Appunti

Cose da dire

Discutibile

Esempi

5. Deployability

From the day we arrive on the planet
And blinking, step into the sun
There's more to be seen than can ever be seen
More to do than can ever be done

—*The Lion King*

There comes a day when software, like the rest of us, must leave home and venture out into the world and experience real life. Unlike the rest of us, software typically makes the trip many times, as changes and updates are made. This chapter is about making that transition as orderly and as effective and—most of all—as *rapid* as possible. That is the realm of continuous deployment, which is most enabled by the quality attribute of deployability.

Why has deployability come to take a front-row seat in the world of quality attributes?

In the "bad old days," releases were infrequent—large numbers of changes were bundled into releases and scheduled. A release would contain new features and bug fixes. One release per month, per quarter, or even per year was common. Competitive pressures in many domains—with the charge being led by e-commerce—resulted in a need for much shorter release cycles. In these contexts, releases can occur at any time—possibly hundreds of releases per day—and each can be instigated by a different team within an organization. Being able to release frequently means that bug fixes in particular do not have to wait until the next scheduled release, but rather can be made and released as soon as a bug is discovered and fixed. It also means that new features do not need to be bundled into a release, but can be put into production at any time.

This is not desirable, or even possible, in all domains. If your software exists in a complex ecosystem with many dependencies, it may not be possible to release just one part of it without coordinating that release with the other parts. In addition, many embedded systems, systems in hard-to-access locations, and systems that are not networked would be poor candidates for a continuous deployment mindset.

This chapter focuses on the large and growing numbers of systems for which just-in-time feature releases are a significant competitive advantage, and just-in-time bug fixes are essential to safety or security or continuous operation. Often these systems are microservice and cloud-based, although the techniques here are not limited to those technologies.

5.1 Continuous Deployment

Deployment is a process that starts with coding and ends with real users interacting with the system in a production environment. If this process is fully automated—that is, if there is no human intervention—then it is called continuous deployment. If the process is automated up to the point of placing (portions of) the system into production and human intervention is required (perhaps due to regulations or policies) for this final step, the process is called continuous delivery.

To speed up releases, we need to introduce the concept of a *deployment pipeline*: the sequence of tools and activities that begin when you check your code into a version control system and end when your application has been deployed for users to send it requests. In between those points, a series of tools integrate and automatically test the newly committed code, test the integrated code for functionality, and test the application for concerns such as performance under load, security, and license compliance.

Each stage in the deployment pipeline takes place in an environment established to support isolation of the stage and perform the actions appropriate to that stage. The major environments are as follows:

- Code is developed in a *development environment* for a single module where it is subject to standalone unit tests. Once it passes the tests, and after appropriate review, the code is committed to a version control system that triggers the build activities in the integration environment.
- An *integration environment* builds an executable version of your service. A continuous integration server compiles[1] your new or changed code, along with the latest compatible versions of code for other portions of your service and constructs an executable image for your service.[2] Tests in the integration environment include the unit tests from the various modules (now run against the built system) as well as integration tests designed specifically for the whole system. When the various tests are passed, the built service is promoted to the staging environment.

[1] If you are developing software using an interpreted language such as Python or JavaScript, there is no compilation step.

[2] In this chapter, we use the term “service” to denote any independently deployable unit.

- A *staging environment* tests for various qualities of the total system. These include performance testing, security testing, license conformance checks, and, possibly, user testing. For embedded systems, this is where simulators of the physical environment (feeding synthetic inputs to the system) are brought to bear. An application that passes all staging environment tests—

which may include field testing—is deployed to the production environment, using either a blue/green model or a rolling upgrade (see Section 5.6). In some cases, partial deployments are used for quality control or to test the market response to a proposed change or offering.

- Once in the *production environment*, the service is monitored closely until all parties have some level of confidence in its quality. At that point, it is considered a normal part of the system and receives the same amount of attention as the other parts of the system.

You perform a different set of tests in each environment, expanding the testing scope from unit testing of a single module in the development environment, to functional testing of all the components that make up your service in the integration environment, and ending with broad quality testing in the staging environment and usage monitoring in the production environment.

But not all always goes according to plan. If you find problems after the software is in its production environment, it is often necessary to roll back to a previous version while the defect is being addressed.

Architectural choices affect deployability. For example, by employing the microservice architecture pattern (see Section 5.6), each team responsible for a microservice can make its own technology choices; this removes incompatibility problems that would previously have been discovered at integration time (e.g., incompatible choices of which version of a library to use). Since microservices are independent services, such choices do not cause problems.

Similarly, a continuous deployment mindset forces you to think about the testing infrastructure earlier in the development process. This is necessary because designing for continuous deployment requires continuous automated testing. In addition, the need to be able to roll back or disable features leads to architectural decisions about mechanisms such as feature toggles and backward compatibility of interfaces. These decisions are best taken early on.

The Effect of Virtualization on the Different Environments

Before the widespread use of virtualization technology, the environments that we describe here were physical facilities. In most organizations, the development, integration, and staging environments comprised hardware and software procured and operated by different groups. The development environment might consist of a few desktop computers that the development team repurposed as servers. The integration environment was operated by the test or quality-assurance team, and might consist of some racks, populated with previous-generation equipment from the data center. The staging environment was operated by the operations team and might have hardware similar to that used in production.

A lot of time was spent trying to figure out why a test that passed in one environment failed in another environment. One benefit of environments that employ virtualization is the ability to have *environment parity*, where environments may differ in scale but not in type of hardware or fundamental structure. A variety of provisioning tools support environment parity by allowing every team to easily build a common environment and by ensuring that this common environment mimics the production environment as closely as possible.

Three important ways to measure the quality of the pipeline are as follows:

- *Cycle time* is the pace of progress through the pipeline. Many organizations will deploy to production several or even hundreds of times a day. Such rapid deployment is not possible if human intervention is required. It is also not possible if one team must coordinate with other teams before placing its service in production. Later in this chapter, we will see architectural techniques that allow teams to perform continuous deployment without consulting other teams.
- *Traceability* is the ability to recover all of the artifacts that led to an element having a problem. That includes all the code and dependencies that are included in that element. It also includes the test cases that were run on that element and the tools that were used to produce the element. Errors in tools used in the deployment pipeline can cause problems in production. Typically, traceability information is kept in an *artifact database*. This database will contain code version numbers, version numbers of elements the system depends on (such as libraries), test version numbers, and tool version numbers.
- *Repeatability* is getting the same result when you perform the same action with the same artifacts. This is not as easy as it sounds. For example, suppose your build process fetches the latest version of a library. The next time you execute the build process, a new version of the library may have been released. As another example, suppose one test modifies some values in the database. If the original values are not restored, subsequent tests may not produce the same results.

DevOps

DevOps—a portmanteau of "development" and "operations"—is a concept closely associated with continuous deployment. It is a movement (much like the Agile movement), a description of a set of practices and tools (again, much like the Agile movement), and a marketing formula touted by vendors selling those tools. The goal of DevOps is to shorten time to market (or time to release). The goal is to dramatically shorten the time between a developer making a change to an existing system—implementing a feature or fixing a bug—and the system reaching the hands of end users, as compared with traditional software development practices.

A formal definition of DevOps captures both the frequency of releases and the ability to perform bug fixes on demand:

> DevOps is a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality. [Bass 15]

Implementing DevOps is a process improvement effort. DevOps encompasses not only the cultural and organizational elements of any process improvement effort, but also a strong reliance on tools and architectural design. All environments are different, of course, but the tools and automation we describe are found in the typical tool chains built to support DevOps.

The continuous deployment strategy we describe here is the conceptual heart of DevOps. Automated testing is, in turn, a critically important ingredient of continuous deployment, and the tooling for that often represents the highest technological hurdle for DevOps. Some forms of DevOps include logging and post-deployment monitoring of those logs, for automatic detection of errors back at the "home office," or even monitoring to understand the user experience. This, of course, requires a "phone home" or log delivery capability in the system, which may or may not be possible or allowable in some systems.

DevSecOps is a flavor of DevOps that incorporates approaches for security (for the infrastructure and for the applications it produces) into the entire process. DevSecOps is increasingly popular in aerospace and defense applications, but is also valid in any application area where DevOps is useful and a security breach would be particularly costly. Many IT applications fall in this category.

5.2 Deployability

Deployability refers to a property of software indicating that it may be deployed—that is, allocated to an environment for execution—within a predictable and acceptable amount of time and effort. Moreover, if the new deployment is not meeting its specifications, it may be rolled back, again within a predictable and acceptable amount of time and effort. As the world moves increasingly toward virtualization and cloud infrastructures, and as the scale of deployed software-intensive systems inevitably increases, it is one of the architect's responsibilities to ensure that deployment is done in an efficient and predictable way, minimizing overall system risk.[3]

[3] Certainly the quality attribute of testability (see Chapter 12) plays a critical role in continuous deployment, and the architect can provide critical support for continuous deployment by ensuring that the system is testable, in all the ways just mentioned. However, our concern here is the quality attribute directly related to continuous deployment over and above testability: deployability.

To achieve these goals, an architect needs to consider how an executable is updated on a host platform, and how it is subsequently invoked, measured, monitored, and controlled. Mobile systems in particular present a challenge for deployability in terms of how they are updated because of concerns about bandwidth. Some of the issues involved in deploying software are as follows:

- How does it arrive at its host (i.e., push, where updates deployed are unbidden, or pull, where users or administrators must explicitly request updates)?
- How is it integrated into an existing system? Can this be done while the existing system is executing?
- What is the medium, such as DVD, USB drive, or Internet delivery?
- What is the packaging (e.g., executable, app, plug-in)?
- What is the resulting integration into an existing system?
- What is the efficiency of executing the process?
- What is the controllability of the process?

With all of these concerns, the architect must be able to assess the associated risks. Architects are primarily concerned with the degree to which the architecture supports deployments that are:

- *Granular*. Deployments can be of the whole system or of elements within a system. If the architecture provides options for finer granularity of deployment, then certain risks can be reduced.
- *Controllable*. The architecture should provide the capability to deploy at varying levels of granularity, monitor the operation of the deployed units, and roll back unsuccessful deployments.
- *Efficient*. The architecture should support rapid deployment (and, if needed, rollback) with a reasonable level of effort.

These characteristics will be reflected in the response measures of the general scenario for deployability.

5.3 Deployability General Scenario

Table 5.1 enumerates the elements of the general scenario that characterize deployability.

Table 5.1 *General Scenario for Deployability*

Portion of Scenario	Description	Possible Values
Source	The trigger for the deployment	End user, developer, system administrator, operations personnel, component marketplace, product owner.
Stimulus	What causes the trigger.	A new element is available to be deployed. This is typically a request to replace a software element with a new version (e.g., fix a defect, apply a security patch, upgrade to the latest release of a component or framework, upgrade to the latest version of an internally produced element). New element is approved for incorporation. An existing element/set of elements needs to be rolled back.
Artifacts	What is to be changed	Specific components or modules, the system's platform, its user interface, its environment, or another system with which it interoperates. Thus the artifact might be a single software element, multiple software elements, or the entire system.
Environment	Staging, production (or a specific subset of either)	Full deployment. Subset deployment to a specified portion of: users, VMs, containers, servers, platforms.
Response	What should happen	Incorporate the new components. Deploy the new components. Monitor the new components. Roll back a previous deployment.
Response measure	A measure of cost, time, or process effectiveness for a deployment, or for a series of deployments over time	Cost in terms of: ▪ Number, size, and complexity of affected artifacts ▪ Average/worst-case effort ▪ Elapsed clock or calendar time ▪ Money (direct outlay or opportunity cost) ▪ New defects introduced Extent to which this deployment/rollback affects other functions or quality attributes. Number of failed deployments.

		Repeatability of the process. Traceability of the process. Cycle time of the process.

Figure 5.1 illustrates a concrete deployability scenario: "A new release of an authentication/authorization service (which our product uses) is made available in the component marketplace and the product owner decides to incorporate this version into the release. The new service is tested and deployed to the production environment within 40 hours of elapsed time and no more than 120 person-hours of effort. The deployment introduces no defects and no SLA is violated."

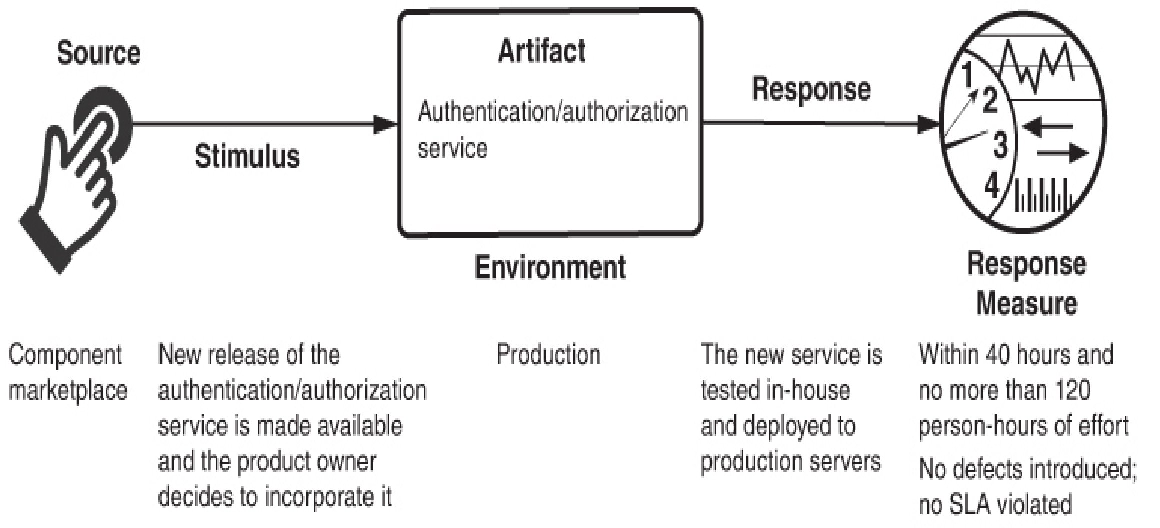

Source
Stimulus
Artifact
Authentication/authorization service
Response
Environment
Response Measure
1
2
3
4
Component marketplace
New release of the authentication/authorization service is made available and the product owner decides to incorporate it
Production
The new service is tested in-house and deployed to production servers
Within 40 hours and no more than 120 person-hours of effort
No defects introduced; no SLA violated

Figure 5.1 *Sample concrete deployability scenario*

5.4 Tactics for Deployability

A deployment is catalyzed by the release of a new software or hardware element. The deployment is successful if these new elements are deployed within acceptable time, cost, and quality constraints. We illustrate this relationship—and hence the goal of deployability tactics—in Figure 5.2.

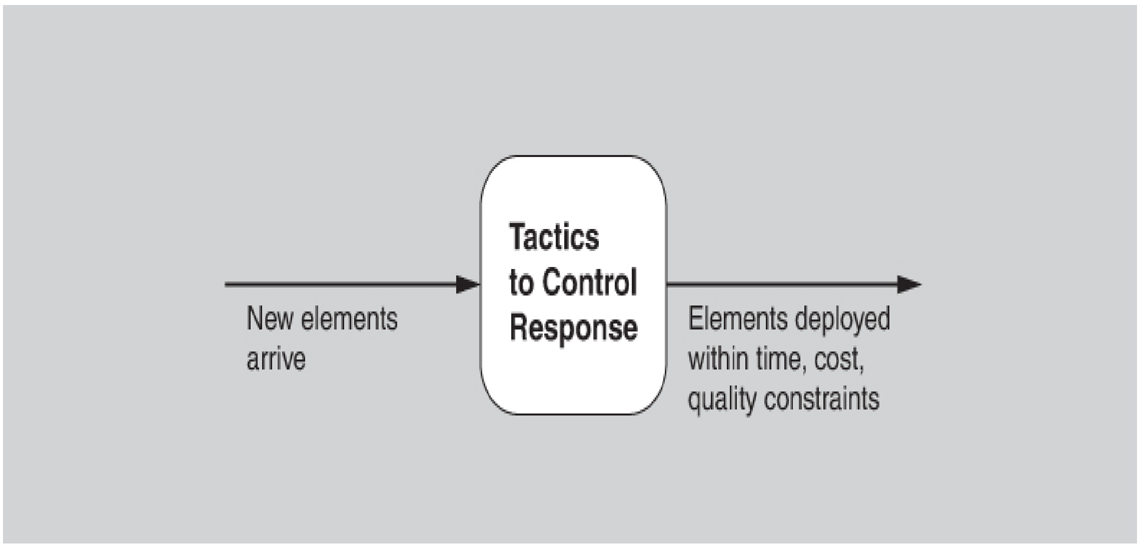

New elements arrive
Tactics to Control Response
Elements deployed within time, cost, quality constraints

Figure 5.2 *Goal of deployability tactics*

The tactics for deployability are shown in Figure 5.3. In many cases, these tactics will be provided, at least in part, by a CI/CD (continuous integration/continuous deployment) infrastructure that you buy rather than build. In such a case, your job as an architect is often one of choosing and assessing (rather than implementing) the right deployability tactics and the right combination of tactics.

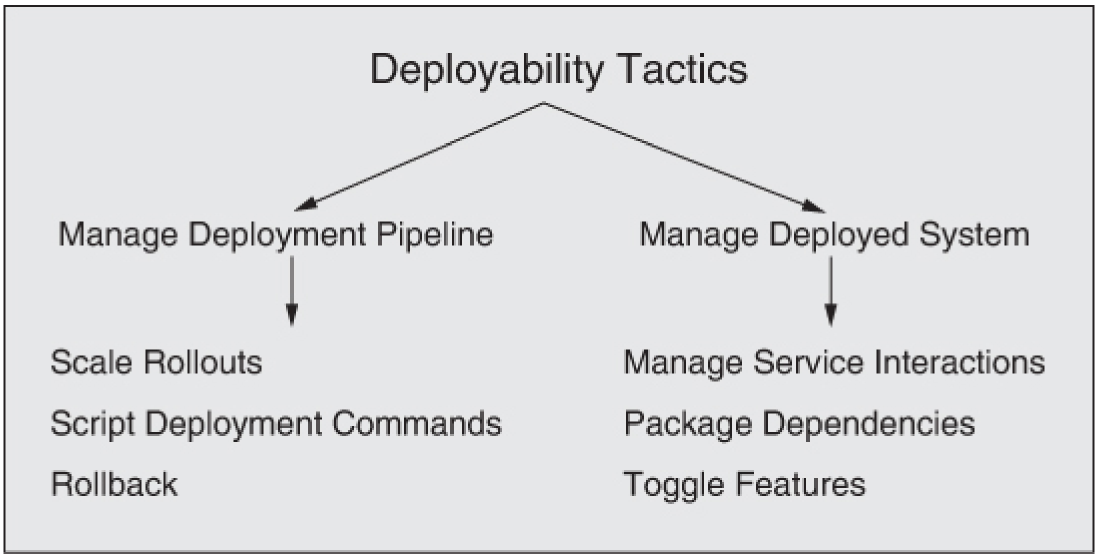


Figure 5.3 *Deployability tactics*

Next, we describe these six deployability tactics in more detail. The first category of deployability tactics focuses on strategies for managing the deployment pipeline and the second category deals with managing the system as it is being deployed and once it has been deployed.

Manage Deployment Pipeline

- *Scale rollouts*. Rather than deploying to the entire user base, scaled rollouts deploy a new version of a service gradually, to controlled subsets of the user population, often with no explicit notification to those users. (The remainder of the user base continues to use the previous version of the service.) By gradually releasing, the effects of new deployments can be monitored and measured and, if necessary, rolled back. This tactic minimizes the potential negative impact of deploying a flawed service. It requires an architectural mechanism (not part of the service being deployed) to route a request from a user to either the new or old service, depending on that user's identity.
- *Roll back*. If it discovered that a deployment has defects or does not meet user expectations, then it can be "rolled back" to its prior state. Since deployments may involve multiple coordinated updates of multiple services and their data, the rollback mechanism must be able to keep track of all of these, or must be able to reverse the consequences of any update made by a deployment, ideally in a fully automated fashion.
- *Script deployment commands*. Deployments are often complex and require many steps to be carried out and orchestrated precisely. For this reason deployment is often scripted. These deployment scripts should be treated like code—documented, reviewed, tested, and version controlled. A scripting engine executes the deployment script automatically, saving time and minimizing opportunities for human error.

Manage Deployed System

- *Manage service interactions*. This tactic accommodates simultaneous deployment and execution of multiple versions of system services. Multiple requests from a client could be directed to either version in any sequence. Having multiple versions of the same service in operation, however, may introduce version incompatibilities. In such cases, the interactions between services need to be mediated so that version incompatibilities are proactively avoided. This tactic is a resource management strategy, obviating the need to completely replicate the resources so as to separately deploy the old and new versions.
- *Package dependencies*. This tactic packages an element together with its dependencies so that they get deployed together and so that the versions of the dependencies are consistent as the element moves from development into production. The dependencies may include libraries, OS versions, and utility containers (e.g., sidecar, service mesh), which we will discuss in Chapter 9.

Three means of packaging dependencies are using containers, pods, or virtual machines; these are discussed in more detail in Chapter 16.

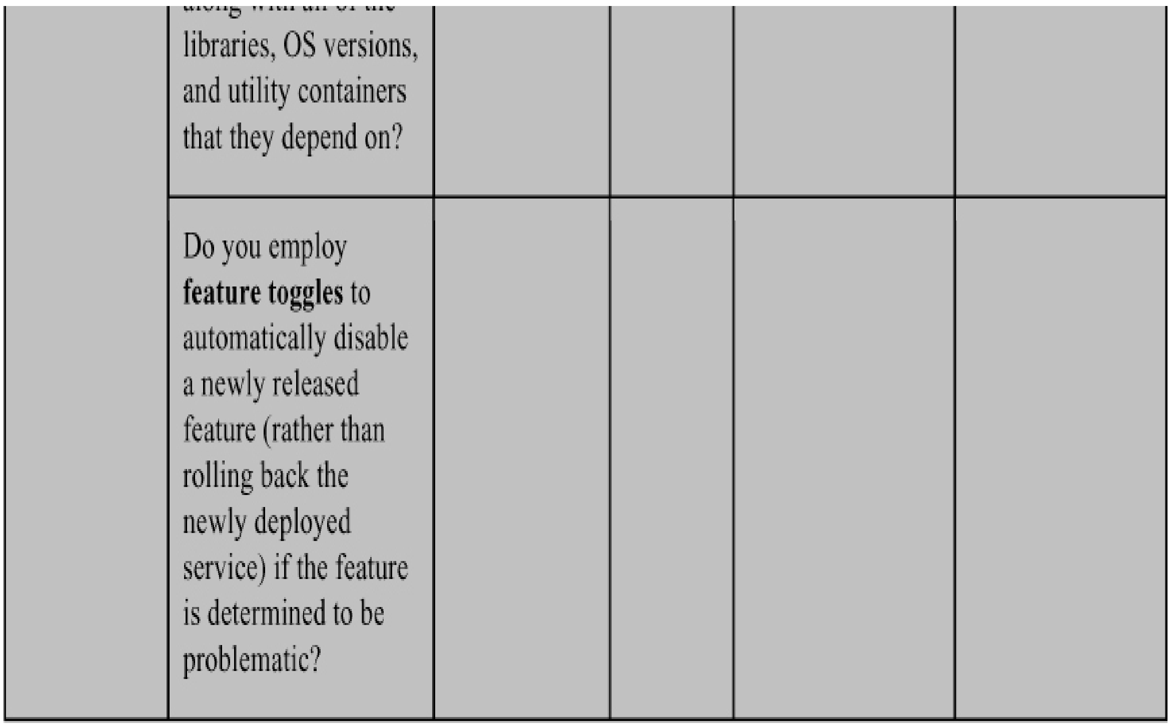
- *Feature toggle*. Even when your code is fully tested, you might encounter issues after deploying new features. For that reason, it is convenient to be able to integrate a "kill switch" (or feature toggle) for new features. The kill switch automatically disables a feature in your system at runtime, without forcing you to initiate a new deployment. This provides the ability to control deployed features without the cost and risk of actually redeploying services.

5.5 Tactics-Based Questionnaire for Deployability

Based on the tactics described in Section 5.4, we can create a set of deployability tactics–inspired questions, as presented in Table 5.2. To gain an overview of the architectural choices made to support deployability, the analyst asks each question and records the answers in the table. The answers to these questions can then be made the focus of subsequent activities: investigation of documentation, analysis of code or other artifacts, reverse engineering of code, and so forth.

Table 5.2 *Tactics-Based Questionnaire for Deployability*

Tactics Groups	Tactics Question	Supported? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
Manage deployment pipeline	Do you **scale rollouts**, rolling out new releases gradually (in contrast to releasing in an all-or-nothing fashion)?				
	Are you able to automatically **roll back** deployed services if you determine that they are not operating in a satisfactory fashion?				
	Do you **script deployment commands** to automatically execute complex sequences of deployment instructions?				
Manage deployed system	Do you **manage service interactions** so that multiple versions of services can be safely deployed simultaneously?				
	Do you **package dependencies** so that services are deployed along with all of the				



	along with all of the libraries, OS versions, and utility containers that they depend on?				
	Do you employ **feature toggles** to automatically disable a newly released feature (rather than rolling back the newly deployed service) if the feature is determined to be problematic?				

5.6 Patterns for Deployability

Patterns for deployability can be organized into two categories. The first category contains patterns for structuring services to be deployed. The second category contains patterns for how to deploy services, which can be parsed into two broad subcategories: all-or-nothing or partial deployment. The two main categories for deployability are not completely independent of each other, because certain deployment patterns depend on certain structural properties of the services.

Patterns for Structuring Services

Microservice Architecture

The microservice architecture pattern structures the system as a collection of independently deployable services that communicate only via messages through service interfaces. There is no other form of interprocess communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. Services are usually stateless, and (because they are developed by a single relatively small team[4]) are relatively small — hence the term *microservice*. Service dependencies are acyclic. An integral part of this pattern is a discovery service so that messages can be appropriately routed.

[4] At Amazon, service teams are constrained in size by the "two pizza rule": The team must be no larger than can be fed by two pizzas.

Benefits:

- Time to market is reduced. Since each service is small and independently deployable, a modification to a service can be deployed without coordinating with teams that own other services. Thus, once a team completes its work on a new version of a service and that version has been tested, it can be deployed immediately.
- Each team can make its own technology choices for its service, as long as the technology choices support message passing. No coordination is needed with respect to library versions or programming languages. This reduces errors due to incompatibilities that arise during integration—and which are a major source of integration errors.
- Services are more easily scaled than coarser-grained applications. Since each service is independent, dynamically adding instances of the service is straightforward. In this way, the supply of services can be more easily matched to the demand.

Tradeoffs:

- Overhead is increased, compared to in-memory communication, because all communication among services occurs via messages across a network. This can be mitigated somewhat by using the service mesh pattern (see Chapter 9), which constrains the deployment of some services to the same host to reduce network traffic. Furthermore, because of the dynamic nature of microservice deployments, discovery services are heavily used, adding to the overhead. Ultimately, those discovery services may become a performance bottleneck.
- Microservices are less suitable for complex transactions because of the difficulty of synchronizing activities across distributed systems.
- The freedom for every team to choose its own technology comes at a cost—the organization must maintain those technologies and the required experience base.
- Intellectual control of the total system may be difficult because of the large number of microservices. This introduces a requirement for catalogs and databases of interfaces to assist in maintaining intellectual control. In addition, the process of properly combining services to achieve a desired outcome may be complex and subtle.
- Designing the services to have appropriate responsibilities and an appropriate level of granularity is a formidable design task.
- To achieve the ability to deploy versions independently, the architecture of the services must be designed to allow for that deployment strategy. Using the manage service interactions tactic described in Section 5.4 can help achieve this goal.

Organizations that have heavily employed the microservice architecture pattern include Google, Netflix, PayPal, Twitter, Facebook, and Amazon. Many other organizations have adopted the microservice architecture pattern as well; books and conferences exist that focus on how an organization can adopt the microservice architecture pattern for its own needs.

Patterns for Complete Replacement of Services

Suppose there are *N* instances of Service A and you wish to replace them with *N* instances of a new version of Service A, leaving no instances of the original version. You wish to do this with no reduction in quality of service to the clients of the service, so there must always be *N* instances of the service running.

Two different patterns for the complete replacement strategy are possible, both of which are realizations of the scale rollouts tactic. We'll cover them both together:

1. *Blue/green*. In a blue/green deployment, *N* new instances of the service would be created and each populated with new Service A (let's call these the green instances). After the *N* instances of new Service A are installed, the DNS server or discovery service would be changed to point to the new version of Service A. Once it is determined that the new instances are working satisfactorily, then, and only then, are the *N* instances of the original Service A removed. Before this cutoff point, if a problem is found in the new version, it is a simple matter of switching back to the original (the blue services) with little or no interruption.
2. *Rolling upgrade*. A rolling upgrade replaces the instances of Service A with instances of the new version of Service A one at a time. (In practice, you can replace more than one instance at a time, but only a small fraction are replaced in any single step.) The steps of the rolling upgrade are as follows:

   a. Allocate resources for a new instance of Service A (e.g., a virtual machine).

   b. Install and register the new version of Service A.

   c. Begin to direct requests to the new version of Service A.

   d. Choose an instance of the old Service A, allow it to complete any active processing, and then destroy that instance.

   e. Repeat the preceding steps until all instances of the old version have been replaced.

Figure 5.4 shows a rolling upgrade process as implemented by Netflix's Asgard tool on Amazon's EC2 cloud platform.

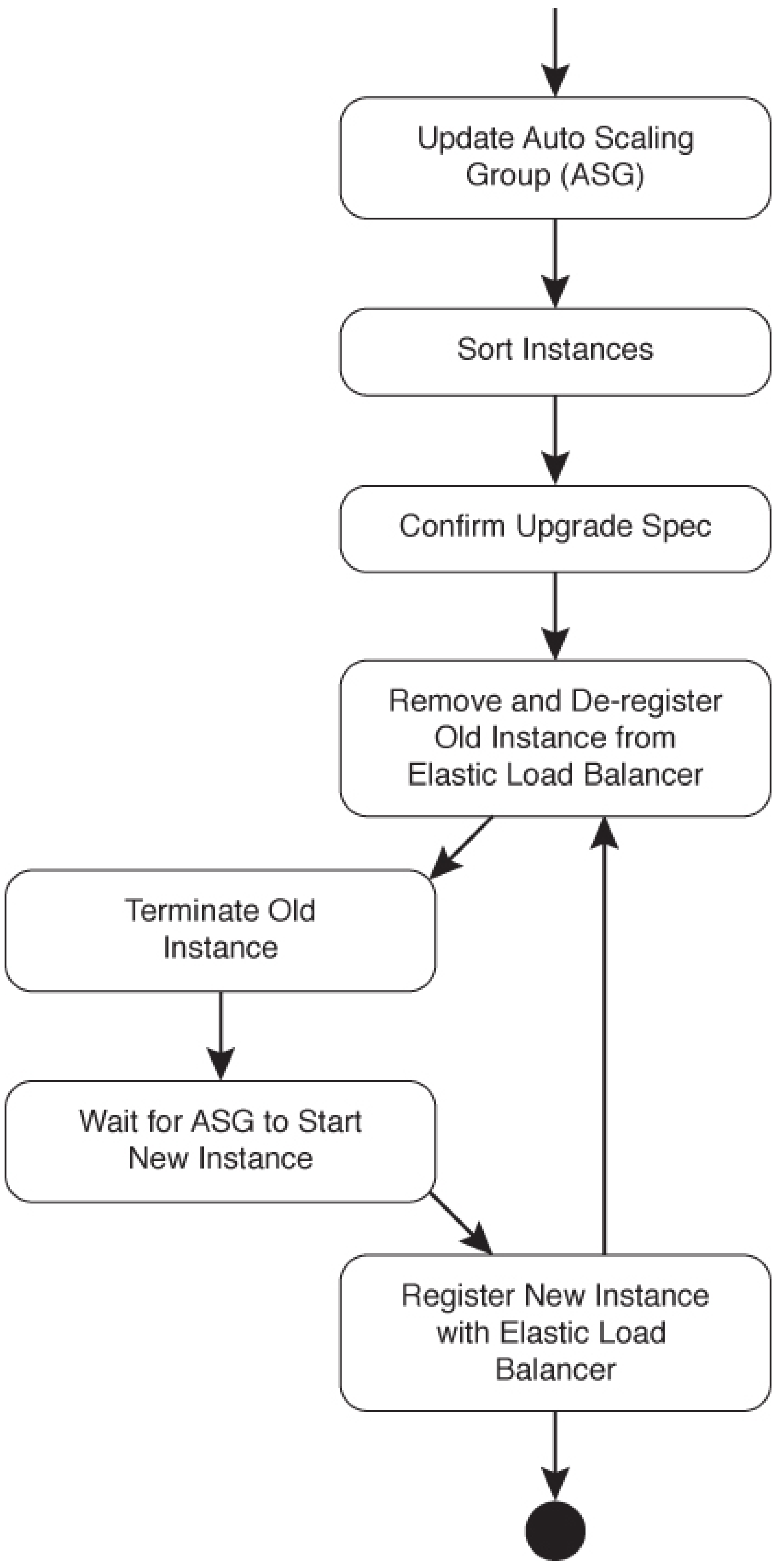

Update Auto Scaling Group (ASG)
Sort Instances
Confirm Upgrade Spec
Remove and De-register Old Instance from Elastic Load Balancer
Terminate Old Instance
Wait for ASG to Start New Instance
Register New Instance with Elastic Load Balancer

Figure 5.4 *A flowchart of the rolling upgrade pattern as implemented by Netflix's Asgard tool*

Benefits:

- The benefit of these patterns is the ability to completely replace deployed versions of services without having to take the system out of service, thus increasing the system's availability.

Tradeoffs:

- The peak resource utilization for a blue/green approach is $2N$ instances, whereas the peak utilization for a rolling upgrade is $N + 1$ instances. In either case, resources to host these instances must be procured. Before the widespread adoption of cloud computing, procurement meant purchase: An organization had to purchase physical computers to perform the upgrade. Most of the time there was no upgrade in progress, so these additional computers largely sat idle. This made the financial tradeoff clear, and rolling upgrade was the standard approach. Now that computing resources can be rented on an as-needed basis, rather than purchased, the financial tradeoff is less compelling but still present.
- Suppose you detect an error in the new Service A when you deploy it. Despite all the testing you did in the development, integration, and staging environments, when your service is deployed to production, there may still be latent errors. If you are using blue/green deployment, by the time you discover an error in the new Service A, all of the original instances may have been deleted and rolling back to the old version could take considerable time. In contrast, a rolling upgrade may allow you to discover an error in the new version of the service while instances of the old version are still available.
- From a client's perspective, if you are using the blue/green deployment model, then at any point in time either the new version or the old version is active, but not both. If you are using the rolling upgrade pattern, both versions are simultaneously active. This introduces the possibility of two types of problems: *temporal inconsistency* and *interface mismatch*.
  - *Temporal inconsistency*. In a sequence of requests by Client C to Service A, some may be served by the old version of the service and some may be served by the new version. If the versions behave differently, this may cause Client C to produce erroneous, or at least inconsistent, results. (This can be prevented by using the manage service interactions tactic.)
  - *Interface mismatch*. If the interface to the new version of Service A is different from the interface to the old version of Service A, then invocations by clients of Service A that have not been updated to reflect the new interface will produce unpredictable results. This can be prevented by extending the interface but not modifying the existing interface, and using the mediator pattern (see Chapter 7) to translate

from the extended interface to an internal interface that produces correct behavior. See Chapter 15 for a fuller discussion.

Patterns for Partial Replacement of Services

Sometimes changing all instances of a service is undesirable. Partial-deployment patterns aim at providing multiple versions of a service simultaneously for different user groups; they are used for purposes such as quality control (canary testing) and marketing tests (A/B testing).

Canary Testing

Before rolling out a new release, it is prudent to test it in the production environment, but with a limited set of users. *Canary testing* is the continuous deployment analog of beta testing.[5] Canary testing designates a small set of users who will test the new release. Sometimes, these testers are so-called power users or preview-stream users from outside your organization who are more likely to exercise code paths and edge cases that typical users may use less frequently. Users may or may not know that they are being used as guinea pigs—er, that is, canaries. Another approach is to use testers from within the organization that is developing the software. For example, Google employees almost never use the release that external users would be using, but instead act as testers for upcoming releases. When the focus of the testing is on determining how well new features are accepted, a variant of canary testing called dark launch is used.

[5] Canary testing is named after the 19th-century practice of bringing canaries into coal mines. Coal mining releases gases that are explosive and poisonous. Because canaries are more sensitive to these gases than humans, coal miners brought canaries into the mines and watched them for signs of reaction to the gases. The canaries acted as early warning devices for the miners, indicating an unsafe environment.

In both cases, the users are designated as canaries and routed to the appropriate version of a service through DNS settings or through discovery-service configuration. After testing is complete, users are all directed to either the new version or the old version, and instances of the deprecated version are destroyed. Rolling upgrade or blue/green deployment could be used to deploy the new version.

Benefits:

- Canary testing allows real users to “bang on” the software in ways that simulated testing cannot. This allows the organization deploying the service to collect “in use” data and perform controlled experiments with relatively low risk.

- Canary testing incurs minimal additional development costs, because the system being tested is on a path to production anyway.
- Canary testing minimizes the number of users who may be exposed to a serious defect in the new system.

Tradeoffs:

- Canary testing requires additional up-front planning and resources, and a strategy for evaluating the results of the tests needs to be formulated.
- If canary testing is aimed at power users, those users have to be identified and the new version routed to them.

A/B Testing

A/B testing is used by marketers to perform an experiment with real users to determine which of several alternatives yields the best business results. A small but meaningful number of users receive a different treatment from the remainder of the users. The difference can be minor, such as a change to the font size or form layout, or it can be more significant. For example, HomeAway (now Vrbo) has used A/B testing to vary the format, content, and look-and-feel of its worldwide websites, tracking which editions produced the most rentals. The "winner" would be kept, the "loser" discarded, and another contender designed and deployed. Another example is a bank offering different promotions to open new accounts. An oft-repeated story is that Google tested 41 different shades of blue to decide which shade to use to report search results.

As in canary testing, DNS servers and discovery-service configurations are set to send client requests to different versions. In A/B testing, the different versions are monitored to see which one provides the best response from a business perspective.

Benefits:

- A/B testing allows marketing and product development teams to run experiments on, and collect data from, real users.
- A/B testing can allow for targeting of users based on an arbitrary set of characteristics.

Tradeoffs:

- A/B testing requires the implementation of alternatives, one of which will be discarded.
- Different classes of users, and their characteristics, need to be identified up front.

Bisogna anche capire oltre al gruppo di utenti anche come decidere qual e' il vincitore (= in base a cosa)

5.7 For Further Reading

Much of the material in this chapter is adapted from *Deployment and Operations for Software Engineers* by Len Bass and John Klein [Bass 19] and from [Kazman 20b].

A general discussion of deployability and architecture in the context of DevOps can be found in [Bass 15].

The tactics for deployability owe much to the work of Martin Fowler and his colleagues, which can be found in [Fowler 10], [Lewis 14], and [Sato 14].

Deployment pipelines are described in much more detail in [Humble 10]

Microservices and the process of migrating to microservices were first described in [Newman 15].

5.8 Discussion Questions

1. Write a set of concrete scenarios for deployability using each of the possible responses in the general scenario.
2. Write a concrete deployability scenario for the software for a car (such as a Tesla).
3. Write a concrete deployability scenario for a smartphone app. Now write one for the server-side infrastructure that communicates with this app.
4. If you needed to display the results of a search operation, would you perform A/B testing or simply use the color that Google has chosen? Why?
5. Referring to the structures described in Chapter 1, which structures would be involved in implementing the package dependencies tactic? Would you use the uses structure? Why or why not? Are there other structures you would need to consider?
6. Referring to the structures described in Chapter 1, which structures would be involved in implementing the manage service interactions tactic? Would you use the uses structure? Why or why not? Are there other structures you would need to consider?
7. Under what circumstances would you prefer to roll forward to a new version of service, rather than to roll back to a prior version? When is roll forward a poor choice?