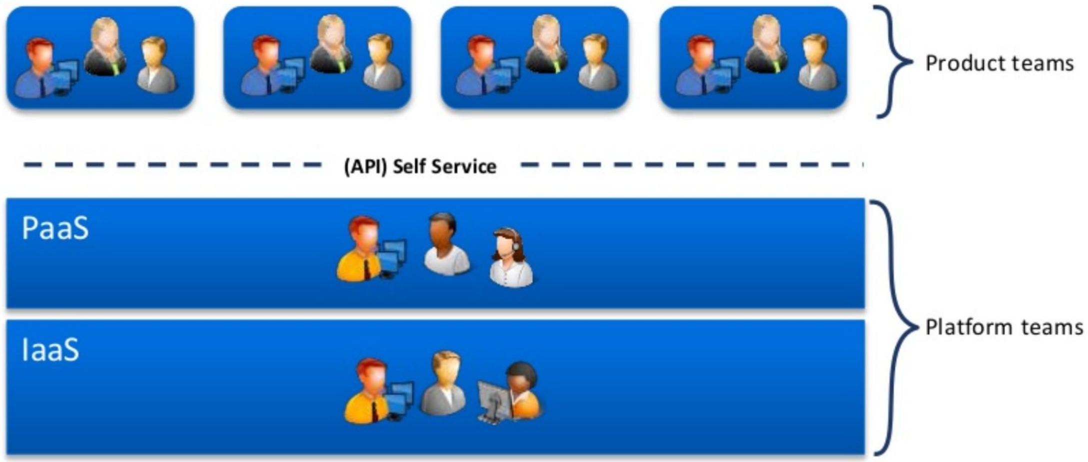
Good practices

- Think values over culture
- Share stories and experiences
- What kinds of problems are we trying to solve?
- Are we solving the right problems?
- Does our team have the necessary knowledge and experience to acknowledge the problem and to understand the repercussions that their potential solutions might have?

Product owner
Developer
Operations engineer
(Sys admin, middleware,
DBA, etc.)
QA / Tester
InfoSec
Release manager

Autonomous teams

Allow small product teams to work safely and architecturally decoupled from the work of other teams who use self-service platforms that leverage the collective experience of Operations and Information Security



Autonomous teams

Decoupling of Product and Platform teams:

- Interfaces between Product teams and Platform teams are clearly defined through Application Programming Interfaces (APIs).
- The Platform teams offer a rich set of standardized self-service capabilities to Product teams, such as logging, monitoring, deployment, backup, recovery and many more. Such a set of capabilities/services allows Product teams to completely manage their (software) products.
- The Product teams are the customers of the Platform teams. They use the products of the Platform teams. These products can be used as fully automated self services.
- The Platform teams are responsible for the qualities of their platform products, such as availability and performance. The Product teams are responsible for the qualities of their (application) products, such as availability and performance.
- The system qualities of the platform products can be monitored and managed independently from the availability of the application products, which use the platform products.

Autonomous teams

“Teams should only be as large as can be fed with two pizzas”

Conway’s Law, states that “organizations which design systems...are constrained to produce designs which are copies of the communication structures of these organizations….The larger an organization is, the less flexibility it has and the more pronounced the phenomenon.”

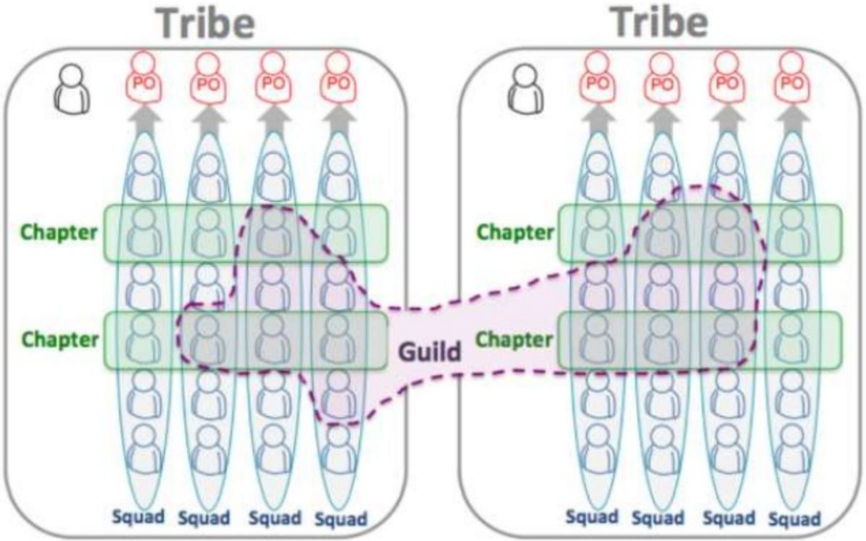
Source: Conway, Melvin E., 1968

Market-oriented organisations tend to be flat, composed of multiple, cross-functional disciplines (e.g., marketing, engineering, etc.), which often lead to potential redundancies across the organization. This is how many organisations adopting DevOps operate. Market-oriented teams are responsible not only for feature development, but also for testing, securing, deploying, and supporting their service in production, from idea conception to retirement. These teams are designed to be cross-functional and independent

Autonomous teams

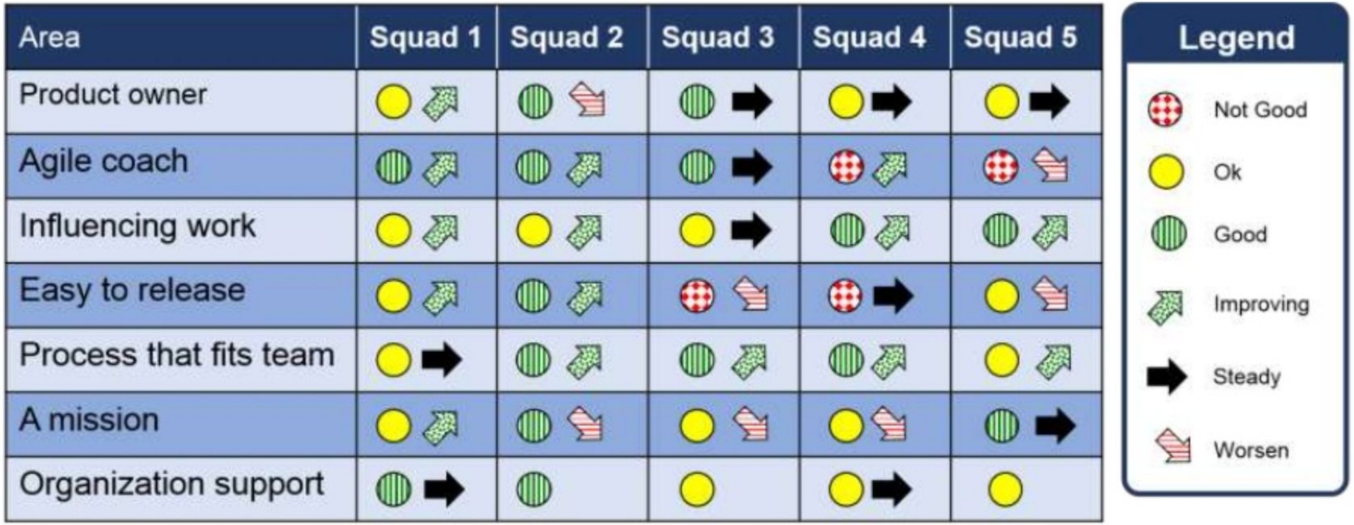
Teams of teams

Scaling with Tribes, Squads, Chapters & Guilds



Autonomous teams

Measuring team performance

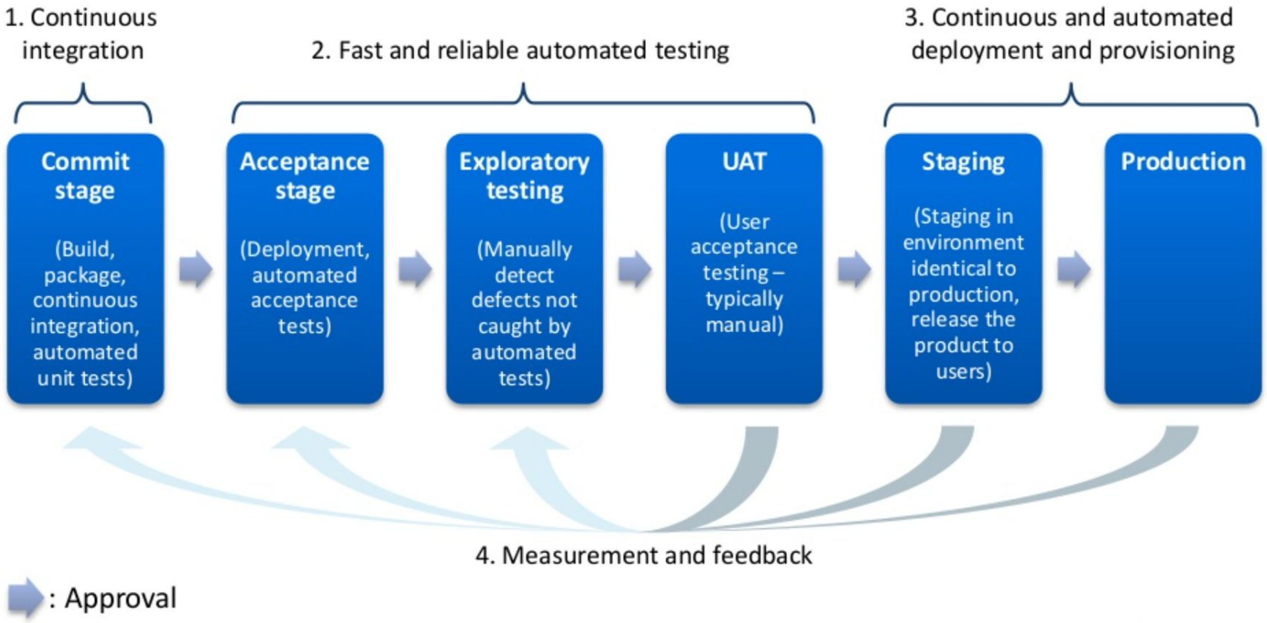


Area	Squad 1	Squad 2	Squad 3	Squad 4	Squad 5
Product owner	🟡 ↗	🟢 ↘	🟢 ➡	🟡 ➡	🟡 ➡
Agile coach	🟢 ↗	🟢 ↗	🟢 ➡	🔴 ↗	🔴 ↘
Influencing work	🟡 ↗	🟡 ↗	🟡 ➡	🟢 ↗	🟢 ↗
Easy to release	🟡 ↗	🟢 ↗	🔴 ↘	🔴 ➡	🟡 ↘
Process that fits team	🟡 ➡	🟢 ↗	🟢 ↗	🟢 ↗	🟡 ↗
A mission	🟡 ↗	🟢 ↘	🟡 ↘	🟡 ↘	🟢 ➡
Organization support	🟢 ➡	🟢	🟡	🟡 ➡	🟡

Legend	
🔴	Not Good
🟡	Ok
🟢	Good
↗	Improving
➡	Steady
↘	Worsen

DevOps Practices

DevOps practices



1. Continuous integration

2. Fast and reliable automated testing

3. Continuous and automated deployment and provisioning

Commit stage

(Build, package, continuous integration, automated unit tests)

Acceptance stage

(Deployment, automated acceptance tests)

Exploratory testing

(Manually detect defects not caught by automated tests)

UAT

(User acceptance testing – typically manual)

Staging

(Staging in environment identical to production, release the product to users)

Production

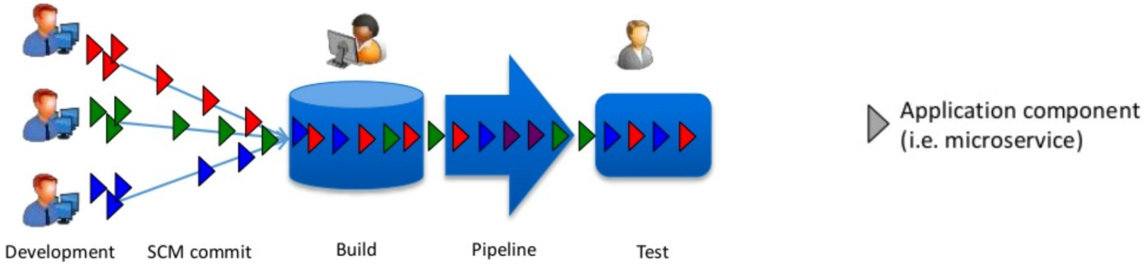
4. Measurement and feedback

➡ : Approval

Continuous integration

Continuous Integration (CI) is the practice, in software engineering, of merging all developer working copies to a shared mainline several times a day.

- Once a team member commits a number of code changes into Software Configuration Management, the changes can be merged automatically, analysed, compiled, unit tested, and assembled automatically.
- The automated build process can create a new deployment package and publish it to a trunk. In this manner, a continuous flow from code commit to validated deployment package can be implemented.
- The automated build process is important for a fail-fast strategy. For example, if there are code merge conflicts, the build should fail so the team members can fix the conflicts.



Continuous integration

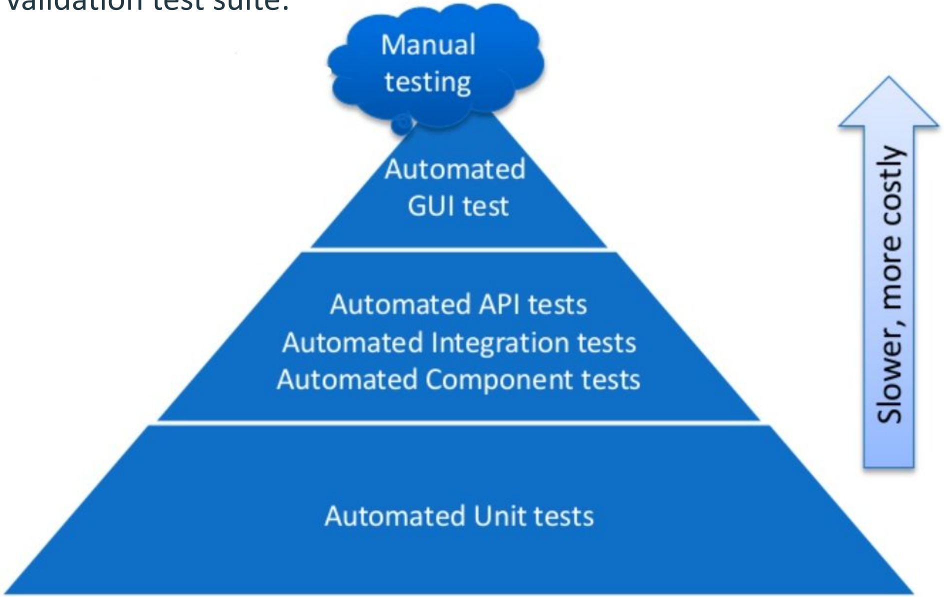
Good practices

- Adopt trunk-based development practices where all developers check in their code to trunk at least once per day.
- Frequent code commits to trunk means that all automated tests can be run on the software system as a whole and alerts received when a change breaks some other part of the application or interferes with the work of another developer.
- The discipline of daily code commits forces the developers to break the work down into smaller chunks while still keeping trunk in a working, releasable state

Fast and reliable automated testing

Build a fast and reliable automated validation test suite:

- Unit tests
- Acceptance tests
- Integration tests



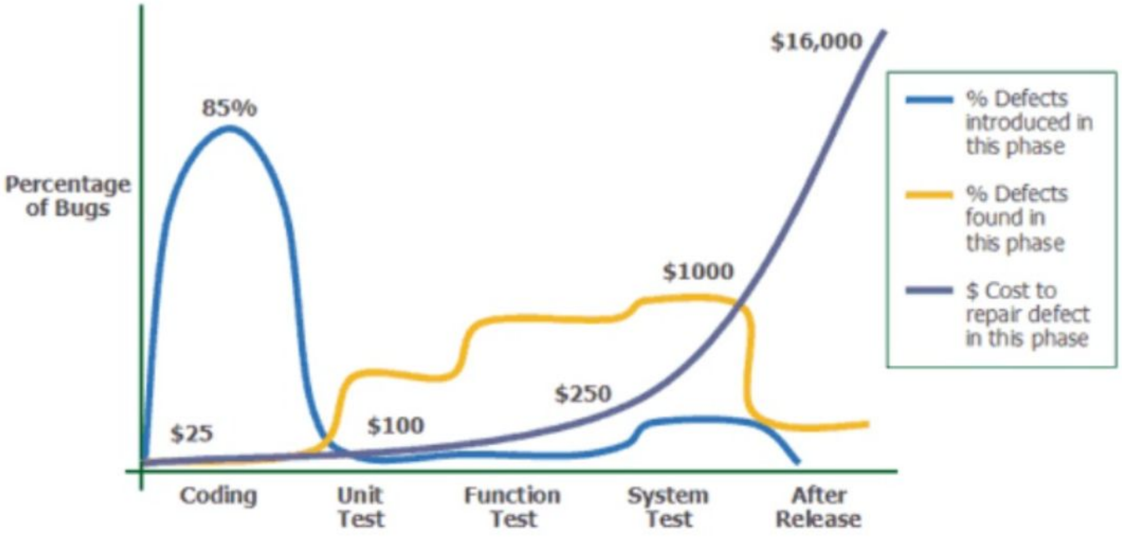
Fast and reliable automated testing

Good practices

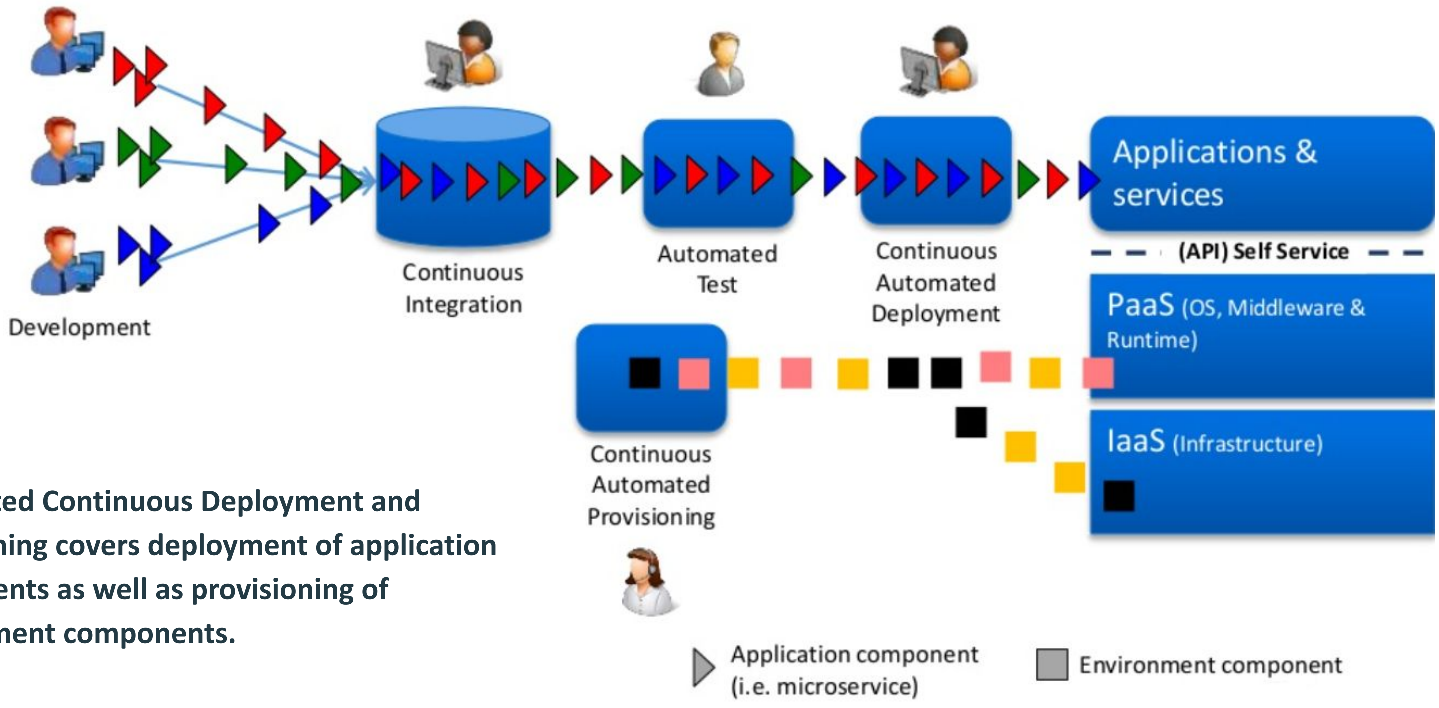
- Catch errors as early in the automated testing as possible
- Ensure tests run quickly (in parallel, if necessary)
- Write automated tests before writing the code ("Test-driven development")
- Automate as many manual tests as possible Integrate performance testing into the test suite
- Integrate non-functional requirements testing into the test suite
- Pull the andon cord when the deployment pipeline breaks

Fast and reliable automated testing

Catching errors as early in the testing as possible always pay back later in the process.



Automated continuous deployment & provisioning



Development
Continuous Integration
Automated Test
Continuous Automated Deployment
Applications & services
(API) Self Service
PaaS (OS, Middleware & Runtime)
IaaS (Infrastructure)
Continuous Automated Provisioning
Application component (i.e. microservice)
Environment component

Automated Continuous Deployment and Provisioning covers deployment of application components as well as provisioning of environment components.

Automated continuous deployment & provisioning

Automated continuous deployment automatically push application components into production when all tests are passed.

Application deployment implies:

- Installing applications
- Updating applications
- Configuring resources
- Configuring middleware components
- Starting/stopping components
- Configuring the installed application
- Configuration systems like load balancers, routers
- Verification of components

Automated continuous deployment & provisioning

Automated provisioning is defined as the fully automated deployment and maintenance of environment components.

- Application environment components are the deployment target containers of the application, such as a database server or an application runtime server.
- Infrastructure changes are viewed as code (Infrastructure as Code)
- Rather than provisioning and managing environment components manually, DevOps teams must be able to acquire new environment components on demand, fully automated.
- New environments are delivered within minutes/hours instead of weeks/months, hence, great speed
- Control, traceability of system changes, no manual changes, and a single, central source of truth

Automated continuous deployment & provisioning

Good practice

- Standardize the platforms and reduce the number of variations
- Automate the deployment and provisioning processes using the same deployment mechanism for every environment.
- Enable automated self-service deployments for both build, test and deployment.
- Automate as many of the manual steps as possible, such as:
    - Packaging code in ways suitable for deployment
    - Creating pre-configured virtual machine images or containers
    - Automating the deployment and configuration of middleware
    - Copying packages or files onto production servers
    - Restarting servers, applications, or services
    - Generating configuration files from templates
    - Running automated smoke tests to make sure the system is working and correctly configured
    - Running testing procedures
    - Scripting and automating database migrations

Automated continuous deployment & provisioning

Decouple deployments from releases

- *Deployment* is the installation of a specified version of software to a given environment
- *Release* is when we make a feature (or set of features) available to all our customers or a segment of customers

Automated continuous deployment & provisioning

Release patterns

Environment-based release patterns: This is where we have two or more environments that we deploy into, but only one environment is receiving live customer traffic

- The simplest pattern is called blue-green deployment. In this pattern, we have two production environments: blue and green. At any time, only one of these is serving customer traffic
- The canary release pattern automates the release process of promoting to successively larger and more critical environments as we confirm that the code is operating as designed. When something appears to be going wrong, we roll back; otherwise, we deploy to the next environment.
- The cluster immune system expands upon the canary release pattern by linking our production monitoring system with our release process and by automating the roll back of code when the user-facing performance of the production system deviates outside of a predefined expected range

Application-based release patterns: This is where we modify our application so that we can selectively release and expose specific application functionality by small configuration changes

- Introduce feature toggles, which provide us with the mechanism to selectively enable and disable features without requiring a production code deployment, e.g. to enable dark launches.

Measurement and feedback

Telemetry can be defined as an automated communications process by which measurements and other data are collected at remote points and are subsequently transmitted to receiving equipment for monitoring.

Pervasive production telemetry in both the code and production environment ensure that problems are detected and corrected quickly:

- In applications
- In the environment
- In the deployment pipeline

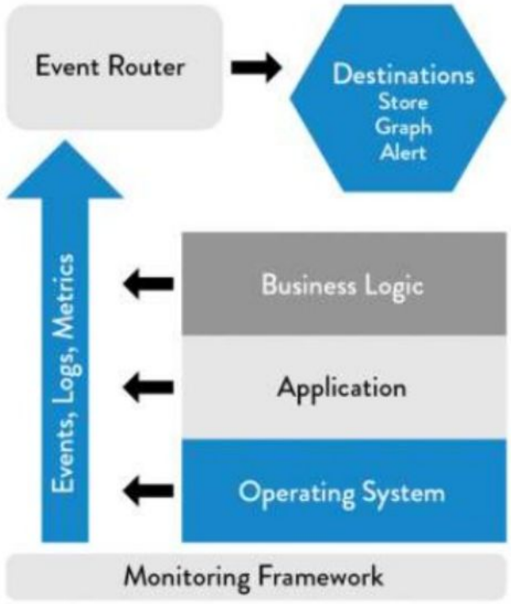
Key DevOps indicators

- Lead time (request to fulfilment)
- Process time (begin work to fulfilment)
- Percent complete and accurate (%C/A)

Measurement and feedback

Monitoring framework

- Collect data at the business logic, application, and environments layer
- In each of these layers, create telemetry in the form of events, logs, and metrics.
- Establish an event router responsible for storing events and metrics
- This capability enables visualization, trending, alerting anomaly detection, and so forth



Measurement and feedback

Good practice

Create telemetry to enable seeing and solving problems

- Create a centralized telemetry infrastructure
- Create application logging telemetry that helps production. Different logging levels, some of which may also trigger alerts, such as Debug, Info, Warning, Error, Fatal
- Create the infrastructure and libraries necessary to make it as easy as possible for anyone in Development or Operations to create telemetry for any functionality they build
- Create self-service access to telemetry and information radiators

Analyze telemetry to better anticipate problems and achieve goals

- Use means, standard deviation and anomaly detection techniques to detect potential problems.

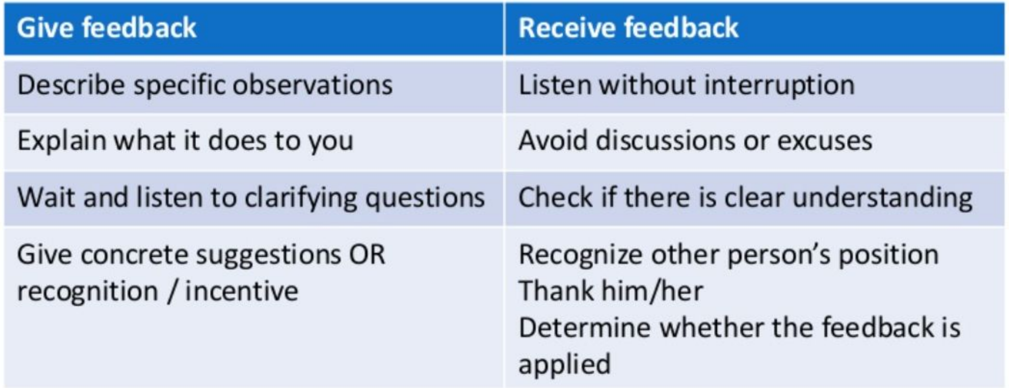
Enable feedback so Development and Operations can safely deploy code

- Use telemetry to make deployments safer
- Have developers follow work downstream. One of the most powerful techniques in interaction and user experience design (UX) is contextual inquiry. This is when the product team watches a customer use the application in their natural environment, often working at their desk.
- Have developers initially self-manage their production service

Measurement and feedback

Human feedback

The key tool that leaders should use and stimulate is human feedback. Giving and receiving feedback is the basis for all improvement and development of teamwork. It is vital for both leaders and team members to learn and practice giving and receiving feedback in a respectful manner.

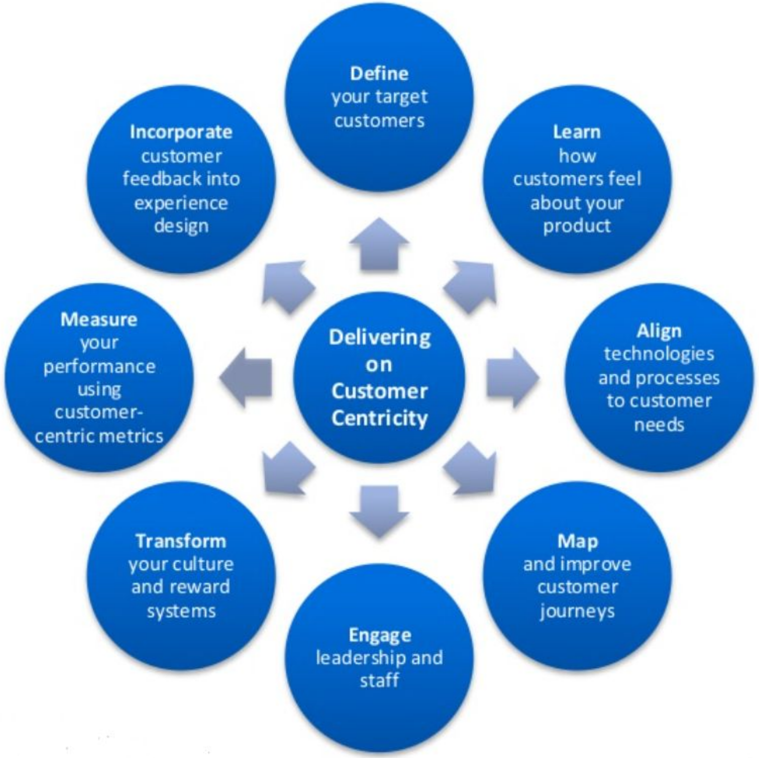


Give feedback	Receive feedback
Describe specific observations	Listen without interruption
Explain what it does to you	Avoid discussions or excuses
Wait and listen to clarifying questions	Check if there is clear understanding
Give concrete suggestions OR recognition / incentive	Recognize other person's position Thank him/her Determine whether the feedback is applied

DevOps People

Customer-centric

It is imperative to have short feedback loops with real customers and end-users. Therefore, all activities involved in building IT products and services should revolve around customers.


Delivering on Customer Centricity
Define your target customers
Learn how customers feel about your product
Align technologies and processes to customer needs
Map and improve customer journeys
Engage leadership and staff
Transform your culture and reward systems
Measure your performance using customer-centric metrics
Incorporate customer feedback into experience design

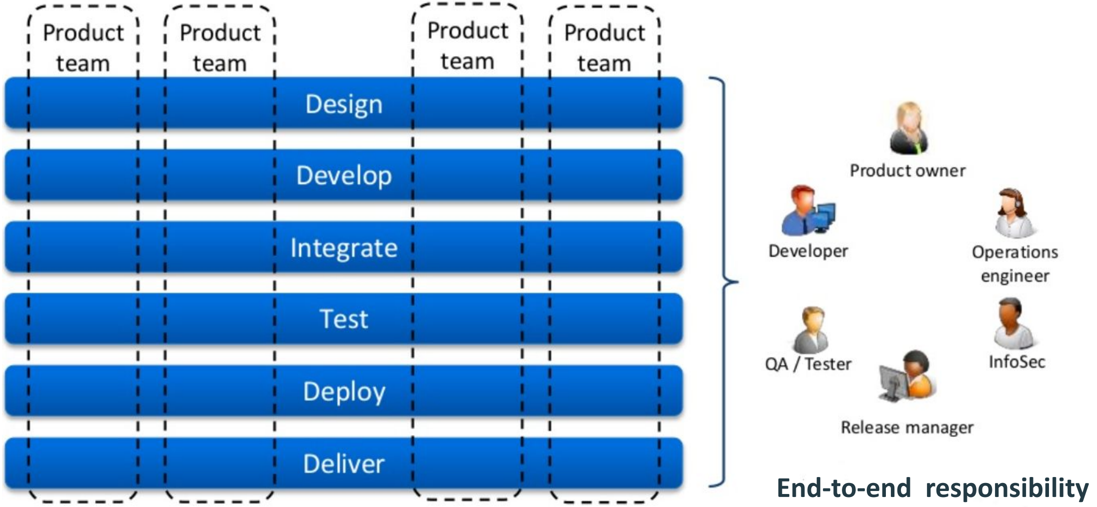
Customer-centric

Good practice

- Begin with end in mind, in retrospectives: ask people why?
- Start off with defining customer objectives (the Voice of Customer). A common understanding of the Voice of Customer (VoC) is important because in later stage you will determine with the team, which activities are really adding to this VoC and which steps are not.
- Encourage customers to attend demos, implement user feedback into a storyboard, allow customers to write about product and respond, organize people around the product, encourage the team to write blogs about product (you build it; you run it).

End-to-end responsibility

In a DevOps organization, teams are vertically organized so that they can be fully accountable for the products and services they deliver. End-to-end responsibility means that the team holds itself accountable for the quality and quantity of services it provides to its customers.



End-to-end responsibility

Good practice

Caring about the end-to-end responsibility might be the most crucial ingredient for DevOps. When people care and have the required skills, knowledge, and resources, they can and will collaborate to live up to their responsibility. If they care, they will learn, adapt, improve, and provide great services and value.

- All team members are responsible for the complete product, which includes the *full delivery cycle* as well as operating/providing customer support throughout the lifecycle of the product.
- *Quality* is built in from the initiation of the teams up to discharge. It is at the heart of every activity. It is never compromised. We value full transparency.
- Encourage the team to utilize their *skills* as they know the best; avoid cutting corners; practice automation (test, deploy, and provision), continuous improvement, and transparency (monitors).
- Do not explain 'how' to do, ask 'what' is required, address derailment openly, *let people figure out how to do things*, reward responsibility, reward failure, bring transparency in what everyone is doing.

Collaboration

The meaning of collaboration is working together to achieve a goal. It is the central theme for DevOps teams. Collaboration means shared organizational goals, empathy and learning between different people

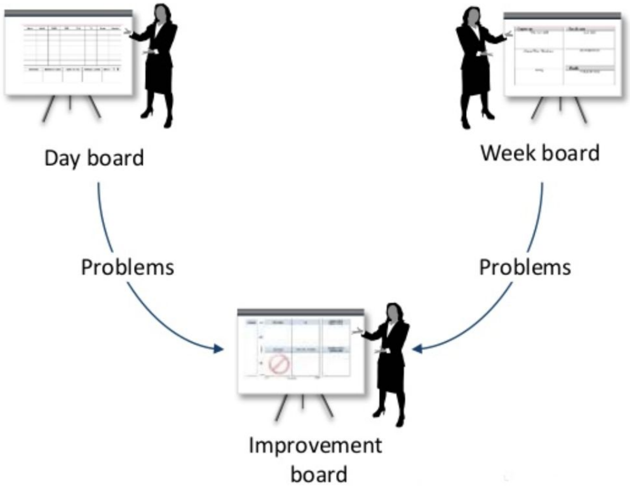
Collaboration

Good practice

Visual Management is one of the strongest tools to stimulate collaboration and ensures that pitfalls are uncovered. The tool helps ensure the work done by a team is constantly visible on boards including:

- Identify work and impediments.
- Communicate important information.
- Show how to perform a task.
- Show planning and priorities.

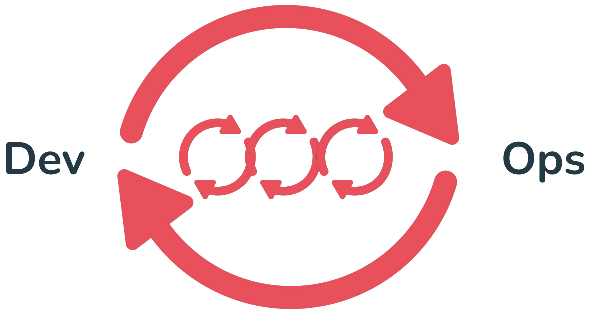
In its most basic form, Visual Management includes three boards, as shown in the figure.



Learning and improvement

When we work within a complex system, it is impossible for us to predict all the outcomes for the actions we take

To enable us to safely work within complex systems, our organizations must become ever better at self-diagnostics and self-improvement and must be skilled at detecting problems, solving them, and multiplying the effects by making the solutions available throughout the organization.



Learning and improvement

Good practice

Continuous improvement is about problem-solving:

- Seeing and prioritizing problems
  - Uncover problems
  - Accept problems as a part of daily life.
  - Initiate an action to identify the problems that need immediate solutions
- Solving problems
  - Invest time and other resources
  - Understand the root causes of problems
  - Resolve the problems completely
- Sharing lessons learned
  - Schedule blameless post-mortem meetings after accidents occur. The better question to focus on is, ‘Why did it make sense to me when I took that action?’”
  - Share the lessons learned with others in the IT organization, so they can benefit from it
- Decrease incident tolerances to find ever-weaker failure signals, by amplifying weak failure signals
- Because we care about quality, we even inject faults into our production environment to learn how our system fails in a planned manner

Learning and improvement

Good practice

- Create a single, shared source code repository for our entire organization
- Use chat rooms and chat bots to automate and capture organizational knowledge. Put automation tools into the middle of the conversation in chat rooms, helping create transparency and documentation.
- Reserve time to create organizational learning and improvement. Dedicated rituals for improvement work has also been called spring or fall cleanings and ticket queue inversion weeks. Other terms have also been used, such as hack days, hackathons, and 20% innovation time.
- Enable everyone to teach and learn
- Create internal consulting and coaches to spread practices
- Furthermore, everyone is constantly learning, fostering a hypothesis-driven culture where the scientific method is used to ensure nothing is taken for granted.

Learning and improvement

Good practice

- Automate standardized processes in software for re-use
- Design for operations through codified non-functional requirements
- Spread knowledge by using automated tests as documentation (test-driven development) and communities of practice
- Build reusable operations user stories into development
- Convert local discoveries into global improvements

Experimentation and risk taking

Experimentation is to test a hypothesis. In other words, trying something new based on a need is known as experimentation. DevOps teams must have the courage to experiment, with the potential of failure. They know how to fail fast, take the next step, or go back a couple of steps under time pressure to ensure there is quality at the source.

- Ensure customer buy-in
- Define and deliver a Minimum Viable Product (MVP)
- Focus on the goal “customer value is delivered first time, right in flow”
- Take small steps and do not carry out large experiments

THERE IS NO INNOVATION WITHOUT **EXPERIMENTATION**

Experimentation and risk taking

Good practice

- Create an environment in which people perform at best, where they feel inspired, where they want to be, feel welcomed and are encouraged to think out of the box.
- Provide time, use instant sandbox environment, remove hurdles, applaud ideas, fail safely. Award failure!
- Institute game days to rehearse failures. The goal for a game day is to help teams simulate and rehearse accidents to give them the ability to practice.
- Inject production failures to enable resilience and learning. For example Netflix has invented a surprising and audacious service called Chaos Monkey, which simulates server failures by constantly and randomly killing production servers
- Hire people with matching ambitions, move away from function-title model, support experimentation, support automating manual tasks, do not focus only on utilization.
- Provide good coffee, create nice and open environments, invest in additional facilities like games, conduct contests or arrange drinks at the end of the week, allow teams to modify/tailor the office to their needs.

DevOps Controls

Change management

Traditional change controls can lead to unintended outcomes, such as contributing to long lead times, and reducing the strength and immediacy of feedback from the deployment process.

Fearlessly cut bureaucratic processes.

Enable coordination and scheduling of changes. Even in a loosely-coupled architecture, when many teams are doing hundreds of independent deployments per day, there may be a risk of changes interfering with each other. To mitigate these risks, we may use chat rooms to announce changes and proactively find collisions that may exist.

Effective change management policies will recognize that there are different risks associated with different types of changes and that those changes are all handled differently. These processes are defined in ITIL, which breaks changes down into three categories:

- Standard changes
- Normal changes
- Urgent changes

Change management

Enable pair programming to improve all our changes. In one common pattern of pairing, one engineer fills the role of the driver, the person who actually writes the code, while the other engineer acts as the navigator, observer, or pointer, the person who reviews the work as it is being performed. Another pair programming pattern reinforces test-driven development (TDD) by having one engineer write the automated test and the other engineer implement the code.

Enable peer review of changes. Instead of requiring approval from an external body prior to deployment, require engineers to get peer reviews of their changes.

- Everyone must have someone to review their changes (e.g., to the code, environment, etc.) before committing to trunk.
- Everyone should monitor the commit stream of their fellow team members so that potential conflicts can be identified and reviewed.
- Define which changes qualify as high risk and may require review from a designated subject matter expert (e.g., database changes, security-sensitive modules such as authentication, etc.).
- If someone submits a change that you can’t understand its impact after reading through it a couple of times - it should be split up into multiple, smaller changes that can be understood at a glance.

Information security

Instead of inspecting security into our product at the end of the process, we will create and integrate security controls into the daily work of Development and Operations, so that security is part of everyone's job, every day

- Making security a part of everyone's job
  - Integrate security into development iteration demonstrations
  - Integrate security into defect tracking and post-mortems
- Integrating preventative controls into our shared source code repository
- Integrating security with the deployment pipeline
- Integrating security with the telemetry to better enable detection and recovery
  - Creating security telemetry in our applications
  - Creating security telemetry in our environment
- Ensure security of the application, the environment and the deployment pipeline
- Reduce reliance on separation of duty

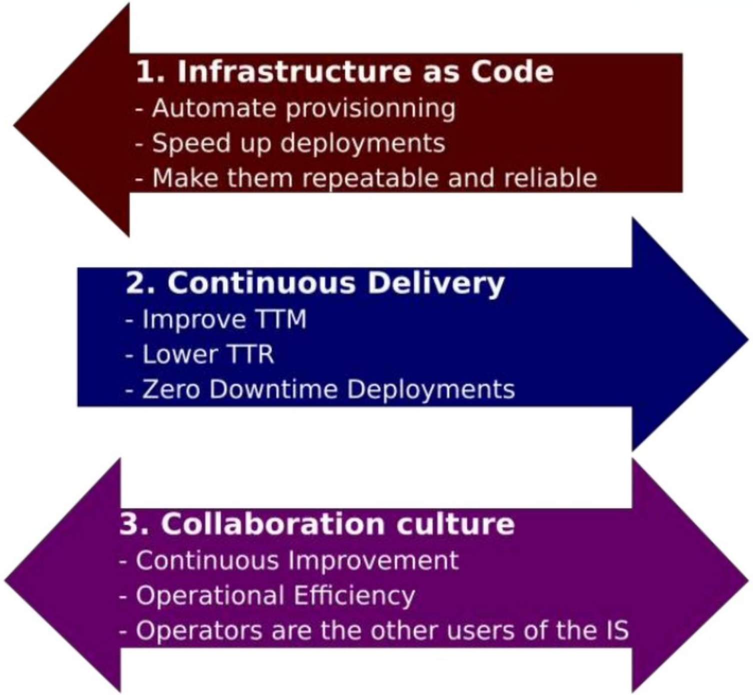
Separation of duties

Reduce reliance on separation of duty

- For decades, separation of duty has been used as one of the primary controls to reduce the risk of fraud or mistakes in the software development process.
- Separation of duty often slows down and reduces the feedback engineers receive on their work. This prevents engineers from taking full responsibility for the quality of their work and reduces a firm’s ability to create organizational learning. Consequently, wherever possible, we should avoid using separation of duties as a control.
- Instead, we should choose controls such as pair programming, continuous inspection of code check-ins, and code review. These controls can give the necessary reassurance about the quality of work.

Conclusion

Conclusion



1. Infrastructure as Code

- Automate provisionning
- Speed up deployments
- Make them repeatable and reliable

2. Continuous Delivery

- Improve TTM
- Lower TTR
- Zero Downtime Deployments

3. Collaboration culture

- Continuous Improvement
- Operational Efficiency
- Operators are the other users of the IS

Q&A