Dev & Ops Become DevOps
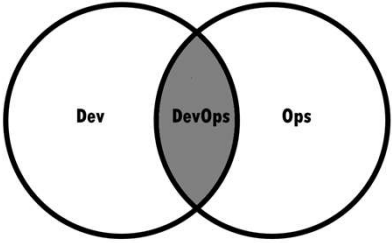
Without alignment on incentives and goals, Development & IT Operations will be at odds with each other.

Narrow the gap between the concept of Development and Operations – creating shared responsibility of developing and releasing software to customers via DevOps practices.

Agile, Continuous Delivery, and The Three Ways

DevOps practices focuses on three core patterns:

1. Maximise flow of work from Business to the Customer
2. Create a fast and constant flow of feedback
3. Maintain a culture of trust, collaboration, and learning

1st Way: The Principles of Flow

Make your work visible

Use a Kanban board to show your entire workstream, making it visible to all stakeholders to drive central prioritization of work

Limit work in process (WIP)

Establish WIP limits at each stage of the Kanban board to limit multi-tasking – measure lead times through the board

Reduce batch sizes

Set WIP limits on your Kanban board to reduce batch sizes by limiting the amount of in-flight work – the optimum batch size will be the lowest total cost of delivery when considering transaction and holding costs

Reduce the number of handoffs

Automate as much as possible in the development process –reorganizing developments teams to have all capabilities required to develop, test, release, and maintain their code in production

Continually identify and address your bottlenecks

Continually identify and remove the most significant bottleneck impacting your speed of delivery – creating change tolerant architectures and automation through development & release.

Eliminate hardships and waste in the value stream

Look for partially done work, extra processes/features, task switching, waiting, motion, manual work, and heroics – and optimize to remove these

2nd Way: The Principles of Feedback

Design a safe system of work

Manage complex work, swarm on problems, transfer knowledge through the organization, and grow leaders with these values

See problems as they occur

Create fast feedback and fast-forward loops via creation of automated builds, integration, and test processes.

Swam & solve problems to build new knowledge

Fix problems as they occur – and build a psychologically safe environment for people to raise concerns real time.

Keep pushing quality closer to the source

Don't hand off work to other teams, minimize approvals, right-size documentation, and make changes in small batches.

Enable optimizing for downstream teams

Design software with architecture, performance, stability, testability, configurability, and security prioritized into the work.

3rd Way: Continuous Learning & Experimentation

Enable an organizational learning & safety culture

Adopt a generative (Westrum) culture where failure leads to inquiry, and information, including risks, is freely shared.

Institutionalize the improvement of daily work

Pay down technical debt, fix defects, refactor and improve problematic areas of the code – the 'boy scout rule' of leaving code better than before

Transform local discoveries into global improvements

Created shared source repo, have blameless post-mortems, and make all documentation accessible & maintained to everyone in the organisation

Inject resilience patterns into our daily work

Relentless experimentation - testing the capacity/resilience of your code by trying to break it & using the learnings to create antifragile systems

Leaders reinforce a learning culture

Leaders create iterative, short term target conditions – and empower teams to experiment in order to solve for it.

Selecting which value stream to start with

Consider both systems of record and engagement

Optimise your value stream to maximise flow – focusing both on quality and speed to create a robust and fast flow of value

Start with the most sympathetic & innovative groups

Find teams that already believe in DevOps, focusing on creating success with those groups to build a coalition of change

Expand DevOps across the organization

Find innovators/early adopters, build a critical mass & silent majority, then once widely adopted – you can focus on the holdouts.

Understand the work in our value stream

Create a value stream map to see the work

No one person can know all the work that must be performed to create value for the customer – visualize this publicly for all to see

Create a dedicated transformation team

Assign dedicated resources to the DevOps transformation who are generalists and respected – create space for them to focus

Establish a shared goal

Create a north star for the transformation team – relentlessly communicate it to reinforce the vision and goal to the business

Keep our improvement planning horizons short

Be adaptive in planning improvements, work in short iterations of change, measure outcomes, and incorporate past learnings in new initiatives

Reserve time for NFR and technical debt

Dedicate effort for addressing non-functional requirements and technical debt – ideally 20-30% of time as a rule of thumb

Use tools to reinforce desired behaviour

Use common backlogs and tools between Dev & Ops teams

Design with Conway's Law in Mind

Enable market-orientated teams

Optimize for speed and embed the functional engineers and skills (Ops, QA, Infosec etc) into each service team

Test, operations, and security as everyone's job, every day

Establish shared goals on quality, availability, and security that are the responsibility of everyone in the development process.

Enable every team member to be a generalist

Focus on establish teams with generalist skills, providing opportunities for all engineers to learn the skill necessary to build and run systems

Fund not projects, but services and products

Fund long-lived teams that focus on the achievement of organizational and customer outcomes such as revenue, value, or adoption

Design team boundaries in accordance with Conway's law

Avoid splitting teams by function or by architectural layer – instead, structure teams around independent flow of value to the customer.

Create loosely-coupled architectures

Decouple your services so they can be independently maintained and deployed – with no shared data structures, and clearly defined boundaries

Keep team sizes small

Use the "two pizza" rule – where teams are small enough that they can be fed with two pizzas, ideally around 7 plus or minus 2.

Integrate operations into the daily work of development

Create shared services to increase developer productivity

Create a set of centralized platforms and tooling that enable dev – automated environments, testing, and common version control

Embed Ops engineers into our service teams

Ensure the operational skills are within the service teams, either by embedding DevOps, or training and empowering the development team

Assign an ops liaison to each service team

Build operational skills and awareness into teams by assigning an ops liaison to each development team

Integrate ops into dev rituals

Have the ops engineers attends development team ceremonies, participating to improve the operational supportability of development

Make relevant ops work visible on shared Kanban boards

Create a shared Kanban board that gives operations and development visibility of what work is flowing into production shortly.

Create the foundations of your Development Pipeline

Enable on demand creation of all environments

Establish automated tools for configuration, OS, environments, and deployment to allow dev teams to establish environments on demand

Create our single repository of truth for the entire system

Have all application code, scripts, schemas, env creation tools, containers, tests, and other technical artefacts in a common source control location.

Make infrastructure easier to rebuild than repair

Establish immutable infrastructure where manual changes to PRD are not allowed – on the construction/de-construction via automated processes.

Done for dev teams includes running in a PRD like env

Ensure development teams demonstrate code in a production-like environment as part of their definition of done.

Design with Conway's Law in Mind

Continuously build, test, and integrate our code

Step towards continuous delivery by automatically building and testing in a production like environment, when code is checked-in to version control.

Build a fast and reliable automated validation test suite

Automate all layers of the testing – balancing the test pyramid across unit, acceptance, integration, and functional testing.

Catch errors as early in our automated testing as possible

Establish an "ideal test pyramid" where we aim to detect issues as early and as fast as possible (ie. Unit tests)

Ensure tests run quickly (in parallel, if necessary)

Automate the commencement and running of tests (from source check-in), rather than waiting for manual approval or trigger from developers

Write our automated tests before we write the code (TDD)

Implement the red-green-refactor pattern of TDD, to write small, incremental changes with associated unit tests.

Automate as many of our manual tests as possible

Start with a set of automated and fully reliable tests, adding iteratively only tests that genuinely validate the business goals we're trying to achieve.

Integrate performance testing into our test suite

Write automated performance tests that validate across the entire application stack as part of the deployment pipeline.

Integration of non-functional requirements testing

Tests should include validation of system attributes we care about – supported applications, compilers, OS, and any other dependencies.

Establish Andon cord for when deployment pipelines break

When test failure occurs – ensure there is shared responsibility for all to react and address the failure before continuing further work.

Enable and practice continuous integration

Use small batch development

Merge early and often – by providing many small merges, as opposed to building up large and infrequent merges.

Adopt trunk-based development practices

Institutionalize that developers need to check-in their code to trunk at least once per day to limit the batch size of changes.

Automate and enable low-risk releases

Automate the deployment process (code, test, and infra.)

Automate all steps across the deployment processes, minimizing the manual effort required through the process to create repeatability

Enable automated self-service deployments

Create a code promotion process that can be performed by Dev or Ops without manual intervention to build, test, and deploy the software

Integrate code deployment into the deployment pipeline

Ensure packages are suitable for PRD deployment, see env readiness at a glance, automated deploy, and record and test automatically.

Decouple deployments from releases

Employ environment based or application based release patterns to decouple deployment from customer release.

Leverage patterns to improve speed and ease of deploy

Implement feature toggles or dark launches to control visibility of changes

Architect for low risk releases

Architect to enable productivity, testability, and safety

Establish a loosely-coupled architecture with well-defined interfaces which enforce how services connect with one another.

Select the best architecture for your needs

Monolithic architectures are fine for early life companies, but may not scale – establish a loosely coupled architecture and adaptable design.

Use the strangler pattern to safely evolve

To decommission legacy software – place it behind an API where it remains unchanged, then gradually replace it with the desired architecture.

Create Telemetry to Enable Seeing and Solving Problems

Create centralized telemetry infrastructure

Centralize logging, transform the logging into valuable metrics, then apply statistical analysis to identify patterns to trigger actionable events

Create application logging telemetry that helps production

Ensure every feature is instrumented and providing telemetry, and create logging hierarchies for both non-functional and feature attributes.

Use telemetry to guide problem solving

Leverage the telemetry to provide fact based problem solving - using the scientific method to create and test hypothesis to obtain learning.

Enable creation of production metrics as part of daily work

Create central and easy to use infrastructure and libraries so that it is easy for development & operations to create telemetry for all new functionality.

Enable self-service to telemetry and information radiators

Provide mechanisms so all teams can get access to production telemetry easily, without needing production access or privileged accounts.

Find and fill any telemetry gaps

Create telemetry at all levels of the application stack, for all environments, and throughout the entire deployment pipeline.

Analyse Telemetry to Anticipate Problems and Hit Goals

Use mean and standard deviations to detect problems

Create alerts that look for outliers from the mean using a standard deviation where data sets are bell curved in nature

Instrument and alert on undesired outcomes

Identify the lead indicators of outages, and instrument to alert on those to create pro-active early detection systems.

No standard deviation on telemetry that's not bell curved

Where normal operation can't be described by the bell curve – don't use the standard deviation as it will create over or under alerting

Leverage anomaly detection for non-bell curve

Establish patterns in your telemetry, and leverage smoothing, period patterns, and seasonality to your data where it described by a bell curve.

Enable Feedback So Dev and Ops Can Safely Deploy Code

Use telemetry to make deployments safer

Actively monitor the metrics associated with your feature during deployment - overlaying metrics with code deployment patterns for insight

Dev shares pager rotation duties with Ops

Make problems visible to Developers by having them be responsible for handling of operational incidents – by implementing and making them responsible for pager duties of priority incidents.

Have developers follow work downstream

Have the developers directly observe the UX of their software on real users – understanding any challenges users are facing.

Have Devs initially self-manage their production service

Dev teams have a *Launch Readiness Review* with Ops on their early life services – then self-manage those until operational stability and a *Hand-off Readiness Review* is completed.

Integrate A/B Testing into Our Daily Work

Integrate A/B testing into your feature testing

Release two version of your product, diverting a number users to the control ("A") or the treatment ("B") – applying statistical analysis of results

Integrate A/B testing into your release

Integrate feature toggles into new releases, and leverage them to control the percentage of users who experience the treatment version.

Integrate A/B testing into your feature planning

Use the feature hypothesis: *We Believe (action), will result in (result), we will have confidence to proceed when see (measure)*

Create Review and Coord. Processes to Increase Quality

Avoid the dangers of change approval processes

Change controls can create negative impacts – be mindful that more controls added means a more rigid processes, and less adaptability.

Ensure you don't "Overly control" changes

You cannot reliably predict successful changes with words - use control methods that resemble peer review & reduce reliance on external bodies

Enable coordination and scheduling of changes

Create loosely-coupled architecture to avoid release dependencies – enabling independent deployment of services by teams.

Enable peer review of changes

Ensure all code is reviewed prior to release – keeping the size of changes small to streamline review & release practices.

Avoid manual testing and change freezes

Automate and integrate testing into your daily work, ensuring a flow of changes into production with high release frequency

Enable pair programming to improve changes

Spread knowledge and develop in small testable batches through pair programming, and practices like TDD/BDD

Fearlessly cut bureaucratic processes

Relentlessly reduce the effort required for engineers to perform work and deliver it to the customer with light controls, and high automation.

Enable and Inject Learning into Daily Work

Establish a just, learning culture

Build a culture that embraces failure as a trigger for inquiry and learning , and not of scapegoating and blame

Schedule blameless post-mortem meetings after accidents

When failures occur, bring all stakeholders together to understand the timeline of events, identify root cause, identifying blameless learnings

Publish our post-mortems as widely as possible

Make the findings and actions of post-mortems transparent to all, all the way through to the customer, if possible. The goal is to spread the knowledge, so others can learn from it.

Decrease incident tolerances to find weaker failure signals

Standardization along cannot prevent software issues – continually experiment and discover to find new software risks.

Redefine failure and encourage calculated risk-taking

You need to fail faster and more often, identifying it as a learning opportunity and applying the necessary correction to prevent recurrence

Inject production failures to enable resilience and learning

Deliberately create failure scenarios in production – Implement a 'Chaos Monkey' to test the resilience of your production systems.

Institute game days to rehearse failures

Regularly simulate failure - This tests the fault resistance of your software in a wide variety of scenarios to identify and address latest defects

Convert Local Discoveries into Global Improvements

Use chat to automate and capture org. knowledge

Document and share observations of system and testing health automatically via a shared chat location that is transparent to all

Automated standardized processes in software for re-use

Capture knowledge and documentation of services in source control, making information available for everyone to search and use.

Create a single, shared source code repository

Establish a central shared source repository that stores all tools/ libraries/infrastructure/config/source for deploying all environments

Spread knowledge through docs and CoP

Develop tests that are self documenting of the code – showing engineers working examples of how to use the system.

Design for operations through codified NFR

Establish standard NRF requirements that set a baseline that all new services must achieve in order to enable operational objectives.

Build reusable operations user stories into development

Relentlessly automate every step of the deployment process – Supporting Ops improvements with Engineering effort in automation and tooling

Ensure technology choices help achieve org. goals

Select technology standards that allow for fast deployment, common learning and skill, and ease of understanding and maintenance.

Reserve Time to Create Org. Learning and Improvement

Institutionalize rituals to pay down technical debt

Regularly schedule improvement blitzes/hack weeks focusing on enabling the team to pay back technical debt and improve their means of delivery

Enable everyone to teach and learn

Dedicate regular time for learning and teaching – being committed to prevent it being deprioritized for other operational work.

Share your experiences from conferences

Apply and experiment with the learnings you obtain from conferences – fostering the relationships you build for continuous learning from peers

Create internal consulting and coaches to spread practices

Allocate specific resources focused on improvement without constraint

Information Security as Everyone's Job, Every Day

Integrate sec into development iteration demonstrations

Incorporate security into the acceptance criteria and DoD for your stories

---

Integrate security into defect tracking and post-mortems

Track all security issues in the same work tracking system as that which Dev and Ops are using – include post-mortem learnings into this

Integrate security controls into source code and services

Centralize a set of pre-validated security blessed libraries that are maintained and pulled in real-time during the CI/CD pipeline

Integrate security into your deployment pipeline

Create security tests that run as part of the deployment pipeline for every committed change.

Ensure security of the application

Tests should include static & dynamic analysis, dependency scanning, and code integrity and signing checks – and be aligned with OWASP guidelines

Ensure security of your software supply chain

Ensure all packages and dependencies used are up to date, and meet the same security tests required of your platform as a whole.

Ensure security of the environment

Establish known good states of environments – automating the monitoring of all production instances against those good states.

Integrate information security into production telemetry

Provide security telemetry via the same tools that Dev, QA, and Operations are using to give everyone vision of security performance.

Create security telemetry in your applications

Establish telemetry into your applications to identify insecure practices or behaviours in the system operation – and flags appropriate alert levels

Create security telemetry in your environment

Establish telemetry into your environments to monitor changes to OS, security, config, infrastructure, or XSS/SQLi attempts & server errors

Protect your deployment pipeline

Harden CI/CD process, review all changes in version control, instrument to detect suspicious API calls, isolate CI processes running.

Protecting the Deployment Pipeline

Integrate security and compliance into change approval

Leverage ITIL's standard/normal/urgent change classifications and incorporate security assessment into those to meet compliance needs

Re-categorize the lower risk changes as standard changes

Categorize and record all changes, focusing on moving changes with patterns of high success and low MTTR to be 'standard' changes

Reduce reliance on separation of duty

Use controls like pair programming, continuous inspection, code reviews and others as the primary sources of control over separation of duty.

Ensure docs and proof for auditors and compliance officers

Work with auditors in the control design process - sending all telemetry to centralized systems for auditor access and auditing.