Module – 3

⇒ Deployment Pipeline

Continuous integration is an enormous step forward in productivity and quality for most projects that adopt it. It ensures that teams working together to create large and complex systems can do so with a higher level of confidence and control than is achievable without it. CI ensures that the code that we create, as a team, works by providing us with rapid feedback on any problems that we may introduce with the changes we commit. It is primarily focused on asserting that the code compiles successfully and passes a body of unit and acceptance tests. However, CI is not enough. CI mainly focuses on development teams. The output of the CI system normally forms the input to the manual testing process and thence to the rest of the release process. Much of the waste in releasing software comes from the progress of software through testing and operations.

⇒ A basic deployment pipelines

The process starts with the developers committing changes to their version control system. At this point, the continuous integration management system responds to the commit by triggering a new instance of our pipeline. The first (commit) stage of the pipeline compiles the code, runs unit tests, performs code analysis, and creates installers. If the unit tests all pass and the code is up to scratch, we assemble the executable code into binaries and store them in an artifact repository. Modern CI servers provide a facility to store artifacts like these and make them easily accessible both to the users and to the later stages in your pipeline. Alternatively, there are plenty of tools like Nexus and Artifactory which help you manage artifacts.

The second stage is typically composed of longer-running automated acceptance tests. Again, your CI server should let you split these tests into suites which can be executed in parallel to increase their speed and give you feedback faster—typically within an hour

or two. This stage will be triggered automatically by the successful completion of the first stage in your pipeline.
At this point, the pipeline branches to enable independent deployment of your build to various environments—in this case, UAT (user acceptance testing), capacity testing, and production. Often, you won't want these stages to be automatically triggered by the successful completion of your acceptance test stage. Instead, you'll want your testers or operations team to be able to self-service builds into their environments manually. To facilitate this, you'll need an automated script that performs this deployment.
Finally, it's important to remember that the purpose of all this is to get feedback as fast as possible. To make the feedback cycle fast, you need to be able to see which build is deployed into which environment, and which stages in your pipeline each build has passed.

⇒ Deployment Pipeline Practices

1. Only Build Your Binaries Once

Many build systems use the source code held in the version control system as the canonical source for many steps. The code will be compiled repeatedly in different contexts: during the commit process, again at acceptance test time, again for capacity testing, and often once for each separate deployment target. Every time you compile the code, you run the risk of introducing some difference. The version of the compiler installed in the later stages may be different from the version that you used for your commit tests. You may pick up a different version of some third-party library that you didn't intend. Even the configuration of the compiler may change the behavior of the application.
This antipattern violates two important principles. The first is to keep the deployment pipeline efficient, so the team gets feedback as soon as possible. Recompiling violates this principle because it takes time, especially in large systems. The second principle is to always build upon foundations known to be sound. The binaries that get deployed into production should be the same as those that went through the acceptance test process.

If we re-create binaries, we run the risk that some change will be introduced between the creation of the binaries and their release, such as a change in the toolchain between compilations, and that the binary we release will be different from the one we tested. For auditing purposes, it is essential to ensure that no changes have been introduced, either maliciously or by mistake, between creating the binaries and performing the release.

2. Deploy the same way to every environment

It is essential to use the same process to deploy to every environment—whether a developer or analyst’s workstation, a testing environment, or production—in order to ensure that the build and deployment process is tested effectively. Developers deploy all the time; testers and analysts, less often; and usually, you will deploy to production fairly infrequently. However, this frequency of deployment is the inverse of the risk associated with each environment. The environment you deploy to least frequently (production) is the most important. Only after you have tested the deployment process hundreds of times on many environments can you eliminate the deployment script as a source of error.

Every environment is different in some way. If nothing else, it will have a unique IP address, but often there are other differences: operating system and middleware configuration settings, the location of databases and external services, and other configuration information that needs to be set at deployment time.

3. Smoke-Test your deployments

When you deploy your application, you should have an automated script that does a smoke test to make sure that it is up and running. This could be as simple as launching the application and checking to make sure that the main screen comes up with the expected content. Your smoke test should also check that any services your application depends on are up and running—such as a database, messaging bus, or external service.

The smoke test, or deployment test, is probably the most important test to write once you have a unit test suite up and running—indeed, it’s arguably even more important. It gives you the confidence that your application actually runs. If it doesn’t run, your smoke test should be able to give you some basic diagnostics as to whether your application is down because something it depends on is not working.

4. Deploy into a copy of the production

The other main problem many teams experience going live is that their production environment is significantly different from their testing and development environments. To get a good level of confidence that going live will actually work, you need to do your testing and continuous integration on environments that are as similar as possible to your production environment.

Ideally, if your production environment is simple or you have a sufficiently large budget, you can have exact copies of production to run your manual and automated tests on. Making sure that your environments are the same requires a certain amount of discipline to apply good configuration management practices.

5. Each change should propagate through the pipeline instantly

Before continuous integration was introduced, many projects ran various parts of their process off a schedule—for example, builds might run hourly, acceptance tests nightly, and capacity tests over the weekend. The deployment pipeline takes a different approach: The first stage should be triggered upon every check-in, and each stage should trigger the next one immediately upon successful completion.

In this example, somebody checks a change into version control, creating version 1. This, in turn, triggers the first stage in the pipeline (build and unit tests). This passes, and triggers the second stage: the automated acceptance tests. Somebody then checks in another change, creating version 2. This triggers the build and unit tests again. However, even though these have passed, they cannot trigger a new instance of the automated acceptance tests, since they are already running. In the meantime, two more check-ins have occurred in quick succession. However, the CI system should not attempt to build both of them—if it followed that rule, and developers continued to check in at the same rate, the builds would get further and further behind what the developers are currently doing.

6. If any part of the pipeline fails, stop the line

As we said in the “Implementing Continuous Integration” section, the most important step in achieving the goals of this book—rapid, repeatable, reliable releases—is for your team to accept that every time they check code into version control, it will successfully build and pass every test. This applies to the entire deployment pipeline. If a deployment to an environment fails, the whole team owns that failure. They should stop and fix it before doing anything else.

⇒ An Overview of Build Tools

Automated build tools have been a part of software development for a very long time. Many people will remember Make and its many variants that were the standard build tools used for many years. All build tools have a common core: They allow you to model a dependency network. When you run your tool, it will calculate how to reach the goal you specify by executing tasks in the correct order, running each task that your goal depends on exactly once. For example, say you want to run your tests. In order to do this, it’s necessary to compile your code and your tests, and set up your test data.

Your build tool will work out that it needs to perform every task in the dependency network. It can start with either init or setting up test data, since these tasks are independent. Once it has done init, it can then compile the source or the tests—but it must do both, and set up the test data, before the tests can be run.

One small point worth noting is that a task has two essential features: the thing it does and the other things it depends on. These two features are modeled in every build tool.

However, there is one area in which build tools differ: whether they are taskoriented or product-oriented. Task-oriented build tools (for example, Ant, NAnt, and MsBuild) describe the dependency network in terms of a set of tasks. A product-oriented tool, such as Make, describes things in terms of the products they generate, such as an executable.

A build tool must ensure that for a given goal, each prerequisite must be executed exactly once. If a prerequisite is missed, the result of the build process will be bad. If a prerequisite is executed more than once, the best case scenario is that the build will take longer (if the prerequisite is idempotent), and the worst case is that the result of the build process is again bad.

1. Make

Make and its variants are still going strong in the world of systems development. It is a powerful product-oriented build tool capable of tracking dependencies within a build and building only those components that are affected by a particular change. This is essential in optimizing the performance of a development team when compile time is a significant cost in the development cycle.

Unfortunately, Make has a number of drawbacks. As applications become more complex and the number of dependencies between their components increases, the complexity of the rules built into Make means that they become hard to debug.

To tame some of this complexity, a common convention adopted by teams working on large codebases is to create a Makefile for each directory, and have a top-level Makefile that recursively runs the Makefiles in each subdirectory.

2. Ant

With the emergence of Java developers started to do more cross-platform development. The limitations inherent in Make became more painful. In response, the Java community experimented with several solutions, at first porting Make itself to Java. At the same time, XML was coming to prominence as a convenient way to build structured documents. These two approaches converged and resulted in the Apache Ant build tool. Fully cross-platform, Ant includes a set of tasks written in Java to perform common operations such as compilation and filesystem manipulation. Ant can be easily extended with new tasks written in Java. Ant quickly became the de facto standard build tool for Java projects. It is now widely supported by IDEs and other tools. Ant is a task-oriented build tool. The runtime components of Ant are written in Java, but the Ant scripts are an external DSL written in XML. This combination gives Ant powerful cross-platform capabilities. It is also an extremely flexible and powerful system, with Ant tasks for most things you could want to do.

3. NAnt and MSBuild

When Microsoft first introduced the .NET framework, it had many features in common with the Java language and environment. Java developers who worked on this new platform quickly ported some of their favorite open source Java tools. So instead of JUnit and JMock we have NUnit and NMock—and, rather predictably, NAnt. NAnt uses essentially the same syntax as Ant, with only a few differences. Microsoft later introduced their own minor variation on NAnt and called it MSBuild. It is a direct descendant of Ant and NAnt, and will be familiar to anyone who has used those tools. However, it is more tightly integrated into Visual Studio, understanding how to build Visual Studio solutions and projects and how to manage dependencies (as a result, NAnt scripts often call out to MSBuild to do compilation).

4. Maven

For a time, Ant was ubiquitous in the Java community—but innovation did not stop there. Maven attempts to remove the large amount of boilerplate found in Ant files by having a more complex domain that makes many assumptions about the way your Java project is laid out. This principle of favoring convention over configuration means that, so long as your project conforms to the structure dictated by Maven, it will perform almost any build, deploy, test, and release task you can imagine with a single command, without having to write more than a few lines of XML. That includes creating a website for your project which hosts your application's Javadoc by default.

The other important feature of Maven is its support for automated management of Java libraries and dependencies between projects, a problem that is one of the pain points in large Java projects. Maven also supports a complex but rigid software partitioning scheme that allows you to decompose complex solutions into smaller components.

The problem with Maven is threefold. First of all, if your project doesn't conform to Maven's assumptions about structure and lifecycle, it can be extremely hard (or even impossible) to make Maven do what you want.

Maven's second problem is that it also uses an external DSL written in XML, which means that in order to extend it, you need to write code. While writing a Maven plugin is not inordinately complex, it is not something you can just knock out in a few minutes; you'll need to learn about Mojos, plugin descriptors, and whatever inversion-of-control framework Maven is using.

The third problem with Maven is that, in its default configuration, it is selfupdating. Maven's core is very small, and in order to make itself functional, it downloads its own plugins from the Internet. Maven will attempt to upgrade itself every time it is run and, as a result of an upgrade or downgrade of one of its plugins, it can fail unpredictably.

5. Rake

Ant and its brethren are external domain-specific languages (DSLs) for building software. However, their choice of XML to represent these languages made them hard to create, read, maintain, and extend. The dominant Ruby build tool, Rake, came about as an experiment to see if Make's functionality could be easily reproduced by creating an internal DSL in Ruby. The answer was "yes," and Rake was born. Rake is a product-oriented tool similar to Make, but it can also be used as a task-oriented tool.

Like Make, Rake has no understanding of anything except tasks and dependencies. However, since Rake scripts are plain Ruby, you can use Ruby's API to carry out whatever tasks you want.

6. Buildr

The simplicity and power of Rake makes a compelling case that build scripts should be written in a real programming language. The new generation of build tools, such as Buildr, Gradle, and Gantt, have taken this approach. They all feature internal DSLs for building software. However, they attempt to make the more complex challenges of dependency management and multiproject builds just as easy.

Buildr is built on top of Rake, so everything you can do in Rake you can continue to do in Buildr. However, Buildr is also a drop-in replacement for Maven—it uses the same conventions that Maven does, including filesystem layout, artifact specifications, and repositories.

7. Psake

Windows users need not miss out on the new wave of internal DSL build tools. Pronounced “saké,” Psake is an internal DSL written in PowerShell, which provides task-oriented dependency networks.

⇒ Principles and practices of build and deployment scripting

Principles and Practices of Build and Deployment Scripting involves the systematic and efficient automation of the software development lifecycle, focusing on the construction and deployment phases. This process revolves around the creation and execution of scripts to automate the building, testing, and deployment of software applications. By adhering to best practices, such as version control, continuous integration, and continuous deployment, organizations can streamline the development process, enhance collaboration, and ensure consistent, reliable, and timely software delivery.

1. Create a script for each stage in your deployment pipeline

We are big fans of domain-driven design, and apply these techniques in the design of any software that we create. This is no different when we design our build scripts. That is perhaps a bit of a grandiose way of saying that we want the structure of our build scripts to clearly represent the processes that they are implementing. Taking this approach ensures that our scripts have a well-defined structure that helps us to keep them clean during maintenance and minimizes dependencies between components of our build and deployment system. Luckily, the deployment pipeline provides an excellent organizing principle for dividing up responsibilities between build scripts.

When you first start your project, it makes sense to have a single script containing every operation that will be performed in the course of executing the deployment pipeline, with dummy targets for steps that are not yet automated. However, once your script gets sufficiently long, you can divide it up into separate scripts for each stage in your pipeline. Thus you will have a commit script containing all the targets required to compile your application, package it, run the commit test suite, and perform static analysis of the code.

2. Use an appropriate technology to deploy your application

In a typical deployment pipeline, most stages that follow a successful commit stage, such as the automated acceptance test stage and user acceptance test stage, depend upon the application being deployed to a production-like environment.
It is vital that this deployment is automated too. However, you should use the right tool for the job when automating deployment, not a general-purpose scripting language.
Most importantly, deployment of your application will be done both by developers (on their local machines, if nowhere else) and by testers and operations staff. Thus, the decision on how to deploy your application needs to involve all of these people. It also needs to happen towards the start of the project.

3. Use the same scripts to deploy to every environment

As described in the “Deployment Pipeline Practices” section, it is essential to use the same process to deploy to every environment in which your application runs to ensure that the build and deployment process is tested effectively. That means using the same scripts to deploy to each environment and representing the differences between environments—such as service URIs and IP addresses.
It is essential that both build and deployment scripts work on developers’ machines as well as on production-like environments, and that they are used to perform all build and deployment activities by developers.
If your application depends on other components developed inhouse, you’ll want to make sure it’s easy to get the right versions—meaning ones that are known to work reliably together—onto developer machines. This is one area where tools like Maven and Ivy come in very handy.
If your application is complex in terms of its deployment architecture, you will have to make some simplifications to get it working on developer machines.

4. Use your operating system’s packaging tools

We use the term “binaries” throughout this book as a catch-all term for the objects that you put on your target environments as part of your application’s deployment process. Most of the time, this is a bunch of files created by your build process, any libraries your application requires, and perhaps another set of static files checked into version control.

However, deploying a bunch of files that need to be distributed across the filesystem is very inefficient and makes maintenance extremely painful. This is why packaging systems were invented. If you are targeting a single operating system, or a small set of related operating systems, we strongly recommend using that OS's packaging technology to bundle up everything that needs to be deployed.

Whenever your deployments involve sprinkling files across the filesystem or adding keys to the registry, use a packaging system to do it. This has many advantages. Not only does it become very simple to maintain your application, but you can also then piggyback your deployment process onto environment management tools.

5. Ensure the deployment process is idempotent

Your deployment process should always leave the target environment in the same (correct) state, regardless of the state it finds it in when starting a deployment.

The simplest way to achieve this is to start with a known-good baseline environment, provisioned either automatically or through virtualization. This environment should include all the appropriate middleware and anything else your application requires to work. Your deployment process can then fetch the version of the application you specify and deploy it to this environment, using the appropriate deployment tools for your middleware.

If your configuration management procedures are not sufficiently good to achieve this, the next best step is to validate the assumptions your deployment process makes about the environment, and fail the deployment if they are not met.

6. Evolve your deployment system incrementally

Everyone can see the appeal of a fully automated deployment process: "Release your software at the push of a button." When you see a large enterprise system that is deployed this way, it looks like magic. The problem with magic is that it can look dauntingly complex from the outside. In fact, if you examine one of our deployment systems, it is merely a collection of very simple, incremental steps that—over time—create a sophisticated system.

Our point here is that you don’t have to have completed all of the steps to get value from your work. The first time you write a script to deploy the application in a local development environment and share it with the team, you have saved lots of work of individual developers.

⇒ Commit stage principles and practices

The commit stage begins with a change to the state of the project—that is, a commit to the version control system. It ends with either a report of failure or, if successful, a collection of binary artifacts and deployable assemblies to be used in subsequent test and release stages, as well as reports on the state of the application. Ideally, a commit stage should take less than five minutes to run, and certainly no more than ten.
The commit stage represents, in more ways than one, the entrance into the deployment pipeline. Not only is it the point at which a new release candidate is created; it is also where many teams start when they begin to implement a deployment pipeline.
The principal goal of the commit stage is to either create deployable artifacts, or fail fast and notify the team of the reason for the failure.

1. Provide fast, useful feedback

Failures in the commit tests can usually be attributed to one of the following three causes. Either a syntax error has been introduced into the code, caught by compilation in compiled languages; or a semantic error has been introduced into the application, causing one or more tests to fail; or there is a problem with the configuration of the application or its environment (including the operating system).
Whatever the problem, in case of failures, the commit stage should notify the developers as soon as the commit tests are complete and provide a concise summary of the reasons for the failures, such as a list of failed tests, the compile errors, or any other error conditions.
Errors are easiest to fix if they are detected early, close to the point where they were introduced. This is not only because they are fresh in the minds of those who introduced them to the system, but also because the mechanics of discovering the cause of the error are simpler.

2. What should break the commit stage

Traditionally, the commit stage is designed to fail in one of the circumstances listed above: Compilation fails, tests break, or there is an environmental problem. Otherwise, the commit stage succeeds, reporting that everything is OK.

A strong argument can be made that the binary constraint we impose upon the commit stage—either success or a failure—is too limiting. It should be possible to provide richer information, such as a set of graphs representing code coverage and other metrics, upon completion of a commit stage run.

We could, for example, fail the commit stage if the unit test coverage drops below 60%, and have it pass but with the status of amber, not green, if it goes below 80%.

3. Tend the commit stage carefully

The commit stage will include both build scripts and scripts to run unit tests, static analysis tools, and so forth. These scripts need to be maintained carefully and treated with the same level of respect as you would treat any other part of your application.

Like any other software system, when build scripts are poorly designed and maintained, the effort to keep them working grows in what seems like an exponential manner.

A poor build system not only draws valuable and expensive development effort away from the important job of creating the business behavior of your application; it also slows down anyone still trying to implement that business behavior.

Constantly work to improve the quality, design, and performance of the scripts in your commit stage as they evolve. An efficient, fast, reliable commit stage is a key enabler of productivity for any development team, so a small investment in time and thought to get it working well is nearly always repaid very quickly.

4. Give developers ownership

At some organizations, there are teams of specialists who are experts at the creation of effective, modular build pipelines and the management of the environments in which they run. We have both worked in this role. However, we consider it a failure if we get to the point where only those specialists can maintain the CI system. It is vital that the delivery team have a sense of ownership for the commit stage.

It is intimately tied to their work and their productivity. If you impose any barriers between the developers and their ability to get changes made quickly and effectively, you will slow their progress and store trouble for later.

5. Use a build master for very large teams

In small and colocated teams of up to twenty or thirty individuals, selforganization can work very well. If the build is broken, in a team this size it is usually easy enough to locate the person or persons responsible and either remind them of the fact if they are not working on it, or offer to help if they are.

In larger or more widely spread teams, this isn't always easy. Under these circumstances it is useful to have someone to play the role of a "build master." Their job is to oversee and direct the maintenance of the build, but also to encourage and enforce build discipline. If a build breaks, the build master notices and gently—or not gently if it has been a while—reminds the culprit of their responsibility to the team to fix the build quickly or back out their changes.

Another situation where we have found this role useful is in teams new to continuous integration. In such teams, build discipline is not yet ingrained, so reminders are needed to keep things on track.

⇒ The result of commit stage

The commit stage, like every stage in the deployment pipeline, has both inputs and outputs. The inputs are source code, and the outputs are binaries and reports. The reports produced include the test results, which are essential to work out what went wrong if the tests fail, and reports from analysis of the codebase.

Analysis reports can include things like test coverage, cyclomatic complexity, cut and paste analysis, afferent and efferent coupling, and any other useful metrics that help establish the health of the codebase. The binaries generated by the commit stage are precisely the same ones that will be reused throughout the pipeline, and potentially released to users.

The artifact repository

The outputs of the commit stage, your reports and binaries, need to be stored somewhere for reuse in the later stages of your pipeline, and for your team to be able to get hold of them. The obvious place might appear to be your version control system. There are several reasons why this is not the right thing to do, apart from the incidental facts that in this way you're likely to work through disk space fast, and that some version control systems won't support such behavior.

- The artifact repository is an unusual kind of version control system, in that it only needs to keep some versions. Once a release candidate has failed some stage in the deployment pipeline, we are no longer interested in it. So we can, if we wish, purge the binaries and reports from the artifact repository.
- It is essential to be able to trace back from your released software to the revisions in version control that were used to create it. In order to do this, an instance of a pipeline should be correlated with the revisions in your version control system that triggered it.
- One of the acceptance criteria for a good configuration management strategy is that the binary creation process should be repeatable. That is, if I delete the binaries and then rerun the commit stage from the same revision that originally triggered it, I should get exactly the same binaries again.

Most modern continuous integration servers provide an artifact repository, including settings which allow unwanted artifacts to be purged after some length of time.

⇒ Commit test suite principles and practices

There are some important principles and practices governing the design of a commit test suite. The vast majority of your commit tests should be comprised of unit tests, and it is these that we focus on in this section. The most important property of unit tests is that they should be very fast to execute. Sometimes we fail the build if the suite isn't sufficiently fast. The second important property is that they should cover a large proportion of the codebase (around 80% is a good rule of thumb), giving you a good level of confidence that when they pass, the application is fairly likely to be working.

1. Avoid the user interface

The user interface is, by definition, the most obvious place where your users will spot bugs. As a result, there is a natural tendency to focus test efforts on it, sometimes at the cost of other types of testing.

For the purposes of commit tests, though, we recommend that you don't test via the UI at all. The difficulty with UI testing is twofold. First, it tends to involve a lot of components or levels of the software under test. This is problematic because it takes effort, and so time, to get all of the pieces ready for the test before executing the test itself. Second, UIs are designed to work at human timescales which, compared to computer timescales, are desperately slow.

If your project or technology choice allows you to avoid both of these issues, perhaps it is worth creating unit-level tests that operate via the UI, but in our experience UI testing is often problematic and usually better handled at the acceptance test stage of the deployment pipeline.

2. Use dependency injection

Dependency injection, or inversion of control, is a design pattern that describes how the relationships between objects should be established from outside the objects rather than from within. Obviously, this advice only applies if you're using an object-oriented language.

If I create a Car class, I could design it so it creates its own Engine whenever I create a new Car. Alternately, I can elect to design the Car so that it forces me to provide it with an Engine when I create it. The latter is dependency injection. This is more flexible because now I can create Cars with different kinds of Engine without changing the Car code. I could even create my Car with a special TestEngine that only pretends to be an Engine while I'm testing the Car. This technique is not only a great route to flexible, modular software, but it also makes it very easy to limit the scope of a test to just the classes that you want to test, not all of their dependent baggage too.

3. Avoid the database

People new to the use of automated testing will often write tests that interact with some layer in their code, store the results in the database, and then confirm that the results were stored. While this has the advantage of being a simple approach to understand, in all other respects it isn’t a very effective approach.

First of all, the tests it produces are dramatically slower to run. The statefulness of the tests can be a handicap when you want to repeat them, or run several similar tests in close succession.

The unit tests that form the bulk of your commit tests should never rely on the database. To achieve this, you should be able to separate the code under test from its storage.

4. Avoid asynchrony in unit tests

Asynchronous behaviors within the scope of a single test case make systems difficult to test. The simplest approach is to avoid the asynchrony by splitting your tests so that one test runs up to the point of the asynchronous break and then a separate test starts.

For example, if your system posts a message and then acts on it, wrap the raw message-sending technology with an interface of your own. Then you can confirm that the call is made as you expect in one test case, perhaps using a simple stub that implements the messaging interface or using mocking as described in the next section.

5. Using test doubles

The ideal unit tests are focused on a small, closely related number of code components, typically a single class or a few closely related classes.

However, in a well-designed system, each class is relatively small in size and achieves its goals through interactions with other classes.

The problem is that, in such a nicely designed modular system, testing an object in the middle of a network of relationships may require lengthy setup in all the surrounding classes.

6. Minimizing state in tests

Ideally, your unit tests should focus on asserting the behavior of your system. A common problem, particularly with relative newcomers to effective test design, is the accretion of state around your tests.

It is easy to envisage a test of almost any form where you input some values to one component of your system and get some results returned. You write the test by organizing the relevant data structures so that you can submit the inputs in the correct form and compare the results with the outputs you expect. In fact, virtually all tests are of this form, to a greater or lesser extent.

It is too easy to fall into the trap of building elaborate, hard to understand, and hard to maintain data structures in order to support your tests.

7. Faking Time

Time can be a problem in automated testing for several reasons. Perhaps your system needs to trigger an end-of-day process at 8 P.M. Perhaps it needs to wait 500 milliseconds before progressing with the next step. Perhaps it needs to do something different on February 29th of a leap year.

Our strategy for any time-based behavior is to abstract our need for time information into a separate class that is under our control. We usually apply dependency injection to inject our wrapper for the system-level time behaviors we use. This way, we can stub or mock the behavior of our Clock class, or whatever suitable abstraction we choose. If we decide, within the scope of a test, that it is a leap year or 500 milliseconds later, it is fully under our control.

8. Brute Force

Developers will always argue for the fastest commit cycle. In reality, though, this need must be balanced with the commit stage’s ability to identify the most common errors that you are likely to introduce. This is an optimization process that can only work through trial and error. Sometimes, it is better to accept a slower commit stage than to spend too much time optimizing the tests for speed or reducing the proportion of bugs they catch.

We generally aim to keep our commit stage at under ten minutes. This is pretty much the upper bound as far as we are concerned. It is longer than the ideal, which is under five minutes. Developers working on large projects may balk at the target of ten minutes, thinking it unachievably low.
There are two tricks you can use to make your commit test suite run faster. First of all, split it up into separate suites and run them in parallel on several machines. Modern CI servers have “build grid” functionality that makes it extremely straightforward to do this. Remember that computing power is cheap and people are expensive. Getting feedback on time is much more valuable than the cost of a few servers. The second trick you can use is to push, as part of your build optimization process, those tests that are both long-running and don’t often fail out into your acceptance test stage. Note, however, that this results in a longer wait to get feedback on whether a set of changes has broken these tests.

Module – 4

⇒ Automated Acceptance Testing

Acceptance tests are a crucial stage in the deployment pipeline: They take delivery teams beyond basic continuous integration. Once you have automated acceptance tests in place, you are testing the business acceptance criteria of your application, that is, validating that it provides users with valuable functionality. Acceptance tests are typically run against every version of your software that passes the commit tests.

An individual acceptance test is intended to verify that the acceptance criteria of a story or requirement have been met. Acceptance criteria come in many different varieties; for one thing, they can be functional or nonfunctional. Nonfunctional acceptance criteria include things like capacity, performance, modifiability, availability, security, usability, and so forth. The key point here is that when the acceptance tests associated with a particular story or requirement pass, they demonstrate that its acceptance criteria have been met, and that it is thus both complete and working.

The acceptance test suite as a whole both verifies that the application delivers the business value expected by the customer and guards against regressions or defects that break preexisting functions of the application.

The focus on acceptance testing as a means of showing that the application meets its acceptance criteria for each requirement has an additional benefit.

Why is automated acceptance testing essential?

There has always been a great deal of controversy around automated acceptance tests. Project managers and customers often think they are too expensive to create and maintain—which indeed, when done badly, they are. Many developers believe that unit test suites created through test-driven development are enough to protect against regressions.

First of all, it is worth pointing out the costs of manual acceptance testing. To prevent defects from being released, acceptance testing of your application needs to be performed every time it is released.
Furthermore, to fulfill its role of catching regression defects, such testing needs to be performed as a phase once development is complete and a release is approaching.

⇒ Implementing acceptance tests

There is more to the implementation of acceptance tests than layering. Acceptance tests involve putting the application in a particular state, performing several actions on it, and verifying the results. Acceptance tests must be written to handle asynchrony and timeouts in order to avoid flakiness.

1. State in acceptance tests

Acceptance tests are intended to simulate user interactions with the system in a manner that will exercise it and prove that it meets its business requirements. When users interact with your system, they will be building up, and relying upon, the information that your system manages. Without such state your acceptance tests are meaningless. But establishing a known-good starting state, the prerequisite of any real test, and then building a test to rely on that state can be difficult.
When we speak of stateful tests we are using a bit of a shorthand. What we mean to imply by the use of the term “stateful” is that in order to test some behavior of the application, the test depends upon the application being in a specific starting state (the “given” clauses of behavior-driven development). Perhaps the application needs an account with specific privileges, or a particular collection of stock items to operate against. Whatever the required starting state, getting the application ready to exhibit the behavior under test is often the most difficult part of writing the test.

2. Process boundaries, encapsulation, and testing

The most straightforward tests, and therefore the tests that should be your model for all acceptance tests, are those that prove requirements of the system without the need for any privileged access to it. Newcomers to automated testing recognize that to make

their code testable they will have to modify their approach to its design, which is true. But often, they expect that they will need to provide many secret back doors to their code to allow results to be confirmed, which is not true. As we have described elsewhere, automated testing does apply a pressure on you to make your code more modular and better encapsulated, but if you are breaking encapsulation to make it testable you are usually missing a better way to achieve the same goal.
In most cases, you should treat a desire to create a piece of code that only exists to allow you to verify the behavior of the application with a great deal of suspicion. Work hard to avoid such privileged access, think hard before you relent, take a hard line with yourself—and don't give in to the easy option until you are absolutely certain that you can't find a better way.

3. Managing asynchrony and timeouts

Testing asynchronous systems presents its own collection of problems. For unit tests, you should avoid any asynchrony within the scope of a test, or indeed across the boundaries of tests. The latter case can cause hard-to-find intermittent test failures. For acceptance testing, depending on the nature of your application, asynchrony may be impossible to avoid. This problem can occur not just with explicitly asynchronous systems but also with any system that uses threads or transactions. In such systems, the call you make may need to wait for another thread or transaction to complete.
The problem here boils down to this: Has the test failed, or are we just waiting for the results to arrive? We have found that the most effective strategy is to build fixtures that isolate the test itself from this problem. The trick is, as far as the test itself is concerned, to make the sequence of events embodying the test appear to be synchronous. This is achieved by isolating the asynchrony behind synchronous calls.
When we are writing unit tests to be run at commit time, we can test each component of our system in isolation, asserting that each interacts with its neighbors appropriately in this little cluster of objects using test doubles. Such tests will not actually touch the filesystem, but will use a test double to simulate the filesystem.

4. Using test doubles

Acceptance testing relies on the ability to execute automated tests in a productionlike environment. However, a vital property of such a test environment is that it is able to successfully support automated testing. Automated acceptance testing is not the same as user acceptance testing. One of the differences is that automated acceptance tests should not run in an environment that includes integration to all external systems. Instead, your acceptance testing should be focused on providing a controllable environment in which the system under test can be run. “Controllable” in this context means that you are able to create the correct initial state for our tests. Integrating with real external systems removes our ability to do this.

You should work to minimize the impact of external dependencies during acceptance testing. However, our objective is to find problems as early as we can, and to achieve this, we aim to integrate our system continuously.

⇒ The Acceptance Test Stage

Once you have a suite of acceptance tests, they need to be run as part of your deployment pipeline. The rule is that the acceptance test suite should be run against every build that passes the commit tests. Here are some practices applicable to running your acceptance tests.

A build that fails the acceptance tests will not be deployable. In the deployment pipeline pattern, only release candidates that have passed this stage are available for deployment to subsequent stages. Later pipeline stages are most commonly treated as a matter of human judgment: If a release candidate fails to pass capacity testing, on most projects someone decides whether the failure is important enough to end the journey of the candidate there and then, or whether to allow it to proceed despite the performance problem. Acceptance testing offers no room for such fudged results. A pass means that the release candidate can progress, a fail means that it never can.

Because of this hard line, the acceptance test gate is an extremely important threshold and must be treated as such if your development process is to continue smoothly. Keeping the complex acceptance tests running will take time from your development team.

Keeping acceptance tests green

Because of the time taken to run an effective acceptance test suite, it often makes sense to run them later in the deployment pipeline. The problem with this is that if the developers are not sitting there waiting for the tests to pass, as they are for the commit tests, so they often ignore acceptance test failures.

This inefficiency is the trade-off we accept for a deployment pipeline that allows us to catch most failures very quickly at the commit test gate, while also maintaining good automated test coverage of our application. Let's address this antipattern quickly: Ultimately it is an issue of discipline, with the whole delivery team responsible for keeping the acceptance tests passing.

It is essential to fix acceptance test breakages as soon as possible, otherwise the suite will deliver no real value. The most important step is to make the failure visible. We have tried various approaches, such as build masters who track down the people whose changes are most likely to have caused the failure, emails to possible culprits, even standing up and shouting, "Who is fixing the acceptance test build?" (this works quite well).

- Identify likely culprit: Determining what may have caused a specific acceptance test failure is not as simple as for a unit test. A unit test will have been triggered by a single check-in by a single developer or a developer pair. If you check something in and the build fails when it was working before, there should be little doubt that it was you who broke it.

Deployment Tests

A good acceptance test is focused on proving that a specific acceptance criterion for a specific story or requirement has been met. The best acceptance tests are atomic—that is, they create their own start conditions and tidy up at their conclusion. These ideal tests minimize their dependency on state and test the application only through publicly accessible channels with no back door access. However, there are some types of test that don't qualify on this basis but that are, nevertheless, very valuable to run at the acceptance test gate.

When we run our acceptance tests, we design the test environment to be as close as reasonably achievable to the expected production environment. If it’s not expensive to do so, they should be identical. Otherwise use virtualization to simulate your production environment as far as possible. The operating system and any middleware you use should certainly be identical to production, and the important process boundaries that we may have simulated or ignored in our development environment will certainly be represented here.

This means that in addition to testing that our acceptance criteria have been met, this is the earliest opportunity for us to confirm that our automated deployment to a production-like environment works successfully and that our deployment strategy works.

As usual, our objective is to fail fast. We want the acceptance test build to fail as quickly as possible if it is going to fail. For this reason, we often treat the deployment tests as a special suite. If they fail, we will fail the acceptance test stage as a whole immediately and won’t wait for the often lengthy acceptance test suite to complete its run.

⇒ Acceptance Test Performance

Since our automated acceptance tests are there to assert that our system delivers the expected value to our users, their performance is not our primary concern. One of the reasons for creating a deployment pipeline in the first place is the fact that acceptance tests usually take too long to run to wait for their results during a commit cycle.

Acceptance tests must assert the behavior of the system. They must do that, as far as possible, from an external user’s viewpoint and not just by testing the behavior of some hidden layer within it. This automatically implies performance penalties even for relatively simple systems. The system and all of its appropriate infrastructure must be deployed, configured, started, and stopped, before we even consider the time it takes to run a single test.

1. Refactoring common tasks

The obvious first step is to look for quick wins by keeping a list of the slowest tests and regularly spending a little time on them to find ways to make them more efficient. This is precisely the same strategy that we advised for managing unit tests.

One step up from this is to look for common patterns, particularly in the test setup. In general, by their nature, acceptance tests are much more stateful than unit tests. Since we recommend that you take an end-to-end approach to acceptance tests and minimize shared state, this implies that each acceptance test should set up its own start conditions. Frequently, specific steps in such test setup are the same across many tests, so it is worth spending some extra time on ensuring that these steps are efficient.
If there is a public API that can be used instead of performing such setup through the UI, then that is ideal.

2. Share expensive resources

We have already described some techniques to achieve a suitable starting state for tests in commit stage testing in earlier chapters. These techniques can be adapted to acceptance testing, but the black box nature of acceptance tests rules some options out.
The straightforward approach to this problem is to create a standard blank instance of the application at the start of the test and discard it at the end. The test is then wholly responsible for populating this instance with any start data it needs. This is simple and very reliable, having the valuable property that each test is starting from a known, completely reproducible starting point. Unfortunately, for most systems that we create it is also very slow, because for anything but the simplest software systems it takes a significant amount of time to clear any state and start the application in the first place.

3. Parallel testing

When the isolation of your acceptance tests is good, another possibility to speed things up presents itself: running the tests in parallel. For multiuser, server-based systems this is an obvious step. If you can divide your tests so that there is no risk of interaction between them, then running the tests in parallel against a single instance of the system will provide a significant decrease in the duration of your acceptance test stage overall.

4. Using compute grids

For systems that are not multiuser, for tests that are expensive in their own right, or for tests where it is important to simulate many concurrent users, the use of compute grids is of enormous benefit. When combined with the use of virtual servers, this approach becomes exceedingly flexible and scalable. At the limit, you could allocate each test its own host, so the acceptance test suite would only take as long as its slowest test.

In practice, more constrained allocation strategies usually make more sense. This advantage has not been lost on some of the vendors in this space. Most modern CI servers provide the facility to manage a grid of test servers for just this purpose.

⇒ Testing Nonfunctional Requirements

Nonfunctional requirements (NFRs) are important because they present a significant delivery risk to software projects. Even when you are clear about what your nonfunctional requirements are, it is very difficult to hit the sweet spot of doing just enough work to ensure that they are met. Many systems fail because they weren’t able to cope with the load applied to them, were not secure, ran too slowly, or, perhaps most common of all, became unmaintainable because of poor code quality. Some projects fail because they go to the other extreme and worry so much about the NFRs that the development process is too slow, or the system becomes so complex and over-engineered that no one can work out how to develop efficiently or appropriately.

Thus, in many ways, the division of requirements into functional and nonfunctional is an artificial one. Nonfunctional requirements such as availability, capacity, security, and maintainability are every bit as important and valuable as functional ones, and they are essential to the functioning of the system. The name is misleading—alternatives such as cross-functional requirements and system characteristics have been suggested—and, in our experience, the way in which they are commonly dealt with rarely works very well.

⇒ Managing nonfunctional requirements

The difficulty with treating nonfunctional requirements of the system differently from functional requirements is that it makes it easy to drop them off the project plan or to pay insufficient attention to their analysis. This may be disastrous because NFRs are a frequent source of project risk. Discovering late in the delivery process that an application is not fit for purpose because of a fundamental security hole or desperately poor performance is all too frequent—and may cause projects to be late or even to get canceled.

In terms of implementation, NFRs are complex because they usually have a very strong influence on the architecture of the system. For example, any system that requires high performance should not involve requests traversing several tiers. Since the architecture of the system is hard to change later on in the delivery process, it is essential to think about nonfunctional requirements at the beginning of the project.

In addition, NFRs tend to interact with one another in an unhelpful manner: Very secure systems often compromise on ease of use; very flexible systems often compromise on performance, and so forth. Our point here is that, while in an ideal world, everyone wants their systems to be highly secure, very high performance, massively flexible, extremely scalable, easy to use, easy to support, and simple to develop and maintain, in reality, every one of these characteristics comes at a cost.

⇒ Analyzing nonfunctional requirements

For a project that is in flight, we sometimes capture NFRs as regular acceptance criteria for functional stories where we don't anticipate that a significant additional effort will be required to meet them. But this can often be an awkward and inefficient way of managing them.

One approach to managing an NFR, such as audibility, is to say something like "All important interactions with the system should be audited," and perhaps create a strategy for adding relevant acceptance criteria to the stories involving the interactions that need to be audited. An alternative approach is to capture requirements from the perspective of an auditor. What would a user in that role like to see? We simply describe the auditor’s requirements for each report they want to see.

The same is true of characteristics like capacity. It makes sense to define your expectations of the system as stories, quantitatively, and specify them in enough detail so you can perform a cost-benefit analysis and thus prioritize them accordingly.

Many projects face the problem of the acceptance criteria of the application being not particularly well understood. They will have apparently well-defined statements like “All user interactions will take less than two seconds to respond” or “The system will process 80,000 transactions per hour.” Such definitions are too general for our needs. Wooly talks about “application performance” which is often used as a shorthand way of describing performance requirements, usability requirements, and many others.

⇒ Measuring Capacity

Measuring capacity involves investigating a broad spectrum of characteristics of an application. Here are some types of measurements that can be performed:

- Scalability testing: How do the response time of an individual request and the number of possible simultaneous users change as we add more servers, services, or threads?
- Longevity testing: This involves running the system for a long time to see if the performance changes over a protracted period of operation. This type of testing can catch memory leaks or stability problems.
- Throughput testing: How many transactions, or messages, or page hits per second can the system handle?
- Load testing: What happens to capacity when the load on the application increases to production-like proportions and beyond? This is perhaps the most common class of capacity testing.

All of these represent interesting and valid measurements of the behavior of the system but can require different approaches. The first two types of testing are fundamentally different from the other two in that they imply relative measurements: How does the performance profile of the system change as we change the attributes of the system? The second group, though, is only useful as absolute measures.

Focused, benchmark-style capacity tests are extremely useful for guarding against specific problems in the code and optimizing code in a specific area. Sometimes, they can be useful by providing information to help with technology selection processes. However, they form only a part of the picture.

How should success and failure be defined for capacity tests?

Much capacity testing that we have seen has definitely been more measurement than testing. Success or failure is often determined by a human analysis of the collected measurements. The drawback of a capacity measurement strategy over a capacity testing strategy is that it can be a lengthy exercise to analyze the results. However, it is an extremely useful property of any capacity test system if it is also able to generate measurements, providing insight into what happened, not just a binary report of failure or success.

⇒ The capacity testing environment

Absolute measurements of the capacity of a system should ideally be carried out in an environment that, as closely as possible, replicates the production environment in which the system will ultimately run.

While it is possible to learn useful information from differently configured environments, unless they are based on measurement, any extrapolation from capacity in the test environment to capacity in the production environment is highly speculative. The behavior of high-performance computer systems is a specialist and complex area. Configuration changes tend to have a nonlinear effect on capacity characteristics.

If capacity or performance is a serious issue for your application, make the investment and create a clone of your production environment for the core parts of your system. Use the same hardware and software specifications, and follow our advice on how to manage configuration to ensure that you are using the same configuration for each environment, including networking, middleware, and operating system configuration.

In the real world, the idea of capacity testing in an exact replica of the production environment isn’t always possible. Sometimes it is not even sensible, for example when the project is small enough, or when the performance of the application is of insufficient concern to warrant the expense of duplicating production hardware.