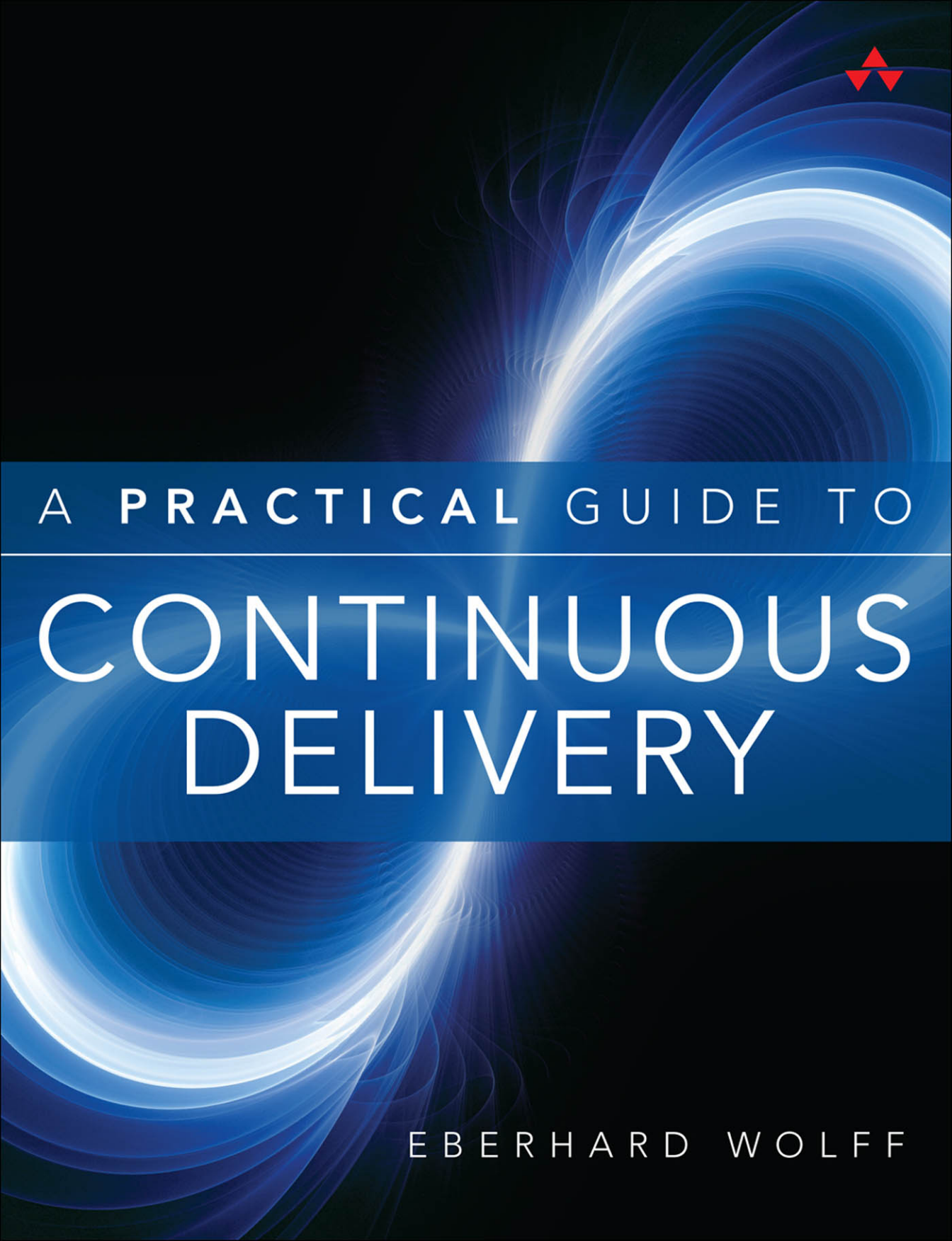
A PRACTICAL GUIDE TO
CONTINUOUS DELIVERY
EBERHARD WOLFF

A Practical Guide to Continuous Delivery

containers that are supposed to run on a physical server. As a basis Kubernetes requires only a simple installation of the operating system—the cluster management is implemented by Kubernetes. Kubernetes is based on the internal Google system for the administration of Linux containers.

- CoreOS[38] is a very lightweight server operating system. Via etcd it supports the cluster-wide distribution of configurations. fleetd allows the deployment of services in the cluster—up to redundant installation, fail-safety, dependencies, and shared deployment on one node. All services have to be deployed as Docker containers, while the operating system itself remains largely unchanged. CoreOS can be used as a foundation for Kubernetes.
- Docker Machine[39] allows for the installation of Docker on different virtualization and Cloud systems (see section 2.5.5). Docker Compose (see next section) can configure a larger number of Docker containers and the links between the containers. Docker Swarm[40] can combine servers that were generated with Docker Machine into a cluster. The Docker Compose configuration of the system can define which parts of the system should be distributed in the cluster and how they should be distributed.

2.5.7 Docker Compose

Docker Compose[41] allows the definition of Docker containers that together deliver a service. YAML is the format of the Docker Compose files.

As an example let us look at the configuration of the example application and the monitoring solution Graphite, which is going to be discussed in more detail in section 8.8.

Listing 2.10 shows the configuration of the example application for the Graphite monitoring. It consists of the different services:

- `carbon` is the server, which saves the monitoring values of the application. The build entry defines that there is a `Dockerfile` in the sub-directory carbon via which the Docker image for the service can be generated. The port entry exposes port 2003 as port 2003 on the host on which the container runs. This port can be used by an application to save values in the database.
- `graphite-web` is the web application that allows the user to analyze the values. It is available under port 8082 on the host, which is redirected to port 80 of the Docker container. The `volumes_from` entry takes care that a hard drive with data from the whisper database of the carbon container is also accessible in this container. Whisper is a library that stores and retrieves the metrics.

- Finally, the `user-registration` container contains the application itself. Via the port of the carbon containers the application delivers the monitoring data—therefore the two containers have a link. Thereby the container `user-registration` can access the container `carbon` under the host name carbon.

Listing 2.10 *Docker Compose configuration for the example application and monitoring*

```
carbon:
  build: carbon
  ports:
   —"2003:2003"
graphite-web:
  build: graphite-web
  ports:
   —"8082:80"
  volumes_from:
  —carbon
user-registration:
  build: user-registration
  ports:
   —"8083:8080"
  links:
 —carbon
```

In contrast to the Vagrant configuration there is no Java container, that is, a container that only contains a Java installation. The reason is that Docker Compose only supports containers that indeed offer a service. Therefore, this basis image is now loaded from the internet.

This creates a system where three containers communicate with each other (Figure 2.7) via network connections or shared file systems.

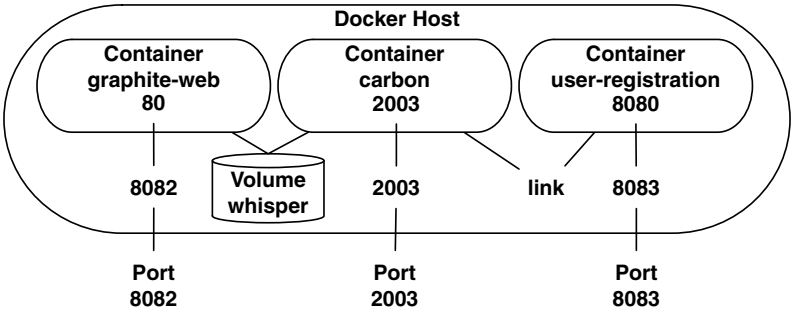

Figure 2.7 *Docker Compose setup*

Try and Experiment

1. First generate a Docker environment, for instance with Docker Machine (see section 2.5.5). At the end the `docker` command should work.
2. Install Docker Compose—see https://docs.docker.com/compose/install/.
3. Clone the example project. With the Git command line tool the required command for this is `git clone https://github.com/ewolff/user-registration-V2.git`.
4. Change into the sub-directory `graphite`.
5. Via `docker-compose` the system is created based on the Docker Compose configuration. The appropriate images are generated.
6. `docker-compose up` will then start the system. Docker Compose works with the same settings as the Docker command line tool. It can also work together with Docker Machine. Therefore, it is transparent whether the system is created on a local VM or somewhere in the Cloud.

   Now the system should be available under the indicated ports.
   Suggestions for additional interesting experiments:

- Recapitulate exactly how the volumes between the `user-registration` and the `graphite-web` container work. To do so have a look at the Dockerfiles. Where exactly is the volume defined?
- Find out how Docker Registry works. Download the example application into a Docker repository and start it from there.
- Extend the setup comprising Docker Machine and Docker Compose by using Docker Swarm (see Endnote 40) to run the application in a cluster. Since Docker Swarm is able to run on a Docker Machine infrastructure and since the Docker Compose configuration can include Docker Swarm settings, this should not be hard to do.
- Based on the Docker Compose configuration, the system can also be run on Mesos/Mesosphere, Kubernetes or CoreOS (see Endnotes 35, 36, 37, and 38). However, this infrastructure differs profoundly from a simple Docker infrastructure so that the required effort is larger.
- Solutions like the Amazon Cloud allow, via the EC2 Container Service,[42] the running of Docker containers. This can also serve as a basis to run a Docker system.

2.6 Immutable Server

Solutions like Chef focus on idempotency (section 2.3): No matter how often an installation script is called, the result should always be the same. Therefore, scripts describe the desired state of the servers. During an installation run the necessary steps to reach this target state are executed.

2.6.1 Disadvantages of Idempotency

However, this approach also has disadvantages:

- Describing the target state of a server can be more complex than defining the required installation steps.
- The description is not without loopholes. When a script does not contain information regarding a resource such as a file or a package, this resource cannot be adapted to a desired state. Further, when this resource is relevant but not in the necessary state, this can result in errors.
- Often servers are permanently updated during run time to the current configuration, but never completely installed anew. In this case, it can happen that it is no longer possible to create a completely new server since the permanent updates have introduced changes that are not performed upon a run of the current installation—for instance, because they have been deleted from the scripts in the meanwhile. This is a dangerous situation since it is not entirely clear anymore what the configuration in production actually looks like and because it is not easy anymore to install a new server with all changes.

Therefore, it is better when a server is always entirely newly created. In this case, it can be ensured that the server is in line with all requirements. Immutable servers are based exactly on this idea. A server is always created in one piece by installing the software on a basis image. When a change in the configuration or installation becomes necessary, a completely new image is created. The installation of a server is never adjusted or modified. This ensures that servers can really be reproduced at any time and that they always have exactly the right configuration.

However, the effort for creating immutable servers appears very high at first glance—in the end the server has to be completely newly installed. But in comparison to the other phases in a Continuous Delivery pipeline this effort can still be rather small—in general the effort for the test phases is the largest.

2.6.2 Immutable Server and Docker

In addition, the optimizations achieved with Docker are especially effective when using immutable servers. For instance, when only a configuration file has been changed and copying this file into the image is the last step in Dockerfile, the creation of the new Docker images will be very fast. Also, it will need hardly any storage space on the hard drive since Docker uses always one image for each step of the installation. Therefore, Docker can reuse the basis images of the previous installations. Only for the last step—copying of the configuration file—is a new image really necessary. However, this image will not be very large and can be rapidly generated.

In this manner, Docker can be combined with an immutable server to guarantee a specific state of a software installation and at the same time to facilitate the installation compared to idempotent installation approaches. Of course, it is also possible to implement an immutable server without Docker or to combine Docker with Chef.

2.7 Infrastructure as Code

The tools described so far will change the character of the infrastructure compared to classical approaches. In fact, infrastructure is turned into code—exactly like the production code of the application. Consequently, the term "infrastructure as code" is generally used. The infrastructure is not generated in a complex manual process, but by an automated process. This has a number of benefits:

- Errors occurring during the creation of environments are largely avoided. Each environment is the result of the same software. Therefore, all environments should look the same. This creates additional safety for the delivery in production.
- In addition, it can be guaranteed that test environments are exactly like the production environments—up to the level of firewall rules and network topology. Especially in this area most test environments differ from production environments. When infrastructure as code is consequently used, such differences can be avoided and thus the predictive value of the tests can be increased. Consequently, fewer errors are found at the stage of production.
- When problems with an environment arise during a test or when the environment is altered unintentionally, only a manual repair of the environment can solve the problem. In the case of infrastructure as code the environment can be

simply deleted and replaced by a fresh one. Alternatively, the environment can be repaired with the help of automation.

- Infrastructure can be versioned together with software. This ensures that software and infrastructure fit together. When a certain software version requires a specific infrastructure change, such a mechanism can ensure that this change is introduced.
- Finally, processes for modifying the infrastructure can be established: This allows the review of all infrastructure changes. In addition, this mechanism ensures that all infrastructure changes are documented and traceable.
- In addition, infrastructure as code helps to keep track of the installed software. Each installation and each component can be found in the installation rules. This facilitates an inventory—the installation of each individual component can be found in the configuration. Consequently, this approach can tackle problems for which normally a Configuration Management System is necessary. Necessary updates—for example because of security issues—can also be rolled out centrally.
- Normally, the number of available environments is limited. Besides, the environments have to be managed: New software has to be installed or test data have to be provided. The automation allows the installation of environments without much effort. When, in addition, servers can be provided flexibly thanks to virtualization, an unlimited (in principle) number of environments can be generated. Especially during a test phase this is of course very helpful since numerous releases and problems can thereby be tested in parallel. Ideally, the environments can be selected simply via a portal and are automatically provided. Likewise, it is possible to start environments in times of high load. For instance, for end-of-year business, extra server farms can be installed that can then be deleted afterwards.
- Operations is subject to permanent cost pressure. At the same time, the number of applications and therefore the number of systems is constantly rising—in part also due to the fact that virtualization has lowered the costs of hardware. In order to be able to keep the ever larger and more complex zoo running with the existing operations team, operations has to become much more productive. Automation and infrastructure as code are the required tools to make this possible.
- To really profit from these advantages, changes of the infrastructure should only be introduced as changes of the infrastructure code. To attain this goal in reality can be quite difficult. Therefore, it can make sense to reduce the complexity of the environment.