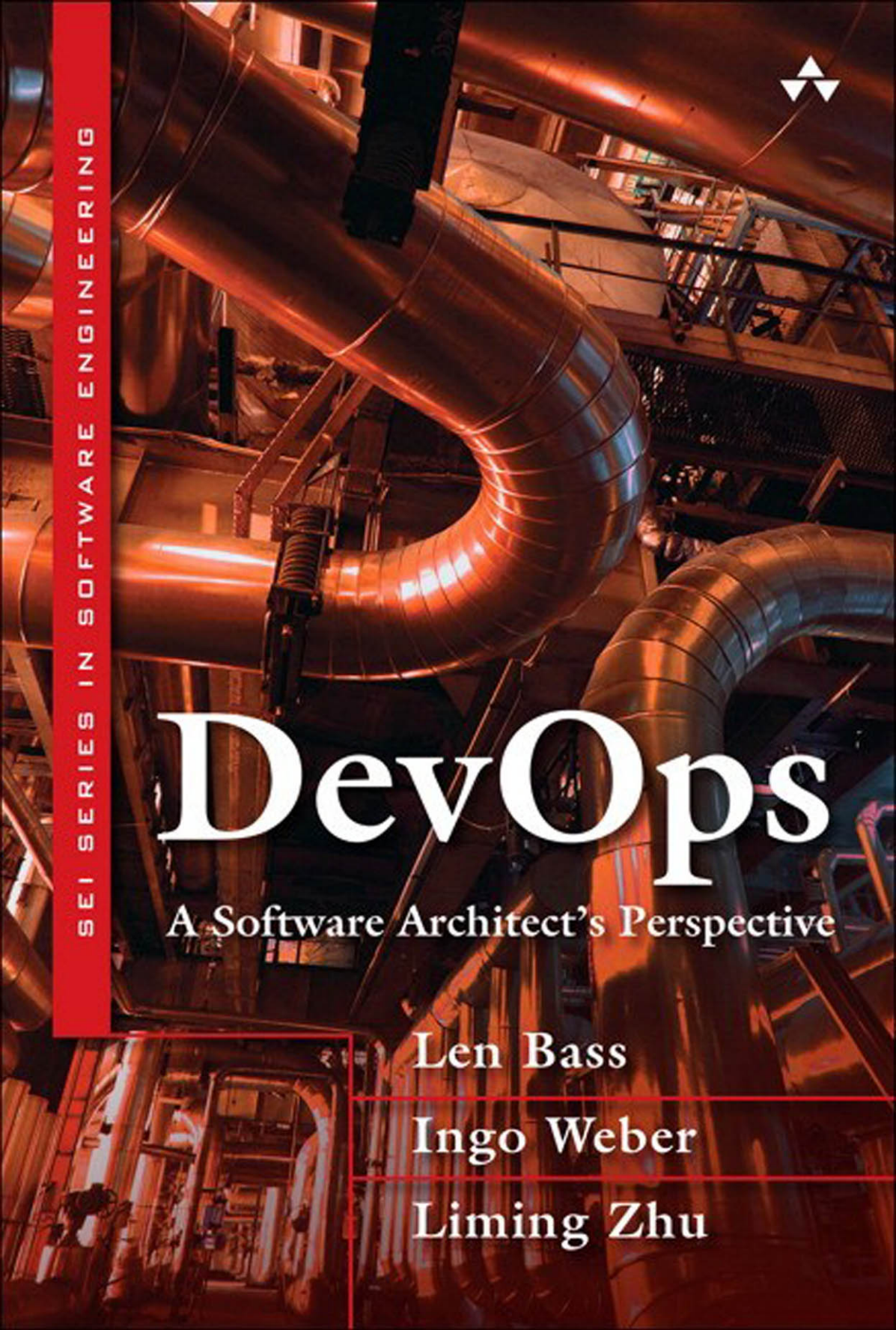
SEI SERIES IN SOFTWARE ENGINEERING
DevOps
A Software Architect's Perspective
Len Bass
Ingo Weber
Liming Zhu

DevOps

usually preceded by a pull. During this pull, changes to the same files (e.g., to the same Java class) are merged automatically. However, this merge can fail, in which case the developer has to resolve any conflicts locally. The resulting changes from an (automatic or semi-manual) merge are committed locally and then pushed to the server.

Almost all version control systems support the creation of new branches. A branch is essentially a copy of a repository (or a portion) and allows independent evolution of two or more streams of work. For example, if part of the development team is working on a set of new features while a previous version is in production and a critical error is discovered in the production system, the version currently in production must be fixed. This can be done by creating a branch for the fix based on the version of the code that was released into production. After the error has been fixed and the fixed version has been released into production, the branch with the fix is typically merged back into the main branch (also called the trunk, mainline, or master branch).

This example is useful in highlighting the need for traceability that we discussed previously. In order to fix the error, the code that was executing needs to be determined (traceability of the code). The error may be due to a problem with the configuration (traceability of the configuration) or with the tool suite used to promote it into production (traceability of the infrastructure).

Although the branch structure is useful and important, two problems exist in using branches.

1. You may have too many branches and lose track of which branch you should be working on for a particular task. Figure 5.3 shows a branch structure with many branches. Determining within this structure on which branch a particular change should be made can be daunting. For this reason, short-lived tasks should not create a new branch.
2. Merging two branches can be difficult. Different branches evolve concurrently, and often developers touch many different parts of the code. For instance, a few developers might make changes to the version currently in production in order to fix bugs, shield the version from newly discovered vulnerabilities, or support urgently required changes. At the same time, several groups of developers might be working toward a new release, each group working on a separate feature branch. Toward the end of the development cycle, you need to merge all feature branches and include the changes resulting from maintenance of the previous release.

An alternative to branching is to have all developers working on the trunk directly. Instead of reintegrating a big branch, a developer deals with integration issues at each commit, which is a simpler solution, but requires more frequent action than using branches. Paul Hammant discussed how Google uses this technique. Development at Google is trunk-based and at full scale: 15,000 developers committing to trunk, with an average of 5,500 submissions per day and 75 million test cases run per day.

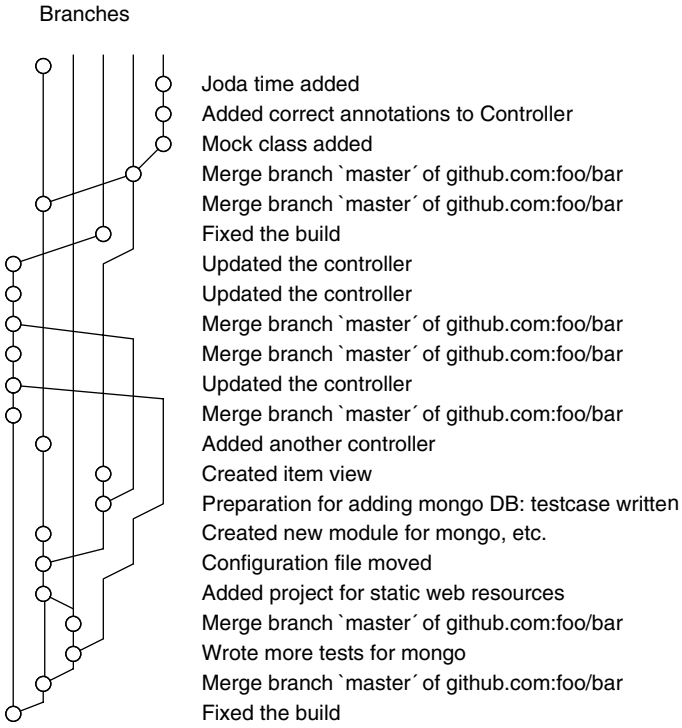

FIGURE 5.3 Git history of a short-lived project with 20 developers showing many merges (Adapted from http://blog.xebia.com/2010/09/20/git-workflow/) [The straight lines represent distinct branches and the diagonal lines represent either forks or merges.]

The problem with doing all of the development on one trunk is that a developer may be working on several different tasks within the same module simultaneously. When one task is finished, the module cannot be committed until the other tasks are completed. To do so would introduce incomplete and untested code for the new feature into the deployment pipeline. Solving this problem is the rationale for feature toggles.

Feature Toggles

A *feature toggle* (also called a *feature flag* or a *feature switch*) is an "if" statement around immature code. Listing 5.1 shows an example. A new feature that is not ready for testing or production is disabled in the source code itself, for example, by setting a global Boolean variable. Once the feature is ready, the toggle is flipped and the respective code is enabled. Common practice places the switches

LISTING 5.1 Pseudo-code sample use of feature toggle

```
If (Feature_Toggle) then
      new code
    else
      old code
    end;
```

for features into configuration, which is the subject of the next section. Feature toggling allows you to continuously deliver new releases, which may include unfinished new features—but these do not impact the application, since they are still switched off. The switch is toggled in production (i.e., the feature is turned on) only once the feature is ready to be released and has successfully passed all necessary tests.

We will discuss another use for feature toggles in Chapter 6.

There are, however, certain dangers in feature toggles. Recall the case of Knight Industries discussed in Chapter 1. The issue that led to a loss of more than (US) $440 million in about 45 minutes included wrong treatment of a feature toggle: The name of a toggle from years earlier was reused in the latest version, but it meant something else in the previous version. Since one of the production servers was still running the old version when the toggle was switched on, (US) $440 million was lost. Lesson 1: Do not reuse toggle names. Lesson 2: Integrate the feature and get rid of the toggle tests as soon as is timely.

When there are many feature toggles, managing them becomes complicated. It would be useful to have a specialized tool or library that knows about all of the feature toggles in the system, is aware of their current state, can change their state, and can eventually remove the feature toggle from your code base.

Configuration Parameters

A configuration parameter is an externally settable variable that changes the behavior of a system. A configuration setting may be: the language you wish to expose to the user, the location of a data file, the thread pool size, the color of the background on the screen, or the feature toggle settings. As you can see, the list of potential configuration parameters is endless.

For the purposes of this book, we are interested in configuration settings that either control the relation of the system to its environment or control behavior related to the stage in the deployment pipeline in which the system is currently run.

The number of configuration parameters should be kept at a manageable level. More configuration parameters usually result in complex connections between them, and the set of compatible settings to several parameters will only be known to experts in the configuration of the software. While flexibility is an

admirable goal, a configuration that is too complex means you are essentially creating a specialized programming language. For instance, the SAP Business Suite had tens of thousands of configuration parameters at one point. While that flexibility allows many companies to use the software in their environments, it also implies that only a team of experts can make the right settings.

Nowadays there are good libraries for most programming languages to provide relatively robust configuration handling. The actions of these libraries include: checking that values have been specified (or default values are available) and are in the right format and range, ensuring that URLs are valid, and even checking whether settings are compatible with multiple configuration options.

You can split configuration parameters into groups according to usage time, for example, whether they are considered at build time, deployment, startup, or runtime. Any important option should be checked before its usage. URLs and other references to external services should be rechecked during startup to make sure they are reachable from the current environment.

One decision to make about configuration parameters is whether the values should be the same in the different steps of the deployment pipeline. If the production system's values are different, you must also decide whether they must be kept confidential. These decisions yield three categories.

1. Values are the same in multiple environments. Feature toggles and performance-related values (e.g., database connection pool size) should be the same in performance testing/UAT/staging and production, but may be different on local developer machines.
2. Values are different depending on the environment. The number of virtual machines (VMs) running in production is likely bigger than that number for the testing environments.
3. Values must be kept confidential. The credentials for accessing the production database or changing the production infrastructure must be kept confidential and only shared with those who need access to them—no sizeable organization can take the risk that a development intern walks away with the customer data.

Keeping values of configuration parameters confidential introduces some complications to the deployment pipeline. The overall goal is to make these values be the current ones in production but keep them confidential. One technique is to give meta-rights to the deployment pipeline and restrict access to the pipeline. When, for instance, a new VM is deployed into production, the deployment pipeline can give it rights to access a key store with the credentials required to operate in production. Another technique is for the deployment pipeline to set the network configuration in a virtual environment for a machine such that it gets to access the production database servers, the production configuration server, and so forth, if the machine is to be part of the production environment. In this case, only the deployment pipeline should have the right to create machines in the production portion of the network.

Testing During Development and Pre-commit Tests

Two types of testing processes occur during development. The first is a design philosophy—test-driven development—and the second is unit testing.

- *Test-driven development*. When following this philosophy, before writing the actual code for a piece of functionality, you develop an automated test for it. Then the functionality is developed, with the goal of fulfilling the test. Once the test passes, the code can be refactored to meet higher-quality standards. A virtue of this practice is that happy or sunny day path tests are created for all of the code.
- *Unit tests*. Unit tests are code-level tests, each of which is testing individual classes and methods. The unit test suite should have exhaustive coverage and run very fast. Typical unit tests check functionality that relies solely on the code in one class and should not involve interactions with the file system or the database. A common practice is to write the code in a way that complicated but required artifacts (such as database connections) form an input to a class—unit tests can provide mock versions of these artifacts, which require less overhead and run faster.

While these tests can be run by the developer at any point, a modern practice is to enforce *pre-commit tests*. These tests are run automatically before a commit is executed. Typically they include a relevant set of unit tests, as well as a few smoke tests. Smoke tests are specific tests that check in a fast (and incomplete) manner that the overall functionality of the service can still be performed. The goal is that any bugs that pass unit tests but break the overall system can be found long before integration testing. Once the pre-commit tests succeed, the commit is executed.

5.5 Build and Integration Testing

Build is the process of creating an executable artifact from input such as source code and configuration. As such, it primarily consists of *compiling* source code (if you are working with compiled languages) and *packaging* all files that are required for execution (e.g., the executables from the code, interpretable files like HTML, JavaScript, etc.). Once the build is complete, a set of automated tests are executed that test whether the integration with other parts of the system uncovers any errors. The unit tests can be repeated here to generate a history available more broadly than to a single developer.

Build Scripts

The build and integration tests are performed by a continuous integration (CI) server. The input to this server should be scripts that can be invoked by a single command. In other words, the only input from an operator or the CI server to

create a build is the command "build"; the rest of the action of the continuous integration server is controlled by the scripts. This practice ensures that the build is repeatable and traceable. Repeatability is achieved because the scripts can be rerun, and traceability is achieved because the scripts can be examined to determine the origin of the various pieces that were integrated together.

Packaging

The goal of building is to create something suitable for deployment. There are several standard methods of packaging the elements of a system for deployment. The appropriate method of packaging will depend on the production environment. Some packaging options are:

- *Runtime-specific packages*, such as Java archives, web application archives, and federal acquisition regulation archives in Java, or .NET assemblies.
- *Operating system packages*. If the application is packaged into software packages of the target OS (such as the Debian or Red Hat package system), a variety of well-proven tools can be used for deployment.
- *VM images* can be created from a template image, to include the changes from the latest revision. Alternatively, a new build can be distributed to existing VMs. These options are discussed next. At any rate, VM images can be instantiated for the various environments as needed. One downside of their use is that they require a compatible hypervisor: VMware images require a VMware hypervisor; Amazon Web Services can only run Amazon Machine Images; and so forth. This implies that the test environments must use the same cloud service. If not, the deployment needs to be adapted accordingly, which means that the deployment to test environments does not necessarily test the deployment scripts for production.
- *Lightweight containers* are a new phenomenon. Like VM images, lightweight containers can contain all libraries and other pieces of software necessary to run the application, while retaining isolation of processes, rights, files, and so forth. In contrast to VM images, lightweight containers do not require a hypervisor on the host machine, nor do they contain the whole operating system, which reduces overhead, load, and size. Lightweight containers can run on local developer machines, on test servers owned by the organization, and on public cloud resources—but they require a compatible operating system. Ideally the same version of the same operating system should be used, because otherwise, as before, the test environments do not fully reflect the production environment.

There are two dominant strategies for applying changes in an application when using VM images or lightweight containers: *heavily baked* versus *lightly baked images*, with a spectrum between the extreme ends. Baking here refers to