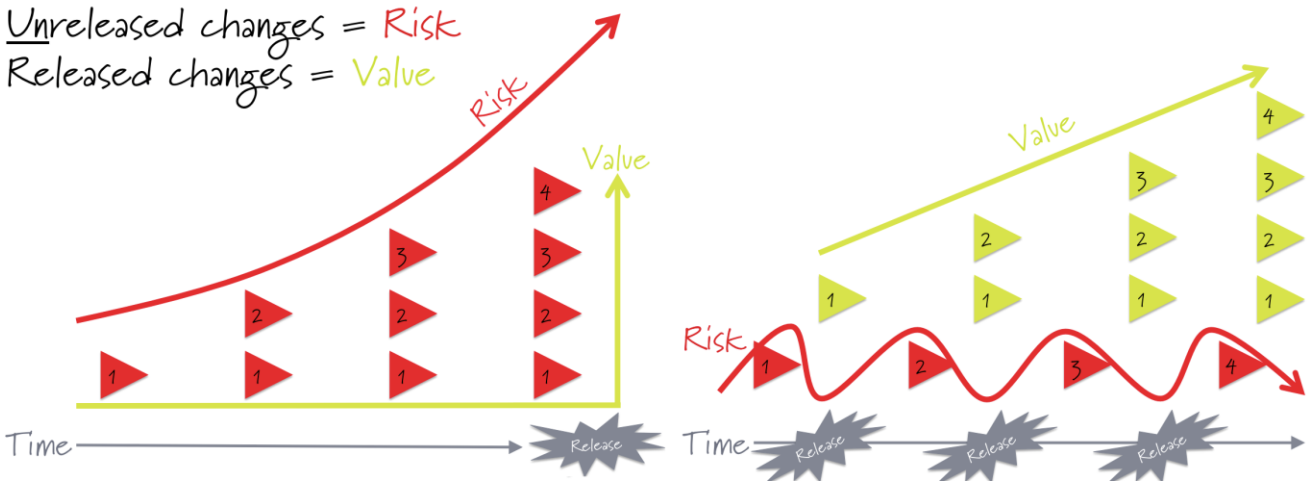
Continuous Delivery Overview

Value proposition, definitions, principles, core practices, and tools

Luca Minudel



InfoQ
ENTERPRISE SOFTWARE
DEVELOPMENT SERIES

Contents

Introduction

This booklet is a rapid and comprehensive overview of continuous delivery (CD) for busy people. It introduces the principles, the practices, and the value proposition of CD. It contains references to the most important resources and it ends with anecdotes and war stories about CD adoptions.

When deadlines loom, often all certainties about the real progress of a project disappear. Sometimes significant problems are discovered only late into the integration or hardening phase. Other times it is the release to production that becomes a lengthy and painful experience, followed by weeks of bug-fixing and firefighting.

Often a new product is released after many months of waiting and hard work. Only then can the business see if the product solves the right problem, if it fits the real needs of the customers, if the market likes it. Other times, even after the release, no one really knows which features are used and generate value and which do not.

CD is that part of lean and agile approaches to software development that promises to make real project progress always obvious and tangible, supports early discovery and gradual development of customers and products, and eliminates the gap between customers, business, and IT. In short, CD turns risky, painful releases to production into safe and regular events.

Defining continuous delivery

Software development is often seen as a bottleneck when developing a new product. What if the business can decide to release new features in production at any time? This is the central concept in the definition of CD.

Continuous Delivery is a software development discipline where you build software in such a way that the software can be released to production at any time - Martin Fowler.

Continuous Delivery aims to reduce the cost, time, and risk of delivering incremental changes to users - Jez Humble.

Furthermore, CD in software development is a set of collaboration practices and engineering techniques, skills and tools that allow software products to be conceived and developed to a high standard and then easily deployed, resulting in the ability to promptly, rapidly, reliably, and repeatedly push out new features, enhancements, and bug fixes at low risk and with minimal overhead.

The CD working group at ThoughtWorks says that you're doing CD when:

- your software is deployable throughout its lifecycle
- your team prioritizes keeping the software deployable over working on new features
- anybody can get fast, automated feedback on the production readiness of their systems whenever somebody makes a change to them
- you can perform push-button deployments of any version of the software to any environment on demand.

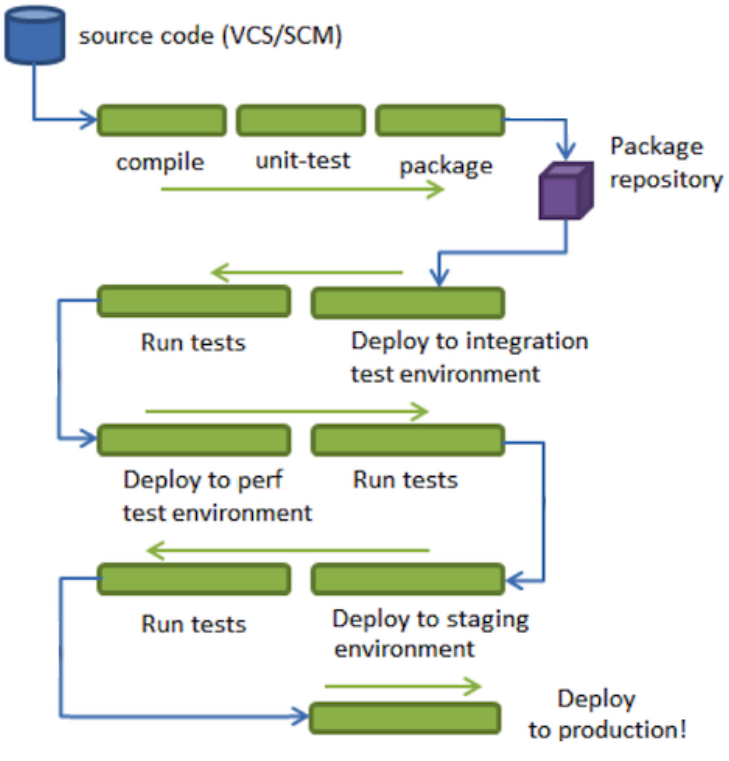
Following the modern industry trend of shortening products' life cycles, more and more teams and organizations release-cycles are moving from years and months to weeks and days. This allows repeatable, safe, weekly - or daily - releases to become the norm.

Suggested readings:

- *Continuous Delivery*, Jez Humble and David Farley: Addison-Wesley (2010)
- Delivery Guide, Martin Fowler
- Continuous Delivery, Martin Fowler, (2013)
- Continuous Delivery (video), Martin Fowler and Jez Humble (2011)
- Jez Humble on Continuous Delivery (video), Jez Humble (2012)

The deployment pipeline

The deployment pipeline is a first-class concept of a CD implementation. A deployment pipeline models and visualises your process for getting software from version control into the hands of your users. It implements and automates that process for each stage that every change to your software goes through, from check-in to release – and it may also contain a few manual stages such as approvals. What's more, it visualises in real time the status of the code base. The following image is an example of a pipeline.



With dependencies between pipelines, it is possible to model the dependencies between different software products or components. As a consequence, it is possible to monitor in real time the status of complex changes that involve or affect multiple products and components. This greatly simplifies program management.

Why continuous delivery?

IT operations aims to keep the production systems up and running in a stable way. Product development aims to release new, live features to stay ahead of the competition.

These two goals have traditionally been seen as conflicting, like in a zero-sum game. Because of this, many organizations' governance and policies are seen as requiring expensive solutions to search for an acceptable trade-off between the two.

Surprisingly, CD has the paradoxical effect of increasing the throughput of new features and stabilizing production systems, increasing innovation and predictability together at the same time. Furthermore, CD closes the gap between market, business, and IT.

Jez Humble emphasizes three main benefits of CD: building the right thing; reducing the risk of release; and making real project progress obvious and tangible. Let's look at common motivators and inhibitors for adopting CD.

Compelling needs

For some organizations in some contexts and for some projects, CD is more than an opportunity - it is a compelling need. An example is the situation where the following two forces are present at the same time:

1. Pressure to change
    - pressure to innovate (i.e. stay ahead of competitors);
    - technical novelty and variety in the technologies used for the project;
    - uncertainty (e.g. from the domain, the market, the users, the partners, the suppliers)

2. Need for stability
    - interruption or disruption of services provided cause significant loss of money;
    - services provided are life-critical

This is where CD comes into play in enabling an organization to increase the throughput, innovation, and stability all at the same time.

Opportunities for improvement and excellence

For some organizations, CD is an opportunity for improvement and excellence. The following is a list of major benefits of CD:

- **Reduced risks**
  Thanks to early and frequent releases, it is possible to see if the organization is building the right thing in the right way. The real progress of the project becomes tangible in every moment, and you overcome the infamous 90% Syndrome.

  Problems are detected early, when it is easier and cheaper to fix them. The consequences of a mistake are limited by the small scope and size of each incremental release.

- **Reduced waste**
  Increased automation reduces the number of errors caused by repetitive manual work in testing and deployment, and reduces the probability of subtle errors caused by manual setup and configuration of the environments and infrastructure.

- **Increased quality**
  Lessons learned and information discovered while releasing the product early and often are reused immediately. This raises the internal and external quality, reducing defects and focusing on what really matters to users and customers.

- **Increased resilience**
  Remediation plans define how to recover from specific errors and showstopper bugs when a new deploy to production fails. Remediation plans are an integral part of CD. Remediation procedures are automated, tested, and put to use so that there will be no unpleasant surprises when they are needed. The small size of each release to production makes remediation plans simpler so that recovery is faster and easier with very limited consequences.

- **Increased responsiveness**
  The reduced lead-time cuts the time required to change direction, to make a change, to react to new circumstances, or to deal with an unexpected event. The small size of each release makes it faster to identify the reason for a problem and consequently the time to fix it.

- **Increased innovation**
  The ability to release early and often presents the opportunity to explore the market, to measure the value, to explore the fitness for purpose of the product, to conduct safe-to-fail experiments, to explore the reaction of users to the product, and so to discover new business opportunities. It enables the possibility of exaptation, one of the fundamental mechanisms of innovation.

CD is an advanced lean-agile practice. To adopt CD is a way to confirm your current lean-agile adoption and to advance it to the next level. In short, it is possible to say that CD increases agility and adherence to lean principles.

Indeed, the first principle behind the Agile Manifesto states:
*“Our highest priority is to satisfy the customer through early and **continuous delivery** of valuable software.”*

And one of the seven principles of lean software development states:
*“**Deliver** as **fast** as possible.”*

CD also spurs closer communication, mutual understanding, synchronization and cooperation between business and IT, and between all roles and departments involved in the project. In the same way, CD reduces the distance between the business and the customers.

Suggested readings:

- *The Lean Startup,* Eric Ries (2011)
- Agile Manifesto Principles, K.Beck, A.Cockburn, W.Cunningham, M.Fowler, J.Highsmith, R.Jeffries, R.C.Martin, K.Schwaber, J.Sutherland et al. (2001)

Inhibitors

There are internal and external factors that make it more difficult to adopt and apply CD. These include the following examples.

Internal factors:

- information silos; specialized functional teams; competition between business units and anything else that discourage and penalize communication and cooperation between departments and functions;
- high levels of confidentiality that prevent information-sharing internally and with partners and customers even in the presence of NDAs;
- disincentives and fear of failure; absence of safe-to-fail experiments;
- lack of training;
- resistance to change.

External factors:

- compliance with external regulations and policies that requires extensive processes; long paperwork trails or long waits before being authorized to publicly release a new version of the product;
- dependency on external resources (e.g. hardware, logistic, etc.) for which most decisions or actions are irreversible;
- dependency on suppliers, partners, customers or stakeholders that have a long response time or service time;
- use of new technologies that do not already have tools to support CD.

Contrary to popular belief, CD can also help to successfully go to market infrequently with huge marketing events. CD also helps with the compliance to standards such as SOX, PCI DSS, ITIL, and COBIT. And in many cases, problems with external policies are due to implementations of an interpretation of a policy, while there are other, lightweight ways to achieve the policy goals that actually produce better outcomes.

What is Is DevOps?, by Kief Morris

DevOps is a term coined by Patrick Debois, who organised a series of DevOpsDays conferences in 2009 to encourage people to think about software development and software support in a holistic way, as opposed to two separate activities. The responsibility for software over its lifecycle should not be divided into silos, with a development team throwing code "over the wall" to another team with different, potentially conflicting incentives.

DevOps is a natural extension of the agile software movement. Agile broke down barriers between customers, business analysts, architects, QAs, and developers, and now DevOps extends this collaboration to include infrastructure, release management, support, security, and other roles in IT operations.

DevOps doesn't require particular roles; in fact, many believe that creating a separate role for DevOps misses the point that developers and operations people should share ownership of the software, end to end. It also doesn't necessarily require changing organisational structures, although oftentimes existing structures, incentives, and culture creates an environment in which collaboration is difficult.

John Willis uses acronym CAMS for culture, automation, measurement, sharing – the main elements of DevOps. Culture is the foundation of DevOps, putting people and process first. Automation of infrastructure and change management can provide a shared platform around which development and operations can collaborate. Measurement is necessary for continuous improvement. And sharing between people and groups creates a loopback for continuous improvement.

Suggested readings:

- *Release It!: Design and Deploy Production-Ready Software,* Michael T. Nygard: The Pragmatic Programmers (2007)
- What Devops Means to Me, John Willis (2010)

Continuous integration, the second step of the CD journey

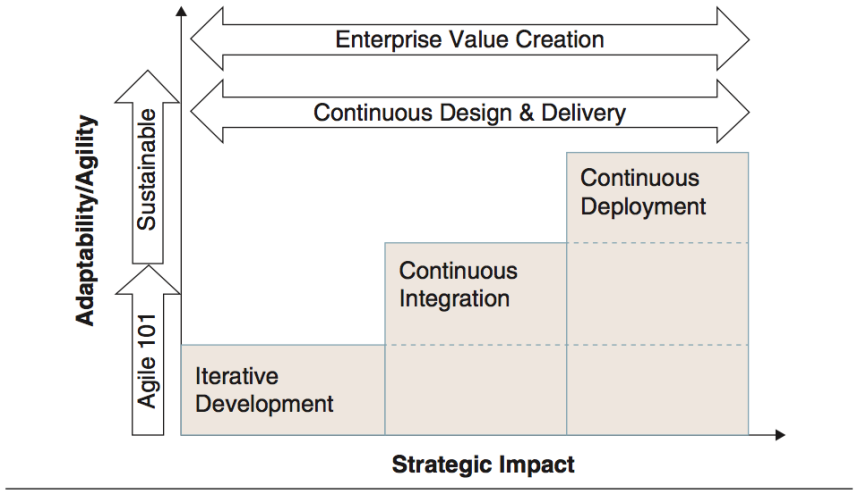
Many teams have implemented continuous integration (CI) servers, and use them every day to ensure high-quality code. An effective CI environment is a basic building block of CD – indeed, you cannot really implement CD without it.

The following is the verbatim definition of CI that Martin Fowler uses on his Web site:

Continuous Integration *is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily, leading to multiple integrations per day. Each integration is verified by an automated build including tests to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly.*

You are probably wondering what first step of the CD journey comes before CI. The following picture from Jim Highsmith's *Adaptive Leadership* answers the question: it is

iterative software development, which is a key element of lean and agile and of any modern software-development process.



Suggested readings:

- *Continuous Integration: Improving Software Quality and Reducing Risk,* Paul Duvall: Addison-Wesley Professional (2007)
- Continuous Integration, Martin Fowler (2006)

About continuous deployment

Continuous deployment is like continuous delivery with the addition that every single change that leads to a release-candidate version of the software is also automatically deployed to production.

In continuous delivery, the deployment to production is a business decision. In continuous deployment, the deployment to production is instead automatic. It's that simple.

Principles of CD

Different teams, different projects, and different organizations need different implementations of CD. There is no standardized version of CD. Instead, there is a mix of skills, tools, and technical and organizational practices. Finding the right mix can be a challenge.
Luckily, all the CD implementations share common principles that are precious foundations and can guide the CD journey.

This main principle incorporates the whole idea of CD: **create a repeatable, reliable way to release software**.

All remaining principles derive from that main principle.

The next two principles of CD are related to people and collaboration practices:

1. **Continuous improvement**
   A big-bang approach for the adoption of CD doesn't work well. Changing one thing at time and observing the results work better.

   A gradual approach is preferable, driven by all the parties involved and affected by CD, gathering together regularly to reflect on the next opportunity of improvement, to analyze feedback, to reach consensus, and to take actions.

2. **Everybody is responsible for the delivery process**
   Everybody is responsible for keeping the software functioning well in production and making sure the software constantly and increasingly generates business value.

   All the people involved throughout the value stream of the product, from concept to cash, should share this common goal and be evaluated as a team against this common goal.

   This is essential to creating an environment where communication and collaboration patterns required to succeed are possible and likely to happen.

The following five principles are related to technical practices:

1. **Keep everything in version control**
   Not only source code, but also libraries, automation scripts, configuration files, database creation and change scripts, environments and infrastructure creation, and configuration scripts.

2. **Automate almost everything**
   Indeed, automation is a requisite for CD. Automate the building, deployment, testing, and releasing.

   A new team member should be able to check out the project's artifacts from the version-control system and use a command to build and deploy the software to any available environment including the development workstation machine.

3. **If it hurts, do it more frequently**
   This is also an extreme-programming principle. Practice makes perfect. Solving a difficult problem with a gradual approach in small incremental steps is easier, and feedback at each step provides valuable information and lessons.

4. **Done means released**
   Done means a feature has been successfully released into production, used by profit-generating customers, and led to validated learning about the requirements, the technology, the market, and the people involved.

5. **Build quality in**
   Making mistakes is a natural human activity. There is a level beyond which trying to prevent mistakes is no longer effective and instead slows down the progress of the work.

> "Build quality in" means to add to the skills, the practices, and the tools of the team the ability to detect mistakes early, when it is cheaper to fix them and possible to fix them quickly and without consequences.

Collaboration practices for CD

Some people find principles too broad and abstract, and prefer to begin the CD journey from concrete practices. In addition to the principles, there is a set of well-known CD practices that are common to many CD implementations. The set of CD practices described below make a good starting point.

Collaboration practices are essential in CD because they support the technical practices and vice versa.

Some collaboration practices are related to cross-functional collaboration among technical roles. Other practices are related to cross-functional collaboration among technical roles, product development, marketing, and sales.

- **Developers and testers work together in the same team.**
  Testers and developers are assigned full time to the same team and share a common goal for each iteration. They both participate at the release and iteration planning and work together everyday from the beginning to the end of the iteration.

- **Developers and IT ops cooperate closely for the release and the support.**
  From the inception of the project, IT ops are involved in the release planning and in the iteration planning where they can express their requirements for the system to be developed and deployed to production and supported. Developers work and cooperate with IT ops to create the required tools, to write the required automation scripts, and to define, test, and execute the release and remediation plan. Both share the responsibility of keeping the system functioning well and profitably in production.

- **Product development, marketing, and sales cooperate with IT in defining the minimum viable product (MVP).**
  The person responsible for product development (a.k.a. product owner) cooperates with developers to first identify the smallest MVP that could be released as early as possible, followed by the definition of the set of smallest-possible minimum marketable features to be released incrementally. Marketing and sales also help to identify focus groups, internal users, potential beta users, and early adopters.

- **IT helps product development to monitor the product in production.**
  Developers and IT ops help to generate and collect information about the use of the product from users and customers in order to support business decisions.

The usual collaboration practices among developers working on the same project and on the same code base are also required to support the technical practices.

Suggested reading:

- *Extreme Programming Explained: Embrace Change,* Kent Beck: Addison-Wesley (2004)
- *The Lean Startup,* Eric Ries (2011)
- *The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win,* Gene Kim, Kevin Behr, and George Spafford (2013)

Large code base and large teams

When the code base grows and teams working on the code base grow in size and number, complexity and costs of communication grow exponentially.

A code base that has grown too large or too complex can be split into autonomous end-to-end business-facing components and services, and small cross-functional teams can be reorganized around these components and services in accordance with Conway's law:

organizations which design systems . . .
are constrained to produce designs
which are copies of the communication structures
of these organizations.

Software that is too complex to integrate, package, build, test with automatic tests, or deploy because the number or the complexity of the dependencies between components and service can indicate that the code base needs to be reorganized into autonomous components and services and assigned to small cross-functional teams.

Suggested readings:

- "A Conversation with Werner Vogels". *ACM Queue*, 4(4), (2006, May)

Technical practices of CD

Engineers as well as developers, testers, and IT ops usually think about CD starting from technical practices and from the new skill set that CD requires.

When they are given the responsibility and authority to contribute to the CD implementation, the technical practices become a challenging aspect that is also interesting and fulfilling. An engineer experienced in CD masters the technical practices and is capable of choosing the mix of practices that fits a specific team, project, and organization.

The technical practices of CD adhere to a new philosophy. These practices do not just try to prevent mistakes, defects, and problems. They build in the ability to detect them as early as possible, when it is easier to recover quickly and with few or no consequences at all.

Following this approach, software development goes faster, quality improves, and the whole workflow becomes more flexible.

The following paragraphs give an overview of the main technical practices of CD and are mainly interesting for readers with technical roles. Executives and managers may skip to the next chapter, "CD and DevOps tools, by Kief Morris".

Continuous integration (CI)

With CI, developers integrate and test their work at least once a day and detect integration problems early, when they are easier to solve. This approach avoids only discovering integration problems late into the integration or hardening phase before the release. CI is also essential to refactoring, because refactoring with feature branches causes a large number of merge conflicts that are impossible to resolve manually.

CI is one of the primary practices of agile software development and extreme programming and is a prerequisite for CD. Essentially, CD is an extension of CI.

In 2000, Martin Fowler and Matthew Foemmel from ThoughtWorks were the first to document the use of CI for large-scale software projects. This resulted in the first continuous-integration server, the open-source project CruiseControl, created by ThoughtWorks.

Suggested readings:

- *Continuous Integration: Improving Software Quality and Reducing Risk,* Paul Duvall: Addison-Wesley Professional (2007)
- Continuous Integration, Martin Fowler (2006)

Automated testing

Iterative software development is a de facto standard nowadays: new features are continuously added and existing features continuously modified and extended. Because of that, all QA and testing is repeated at every iteration. With CD, QA and testing get carried out after every single change to the system under development.

Manual QA and testing require many hours or even days of work. It is neither sustainable nor convenient to repeat manual QA and testing one or more times per day. That’s why testing automation is absolutely key here. Indeed, automated testing is one of the prerequisites for CD.

All the testing that can be automated (unit testing, integration testing, functional testing, etc.) needs to be automated, with exception of tests where humans add value (exploratory testing, usability testing, etc.).

In addition, automatic tests can be repeated many times without the inevitable errors, omissions, and delays caused by long and repetitive manual work. This increases quality, and the quick feedback decreases the time and cost of removing the identified defects.

Suggested readings:

- *Test Driven Development: By Example,* Kent Beck: Addison-Wesley Professional (2003)
- *Growing Object-Oriented Software Guided by Tests,* Steve Freeman and Nat Pryce: Addison-Wesley Professional (2009)
- *Specification by Example: How Successful Teams Deliver the Right Software,* Gojko Adzic: Manning Publications (2011)

Trunk-based development

Branching and merging configuration-management (CM) techniques are used to prevent and manage conflicting changes when parallel software developments are going on at the same time in the same code base. These techniques have huge hidden costs in terms of planning and process, time and effort, and risk and rigidity. Long integration and stabilization phases with an uncertain outcome must precede every release when you have long-lived branches, and many man-months of work can get lost because of merge conflicts that are hard or impossible to resolve.

Trunk-based development (TBD) is a fundamental practice of CD that replaces branching in modern software development because it increases development speed and flexibility while reducing effort and risks.

In TBD, all development is done on the mainline (also known as the head or trunk) of the source-code repository. All the developers commit the code to the mainline at least once per day. They commit only potentially releasable code using practices like latent-code patterns, feature toggles, and branch by abstraction. Also, every new release is built from the mainline. CI, automated testing, and iterative software development are technical practices that support TBD.

With TDB, release branches are used only to support older versions of a product. Experiment branches are used for experiments and spikes that are never merged back to the mainline. All other kinds of branches in TBD are avoided, and in any case a branch always lasts less than one working day.

Suggested readings:

- *What Is Trunk Based Development?,* Paul Hammant (2013)
- *Feature Branch,* Martin Fowler (2009)
- "Forces of Branching and Parallel Development", in *Branching Patterns for Parallel Software Development,* B. Appleton, S. Berczuk, R. Cabrera, and R. Orenstein (1998)

Latent-code patterns

Latent-code patterns are used to support TBD. They enable developers to commit frequently to the mainline only potentially releasable code even when the feature or change request

they are implementing is not yet finished – or when it is finished but the business decision to release the feature has not yet been taken.

Latent code is code that is committed to the mainline of the source-code repository, is in the code base, and implements features that get tested by the automatic tests, but remains inaccessible to the user in the application released to the public.

Latent-code patterns are all these patterns that allow latent code in the code base. Doing TDD along with ATDD though an API layer while keeping the code inaccessible from the GUI is the simplest way to have latent code. The most well-known are feature toggles, branch by abstraction, and dark launching.

Feature toggles

Feature toggle is a well-known latent-code pattern. It is also known as "feature flab" or "feature bit".

Martin Fowler explains that features toggles basically have a configuration file that defines a bunch of toggles for various features that are pending. The running application then consults these toggles in order to decide whether or not to show the feature.

Branch by abstraction

Branch by abstraction is another well-known latent-code pattern.

Paul Hammant from ThoughtWorks coined the term to describe the technique of defining an abstraction in the code so that two implementations can be provided for the abstraction. One implementation is the original code that will be used in production and the other is the feature or change request under development that is not yet finished or not ready to be released, used only in tests.

Dark launching

Dark launching is a well-known latent-code pattern that goes one step further.

With dark launching, a new feature is released to a subset of users and is not shown in the user interface. Instead, special code automatically triggers the hidden feature with the aim of testing the infrastructure involved in serving that feature.

See also: "Canary Releasing" in *Continuous Delivery,* Jez Humble and David Farley: Addison-Wesley (2010)

Automatic deployment

After a CI build, the application has to be installed in test environments in order to run the tests. The application should be installed into the production environment in exactly the

same way to avoid variations that could produce non-detectable bugs in the test environments.

Automatic deployment avoids these problems and is faster, repeatable, testable, and traceable.

Automatic deployment here refers to the ability to automatically deploy, into testing and production environments, a new version of an application simply by running the same automatic script. Automatic deployment takes care of all the operations required (including changes to the application and infrastructure configurations, updates to the database, etc.) without requiring any further manual activities. The deploy scripts are stored in and retrieved from a version-control system.

Hot deployment with zero downtime

One important goal of CD is the ability to deploy during normal business hours instead of being limited to deployments during off-peak hours. A further improvement consists of limiting the interruptions and disruptions for users when deploying to production. This allows the business to decide when and how often to release a new version of the application without affecting the users.

The ideal situation is to do hot deployments with zero downtime. It means to deploy to production without the need to stop and restart the system or the application and without interrupting the work of the users. This definition applies, with small differences, to mobile applications, desktop applications, client-server applications, Web applications, and Web services.

This capability depends on the design and architecture of the system, on the automatic-deployment procedure and on the data-migration strategy.

See also: Blue-green deployment a.k.a. red-black deployment.

Remediation plans and database remediation

Can you imagine driving a car without brakes? You would drive slowly or you would crash. Remediation plans, like the brakes in a car, let a dev team run faster and safer at the same time.

In general, CD practices provide the ability to recover quickly from mistakes and defects with little or no consequence.

When a new version of an application is released into production but starts to malfunction or users report a showstopper bug, different techniques can help to remedy the problem. The following is a list of the common techniques.

Load-balancing clusters and crash-only software

Servers, services, and Web services are expected to be available without unplanned downtime. When a bug makes the system crash, these two techniques can mitigate the problem until a fix is developed and deployed. Using a load-balancing cluster of servers and making the service idempotent lets another node of the cluster take over when the server crashes. Developing the service as crash-only software lets it simply restart and continue to work from the previously known good state in cases of failure.

Failover clusters and database versioning

Changes to the database data or schema can cause showstopper bugs. Restoring the previous database backup is often time-consuming. Using a failover allows quickly switching to the other node that has not been updated. Another option is database versioning, which makes use of versioned delta scripts. Each has a corresponding script that reverts every change (see the dbdeploy open-source project).

Rollbacks and forward-compatible interim versions

A fast and effective way to react to a showstopper bug is to rollback to the previous, stable version of the application.

Sometimes the update that causes a showstopper bug contains changes to related moving parts, e.g. an update of the database schema and a change to a related feature, or a change to a service and a change to its public API, which affects the clients. The rollback of both moving parts at the same time may not be a viable solution, e.g. when the database size or the number of different clients is too large.

A good strategy to deal with this situation is to decouple the changes of the related parts and deploy to production one part at a time. In this way, rollback is a viable option again. A way to achieve this is to create and deploy a forward-compatible interim version of one of the moving parts, i.e. an application that is identical to the previous version except that is compatible with both the current database schema and the new one, or a service that is identical to the previous one except that it supports both the current public API and the new one.

Event-sourcing and two-ways live-data-migration techniques

The downtime caused by a remediation plan in some scenarios can cause loss of data that businesses may want to avoid, e.g. when the costs of data loss are much higher than the costs to avoid it. Event-sourcing techniques can be used to record and re-apply to the database all the data lost. Another solution for the same problem is the two-ways live-data-migration technique, meaning the ability to incrementally update or revert an update of a NoSQL database while applications are using the database.

Large code base and slow builds

A code base is all the source code associated with an application or a collection of applications. It is self-contained in the sense that it can be built without the need to reference resources outside the code base itself.

When the code base grows, the time required for the automatic build grows as well.

When the automatic build spends most of the time running automated tests, the cause can be the proportion of different types of test (e.g. unit tests, integration tests, acceptance tests). The tests pyramid defines a good heuristic for the proportion of the different types of tests: you should have many more low-level unit tests than high-level end-to-end tests through the GUI. It is worthwhile checking the tests against the guidelines from the tests pyramid.

Suggested reading:

- *Succeeding with Agile,* Mike Cohn: Addison-Wesley Professional (2009)

When you have hyperactive builds because shared components and services trigger a chain reaction, other approaches should be used.

Google's approach is to provide every developer with the tools to verify the code and get valuable feedback before committing the changes and triggering a new build. In this way, errors can be identified before triggering builds. Google uses a cluster of machines to execute the builds. It is a brute-force approach that, among other things, enables the collection and analysis of empirical data about the production code (e.g. code-base growth trends), the test code, and the test results (e.g. the system suggests deleting tests that teams spend much time maintaining and often flag errors for reasons other than real bugs).

Amazon's approach is to turn existing components and services into autonomous units that are split into distinct code bases, each one treated as a separate product. Compile time is reduced because a component or service does not need to be rebuilt by every application that uses it and only a new, official release of a component or service triggers the build of the applications that use it.

Components and services enable software reuse, encapsulate complexity, and group similar responsibilities.

Suggested readings:

- Packaging cohesion and coupling principles, Robert C. Martin, 2002
- "A Conversation with Werner Vogels". *ACM Queue*, 4(4), (2006, May)
- Google's Scaled Trunk Based Development, Paul Hammant (2013)
- Microservices, James Lewis and Martin Fowler (2014)

Infrastructure automation, infrastructure as code

A deployment pipeline uses different environments to build the code and run a variety of tests. Some of these environments are similar to the production environment and include the database and other services and configurations.

Infrastructure automation is intended to automatically create from scratch every test and production environment, from the operating system to the installation and configuration of the required software and services (networking, DNS, Web server, email server, etc.), and to automatically apply all the configurations expected for the environment.

Infrastructure as code means to treat the configuration of systems the same way that software source code is treated.

The goal of infrastructure automation and infrastructure as code is to have environments that are always in a well-known state and to avoid unnecessary variations that can invalidate the result of the tests or be a source of bugs in production that are very hard to reproduce.

Suggested readings:

- Snowflake Server, Martin Fowler (2012)
- Immutable Server, Kief Morris (2013)

Monitoring and logging

A proper monitoring strategy allows detection of problems in the production system as early as possible and gives IT ops more time to react, identify the source of the problem, and fix it.

Monitoring also provides valuable data for QA about the functioning of the application in production. Monitoring should also be applied in the test environment for the same reason and this is also an effective way to test the monitoring strategy.

When monitoring identifies a problem, good applicative logging is capable of providing effective information that helps IT ops to diagnose the problem quickly and identify possible solutions.

CD and DevOps tools, by Kief Morris

When considering how to bring the benefits of CD to an organisation, talk often turns to tools. However, tooling must be a secondary concern to the culture, people, and processes that underpin effective software delivery. In practice, all of these things tend to change and evolve over time as an organisation learns and continuously improves.

So if there is any rule to selecting tools to support software delivery and support, it is to assume that:

any tools chosen may need to be changed in the future.

Tools need to evolve and adapt to the changes and evolution of the product and the code base. It's a dance, and no one can stand still. Whatever tools your team decides to use, the most important thing is to continuously review how well they are working for you, and to be able to make improvements and changes as needed.

Following good practice around software and system design helps by ensuring that individual tools are loosely coupled and surrounded by automated testing, monitoring, and validation to allow refactoring with confidence.

The elements of a typical deployment pipeline include a tool to orchestrate building software components and applications, triggering deployments to environments, launching automated tests, and managing the promotion of artifacts through a progressive series of environments and gates. Ideally, the orchestration tool should not be tightly coupled to these concerns but should simply be used to manage when these things happen.

All of the software used for building, deploying, and testing software should be available for all members across the delivery organisation to use. Developers should be able to deploy software to a production-like operating system, perhaps locally using Vagrant, so they can replicate and troubleshoot issues. Developers should also be able to read and run automated tests themselves so they can check their changes and validate whether they have fixed defects reported by QA.

There is often a desire to choose a single tool for software deployment across the organisation. However, it's usually better to use tools and packaging formats designed for the specific application type and system platform, such as RPMs for Red Hat systems. Similarly for application build scripting, unit testing and similar tooling that relates to a specific programming language or platform are best used in preference to universal tools. Quite often, a tool that tries to work for a variety of different systems offers consistency at the cost of effectiveness.

"Infrastructure as code" describes an approach to infrastructure automation in which configuration is treated like software. It is normally managed in text-based files, similar to source code, which is committed to a software-configuration management (SCM) tool. This enables the use of software development practices such as test-driven development (TDD), automated testing, and CI, so that infrastructure changes can themselves be managed following CD practices. The following list contains examples of well-known tools commonly used in CD:

- **Version control:** Subversion, Mercurial, Git, Team Foundation Version Control
- **Binaries repository:** JFrog's Artifactory, Sonatype's Nexus
- **Dependencies management**: Bundler, NuGet
- **Database management:** dbdeploy, dbdeploy.NET 2, Liquibase
- **Infrastructure management:** Puppet, Chef, Vagrant, CFEngine
- **Build automation:** Ant, NAnt, Rake, Maven, Gradle, MSBuild
- **CI server:** TeamCity, Jenkins, CruiseControl.NET, Team Foundation Build
- **CD server:** Go
- **Test tools:** JUnit, NUnit, MbUnit, Selenium, Watir, Cucumber, SpecFlow, JBehave, Fitnesse, Microsoft unit test framework
- **Package management:** RPM, Debian, Windows Installer, WiX, Wise, InstallShield

The CD journey, changing together

A successful journey to CD more often than not is a sequence of small, gradual improvements. It involve changes in the technical coding and engineering practices and skills, organizational changes in the collaboration and communication patterns for all people involved from the concept to cash, and changes in the tools. All three of these elements are fundamental.

Every organization is different, on a different point of its own path to CD and faces a different bottleneck that needs to be tackled.

Suggested reading:

- *The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win,* Gene Kim, Kevin Behr, and George Spafford (2013)

Anecdotes and stories about CD

The number of small and large organizations deploying software into production weekly or daily is constantly increasing. This is the trend of recent years in the IT industry.

LinkedIn

Between 2011 and 2013, LinkedIn switched from a waterfall-like development process to CD, taking a system that requires a month to release new features and turning it into one that pushes updates into production multiple times per day.

Kevin Scott, longtime Google veteran, joined LinkedIn in 2011 and decided to import Google's practice of CD at LinkedIn. Read the article "The Software Revolution Behind LinkedIn's Gushing Profits" from *Wired.*

Facebook

Facebook has deployed software into production multiple times per week since 2005. Nowadays, with more than 1,000 developers, they release into production many times per day, affecting more than 845 million people.

Chuck Rossi, release engineer at Facebook since 2008, advises, "Ship early and ship often."

Ferrari F1 racing team

During a 2006 agile conference in Italy, Ferrari's F1 team reported their ability to rollback software deployments to the previous stable versions whenever a showstopper bug was found during a race weekend. The F1 team also presented their approach to TBD and CD. Those results came out of an effort to establish feedback loops that detect defects early, when they can be fixed quickly and without consequences. A fortuity contributed to their discovery of TBD: a bug in the version-control system forced software engineers to work on the trunk without branches while continuing to release new features for mission-critical applications frequently and on demand.

HP LaserJet Firmware

The LaserJet FutureSmart Firmware division at HP has more than 400 software engineers working on a large code base of 10 million lines of code. In 2008, the software division was the bottleneck for the business and the company's printer products were lagging behind the competition. They were following a traditional approach for software delivery, using multiple branches and taking more than six weeks to integrate and complete testing cycle.

Between 2008 and 2011, the firmware division made a successful transition to a modern approach in the direction of CI and CD, reaching the capacity to run 15 builds per day and run all automated regressions testing in only 24 hours, reducing costs by 70% and creating extra capacity for innovation. This case study is documented in the book *A Practical Approach to Large-Scale Agile Development: How HP Transformed LaserJet FutureSmart Firmware*.

Flickr

Flickr in 2004 was a Web startup that one year later was acquired by Yahoo, and in 2013 reached 87 million registered members. Flickr was one of the first well-known examples of companies using CD. In 2008, the famous code.flickr.com page went live to report on releases and the authors of the code changes. Starting in 2008, people from Flickr began to share their experiences in CD at international conferences.

Your organization here

What is the value of CD to the needs and opportunities of your organization? How is the current implementation of iterative development, CI, test and infrastructure automation, and CD practices? What are the next most-beneficial improvements? How do you imagine the CD journey in your organization?

About the authors

Luca Minudel

Luca Minudel has been working in professional software delivery since 1989. He delivers training to and coaching in top-tier organisations in Europe and the United States for ThoughtWorks. He has spent four years in Stockholm as hands-on agile coach. During 2006-2009, he contributed to the adoption of agile practices by Ferrari's F1 racing team.

Kief Morris

Kief Morris is ThoughtWorks' Europe practice lead for continuous delivery. He has been leading teams that deliver and manage software as a service for 15 years, and now enjoys helping ThoughtWorks clients to harness disruptive ideas in software delivery and operations, such as DevOps, lean, and cloud platforms.