Brief Summary of
The DevOps Handbook
How to create world class agility, reliability and security in technology organizations

Gene Kim, Jez Humble, Patrick Debois, John Willis

Part I - Introduction

1. Focus on the Principles of Flow (which accelerate the delivery of work from Development to Operations to customers), Principles of Feedback (which enable us to create ever safer systems of work, Principles of Continual Learning and Experimentation – which fosters a high trust culture and a scientific approach to organizational improvement.
2. DevOps is the outcome of applying the most trusted principles from the domain of physical manufacturing and leadership to the IT value stream.
3. DevOps relies on bodies of knowledge from Lean, Theory of Constraints, The Toyota Production System, resilience engineering, learning organizations., safety culture, human factors and many others – such as high trust management cultures, servant leadership and organizational change management.

Chapter 1 – Agile, Continuous Delivery and the Three Ways

1. One of the fundamental concepts in Lean is the value stream – "the sequence of activities of an organization undertakes to deliver upon a customer request"
2. In DevOps, a technology value stream is the process required to convert a business hypothesis into a technology enabled service that delivers value to the customer.
3. The value stream is often easy to see and observe in manufacturing operations.
4. In IT, the value stream begins when any engineer checks in a change in version control and ends when the change successfully run in production, providing value to the customer.
5. The goal is to have testing and operations happening simultaneously with design/ development, enabling fast flow and high quality.
6. The three-key metrics in technology value stream are Lead time, Process time (Cycle time) and Percentage Complete and Accurate (%C/A). %C/A reflects the quality of the output of each step in our value stream.
7. The Three ways – the principles underpinning DevOps are
    a. Enable fast left to right flow of work from Dev to Ops to Customer. This is done through making work visible, reduce batch sizes, build in quality by preventing defects and constantly optimize for global goals.
    b. Enable fast and constant flow of feedback from left to right at all stages of the value stream. By seeing problems as, they occur, swarming them until effective counter measures, feedback loops are shortened and amplified.

c. The Third way enables creation of a generative high trust culture that supports a dynamic, disciplined and scientific approach to experimentation and risk taking.

Chapter 2 – The First way: The Principles of Flow

1. We increase flow by making work visible, by reducing batch sizes, and intervals of work and by building quality in, preventing defects from being passed to downstream work centers.
2. The goal is to decrease the amount of time required for changes to be deployed in production and to increase the reliability and quality of those services.
3. A significant difference between technology and manufacturing value streams is that our work is invisible. To help us see where work is flowing well and where work is queued or stalled, we need to make our work as visible as possible.
4. By putting all work for each work center in queues and making it visible - all stakeholders can more easily prioritize work in the context of global goals.
5. Controlling queue size (Limiting WIP) is an extremely powerful management tool as it is one of the few indicators of Lead time. Limiting WIP also makes it easier to see problems that prevents the completion of work.
6. Another key component to create a fast and smooth flow is performing work in small batch sizes.
7. Large batch sizes result in high levels of WIP and high levels of variability and flow that cascade through the entire process – resulting in long lead times and poor quality.
8. One of the factors in longer lead times is the large number of handoffs which we often see in a value stream. We must strive to reduce the number of handoffs by automating significant portions of work or by reorganizing teams that can deliver value to the customers themselves.
9. To reduce lead times and increase throughput we need to continually identify our system's constraints and improve its work capacity.
10. Goldratt's five focusing steps in addressing constraints include
    a. Identify the system's constraint
    b. Decide how to exploit the system's constraint
    c. Subordinate everything else to the above decision
    d. Elevate the system's constraint
    e. If in the previous steps, a constraint has been broken, go back to step one.
11. In typical DevOps transformations, the constraints could possibly be in
    a. Environment creation - Have environments that can be created on demand, and completely self-serviced so that they are available when we need them
    b. Code deployment – This is about automating deployments as much as possible with the goal of having it completely automated
    c. Test setup and run - Automate tests so that we can execute deployments safely and to parallelize them so that the test rate can keep up with the development rate.

   d. Overly tight architecture – create loosely coupled architecture so that changes can be made safely and with more autonomy, increasing developer productivity.
12. Eliminating waste in software development
    a. Partially done work – becomes obsolete and loses value as time progresses
    b. Extra processes – add effort and increase lead times
    c. Extra features – add complexity and effort to testing and managing functionality
    d. Task switching - leads to additional time and effort
    e. Waiting – increase cycle time and prevent the customer from getting value
    f. Motion – Handoffs create motion wastes and often require additional communication to resolve ambiguities
    g. Defects – the longer the time between defect creation and defect detection the more difficult it is to resolve the defect
    h. Nonstandard or manual work – reliance on nonstandard or manual work from others such as using non-rebuilding servers, test environments, configurations etc. causes issues
    i. Heroics – heroic deeds – working late hours regularly, deployment at odd hours sap the energy and enthusiasm of the team.

Chapter 3 – Principles of Feedback

1. We make our system of work safer by creating fast, frequent, high quality information flow throughout the value stream and our organization which includes feedback and feedforward loops.
2. Complex systems typically have a high degree of interconnectedness of tightly coupled components and system level behavior – and failure is inherent and inevitable in such complex systems – hence the need for designing a safe system of work.
3. It is not sufficient to merely detect issues when the unexpected happens – we must also swarm them, mobilizing whoever is required to solve the problem.
4. Swarming is required to preventing the problem from going downstream, and preventing the work center from starting new work.
5. To enable fast feedback in the technology value stream, we must create the equivalent of an Andon cord and the related swarming response.
6. Gary Gruver - “It is impossible for a developer to learn anything when someone yells at them for something they broke six months ago- that is why we need to provide feedback to everyone as quickly as possible, in minutes, not months.”

Chapter 4 – Principles of Continual Learning and Experimentation

1. High performing manufacturing operations promote learning – the work is not very rigidly defined, the system of work is dynamic, they conduct experiments to generate new improvements.

2. Ron Westrum, one of the first to observe the importance of organizational culture on safety and performance defined three types of culture
    a. Pathological organizations – characterized by large amounts of fear and threat. People often hoard information, withhold it, distort it for their own good. Failure is hidden.
    b. Bureaucratic organizations – characterized by rules and processes, often help individual departments to hold on to their "turf". Failure is processed through a system of judgement resulting in either punishment or justice and mercy.
    c. Generative organizations – characterized by actively seeking and sharing of information to better enable the organization to achieve its mission. Responsibilities are shared and failure results in reflection and genuine inquiry.
3. We improve daily work by explicitly reserving time to pay down technical debt, fix defects, and refactor and improve problematic ideas of our code and environments.
4. Example of Alcoa – who improved their safety record significantly over 2 years – and now have one of the most enviable safety records in the industry.
5. When new learnings are discovered locally, there must also be a mechanism to enable the rest of the organization to use and benefit from that knowledge – convert tacit knowledge into explicit, codified knowledge which becomes someone else's expertise through practice.
6. Lower performing organizations buffer themselves from disruptions in many ways – they bulk up or add flab (an increased inventory buffer, hiring more people than required – often leading to increased costs). High performing organizations achieve the same results by improving daily operations, introducing tension to elevate performance, and engineering more resilience into their system.
7. Leaders reinforce a learning culture by making all the right decisions.
8. The Leader helps coach the person conducting experiments with questions such as –
    - what was the last step, what happened?
    - what did you learn?
    - what is your next target condition?
    - what is your next step?
    - What obstacle are you working on now?
    - What is the expected outcome?

Part II – Where to start

Chapter 5 – Selecting which value stream to start with

1. Software products and services can often be categorized as Greenfield or Brownfield.
2. In technology, a greenfield project is a new software project or initiative in the early stages of planning or implementation where we build applications and infrastructure anew with few constraints. Typically used as pilot projects to demonstrate feasibility of something new.

3. Brownfield projects are existing products or services that are already serving customers and have in operations for years. These projects often come with significant amounts of technical debt, have no test automation in place and run on unsupported platforms.
4. Though it is usually believed that DevOps is ideally suited for Greenfield project, we find that this has been successful for Brownfield projects as well.
5. Similarly, it is important to consider both Systems of Record and Systems of Engagement for implementation of DevOps
6. Systems of Record (ERP systems) where correctness of the transactions and data are paramount have typically a slower pace of change due to regulatory and compliance requirements. This focuses on "Doing it right".
7. Systems of Engagement are customer facing systems, have a higher pace of change to support rapid feedback loops to enable them to conduct experimentation how best to meet customer needs. This focuses on "Doing it fast"
8. When we improve brownfield systems, we should not only strive to reduce their complexity and improve their reliability and stability, we should also make them faster, safer and easier to change.
9. We expand our DevOps operations across the organization in small incremental steps. Some of the steps to broad base the support for DevOps include :
    a. Find innovators and early adopters – identify those who are respected and have a high degree of influence and can give credibility to the initiative
    b. Build critical mass and silent majority - work with teams who are receptive to new ideas and expand the coalition, generating more successes
    c. Identify the holdouts – these are the high profile influential detractors who are likely to resist our efforts. Tackle them after having made substantial gains in the organization.

Chapter 6 – Understanding the work in the value stream, making it visible and expanding it in the organization.

1. It is important to gain a sufficient understanding of how value is delivered to the customer, what work is performed and by whom and what steps we can take to improve the flow.
2. After which, identify the members of the value stream who are responsible for working together to create value for the customer – Product Owner, Development, QA, Operations, InfoSec, Release Managers, Technology Executives and others form part of this group.
3. Once the "actors" are identified, the next step is to gain an understanding of how work is performed, documented in the form of a "Value Stream Map". The goal is not to document every step and associated details, but understand the areas of the value stream that are jeopardizing the goals of fast flow, short lead times and reliable customer outcomes.
4. Then create a dedicated transformation team and have them accountable for achieving a clearly defined, measureable system level result. For this, we would need to

   a. Assign members of the dedicated team to be solely allocated to the DevOps transformation efforts
   b. Select team members who are generalists who have skills across a wide variety of domains
   c. Select team members who have long standing and mutually respectful relations with the rest of the organization
   d. Create a separate physical space for the dedicated team if possible to maximize communication flow within the team.
5. Agree on a Shared goal – Defining a measurable goal with a clearly defined deadline agreeable to all stakeholders is important. Examples could be
   a. Reduce percentage of budget spent on production support and unplanned work by 50%
   b. Ensure lead time from code check in to production release is one week or less for 95% of changes
   c. Ensure releases can be performed during normal business hours with zero downtime.
6. Keep improvement planning horizons short – to achieve greater flexibility to reprioritize and re-plan quickly, quicker realization of improvements that make meaningful differences in the way we work, increased transparency and for faster feedback.

Chapter 7 - How to design our Organization and Architecture with Conway's Law in mind

1. This chapter talks about how we organize ourselves to best achieve our value stream goals – and how we organize our teams affects how we perform our work.
2. In 1968, Melvin Conway performed an experiment with 8 people to produce a COBOL and ALGOL compiler, split into 5 members and 3 members respectively. The resulting COBOL compiler ran in five phases and ALGOL in three – observations which led to what is known as the Conway's Law which states that organizations which design systems are constrained to produce designs which are copies of their communication structures.
3. There are three primary types of organizational structures that inform how we design our DevOps value streams with Conway's Law in mind – Functional, Matrix and Market
   a. Functional oriented organizations optimize for expertise, division of labor or reducing cost. They have hierarchical organizational structures and expertise is centralized which enables career growth and skill development
   b. Matrix oriented organizations attempt to combine functional and market orientation. This could complicate organization structures – where an individual could report to two managers.
   c. Market oriented organizations – they optimize for responding quickly to customer needs. These organizations tend to be flat, composed of multiple cross functional disciplines (marketing, engineering, sales etc.) which often lead to potential redundancies across the organization.
4. Problems caused by overly functional orientation (Optimizing for cost)

   a. Long lead time for complex activities like large deployments where we must open up tickets with multiple groups and coordinate handoffs
   b. Little visibility for the person performing the work into how their work relates to value stream goals
   c. Long queues, long lead times, result in poor handoffs, large amounts of rework, quality issues, bottlenecks etc.
5. To achieve DevOps outcomes, we need to reduce the effects of functional orientation (optimizing for cost) and enable market orientation (optimizing for speed) and have many small teams working together to deliver value to the customer quickly.
6. Jody Mulkey, CTO of Ticketmaster – talks of how he had used American football as a metaphor to describe Dev and Ops earlier. Ops was defense, who keeps the other team from scoring and Dev is Offense trying to score goals. He realized the metaphor was flawed since they never played the field at the same time – and are not actually on the same team.
7. Now John uses the analogy of Ops as Offensive linesmen and Dev as skill positions (quarterback, wide receivers), whose job is to move the ball down the field, the job of Ops is to help make sure Dev has enough time to properly execute the plays.
8. Since any complex operational activity requires multiple handoffs and queues between different areas – it is suggested that members are Generalizing Specialists – move away from the "I" shaped skills to the T Shaped and E shaped skills. This might require change in hiring practices – and the need to look for people who had curiosity, courage and candor.
9. It is good to have an architecture that enables small teams of developers to independently implement, test and deploy code to production safely and quickly – helps increase developer productivity and improve deployment outcomes.
10. Having loosely coupled architecture with bounded contexts can be achieved through Service Oriented Architecture (SOA). Bounded contexts help developers understand and update the code of a service without knowing anything about the internals of the peer service. Bounded contexts ensure that services are compartmentalized and have well defined interfaces which enables easier testing.
11. Keeping team sizes small reduces the amount of inter team communication and encourage the scope of each team's domain small and bounded.

Chapter 8 – How to get great outcomes by integrating operations into the daily work of development

1. Integrating Ops capabilities into Dev teams would make the teams efficient, productive and deliver value faster.
2. One way to enable market oriented outcomes is for Ops to create a set of centralized platforms and tooling services that any Dev team can use to become more productive such as getting production like environments, deployment pipelines, automated testing tools, production telemetry dashboards etc. his would enable Dev teams to spend more time on developing functionality for the customer as opposed to obtaining the infrastructure required to deliver and support that feature in production.

3. Another way we can enable more market oriented outcomes is by enabling product teams to become more self-sufficient by embedding Ops engineers within them, thus reducing their reliance on centralized operations. These product teams may also be completely responsible for service delivery and service support.
4. The main advantage of embedding Ops engineers into Dev teams is that their priorities are driven almost entirely by the goals of the product teams they are part of - and also become closely connected with the internal and external customers.
5. The other option is to have an Ops liaison to each service team – also called "Designated Ops". Their centralized Ops group continues to manage all environments (Prod and Pre-Prod) and helps ensure they remain consistent.
6. Like in the Embedded Ops model, the liaison attends team standups, integrating their needs into the Operations roadmap and performing any needed tasks.
7. Having Ops liaisons allows us to support more product teams than the embedded Ops model. The Ops team liaisons could be invited to Dev Standups and Retrospectives which would help them understand any issues faced by the Dev teams and help resolve them at the earliest.

Part III – The technical practices of Flow

Chapter 9- Create the foundations of Deployment pipeline

1. In order to create fast and reliable flow from Dev to Ops, we must ensure that we always use Production like environments at every stage of the value stream. These 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.
2. Long lead times in creating different environments (testing and pre-production) would impair testing and may result in production issues.
3. Developers should be able to run production like environments on their own workstations created on demand and self-serviced.
4. Creation of these environments does not mean having big documents with a long list of steps – but having a common build mechanism which would enable creation of any environment within minutes.
5. Automation could be used for building environments by
    a. Copying the virtualized environment (VMware image running scripts)
    b. Building an automated environment creation process
    c. Using Infrastructure as code configuration management tools such as Puppet, Chef, Ansible, Salt, CF Engine)
    d. Using automated operating system configuration tools (Solaris Jumpstart, Redhat Kickstart, Debian Preseed)
    e. Spinning an environment in a public cloud (Amazon web services, Google App Engine, Microsoft Azure services)
6. Once the creation of development, test and production environments are enabled, the next step is to create a single repository of truth for the entire system.
7. By having all production artifacts into version control, we can reliably and repeatedly reproduce all components of our working software system.

8. The following artifacts can be part of the shared version control repository
    a. All application code and dependencies
    b. Scripts to create database schemas, application references etc.
    c. Environment creation tools and artifacts
    d. Files used to create containers
    e. Supporting automated tests and manual test scripts
    f. Scripts that support code packaging, deployment, database migration and environment provisioning
    g. Project artifacts
    h. Cloud configuration files
9. Create conditions to make infrastructure easier to rebuild than to repair. This would mean that no manual changes are allowed – and only way production changes can be made is to put the changes in version control and recreate the code and environments from scratch. By doing this no variance is able to creep into production.
10. Modify Definition of Done only when code can be successfully built, deployed and confirms that it runs in production like environment – instead of merely a developer saying it is done.

Chapter 10 – Enable Fast and Reliable Automated testing

1. Gary Gruver observes “without automated testing, the more code we write, the more time and money is required to test our code – in most case this is a totally unscalable business model for any technology organization”
2. When any Google developer commits code, it is automatically run against a suite of hundreds of thousands of automated tests. If the code passes, it is automatically merged into the trunk, ready to be deployed into production. Many Google properties build hourly or daily, then pick which builds to release, others adopt a continuous “Push on Green” delivery philosophy.
3. The goal is to build quality into the product even early in development and build automated tests as part of their daily work. This creates fast feedback loop that helps developers to find problems early and fix them quickly when there are fewest constraints.
4. A variety of tools have been designed to provide deployment pipeline functionality – many of them open source (Jenkins, Go, Gocourse, Bamboo, TFS, Teamcity, Gitlab CI) as well as cloud based solutions such as Travis CI and Snap.
5. We begin the deployment pipeline by running the commit stage, which builds and packages the software, runs automated unity tests, and performs additional validation such as static code analysis, duplication and test coverage analysis and checking style. If successful, this triggers the acceptance stage which automatically deploys the packages created in the commit stage into a production like environment and runs the automated tests.
6. There also needs to be set of Continuous Integration practices which require three capabilities:
    a. A comprehensive and reliable set of automated tests that validate we are in a deployable state

   b. A culture that "stops the entire production line" when our validation tests fail
   c. Developers working in small batches on trunk rather than long lived feature branches.
7. In general, automated tests fall into one of the following categories from the fastest to the slowest
   a. *Unit tests*: These typically test a single method, class or function in isolation providing assurance to the developer that their code operates as designed. Unit tests often stub out databases and other external dependencies.
   b. *Acceptance tests*: These typically test the application as a whole to provide assurance that a higher level of functionality operates as designed and that regression errors have not been introduced. "The aim of the unit test is to show that a single part of the application does what the programmer intends it to – the objective of the acceptance tests is to provide that our application does what the customer meant it to, not that it works the way its programmers think it should" – Humble and Farley.
   c. *Integration Tests:* Integration tests are where we ensure that our application correctly interacts with other production applications and services as opposed to calling stubbed out interfaces. Integration tests are performed on builds that have passed unit and acceptance tests. Because integration tests are often brittle, we want to minimize the number of integration tests, and find as many of our defects as possible in unit and acceptance testing.
8. A design goal of automated test suite is to find errors as early in the testing as possible. That is why we run faster running automated tests (unit tests) before slower running automated tests (acceptance / integration tests) which are both run before any manual testing.
9. Design tests to run in parallel, potentially across different servers. We may also want to run different categories of tests in parallel.
10. One of the most effective ways to ensure we have reliable automated testing is to write those tests as part of our daily work using techniques such as Test Driven Development (TDD) and Acceptance Test Driven Development (ATDD). We begin every change to the system by first writing an automated test that validates the expected behavior fails and then we write the code to make the tests pass.
11. Try to automate as many of our manual tests as possible – reducing reliance on manual testing. However merely automating all the manual tests may create undesired outcomes and could generate false positives. *A small number of reliable automated tests is always preferable over a large number of manual or unreliable automated tests.*
12. Write and run automated performance tests that validate the performance across the entire application stack (code, database, storage network, virtualization) as part of the deployment pipeline, so we detect problems early when the fixes are cheapest and fastest. To find performance problems early we should log performance results and evaluate each performance run against previous results.
13. Integrate Non-Functional requirements (availability, scalability, capacity, security etc.) testing into the test suite – many of them are fulfilled by the correct configuration of our environments.

14. In order to keep our deployment in a green stage, we will create a virtual Andon cord similar to TPS. When someone introduces a change that causes our build or automated tests to fail, no new work is allowed into the system until that problem is fixed.

Chapter 11 – Enable and practice Continuous Integration

1. The ability to "branch" in version control was created primarily to enable developers to work on different part of the software system in parallel without the risk of individual developers checking in changes that could destabilize or introduce errors into the trunk.
2. However, the longer the developers were allowed to work in their branches in isolation, the more difficult it became to integrate and merge everyone's changes back into the trunk.
3. Have developers check in code to the trunk at least once a day if not more frequently. Frequent code commits to trunk means that we can run automated tests on the software system as a whole and receive alerts when a change breaks some part of the application.
4. We may even configure our deployment pipeline to reject any commits that takes us out of a deployable state. This method is called "*gated commits*" where the deployment pipeline first confirms that the submitted change will successfully merge, build as expected, pass all automated tests before actually being merged into the trunk.
5. Having these practices in place, we can modify of Definition of Done as – "At the end of each development interval, we must have integrated, tested, working and potentially shippable code demonstrated in a production like environment *created from trunk using a one click process and validated with automated tests"*

Chapter 12 – Automate and enable low risk releases

1. Prior to automating the deployment process, we need to document the steps in the deployment process such as a value stream mapping exercise and simplify and automate as many of the manual steps as possible such as
    - Packaging code in ways suitable for deployment
    - Creating preconfigured 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
2. The requirements for our deployment pipeline include

   a. Deploying the same way to every environment - by using the same deployment mechanism for every environment (Dev, Test, Prod), our production deployments are likely to be far more successful
   b. Smoke testing the deployments – connect to any supporting systems and run a single test transaction through the system to ensure the system is performing as designed.
   c. Ensure we maintain consistent environments – have a common build mechanism to create dev, test and prod environments
3. Enable automated self-service deployments – It is common practice for Operations to perform code deployments because separation of duties is a widely accepted practice to reduce the risk of outages and fraud. To achieve DevOps outcomes, our goal is to shift our reliance to other control mechanisms that can mitigate these risks through automated testing, automated deployment and peer review of changes.
4. Once the code deployment process is automated, we can make it a part of the deployment pipeline. Consequently, our deployment automation must provide for the following capabilities
   a. Ensure that packages created during the continuous integration process are suitable for deployment into production
   b. Show readiness of the production environments at a glance
   c. Provide a push button self-service button for any suitable version of the packaged code to be deployed into production
   d. Record automatically for auditing and compliance purposes wich commands were run on which machine, when and who authorized it, what the output was etc.
   e. Run a smoke test to ensure the system is operating correctly and the configuration settings including items such as database connection strings are correct
   f. Provide fast feedback for the deployer so that they can quickly determine if the deployment is successful.
5. Decouple deployments from releases – Deployment and Release are often used interchangeably – however they are two distinct actions that serve two very different purposes
   a. Deployment is the installation of a specified version of software to a given environment. Specifically, a deployment may or may not be associated with a release of a feature to customers
   b. Release is when we make a feature or a set of features available to customers or a segment of customers. Our code and environments should be architected in such a way that release of functionality does not require changing of our application code.
6. As we become able to deploy on demand, how quickly we expose new functionality to customers becomes a business and marketing decision, not a technical one. These are two broad categories of release patterns we can use
   a. 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 (by configuring load balancers). New code is deployed into a

non-live environment and the release is performed moving traffic to this environment.

b. 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. For instance, we can implement feature flags that progressively expose new functionality in production to the development team, all internal employees, 1% of our customers, or when we are ready, the entire customer base.

7. Environment based release patterns
    a. Decoupling environments from our releases dramatically changes how we work. We no longer have to perform deployments in the middle of the night or on weekends to lower the risk of negatively impacting customers
    b. We do this by having multiple environments to deploy into – but only one of them receives live traffic.
    c. Blue Green Deployment Pattern
        - We have two production environments – blue and green. At any time, only of these is serving customer traffic.
        - To release a new version of our service, we deploy to the inactive environment where we can perform without interrupting the user experience. When confident everything is functioning as designed, we execute our release by directing traffic to the blue environment.
        - Thus, blue becomes live and green becomes staging. Rollback is performed, if required, by sending customer traffic back to the green environment.
    d. The Canary and Cluster Immune System Release patterns
        - The Canary Release pattern automates the release process by promoting to successively larger and more critical environments as we confirm that the code is operating as designed.
        - In this pattern, when we perform a release, we monitor how the software of each environment is performing. When something appears to be going wrong, we roll back, otherwise we deploy to the next environment. E.g. Sets of production servers serving only internal employees, small percentage of customer and the entire customer base.
        - The Cluster Immune system expands on the canary release pattern by linking our production monitoring system with our release process and by automating the rollback of code when the user facing performance of the production system deviates outside of a predefined expected range such as the conversion rates for new users drops below norms of 15-20%
8. Application based patterns to enable safer releases
    a. This allows even greater flexibility in how we safely release new features to our customer often on a per feature basis. Because application based release patterns are implemented in the application, these require involvement from Development

b. Implement Feature Toggles
    - The primary way we enable application based release patterns is by implementing feature toggles, which provide us the mechanism to selectively enable and disable features without requiring a production code deployment.
    - Feature toggles also control which features are visible and available to specific user segments (internal employments, segments of customers etc.)
    - Feature toggles enable us to rollback easily and reduce the quality of service by increasing the number of users we serve by reducing the level of functionality delivered.
c. Perform Dark Launches
    - Feature toggles allow us to deploy features in production without making them accessible to users enabling a technique called dark launching.
    - This is where we deploy all the functionality into production and then perform testing of that functionality while it is still invisible to customers.
    - For large and risky changes, we often do this for weeks before the production launch enabling us to safety test with the anticipated production like loads.
d. By doing this, we no longer have to wait until a big bang release to test whether customers want to use the functionality we build. Instead by the time we announce and release our big feature, we have already tested our business hypothesis and run countless experiments to continually refine our product with customers who help us validate with features.

Chapter 13 – Architect for Low Risk releases

1. Jez Humble observes that the architecture of any successful product or organization will necessarily evolve over its life cycle.
2. eBay used a technique called the Strangler application pattern where instead of ripping out and replacing old services with architectures that no longer support our organizational goals, we put our existing functionality behind an API and avoid making further changes to it.
3. All new functionality is then implemented in the new services that use the new desired architecture making calls to the old system when necessary.
4. The Strangler application pattern is especially useful for helping migrate portions of monolithic applications or tightly coupled services to one that is more loosely coupled.
5. A loosely coupled architecture will well defined interfaces that enforce how modules connect with each other promotes productivity and safety.
6. It enables small productive two pizza teams that are able to make changes that can be safely and independently deployed. And because each service also has a well-

defined API, it enables easier testing of services and the creation of contracts and SLAs between teams.

7. By repeatedly decoupling functionality from our existing tightly coupled system, we move our work into a safe and vibrant ecosystem where developers can be far more productive resulting in the legacy application shrinking in functionality. It might even disappear entirely as all the needed functionality migrates to our new architecture.

Chapter 14 Create Telemetry to enable seeing and solving problems

1. A study in 2001 found that organizations with the highest service levels rebooted their servers twenty times less frequently than average and have five times fewer "blue screens of death".
2. In other words, best performing organizations were better at diagnosing and fixing service incidents, used a disciplined approach to solving problems using production telemetry to understand possible contributing factors to focus their problem solving as opposed to lower performers who would blindly reboot servers.
3. This would require an automated communication process by which measurements and other data are collected at remote points and are subsequently transmitted to receiving equipment for monitoring.
4. One of the findings of the 2015 State of DevOps Report was that high performers could resolve production incidents 168 times faster than their peers with the median high performer having a MTTR measured in minutes while the median low performer had an MTTR measured in days.
5. It is important to design and develop our applications and environments so that they generate sufficient telemetry allowing us to understand how our system is behaving as a whole – not having the information in silos.
6. This would mean that data is collected at the business logic, application, and environment layers in the form of events logs and metrics. And an event router responsible for storing metrics and events – to help further analysis and perform health checks.
7. Ideally we will create telemetry that tells us exactly when anything of interest happens as well as where and how. "Monitoring is so important that our monitoring systems need to be more available and scalable than the systems being monitored" – Adrian Cockcroft
8. Logging can be done at the Debug level, Info level, Warn level, Error level or Fatal level.
9. "When deciding whether a message should be an ERROR or a WARN, imagine being woken up at 4 AM. Low printer tone is not an ERROR" – Dan North.
10. Telemetry enables us to use the scientific method to formulate hypothesis about what is causing a particular problem and what is required to solve it.
11. One could enable creation of production metrics as part of daily work that could be displayed and analyzed
12. The next step is to identify any gaps in the telemetry that impede our ability to detect and resolve incidents – this requires creating enough telemetry at all levels of the application stack at different levels such as Business level, Application level, Infrastructure level, Client software level and Deployment pipeline level.

Chapter 15 – Analyze Telemetry to better anticipate problems and achieve goals

1. "Given a herd of cattle that should all look and act the same which cattle look different from the rest" – sums up what is needed to anticipate problems and resolve them proactively.
2. "Outlier detection" is one of the statistical techniques used to detect any abnormal running conditions resulting in significant performance degradation at Netflix.
3. One of the simplest statistical techniques that can be used to analyze production metric is computing its mean and standard deviation.
4. For instance, if a web server stopped responding to requests – we would look at leading indicators that could have warned us earlier such as
    a. Application level – increasing web page load times
    b. OS level – server free memory running low, disk space running
    c. Database level – transaction times taking longer than normal
    d. Network level – number of functioning servers behind load balancer, dropping etc.
5. Another statistical technique which can be used is "Smoothing" – for data which is of time series. Smoothing involves using moving averages which transform the data by averaging each point with all other data within the sliding window,

Chapter 16 – Enable feedback so development and operations can safely deploy code

1. Feedback mechanisms enable us to improve the health of the value stream at every stage of the service life cycle from product design through development and deployment and operations. This ensures that the services are production ready even at the earliest stages of the project.
2. Production deployments are actively monitored through metrics associated with the feature and ensure that the service is not broken inadvertently.
3. The goal is to catch errors in deployment pipeline before they get into production. Production telemetry is used to quickly restore service - either through fix forward (make code changes to fix the defect and pushed into deployment pipeline) or rollback (switch back to the previous release)
4. It is good to have everyone in the value stream share the responsibilities of handling operational incidents – by having developers, Dev Managers and Architects on pager rotation.
5. Another way is to have the product team watch a customer use the application in their natural environment often working at their desks. This often uncovers startling ways in which customers struggle to use the application and require multiple clicks to perform simple tasks. These customer observations result in significant learning and a fervent desire to improve the situation for the customer.

Chapter 17 – Integrate hypothesis driven development and A/B testing into our daily work

1. 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 would actually achieve the desired outcome.
2. Techniques such as hypothesis driven development, AB testing, customer acquisition funnel can be used towards this end.
3. The faster we can experiment, iterate, and integrate feedback into our product or service the faster we can learn and out-experiment the competition. And how quickly we can integrate our feedback depends on our ability to deploy and release software.
4. The most commonly used A/B technique in modern UX practice involves a website where visitors are randomly selected to be shown one of two versions of a page – either a control ("A") or a treatment ("B"). Based on statistical analysis, we demonstrate whether there is a significant difference in the outcomes o the two establishing a causal link between the treatment.

Chapter 18 – Create Review and Coordination processes to increase quality of our current work

1. The goal is to ensure Dev, Operations and InfoSec are continuously collaborating so that changes we make to our systems will operate reliably, securely and safely as designed.
2. The reality is that in environments with low trust, command and control cultures, the outcomes of rigorous change control and testing countermeasures often result in an increased likelihood that problems will occur again potentially with even worse outcomes.
3. Traditional change controls can lead to unintended outcomes, such as contributing to long lead times, and reducing the strength and immediacy of feedback form the deployment process.
4. One of the key findings of the State of DevOps report was that high performing organizations relied more on peer review and less on external approval of changes.
5. In many organizations Change Advisory Boards serve an important role in coordinating and governing the delivery process, but their job should not be to manually evaluate every change.
6. Instead of requiring approval from an external body prior to deployment, we may require engineers to get peer review of their changes. At a minimum, fellow engineer should review the change, but for higher risk areas, such as DB changes or business critical components, we may need approval from a Subject Matter Expert.
7. The principle of small batch sizes also applies to code reviews.
8. Code reviews come in various forms such as Pair Programming, "over the shoulder", email pass around or a tool assisted code review.
9. According to Dr. Laurie Williams "paired programmers are 15% slower than two independent individual programmers, which error free code increase from 70% to 85%"

10. Creating the conditions that enable change implementers to fully own the quality of their changes is an essential part of the high trust generative culture that one strives to build. These conditions enable us to create an even safer system of work, where we are all helping each other achieve our goals, spanning whatever boundaries necessary to get there.

Part V – The Third way – The Technical Practices of Continual Learning and Experimentation

Chapter 19 – Enable and Inject Learning into Daily work

1. 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 of making the solutions available throughout the organization. Such organizations can heal themselves.
2. For such organizations, responding to crises is not idiosyncratic work. It is something that is done all the time. It is this responsiveness that is their source of reliability.
3. Netflix built their system around loosely coupled architected with each component having aggressive timeouts to ensure that failing components didn't bring the entire system down.
4. They also introduced *Chaos Monkey* – which simulated AWS failures by constantly and randomly killing production servers. The intention was to have all the engineering teams to be used to a constant level of failure so that services could automatically recover without any manual intervention.
5. One of the prerequisites for a learning culture is that when accidents occur, the response to those accidents is seen as "just" and maximize opportunities for organizational learning – no naming, blaming and shaming the person who caused the failure.
6. To enable a *just* culture when accidents and significant incidents occur (failed deployment, production issue etc.), a blameless post mortem should be conducted – where in engineers are allowed to give detailed accounts of their contributions to failures and propose counter measures to prevent a similar accident recurring in future.
7. Importantly, these post mortems should be published as widely as possible in the organization, encouraging others to read them and learn from them.
8. Organizations need to decrease the threshold of what constitutes a problem in order to keep learning – seek to amplify weak failure signals. E.g. Columbia space shuttle – a piece of insulating foam breaking off from the external fuel tank on take-off.
9. Organizations are typically structured in one of two models
    a. Standardized model – where routine and systems govern everything including strict compliance with timelines and budgets
    b. Experimental model – where every day exercise and every piece of new information is evaluated and debated in a culture that resembles an R & D lab.

10. Injecting faults into the production environment is one way we can increase our resilience. Resilience requires that we first define our failure modes and then perform testing to ensure that these failure modes operate as designed.
11. Jesse Robins – "resilience engineering is an exercise designed to increase resilience through fault injection through critical systems. A Service is not really tested until we break it in production."
12. Schedule *Game days* to rehearse failures – such as DB failovers, turning off an important network connection to expose problems, powering off a facility without notice – and see how fast systems come up.
13. Peter Senge "The only sustainable competitive advantage is an organization's ability to learn faster than the competition".

Chapter 20 – Convert Local Discoveries into Global Improvements

1. This is about creating mechanisms that make it possible for new learnings and improvements discovered locally to be captured and shared globally throughout the organization multiplying the effect of knowledge and improvement.
2. Many organizations have created chat rooms to facilitate fast communication within teams. Chat rooms are also used to trigger automation.
3. Chat logs are a great learning tool for new comers to the team.
4. Integrating automation into chat rooms helps document and share observations and problem solving and reinforces a culture of transparency and collaboration in the organization.
5. Create a single shared source code repository for the entire organization which includes
    a. Configuration standards for libraries, infrastructure and environments
    b. Deployment tools
    c. Testing standards and tools including security
    d. Deployment pipeline tools
    e. Monitoring and analysis tools
    f. Tutorials and standards
6. Spread knowledge by using automated tests as documentation and communities of practice – create discussion groups or chat rooms so that anyone who has questions can get responses from other users quickly. This facilitates exchange of knowledge and experience and each one helps the other in case of any problems.
7. Create well defined Operations User stories that represents work activities that can be reused across all projects. This would enable better planning and more repeatable outcomes.

Chapter 21 Reserve Time to Create Organizational Learning and Improvement

1. Toyota Production System followed the Improvement Blitz (Kaizen Blitz) which is defined as a dedicated and concentrated period of time to address a particular issue over the course of several days. The output of the team is often a new approach to problem solving, new means of conveying information, a more organized workspace

and also a list of to do changes to be made down the road. Similar to this – Target had the DevOps Dojo Challenge – an intense 30 days with the objective of getting a breakthrough in the problems they face on a daily basis.

2. Other similar terms for this includes Sprint or Fall Cleaning and Ticket Queue Inversion weeks, Hack days, Hackathons, 20% innovation time etc.
3. At the end of the blitz / hackathons, each team makes a presentation to their peers that discusses the problem they were tackling and what they built. This reinforces fixing problems as part of daily work, taking pride and ownership in the innovations they create and look at continuously improving the system – this eventually leads to paying down technical debt.
4. A dynamic culture of learning creates conditions so that everyone can, not only learn but also teach, whether through traditional didactic methods or more experiential or open methods. Nationwide Insurance came out with "Teaching Thursday" – 2 hours of teaching / learning for every employee.
5. Creating an internal coaching and consulting organization is a method commonly used to spread expertise across the organization. E.g. Holding Office hours where any one can consult, ask questions.
6. Google came out with a weekly testing periodical – "Testing on the Toilet" (TotT) – the goal was to raise the degree of testing knowledge and sophistication through the company.

Part VI – The Technological practices of integration information, Security, Change management and compliance

Chapter 22 – Information Security as everyone's job every day

1. The ratio of engineers in Dev, Ops and InfoSec in a typical technological organization is 100:10:1. When InfoSec is that outnumbered, without automation and integrating information security into the daily work of Dev and Ops, InfoSec can only do compliance checking, which is the opposite of security checking.
2. The goal should be to have feature teams engaged with InfoSec as early as possible as opposed to primarily engaging at the end of the project. One way this can be done is by inviting InfoSec to the product demonstrations given at the end of the Sprints, so that they can better understand the team goals with respect to organization goals and provide guidance and feedback in the early stages of the project.
3. Next, track all open security issues in the same work tracking system that Dev and Ops are using ensuring that the work is visible and can be prioritized against other work.
4. Provide security training to Dev and Ops and review what they have created to help ensure that security objectives are implemented correctly.
5. Run automated security tests in the deployment pipeline alongside other static code analysis tools
6. As part of the testing include

   a. Static analysis - tools inspect program code for all possible run time behaviors and seek out coding flaws, back doors and potentially malicious code
   b. Dynamic analysis - execution of tests while a program is in operation. These monitor items such as system memory, functional behavior, response time and overall performance of the system
   c. Dependency scanning – involves inventorying all the dependencies for binaries and executables and ensuring that these dependencies are free of vulnerabilities or malicious binaries
   d. Source code integrity and code signing – all developers should have their own PGP key managed by keybase.io. All commits to version control should be signed and all packages created by version control should be signed.
7. Ensure security of software supply chain – use of Open source software and components have known vulnerabilities and could result in data breaches.
8. Ensure security of the environment through generating automated tests to ensure that all settings are correctly applied and environments are scanned for known vulnerabilities
9. Integrate security telemetry in the same tools that Dev, QA and Ops are using – giving everyone in the value stream visibility into how the applications are performing in hostile environments where attackers are attempting to exploit vulnerabilities, gain unauthorized access, commit fraud etc.
7. Creating security telemetry in the environment by monitoring OS changes, security group changes, configuration setting changes, etc.
8. Come out with mitigation strategies to protect the continuous build, integration and deployment pipeline such as
   a. Hardening continuous build and integration servers and ensuring we can reproduce in them in an automated manner
   b. Review all changes introduced in version control through pair programming or code review process
   c. Instrumenting the repository to detect when test code contains suspicious API calls
   d. Ensuring every CI process runs on its own isolated container or VM
   e. Ensuring version control credentials used by the CI system are read only

Chapter 23 – Protecting the Deployment Pipeline

1. Most IT organizations would have change management processes in place which are the primary controls to reduce operations and security risks.
2. Effective change management policies will recognize that there are different risks associated with different types of changes and those changes are handled differently. These changes fall into these three categories
   a. Standard Changes - Lower risk changes that follow an established and approved processes. E.g. monthly updates of application tax tables, application / OS patches that have a well understood impact. Change deployments can be completely automated and should be logged to have traceability

   b. Normal changes – Higher risk changes that require approval from agreed upon change authority like CCB. These will normally have a Request for Change form which includes the desired business outcomes, planned utility and warranty business case with risk and alternatives and proposed schedule
   c. Urgent changes – emergency changes – potentially high risk and often require senior management approval.
3. It is a good practice to link change request records to specific items in the work planning tools (Jira, Rally, TFS etc.) allowing us to create more context for the changes by identifying why we are making the change, who is affected by the change and what is going to be changed and linking it to defects, production incidents or user stories.
4. Use controls such as pair programing, continuous inspection of code check ins, and code reviews prior to deployment rather than having a bureaucratic process of getting multiple approvals.

****