O'REILLY®

DevOps for Finance

Reducing Risk Through Continuous Delivery

Jim Bird

DevOps for Finance

Jim Bird

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

DevOps for Finance
by Jim Bird

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Brian Anderson

Production Editor: Kristen Brown
Proofreader: Rachel Head
Interior Designer: David Futato
Cover Designer: Karen Montgomery

September 2015: First Edition

Revision History for the First Edition
2015-09-16: First Release

978-1-491-93822-5

[LSI]

Table of Contents

Introduction

NOTE Disclaimer: The views expressed in this book are those of the author, and do not reflect those of his employer or the publisher.

DevOps, until recently, has been a story about unicorns. Innovative, engineering-driven online tech companies like Flickr, Etsy, Twitter, Facebook, and Google. Netflix and its Chaos Monkey. Amazon deploying thousands of changes per day.

DevOps was originally about WebOps at Internet companies working in the Cloud, and a handful of Lean Startups in Silicon Valley. It started at these companies because they had to move quickly, so they found new, simple, and collaborative ways of working that allowed them to innovate much faster and to scale much more effectively than organizations had done before.

But as the velocity of change in business continues to increase, enterprises—sometimes referred to as "horses," in contrast to the unicorns referenced above—must also move to deliver content and features to customers just as quickly. These large organizations have started to adopt (and, along the way, adapt) DevOps ideas, practices, and tools.

This short book assumes that you have heard about DevOps and want to understand how DevOps practices like Continuous Delivery and Infrastructure as Code can be used to solve problems in financial systems at a trading firm, or a big bank or stock exchange. We'll look at the following key ideas in DevOps, and how they fit into the world of financial systems:

- Breaking down the "wall of confusion" between development and operations, and extending Agile practices and values from development to operations
- Using automated configuration management tools like Chef, Puppet, CFEngine, or Ansible to programmatically provision and configure systems (Infrastructure as Code)
- Building Continuous Integration and Continuous Delivery (CI/CD) pipelines to automatically test and push out changes, and wiring security and compliance into these pipelines
- Using containerization and virtualization technologies like Docker and Vagrant, together with Infrastructure as Code, to create IaaS, PaaS, and SaaS clouds
- Running experiments, creating fast feedback loops, and learning from failure

To follow this book you need to understand a little about these ideas and practices. There is a lot of good stuff about DevOps out there, amid the hype. A good place to start is by watching John Allspaw and Paul Hammond's presentation at Velocity 2009, "10+ Deploys Per Day: Dev and Ops Cooperation at Flickr", which introduced DevOps ideas to the public. IT Revolution's free "DevOps Guide" will also help you to get started with DevOps, and point you to other good resources. *The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win* by Gene Kim, Kevin Behr, and George Spafford (also from IT Revolution) is another great introduction, and surprisingly fun to read.

If you want to understand the technical practices behind DevOps, you should also take the time to read *Continuous Delivery* (Addison-Wesley), by Dave Farley and Jez Humble. Finally, *DevOps in Practice* is a free ebook from O'Reilly that explains how DevOps can be applied in large organizations, walking through DevOps initiatives at Nordstrom and Texas.gov.

Common Challenges

From small trading firms to big banks and exchanges, financial industry players are looking at the success of Google and Amazon for ideas on how to improve speed of delivery in IT, how to innovate faster, how to reduce operations costs, and how to solve online scaling problems.

Financial services, cloud services providers, and other Internet tech companies share many common technology and business challenges.

They all deal with problems of scale. They run farms of thousands or tens of thousands of servers, and thousands of applications. No bank—even the biggest too-big-to-fail bank—can compete with the number of users that an online company like Facebook or Twitter supports. On the other hand, the volume and value of transactions that a major stock exchange or clearinghouse handles in a trading day dwarfs that of online sites like Amazon or Etsy. While Netflix deals with massive amounts of streaming video traffic, financial trading firms must be able to keep up with low-latency online market data that can peak at several millions of messages per second, where nanosecond accuracy is necessary.

These Big Data worlds are coming closer together, as more financial firms like Morgan Stanley, Credit Suisse, and Bank of America adopt data analytics platforms like Hadoop. Google (in partnership with SunGard) is one of the shortlisted providers bidding on the Securities and Exchange Commission's new Consolidated Audit Trail (CAT), a secure platform that will hold the complete record of every order, quote, and trade in the US equities and equities options markets: more than 50 billion records per day from 2,000 trading firms and exchanges, all of which needs to be kept online for several years. This will add up to several petabytes of data.

The financial services industry, like the online tech world, is viciously competitive, and there is a premium on innovation and time to market. Businesses (and IT) are under constantly increasing pressure to deliver faster, and with greater efficiency—but not at the expense of reliability of service or security. Financial services can look to DevOps for ways to introduce new products and services faster, but at the same time they need to work within constraints to meet strict uptime and performance service-level agreements (SLAs) and compliance and governance requirements.

DevOps Tools in the Finance Industry

DevOps is about changing culture and improving collaboration between development and operations. But it is also about automating as many of the common jobs in delivering software and maintaining operating systems as possible: testing, compliance and security checks, software packaging and configuration management, and deployment. This strong basis in automation and tooling explains why so many vendors are so excited about DevOps.

A common DevOps toolchain includes:

- Version control and artifact repositories
- Continuous Integration/Continuous Delivery servers like Jenkins, Bamboo, TeamCity, and Go
- Automated testing tools (including static analysis checkers and automated test frameworks)
- Automated release/deployment tools
- Infrastructure as Code: software-defined configuration management tools like Ansible, Chef, CFEngine, and Puppet
- Virtualization and containerization technologies such as Docker and Vagrant

Build management tools like Maven and Continuous Integration servers like Jenkins are already well established across the industry through Agile development programs. Using static analysis tools to test for security vulnerabilities and common coding bugs and implementing automated system testing are common practices in developing financial systems. But as we'll see, popular test frameworks like JUnit and Selenium aren't a lot of help in solving some of the hard test automation problems for financial systems: integration testing, security testing, and performance testing.

Log management and analysis tools such as Splunk are being used effectively at financial services organizations like BNP Paribas, Credit Suisse, ING, and the Financial Industry Regulatory Authority (FINRA) for operational and security event monitoring, fraud analysis and surveillance, transaction monitoring, and compliance reporting.

Automated configuration management and provisioning systems and automated release management tools are becoming more widely adopted. CFEngine, the earliest of these tools, is used by 5 of the 10 largest banks on Wall Street, including JP Morgan Chase. Puppet is being used extensively at the International Securities Exchange, NYSE, E*Trade, and the Bank of America. Bloomberg, the Standard Bank of South Africa (the largest bank in Africa), and others are using Chef. Electric Cloud's automated build and deployment solutions are being used by global investment banks and other financial services firms like E*Trade.

While most front office trading systems still run on bare metal in order to meet low latency requirements, Docker and other containerization and virtualization technologies are being used to create private clouds for testing, data analytics, and back office functions in large financial institutions like ING, Société Générale, and Goldman Sachs.

Financial players are truly becoming part of the broader DevOps community by also giving back and participating in open source projects. For example, LMAX, who we will look at in more detail later, has open sourced its automated tooling and even some of its core infrastructure technology (such as the low-latency Disruptor inter-thread messaging library). And at this year's OSCON, Capital One released Hygieia, an open source Continuous Delivery dashboard.

Financial Operations Is Not WebOps

Financial services firms are hiring DevOps engineers to automate releases and to build Continuous Delivery pipelines, and Site Reliability Engineers (patterned after Google) to work in their operations teams. But the jobs in these firms are different in many ways, because a global bank or a stock exchange doesn't operate the same way as Google or Facebook or one of the large online shopping sites. Here are some of the differences:

- Banks or investment advisers can't run continuous, online behavioral experiments on their users, like Facebook has done. Something like this could violate securities laws.
- DevOps practices like "Monitoring as Testing" and giving developers root access to production in "NoOps" environments so that they can run the systems themselves work for online social media startups, but won't fly in highly regulated environments with strict requirements for testing and assurance, formal release approval, and segregation of duties.
- Web and mobile have become important channels in financial services—for example, in online banking and retail trading—and web services are used for some B2B system-to-system transactions. But most of what happens in financial systems is

system-to-system through industry-standard electronic messaging protocols like FIX, FAST, and SWIFT, and low-latency proprietary APIs with names like ITCH and OUCH. This means that tools and ideas designed for solving web and mobile development and operations problems can't always be relied on.

- Continuous Deployment, where developers push changes out to production immediately and automatically, works well in stateless web applications, but it creates all kinds of challenges and problems for interconnected B2B systems that exchange thousands of messages per second at low latencies, and where regulators expect change schedules to be published up to two quarters in advance. This is why this book focuses on Continuous Delivery: building up automated pipelines so that every change is tested and *ready* to be deployed, but leaving actual deployment of changes to production to be coordinated and controlled by operations and compliance teams, not developers.
- While almost all Internet businesses run 24/7, most financial businesses, especially financial markets, run on a short trading day cycle. This means that a massive amount of activity is compressed into a small amount of time. It also means that there is a built-in window for after-hours maintenance and upgrading.
- While online companies like Etsy must meet PCI DSS regulations for credit card data and SOX-404 auditing requirements, this only affects the "cash register" part of the business. A financial services organization is effectively one big cash register, where almost everything needs to be audited and almost every activity is under regulatory oversight.

Financial industry players were some of the earliest and biggest adopters of information technology. This long history of investing in technology also leaves them heavily weighed down by legacy systems built up over decades; systems that were not designed for rapid, iterative change. The legacy problem is made even worse by the duplication and overlap of systems inherited through mergers and acquisitions: a global investment bank can have dozens of systems performing similar functions and dozens of copies of master file data that need to be kept in sync. These systems have become more and more interconnected across the industry, which makes changes much more difficult and riskier, as problems can cascade from one system—and one organization—to another.

In addition to the forces of inertia, there are significant challenges and costs to adopting DevOps in the financial industry. But the benefits are too great to ignore, as are the risks of not delivering value to customers quickly enough and losing them to competitors—especially to disruptive online startups powered by DevOps. We'll start by looking at the challenges in more detail, to understand better how financial organizations need to change in order for them to succeed with DevOps, and how DevOps needs to be changed to meet their requirements.

Then we'll look at how DevOps practices can be—and have been—successfully adopted to develop and operate financial systems, borrowing ideas from DevOps leaders like Etsy, Amazon, Netflix, and others.

CHAPTER 1

Challenges in Adopting DevOps

DevOps practices like Continuous Delivery are being followed by some startup online banks and other disruptive online fintech platforms, often leveraging cloud services to get up and running quickly without spending too much up front on technology and to take advantage of elastic on-demand computing capacity as they grow.

But what about global investment banks, or a central securities depository or a stock exchange—large enterprises that have massive investments in legacy technology?

Enterprise Problems

So far, enterprise success for DevOps has been mostly modest and predictable: Continuous Delivery in consumer-facing web apps or greenfield mobile projects; moving admin apps and office functions into the Cloud; Agile programs to introduce automated testing and Continuous Integration, branded as DevOps to sound sexier.

In her May 2014 *Wall Street Journal* article, "DevOps is Great for Startups, but for Enterprises It Won't Work—Yet", Rachel Shannon-Solomon outlines some of the major challenges that enterprises need to overcome in adopting DevOps:

- Siloed structures and organizational inertia make the kinds of change that DevOps demands difficult and expensive.
- Most of the popular DevOps toolkits are great if you have a web system based on a LAMP stack, or if you need to solve specific automation problems. But these tools aren't always enough if

you have thousands of systems on different architectures and legacy technology platforms, and want to standardize on common enterprise tools and methods.

- Building the financial ROI case for a technology-driven business process transformation that needs to cross organizational silos doesn't seem easy—although, as we'll see by the end of this book, the ROI for DevOps should become clear to all of the stakeholders once they understand how DevOps works.
- Many people believe that DevOps requires a cultural revolution. Large-scale cultural change is especially difficult to achieve in enterprises. Where does the revolution start? In development, or in operations, or in the business lines? Who will sponsor it? Who will be the winners—and the losers?

These objections are valid, but they're less convincing when you recognize that DevOps organizations like Google and Amazon are enterprises in their own right, and when you see the success that some other organizations are having with DevOps at the enterprise level. They've already proven that DevOps can succeed at scale, if the management will and vision and the engineering talent are there.

A shortage of engineering talent is a serious blocker for many organizations trying to implement DevOps. But this isn't as much of a concern for the financial industry, which spends as much on IT talent as Silicon Valley, and competes directly with Internet technology companies for the best and the brightest.[1]

So what is holding DevOps adoption back in the financial markets? Let's look at other challenges that financial firms have to overcome:

- The high risks and costs of failure in financial systems
- Chaining interdependencies between systems, making changes difficult to test and expensive (and high risk) to roll out
- The weight of legacy technology and legacy controls
- Perceived regulatory compliance roadblocks
- Security risks and threats, and the fear that DevOps will make IT less secure

1 See *http://on.mktw.net/1MdiuaF*.

The High Cost of Failure

DevOps leaders talk about "failing fast and failing early," "leaning into failure," and "celebrating failure" in order to keep learning. Facebook is famous for its "hacker culture" and its motto, "Move Fast and Break Things." Failure isn't celebrated in the financial industry. Regulators and bank customers don't like it when things break, so financial organizations spend a lot of time and money trying to prevent failures from happening.

Amazon is widely known for the high velocity of changes that it makes to its infrastructure. According to data from 2011 (the last time that Amazon publicly disclosed this information), Amazon deploys changes to its production infrastructure every 11.6 seconds. Each of these deployments is made to an average of 10,000 hosts, and only .001% of these changes lead to an outage.

At this rate of change, this still means that failures happen quite often. But because most of the changes made are small, it doesn't take long to figure out what went wrong, or to recover from failures —most of the time.

Sometimes even small changes can have unexpected, disastrous consequences. Amazon EC2's worst outage, on April 21, 2011, was caused by a mistake made during a routine network change. While Netflix and Heroku survived this accident, it took out many online companies, including Reddit and Foursquare, part of the *New York Times* website, and several smaller sites, for a day or more. Amazon was still working on recovery four days later, and some customers permanently lost data.[2]

When companies like Amazon or Google suffer an outage, they lose online service revenue, of course. There is also a knock-on effect on the customers relying on their services as they lose online revenue too, and a resulting loss of customer trust, which could lead to more lost revenue as customers find alternatives. If the failure is bad enough that service-level agreements are violated, that means more money credited back to customers, and harm to the company brand through bad publicity and damage to reputation. All of this adds up fast, in the order of several million dollars per hour: one estimate is

2 For a list of articles giving various viewpoints on the Amazon outage, see *http://bit.ly/1UBWURz*.

that when Amazon went down for 30 minutes in 2013, it lost $66,240 per minute.

This is expensive—but not when compared to a failure of a major financial system, where hundreds of millions of dollars can be lost. The knock-on effects can extend across an entire financial market, potentially impacting the national (or even global) economy, and negatively affecting investor confidence over an extended period of time. Then there are follow-on costs, including regulatory fines and lawsuits, and of course the costs to clean up what went wrong and make sure that the same problem won't happen again. This could—and often does—include bringing in outside experts to review systems and procedures, firing management and replacing the technology, and starting again. As an example, in the 2000s the London Stock Exchange went through two CIOs and a CEO, and threw out two expensive trading systems that cost tens of millions of pounds to develop, because of high-profile system outages. These outages, which occurred eight years apart, each cost the UK financial industry hundreds of millions of pounds in lost commissions.

NASDAQ Fails on Facebook's IPO

On May 18, 2012, Facebook's IPO—one of the largest in history—failed while the world watched.

Problems started during the pre-IPO auction process. NASDAQ's system could not keep up with the high volume of orders and cancels, because of a race condition in the exchange's matching engine. As more orders and requests to cancel some orders came in, the engine continued to fall further behind, like a puppy chasing its own tail.

NASDAQ delayed the IPO by 30 minutes so that its engineers could make a code fix on the fly and fail over to a backup engine running the new code. They assumed that in the process they would miss a few orders, not realizing just how far behind the matching engine had fallen. Tens of thousands of orders (and requests to cancel some orders) had built up over the course of almost 20 minutes. These orders were not included in the IPO cross, violating trading rules. Orders that should have been canceled got executed instead, which meant that some investors who had changed their minds and decided that they didn't want Facebook shares got them anyway.

For more than two hours, traders and their customers did not know the status of their orders. This created confusion across the market, and negatively affected the price of Facebook's stock.[3]

In addition to the cost of lost business during the incident, NASDAQ was fined $10 million by the SEC and paid $41.6 million in compensation to market makers (who had actually claimed up to $500 million in losses) and $26.5 million to settle a class action suit brought by retail investors. And although NASDAQ made significant changes to its systems and improved its operations processes after this incident, the next big tech IPO, Alibaba, was awarded to NASDAQ's biggest competitor, the New York Stock Exchange.

The risks and costs of major failures, and the regulatory requirements that have been put in place to help prevent or mitigate these failures, significantly slow down the speed of development and delivery in financial systems.

System Complexity and Interdependency

Modern online financial systems are some of the most complex systems in the world today. They process massive transaction loads at incredible speeds with high reliability and integrity. All of these systems are interlinked with many other systems in many different organizations, creating a large and fragile "system of systems" problem of extreme scale and complexity.

While these systems might share common protocols, they were not necessarily all designed to talk with each other. All of these systems are constantly being changed for different reasons at different times, and they are rarely tested all together. Failures can and do happen anywhere along this chain of systems, and they cascade quickly, taking other systems down as load shifts or as systems try to handle errors and fail themselves.

It doesn't matter that all of these systems are designed to handle something going wrong: hardware or network failures, software failures, human error. Catastrophic failures—the embarrassing accidents and outages that make the news—aren't caused by only one thing going wrong, one problem or one mistake. They are caused by

3 For full details on the incident, see *http://on.wsj.com/1bd6MJk*.

a chain of events, mostly minor errors and things that "couldn't possibly happen."[4] Something fails. Then a fail-safe fails. Then the process to handle the failure of a fail-safe fails. This causes problems with downstream systems, which cascade; systems collapse, eventually leading to a meltdown.

Financial transactions are often closely interlinked: for example, where an investor needs to sell one or more stocks before buying something else, or cancel an order before placing a new one; or when executing a portfolio trade involving a basket of stocks, or simultaneously buying or selling stocks and options or futures in combination across different trading venues.

Failures in any of the order management, order routing, execution management, trade matching, trade reporting, risk management, clearing, or settlement systems involved can make the job of reconciling investment positions and unrolling transactions a nightmare.

Troubleshooting can be almost impossible when something goes wrong, with thousands of transactions in flight between hundreds of different systems in different organizations at any point in time, each of them handling failures in different ways. There can be many different versions of the truth, and not all of them will be correct. Synchronized timestamps and sequence accounting are relied on to identify gaps and replay problems and duplicate messages—the financial markets spend millions of dollars per year just trying to keep all of their computer clocks in sync, and millions more on testing and on reporting to prove that transactions are processed correctly. But this isn't always enough when a major accident occurs.

Nobody in the financial markets wants to "embrace failure."

4 For more on how this happens, read Dr. Richard Cook's paper, "How Complex Systems Fail".

The Knight Capital Accident

On August 1, 2012, Knight Capital, a leading market maker in the US equities market, updated its SMARS high-speed automated order routing system to support new trading rules at the New York Stock Exchange. The order routing system took parent orders, broke them out, and routed one or more child orders to different execution points, such as the NYSE.

The new code was manually rolled out in steps prior to August 1. Unfortunately, an operator missed deploying the changes to one server. That's all that was needed to cause one of the largest financial systems failures in history.[5]

Prior to the market open on August 1, Knight's system alerted operations about some problems with an old order routing feature called "Power Peg." The alerts were sent by email to operations staff who didn't understand what they meant or how important they were. This meant that they missed their last chance to stop very bad things from happening.

In implementing the new order routing rules, developers had repurposed an old flag used for a Power Peg function that had been dormant for several years and had not been tested for a long time. When the new rule was turned on, this "dead code" was resurrected accidentally on the one server that had not been correctly updated.

When the market opened, everything went to hell quickly. The server that was still running the old code rapidly fired off millions of child orders into the markets—far more orders than should have been created. This wasn't stopped by checks in Knight's system, because the limits checks in the dead code had been removed years ago. Unfortunately, many of these child orders matched with counterparty orders at the exchanges, resulting in millions of trade executions in only a few minutes.

Once they realized that something had gone badly wrong, operations at Knight rolled back the update—which meant that all of the servers were now running the old code, making the problem temporarily much worse before the system was finally shut down.

5 The SEC report on the Knight failure is available at *https://www.sec.gov/litigation/admin/2013/34-70694.pdf*.

The incident lasted a total of around 45 minutes. Knight ended up with a portfolio of stock worth billions of dollars, and a shortfall of $460 million. The company needed an emergency financial bailout from investors to remain operational, and four months later the financially weakened company was acquired by a competitor. The SEC fined Knight $12 million for several securities law violations, and the company also paid out $13 million in a lawsuit.

In response to this incident (and other recent high-profile system failures in the financial industry), the SEC, FINRA, and ESMA have all introduced new guidelines and regulations requiring additional oversight of how financial market systems are designed and tested, and how changes to these systems are managed.

With so many variables changing constantly (and so many variables that aren't known between systems), exhaustive testing isn't achievable. And without exhaustive testing, there's no way to be sure that everything will work together when changes are made, or to understand what could go wrong.

We'll look at the problems of testing financial systems—and how to overcome these problems—in more detail later in this book.

Weighed Down by Legacy

Large financial organizations, like other enterprises, have typically been built up over years through mergers and acquisitions. This has left them managing huge application portfolios with thousands of different applications, and millions and millions of lines of code, in all kinds of technologies. Even after the Y2K scare showed enterprises how important it was to keep track of their application portfolios, many still aren't sure how many applications they are running, or where.

Legacy technology problems are endemic in financial services, because financial organizations were some of the earliest adopters of information technology. The Bank of America started using automated check processing technology back in the mid 1950s. Instinet's electronic trading network started up in 1969, and NASDAQ's computerized market was launched two years later. The SWIFT international secure banking payment system went live in 1977, the same year as the Toronto Stock Exchange's CATS trading system. And the

"Big Bang" in London, where the LSE's trading floor was closed and the UK financial market was computerized, happened in 1986.

Problems with financial systems also go back a long time. The NYSE's first big system failure was in 1967, when its automated trade reporting system crashed, forcing traders to go back to paper. And who can forget when a squirrel shut down NASDAQ in 1987?

There are still mainframes and Tandem NonStop computers running business-critical COBOL and PL/1 and RPG applications in many large financial institutions, especially in the back office. These are mixed in with third-party ERP systems and other COTS applications, monolithic J2EE systems written 15 years ago when Java and EJBs replaced COBOL as the platform of choice for enterprise business applications, and half-completed SOAs and ESBs. Many of these applications are hosted together on enterprise servers without virtualization, making deployment and operations much more complex and error prone.

None of this technology supports the kind of rapid, iterative change and deployment that DevOps is about. Most of it is nearing end of life, draining IT budgets into support and maintenance, and taking resources away from new product development and technology-driven innovation. In a few cases, nobody has access to the source code, so the systems can't be changed at all.

Legacy technology isn't the only drag on implementing changes. Another factor is the overwhelming amount of data that has built up in many different systems and different silos. Master Data Management and other enterprise data architecture projects are never-ending in big banks as they try to reduce inconsistencies and duplication in data between systems.

Dealing with Legacy Controls

Legacy controls and practices, mostly Waterfall-based and paperwork-heavy, are another obstacle to adopting DevOps. Entrenched operational risk management and governance frameworks like CMMI, Six Sigma, ITIL, ISO standards, and the bureaucracy that supports them also play a role. Operational silos are created on purpose: to provide business units with autonomy, for separation of control, and for operational scale. And outsourcing of critical functions like maintenance and testing and support, with SLAs and more bureaucracy, creates more silos and more resistance to change.

DevOps initiatives need to fight against this bureaucracy and inertia, or at least find a way to work with it.

ING Bank: From CMMI to DevOps

A few years ago at ING, development and operations were ruled by heavyweight process frameworks. Development was done following Waterfall methods, using Prince2, Rational Unified Process, and CMMI. Operations was ruled by ITIL. ING had multiple change advisory boards and multiple acceptance gates with detailed checklists, and process managers to run all of this.

Changes were made slowly and costs were high. Project delivery problems led the company to adopt even more stringent acceptance criteria, more gates, and more paperwork.

Then some development teams started to move to Scrum. After an initial learning period, their success led the bank to adopt Scrum across development. Further success led to a radical restructuring of the IT organization. There were no more business analysts, no more testers, and no more project managers: developers worked directly with the business lines. Everyone was an application engineer or an operations engineer.

At the same time, ING rationalized its legacy application portfolio, eliminating around 500 duplicate applications.

This Agile transformation was the trigger for DevOps. The development teams were delivering faster than Ops could handle, so ING went further. It adopted Continuous Delivery and DevOps, folding developers and operators into 180 cross-functional engineering teams responsible for designing, delivering, and operating different applications.

The teams started with mobile and web apps, then moved to core banking functions such as savings, loans, and current accounts. They shortened their release cycle from a handful of times per year to every few weeks. Infrastructure setup that used to take 200 days can now be done in 2 hours. At the same time, they reduced outages significantly.

Continuous Delivery is mandatory for all teams. There is no outsourcing. ING teams are now busy building a private internal cloud, and replacing their legacy ESB with a microservices architecture. They still follow ITIL for change management and change control, but the framework has been scaled down and radically streamlined to be more efficient and risk-focused.[6]

The Costs of Compliance

Regulatory compliance is a basic fact of life in the financial industry, affecting almost every system and every part of the organization; it impacts system requirements, system design, testing, and operations, as well as the personal conduct of industry employees.

Global firms are subject to multiple regulators, different regimes with different requirements for different activities and financial products. In the US alone, a bank could be subject to regulation by the OCC, the Federal Reserve, the SEC, FINRA, the regulatory arms of the different exchanges, the CFTC, and the FDIC.

Regulations like Dodd-Frank, GLBA, Regulation NMS, Regulation SCI, and MiFID (and of course, for public financial institutions, SOX) impose mandatory reporting requirements; restrictions around customer data privacy and integrity; mandatory operational risk management and credit management requirements; mandatory market rules for market data handling, order routing, trade execution, trade reporting; rules for fraud protection and to protect against money laundering, insider trading, and corruption; "know your customer" rules; rules for handling data breaches and other security incidents; business continuity requirements; restrictions on and monitoring of personal conduct for employees; and auditing and records retention requirements to prove all of this. Regulations also impose uptime requirements for key services, as well as requirements for reporting outages, data breaches, and other incidents and for preannouncing and scheduling major system changes. This means that regulatory compliance is woven deeply into the fabric of business processes and IT systems and practices.

The costs and complexities of regulatory compliance can be overwhelming: constant changes to compliance reporting requirements, responding to internal and external audits, policies and procedures that need to be continuously reviewed and updated and approved,

6 This case study is based on public presentations made by ING staff.

testing to make sure that all of the controls and procedures are being followed. Paperwork is required to track testing and reviews and approvals for system changes, and to respond to independent audits on systems and controls.

Callout: Regulation SCI

In November 2015, the SEC's Regulation Systems Compliance and Integrity (Reg SCI) comes into effect, as a way to deal with increasing systemic market risks due to the financial industry's reliance on technology, including the widespread risk of cyber attacks. It is designed to minimize the likelihood and impact of technology failures, including the kinds of large-scale, public IT failures that we've looked at so far.

Initially, Reg SCI will apply to US national stock exchanges and other self-regulatory organizations (SROs) and large alternative trading systems. However, the SEC is reviewing whether to extend this regulation, or something similar, to other financial market participants, including market makers, broker-dealers, investment advisers, and transfer agents.

Reg SCI covers IT governance and controls for capacity planning, the design and testing of key systems, change control, cyber security, disaster recovery, and operational monitoring, to ensure that systems and controls are "reasonably designed" with sufficient capacity, integrity, resiliency, availability, and security.

It requires ongoing auditing and risk assessment, immediate notification of problems and regular reporting to the SEC, industry-wide testing of business continuity planning (BCP) capabilities, and extensive record keeping for IT activities. Failure to implement appropriate controls and to report to the SEC when these controls fail could result in fines and legal action.

In Europe, MiFID II regulations address many of the same areas, but extend to trading firms as well as execution venues like exchanges.

What do these regulations mean to organizations adopting or looking to adopt DevOps?

The regulators have decided that relevant procedures and controls will be considered "reasonably designed" if they consistently follow generally recognized standards—in the SEC's case, these are published government standards from the ISO and NIST (such as NIST

800-53). However, the burden is on regulated organizations to prove that their processes and control structures are adequate, whether they follow Waterfall-based development and ITIL, or Agile and DevOps practices.

It is too soon to know how DevOps will be looked at by regulators in this context. In Chapter 2 we'll look at a "Compliance as Code" approach for building compliance controls into DevOps practices, to help meet different regulatory requirements (such as Reg SCI).

Compliance Roadblocks to DevOps

Most regulators and auditors are lawyers and accountants—or they think like them. They don't necessarily understand Agile development, Infrastructure as Code, or Continuous Delivery. The accelerated pace of Agile and DevOps raises a number of concerns for them.

They want evidence that managers are directly involved in decisions about what changes are made and when these changes are implemented. They want to know that compliance and legal reviews are done as part of change management. They want evidence of security testing before changes go in. They are used to looking at written policies and procedures and specifications and checklists and other documents to prove all of this, not code and system logs.

Regulators and auditors like Waterfall delivery and ITIL, with approval gates built in and paper audit trails. They look to industry best practices and standards for guidance. But there are no standards for Continuous Delivery, and DevOps has not been around long enough for best practices to be codified yet. Also, auditors depend on the walls built up between development and operations to ensure separation of duties.

Separation of Duties

Separation of duties—especially separating work between developers and operations engineers—is spelled out as a fundamental control in security and governance frameworks like ISO 27001, NIST 800-53, COBIT and ITIL, SSAE 16 exams, and regulations such as SOX, GLBA, MiFID II, and PCI DSS.

Auditors look closely at separation of duties, to ensure that requirements for data confidentiality and integrity are satisfied: that data and configuration cannot be altered by unauthorized individuals, and that confidential data cannot be viewed by unauthorized individuals. They review change control procedures and approval gates to ensure that no single person has end-to-end control over changes to the system. They want to see audit trails to prove all of this.

Even in compliance environments that do not specifically call for separation of duties, strict separation of duties is often enforced to avoid the possibility or the appearance of a conflict of interest or a failure of controls.

DevOps, by breaking down silos and sharing responsibilities between developers and operators, seems to be in direct conflict with separation of duties. Allowing developers to push code and configuration changes out to production in Continuous Deployment raises red flags for auditors. However, as we'll see in "Compliance as Code" on page 45, it's possible to make the case that this can be done, as long as strict automated and manual controls and auditing are in place.

Another controversial issue is granting developers access to production systems in order to help support (and sometimes even help operate) the code that they write, following Amazon's "You build it, you run it" model. At the Velocity Conference in 2009, John Allspaw and Paul Hammond made strong arguments for giving developers access—at least limited access—to production:

> Allspaw: "I believe that ops people should make sure that developers can see what's happening on the systems without going through operations… There's nothing worse than playing phone tag with shell commands. It's just dumb.
>
> "Giving someone [i.e., a developer] a read-only shell account on production hardware is really low risk. Solving problems without it is too difficult."
>
> Hammond: "We're not saying that every developer should have root access on every production box."

At Etsy, for example, even in PCI-regulated parts of the system developers get read access to metrics dashboards ("data porn") and exception logs so that they can help find problems in the code that they wrote. But any fixes to code or configuration are done through Etsy's audited and automated Continuous Deployment pipeline.

Any developer access to a financial system, even read-only access, raises questions and problems for regulators, compliance, InfoSec, and customers. To address these concerns, you need to put strong compensating controls in place. Limit access to non-public data and configuration to a minimum. Review logging code carefully to ensure that logs do not contain confidential data. Audit and review everything that developers do in production: every command they execute, every piece of data that they look at. You need detective change control in place to report any changes to code or configuration. In financial systems, you also need to worry about data exfiltration: making sure that developers can't take data out of the system. These are all ugly problems to deal with.

You also need to realize that the closer developers are to operations, the more directly involved they will get in regulatory compliance. This could lead to developers needing to be licensed, requiring examinations and enforcing strict restrictions on personal conduct. For example, in March 2015 FINRA issued a regulatory notice proposing that any developer working on the design of algorithmic trading strategies should be registered as a securities trader.

Security Threats to the Finance Industry

Cyber security and privacy are important to online ecommerce sites like Etsy and Amazon (and, after then-candidate Obama's handle was hacked, to Twitter). But security is even more fundamentally important to the financial services industry.

Financial firms are obvious and constant targets for cyber criminals —there is simply too much money and valuable customer data that can be stolen. They are also targets for insider trading and financial fraud; for cyber espionage and the theft of intellectual property; and for hacktivists, terrorists, and nation state actors looking to disrupt a country's economic infrastructure through denial of service attacks or more sophisticated integrity attacks.

These threats are rapidly increasing as banks and trading firms open up to the Internet and mobile and other channels. The extensive integration and interdependence of online financial systems provides a massive attack surface.

For example, JP Morgan Chase, which spends more than a quarter of a billion dollars on its cyber security program per year, was hacked in June 2014 through a single unpatched server on the bank's vast network.[7] An investigation involving the NSA, the FBI, federal prosecutors, the Treasury Department, Homeland Security, and the Secret Service found that the hackers were inside JPMC's systems for two months before being detected. The same hackers appear to have also attacked several other financial organizations.

In response to these and other attacks, regulators including the SEC and FINRA have released cyber security guidelines to ensure that financial firms take security risks seriously. Their requirements extend out to partners and service providers, including "law firms, accounting and marketing firms, and even janitorial companies."[8]

Callout: The NASDAQ Hack

In late 2010, hackers broke into NASDAQ's Directors Desk web application and planted malware. According to NASDAQ, the hackers did not get access to private information or breach its trading platform.

At least, that's what they thought at the time.

However, subsequent investigations by the NSA and the FBI found that the hackers were extremely sophisticated. They had used two zero-day vulnerabilities—evidence of a nation state actor—and planted advanced malware (including a logic bomb) created by the Federal Security Service of the Russian Federation in NASDAQ's systems.

Agents also found evidence that several different hacking groups (including cyber criminals and "Chinese cyberspies") had compromised NASDAQ's networks, and may have been inside for years. More than a year after the hack, it was still not clear to the investigators who the attackers were or if the attackers were attempting to steal NASDAQ's technology IP, or get access to inside information about the market, or if they had intended to plant a digital weapon to disrupt the US economy.[9]

7 For details on this attack, see *http://nyti.ms/1zdvK32*.

8 See *http://nyti.ms/1L2JhoC*.

9 See *http://bloom.bg/1KaFBQU*.

Making the Case for Secure DevOps

Because of these increased risks, it may be hard to convince InfoSec and compliance teams that DevOps will make IT security better, not worse. They have grown accustomed to Waterfall project delivery and stage gate reviews, which gives them a clear opportunity and time to do their security checks and a way to assert control over projects and system changes.

Many of them think Agile is “the A word”: that Agile teams move too fast and take on too many risks. Imagine what they will think of DevOps, breaking down separation of duties between developers and operators so that teams can deploy changes to production even faster.

In “DevOpsSec: Security as Code” on page 38, we’ll look at how security can be integrated into DevOps, and how to make the case to auditors and InfoSec for DevOps as a way to manage security risks.

CHAPTER 2

Adopting DevOps in Financial Systems

Enough of the challenges. Let’s look at the drivers for adopting DevOps in financial systems, and how it can be done effectively.

Enter the Cloud

One of the major drivers for DevOps in financial enterprises is the adoption of cloud services. Online financial institutions like exchanges or clearinghouses are essentially cloud services providers to the rest of the market. And most order and execution management system vendors are, or are becoming, SaaS providers to trading firms. So it makes sense for them to adopt some of the same ideas and design approaches as cloud providers: Infrastructure as Code; virtualization; rapid, automated system provisioning and deployment.

The financial services industry is spending billions of dollars on building private internal clouds and using public cloud SaaS and PaaS (or private/public hybrid) solutions. This trend started in back-end, general-purpose systems, with HR, CRM, and office services using popular SaaS platforms and services like Microsoft’s Office 360 or Azure. Now more financial services providers are taking advantage of cloud platforms for data intelligence and analytics, using cloud storage services, and building test platforms in the Cloud.

Today, even the regulators are in the Cloud. FINRA's new surveillance platform runs on Amazon's AWS, using public trade data.[1] The SEC has moved its SEC.gov website and Edgar company filing system, as well as its MIDAS data analytics platform, to a private/public cloud to save operations and maintenance costs, improve availability, and handle surges in demand (such as the one that happened, for example, during Facebook's IPO).[2]

Cloud adoption is still being held back by concerns about security and data privacy, data residency and data protection, and other compliance restrictions, according to a recent survey from the Cloud Security Alliance.[3] However, as cloud providers continue to raise the level of transparency and improve auditing controls over operations, encryption, and ediscovery, and as regulators provide clearer guidance on the use of cloud services, more and more financial data will make its way into the Cloud.

Introducing DevOps: Building on Agile

DevOps is a natural next step in organizations where Agile development has proved successful. Development teams who have proven that they can iterate through designs and deliver features quickly, and the business sponsors who are waiting for these features, grow frustrated with delays in getting systems into production. They start looking for ways to simplify and streamline the work of acceptance testing and security and compliance reviews; dependency analysis and packaging; release management and deployment.

Capital One: From Agile to DevOps

The ING story is continuing in a way at Capital One, which purchased ING Direct USA in 2012. Until then, Capital One outsourced most of its IT. Today, Capital One is fully committed to Agile and DevOps.

1 See *http://aws.amazon.com/solutions/case-studies/finra/* for details.

2 See *http://ubm.io/1hZMMjT*.

3 Cloud Security Alliance, "How Cloud is Being Used in the Financial Sector: Survey Report", March 2015.

Capital One's Agile experiment started in late 2011, with just two teams. As more teams were trained in Agile development, as at ING, they found that they were building software quickly, but it was taking too long to get working software into production. Development sprints led to testing and hardening sprints before the code was finally ready to be packaged and handed off to production. This wasn't Agile; it was "Agilefall."

Capital One developers were following the Scaled Agile Framework (SAFe). They leveraged the idea of System Teams in SAFe, creating dedicated DevOps teams in each program to help streamline the — offs between development and operations. These teams were responsible for setting up and managing the development and test environments, automating build and deployment processes, and release management, acting as "air traffic controllers to navigate through the CABs."

Integration testing, security testing, and performance testing were all being done outside of development sprints by separate test teams. They brought this testing into the dedicated DevOps teams and automated it. Then they moved all testing into the development sprints, adopting behavior-driven/acceptance-test-driven development and wiring integration, security, and performance testing into a Continuous Delivery pipeline. Today they have 700 Agile teams following Continuous Delivery.[4]

Agile ideas and principles—prioritizing working software over documentation, frequent delivery, face-to-face collaboration, and a focus on technical excellence and automation—form the foundation of DevOps. And Continuous Delivery, which is the control framework for DevOps, is also built on top of a fundamental Agile development practice: Continuous Integration.

From Continuous Integration to Continuous Delivery

In Continuous Integration, developers make sure that the code builds and runs correctly on each check-in. Continuous Delivery takes this to the next step.

4 This case study is based on public presentations made by Capital One staff.

It's not just about automating unit testing (something that the development team owns). Continuous Delivery is about configuring test environments to match production as closely as possible, automatically; packaging the code and deploying it to test environments, automatically; running acceptance tests and stress tests and performance tests and security tests and other checks, with pass/fail feedback to the team—again, automatically. It's about auditing all of these steps and communicating status to a dashboard, then later, using the same pipeline and deployment steps to deploy the changes to production.

Continuous Delivery is the backbone of DevOps. It's an automated framework for making software and infrastructure changes; pushing out software upgrades, patches, and changes to configurations; and is repeatable, predictable, efficient, and fully audited.

Putting a Continuous Delivery pipeline together requires a high degree of cooperation between development and operations, and a much greater shared understanding of how the system works, what production really looks like, and how it runs. It forces teams to start talking to each other, exposing details about how they work.

There is a lot of work that needs to be done. Understanding dependencies, standardizing configurations, and bringing configuration into code. Cleaning up the build—getting rid of inconsistencies, hardcoding, and jury rigging. Putting everything into version control: application code and configuration, binary dependencies (like the Java Runtime), infrastructure configuration (recipes/manifests), database schemas, and configurations for the CI/CD pipeline itself. Automating testing. Getting all of the steps for deployment together and automating them carefully. Doing all of this in a heterogeneous environment, with different architectures and technology platforms and languages.

This work isn't development, and it's not operations either. This can make it hard to build a business case for: it's not about delivering specific business features or content, and it can take time to show results. But the payoff can be huge.

Continuous Delivery at LMAX

The London Multi-Asset Exchange (LMAX) is a highly regulated FX retail market in the UK, where Dave Farley (coauthor of the *Continuous Delivery* book) helped pioneer the model of Continuous Delivery.

LMAX's systems were built from scratch following Agile best practices: TDD, pair programming, and Continuous Integration. But LMAX took this further, automatically deploying code to integration, acceptance, and performance testing environments, building up a Continuous Delivery pipeline.

LMAX has made a massive investment in automated testing. Each build runs through 25,000 unit tests with code coverage failure, simple code analysis (using tools like FindBugs, PMD, and custom architectural dependency checks), and automated integration sanity checks. All of these tests and checks must pass for every piece of code submitted.

The last good build is automatically picked up and promoted to integration and acceptance testing, where more than 10,000 end-to-end tests are run on a test cluster, including API-level acceptance tests, multiple levels of performance tests, and fault injection tests that selectively fail parts of the system and verify that the system recovers correctly without losing data. More than 24 hours' worth of tests are executed in parallel in less than 1 hour.

If all of the tests and reviews pass, the build is tagged. All builds are kept in a secure repository, together with dependent binaries (such as the Java Runtime). Everything is tracked in version control.

QA can conduct manual exploratory testing or other kinds of tests on a build. Operations can then pull a tagged build from the development repository to their separate secure production repository, and use the same automated tools to deploy to production. Releases to production are scheduled every two weeks, on a Saturday, outside of trading hours.

There is nothing sexy about the technology involved: they rolled a lot of the tooling on their own using scripts and simple conventions. But it's everything that we've come to know today as Continuous Delivery.

Protecting the Pipeline

DevOps in a high-integrity, regulated environment relies heavily on the audit trail and checks in the Continuous Delivery pipeline. The integrity and security of this environment must therefore be ensured:

- Every step must be audited, from check-in to deployment. These audit logs need to be archived as part of records retention.
- You have to be able to prove the identity of everyone who performed an action: developers checking in code, reviewers, people pulling or pushing code to different environments.
- You need to ensure the integrity of the CI/CD pipeline and all the artifacts created, which means securing access to the version control system, the Continuous Integration server configuration, the artifact repositories containing the binaries and system configuration data and other dependencies, and all of the logs.
- There must be a secure way to manage secrets: keys and other credentials.
- Separate development and production repositories are required. Only authorized people can pull from a development repository to the production repository, and, again, all actions must be audited.

Test Automation

Testing is a critical function in financial systems. Regulators require proof that core financial systems have been thoroughly tested. Some regulatory guidance even lays out how testing needs to be conducted. For example, MiFID II requires trading firms to test their trading systems and algorithms with exchanges, which need to provide production-like testing facilities with representative data. A lot of time is spent on regression testing to make sure that changes don't break existing functions and interfaces.

Automated testing is fundamental to Continuous Delivery. Without automated tests, you can't do Continuous Delivery of changes through a pipeline. While some organizations (like exchanges) have invested a lot in automating testing, many financial institutions still rely heavily on manual testing for important areas like functional acceptance testing, integration testing, and security testing. A PwC study in 2014 found that only 15% of testing activities have been automated at major financial institutions.[5]

Because manual testing for large systems is so expensive, many firms outsource or offshore testing to take advantage of lower-cost skills, handing the code off to test teams in India or somewhere else in a “follow the sun” approach to be tested overnight.

Agile development, especially for web and mobile applications, is already pushing organizations away from manual acceptance testing and offshore teams and toward automated testing in-phase as part of development, because testing cannot keep up with the pace of change in rapid, iterative development. DevOps and Continuous Delivery push this even further.

The path toward automated testing is straightforward, but it’s not easy. It starts with the basics of Continuous Integration: automating unit testing and basic functional testing, and moving responsibility for regression testing onto developers.

This makes sense to teams already practicing Agile development and TDD. It’s much harder when you’re working on monolithic legacy systems that were never designed to be testable. Here, you can get help from Michael Feathers, and his excellent book *Working Effectively with Legacy Code* (Prentice Hall).

Continuous Delivery requires a big investment up front in setting up testing infrastructure, moving testing from offshore test teams into development, creating virtualized test platforms, and generating synthetic test data or anonymizing test data to protect confidentiality and privacy of information.

It will take a long time to write the thousands (or tens of thousands, or more) of tests needed to cover a big financial system. Many of the most important of these tests—integration tests, performance and capacity tests, security tests—are particularly difficult to automate in Continuous Delivery. Let’s look at what is needed to get this done.

5 PwC, “An ounce of prevention: Why financial institutions need automated testing”, November 2014.

Integration Testing

With the exception of online retail applications such as online banking, most financial transactions are done system-to-system through APIs. Central capital markets institutions like exchanges or major clearinghouses can be connected to hundreds of trading firms, while large OMS/EMS systems at trading firms may be connected to dozens of different trading venues and market data sources and back-office systems, all through different protocols. This makes integration testing and end-to-end testing at the API level critically important.

Regression testing of these interfaces is expensive and difficult to set up. Because test systems are not always available and are often not deterministic, you'll need to stub them out, creating test doubles or simulators that behave in predictable ways.

There are risks to testing using simulators or test harnesses. Because you've made the mock testing environment predictable and deterministic, you won't catch the kinds of exceptions and problems that happen in real-life systems, and that can lead to wide-scale failures. Race conditions, timeouts, disconnections, random behavior, and other exceptions will escape your automated testing safety net—which means that your exception-handling code needs to be carefully reviewed.

This also means that if you're making changes that could affect outside behavior, you need to do certification testing with other parties. Luckily, for widely used financial protocols like FIX or SWIFT at least, there are several automated tools to help with this.

One potential shortcut to automating integration testing in large systems is through model-based testing. According to Bob Binder at Carnegie Mellon's Software Engineering Institute, a well-defined protocol specification such as FIX or SWIFT can be used to automatically generate many of the integration test cases needed to cover the behavior of a system, including catching mistakes in detailed scenarios that could trip up a system in production.

Model-based testing is still a niche idea, but this may be changing soon, at least in some parts of the financial industry. Jim Northey, a financial systems integration testing expert and the Global Technical Committee Chair of the FIX Trading Community, is helping to lead an initiative to create machine-readable versions of FIX specifications. These formal, machine-readable specs could be exchanged and compared between systems to catch incompatibilities, and eventually used by protocol engines to automatically map between protocol implementations. They could also be fed into model-based testing tools to automatically generate integration tests.[6]

Performance and Capacity Testing

Regulators mandate regular capacity testing to ensure that financial systems can hold up to demand. Online trading and market data and risk management systems are all extremely sensitive to latency, which means that even small changes have to be carefully tested to ensure that they don't slow down latency-critical parts of the system.

There are three basic kinds of performance tests that need to be automated:

- System load testing using standard workloads against a baseline
- Stress testing to find the edge of the system's capability
- Micro-benchmark tests at the functional/unit level in performance-critical sections of code, to catch small degradations

The challenges in implementing automated performance testing include:

- Creating a controlled test environment and protecting tests from runtime variability, including runtime jitter for micro-benchmarks
- Designing representative scenarios and building load generation tools that handle financial protocols
- Putting an accurate measurement system in place (including instrumenting the system and capturing metrics, down to microseconds)
- Deciding on clear pass/fail criteria

The tricky part will be integrating all of this cleanly into Continuous Delivery, in a simple, repeatable way. From a legal standpoint, you should also be careful in how you design and implement automated

6 For a list of open source tools for model-based testing, go to Bob Binder's blog: *http://robertvbinder.com/open-source-tools-for-model-based-testing/*.

performance testing in Continuous Delivery, to make sure that you don't step on the patent that HP has filed on doing this.

Security Testing

Automating security testing in Continuous Delivery requires a rethink of how security testing is done. We'll look at how to do this in detail in "DevOpsSec: Security as Code" on page 38.

Automated Infrastructure Testing

Infrastructure as Code introduces a new dimension to operations engineering. It requires a disciplined software engineering approach to provisioning and configuring systems: no more ad hoc scripting or manual configuration or hardening steps.

Operations engineers need to understand and follow the same coding disciplines as application developers. This includes writing automated unit tests and integration tests using frameworks like rspec-puppet, Chef Test Kitchen, or Serverspec; learning about test-driven infrastructure and how tests should drive design and implementation; and wiring these tests into Continuous Integration and Continuous Delivery as part of an automated configuration management pipeline. Like developers, they need to learn to spend as much time, or more time, writing tests as they do writing code. And, like developers, they need to learn how to make changes in small, safe, incremental and iterative steps.

Manual Testing in Continuous Delivery

In Continuous Delivery, you try to automate testing as much as possible. All of these tests have to be designed to run within short time constraints, which might mean breaking tests into parallel pipelines and executing them across a grid (like LMAX did, as discussed earlier in this chapter).

But there is still an important place for manual testing in large, business-critical system. In particular, a manual approach is important for:

- Risk-based exploratory testing to look for holes and edge cases, including group-based multiparty testing sessions or "war games." Multiparty testing can be an especially useful way to find important bugs (like timing problems and race conditions and workflow problems) in interactive, online systems such as trading systems, by trying to recreate real-world conditions and introducing some randomness and stressors into testing. This is about bug hunting, not acceptance testing.

- Usability testing for any user interfaces.
- Penetration testing and other kinds of adversarial or destructive testing: trying to break things to see what happens.

With Continuous Delivery, there is always a window where this kind of testing can and should be done.

Changing Without Failing

Making changes to financial systems can be expensive and risky. Changes (especially street-wide regulatory reporting changes or rules changes) often need to be made in lockstep, coordinated across many different systems, both inside the enterprise and out. Change introduces new forms of failure, which is why changes can be so scary to large organizations—and why they are made so infrequently.

DevOps is about changing faster and changing more often, which sounds like it will make these risks even worse. But it turns out that changing more often actually reduces the risk of change—as long as you do it properly.

The Puppet Labs "2015 State of DevOps Report" found that DevOps high performers deploy changes 30x more often than their lower-performing peers, with lead times as much as 200x shorter. But they also have a much higher change success rate: 60x higher, in fact. And they recover from these failures much, much faster (168x).

How have they been able to achieve this? By breaking changes down into small increments that can be easily reviewed and tested. By automating and standardizing most or all of the steps required to take a change and put it into production. By carefully controlling and monitoring changes as they go in, and being prepared to respond to any problems as quickly as possible. And by learning from failures and mistakes and continuously improving.

All of this requires changing not only how you deliver software, but also how you think about software and how you decide when software is "done." Developers have to think more like operators, and consider how to make changes safe for production: operations requirements become as important as user requirements.

Minimize the Risk of Change

In DevOps you minimize the risk of change by leaning on your Continuous Delivery pipeline. If you focus on reducing risks and costs, using automation as much as possible, speed will come as a side effect.

Automate everything that you can. Automate testing. Use static analysis to automatically find common bugs and problems in code. And automate the deployment steps to test, pre-production, and production.

By automating and exercising these steps over and over, in test and in production, you reduce the risk of deploying changes. This is a case for "if it hurts, do it more often," until it stops hurting. This will force you to simplify steps and checks, minimize manual steps, and get the bugs out of deployment—and improve your confidence. Build checks during and at the end of the deployment process. Check dependencies and configurations. Replace checklists and procedures with automatically executed steps, scripted health checks, and self tests.

Use Continuous Delivery to eliminate, as much as possible, differences between the production and test environments. This makes testing results more meaningful, and it will help developers to become more familiar with what production looks like and how it actually works.

The more that developers and operations engineers work together, the more they will learn from each other. Operators will get more opportunities to understand the application and how it is designed. Developers will learn more about how production is set up and how the system actually runs. The more visibility that they have into each other's work, the fewer assumptions they will make—which means that they will make better design decisions, and fewer mistakes.

Reduce the Batch Size of Changes

The advantages of incremental, smaller changes are all well known in IT. Yet in the financial industry we still see big bang releases, even industry-wide changes that create unnecessary risk.

Automated Continuous Delivery reduces the cost of deploying individual changes. This means that making smaller changes, more often, becomes a viable way of reducing operational risk.

Smaller changes can be made even safer by deploying changes incrementally to production. One way of doing this is using “canaries”: releasing to one server and checking to make sure that the change is okay, then releasing to two servers and checking again, then four, and so on.

Another way to reduce risk is through “dark launching” a feature using runtime feature switches to turn on functionality only for some users, or only for some products, and again closely monitoring the results before promoting the change to more of the user community. This is a common approach to rolling out changes in financial systems. For example, exchanges often roll out new features or rules for products starting with the letters A–C, then D–F, and so on, as part of announced incremental release programs.

You’ll need to encourage both developers and operations engineers to think small, and to work out changes in small, safe, incremental steps. Put scaffolding in first. Protect changes through feature switches. And refactor in the small too: no “root-canal refactoring” allowed.[7] This is hard for developers to get comfortable with—even developers who have been working in Agile methods—and it can be even harder for operations engineers, because vendors often dictate how network, database, operating system, server, and storage updates are done.

Identify Problems Early

To recover fast from a failure, you need to recognize that something is going wrong as early as possible—that is, to minimize the mean time to detect a failure (MTTD).

The DevOps community is continuously working on how to improve metrics and alerting in order to find the exceptional needle in a haystack of events, and correlating application and system events and metrics to find patterns and trends.

7 For more on refactoring tactics, see Emerson Murphy-Hill and Andrew P. Black’s paper “Refactoring Tools: Fitness for Purpose”.

Financial systems already have strong monitoring capabilities in place, watching for exceptions and latency and event queuing, and online surveillance to catch anomalies in system use. This is another area where financial services operations can work with the broader DevOps community to learn and improve together, through conferences like Monitorama and by contributing to open source monitoring tools and frameworks such as Etsy's StatsD/Graphite/Carbon/Whisper monitoring stack.

DevOps also expands the responsibility for monitoring the system from the operations center to development, exposing production metrics, exception logs, and alerts to developers, especially after a change has been rolled out. Developers wrote the code; they know to freak out if an impossible error message shows up. Facebook has a rule around deployment: a developer's code won't go into production if that developer is not online to make sure that it goes in successfully.

In DevOps shops, developers are also on call for their code if something goes wrong after hours. Luckily, it's easy to add developers to the escalation ladder using services like Pager Duty.

In DevOps, logging, alerting, and instrumentation need to be part of the Definition of Done: the acceptance criteria for changes. Code reviews and testing should include checking alerts, logging, error handling, and runtime instrumentation. Tests should make sure that new features or services are properly wired into monitoring, and that sensitive data is not exposed in logs.

Minimize MTTR

Mean time to recovery (MTTR) is a key metric for DevOps shops. They assume that failures will happen, especially at Internet scale. As James Hamilton (now a Distinguished Engineer at Amazon) points out in his paper "On Designing and Deploying Internet-Scale Services", even extremely rare, "one-in-a-million" combinations of events can become commonplace when you're running thousands of servers that provide millions of opportunities for component failures each day. And in systems of this scale, operations mistakes will also happen, especially when making changes.

Knowing exactly how everything is configured and the status of every part of the system, leveraging automated configuration management, is an important first step. Knowing that you can quickly identify and roll back changes is next.

Always Be Ready to Roll Back

When serious problems come up in a big online financial system, rolling forward and trying to push a patch out immediately is not always a viable option. You must know with complete confidence that you can roll back, that it will work, and that it will be safe and fast and not introduce more problems. This means designing to make rollback easy, and building forward and backward compatibility into database and configuration changes and into APIs.

It also means building tests in to make sure that the rollback steps work and that the code runs properly, as part of your deployment process and Continuous Delivery.

Incident Response—Always Be Prepared

In the financial industry, an outage is often treated like a security data breach, with the response involving compliance and legal and usually requiring formal escalation and notification to participants and regulators.

Because it can be painful and expensive, most organizations don't test handling outages and other failures often enough. This is why Netflix's Chaos Monkey is so compelling and so controversial: it automatically injects random failures to test the resiliency of the system and the team's ability to respond to failures, live and in production.

GameDay exercises—running real-life, large-scale failure tests (like shutting down a data center)—have also become common practices in DevOps organizations like Amazon, Google, and Etsy. They could involve (at Google, for example) hundreds of engineers working around the clock for several days, to test out disaster recovery cases and to assess how exhaustion could impact the organization's ability to deal with real accidents.[8]

At Etsy, GameDay exercises are run in production, even involving core functions such as payments handling. As John Allspaw put it:

8 See the *ACM Queue* discussion "Resilience Engineering: Learning to Embrace Failure".

> Why not simulate this in a QA or staging environment? First, the existence of any differences in those environments brings uncertainty to the exercise, and second, the risk of not recovering has no consequences during testing, which can bring hidden assumptions into the fault tolerance design and into recovery. The goal is to reduce uncertainty, not increase it.[9]

These exercises are carefully tested and planned in advance. The team brainstorms failure scenarios and prepares for them, running through failures first in test and fixing any problems that come up. Then it's time to execute scenarios in production, with developers and operators watching closely and ready to jump in and recover, especially if something goes unexpectedly wrong.

Once the exercise is over, the team conducts a postmortem to learn about what happened, look into any surprises, and figure out what they need to improve.

The point of failure injection in production isn't just to find reliability problems in the system. It also tests your organization's ability to deal with failures, and builds confidence in your technology and your team.

Will we see stock exchanges doing large-scale failure testing in production? Not likely. While the arguments for doing this kind of testing in production are all valid, the risks of running live production incident tests in financial systems are too great. Even if you have high confidence in how your systems will behave in a failure situation, you can't predict how participants' systems will behave and what the impact will be on them, and on other systems downstream.

For this reason, fire drills are sometimes done in production outside of trading hours, under controlled conditions. This includes industry-wide BCP testing, such as the annual exercises conducted by SIFMA in the US. Alternatively, they may be run as simulations that don't impact live systems, like with Quantum Dawn, which simulates industry-wide cyber attacks during the business day to exercise each organization's incident response capabilities and interorganizational communications and escalation processes.

9 See his article in *ACM Queue*, "Fault Injection in Production: Making the case for resilience testing".

These kinds of tests, while imperfect and incomplete, are still valuable in evaluating system resilience and building a better organizational incident handling capability.

Get to the Root Cause(s)

Production incidents are expensive and stressful—but they also provide valuable information. One of the key ideas in DevOps is to take this information and use it to learn and improve at multiple levels.

Part of this involves collecting metrics to understand what kinds of changes are risky and whether the team is changing too much or too often, or not often enough, by looking at the type and size and frequency of changes, and correlating this to production problems. This idea is nicely described by John Allspaw in his 2010 Velocity presentation "Ops Meta-Metrics: The Currency You Use to Pay For Change".

Blameless postmortems are another important part of DevOps culture: getting Ops and development together after a failure to constructively review what went wrong and understand why it went wrong, discussing what can be done to prevent a problem like this from happening again, and sharing all of this information across the organization and with customers.

Postmortem analysis isn't a new idea, especially in the financial industry. Formal investigations after a major production problem are done routinely, often by regulators. The major difference in DevOps is the emphasis on *blameless* exploration of problems and sharing of information by the team, and conducting postmortems on smaller incidents and even "near misses" as learning experiences.

Recognizing that people will make mistakes, that fail-safes can fail and that accidents will happen, a DevOps postmortem gets people working together in an open and safe environment to share what happened and to understand why it happened, focusing on the facts and on problem solving, opening up a dialogue and creating opportunities to learn. It is not about determining liability or apportioning blame and deciding who is going to be fired. And it is not just about coming up with a list of bugs that need to be fixed, or procedures that need to be improved. A postmortem is an opportunity to explore mistakes and why they were made; to confront deeper technical and organizational issues like design resilience, training, decision making, and communications; to try to get to the root causes of problems, and to figure out how to get better as an organization. And, done properly, by focusing on the facts and by trusting and sharing with each other, a postmortem is another way to bring development and operations closer together.

I said we need to get to the "root causes" here. Because (back to Allspaw again, from his 2011 Velocity presentation "Advanced Post-Mortem Fu and Human Error 101"):

> There is no such thing as a root cause for any given incident in complex systems. It's more like a coincidence of several things that make a failure happen.
>
> In hindsight, there often seems to be a single action that would have prevented the incident to happen. This one thing is searched for by managers when they do a root cause analysis. They hope to be able to prevent this incident from ever happening again. But if the incident was possible due to the coincidence of many events, it makes no sense to search for a singular root cause. This would lead to a false sense of security, as in a complex system there are too many ways an incident can possibly happen.

The Knight Accident Through DevOps Eyes

Let's look at the Knight accident again,[10] through the lens of DevOps. It demonstrates a series of control failures in some key DevOps areas that we've just gone through.

Automated Release/Deployment

An operations engineer followed manual procedures to deploy changes, but missed deploying the code on one server, and unfortunately nobody noticed the mistake.

This problem is what automated configuration management and deployment is intended to prevent. An audited, automated deployment pipeline, with post-release checks and smoke tests (including looking for version mismatches on all servers) to check that the deployment was successful, would have avoided this problem.

10 For a good summary of the Knight Trading accident from a DevOps perspective, read "Knightmare: A DevOps Cautionary Tale" by Doug Seven.

Dark Launching and Branching in Code

One way to minimize risk with code changes is to hide the changes behind a feature switch, so that operators can control the behavior of the system at runtime by turning a flag on or off. In Knight's case, the code was executed based on a flag in an order message, not a runtime switch value, which meant that there was no easy way for operations to stop the code from executing.

This case also highlights some of the risks of using conditional switches and branching in code to control runtime behavior of a system. For this new feature, Knight's developers chose to repurpose a flag that meant something quite different in the old code. And because of the deployment mistake discussed above, this old code was still running on one server, so it got triggered accidentally.

Using conditional logic to "branch in code" allows you to introduce changes in steps and to control system behavior at runtime. But it also makes code harder to understand, harder to change, and harder to test. The longer conditional logic and switches are left in the code, and the more switches are added over time, the worse these problems become. After a while, nobody understands or is prepared for what will happen if some flag or a combination of flags gets turned on—which is what happened to Knight.

Feature switches and branches in code are a dangerous kind of technical debt. Teams need to follow a disciplined approach in managing code like this, making sure to remove it as soon as it's no longer needed.

Visibility and Monitoring/Feedback Loops

Another important practice in DevOps is making sure developers are on call and available to help with any changes that they make. If the developers who had been working on the code had seen the "Power Peg" alerts in the early morning, they may have recognized what was happening, or at least been surprised enough to look further, and been able to help stop things from going wrong before the market opened.

Responding to Failure

We've gone through how important it is to always be prepared for failure, and to have a proven incident response capability in place. This includes knowing when to roll back code and knowing that it will work, and having a well-defined escalation ladder to someone who can "pull the kill switch" and shut things down quickly, before a problem gets out of control. Knight's team took too long to make critical decisions—by the time the system was shut down, the company was effectively already out of business.

DevOpsSec: Security as Code

The approach that most financial organizations take to IT security today is "scan, then fix." They depend heavily on security reviews in Waterfall project gates: reviewing specifications and architecture, scanning code before it's handed off to test, pen testing the system before it goes live.

But in DevOps there are no Waterfall gates where security audits or penetration tests can be scheduled. There aren't even any Agile security sprints or hardening sprints. Security needs to be brought into development and operations, and included in Continuous Delivery stages.

Whether it's called DevOpsSec or DevSecOps or Rugged DevOps, or whether it has a name at all, security in DevOps is based on a few key ideas:

- Breaking down walls between development, operations, *and* InfoSec, and bringing them all together to solve (and, more importantly, to prevent) security problems
- Shifting security controls and checks earlier, into design and development
- Automating security testing and security checks in Continuous Integration and Continuous Delivery, including security checks on dependencies
- Taking advantage of Infrastructure as Code and Continuous Delivery to standardize and secure the runtime environment
- Leveraging the logging and workflow controls in Continuous Delivery to provide an audit trail of security checks for regulators
- Wiring security into application operations monitoring and feedback loops

Shift Security Left

To keep up with the pace of Continuous Delivery, security has to be "shifted left," earlier into the design and coding processes, and into the automated test cycles, instead of waiting and running security checks just before release. Security has to fit into the way that engineers think and work: more iterative and incremental, and automated in simple ways.

Some organizations do this by embedding InfoSec specialists into development and operations teams. But it is difficult to scale this way, because there are too few InfoSec engineers to go around—especially ones who can work at the design and code level. This means developers and operators need to be given more responsibility for security, training in security principles and practices, and tools to help them build and run secure systems.

Developers need to learn how to identify and mitigate security risks in design through threat modeling, and how to leverage security features in their application frameworks and security libraries to prevent common security vulnerabilities like injection attacks. The OWASP and SAFECode communities provide a lot of useful, free tools and frameworks and guidance to help developers with understanding and solving common application security problems in any kind of system.

Making smaller changes in DevOps not only reduces operational risk of failure; it also reduces security risks, because most small, incremental changes do not meaningfully increase the system's attack surface. But a red flag should be raised whenever anyone makes a high-risk change, such as changing crypto code or the configuration of a public network facing device. This can be done automatically on check-in. For example, at Etsy, they hash high-risk code and automatically run unit tests as part of Continuous Integration that will alert InfoSec if any of this code changes.

Self-Service Automated Security Scanning

If you want to make developers more responsible for application security, you need to give them simple tools that work iteratively and incrementally, and that provide fast and simple feedback.

Scanning applications for security vulnerabilities using automated tools is fundamental to most security programs today. But rather than relying on a centralized security scanning factory run by InfoSec, DevOps organizations like Twitter and Netflix implement self-service security tools for developers.

While Dynamic Analysis Security Testing (DAST) tools and services are important in testing web and mobile apps, they don't play that nicely in Continuous Integration or Continuous Delivery. Most of these tools are designed to be run by security analysts or pen testers, not a Continuous Integration server like Jenkins or Bamboo. While you can run an attack proxy like OWASP's ZAP in headless mode to automatically scan a web app for common vulnerabilities, it's difficult to return unambiguous pass/fail results. And more importantly, these tools can't be used to test system-to-system APIs.

This means that Static Analysis Security Testing (SAST) becomes the scanning technology of choice in Continuous Delivery. Developers can take advantage of IDE plug-ins like Cigital's SecureAssist, or plug-ins from Coverity, Klocwork, Fortify, or Checkmarx, to catch security problems and common coding mistakes as they are writing code. Incremental static analysis pre-commit and commit checks can also be wired into Continuous Integration to catch common mistakes and anti-patterns quickly (full scans, which can take several hours to run on a large code base, need to be run separately, outside of the pipeline). It's important to tune these tools to minimize false positives, in order to provide developers with simple, actionable, pass/fail feedback.

Wiring Security Tests into CI/CD

Scanning code for common security vulnerabilities and coding mistakes isn't enough. Developers need to include security testing as part of their automated unit and integration tests for security features and functions: positive and negative tests on authentication, access control, and auditing functions and security libraries. Write positive and negative API-level integration tests to check that security functions are working correctly: that you can't perform an action if you haven't authenticated, that you can't see or change information for a different account, and so on.

Then script system-level attacks in Continuous Delivery using tools that behave well in CI/CD, like Gauntlt/, Mittn, or BDD-Security. Some common tests that can be done using tools like Gauntlt include using nmap to check for open ports, checking that SSL is configured correctly, attempting SQL injections, and testing for high-severity vulnerabilities like Heartbleed.

Coming up with good tests takes a good understanding of the application, the runtime environment, and security tools, bringing developers, Ops, and InfoSec together. Like automating integration testing or acceptance testing, it will take a while to build up a strong set of security tests in Continuous Delivery. Start by building a security smoke test: a basic regression test that can be run early in the pipeline and in production to catch common and important security problems, and to ensure that security configurations are correct.

Automating security testing makes it easy to collect metrics on the security posture of the application, and to make this information available to everyone—development, Ops, InfoSec, and compliance—as part of the team's CI/CD dashboard.

There is still a place for pen tests and comprehensive security audits in Continuous Delivery, and not just to meet regulatory requirements. The real value in a pen test or a security audit is as a health check on the effectiveness of your security practices and controls. Treat the results the same as a production failure. Run them through a postmortem review to understand the root causes: what you need to improve in your training, reviews, testing, and other checks; what you need to change in your design or coding practices. Just like with a production failure, it's not enough to fix the problem. You have to make sure to prevent problems from happening again.

Supply Chain Security: A System Is Only as Secure as the Sum of Its Parts

Today's Agile and DevOps teams take extensive advantage of open source libraries to reduce development time and costs—which means that they also inherit quality problems and security vulnerabilities from other people's code.

According to Sonatype (who run the Central Repository, the world's largest repo for open source software), 80% of application code today comes from libraries and frameworks—and a lot of this code has serious problems in it.[11] They looked at 17 billion download requests from 106,000 different organizations in 2014 and found that:

11 See *http://bit.ly/1OfrDig*.

> Large software and financial services companies are using an average of 7,600 suppliers. These companies sourced an average of 240,000 software “parts” in 2014, of which 15,000 included known vulnerabilities.[12]

More than 50,000 of the software components in the Central Repository have known security vulnerabilities. One in every 16 download requests is for software with at least one known security vulnerability. And on average, 50 new critical vulnerabilities in open source software are reported every day.

Scared yet? You should be. You need to know what open source code is included in your apps and when this changes, and review this code for known security vulnerabilities.

Luckily, this can be done automatically. Open source tools like OWASP’s Dependency Check and commercial tools like Sonatype Nexus Lifecycle can be wired into the CI/CD pipeline to detect open source dependencies, identify known security vulnerabilities, and fail the build automatically if serious problems are found.

Secure Infrastructure as Code

The same ideas and controls need to be followed when making changes to infrastructure. This can easily be done using modern configuration management tools like Puppet, Chef, and CFEngine.

These tools make it easy to set up standardized configurations across the environment using templates, minimizing the security risk that one unpatched server can be exploited by hackers, as well as the operational risks of a server being set up incorrectly (as we saw in the Knight case study). All the configuration information for the managed environment is visible in a central repository, and under version control. This means that when a vulnerability is reported in a software component like OpenSSL, it is easy to identify which systems need to be patched, and it is easy to push the patch out too. These tools also provide file integrity monitoring and give you control over configuration drift: they automatically audit runtime configurations to make sure that they match definitions, alert when something is missing or wrong, and automatically correct it.

12 Source: *http://bit.ly/1ig8HE4*.

Puppet manifests and Chef cookbooks need to be written and reviewed with security in mind. Unit tests for Puppet and Chef should include security checks. Build standard hardening steps into your recipes, instead of using scripts or manual checklists. Security standards like the Center for Internet Security (CIS) benchmarks and NIST requirements can be burned into Puppet and Chef definitions. There are several examples of Puppet modules and Chef cookbooks available to help harden Linux systems against security guidelines like CIS and the Defense Information Systems Agency's Security Technical Implementation Guide (STIG).

Security Doesn't End with Development or Deployment

Another key part of DevOpsSec is tying security into application monitoring and metrics and runtime checks.

Security monitoring in many enterprises is the responsibility of a Security Operations Center (SOC), manned by security analysts who focus on anomalies in network traffic. But security also needs to be tied into application and operations monitoring. This means building instrumentation and intrusion detection into the application using a design framework like OWASP's AppSensor, and making application attack data and other anomalies visible to operations and developers, as well as to the SOC. This enables what Zane Lackey at Signal Sciences calls "attack-driven defense".

Security runtime checks should also be done as part of application operations. Netflix's Security Monkey and Conformity Monkey illustrate the kinds of automated continuous checks that can be done in online services. These are rule-driven services that automatically monitor the environment to detect changes and to ensure that configurations match predefined definitions, looking for violations of security policies and common security configuration weaknesses (in the case of Security Monkey) or configurations that do not meet predefined standards (Conformity Monkey). They run periodically online, notifying operations and InfoSec when something is wrong.

While checks like these are particularly important in a public cloud environment like Netflix's where changes are constantly being made by developers, the same ideas can be extended to any system, constantly checking to ensure that systems are always set up correctly and safely.

Continuous Delivery (and DevOps) as a Security Advantage

A major problem that almost all organizations face is that even when they know that they have a serious security vulnerability in a system, they can't get the fix out fast enough to stop attackers from exploiting the vulnerability.

The longer vulnerabilities are exposed, the more likely it is that the system will be, or has already been, attacked. WhiteHat Security, which provides a service for scanning websites for security vulnerabilities, regularly analyzes and reports on vulnerability data that it collects. Using data from 2013 and 2014, WhiteHat found that 35% of finance and insurance websites were "always vulnerable," meaning that these sites had at least one serious vulnerability exposed every single day of the year. Only 25% of finance and insurance sites were vulnerable for less than 30 days of the year. On average, serious vulnerabilities stayed open for 739 days, and only 27% of serious vulnerabilities were fixed at all, because of the costs, risks, and overhead involved in getting patches out.[13]

Continuous Delivery, and collaboration between developers, operators, and InfoSec staff working closely together, can close vulnerability windows. Most security patches are small and don't take long to code. A repeatable, automated Continuous Delivery pipeline means that you can figure out and fix a security bug or download a patch from a vendor, test to make sure that it doesn't introduce a regression, and get it out quickly, with minimal cost and risk. This is in direct contrast to "quick fixes" done under pressure that have resulted in failures in the past.

13 See *https://www.whitehatsec.com/press-releases/featured/2015/05/21/pressrelease.html*.

Callout: The Honeymoon Effect

There appears to be another security advantage to moving fast in DevOps. Recent research shows that smaller, more frequent changes may make systems safer from attackers, through "the Honeymoon Effect".

Legacy code with known vulnerabilities is a more common and easier point of attack. New code that is changed frequently is harder for attackers to follow and understand, and once they understand it, it might change again before they can exploit a vulnerability. Sure, this is a case of "security through obscurity"—a weak defensive position—but it could offer an edge to fast-moving organizations.

Security Can No Longer Be a Blocker

In DevOps, "security can no longer be a blocker—in places where this is part of the culture, a big change will be needed."[14] Information security needs to be engaged much closer to development and operations, and security needs to become part of development and operations, how they think and how they work. This means security has to become more engineering-oriented and less audit-focused, and a lot more collaborative—which is what DevOps is all about.

Compliance as Code

Earlier we looked at the extensive compliance obligations that financial organizations have to meet. Now let's see how DevOps can be followed to achieve what Justin Arbuckle at Chef calls "Compliance as Code": building compliance into development and operations, and wiring compliance policies and checks and auditing into Continuous Delivery, so that regulatory compliance becomes an integral part of how DevOps teams work on a day-to-day basis.

One way to do this is by following the DevOps Audit Defense Toolkit, a free, community-built process framework written by James

14 Quote from Zane Lackey of Signal Sciences in discussion with the author, August 11, 2015.

DeLuccia IV, Jeff Gallimore, Gene Kim, and Byron Miller. The Toolkit builds on real-life examples of how DevOps is being followed successfully in regulated environments, on the Security as Code practices that we've just looked at, and on disciplined Continuous Delivery.[15] It's written in case study format, describing compliance at a fictional organization, laying out common operational risks and control strategies, and showing how to automate the required controls.

Up-Front Policies

Compliance as Code brings management, compliance, internal auditors, the project management office, and InfoSec to the table, together with development and operations. Compliance policies and rules and control workflows need to be defined up front by all of these stakeholders working together. Management needs to understand how operational risks and other risks will be controlled and managed through the pipeline. Any changes to policies, rules, or workflows need to be formally approved and documented, for example in a Change Advisory Board (CAB) meeting.

But instead of relying on checklists and procedures and meetings, the policies and rules are enforced (and tracked) through automated controls, which are wired into the Continuous Delivery pipeline. Every change ties back to version control and a ticketing system for traceability and auditability: all changes have to be made under a ticket, and the ticket is automatically updated along the pipeline, from the initial request for work all the way to deployment.

Automated Gates and Checks

The first approval gate is mostly manual. Every change to code and configuration must be reviewed pre-commit. This helps to catch mistakes, and makes sure that no changes are made without at least one other person checking to make sure that they were done correctly. High-risk code (defined by the team, management, compliance, and InfoSec) must also have a subject-matter expert (SME) review: for example, security-sensitive code must be reviewed by a security expert. Periodic checks are done by management to ensure that reviews are being done consistently and responsibly, and that

15 For example, see how Etsy supports PCI DSS: *http://bit.ly/1UD6J1y*.

no “rubber stamping” is going on. The results of all reviews are recorded in the ticket. Any follow-up actions that aren’t immediately addressed are added to the team’s backlog as another ticket.

In addition to manual reviews, automated static analysis checking is also done to catch common security bugs and coding mistakes (in the IDE, and in the CI/CD pipeline). Any serious problems found will fail the build.

Once checked in, all code is run through the automated test pipeline. The Audit Defense Toolkit assumes that that the team follows test-driven development, and outlines an example set of tests that should be executed.

Infrastructure changes are done using an automated configuration management tool like Puppet or Chef, following the same set of controls:

- Changes are code reviewed pre-commit.
- High-risk changes (again, as defined by the team) must go through a second review by an SME.
- Static analysis/lint checks are done automatically in the pipeline.
- Automated tests are executed using a test framework like rspec-puppet, Chef Test Kitchen, or Serverspec.
- Changes are deployed to test and staging in sequence with automated smoke testing and integration testing.

And again, every change is tracked through a ticket and logged.

Managing Changes

Because DevOps is about making small changes, the Audit Defense Toolkit assumes that most changes can be treated as standard (routine): changes that are essentially preapproved by management and therefore do not require CAB approval.

It also assumes that bigger changes will be made “dark”: that is, that they will be made in small, safe, and incremental steps, protected behind runtime feature switches that are turned off by default. The features will only be fully rolled out with coordination between development, Ops, compliance, and other stakeholders.

Any problems found in production are reviewed through postmortems, and tests are added back into the pipeline to catch the problems (following TDD principles).

Code Instead of Paperwork

Compliance as Code tries to minimize paperwork and overhead. You still need clear, documented policies that define how changes are approved and managed, and checklists for procedures that cannot be automated. However, most of the procedures and the approval gates are enforced through automated rules in the CI/CD pipeline, and you can lean on the automated pipeline to ensure that all of the steps are followed consistently and take advantage of the detailed audit trail that gets automatically created.

This lets developers and operations engineers make changes quickly and safely, although it does require a high level of engineering discipline. And in the same way that frequently exercising build and deployment steps reduces operational risks, exercising compliance on every change, following the same standardized process and automated steps, reduces the risks of compliance violations. You—and your auditors—can be confident that all changes are made the same way, that all code is run through the same tests and checks, and that everything is tracked the same way: consistent, complete, repeatable, and auditable.

Standardization makes auditors happy. Audit trails make auditors happy (obviously). Compliance as Code provides a beautiful audit trail for every change, from when the change was requested and why, to who made the change and what they changed, who reviewed the change and what they found in their review, how and when the change was tested, and when it was deployed. Except for the discipline of setting up a ticket for every change and tagging changes with a ticket number, compliance becomes automatic and seamless to the people who are doing the work.

Just as beauty is in the eye of the beholder, compliance is in the opinion of the auditor. Auditors may not understand or agree with this approach at first. You will need to walk them through it and prove that the controls work—but that shouldn't be too difficult. As Dave Farley of Continuous Delivery Ltd put it in a conversation in July 2015:

> I have had experience in several finance firms converting to Continuous Delivery. The regulators are often wary at first, because Continuous Delivery is outside of their experience, but once they understand it, they are extremely enthusiastic. So regulation is not really a barrier, though it helps to have someone that understands

> the theory and practice of Continuous Delivery to explain it to them at first.
>
> If you look at the implementation of a deployment pipeline, a core idea in Continuous Delivery, it is hard to imagine how you could implement such a thing without great traceability. With very little additional effort the deployment pipeline provides a mechanism for a perfect audit trail. The deployment pipeline is the route to production. It is an automated channel through which all changes are released. This means that we can automate the enforcement of compliance regulations—"No release if a test fails," "No release if a trading algorithm wasn't tested," "No release without sign-off by an authorised individual," and so on. Further, you can build in mechanisms that audit each step, and any variations. Once regulators see this, they rarely wish to return to the bad old days of paper-based processes.

Continuous Delivery Versus Continuous Deployment

The DevOps Audit Defense Toolkit tries to make a case to an auditor for Continuous Deployment in a regulated environment: that developers, following a consistent, disciplined process, can safely push changes out automatically to production once the changes pass all of the reviews and automated tests and checks in the CD pipeline.

Continuous Deployment has been made famous at places like Flickr, IMVU (where Eric Ries developed the ideas for the Lean Startup method), and Facebook:

> Facebook developers are encouraged to push code often and quickly. Pushes are never delayed and [are] applied directly to parts of the infrastructure. The idea is to quickly find issues and their impacts on the rest of the system and surely [fix] any bugs that would result from these frequent small changes.[16]

While organizations like Etsy and Wealthfront (who we will look at later) work hard to make Continuous Deployment safe, it is scary to auditors, to operations managers, and to CTOs like me who have been working in financial technology and understand the risks involved in making changes to a live, business-critical system.

16 E. Michael Maximilien, "Extreme Agility at Facebook", November 11, 2009.

Continuous Deployment requires you to shut down a running application on a server or a virtual machine, load new code, and restart. This isn't that much of a concern for stateless web applications with pooled connections, where browser users aren't likely to notice that they've been switched to a new environment in Blue-Green deployment.[17] There are well-known, proven techniques and patterns for doing this that you can follow with confidence for this kind of situation.

But deploying changes continuously during the day at a stock exchange connected to hundreds of financial firms submitting thousands of orders every second and where response times are measured in microseconds isn't practical. Dropping a stateful FIX session with a trading counterparty and reconnecting, or introducing any kind of temporary slowdown, will cause high-speed algorithmic trading engines to panic. Any orders that they have in the book will need to be canceled immediately, creating a noticeable effect on the market. This is not something that you want to happen *ever*, never mind several times in a day.

It is technically possible to do zero-downtime deployments even in an environment like this, by decoupling API connection and session management from the business logic, automatically deploying new code to a standby system, starting the standby and primary systems up, and synchronizing in-memory state between the systems, triggering automated failover mechanisms to switch to the standby, and closely monitoring everything as it happens to make sure that nothing goes wrong.

But do the benefits of making small, continuous changes in production outweigh the risks and costs involved in making all of this work?

During trading hours, every part of every financial market system is expected to be up and responding consistently, all the time. But unlike consumer Internet apps, financial systems don't need to run 24/7/365. This means that most financial institutions have mainte-

17 In Blue-Green deployment, you run two production environments ("blue" and "green"). The blue environment is active. After changes are rolled out to the green environment, customer traffic is rerouted using load balancing from the blue to the green environment. Now the blue environment is available for updating.

nance windows where they can safely make changes. So why not continue to take advantage of this?

Some proponents of Continuous Deployment argue that if you don't exercise your ability to continuously push changes out to production, you cannot be certain that it will work if you need to do it in an emergency. But you don't need to deploy changes to production 10 or more times per day to have confidence in your release and deployment process. As long as you have automated and standardized your steps, and practiced them in test and exercised them in production, the risks of making a mistake will be low.

Another driver behind Continuous Deployment is that you can use it to run quick experiments, to try out ideas for new features or to evaluate alternatives through A/B testing. This is important if you're an online consumer Internet startup. It's not important if you're running a stock exchange or a clearinghouse. While a retail bank may want to experiment with improvements to its consumer website's look and feel, most changes to financial systems need forward planning and coordination, and advance notice—not just to operations, but to partners and customers, to compliance and legal, and often to regulators.

Changes to APIs and reporting specifications have to be certified with counterparties. Changes to trading rules and risk management controls need to be approved by regulators in advance. Even algorithmic trading firms that are constantly tuning their models based on live feedback need to go through a testing and certification process when they make changes to their code.

In order to minimize operational and technical risk, financial industry regulators are demanding *more* formal control over and transparency in changes to information systems, not less. New regulations like Reg SCI and MiFID II require firms to plan out and inform participants and regulators of changes in advance; to prove that sufficient testing and reviews have been completed before (and after) changes are made to production systems; and to demonstrate that management and compliance are aware of, understand, and approve of all changes.

It's difficult to reconcile these requirements with Continuous Deployment—at least, for heavily regulated core financial transaction processing systems. This is why we focus on Continuous Delivery in this book, not Continuous Deployment.

Both approaches leverage an automated testing and deployment pipeline, with built-in auditing. With Continuous Delivery, changes are *always ready* to be deployed—which means that if you need to push a fix or patch out quickly and with confidence, you can. Continuous Delivery also provides a window to review, sign off on, and schedule changes before they go to production. This makes it easier for DevOps to work within ITIL change management and other governance frameworks, and to prove to regulators that the risk of change is being managed from the top down. Continuous Delivery puts control over system changes clearly into the hands of the business, not developers.

DevOps for Legacy Systems

Introducing Continuous Delivery, Infrastructure as Code, and similar practices into a legacy environment can be a heavy lift. There are usually a lot of different technology platforms and application architectures to deal with, and outside of Linux and maybe Windows environments, there isn't a lot of good DevOps tooling support available yet for many legacy systems.

From Infrastructure to Code

It's a massive job for an enterprise running thousands of apps on thousands of servers to move its infrastructure into code. Even with ITIL and other governance frameworks, many enterprises aren't sure how many applications they run and where they are running, never mind the details of how the systems are configured. How are they supposed to get this information into code for tools like Chef, Puppet, and Ansible?

This is what a tech startup called ScriptRock is taking on. ScriptRock's cloud-based service captures configuration details from running systems (physical or virtual servers, databases, or cloud services), and tracks changes to this information over time. You can use it as a Tripwire-like detective change control tool, to alert on changes to configuration and track changes over time, or to audit and visualize configuration management and identify inconsistencies and vulnerabilities.

ScriptRock takes this much further, though. You can establish policies for different systems or types of systems, and automatically create fine-grained tests to check that the correct version of software is installed on a system, that specific files or directories exist, that specific ports are open or closed, or that certain processes are running. ScriptRock can also generate manifests that can be exported into tools like Puppet, Chef, or Ansible, or Microsoft PowerShell DSC or Docker. This allows you to bring infrastructure configuration into code in an efficient and controlled way, with a prebuilt test framework.

IBM and other enterprise vendors are jumping in to fill in the tooling gap, with upgraded development and automated testing tools, cross-platform release automation solutions, and virtualized cloud services for testing. Organizations like Nationwide Insurance are implementing Continuous Integration and Continuous Delivery on zSeries mainframes, and a few other success stories prove that DevOps can work in a legacy enterprise environment.

There's no reason not to try to speed up development and testing, or to shift security left into design and coding in any environment. It's just good sense to make testing and production configurations match; to automate more of the compliance steps around change management and release management; and to get developers more involved with operations in configuring, packaging, deploying, and monitoring the system, regardless of technology issues.

But you will reach a point of diminishing returns as you run into limits of platform tooling and testability. According to Dave Farley:[18]

> Software that was written from scratch, using the high levels of automated testing inherent in Continuous Delivery looks different from software that was not. Software written using automated testing to drive its design is more modular, more loosely coupled, and more flexible—it has to be to make it testable. This imposes a barrier for companies looking to transition. There are successful strategies to make this transition but it is a challenge to the development culture, both business and technical, and at the technical level in terms of "how do you migrate a legacy system to make it testable?"

Legacy constraints in large enterprises lead to what McKinsey calls a "two-speed IT architecture", where you have two types of systems:

18 Dave Farley of Continuous Delivery Ltd in discussion with the author, July 24, 2015.

1. Slower-changing legacy backend "systems of record," where all the money is kept and counted
2. More agile frontend "systems of engagement," where money is made or lost—and where DevOps makes the most sense

DevOps adoption won't be equal across the enterprise—at least, not for a long time. But DevOps doesn't have to be implemented everywhere to realize real benefits. As the Puppet Labs "2015 State of DevOps Report" found:

> It doesn't matter if your apps are greenfield, brownfield or legacy—as long as they are architected with testability and deployability in mind, high performance is achievable… The type of system—whether it was a system of engagement or a system of record, packagedor custom, legacy or greenfield—is not significant. Continuous Delivery can be applied to any system.

Implementing DevOps in Financial Markets

The drivers for adopting better operations practices in financial enterprises are clear. The success stories are compelling. There are challenges, as we've seen—but these challenges can be overcome.

So, where to start?

DevOps in the end is about changing the way that IT is done. This can lead to fundamental changes in the structure and culture of an entire organization. Look at what ING and Capital One did, and are still doing.

Wealthfront: A Financial Services Unicorn

There are already DevOps unicorns in the financial industry, as we've seen looking at LMAX, ING, and Capital One. Wealthfront is another DevOps unicorn that shows how far DevOps ideas and practices can be taken in financial services.

Wealthfront, a retail automated investment platform ("robo advisor") that was launched in 2011, is not a conventional financial services company. It started as an online portfolio management game on Facebook called "KaChing," and then, following Eric Ries's Lean Startup approach, continued to pivot to its current business model. Today, Wealthfront manages $2.5 billion in assets for thousands of customers.

Wealthfront was built using DevOps ideas from the start. It follows Continuous Deployment, where changes are pushed out by developers directly, 10 or 20 or 50 or more times per day, like at Etsy. And, like at Etsy, Wealthfront has an engineering-driven culture where developers are encouraged to push code changes to production on their first day of work. But this is all done in a highly regulated environment that handles investment money and private customer records.

How do they do it? By following many of the practices and ideas described in this book—to the extreme.

Developers at Wealthfront are obsessed with writing good, testable code. They enforce consistent coding standards, run static analysis (dependency checks, identifying forbidden function calls, source code analysis with tools like FindBugs and PMD to find bad code and common coding mistakes), and review all code changes. They've followed test-driven development from the beginning to build an extensive automated test suite. If code coverage is too low in key areas of the code, the build fails. Every couple of months they run Fix-It days to clean up tests and improve test coverage in key areas. The same practices are followed for infrastructure changes, using Chef.

Wealthfront engineers' priorities are to optimize for safety as well as speed. The company continually invests in its platforms and tools to make it easy for engineers to do things the right way by default. They routinely dark launch new features; they use canary deployments to roll changes out incrementally; and they've built a runtime "immune system," as described in the Lean Startup methodology, to monitor logs and key application and system metrics after changes are deployed and automatically roll back the most recent change if it looks like something is going wrong.

Wealthfront has no operations staff or QA staff: the system is designed, developed, tested, and run by engineers. All of this sounds more like an engineering-driven Internet startup than a financial services provider, and Wealthfront is the exception, rather than the rule—at least, for now.[19]

Books like Gary Gruver and Tommy Mouser's *Leading the Transformation* (IT Revolution) and Jez Humble, Joanne Molesky, and Barry O'Reilly's *Lean Enterprise* (O'Reilly) can help you understand how to implement Agile and DevOps in large-scale programs, how to manage cultural change within the organization, secure executive sponsorship, and shift toward Lean thinking across development and IT operations and across the business as a whole.

Organizational change on this scale is expensive and risky. DevOps can also be implemented incrementally, in small batches, from the ground up, by building first on Agile development. Start by creating self-service tools and putting them into the hands of developers, and making testing more streamlined and efficient.

There's a lot to be gained by going after obvious pain points first, like manual configuration and deployment. As one example, just by implementing automated deployment, Fidelity Worldwide Investment was able to speed up development and testing on important trading applications, significantly reducing time to market and saving millions of dollars per year.[20]

Other initiatives like this are already underway in many financial organizations. Some of them are creating cross-functional DevOps teams like Capital One did to start: teams focused on automating builds and release engineering, automating testing, extending Continuous Integration into Continuous Delivery.

19 This profile is based on public presentations by Wealthfront employees, information published on Wealthfront's engineering blog, and a conversation with CTO David Fortunato on August 21, 2015.

20 See *http://www.ibm.com/ibm/devops/us/en/casestudies/fidelity.html*.

While some practitioners see DevOps teams as an anti-pattern,[21] these teams can help bridge silos between development, operations, compliance, and InfoSec; open up communications; identify and deal with inefficiencies; and bootstrap the adoption of new practices and different ways of thinking and problem solving.

Where I work, we didn't know about DevOps when we started down this path—but DevOps happened anyway. When we launched the business, the CEO made it clear that we all shared the same goals: to ensure the integrity, reliability, and regulatory compliance of the service that we offered to our customers.

After we went live, we had to switch from a project delivery mindset to an operational one. This meant putting operational readiness and risk management ahead of features and schedules; spending more time on change control, building in backward compatibility, testing failover and rollback, preventing alert storms, and writing health checks.

We started making smaller changes, because smaller changes were easier to test and safer to deploy, and because working this way helped us to keep up with rapidly changing operational and support requirements as more customers came on board. And because we were making smaller changes, and making them more often, we had to automate more of the steps in delivery: testing and compliance checks, system provisioning and configuration, deployment. The more that we automated this work, the safer and easier it was for us to make changes. The more often that we made changes, the better we got at it, and the closer developers and operators became.

In my organization, operations and development are separate organizational silos reporting up to different executives, in different cities. We also have independent QA. Although we adopted a culture of code reviews and built our automated Continuous Integration platform a long time ago, and we continue automating checks and tests and deployment steps in Continuous Delivery, we rely on the QA team's manual testing and reviews to catch edge conditions and to hunt for bugs and look for holes in our automated test suites. Their job—and their value—is to identify risks, to make sure our controls are effective, and to help us improve.

21 See *http://www.thoughtworks.com/insights/blog/there-no-such-thing-devops-team*.

We have these organizational silos because they help us to maintain control over change, to minimize security risks, and to meet compliance and governance requirements. This structure doesn't get in the way of people working together. Developers and QA and Ops collaborate closely on design and problem solving, setting up and configuring environments, conducting security reviews, coordinating changes, responding to incidents. But market operations and QA and compliance decide if and when changes go into production—not developers. Deployment is done by operations, after the reviews and checks are complete, with developers watching closely and standing by. We don't do Continuous Deployment, or anything close to it. But we can still make changes quickly, taking advantage of automation and agility. This is DevOps—just a different kind of DevOps.

In the financial industry, regulators, compliance, risk managers, and InfoSec are all concerned that business lines and development put speed of delivery ahead of safety, security, and reliability. For us, and for other financial firms, adopting DevOps practices like Continuous Delivery, Infrastructure as Code, and improved collaboration between developers and operations engineers is about reducing operational and technical risks, improving efficiency, and increasing transparency—not just improving time to market. Done this way, the ROI case for DevOps seems clear. An approach to managing IT changes that reduces both time to delivery and operational costs, minimizes technical and operational risks, and at the same time makes compliance happy? That's a win, win, win.

About the Author

Jim Bird is a CTO, software development manager, and project manager with more than 20 years of experience in financial services technology. He has worked with stock exchanges, central banks, clearinghouses, securities regulators, and trading firms in more than 30 countries. He is currently the CTO of a major US-based institutional alternative trading system.

Jim has been working in Agile and DevOps environments in financial services for several years. His first experience with incremental and iterative ("step-by-step") development was back in the early 1990s, when he worked at a West Coast tech firm that developed, tested, and shipped software in monthly releases to customers around the world—he didn't realize how unique that was at the time. Jim is active in the DevOps and AppSec communities, is a contributor to the Open Web Application Security Project (OWASP), and occasionally helps out as an analyst for the SANS Institute.