O'REILLY®

Compliments of
Booz | Allen | Hamilton

# Enterprise DevOps Playbook

## A Guide to Delivering at Velocity

**Bill Ott, Jimmy Pham & Haluk Saker**
Foreword by Gene Kim

# Enterprise DevOps Playbook

## *A Guide to Delivering at Velocity*

*Bill Ott, Jimmy Pham, and Haluk Saker*

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

**Enterprise DevOps Playbook**
by Bill Ott, Jimmy Pham, and Haluk Saker



Printed in the United States of America.

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*.

**Editors:** Brian Anderson and Virginia Wilson
**Production Editor:** Colleen Lobner
**Copyeditor:** Octal Publishing Inc.

**Interior Designer:** David Futato
**Cover Designer:** Randy Comer
**Illustrator:** Rebecca Demarest

November 2016: First Edition

**Revision History for the First Edition**
2016-10-28: First Release



978-1-491-97415-5

[LSI]

# Table of Contents

# Foreword

DevOps principles and practices are increasingly influencing how we plan, organize, and execute our technology programs. One of my areas of passion is learning about how large, complex organizations are embarking on DevOps transformations.

Part of that journey has been hosting the DevOps Enterprise Summit, where leaders of these transformations share their experiences. I've asked leaders to tell us about their organization and the industry in which they compete, their role and where they fit in the organization, the business problem they set out to solve, where they chose to start and why, what they did, what their outcomes were, what they learned, and what challenges remain.

Over the past three years, these experience reports have given us ever-greater confidence that there are common adoption patterns and ways to answer important questions such as: Where do I start? Who do I need to involve? What architectures, technical practices, and cultural norms do we need to integrate into our daily work to get the DevOps outcomes we want? The team at Booz Allen Hamilton has published their model of guiding teams through DevOps programs, and it is clearly based on hard-won experience with their clients. I think it will be of interest to anyone to embarking on a DevOps transformation.

— *Gene Kim, coauthor of* The DevOps Handbook *and* The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win

# Enterprise DevOps Playbook

## Introduction

If Agile software development (SD) had never been invented, we'd probably have little reason to talk about DevOps. However, there is an intriguing corollary worth pondering, as well: the rise of DevOps has made Agile SD viable.

Agile is a development methodology based on principles that embrace collaboration and constant feedback as the pillars of its iterative process, allowing features to be developed faster and in alignment with what businesses and users need. However, operations today are generally moving at a pace that's still geared toward sequential waterfall processes. As Agile SD took off, new pressures and challenges began building to address delivering new code into test, quality assurance, and production environments as quickly as possible without losing visibility and quality.

We define *DevOps* simply as the culture, principles, and processes that automate and streamline the end-to-end flow from code development to delivering the features/changes to users in production. Without DevOps, Agile SD is a powerful tool but with a prominent limitation—it fails to address software delivery. As with other software development processes, Agile stops when production deployment begins, opening a wide gap between users, developers, and the operations team because the features developed for a timeboxed sprint won't be deployed to production until the scheduled release goes out, often times many months later. DevOps enhances Agile SD by filling this critical gap, bridging operations and development as a unified team and process.

Agile SD is based in part on short sprints—perhaps a week in duration—during which a section of an application or a program feature is developed. Questions arise: "How do you deliver each new version of this software quickly, reliably, securely, and seamlessly to your entire user base? How do you meet the operational requirements to iterate frequent software development and upgrades without constant disruption and overhead? How do you ensure that continuous improvement in software development translates into continuous improvement throughout the organization? How do you ensure that there is continuous delivery of programs during a sprint as they are developed?" It is from such questions that DevOps has emerged—a natural evolution of the Agile mindset applied to the needs of operations. The goals of a DevOps implementation are to fully realize the benefits that Agile SD aims to provide in reducing risk, increasing velocity, and improving quality.

By integrating software developers, quality control, security engineers, and IT operations, DevOps provides a platform for new software or for fixes to be deployed into production as quickly as it is coded and tested. That's the idea, anyway—*but it is a lot easier said than done.* Although DevOps addresses a fundamental need, it is not a simple solution to master. To excel at DevOps, enterprises must do the following:

- Transform their cultures
- Change the way software is designed and built following a highly modular mindset
- Automate legacy processes
- Design contracts to enable the integration of operations and development
- Collaborate, and then collaborate more
- Honestly assess performance
- Continually reinvent software delivery strategies based on lessons learned and project requirements.

To achieve the type of change described is a daunting task, especially with large enterprises that have processes and legacy technologies that are ingrained as part of their business. There are numerous patterns, techniques, and strategies for DevOps offered by well-known technology companies. However, these approaches tend to be too

general and insufficient by themselves to address the many issues that arise in each DevOps implementation, which vary depending on the organization's size, user base, resources, priorities, technology capabilities, development goals, and so on. Given these shortcomings, evident to us from our extended experience with DevOps implementations, Booz Allen has devised an approach for adopting DevOps that is comprehensive yet flexible. Think of this DevOps playbook as your user guide for implementing a practical style of DevOps that stresses teamwork and mission focus to achieve a single unyielding goal: *deliver new value to users continually by delivering software into production rapidly and efficiently on an ongoing basis*.

As Adam Jacob, founder of Chef, described in his DevOps Kung Fu presentations (available on GitHub and YouTube), there can be different styles of DevOps that are unique to each organization but fundamentally there are basic foundations, forms, and common principles that make up the elements of DevOps. This book represents our perspective and style as we distill DevOps into seven key practice areas that can be adapted for different DevOps styles. The key takeaways are the shared principles, the common practice areas ("elements"), and the goal for each of the seven practice areas.

## How to Use This Playbook

This playbook is meant to serve as a guide for implementing DevOps in your organization—a practical roadmap that you can use to define your starting point, the steps, and the plan required to meet your DevOps goals. Based on Booz Allen's experience and patterns implementing numerous DevOps initiatives, this playbook is intended to share our style of DevOps that you can use as a reference implementation for a wide range of DevOps initiatives, no matter their size, scope, or complexity. We have organized this playbook into five plays, as shown in Figure 1-1.

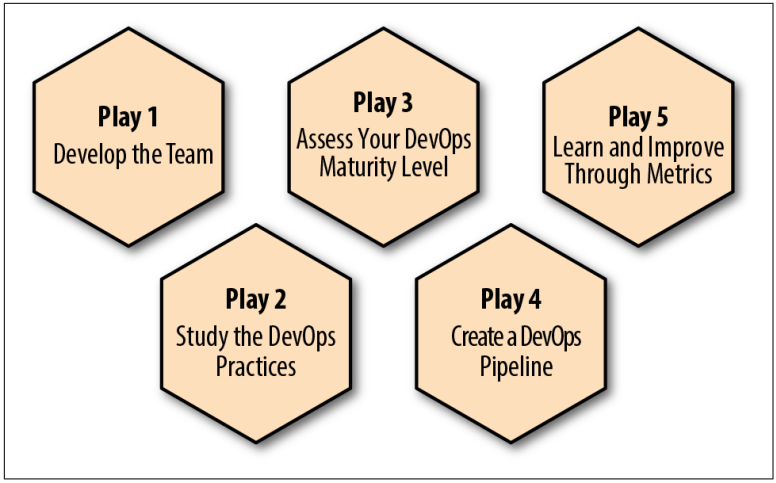


*Figure 1-1. The 5 Plays of the Enterprise DevOps Playbook*

Gene Kim, one of the top DevOps evangelists in the industry, described DevOps in 2014 as more of a philosophical movement than a set of practices. This is what makes it difficult for organizations to embrace DevOps and determine how to begin. This report is intended for teams and organizations of all maturity levels that have been exposed to the benefits and need for DevOps. We do not dive into the economics and the ROI aspects: the goal of this report is to provide organizations with a clear guide, through the five plays that cover all the practice areas that encapsulate DevOps, to assess where you are, to determine what they mean to you and your specific business requirements, and to get you started with an early adopter project.

*Play 1: Develop the team—culture, principles, and roles*

Successful transformational change in an organization depends on the capabilities of its people and its culture. With DevOps, a collaborative effort that requires cross-functional cooperation and deep team engagement is critical. This play details the key DevOps principles and tenets and describes how the organizational culture should be structured to achieve a top DevOps performance. You will be able to compare the structure of your organization to the principles in this play to drive the necessary culture change, especially for enterprises in which multiple functional groups (development, testing, and operations), vendors, and contractors might need to be restructured to enable

the transparency and automation across the groups. Having the people, culture, and principles in place is essential to an enduring DevOps practice; the people and the culture will drive success and continual improvement.

*Play 2: Study the DevOps practices*
: This play offers a deep dive into each of the seven DevOps practices—what they are and how they should be implemented and measured. The objective is for the DevOps project team to gain a baseline understanding of the expectations for each tactical step in the DevOps practice. We include a set of workshop questions to facilitate discussions among the DevOps team about the definition and scope of each practice as well as a checklist of key items that we believe are critical in implementing the practice's activities.

*Play 3: Assess your DevOps maturity level and define a roadmap*
: After there is a common understanding within the DevOps team about each practice, this play enables you to assess your organization's strengths and weaknesses pertaining to these practices. With that baseline knowledge, you can determine how to improve the practice areas where your organization needs improvement. As you go through this assessment and subsequent improvement efforts, you should refer back to Play 2 to review the definition of each practice area and to scan the checklist to ensure that the organization's skills are in increasing alignment with DevOps requirements.

*Play 4: Create a DevOps pipeline*
: The DevOps pipeline is the engine that puts your DevOps processes, practices, and philosophy into action. The pipeline is the end-to-end implementation of the DevOps workflow that establishes the repeatable process for code development—from code check-in to automated testing, to required manual reviews prior to deployment. In this play, we include a DevOps pipeline reference to illustrate DevOps workflow activities and tools.

*Play 5: Learn and improve through metrics and visibility*
: > You can't manage what you can't measure.
>
> —Peter Drucker

  The objective of this play is to define the metrics that you will use to measure the health of your DevOps efforts. Defining

metrics is critical to learn how your DevOps efforts can be improved, modified, or extended. The metrics in this play provide a holistic viewpoint—they help you know where you are, where you're going, and how to get there.

# Play 1: Develop the Team—Culture, Principles, and Roles

All DevOps success stories—those for which teams are able to handle multiple workstreams while also supporting continuous deployment of changes to production—have one thing in common: the attitudes and culture of the organization are rooted in a series of established DevOps principles and tenets. In this report, we do not explore the specific implementation strategies, because the solutions to achieve these are very unique to your organization; thus, our typical DevOps adoption engagements begin with an assessment process during which we do a dive deep to understand the organizational construct, existing processes, gaps, and challenges. We then overlay a DevOps model to see how it would look and determine the steps needed to develop the team.

In the next section, we introduce a list of key DevOps principles and cultural concepts. Each organization might have a different approach for adopting these tenets, but no matter what techniques you use, integrating the ideas themselves is critical to the success of the DevOps project. In "New DevOps Roles" on page 9, we describe the new roles and responsibilities required for a successful DevOps project.

## Principles and Culture

Many organizations have ambitious goals for adopting DevOps strategies, and an underlying need to do so. Their ability to effectively serve their customers and clients depends on nimble development and implementation practices. When such endeavors fail, it is often because the enterprise's culture is not suited for a DevOps program. Following are principles and cultural landmarks that organizations with successful DevOps implementations exhibit.

### Treat operations as first-class citizens

When operation engineers are supporting a system in production, developers are working on future releases with their scope and timing often determined by new feature requests from product owners and stakeholders. However, operational needs raised by the team during production can occur at any time, and the entire team, including developers, testers, and operations engineers, must treat such requests as priorities with equal weighting as other backlog items.

### Developers act as first responders to issues with the production system

For traditional SD, operations teams have their own engineers to address problems with applications as they occur. Even though these employees have the same basic skill sets as development team members, they were not the developers that wrote the code; thus, they are not familiar with how the code was built and probably unaware of the code's idiosyncrasies. For this reason, developers and operations engineers should work as a team to troubleshoot and repair issues that arise. The actual code developer should be part of the diagnosis team, communicating with the help desk and accessing the logs to find a solution without waiting for the operations team to provide background data. This enforces more accountability on the development team, extending its involvement to even after the software is delivered into production—an essential DevOps principle. With this added responsibility in the latter phases of the project, the development team's performance during the writing of the software improves because the group is no longer isolated from the mistakes discovered during production.

### Shorten the time between identification of a production issue and its repair

Often times, production support is reactionary. It is critical to change that mindset if the organization is to become more proactive in identifying potential needs. This could only be achieved if the developer is part of the operations support team. From a production issue perspective, there are many obstacles to communicating or integrating with the development team. Most times, the barrier is the structure of the development and sustainment contracts. For example, separate teams with separate contracts might have a delineation between their responsibilities, different Service-Level Agreements (SLAs), and varied processes. Even when separate contracts

are not a problem—that is, when there is only a single contract that covers development, operations, and sustainment—responsibilities and accountability might slow efforts to fix the production problem due to the separation of accountability and manual processes; for example, we often hear statements like, "This is a hosting problem, not an application problem; the sysadmins need to look into it," or, "We checked everything, it's over to the quality assurance (QA) team to validate and then the operations team to deploy."

### Shorten the time between code commit and code deploy

There are typically multiple gates that must be passed through before deploying software changes into production to ensure the integrity of the system. Most of these steps can be automated to reduce deployment time as well as to improve quality. However, to enable automation, the architecture of the system must be designed in a manner that is conducive to deploying discrete changes versus deploying the entire application every time.

### Minimize coordination to deploy releases

In traditional SD, there is a lot of inefficient coordination and communication overhead among teams. This is often necessary because each team has specific and siloed responsibilities and accountability for the project. If constant communication is not the norm, checks and balances will not occur, and quality suffers. Ironically, in a DevOps implementation, it is essential to minimize the formal communications channels between each group; instead, it is important to establish a culture of shared responsibilities, ongoing interaction, and shared knowledge of every aspect of the project and the automated repeatable processes among all teams. Maximizing transparency enhances efficiency and quality and reduces the need for constant forced communications.

### Stop and fix potential defects identified by continuous flow and monitoring

Frequently, defects and bugs in programs reported by users catch organizations by surprise, forcing them into a reactive mode to fix the issue. To avoid this inefficient and potentially harmful situation —which can impact organizational productivity and waste operational resources—you should implement a DevOps culture that focuses on constant visibility into all programs and systems to mon-

itor, identify, and resolve issues before they become so problematic that users are inconvenienced and critical workflow suffers.

#### Enforce standardized processes to ensure predictable outcomes

Exceptions should not be the norm. Informal and diverse processes increase instability and the potential for unpredictable outcomes. Establishing and enforcing repeatable standardized processes is essential. Moreover, it is the cornerstone of automation, which itself is central to an efficient and unified DevOps process.

#### Become a learning organization through continual feedback and action

Every environment is different and all aspects of development and operations are subject to change. The goal is to constantly learn, assess, and analyze what works and what doesn't. Organizational visibility and a culture built on constant improvement is critical to DevOps success. Peter Senge, the founding chair of the Society for Organization Learning, explains in his book, *The Fifth Discipline* (Doubleday), the importance of having an environment in which "we must create a culture that rewards learning, which often comes from failure. Moreover, we must ensure that what we learn becomes embedded into our institutional memory so that future occurrences are prevented."

The adoption of these principles and culture changes requires organizations to define new roles for development and operations teams. We describe several new roles and their associated responsibilities in a DevOps world in the next section.

## New DevOps Roles

Currently, there is a lack of clarity of the new roles and responsibilities required for DevOps. In this section, we identify the new DevOps roles for you to consider in augmenting your current team with the enhanced mix of talent to implement DevOps practices.

In an ideal situation, these responsibilities would exist in an *autonomous team* that manages its own work and working practices and builds, deploys, operates, and maintains software functionality without any additional support from other parts of the organization.

### The DevOps Team Antipattern

We do *not* believe in the existence of DevOps standalone teams—DevOps responsibilities must exist within the Agile teams. Just like a software developer or tester, a DevOps engineer or an engineer with DevOps skills must take ownership of the feature on which the team is working and work to make the software better for the end user.

#### The DevOps architect

The DevOps architect uses automation to create efficient, effective processes and standards to continuously improve quality and estimation. The DevOps architect must have deep knowledge, hands-on experience, and a passion for making a difference in the DevOps space. The architect implements Agile practices and automation and has the technical depth necessary to provide advice for making appropriate technology choices, defending recommendations, and driving technical implementation. These technologies and methods include understanding containerization and the orchestration of containers across multiple Infrastructure as a Service (IaaS) vendors and deployment automation using IaaS providers. Roadmaps to support automation are an essential element of the architect's role, which includes end-to-end test automation and tool strategies. An architect oversees tool analysis and selection and implements test automation frameworks and approaches.

#### The DevOps engineer

A DevOps engineer performs in a hybrid technical role that comprises development and operations, and must be proficient in coding or scripting, process reengineering, and collaborating with multiple groups. The DevOps engineer also must be well-versed in multiple popular and commonly used operating systems and platforms. The DevOps engineer is responsible for implementing the automation vision of the DevOps architect's development and deployment pipeline. This includes the critical responsibilities for developing Infrastructure as Code that enables immutable infrastructure, repeatability, and automation of the entire infrastructure for each environment.

### The test automation engineer

The test automation engineer automates as many steps as possible to achieve the test coverage and confidence required to quickly push changes to production.. The test automation engineer should be well-versed in software testing processes and tools (such as unit testing, mock integration testing, and test automation tools like Selenium) and be proficient in the scripting needed to implement automated test processes for each DevOps pipeline step. These automations include unit tests, static code tests, smoke tests, and more specific tests, such as Section 508 compliance checks, SLA validations, and vulnerability scans. Test automation should not be solely the responsibility of the test automation engineer; it should also be the province of every developer and tester on the team. A developer must be able to add any type of automated test that integrates with the delivery pipeline defined by the DevOps engineer. Typically, QA is performed by a separate group. The test automation engineer provides the framework and expertise to enable developers to embed test automation as part of the continuous process.

### The site reliability engineer

The site reliability engineer is responsible for monitoring the applications and/or services post-deployment. Site reliability engineering occurs when you ask a software developer to design an operations team. Using a variety of metrics (e.g., application performance monitoring [APM], SIEM, user metrics, and infrastructure health) aligned with the deployment strategies being used (e.g., canary releases, blue/green deployment), the site reliability engineer should be exceptional at troubleshooting and at metrics analysis to establish key thresholds for alert automation to baseline the different health levels for applications and services. Simply put, the primary responsibility of site reliability engineers is to automate themselves to be as efficient as possible.

### The software engineer

A frequently discussed topic with DevOps is the responsibility of software engineers (also called developers). To be successful in a DevOps world, a developer must embrace all the principles discussed previously and have a complete team mindset, where traditional responsibilities such as testing, infrastructure, and system

administration are all part of the developer's roles. This aligns with the mantra of "you build it, you run it, you fix it."

The roles and responsibilities we covered are meant to provide insights and to generate healthy discussions among your organization with regard to how to assess resources within your team, and to identify any potential gaps. The important takeaways are the responsibilities that you need to cover versus the actual role titles because other organizations might call the roles different names such as "release engineer" or "ops lead."

## Play 2: Study the DevOps Practices

As DevOps became an industry buzzword, the term lost some of its meaning. It's obvious that DevOps is a collaboration between the development, testing, and operations teams, but where it fits within an organization and how teams adopt DevOps vary from enterprise to enterprise. For example, some say DevOps is primarily a by-product of culture; others say it is primarily a technological application. No matter through what lens you view DevOps, it is essential that the teams have a clear and common understanding of the following seven core DevOps practices:

- Practice 1: Configuration management
- Practice 2: Continuous integration
- Practice 3: Automated testing
- Practice 4: Infrastructure as Code
- Practice 5: Continuous delivery
- Practice 6: Continuous deployment
- Practice 7: Continuous monitoring.

These practices encompass the full end-to-end cycle, from code commit to deployment to operations and maintenance.

### How to Read the Practices

In the subsections that follow, we dive into each practice area by providing the following components:

*Definition*
: A common introductory reference point for how the practice can best be described and what it entails.

*Workshop guiding questions*
: The basis for a dialogue and discussion with your team about the parameters and scope of your project. The intent of the guiding questions is for a thought exercise to help gauge where things are, what needs to be considered, and what's actually important for your team.

*Checklist*
: A number of items that may become tasks in your project backlog to implement or serve to improve your DevOps maturity.

## Practice 1: Configuration Management

#### Configuration management definition

Configuration management (CM) is a broad term, widely used as a synonym for version or source code control, but can also be used in the context of managing, deployment artifacts, or configurations of software/hardware. In our context here, we are talking about CM strictly from code management perspective. In the DevOps landscape, CM focuses on having a clear and clean code repository management strategy that helps maximize automation and standardize deployments.

Developers often are frustrated by not being able to replicate a problem that appears in different environments. You may hear developers say, "It works in my environment. I cannot reproduce the glitch." This is mainly because CM of the application throughout the process of development and deployment is controlled manually and there are definite differences between a program's behavior in a deployed environment and on a developer's computer. This results in configuration drift or "snowflake" environments, in which man-

agement of the various environments and code repositories become a heavy lift and a nightmare to manage. Although a software team might carefully document the evolving versions of the configuration of a server or a container, the need to manually execute the documented process might still result in human errors because the server will be used in the development, code testing, user acceptance testing, and production environments.

You can achieve effective and less problematic version and source code control by using various automated CM workflows. The following list of workflows is a compilation of common industry practices that ranges from basic to advanced. In this playbook we use GitHub for CM implementations, but other source code control systems, such as GitLab or Bit Bucket, are good alternatives, and you can follow similar workflow strategies for them as well.

### Centralized workflow

In a centralized workflow (Figure 1-2), a central repository serves as the single point of entry for all changes to the project. The development branch is called the "master" and all changes are made to this branch. A centralized workflow does not require any other branches. Developers have the complete copy of the central repository locally; they write their changes on their own equipment; and then synchronize with the master at agreed upon intervals. If there is a conflict among developers initiating changes, Git (for example) will stop the commit and force a manual merge. This workflow is usually used by teams to transition out of traditional source code control using Apache Subversion (SVN).

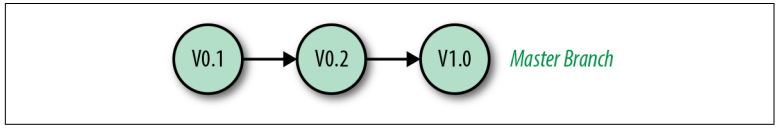


*Figure 1-2. Centralized flow*

### Feature workflow

Feature workflow (Figure 1-3) is for teams that are comfortable with centralized workflow. Isolated branches are added for each new feature, which enables independent and loosely coupled design principles without affecting other parts of the system. The main difference between feature workflow and centralized workflow is that in feature workflow, feature development takes place in a dedicated

branch instead of in the master. This enables various groups of developers to work on features without routinely disturbing the master codebase. Feature workflow also provides a foundation for continuous delivery and continuous deployment DevOps practices. For a microservices architecture, which treats each microservice as a separate product, feature workflow branches are an ideal configuration management mechanism because the code for each feature is isolated.

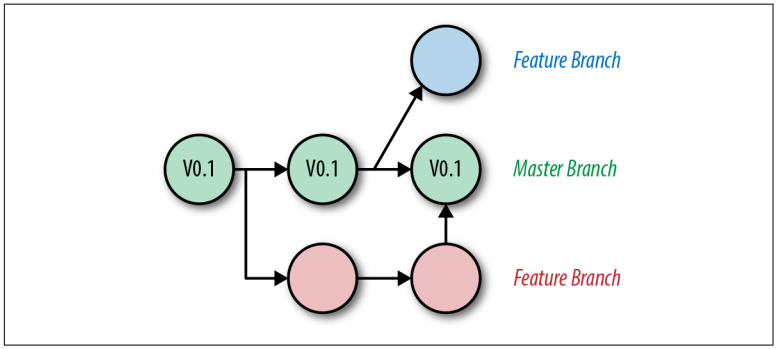


*Figure 1-3. Feature branch workflow*

### Gitflow workflow

Gitflow workflow is for established and advanced projects. It includes the attributes of centralized and feature workflows and also other capabilities. Based on Vincent Driessen's Git Branching model, Gitflow workflow is a good foundation for many of the seven DevOps practices.

Gitflow workflow (Figure 1-4) is composed of a strict branching model designed around project releases and enhancements. Specific roles are assigned to each of five branches: Master, Hotfix, Release, Development, and Feature. The way the branches interact with one another as well as the availability of multiple branches coming off an individual branch, is carefully defined. For example, while new development is occurring in one feature branch, another feature branch can be created for changed requirements or to fix a defect for an update that will be deployed with a patch release.

Table 1-1 explains each of the Gitflow branches.

*Table 1-1. Gitflow branches*

| Branch | Description |
|---|---|
| Master branch | Stores the official release history; the code is merged to it only when a new release is deployed. All historical releases are tagged in the master branch. |
| Hotfix branch | Supports maintenance or patch fixes that must be released quickly into production. This is the only branch that forks off of the master directly. After the fix is completed, the changes are merged back into the master, development, and/or release branches, as appropriate. |
| Release branch | Acts as the safeguard between the development branch and the public releases (master branch). |
| Development branch | Serves as the integration branch for features. Continuous integration deployments to development servers are performed from this branch. Developers code on a local copy of the development branch. When they are finished, the code is merged to the development branch. 100 percent code review Agile practice is used to check the code against the team's development practices. |
| Feature branches | Represent each feature just like the feature branch workflow. Instead of branching off from the master, feature branches use the development branch as the originator. When the feature is complete, it is merged back to the development branch for integration into the project. As the features are built, the developer local code and feature branch interaction could be automated using continuous integration. When that option is chosen, the feature branch is a separate module until the feature is complete. |

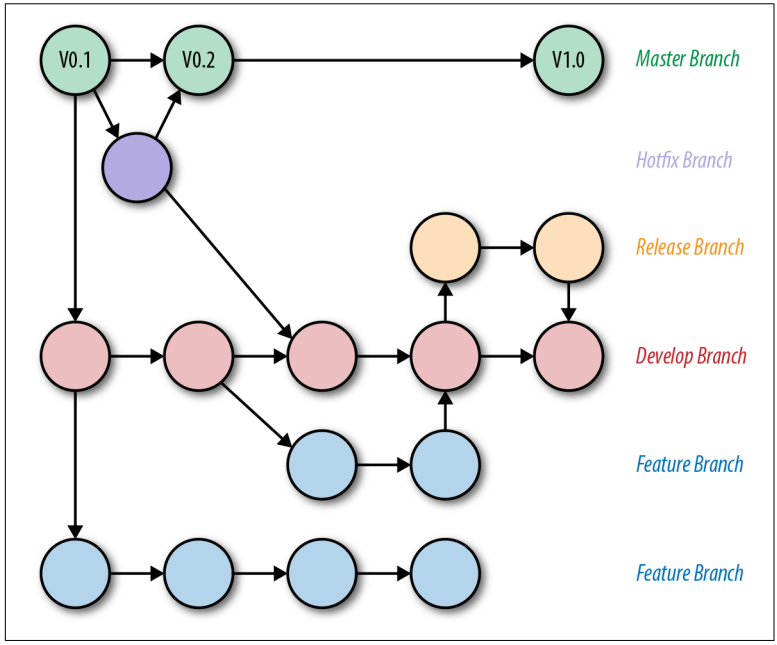


*Figure 1-4. Gitflow workflow*

### CM workshop guiding questions

- How do you manage source code versions?
- Do you have a production system that you are maintaining and improving or are you building a new system?
- How many features are being built at the same time?
- What happens when you receive a defect from the operations team? How do you fix the defect and manage the source code change?
- Who is the owner of the source code control system—development team, maintenance team, operations team?
- What are the criteria you use to decide on your source code branching strategy?
- Are you building a monolithic system or a microservices-based distributed system?
- How many code repositories do you have (or expect to have) for the project?
- Are you planning continuous delivery or continuous deployment?

### CM checklist

- Document your configuration management approach in a simple, easy-to-read document accessible to the entire team.
- Select a source code control system that allows all types of branching techniques even though you might not use advanced flow in the beginning.
- Select your configuration workflow based on operations need and continuous delivery objectives. The most effective flow for delivering under all scenarios is Gitflow.
- Document your branching strategy in a way that makes it easy for the entire team to understand.
- Define the limitations of your configuration management and source code branching strategy.

## Practice 2: Continuous Integration

### Continuous integration definition

Continuous integration (CI) requires developers to integrate code into a centralized repository as they finish coding and successfully pass unit testing, several times per day. The end goal is to create small workable chunks of code that are validated and integrated back into the code repository as frequently as possible. CI is the foundation for both continuous delivery and continuous deployment DevOps practices. There are two CI objectives:

- Minimize integration efforts throughout the development and deployment process
- Have the capability to deliver functions to the end-user community at any point in the development process (see "Practice 3: Automated Testing" on page 20)

CI is also a key Agile SD practice that connects Agile development with DevOps. CI is certainly not a new concept; it has been around for a while, and without a well-established CI practice, a development team cannot expect to achieve DevOps success.

To facilitate advanced DevOps practices, such as continuous delivery and continuous deployment, CI must be planned and implemented well. When a developer divides functionality into smaller chunks, it is critical to ensure that the software not only works and is integrated into the main code repository, but can be safely deployed to production if there is a business reason to do so before the rest of the application's features are complete. CI forces the developer to think differently during planning; that is, she must view functionality in small, discrete bites that can be integrated into the rest of the code base as finished pieces of larger programs.

Before CI became a best practice, integration was a major and often difficult ordeal. Developers would code in their local environments and, when ready, collaborate to integrate their code into the larger project. More often than not, the working code had to be rewritten for successful handshakes among the disparate sections of code to occur.

With DevOps, the goal is to achieve continuous releases. When program enhancements or features are complete, they should be

deployed or deployable; CI enables this unique aspect of DevOps. However, CI alone is not sufficient for the deployment of "done" code to production; full confidence from both comprehensive test coverage and immutable infrastructure is needed. In essence, CI is a prerequisite to achieving continuous deployment and works in conjunction with the other DevOps practices to build and ensure confidence throughout the process, which ultimately creates the level of trust required to automate deployments directly to production.

### CI workshop guiding questions

- How ready are you for CI? Do you have your CI tool and expertise in automating builds?
- How long does it take to build your code?
- What is your unit test coverage percentage? Are you confident that if a developer breaks another developer's code, the unit tests will detect it?
- How many Agile teams do you have on your project? If you have multiple teams, what is the definition of CI to you?

### CI checklist

- Select a CI tool that is capable of achieving your automation objectives.
- Connect the CI tool to your code repository. Test check the code.
- Integrate your build script to CI.
- Manually trigger the build from your CI server.
- Test the build scheduler.
- Define the build branch for each environment from your branch structure. The following steps are for Gitflow:

  1. Set up build for Dev environment from development branch
  2. Schedule Dev build for every check-in (core activity of CI begins with this build; ends if all automated tests pass)
  3. Set up build for test environment from the development branch

4. Set up build for preproduction (or QA) from the release ranch
5. Set up build for production from the release branch
6. Set up auto commit to the master branch after production deployment.

- Define automated tests for each commit to the development branch.
  - — Perform unit tests (frontend, backend, API)
  - — Perform vulnerability scan (use ZAP for the development branch)
  - — Perform code quality scan
- Define alerting solution for developer commits.
- Define criteria to accept or reject commits.
- Define commit reject mechanisms when committed code is not acceptable (e.g., if there are vulnerability findings, or if a unit test fails). *Do not* ignore any test fail. *Stop* and fix.
- Define automated tests for each commit to the release branch.
  - — All tests run for development branch commits
  - — Performance tests
  - — Other security scans
  - — Functional tests (e.g., Selenium tests and browser compatibility tests)

## Practice 3: Automated Testing

### Automated testing definition

Automation is the very heart of DevOps. The more tests you can automate, the more likely that the quality of the program and the efficiency of the deployment will be production-ready. DevOps teams use the results of automated testing to determine their next steps. The end goal here is to shift testing and compliance to the left of the delivery process as much as possible, in essence ensuring that tests are conducted early and often in the development process. During CI, automated testing can identify defects early on so they can be addressed. During continuous delivery, automated testing

can increase the team's confidence level that it can deploy a new capability into production.

In DevOps, automated testing should be used during CI (on development), Infrastructure as Code (on development), continuous delivery (on test and QA), continuous deployment (on QA, acceptance, and production), and continuous monitoring (on development, test, QA, and production).

Recommended automated DevOps tests, by environment, are shown in Figure 1-5. We use a modified version of the Chef delivery pipeline terminology for each of the steps.

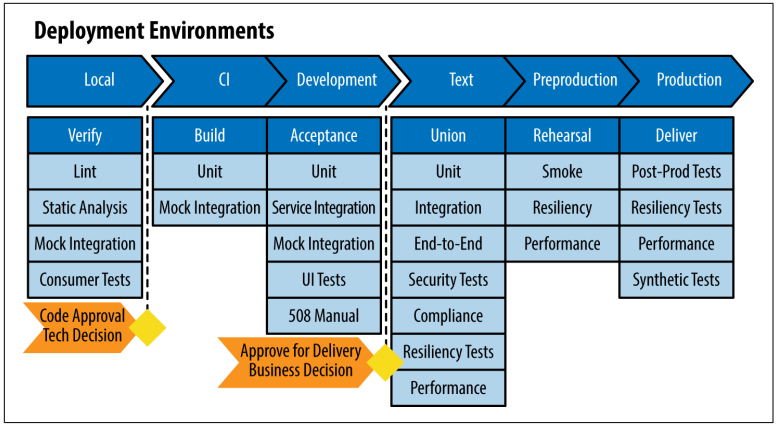


*Figure 1-5. Recommended automated tests*

Let's take a quick look into each of the environments and the automated testing activities for each one.

### Local environment

This is local developers' environment, whether it is their own machine or a VM. If any tests fail locally, the developer stops and fixes the issue.

*Lint and static analysis test*
: Identifies potential bad source code (e.g., nonconforming code, bad practices, compilation errors, security flaws) by examining the code itself without executing the program. Although static code analysis is useful, it generates both false positives and false negatives. Ideally, all of the outcomes of static code analysis should be reviewed. The frequency of static code analysis can be

determined by the team's DevOps maturity on other automated tests.

*Unit test*
: The first line of defense in assuring quality, a unit test should be developed for both frontend and backend code. One hundred percent unit test coverage is necessary to have the confidence in the source code that is required for continuous deployment. Unit tests should be:

- Small in scope
- Developed by the programmer based on acceptance criteria
- Able to run quickly so that developers use them routinely for every change

*Mock integration test*
: Validate the expected result of APIs, which are a facet of any modular architecture, such as microservices or component-based design. For simple parts of an application, API tests could be the same as unit tests. The intent is to simulate and provide test integration points through mock outputs.

*Consumer test*
: Validate integration points with external services that are being used to ensure correctness on both sides.

### CI environment

After local testing is passed and code review is completed and approved, CI environment testing begins with another quick round of checks with unit and mock integration testing.

*Unit test*
: The first line of defense in assuring quality, it should be developed for both frontend and backend code. One hundred percent test coverage is necessary to have the confidence in the source code that is required for continuous deployment. Unit tests should be the following:

- Small in scope
- Developed by the programmer

- Able to run quickly so that developers use them routinely for every change.

*Mock integration test*
: Integration testing is critical and is repeated in this environment. We want to ensure and validate the expected result of the APIs.

### Development environment

After successful CI environment testing, we proceed to the development environment, in which a series of tests are run to ensure readiness for the test environment. If any of the tests are failed, the developer stops and fixes the issue.

*Unit test*
: Rerun the unit tests for all the components.

*Mock integration (service and consumer) test*
: Tests all the service integration points.

*UI test*
: Runs over larger parts of the application and simulates user actions. All of the capabilities and features of a user interface (UI) must be tested by automated functional testing. Selenium is a common tool for this. The drawback of functional tests is that they are slow, which explains why they are not usually used in the development environment. The rule of thumb is this: no build should propagate to the QA environment before there is a functional test for all UI capabilities and before all of the functional tests run successfully.

*508 manual test*
: For applications that require Section 508 compliance, this is an important step. Section 508 testing is mostly manual, but tools such as CodeSniffer can check pages and detect conformance to the W3C's Web Content Accessibility Guidelines (WCAG) 2.0 and Section 508 guidelines.

### Test environment

Upon successful development testing and business approval (for situations in which a manual business decision is required), the test environment follows. This is the environment that unites all the dif-

ferent development streams and ensures comprehensive testing across all major areas.

*Unit test*
: Rerun the unit tests for all the components for all the development streams.

*Integration test*
: Validates the integration points (internal and external) and tests the expected inputs and outputs.

*End-to-end test*
: Run tests across functional and nonfunctional areas (e.g., infrastructure, build scripts, and configurations).

  *Functional test*
  : Runs over larger parts of the application and simulates user actions. All of the capabilities and features of a UI must be tested by automated functional testing.

  *Build script test*
  : Because DevOps automation relies on high-quality build scripts, build script testing is used to ensure that there are no errors during the build process and that the scripts are consistent across each environment.

  *Properties file/configuration parameter test*
  : All software development projects have configuration parameters that must be tested for target boundary conditions.

  *Infrastructure code test*
  : Similar to applications code, infrastructure code must be tested for every change. This is a critical for ensuring immutability and repeatability. We discuss this in detail in the next practice section ("Practice 4: Infrastructure as Code" on page 29).

*Security test*
: Ensures the security posture of your applications and environments by mitigating potential issues through automated detection. In reality, not all items will be revealed, but security and compliance tests provide a strong baseline and confidence to move forward:

*Vulnerability test (or penetration test)*
: New code must pass a vulnerability scanner to validate that no critical application-level vulnerabilities are present. About 95 percent of reported software breaches in 2015 were the result of 1 out of 10 well-known vulnerabilities; 8 of them have been in the crosshairs of the software development sector for more than 10 years. Zed Attack Proxy (ZAP) is a common vulnerability test tool.

*Functional security test*
: Verifies security features, such as authorization, authentication, field-level validation, and personally identifiable information (PII) compliance.

*Security scanning test*
: Targets applications, containers, networks, firewalls, and operating systems for vulnerabilities.

*Compliance test*
: Many systems are required to be compliant with specific sets of standards. The goal of compliance testing is to use tools like OpenSCAP and Chef Compliance to automate the process that will continuously validate the posture of your system as changes take place versus the tedious exercise of addressing these changes after a release candidate is established.

*Resiliency test*
: Applies a series of random simulations to assess the availability of your system in outage events. For example, using Netflix's Simian Army (Chaos Monkey and Chaos Gorilla), resiliency is tested by taking down random AWS instances to an entire Amazon Web Services (AWS) availability zone to test the fault-tolerant architecture of your system.

*Performance test*
: Measures the performance and scalability of an application. After the team defines SLAs for web page and API performance, these tests will alert the team if the applications/APIs are not performing as required. Some of these tests (e.g., soak tests) are used only for specific situations rather than running continuously for every change.

*Load test*
: Analyzes the impact of load on an application. For example, if a performance issue is identified and subsequently addressed, only a load test can determine if the problem has been resolved.

*Component load test*
: Applies a load test to a single component (or microservice).

*Soak test*
: Identifies memory leaks. Some severe memory leak problems do not surface during load testing. To uncover these issues, you must run soak tests over extended periods of time (at least overnight) with steady high loads.

*Capacity test*
: Determines how many simultaneous users a web application can handle before response time becomes an issue (or before the system performs below its SLA).

*Stress test*
: Pushes a web application beyond normal workload to identify which components or services will fail under extreme conditions and spikes in production.

#### Preproduction environment

This environment serves as a test ground to simulate delivery to production. In live implementations, the tests here vary from organization to organization, depending on how confident and comfortable you are up to this point.

*Smoke test*
: Run a comprehensive set of functional tests under low load to ensure major features of the applications work as expected.

*Resiliency test*
: Perform random simulations that are essential to test the availability of the system in outage events and continuously validate the fault-tolerant architecture of the system.

*Performance test*
: Assess the expected performance of the new deployment. In "Practice 7: Continuous Monitoring" on page 40, we discuss the

metrics and monitoring for expected performance levels and the health of the system.

#### Production environment

At this point, the tests and validation processes needed for production deployment are completed. In this section, we list tests that are recommended for the production environment. In "Practice 5: Continuous Delivery" on page 32, we explore effective production deployment strategies, such as canary releases and blue/green deployment.

*Post-production test*
: Run a series of tests on the application as well as infrastructure to ensure that the system is in a healthy state (should include subset of functional UI tests, configuration tests, and others) after deployment. The extent of the post-production tests depends on the size and maturity of the environment.

*Resiliency test*
: Conduct random simulations that are essential to test the availability of the system in outage events and continuously validate the fault-tolerant architecture of the system.

*Performance test*
: Assess the expected performance of the new deployment. In "Practice 7: Continuous Monitoring" on page 40, we discuss the metrics and monitoring for expected performance levels and the health of the system.

*Synthetic test*
: Set up these tests to simulate real-world usage patterns by mimicking the locations, devices, and patterns of users. Some application performance monitoring solutions, such as New Relic, offer synthetic testing programs as well. The intent is to proactively discover issues users are encountering and gain insight for troubleshooting.

Although this is a comprehensive list of automated tests, manual tests are also an important part of system delivery. Some functionality may require manual validation. Types of manual tests include exploratory testing, user acceptance testing, Section 508 accessibility testing, and usability testing.

### Automated testing workshop guiding questions

- Which tests are currently automated and which do you plan to automate?
- What percentage of your code base is automated?
- How long does it take to run the automated test harness? If you have different harnesses for unit tests and functional tests, how long do they take to run separately, and how often do you run them?
- What test automation tools and frameworks do you use?
- What is the process for a developer to add a new unit test?
- How many concurrent users are expected to use your system?
- Do you have browser page load requirements?
- How do you test load, performance, and stress?
- What do you do when performance degrades? What is your scaling strategy?
- Do you have system availability requirements?

### Automated testing checklist

- Set high unit-test coverage and add this coverage as a requirement for developers to complete before a project can be finalized.
- Select unit test technologies for UI, middle-tier, and backend.
- Implement unit-test reference implementations and add them to developer practices.
- Build mechanisms to measure the test coverage for unit tests.
- Integrate your unit tests into test harnesses. Build mechanisms for integrating the unit tests into the harness.
- Integrate the unit-test harness into the build script and CI.
- Build mechanisms to take action when unit tests fail.
- After unit-test coverage is finished, identify which integration tests should be implemented. Integration tests should cover end-to-end paths in user stories.
- Implement and automate integration tests.

- Select technology for acceptance tests or end-to-end functional tests. Selenium is a possible option.
- Order your functionality, from most critical to least critical, and build end-to-end automated functional tests starting with the most critical ones. (There will be fewer end-to-end tests than unit tests. The quality of the functional tests is more important than the quantity.)
- Define actions when a functional test fails. Build an alerting tool so that the developer is notified.
- After you achieve the maturity you are targeting with unit tests, integration tests, and end-to-end functional tests, prioritize other types of tests defined in this section and integrate them into the aforementioned environments. Begin with the security tests.

## Practice 4: Infrastructure as Code

### Infrastructure as Code definition

It can be argued that the recent maturity and adoption of tools and technology enabling Infrastructure as Code (IaC) are among the primary reasons DevOps has exploded onto the scene with true end-to-end automation. IaC goes beyond simply automating processes using scripts, as traditional systems administrators have been doing for years. Instead, IaC enables configuration and provisioning of infrastructure through applications using APIs. With IaC, configurations are treated just like application code with all the best software development practices of version control, unit tests, code review, gating, and so on. The barriers and walls between system administration and operations staff and developers are removed, with full transparency, automation, and shared responsibilities. IaC addresses configuration drifts by ensuring immutability of both the actual server configurations/instantiation and also standardizing how environments are built and deployed.

The ability to view IaC opens the possibility to control and automate configurations so that they are highly repeatable and consistent, no matter how many times the environment is torn down and brought back up. Not surprisingly, best practices for IaC are similar to those for software development, including the following:

*Source control*
: Code version control and auditing

*Peer reviews*
: Code review practices as application code processes

*Testing*
: Unit testing, functional testing, and integration testing

*Code as documentation*
: Code is developed in an easily readable manner as the IaC itself serves as living, always-up-to-date documentation

*Collaboration*
: The barrier between development and operations is broken down

There are two primary approaches for implementing IaC. It is important to assess the pros and cons of each and determine which model fits best with your team and environment:

*Declarative model*
: Similar to the concept of declarative programming, code is written to describe *what* the code should do as opposed to *how* the code should do it. Puppet, for example, is declarative because it doesn't run scripts or execute code on the infrastructure. You can use Puppet's declarative language to model and describe the desired target server end state. You tell Puppet what you want the server to look like, not the steps to take to get there. The typical advantage here is a cleaner and more structured approach, as you are just modeling and describing what is needed. The downside is less flexibility in what you can control.

*Imperative model*
: Code is written that explicitly lays out the specific steps that must be executed to complete the desired task or, in this case, the desired target server end state. Instead of modeling the end state, you use the imperative approach to describe how to get to the end state. In this model the advantage is full control of what steps and actions you execute and the conditions around them. However, the code/scripts can get complicated. For example, Chef is imperative—users define commands and their execution order and logic. These instructions, called Chef Recipes, in turn can be organized in Cookbooks. To illustrate this, part of a

recipe can be to check if a specific dependency is available on a target node. If the dependency exists, Chef will install it; otherwise, a warning is raised and/or other failed execution steps are run.

No matter which solution you decide on, having IaC in place will reduce and in most cases eliminate configuration drift/snowflake environments. In making your decision, you should consider the skillset of your team and how well it lines up with the various options, such as Chef, Ansible, or Puppet.

### IaC workshop guiding questions

- Does your datacenter or IaaS provider have scripted provisioning capability?
- Can you script the installation of every product in your system?
- What are the differences for installing components of your system when they are deployed on these distinct environments: developer computers, development, test, QA, and production?
- Do you perform version control on the configuration control parameters of your components? If yes, how?
- Who on your team knows the details of these configuration parameters?
- Have you heard this all-too-common complaint?: "I cannot reproduce this error in my environment."

### IaC checklist

- Set the objectives of your infrastructure automation and plan your automation in an Agile way. You do not need to wait to complete everything before you use it. Start using the IaC as pieces are completed.
- Define what you can automate and what you cannot. For example, most application servers can be automated, but network devices such as firewalls can be configured automatically if they have APIs to do that.
- Write the code to automate the configuration of the container in which your main application is going to run.

- Integrate the IaC for your app servers/containers to CI. Ideally, re-create the servers and containers when there is a patch instead of just applying the patch. After you do this, you will have achieved an immutable infrastructure for your application servers.
- Use the same IaC you build for your app servers on your developers' computers and in your development, testing, quality, and production environments. There should not be any difference other than the IP, URL, and so on.
- Write the code for building your database server and container and follow the same steps you used for the applications server.

## Practice 5: Continuous Delivery

### Continuous delivery definition

Continuous delivery simply means that every change is ready to be deployed to production as soon as automated testing validates it. Figure 1-6 illustrates a reference pipeline flow (also discussed in "Practice 3: Automated Testing" on page 20). Note there are two manual checkpoints in this example. One is for a technical decision to approve the code before initiating activities in the CI environment. The second is a business decision to accept the changes and continue with the automated steps to production deployment.

The goal of continuous delivery is to deliver incremental or small changes quickly and continually as they go through the pipeline. This not only helps distribute microfeatures more rapidly, but also reduces the risk of a large release failing. It is important to note that not all microfeatures that go to production will be seen by users. These features or fixes might be part of the production version, but potentially "turned off" for a period of time because you want to have specific changes validated as part of a build, even though the organization is not necessarily ready to install them on end-user systems.

DevOps is a reaction to features and releases being deployed slowly. Even using Agile SD principles, the mindset is to build programs incrementally and produce releasable software at the end of every sprint. In each sprint, the team develops features/changes for the software, but the completed story must wait for the other stories to

be finished and grouped together to be deployed in a scheduled release. In contrast, continuous delivery gives power to the product owner, who is the gatekeeper in DevOps terms, to decide whether to deploy the new capabilities before the rest of the release is ready.

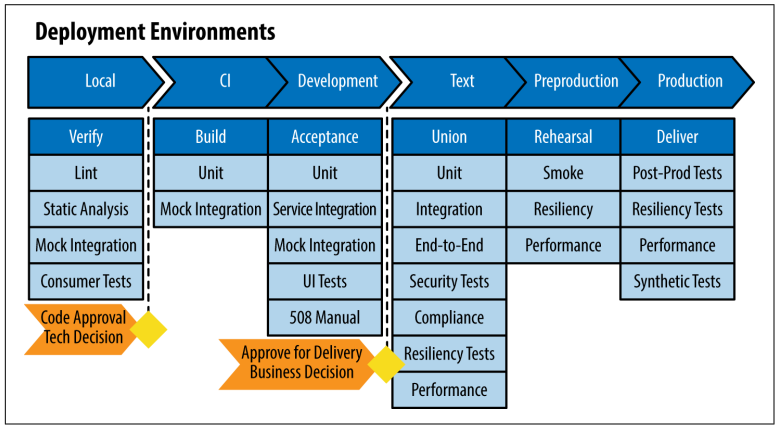


*Figure 1-6. Reference pipeline—continuous delivery*

Continuous delivery usually requires a change in organizational culture. Team and customer expectations must shift from full-fledged deployment when all aspects of the project are finished and tested to incremental and iterative application distribution. Simply put, you can view the objectives of continuous delivery as follows:

- New features, configuration changes, bug fixes, and experiments are sent into production safely and quickly in a sustainable way.
- Forget about having to spend nights and weekends in the datacenter to deploy releases; design the architecture to deploy programs whenever the time is appropriate for the DevOps team and the user community.

To achieve these objectives, DevOps teams must develop maturity and capabilities in the four DevOps practices described previously, or continuous delivery is *not* viable. The four critical practices and the required capabilities are as follows:

- CM
  - Builds should be scripted. Developers should use the same deployment scripts that are used for production.

  - Environments should be automated and scripted. There should be no differences between the developer and production environments.
  - It should not be a lengthy task to bring a new developer onto the team.
- CI
  - Automated builds are triggered for every check-in.
  - Developers check into the trunk (of the source code repository) at least once per day.
  - When the build goes red (i.e., when an automated test fails), the team should fix it quickly, ideally within 10 minutes.
- Automated testing
  - Unit tests for all public interfaces are in place to provide some level of confidence before exploratory testing (which is based on manual testing) and acceptance.
  - Acceptance tests are automated. The gatekeeper agrees that the capability is ready to push to production deployment.
- IaC
  - The instantiation and teardown of the infrastructure for every environment us managed, automated, and executed through scripts and code using tools such as Chef, Puppet, Ansible, and others.
  - The infrastructure is immutable and there are no differences between the developer, implementation, and user environments.

These four DevOps practices cover the following two rules of continuous delivery:

*The software (trunk) should always be deployable*
: Trunk software should be healthy at all times and ready to be deployed to production. It's up to the gatekeeper to decide whether to deploy it once per day or multiple times per day—this is a business decision, not a technical decision.

*Everyone checks into the trunk from feature branches at least once per day*

This forces developers to divide user stories and features into meaningful, smaller pieces. Every check-in is performed to determine if the team's "definition of done" has been achieved.

In deploying to production, there are several advanced strategies that you can adopt to achieve zero downtime and flexibility:

*Canary releases*

By definition, continuous delivery deploys many builds to production. In "Practice 3: Automated Testing" on page 20, we discuss the importance of having a sophisticated set of automated tests that provide a high level of confidence in the quality of the code. Canary is a technique that is used to further increase confidence levels before deploying new code to production. In a canary deployment, the new code is delivered only to a percentage of the existing infrastructure. For example, if the system is running on 10 load-balanced virtual servers, you can define a canary cluster of one or two servers. This way, if the deployment is not successful due to an escaped defect, it can be caught before the build is deployed to all of the servers. Canary releases are also used for pilot features to determine performance and acceptance prior to a full rollout.

*Blue/green deployment*

This is a zero-downtime deployment technique that involves a gradual release to ensure uninterrupted service. The blue/green approach is effective in virtualized environments, especially if IaaS is used. Although a blue/green deployment is a deep topic that deserves an entire chapter on its own, simply put, it includes maintaining two identical development environments —Blue and Green. One is a live environment for production traffic, whereas the other is used to deploy the new release. In our example, let's say that Green is the current live production environment and Blue is the idle identical production environment. After the code is deployed and tested in the Blue environment, we can begin directing traffic of incoming requests from Green (current production) to Blue. You can do this gradually until the traffic redirect is 100 percent to the Blue environment. If unexpected issues occur during this gradual release, we can roll back. When it is completed, the Green environment

becomes idle and the Blue environment is now the live production environment.

Getting to this point in continuous delivery is an incredible milestone, and means that your organization has enough confidence and automation from your implementations CI, CM, automated testing, and IaC that you are able to deploy any point into production.

#### Continuous delivery workshop guiding questions

- Do you have CI in place?
- What is your confidence level in the coverage and quality of your automated testing?
- Do you have IaC for infrastructure? Have you heard developers complaining that they are unable to reproduce an error identified by quality assurance?
- What is the "definition of done" for your code?
- Do you have 100 percent code review for each commit?
- Do you run code quality scans for committed code?
- How does the product owner approve your definition of done?
- Does your contract let you deploy to production, or is there another contractor in the middle?
- Do you run automated tests (continuously) in production for feedback?
- Do you have practices to revert the build?

#### Continuous delivery checklist

- Before attempting continuous delivery, assess your maturity level for the prerequisite practices described in this section. If you do not have sufficient maturity in these practices, make sure to first improve those practices.
- Build your continuous delivery pipeline. Do not reinvent the wheel. Instead, use products such as Jenkins, Chef Delivery, and Spinnaker.
- Integrate your test automation and CI server to the continuous delivery pipeline.

- Decide whether to use canary or blue/green deployments. If you do, build the mechanisms and determine how the deployed build will be monitored.
- There are two manual checkpoints for continuous delivery: code approval and delivery approval. Document these two approval points and communicate them:
  - Code approval
    - Define who will perform this approval. Usually, it is the technical lead.
    - Define the developer's tasks to achieve the "definition of done" and submit the code for approval.
    - Define the reviewer's tasks and standardize these across Agile teams.
    - Configure the delivery pipeline to deploy builds to the proper target environments (development, testing, quality assurance, or production).
    - Construct or configure the trigger that the person responsible for code approval will use to initiate the build automation for the delivery pipeline.
    - Build mechanisms to alert the team if any of the automated tests fail. This mechanism must stop the pipeline when something is found to have gone wrong.
  - Delivery approval
    - Define who is accountable for this approval, which is the last step before deploying the code to production assuming all of the planned automated tests run successfully. Because the code is a candidate to be deployed to production, the product owner should be involved.
- Build advanced deployment techniques (e.g., blue/green deployment, canary release, and or A/B tests) to improve the success of continuous delivery.
- Create mechanisms to monitor production and the success of the build.
- Create mechanisms to revert the build in case of a failure.

## Practice 6: Continuous Deployment

### Continuous deployment definition

Unlike continuous delivery, which means that every change is deployable but might be held back because of business considerations or other manual steps, continuous deployment strives to automate production deployment end to end. With this practice, confidence in the automated tests is extremely high, and as long as the code has passed all the tests, it will be deployed.

Figure 1-7 shows our reference pipeline. We maintain a manual checkpoint for implementation of a business decision about deployment prior to continuing the process to production. You can remove this step for full end-to-end automation of continuous and constant deployment of production changes. In a fully automated scenario, it is vital that you have high maturity levels in each of the DevOps practices and full visibility into the health of the systems at all times. If unexpected issues arise, it is critical to have the metrics, insight, and process to either quickly revert to a known stable state or to fix the issue immediately with a new build.

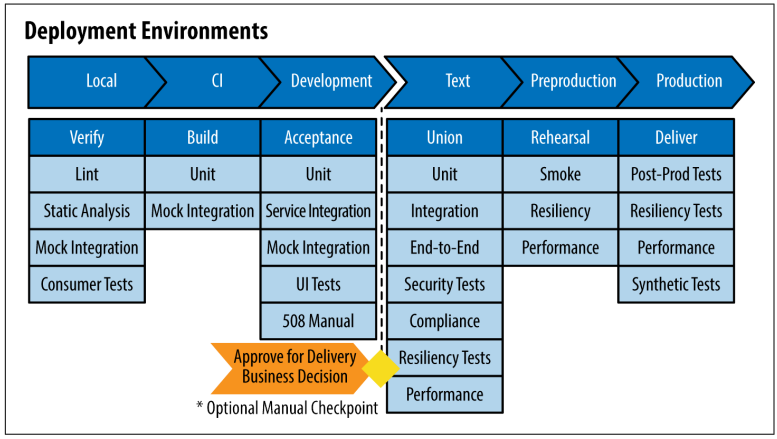


*Figure 1-7. Reference pipeline—continuous deployment*

Continuous deployment is the ultimate goal for modern efficient organizations. However, this DevOps practice does not imply that all applications are deployed without any manual interaction. Here are some factors to examine when considering continuous deployment:

- Automated test coverage
- Automated test types, including unit, functional, accessibility, security, compliance, different types of performance, and acceptance tests
- Robust multilayer code review during developer commit for every change
- Validation of functionality by the product owner during developer commit
- Advance deployment mechanisms, including blue/green deployments, canary testing, post-production testing, and revert scripts.

In deciding which services can be deployed automatically and which need manual validation, most organizations use a hybrid mechanism early on before becoming fully confident as to whether the feature is something that can be fully automated to production. For example, a change in a business rule that requires extensive scenario testing by a subject matter expert (SME) might be better suited for continuous delivery because it would require the SME (and the product owner) to give a thumbs up after going over all possible scenarios. This is especially true in the government world in which there are policies in place that require individuals to give their approval prior to moving on to production.

#### Continuous deployment workshop guiding questions

- Do you have continuous delivery in place? How successful are you with it?
- What are the driving factors to move to continuous deployment?
- How comfortable is your client with continuous deployment?
- How will you ensure that the functionality you deploy still conforms to the product owner's preferences when you eliminate the manual delivery approval step?

- Assess whether you can use continuous delivery and continuous deployment together. What is ideal to be part of continuous deployment for your team? (This could be all components or perhaps just APIs and backend capabilities.)

### Continuous deployment checklist

- Ensure that you are successful with continuous delivery before pursuing continuous deployment.
- Integrate delivery approval with code approval—that is, make the business decision about the deployment when you assess the readiness of the code right before CI.
- Build mature centralized logging and continuous monitoring practices to ensure that you're aware of the impact of every deployment.
- Adopt canary deployment mechanisms.
- Embrace blue/green deployment mechanisms. For continuous deployment, the blue/green approach is a necessity. No matter how good your automated tests are, it is still possible to miss defects. Blue/green techniques help to minimize the chances of releasing defects to all users.
- Integrate all of the chosen deployment techniques with your delivery pipeline. Some continuous delivery pipelines (e.g., Spinnaker) have advanced deployment techniques out of the box.

## Practice 7: Continuous Monitoring

### Continuous monitoring definition

Continuous monitoring is the practice that connects operations back to development, providing visibility and relevant data throughout the development lifecycle including production monitoring. In traditional IT operations, an operations team composed of system administrators, various help desks, and analysts performs operations. But with continuous monitoring, modern DevOps operations teams also include the developers who originally built the system and then are responsible for site reliability, overseeing the software, and acting as first responders for production problems. Simply put,

continuous monitoring aims to reduce the time between identification of a problem and deployment of the fix.

This practice should not be an afterthought. Monitoring begins with Sprint 1 and should be integrated into the development work. As the system is built, monitoring solutions are also designed. In this playbook, we focus on four different types of continuous monitoring:

*Infrastructure monitoring*
: Visualize infrastructure events coming from all computing resources, storage and network, and measure the usage and health of infrastructure resources. AWS CloudWatch and CloudTrail are examples of infrastructure monitoring tools.

*Application performance monitoring (APM)*
: Target bottlenecks in the application's framework. Appdynamics and New Relic are industry-leading APM tools.

*Log management monitoring*
: Collect performance logs in a standardized way and use analytics to identify application and system problems. Splunk and ELK are two leading products in this area.

*Security monitoring*
: Reduce security and compliance risk through automation. Security configuration management, vulnerability management, and intelligence to detect attacks and breaches before they do serious damage are achieved through continuous monitoring. For example, Netflix's Security Monkey is a tool that checks the security configuration of your cloud implementation on AWS.

We can divide continuous monitoring into three major steps. First, the obvious one—monitoring. Second, an alert system to warn the team about a problem. The alert system should be capable of automatically elevating the alert if the team members who initially received notice of the issue take no action. The third step of continuous monitoring is actions to take when an alert occurs. No matter what the nature of the alert, all should be captured and reviewed. For example, there can be false negatives, but they should not be ignored. Instead, false negatives should be reported as such to the continuous monitoring tool so that if the same set of circumstances occurs again, the monitoring equipment will not trigger an alert. Similarly, if an alert is issued, but the software appears to be operating satisfactorily, it should not be neglected. Rather, it should be

investigated and the outcome of the investigation must become a part of the record that the monitoring tool can use in determining the severity of future similar incidents.

### Continuous monitoring workshop guiding questions

- How and why is early detection of defects important to your project?
- What performance SLAs do you have and when do you plan to start achieving them?
- What is the system availability requirement and what is the plan to achieve it?
- When do you do performance testing as part of your continuous integration and automated testing practices? How do you know that the tests represent actual production conditions?
- What is the labor cost of your operations? How many systems engineers, system administrators, and database administrators do you have monitoring the production environments, and how do they react when there is an issue?
- How many errors are there in your application? How do you trace the errors back in your code?
- How do you forecast infrastructure utilization when your code changes?

### Continuous monitoring checklist

- Implement APM, log analytics, and security monitoring solutions for your system.
- Design and start using these implementations as the system is built, not after it is deployed.
- Design and implement alerting solutions composed of alert generation, mitigation actions, verification, baseline calculations, and alarm setup. Build alerts for each try/catch in your code as the code is designed.
- Know your monitoring coverage and identify what it should be.
- Start monitoring performance during CI. Define page-level SLAs, build automated performance tests for these SLAs, and define triggers.

- Define baselines during development to be able to identify later anomalies.
- After defining baselines, define quantitative forecasts for critical processes. For example, login page SLA is two seconds. Based on 95 percent of responses, the system can achieve a 2-second response time for up to 1,000 concurrent users.
- Collect data from servers. After aggregating and storing the data, present the data in graph format, prepared for time series analysis.
- Automatically create issues when the monitoring solution captures a problem and triggers an alert. Automatically group the created issues.
- Allocate developer resources to filter false positives and negatives.

## Play 3: Assess Your DevOps Maturity Level and Define a Roadmap

DevOps is not a set of prescriptive procedures, but it is critical that organizations assess the maturity of their adherence to DevOps practices by measuring them against a standard model. Without this, it would be impossible to identify gaps or potential areas for improvement that could interfere with a successful DevOps implementation.

To make this assessment, we created a Maturity Questionnaire that provides a series of questions related to each of the seven DevOps practices. Each answer has a point value. When you complete the questionnaire, add up the total points and use Table 1-2 to determine your project's DevOps maturity level.

*Table 1-2. The DevOps Maturity Questionnaire*

| Maturity Level | | Points |
|---|---|---|
| Base | Agile practices might be in use, but DevOps practices are limited | 0–32 |
| Beginner | Foundational use of DevOps practices | 33–98 |
| Intermediate | Stabilized use of DevOps practices | 99–164 |
| Advanced | Continuous improvement in place<br>Team has the competence and confidence to use DevOps practices<br>Cultural barriers have been overcome | 165–230 |
| Extreme | No cultural barriers<br>Role model DevOps implementation | 231 |

Booz Allen's Maturity Questionnaire should be the first step toward achieving a more mature DevOps capability because it highlights the current state and provides a roadmap that indicates where to focus on increasing your organizational maturity. Some DevOps practices are more difficult to introduce than others and will require substantial changes in the fundamental management of software development. Over time, as an organization grows in DevOps maturity, improvements will bring tangible increases in both efficiency and quality of software development and implementation.

Tables 1-3 through 1-9 highlight our maturity assessment.

*Table 1-3. Configuration management*

| Maturity Level | Base | Beginner | Intermediate | Advanced | Extreme | *YOUR SCORE* |
|---|---|---|---|---|---|---|
| **Source code management (SCM) use** | No SCM | SCM for application code | + Configuration, configuration scripts, and infrastructure code | + Test source code | + Builds and containers | **Your score** |
| **Points** | **0** | **1** | **3** | **5** | **7** | |
| **Source-code branching** | No branching strategy | Multiple repositories (copies of source code) used instead of branching | Centralized workflow (single point of entry for all changes) | Feature branch workflow (dedicated branch for each feature versus using a centralized single location) | Gitflow workflow (structured branching policy that accounts for features, hotfixes, and releases) | **Your score** |
| **Points** | **0** | **1** | **3** | **5** | **7** | |

*Table 1-4. Continuous integration*

| Maturity Level | Base | Beginner | Intermediate | Advanced | Extreme | *YOUR SCORE* |
|---|---|---|---|---|---|---|
| **CI prerequisites** | Not ready for CI | Regular developer check-in to development branch | Comprehensive automated test harness exists | Developer environment (local) has access to refreshable test data, application build scripts, standardized environment (app and database) | Automated build scripts and short test process exist AND developer runs for each check-in | **Your score** |
| **Points** | **0** | **1** | **3** | **5** | **7** | |
| **CI tool use** | No CI tool | CI tool with manual build controls | CI tool with scheduled automated builds (tool does not have the knowledge of change in source code branch) | CI tool with automated change detection and automated deployment from development branch ONLY to development environment | CI tool with automated change detection and automated deployment from all branches (development, feature, release) to all environments (Dev, test, QA) | **Your score** |
| **Points** | **0** | **1** | **3** | **5** | **7** | |
| **Automation controlled by CI tool** | Only build automation to development | + Build automation to all of the environments (test, QA, and others) | + Unit test for all layers of the application | + Security tests | + Performance tests | **Your score** |
| **Points** | **0** | **1** | **3** | **5** | **7** | |

| Maturity Level | Base | Beginner | Intermediate | Advanced | Extreme | *YOUR SCORE* |
|---|---|---|---|---|---|---|
| **Integration of developer code** | No check-in, integrate until the whole capability (e.g., module or feature) is complete | Integrate daily (no unit test runs before commit) | Integrate as methods (smallest executable code) are complete (no unit test runs before commit) | Integrate daily after running all unit tests, if unit tests pass; otherwise, STOP | Integrate as methods (smallest executable code) are complete after running all unit tests, if unit tests pass; otherwise, STOP | **Your score** |
| **Points** | **0** | **1** | **3** | **5** | **7** | |
| **Alerts/ notifications and actions during CI** | No ACTION is taken for results of CI activities | If the build breaks due to a check-in, ACTION = STOP<br>(a) stop build process<br>(b) prevent other check-ins to the broken code | When build does not break with check-in, but unit test fails,<br>ACTION = REVERT back to the file, but other developers can continue to check their code | When there is an action, have mechanisms in place to communicate to a configurable set of team members (e.g., for a defect, notify developer and technical lead) | Automatically stop the build process until all automated tests pass and there are no build errors | **Your score** |
| **Points** | **0** | **1** | **3** | **5** | **7** | |

*Table 1-5. Automated testing*

| Maturity Indicator | Base | Beginner | Intermediate | Advanced | Extreme | *YOUR SCORE* |
|---|---|---|---|---|---|---|
| **Automated testing** | No automated tests | Unit tests | + Functional Scenario Tests | + Security Tests | + Performance Tests | **Your score** |
| **Points** | **0** | **1** | **3** | **5** | **7** | |
| **Actions for automated test results** | No action taken | Actions are documented and it is left up to developers to fix the problems | Actions are taken for each sprint | Problems are reviewed by development team and actions planned into the sprint plans | When any automated test fails, team stops to triage the problem | **Your score** |
| **Points** | **0** | **1** | **3** | **5** | **7** | |
| **Unit tests** | No unit tests | Few simple tests | Design for testability | Test-driven development for both UIs and APIs | Code coverage | **Your score** |
| **Points** | **0** | **1** | **3** | **5** | **7** | |
| **Unit test coverage** | No unit test coverage tool used | Coverage <25% | Coverage >25% to <50% | Coverage >50% to <75% | Coverage >75% | **Your score** |
| **Points** | **0** | **1** | **3** | **5** | **7** | |
| **Unit test frequency** | No unit test | Developers run their own tests in ad hoc fashion | Developers run the entire harness at commit | CI tool runs the harness for development environment and builds for the entire build | CI tool runs the harness for every build on every environment | **Your score** |
| **Points** | **0** | **1** | **3** | **5** | **7** | |

| Maturity Indicator | Base | Beginner | Intermediate | Advanced | Extreme | *YOUR SCORE* |
|---|---|---|---|---|---|---|
| **Scenario (functional or story) test coverage** | No automated scenario test | User story coverage <25% | User story coverage >25% to <50% | User story coverage >50% to <75% | User story coverage >75% | **Your score** |
| **Points** | **0** | **1** | **3** | **5** | **7** | |
| **Performance tests** | No performance test | Performance test selected functionality (smoke tests) | Performance test everything | Performance test with SLAs (transaction SLA, page load SLA) | Performance measurement and tests in production | **Your score** |
| **Points** | **0** | **1** | **3** | **5** | **7** | |
| **Performance test frequency** | No performance test | Ad hoc | For every release | For every sprint | Continuously | **Your score** |
| **Points** | **0** | **1** | **3** | **5** | **7** | |
| **Performance test types** | No performance tests | Performance test | + Load test | + Soak test (application might work well for a period of time under load, and then fail) | + Stress test (test application's limits) | **Your score** |
| **Points** | **0** | **1** | **3** | **5** | **7** | |
| **Security test types** | No security tests | Vulnerability scanning | + Automated code review | + Static code analysis | + Penetration testing | **Your score** |
| **Points** | **0** | **1** | **3** | **5** | **7** | |
| **Vulnerability scanning frequency** | No vulnerability scanning | Performed by another team after the release is ready | For every release by the development team | For every sprint, by the development team | For every checked-in file, by the CI tool | **Your score** |
| **Points** | **0** | **1** | **3** | **5** | **7** | |

| Maturity Indicator | Base | Beginner | Intermediate | Advanced | Extreme | *YOUR SCORE* |
|---|---|---|---|---|---|---|
| **Automated code quality review frequency** | No automated code quality scanning | Performed by technical lead at random code reviews | For every release by the development team | For every sprint, by the development team | For every checked in file, by the CI tool | **Your score** |
| **Points** | **0** | **1** | **3** | **5** | **7** | |
| **Static analysis** | No static analysis tool used | Heavy-duty static analysis tool (e.g., IBM AppScan) used at project level | Developers use static analysis tool (suitable for the technical stack) for each file commit | CI initiates the tool developers use at every code deployment | CI initiates the developer code at every deployment for the code base and the project tool for defined frequency (e.g., every sprint, release) | **Your score** |
| **Points** | **0** | **1** | **3** | **5** | **7** | |
| **Penetration Testing Frequency** | No penetration scanning | Performed by another team at the time of a security accreditation | Performed by another team for every release | Performed by another team for every sprint | For every checked in file, by the CI tool | **Your score** |
| **Points** | **0** | **1** | **3** | **5** | **7** | |

*Table 1-6. IaC*

| Maturity Indicator | Base | Beginner | Intermediate | Advanced | Extreme | *YOUR SCORE* |
|---|---|---|---|---|---|---|
| **Automated infrastructure provisioning** | No automated infrastructure provisioning | Data center set up, physical servers virtualized, and allocated to applications manually | IaaS is used; on demand infrastructure resources creation when manually requested | Infrastructure provisioning automated by APIs, but not tied to application scalability | Infrastructure is provisioned dynamically as the use increases and decreases | **Your score** |
| **Points** | **0** | **1** | **3** | **5** | **7** | |
| **Containerization use** | No containerization | Containerized monolithic application deployment | Containerized modules and independent provisioning of the containerized modules | Containerized microservices and auto scale of microservices on existing ready–to-use infrastructure | Containerized microservices and integrated auto scale of containers and underlying infrastructure | **Your score** |
| **Points** | **0** | **1** | **3** | **5** | **7** | |

*Table 1-7. Continuous delivery*

| Maturity Indicator | Base | Beginner | Intermediate | Advanced | Extreme | *YOUR SCORE* |
|---|---|---|---|---|---|---|
| **Automated acceptance test**s | No automated acceptance test suite<br>There is human interaction and gate reviews | Automated scenario test results provided to the client, but the results do not impact acceptance because the coverage is not 100% | 100% coverage with automated acceptance tests used as the "definition of done" for the development team | Product owner uses the results, but performs manual tests as well before go-live decision | Product owner relies on automated acceptance tests completely to go live with the build | **Your score** |
| **Points** | **0** | **1** | **3** | **5** | **7** | |
| **Contract constraints** | Contract does not allow results of automated tests to be used for delivery | Testing for acceptance done by another contractor (or party)<br>Development contractor is not responsible for this acceptance | Builds tested and delivered to development and test environments without manual intervention (fully automated) | Builds to QA and UAT environments pass through manual testing gate | Builds delivered to QA and UAT environments without any manual intervention (i.e., data, configuration, networking) | **Your score** |
| **Points** | **0** | **1** | **3** | **5** | **7** | |
| **Deployment pipeline** | No automated deployment pipeline | Deployment pipeline allows only CI activities | Deployment pipeline allows CI and deployment to QA and test environments | Deployment pipeline is configurable to enable full automation from development to production deployment, but cannot be used due to contractual gates | Fully automated deployment pipeline exists<br>It's possible to insert a manual approval step, which is used with or without a manual check for production deployment | **Your score** |
| **Points** | **0** | **1** | **3** | **5** | **7** | |

*Table 1-8. Continuous deployment*

| Maturity Indicator | Base | Beginner | Intermediate | Advanced | Extreme | *YOUR SCORE* |
|---|---|---|---|---|---|---|
| **Deployment to production** | System is taken offline<br>All conducted manually using standard operating procedures and step-by-step installation guide | System is taken offline<br>Step-by-step installation, but SOME steps include automated configurations | System is taken offline<br>Step-by-step installation, but ALL steps include automated configurations | System is taken offline<br>One-click deployment is performed and tested before go live | Zero downtime release<br>System is upgraded with the build without taking the system down | **Your score** |
| Points | 0 | 1 | 3 | 5 | 7 | |
| **Deployment capabilities** | No automated deployment | Deployment replaces the existing build | Canary releasing; build deployed to canary environment first<br>After testing the canary in production build, the system is taken down and the upgrade is performed | First deploy to canary and then upgrade servers one at a time until the deployment is complete<br>No downtime | Blue/green deployment<br>No downtime (two identical versions of production: blue and green; one is upgraded and replaced with the other at the DNS level) | **Your score** |
| **Points** | **0** | **1** | **3** | **5** | **7** | |

*Table 1-9. Continuous monitoring*

| Maturity Indicator | Base | Beginner | Intermediate | Advanced | Extreme | *YOUR SCORE* |
|---|---|---|---|---|---|---|
| **Monitoring solution** | No monitoring solution<br>Monitoring is performed by operations team (system administrators and DBAs) | Limited used of monitoring tools by the operations team | Monitoring types:<br>• Application performance monitoring<br>• Log monitoring and analysis<br>• Security monitoring | Three monitoring types:<br>• Application performance monitoring<br>• Log monitoring and analysis<br>• Security monitoring | All of these monitoring types:<br>• Application performance monitoring<br>• Log monitoring and analysis<br>• Security monitoring | **Your score** |
| **Points** | **0** | **1** | **3** | **5** | **7** | |
| **Application performance monitoring** | No application performance monitoring | Manual application performance monitoring using OS tools | Easy access to real-time statistics on application performance in production | Ability to isolate issues in production down to the individual servers/VM/ containers and processes | Ability to view the entire stack of any issue identified, from initial request down to the database | **Your score** |
| **Points** | **0** | **1** | **3** | **5** | **7** | |
| **Log monitoring** | No log monitoring | Log monitoring using OS tools | All logs (i.e., application, security, web access) easily accessible for review | All logs consolidated and put in a central location | All logs are indexed and quickly searchable | **Your score** |
| **Points** | **0** | **1** | **3** | **5** | **7** | |

| Maturity Indicator | Base | Beginner | Intermediate | Advanced | Extreme | *YOUR SCORE* |
|---|---|---|---|---|---|---|
| **Security monitoring** | No security monitoring | Security monitoring using basic OS and network tools | Simple threat identification, such as various DoS attacks | Advanced network monitoring to actively find vulnerabilities or active attacks | Monitor payloads that are hitting the system for identifying possible attacks | **Your score** |
| **Points** | **0** | **1** | **3** | **5** | **7** | |
| **Alerting solution** | No alerting | Responsible parties are alerted by team monitoring logs, application, and security | Alerts provide detailed information about the nature of monitoring trigger | Alerting solution provides historic list of previous events, including event details | Alerting thresholds are modifiable | **Your score** |
| **Points** | **0** | **1** | **3** | **5** | **7** | |

# Play 4: Create a DevOps Pipeline

So how do you put DevOps into action once you have a clear roadmap and identify what DevOps means to your organization? We suggest you begin by defining and creating a DevOps delivery pipeline. This is the set of tools and processes working together to provide workflow automation that reflects your DevOps practices taking code changes all the way through into production. The shape of the pipeline, the activities inside each step, what steps are automated versus manual, and code release and deployment strategies will reflect your DevOps practices, requirements, and philosophies. Every environment and pipeline is different, but the characteristics of a successful delivery pipeline are the same, providing the following:

- Automation of building, testing, and deploying
- Automation of infrastructure, which can be created and destroyed without impacting the health of the software
- Reflective of your release DevOps philosophy and strategy
- Repeatable and expected results with immutable infrastructure and processes
- Visibility into the entire pipeline workflow steps.

There is a wide range and constantly evolving set of technology and tools for implementing your delivery pipeline. Key factors to consider when choosing your set of DevOps tools include team skillset, breadth of required hosting providers/infrastructure, availability of the tools' APIs, and, ultimately, the tools' ability to execute your DevOps practices' requirements and philosophy.

The delivery pipeline you build should first satisfy the basic flow. With it, you should be able to do the following:

- Check out code
- Change the code and integrate it into the repository
- Run validation and predeployment automated tests to ensure the code meets required needs, does not break other existing code, and runs as expected
- Deploy your builds on predefined infrastructure clusters

- Move the build from development to testing, testing to QA, and finally to production.

When the basic flow is perfected, you can begin exploring advanced concepts that can help you to safely and reliably deploy, easily scale, and roll back or forward. Among the advanced concepts, you should become familiar with these:

- Spin up or down virtual machines (or on IaaS platforms) based on user load
- Containerize (e.g., use Docker) your application and deploy it on a scalable server cluster
- Provision containers based on user load
- Explore microservices architecture. This is not easy to implement and there are many considerations that must be addressed involving this approach, including service discovery, communications, and orchestration. These aspects are beyond the scope of this playbook.

It is important to approach and look at your pipeline as an enterprise change management workflow because handling one project (delivery pipeline) is not the same when you have multiple delivery streams that need to be validated and merged into a shared workflow for handling dependencies. Simplh having automation and repeatability does not necessarily equate to an effective and scalable pipeline. You want to avoid creating a complex Rube Goldberg–type contraption and keep steps simple as appropriate and select tools for what they are good/meant for versus doing heavy customization and or using a large of amount of plug-ins.

## Guiding Questions

- Are there any steps that cannot be automated and will need manual review and/or acceptance?
- Is the goal to move to a microservices architecture?
- How many builds to production do you want to target/require on a daily/weekly/monthly basis?
- What SLAs do you want to automate?
- What platforms and hosting providers do you need to support?

- How many features are you anticipating?
- Do you currently use a canary release and or blue/green deployment strategy when rolling features out to production?
- How are production rollbacks typically handled?
- Are you planning to move to containerization architecture soon?

### Checklist

- Defined repeatable automated and manual steps that every code change will go through—the workflow does not change and provides expected steps and results every time.
- Established and verified full traceability for each step in the pipeline—the ablility to see where a change is in the pipeline at any time and its status.
- Implemented notification and resolution process for each success/fail action for each step—ensure that you clearly define responsibility groups for each action issue.
- Verified immutable infrastructure—your IaC is able to tear down and bring up each environment over and over again with the same expected state and results.
- Ensured metrics defined in continuous monitoring are captured, visible, and integrated with your notification process.

## Play 5: Learn and Improve through Metrics and Visibility

Now that you have created a pipeline and have a delivery flow that's running, you'll need to know how effective it is and what you can improve. One of the key principles we highlighted earlier is being a learning organization, and that the mastery of DevOps requires constant feedback and an environment that fosters continuous learning. To learn, you need to have the metrics and visibility into the effectiveness of the processes, environments, and operations. In a DevOps project, metrics for monitoring project performance and capturing project data serve five critical purposes:

- Detect failure
- Diagnose performance problems
- Plan capacity
- Obtain insights about user interactions
- Identify intrusions

Because systems are constantly increasing in complexity, breadth of distribution, scope, and size, measuring their activities and levels of efficacy—and logging the results in data banks—demands a new generation of infrastructure and services to support these efforts. Given with the right equipment in place, the value of metrics spans a broad swath of information, from systems health and performance to end-user habits.

For example, when applications or programs fail, metrics provide context to alerts, opening windows into what activities occurred and what interactions took place leading up to each failure. Equally important, metrics offer historical awareness of usage patterns, which is critical for anticipating potential failures, writing fixes that could shore up programs during oversubscribed periods, and determining how robust future software must be. For this purpose, questions that metrics can answer include the following:

- What are the peak hours of the day, days of the week, or months of the year for utilization?
- Is there a seasonal usage pattern, such as summertime lows, holiday highs, more activity when school is in session or when it isn't, and so on?
- How do maximum (peak) values compare against minimum (valley) values?
- Do peak and valley relationships change in different regions around the globe?

In a large-scale system, ubiquitous monitoring can generate data involving millions of events with countless numbers of log lines devoted to metrics measurements. This, in turn, can monopolize overhead and affect performance, transmission, and storage. The emergence of big data analytics and modern distributed logging alleviates this problem. Moreover, advanced machine learning algorithms can deal with noisy, inconsistent, and voluminous data.

When deciding how much data resolution to maintain for metrics, you need to think about the type and amount of information that you want to get from them. Will you be depending on metrics for insight into what is causing an outage or degradation? If so, you'll most likely want to have a fine resolution, less than a minute. Or will you be using the data primarily for capacity planning on a three-, six-, or nine-month timeline? If so, you'll want to ensure that you can retain the historical details about maximum and minimum over a long period of time.

At the very least, the metrics in place should effectively and continuously monitor the following four fundamental DevOps facets:

*Deployment frequency*
: How often does new code reach customers? DevOps practices make frequent or continuous program delivery possible, and large, high-traffic websites and cloud-based services make it a necessity. With fast feedback and small-batch development, updated software can be deployed every few days, or even several times per day. In a DevOps environment, delivery (i.e., deployment to production) frequency can be a direct or indirect measure of response time, team cohesiveness, developer capabilities, development tool effectiveness, and overall DevOps team efficiency.

*Change lead time (from development to production)*
: How long does it take, on average, to move code from development through a cycle of A/B testing to 100 percent deployed and upgraded in production? The time from the start of a development cycle (the first new code) to deployment is the change lead time. It is a measure of the efficiency of the development process, of the complexity of the code and the development systems, and (like deployment frequency) of team and developer capabilities. If the change lead time is too long, it might be an indication that the development and deployment process is inefficient in certain stages or that it is subject to performance bottlenecks.

*Change failure rate (per week)*
: What percentage of deployments to production failed or reverted back to be fixed with another patch? One of the main goals of DevOps is to turn rapid, frequent deployments into an everyday affair. For such deployments to have value, the failure rate

must be low. In fact, the failure rate must decrease over time, as the experience and the capabilities of the DevOps teams increase. A rising failure rate, or a high failure rate that does not decline over time, is a good indication of problems in the overall DevOps process.

*Mean time to recovery (MTTR)*
: What is the mean time to recover from a failed deployment—that is, the time from failure to recovery from that failure? This generally is a good measure of team capabilities and, like the failure rate, it should show an overall decrease over time (allowing for occasional longer recovery periods when the team encounters a technically unfamiliar problem). MTTR can also be affected by such things as code (or platform) complexity, the number of new features being implemented, and changes in the operating environment (e.g., migration to a new cloud server).

In addition to these essential four metrics, there are others that we recommend DevOps teams consider. The more information you have, the more successful your DevOps projects will be. Among the other benchmarks to assess are the following:

*Delivery frequency*
: How often is code deployed to the development and test environments?

*Change volume*
: For each deployment, how many user stories and new lines of code are making it to production?

*Customer tickets (per week)*
: How many alerts are generated by customers to indicate service issues?

*Percentage change in user volume*
: How many new users are signing up and generating traffic?

*Availability*
: What is the overall service uptime and were any SLAs violated?

*Response time*
: Does the application's performance reach the predetermined thresholds?

In addition to the nitty-gritty, day-to-day performance and usage patterns that DevOps metrics excel in providing, there are two other areas of organizational activities that well-designed standards can monitor for strengths and weaknesses: cultural metrics and process metrics. Let's look more closely at each one.

## Cultural Metrics

DevOps is meant to include a set of efficiency and improvement principles that should minimize project development conflict and eliminate stress and burnout. In turn, team members will ideally be more healthy, loyal to the organization, and deeply engaged in workplace activities. It's possible to measure across a number of key cultural indicators, including sentiment toward change, failure, and a typical day's work. Among the most telling metrics to be sought in this regard are the following:

*Cross-skilling*
: How much knowledge sharing and pairing exists among teams?

*Focus*
: Are teams working in a fluid and focused manner toward achieving common goals or objectives?

*Multidisciplinary teams*
: Do teams comprise members with varied but complimentary experience, qualifications, and skills?

*Project-based teams*
: Are teams organized around projects rather than solely skill-sets?

*Business demand*
: Are the demands placed on development teams by the business side too onerous?

*Extra lines of code*
: How many extraneous lines of code exist in the project?

*Attitude*
: Are team members receptive to and positive about continuous improvement?

*Number of metrics*
: Is the obsession with metrics perceived to be too high?

*Technological experimentation*
: What is the degree of experimentation and innovation within the project?

*Team autonomy*
: How successfully does the team manage its own work and working practices?

*Rewards*
: Do team members feel appreciated and rewarded for their work and successes?

As you can tell, many of these cultural metrics cannot be directly measured. That is why we have stressed the mindset of becoming a learning organization and having transparency and visibility into the end-to-end process. For example, with regard to cross-skilling, one way to assess that is to track to see if there's a high variance in the velocity across Agile teams, especially knowing that team members are being shuffled. The takeaway here is that in order to gauge the impact and effectiveness of cultural changes, you need to establish a means for constant feedback and dialogue with the team.

## Process Metrics

One goal of a typical DevOps project is to achieve continuous deployment. This occurs by linking software development processes and tools together to allow fully tested, production-ready, committed code to proceed to a live environment without user interaction. This software infrastructure portion of a DevOps project is often termed the *DevOps toolchain*. It's useful to measure the relative maturity of the component processes of the toolchain as a proxy for overall DevOps capabilities. Typically, we look at an organization's skills in the following areas:

- Project requirements gathering and management
- Adherence to Agile development principles
- Whether the software build is generally defect-free
- Fluidity of releases and deployment
- Degree to which units of code are tested to determine their suitability for use
- Degree of user acceptance testing

- Quality assurance programs
- Performance monitoring to ensure the program is reliable and can scale
- Cloud testing to be certain that the application and its load can be supported

Also under the umbrella of process is *sharing*, which is another area that is often overlooked but should be encouraged—and measured. People from different parts of an organization often have different, but overlapping, skillsets. For example, this is true of staffers on the development side and the operations side, the disparate parts of the enterprise that DevOps is meant to link together. Given the importance of sharing between these teams, and the benefits to be gained by an organization when there is a maximum amount of sharing, it's useful to measure the frequency of sharing.

Examples of workplace sharing that you can measure, and the aspects of a DevOps project that these collaborative efforts affect, include the following:

- Shared Goal: Reliability and speed
- Shared Problem Space: Deployment and delivery
- Shared Priorities: Improvement decisions
- Shared Location: Communications
- Shared Communication: Chat, wiki, mailing list
- Shared Codebase: Code and infracode
- Shared Responsibility: Building and deployment
- Shared Workflow: One-button deployment
- Shared Reusable Environments: Reusable recipes
- Shared Process: Standups and releases
- Shared Knowledge: One ticketing system
- Shared Success and Failure: Common experience and history

## Metrics Tools

There are many monitoring and metrics systems and tools available, both from open source and commercial developers. Typical systems

include Nagios; Sensu and Icinga; Ganglia; and Graylog2, Logstash, and Splunk:

*Nagios*
: Nagios is probably the most widely used monitoring tool due to its large number of plug-ins, which are basically agents that collect metrics in which you are interested. However, Nagios' core is essentially an alerting system with limited features, and Nagios is weak in dealing with the frequent changes of servers and infrastructure encountered in cloud environments.

*Sensu and Icinga*
: Sensu is a highly extensible and scalable system that works well in a cloud environment. Icinga is a fork of Nagios with a more scalable distributed monitoring architecture and easy extensions. Icinga also has stronger internal reporting systems than Nagios. Both Sensu and Icinga can run Nagios's large plug-in pool.

*Ganglia*
: Ganglia was originally designed to collect cluster metrics. It is designed to have node-level metrics replicated to nearby nodes to prevent data loss and over-chattiness to the central repository. Many IaaS providers support Ganglia.

*Graylog2, Logstash, Splunk*
: These distributed log management systems are tailored to process large amounts of text-based metrics logs. They have frontends for integrative exploration of logs and powerful search features.

# Summary

There is plenty of information, excitement, value, promise, and confusion that comes with DevOps. The benefits are clear: improved quality, flexibility, speed to value, increased efficiency, and potential cost savings. Less clear, however, is the best approach to adopting DevOps practices. Adopting DevOps practices involves a mindset change that is built on the right mix of people and culture, an understanding of DevOps practices and how they relate to your projects, and, ultimately, choosing and implementing tools to put DevOps practices into action through a delivery pipeline.

Selecting DevOps tools is a challenging task given the many tools available. We recommend aligning the tools with your organization's skillsets, flexibility needs, and modularity bias. This technical landscape is changing constantly, with updated versions, open source efforts, and new solutions. Make sure the tools you select do not require custom integration or a high level of consolidation, which might lead to a large effort to swap out the application down the road.

Most organizations have trouble establishing appropriate requirements and goals for a DevOps program. You will need initial targets to quantify your successes, and those targets will not be the same from one team to another. Consequently, every organization will implement DevOps to different levels of maturity. We hope this report has provided you with a solid foundation of what DevOps means, and more importantly, a framework for developing an effective adoption plan or to incorporate/assess your current efforts:

- Understand each DevOps practice and how it conforms with your organization's objectives and goals
- Assess the level of your organization's DevOps capabilities
- Determine how far you need to go and what you need to do to achieve the DevOps level of performance that you want.

Understanding these three items will put you on the road to a successful and enduring DevOps practice. We look forward to hearing your success stories!

# Recommended Reading

We recommend the following reading that dives deeper into each of the areas we touched upon in this report, from culture to technical details around continuous delivery and microservices:

- *The Phoenix Project: A Novel about IT, DevOps, and Helping Your Business Win,* Gene Kim and Kevin Behr (IT Revolution Press).
- *The Fifth Discipline: The Art & Practice of The Learning Organization*, Peter M. Senge (Doubleday Business).

- *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*, Gene Kim and Patrick Debois (IT Revolution Press).
- *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, by Jez Humble and David Farley (Addison-Wesley Professional).
- *Building a DevOps Culture*, Mandi Wells (O'Reilly).
- *The DevOps 2.0 Toolkit: Automating the Continuous Deployment Pipeline with Containerized Microservices*, Viktor Farcic (CreateSpace Independent Publishing Platform).
- *Building Microservices*, Sam Newman (O'Reilly).

## About the Authors

**Bill Ott** is a Vice President with Booz Allen Hamilton, where he leads a group of creative and technology professionals who are passionate about integrating human-centered design, Agile development, DevOps, security, and advanced analytics to build digital services that users will use and enjoy, securely. His inspiration comes from his three boys who love technology—specifically Minecraft gaming/programming and creating and watching YouTube videos. Mr. Ott holds a BS in electrical engineering from Drexel University and an MBA from Emory University.

**Jimmy Pham** is an avid technologist who has designed, developed, and managed large software solutions for major private and public customers. He is currently a Chief Technologist focusing on modern software development. His interests and experience also span web acceleration/performance and cloud security. Prior to Booz Allen Hamilton, he worked at Akamai and ran a startup. He holds a degree in Computer Science (BSE) and minors in Mathematics and Psychology.

**Haluk Saker** is a director with the Digital team and a 20-year veteran of Booz Allen. An experienced system/cloud architect, he leads Digital's DevOps practice, microservices architecture, and numerous cloud platforms investments. He is also one of the coauthors of the Booz Allen Agile Playbook that is used by all software development teams at the firm. He has an extensive background in turnkey system and cloud implementations, modern technology stacks, and Continuous Deployment. Haluk holds a BS in Electrical Engineering, an MS in Engineering Management, and an MS in Management Information Systems.