Chapter 6 – Deployment pipeline

In this Chapter, you will learn about the different stages of the deployment pipeline, the tools used in the different stages, and their associated environments.

The stages are development, integration, and staging. Each stage has an environment with an explicit life cycle that allocates the necessary resources and enforces isolation among the environments. Each life cycle ends with a tear down phase where all of the resources associated with that environment are released.

The chapter ends with a discussion of the blue/green and rolling upgrade models of deployment and how to achieve consistency when performing the upgrade. Finally, you will learn about partial deployments – canary and A/B testing – and the reasons for partial deployments.

Overview of a deployment pipeline

Figure 6.xx shows an abstracted deployment pipeline. You as a developer develop and test code in a development environment. When you are happy with your code, you check it into a version control system and that triggers activity by the continuous integration server. The continuous integration server operates in the integration environment. It compiles your code, the code of other portions of your microservice and constructs an executable process. This process is then tested for functional correctness. In the development environment, your tests were of the single module on which you were working. In the integration environment, the tests are of all of the components that make up your microservice. These tests, however, are limited to functional correctness. Once the constructed process passes the functional tests it is packaged into a virtual machine or a container and promoted to the staging environment. The staging environment tests the quality o the packaged process – how well it handles load, whether security vulnerabilities are correctly handled, license compliance and, potentially, user testing. Passing the staging tests entitles your packaged process to be deployed into production.

If the process through the pipeline into deployment is totally automated, i.e., no human intervention, then this process is called “continual deployment”. If human intervention is required to place the microservice into production – as is required by some regulations – then the process is called “continual development”.

The pace of progress through the pipeline is called its cycle time and many organizations will deploy multiple or dozens of times a day. Such rapid deployment is not possible if human intervention is required.

The description of how your code is committed, built, staged, and deployed into production is an outline of this chapter. We discuss in more detail not only the stages in the deployment pipeline but also version control systems, the concept of an environment, the deployment process, and the types of tests that happen at each stage of the process.

Version Control

How many times have you lost work because of stupid errors such as saving one file on top of another or modifying a file incorrectly and writing it over the old version? In a team setting, the problem gets worse because one team member can overwrite the work of another team member. These are the problems that version control systems are created to avoid.

A version control system manages a repository in which versions of textual artifacts such as code, scripts, and tests are kept. By keeping versions of files, older versions can be recalled in the case of an error. Furthermore, by sharing the repository among multiple users, the files in the repository can be retrieved or modified by any of the users.

Version control systems are based on a "commit/check out" process. That is, committing a file to the version control system creates a copy of that file in the repository and checking out a file makes a local copy for your use.

Collections of files are given a name. A named collection can be "branched". That is, there are logically now two copies of this collection. Physically, the version control system only keeps one copy but the two logical copies can now evolve independently. For example, one branch may represent the files contained in the current production system and another the files being modified to add new features or to fix bugs.

Branches can also be "merged". Asa result of a merger, one of the branches no longer is active although files in it could, of course, be retrieved. A merger requires that differences in the two copies of a single file be resolved. That is, if branch A has some code in a file that is different from the code in that file in branch B, you must choose which code is carried forward in that file. The longer between the branch and the merge, the more the two files will differ and the more difficult the merger becomes. For this reason, best practice suggests merging often. Figure 6.xx shows the branch merge structure of a particular system.

The distinction between centralized and distributed version control systems is worth mentioning at this point. A centralized system such as Subversion requires connectivity with a central server in order to check in or check out files. A distributed system such as Git copies a branch from the central server onto your local machine (a pull) and you check in and check out from that branch. You then copy that branch back to the central server (a push) when you choose. Suppose, for example, you are travelling and without access to the central server. With a centralized system you cannot check in or out, with a distributed system you can. Both styles require connectivity in order to share your changes with others in your team or to have the file safely recorded in the event your laptop fails.

All of the textual artifacts that go into constructing a production system – code, tests, scripts, and so forth, should be kept in a version controlled repository. This not only supports sharing all of these artifacts among team members but also allows you to rebuild the production system for error recovery or diagnostic purposes.

Environments

You should have noticed in the description of the deployment pipeline that environments are an important concept. Separate environments are used for development, for integration, for staging and for production. The purpose of every environment except for production is to provide for testing. The purpose of the production environment is to support end users.

In this section, we discuss environments as a separate concept with its own requirements and life cycle.

Requirements for an environment

What are the requirements for an environment? One key requirement is that each environment is kept logically separated from other environments. This includes address spaces, inputs, modifications to any data base, and other dependencies on the processes being executed in the environment. A second key requirement is that to the extent possible and consistent with the purpose of the environment, each environment should be as realistic as possible. A third key requirement is that the elements that go into promoting your code to the next environment should all be recorded for traceability purposes. The artifact database as the location where this information is recorded.

Figure 6.xx shows the elements of an environment. Each environment has

- a collection of virtual machines or containers.
- a load balancer.
- a source of input.
- a data base.
- A configuration description.
- External services.

We now elaborate on these elements.

A collection of virtual machines or containers.

These contain the modules, microservices or application being executed in this environment. The collection of microservices grows as the application moves through the pipeline. In a development environment, a single module is being tested, in a integration environment, the microservice is being tested, and in a staging environment the microservice plus other microservices is tested.

Load balancer.

Your microservice will be executed under the control of a load balancer. This involves registration and health monitoring. Having a load balancer in the environment from the inception of testing will enable you to catch problems with interactions between your microservice and the load balancer. In a development environment, you may be testing a module that has no requirement for interacting with the load balancer but treating all environments uniformly simplifies the management of the environment. Treating all environments uniformly means including a load balancer in each environment.

Source of input.

The environment is being used for a series of tests. There must be input into the environment. This input comes from a test harness, live users, or some capturing of live users input. Figure 6.xx shows a test harness. There is a source of input – the test driver– and the output is sent back to the test driver to compare to the expected output. Errors are reported by the test driver.

Data base.

The data in the database will depend on the purpose of the environment and we will discuss this in the sections on the specific environments. A key element across all environments, however, is that the database must be restored after each test. The tests must be repeatable and give the same results every time they are run. If not, finding errors becomes very difficult. Since any given test may modify data in the database, restoring the database after each test will ensure that each execution of each test begins from the same state.

Configuration parameters

As we said in Chapter 5, a configuration parameter is a value that is bound at run time, typically at initialization. In the context of environments, configuration parameters are everything that will change depending on the environment. The URL of the database, credentials, and location of external services are some of the items that will vary from environment to environment. We do not advocate placing credentials in a file such as is represented in Figure 6.xx since that may represent a security vulnerability. We discuss the options for the management of configuration parameters in Section xx.xx and the options for managing credentials in Section xx.xx. Our point here is that the system being executed in a given environment will need configuration parameters furnished to it.

External services.

Your microservice may also use external services. These can range from a service that broadcasts the weather to one that performs authorization. The treatment of these external services depends on which environment your microservice is in and whether the external service is read only or read/write. If your microservice is not in the production environment and the external service is read/write then it must be stubbed or mocked. Any writing to an external service from other than the production environment may affect the behavior of that service. In general, no action from other than the production environment should affect the production environment and writing to an external service may have that effect.

Life Cycle of an Environment

Each of the different types of environments that we discuss below has a life cycle. These life cycles differ slightly in the different types of environments and we will discuss the details of each life cycle in the appropriate context. Figure 6.xx summarizes the life cycle for each stage of the deployment pipeline. The life cycles have two important elements in common and we discuss these now.

Build Step

Each life cycle begins with a build step. This build step is triggered by some event and scripted. This allows the build step to be automated. We will discuss the triggering event when we discuss the different types of environments but the build step always has the following activities.

- Create a VM or container loaded with your software. If your environment is hosted locally, this might involve a provisioning tool such as Vagrant. If your environment is hosted in the cloud, your cloud provider has a means for scripting the creation of VMs by specifying their attributes. The attributes include, for each VM, the size of the VM, the number of initial instances of the VM, the autoscaling rules, the security settings, and so forth. If you are using a container as your packaging mechanism, the scripts will be for the container orchestration system.
- Create a load balancer. The load balancer is a separate VM. The build script will create the load balancer and, since it knows the VMs to be balanced from the previous activity, it can register the VMs with the load balancer.
- Create the test harness. You will likely have a standard testing tool. The information necessary to create this testing tool and link it into the environment should be in the triggering script. The test cases should be version controlled and so the test harness knows where to find the latest version of the test cases by querying the version control graph for the test cases.

- Populate the database for the environment is created. The mechanism for doing this will vary depending on the environment but the build script is responsible for populating the database.
- Create environment dependent configuration parameters. These parameters will be items such as the URL of the database and the URLs of the external services being invoked by the application moving through the pipeline.

Teardown

The last step in the life cycle of any environment is to release all of the resources used by the environment. As we mentioned earlier, VM sprawl is the problem of losing track of VMs. This not only increases the costs but has security implications. VMs must be patched when new vulnerabilities are discovered and VMs which have been lost will not get patched and hence will be vulnerable to attacks.

Scripting the teardown step helps to reduce VM sprawl and keep only active VMs on your account. As with the build step, the trigger for teardown will vary depending on the type of environment but having a teardown step makes the clean up process automatic.

Tradeoffs

The automatic creation of an environment based on triggering mechanisms will result in the creation and destruction of many environments. Creating an environment takes time an uses resources. A natural question therefore is “is the automatic creation and deletion of environments worth the effort”?

The purpose of a distinction environment is isolation – keep activities in one phase of your development life cycle from impacting activities by others and activities in other phases. The trade off then is between the errors prevented by isolating all activities into their own environment and the costs – both human and computing resources – of having multiple environments.

Just as version control systems are a reaction to the types of errors associated with file management, so the creation of environments is a reaction to the types of errors associated with interference of one set of activities with another. If everyone were perfect, neither type of system would be necessary. However, such perfection is not often found and protecting a developer from himself or herself or from teammates has proven to be useful.

Now we turn to the specific types of environments, beginning with the development environment.

Development Environment

The development environment is where you will create and test the module you are currently working on. This module may represent a new microservice or it may be the maintenance of an existing microservice. In either case, you will interact with the version control system and an IDE (Integrated Development Environment) Figure 6.xx shows the workflow during the development environment.

You should have a branch of the version control system where you keep the code you are developing. This branch may be newly created if you are developing a module from scratch or you may have checked out code from an existing branch. Either one of these activities is the trigger to create a separate development environment for your individual use. This is the build step of the development environment life cycle.

Build

The build step creates either a VM or a container pre-loaded with the software you need for your module. This includes libraries, operating system, and dependent modules. Your IDE should be set up to use this environment as the destination for its activities. That is, the IDE completes building an executable by incorporating the code you are currently modifying. Thus, you have an executable which you can use to test your code.

It is important that the software loaded into your development environment be identical to the software loaded into your team mates' development environments. Identical means down to the version numbers of the libraries and operating systems. Incompatibilities among different versions of libraries or operating systems is a major source of errors during the Integration stage of the deployment pipeline. Furthermore, the development environment you create should match the production environment, as well, in terms of version numbers. The rationale is the same, different versions may have different behaviors.

This brings us to the next step in the life cycle of a development environment, the Test step.

Test

Associated with your module, you should have a collection of tests. These tests cover both sunny day tests – no exceptions or error conditions – and rainy day tests where error conditions and exceptions are the focus of the test.

Tests can be written before you write the code, can be generated as a result of writing the code or can be regression tests. A regression test is a test to catch an error from a prior version of your module. Sometimes errors creep back in for various reasons and regression tests are intended to ensure that your module does not re-introduce any known errors.

Tests should be version controlled and saved in the version control repository.

You have loaded tests into the version control system, the IDE has loaded your module into an executable form, and this step runs the tests against the module using the test harness. The tests are run by a specialized testing application and it reports errors to you. Look at Figure 6.xx again to see this portion of the workflow.

In addition to testing the execution of your module, you can also test code quality. A static analyzer can look at your code and detect certain types of errors. Static analyzers will report errors that are not truly errors (false positives) and so you should not expect to get a clean bill of health from the static analyzer. This differs from the execution tests where you should expect all of the tests to pass.

This is also the step where a peer review of your code should occur. In theory, every artifact should be reviewed and the code you have generated is not an exception. Peer reviewing everything is time consuming both from your perspective and the perspective of the reviewer so it is a judgement call whether your code should be peer reviewed. The decision will depend on the importance of your module and your coding maturity.

Bake

Baking refers to saving the VM or container that you created in the build step. There are two reasons for saving the VM or container from the development environment. It is available for use by future steps in the deployment pipeline and it can be used to quickly run additional tests or to re-run existing tests. Saving eliminates the need to recreate the VM or the container and, thus, saves the time required for these activities.

Release

Once your module has passed the tests in the deploy step, your module can be checked in. Also, your branch of the version control graph should be merged into another branch – typically the one from which it was created. Checking in will trigger the integration step.

The release step will also trigger recording information in the artifact data base. The version number of the module being checked in as well as the version numbers of the tests should be recorded. Also record the configuration parameters, the version of the test tool used and the version of the IDE and any plugins used.

Teardown

As we said above, all of the resources used in the development environment should be released.

To summarize, having a defined life cycle for the development environment allows you to create scripts or use scripts created by other members of your team, to perform the activities. This eliminates much of the overhead from the development process and frees you to focus more of your energies on actually creating the module.

Integration environment

The purpose of the integration environment is to build an executable version of your microseervice and test it for functionality. It is triggered by a commit of a module. The activities during the integration stage are to get the latest copy of all of the modules included in your microservice and all of the dependencies, perform compilation of all of these modules, and to build an executable. The executable is then tested for functional correctness and, if it passes, it is promoted to the staging environment.

We now discuss these activities in the context of the life cycle for the integration environment.

Build

The build step is triggered by a commit to the version control graph for your source code. It creates the virtual machines or containers for your microservice. This involves loading, compiling, and linking your source code as well as the source code for all of the other modules in your microservice. It also involves creating the rest of the environment - populating the test database, creating the test harness, creating a VM with a load balancer, setting up the configuration parameters for integration and linking either to external services or to mocks of the external services.

This step is executed by a continuous integration (CI) server and will detect errors that prevent the linking such as Incorrect interface names, incorrect signatures for some languages, and other fundamentally syntactic types of errors. You are informed of these errors via email or via a web page from the continuous integration server.

Test

The purpose of the test step during the Integration environment is to test the microservice for functional correctness. That is, the focus is on the output values being correct. Testing for qualities such as performance or security is done during the Staging Environment.

The test database was created during the build step. The data in this database should be realish but limited in volume. Realish means that real data should be used as long as that real data does not involve any Personally Identifiable Information (PII). Realish also means that a variety of different data values are included that represent the kinds of values that will show up in the production database.

The volume of data is limited for two reasons. First, it will affect the time taken by the tests. There is a tradeoff in this step between the realism of the data base and how long it takes to perform the tests. Testing is a time consuming portion of the integration stage and the goal is to shorten the time required while still discovering as many errors as possible. The second reason the data is limited is that the database should be refreshed after each test. Starting the database from a consistent state is necessary so that the tests are repeatable. You do not want to be informed of an error in the tests and then not be able to reproduce the error. Consequently, the data should begin each test in a consistent state. Recovering the original state of the data can also be a time consuming process depending on the volume of data involved. Both of these reasons argue for limiting the size of the data base in this stage. The staging environment, on the other hand, should have as realistic a database as possible.

Each module that is loaded from the version control system – the module you just checked in as well as other modules in the midcroservice - is tested by its unit tests. Although unit tests happen during the Development Environment, they are repeated here as well. Two reasons exist for this repetition. First, the unit tests during the Development Environment stage were run on a smaller database than is used in the integration environment. Errors that depend on particular data values can be found more easily if the unit tests are run again than if they occur during integration test. Secondly, the set of unit tests may have been extended from when the module was originally tested. Regression tests could have been added or other specialized tests could have been added to the original test suite. Unit tests are fast and so repeating the unit tests does not extend the test time extensively.

In addition to the unit tests, integration tests are also run. These integration tests cover the whole microservice as opposed to just one module. Integration tests may come from a quality control group, from microservice use cases, from regression tests, or from the development team. Microservices are owned by a single team and that team assumes responsibility for integration tests, regardless of their source.

Bake

Once the microservice has passed its tests the VM or container with that microservice is copied into a location where it can be included in future tests. The integration environment tests the microservice. The next environment – the staging environment – tests the whole application. The continuous integration service has created a microservice specific package – either VM or container = and this package is saved in a known location for further use.

Release
The release step triggers the staging environment. It passes the location of the microservice specific package to the staging environment. It also record information into the artifact database. The information recorded includes the location of the microservice specific package, the version numbers and source of all of the included modules, and the version numbers of all of the tests run. The version of the test tool and the continuous integration server are also recorded since defects in them may cause errors and also because any errors not caught by the tests may be traced to these tools. Also record the configuration parameters since the behavior of your microservice will depend on these parameters.

Teardown
Teardown, as we said previously, releases all of the resources used by the continuous integration environment.

Staging environment
The staging environment is the place where the application is tested for its qualities. Primarily performance but also security and several other nonfunctional qualities. The environment in general, is as real as possible. For systems with global scope such as Google or Amazon, replicating the production environment is not possible but making the environment as real as possible is a goal of this stage. As always we begin with the build step of the life cycle.

Build
The staging environment is triggered by your application passing the integration stage tests. All of the VMs or containers that exist in the production environment should be created by the build step of the integration environment. The configuration parameters created for the staging environment should be the same as the production environment with the following exceptions: The database should be a separate staging test database not the production database, the credentials should be those appropriate to the staging environment, not production credentials, and any external services to which your microservice writes should be mocked.

The database for the staging tests should be as real as possible with the exception of the Personally Identifiable Information (PII). If it is possible, a copy of the production database should be used with the PII obscured. This is only possible if the production database is of a size small enough to copy quickly. We discuss the obscuring of PII in Section xx.xx.

Test
The test phase involves multiple different types of tests and different potential sources of input. We begin with load testing.

Load Testing
The purpose of load testing is to determine how well your application performs under load. Three possibilities exist for the source of the load

1. A load testing tool. A load testing tool’s purpose is to generate synthetic loads for applications. Artillery is such a tool. You provide the tool with a description of the input and its distribution. The tool generates synthetic loads in accordance with your description and measures the latency of the responses.

2. Playback. Record input from to the production version of the application. This input is in the form of http or https requests. Play these requests back on the new version of the application with the time interval that was used to record the requests. By recording the responses from the production version, you have a means of testing the responses of the new version. By time stamping requests and responses, you have an estimate of the latency and throughput. We discussed the problem with time stamps across different computers in Chapter xx but in this case since you are measuring from the requestor's computer which is also the recipient computer. Thus, the latencies and throughputs you calculate will be consistent.
3. Tee the input to the production version. "Tee" is a Unix command that takes one input stream and generates two output streams that are identical to the input stream. One branch of the tee will go to the production version and the other to the test version. Comparing the responses of the two versions will give you a correctness test and comparing the times of the responses will give you latency and throughput comparisons.

Security testing

The staging environment is also where security testing occurs. There are two types of security testing – runtime and static analysis.

Runtime security testing

Applications have vulnerabilities. A vulnerability is some portion of the application where an attacker can gain control of the application for malicious purpose. Vulnerabilities are discovered and reported to both the vendors and to centralized vulnerability collection agencies. Vulnerabilities are subsequently made publicly available so that systems administrators and others can apply patches and remove the vulnerability. One class of interested parties are vendors of security testing tools.

OWASP (Open Web Application Security Project) is one of those security testing tool vendors. It tests for known web vulnerabilities. Other security testing tools fall under the heading of Pen (Penetration) testing tools. They test not only for web based vulnerabilities but also vulnerabilities in other portions of the stack.

These types of tools are used during the staging environment to perform runtime security testing on your application.

Static analysis

A static analysis tool examines the source code of our application. It looks for questionable code within your application using techniques derived from compiler theory.

For example, a common vulnerability is exploiting buffer overflow. You in your code allocate a certain amount of memory for an input buffer, for example. An attacker provides input much longer than your allocated memory and the resulting error provides an opportunity for the attacker to gain control of your application. The static analyzer will ensure that you check every input into the allocated buffer to verify that you prevent buffer overflow.

Any code discovered by the static analyzer that does not conform to a predefined set of rules such as "test input for length" is flagged and results in a message to the tester. Static analyzers have the potential to generate a great many messages many of which are false positives. A good static analyzer will have severity level associated with its messages so that the tester can focus attention on the most serious problems.

License conformance

A third type of testing that occurs during the staging environment is testing for conformance to licenses. This is also done using static analyzers. All software comes with a license except software in the public domain. The license specifies the terms under which you may use the software. Your use of this software has to conform to these license terms. Different open source licenses allow you to do different things. The GPL for example, restricts your commercial use of the software. Some licenses do not allow you to modify the software.

The static analyzer will look at all of the source code included in your application and check their licenses against a set of rules established, probably, by the legal department of your organization.

In general, the message is that you should be aware of the existence of different types of licenses and the obligations that you and your organization are assuming by using code protected by these licenses.

Deployment

Once your application has passed the tests that occur in the staging environment, the executable whether packaged as a VM or a container should be placed on a server for deployment into production. It may be deployed automatically or a human may be required to authorize the deployment depending on your domain and your organization's policies.

As before, artifact information is recorded in the artifact database. This information includes the URL of the testing database, the version of any tools used in this stage, and the settings of configuration parameters.

Teardown

The final step in the staging environment, as with all of the other environments, is to deallocate all of the resource used in the staging environment.

Deployment

Placing a microservice into production is more complicated than just installing an instance of the microservice and directing messages to it. Compatibility with other microservices is one issue. Another is the sequencing of instantiating instances of the midcroservice. In Chapter 5 we introduced "feature toggles" as a mechanism to allow you to distinguish between installing a new version and activating the new code in that instance. Our discussion of deployment will build on this use of feature toggles. We begin by discussing the two basic all or nothing deployment models and then we discuss two different partial deployment models.

All or Nothing Models

The context is that we have an application designed as a microservice architecture and we have a new version of Service A to be deployed. Figure 6.xx shows Service A located somewhere in the service graph with clients, Service C, and with dependent software, Service D. The current version of Service A is A' and we wish to deploy the new version of Service A – A''.

N instances of Service A' are being used and we wish to arrive at a situation where N instances of Service A'' are being used and no instances of Service A'. We wish to do this with no reduction in service to the clients of Service A.

Two different options exist for accomplishing the deployment

- Blue/green. Blue/green is also called Red/Black or you could choose your colors. This option calls for allocating N new instances and populating them with Service A''. Once the N instances of Service A'' has been installed, change the DNS server to point to Service A'' and drain and delete the N instances of Service A'.
- Rolling Upgrade. Rolling Upgrade essentially replaces the instances of Service A' with instances of Service A'' one at a time. You could replace more than one at a time but a small percentage of N are replaced in any single step. The steps of the Rolling Upgrade are:
    - Allocate a new instance
    - Install Service A''
    - Begin to direct requests to Service A''
    - Drain Service A' from one instance
    - Delete that instance
    - Repeat above steps until all instances have been replaced.

Figure 6.xxx shows a rolling upgrade process for a system using Amazon Ec2.

Trade offs between Blue/green and Rolling Upgrade

One trade off is financial. Blue/green will result in 2N instances being simultaneously active whereas Rolling Upgrade will result in N+1 instances being simultaneously active. When organizations purchased computers the economic trade off was compelling and rolling upgrade was essentially standard. Now that computing resources are rented rather than purchased, the economic trade off is less compelling but rolling upgrade is widely used.

A second trade off between the two models is in logical difficulty. When using the Blue/green deployment model at any point in time, either Service A' or A'' is active but not both. When using Rolling Upgrade both Service A' and Service A'' are simultaneously active. This introduces the possibility of a type of logical inconsistency – horizontal inconsistency.

A final trade off is the possibility of an error in Service A''. When using Blue/green deployment by the time an error is discovered in Service A'' all of the instances of Service A' may have been deleted and rolling back to Service A' will be time consuming. A rolling upgrade gives you the possibility of discovering an error in Service A'' while instances of Service A' are still available.

We now discuss how to maintain horizontal consistency when using Rolling Upgrade.

Horizontal consistency with rolling upgrade

The consistency problem is that a client C may interact with Service A and get an instance loaded with Service A'' on one call and get a response which it then returns to Service A. On the second interaction, it may get an instance that has not yet been upgraded and interact with Service A'. Service A' may not understand the interaction and generate an error. Figure 6.xx shows a sequence diagram with this error.

Maintaining horizontal consistency involves utilizing feature toggles and making a distinction between "installing" Service A'' and "activating" Service A''. The following steps will maintain horizontal consistency among different versions of Service A.

- Write new code for Service A'' under control of a feature toggle.

- Install N instances of Service A’’ one a time using the sequence given above. When a new instance is installed begin sending requests to it. The new code is toggled off.
- When all of the instances of Service A are running Service A’’, activate the new code using the feature toggle. Use a distributed coordination service to ensure that all instances are turned on simultaneously.

Vertical Consistency

Vertical consistency means that the client of Service A can call Service A and get a correct response regardless of whether it is serviced by an instance with Service A’ or Service A’’. Suppose Client C assumes that Service A has been upgraded and it has not. Conversely, assume Client C does not assume Service A has been upgraded but it has and the upgrade involved changing an interface. In either of these cases, an error will result. Vertical consistency is a problem regardless of whether Blue/green or Rolling Upgrade is used as a deployment strategy.

Two techniques are used to achieve vertical consistency

- Feature toggles. This is the same use of feature toggles that we saw when discussing horizontal consistency. The code implementing a new feature in Client C is placed under the control of a feature toggle as is the code implementing that feature in Service A. The feature is turned on simultaneously for both Service A and Client C.
- Backward and forward compatibility. Not all clients of Service A may have been involved in the new feature and these clients must continue to be supported. Backward compatibility means ensuring that old interfaces continue to be supported. An interface is never deprecated (removed) but if a change is required, the interface is extended. Internally within Service A, the interfaces can be modified however it makes sense to you but the external interfaces can only be extended and never otherwise modified. This leads to a translation layer for Service A as shown in Figure 6.xx. The compatible interfaces are translated into the internal interfaces. You may argue that this is cumbersome and it is but Google recently replaced a module in use since 2002 and interface version 1 was still supported through over 100 interface extensions.

  Forward compatibility means that unknown calls are handled gracefully. It may be that your client is assuming an interface that has yet to be placed into service. So rather than failing the call return a meaning error code that indicates the call in not recognized. Your client also needs to be able to handle such a return code gracefully and either retries hoping the message is routed to the new version or uses some fallback mechanism.

  Dependent services under your control will be backward compatible and you need to set up your microservice to gracefully handle error returns. For dependent services not under your control – from third parties, for example – you should localize all interactions to such services into a single module so that inconsistencies are limited in scope.

Data consistency

Data in the database can become inconsistent if the schema changes. Then Service A’ assumes a different schema from Service A’’. The simplest solution is do not change the schema but this is not always possible. A more realistic solution is to treat schemas as interfaces are treated. That is, only extend the schema, do not change any existing fields. Each version of Service A will access data using field identifiers that it knows are supported.

Modern database systems will convert data from one schema to another in the background while still supporting both schemas. This automatic conversion requires you to write specific translation routines to derive the new elements of the schema from the existing form. You will also have to write translations routines that go in the other direct to derive the old version of the data from the new version. This backwards translation is necessary to allow access from the old schema.

Partial Deployments

We have discussed models for all or nothing deployments but there are also partial deployments. Canary testing is done for quality control purposes and A/B testing is done for marketing purposes.

Canary Testing

Before rolling out a new release it is prudent to test it in production but with a limited set of users. This is the function that beta testing used to serve and now it is done with “canary testing”. Canary testing is named after the practice from the 19th century of bringing canaries into coal mines. A variety of noxious gases both explosive and poisonous are released during the coal mining process. Canaries are more sensitive to these gases than humans and so coal miners used to bring canaries into the mines and observe them for signs of reaction to the gases. The canaries acted as early warning devices for the miners. See Figure 6.xx for a picture of a coal miner with a canary.

In modern software use, canary testing means to designate a set of testers who will use the new release. These testers are typically members of the organization that is developing the software. Thus, Google employees almost never use the release that non Google users would be using but, instead, they act as testers for upcoming releases.

The testers get access to the canaries through DNS settings. The local DNS for the testers is set to point to the new version. Then, if the canary tests are passed, all of the relevant DNS settings will be set to point to the new version.

A/B testing

A.B testing is used by marketers to decide which one of several alternatives yields the best business results. Examples include Ebay testing whether allowing credit card purchases drives up participation in auctions and which of two enticement offers at a bank resulted in more new accounts. Probably the most famous story is Google’s deciding which shade of blue to use to report their search results by testing 40 different shades of blue.

The implementation of A/B testing is the same as the implementation of canary testing. DNS servers are set to send requests to different versions and the different versions are monitored to see which one provides the best response from a business perspective.

Rollback

Not every new version works correctly, A version may have either logical or quality issues that require replacement of the version. Recall from Section 5.xx that your microservice has Service Level Objectives. Once it goes into production, you should monitor these SLOs to verify they are being met. If you have determined that they are not being met, you may wish to replace the release.

Two options exist for replacing a release: rollback and roll forward.

Roll back means replacing the current version with a prior version. This can be as simple as turning off a feature toggle or it may involve discontinuing the deployment of the new release and redeploying a prior release that is known to meet your quality goals.

Roll forward means fixing the problem and generating a new version. This depends on being able to deploy the new version quickly.

Summary

The deployment pipeline begins when you commit your code to a version control system and ends when your application has been deployed for users to send it requests. In between, a series of tools and automated tests integrates the newly committed code, tests the integrated microservice for functionality, tests the application for performance under load, for security, and for license compliance.

Each stage in the deployment pipeline consists of an environment established to support isolation of the stage and perform the actions appropriate to that stage. The creation of the environment is triggered by some explicit action and ends with the release of all of the resources used by that environment.

The deployment stage of the pipeline deploys the tested application either using a Blue/green model or a rolling upgrade. Partial deployments are used for quality control or to test market response to a proposed change or offering.

Exercises

1. Write a script to create a development environment. Develop a Java module within that environment that tests whether an input value is a prime. Use Git to manage the versions of the script.
2. Write a script to create an integration environment for an application that prints out the first N prime. The application should have two instances of LAMP and use HAproxy as a load balancer. Use Junit as your test harness.
3. Use Artillery to test your environment in exercise 1 with 10 and 100 simulated users.
4. Use Jenkins to build a .jar package for the prime number application.
5. Deploy a new version of the prime number generator in AWS using the Opsworks Rolling Deployment mechanism.

Discussion

1. Generate a description of how a buffer overflow enables an attacker to gain control of your application.
2. Write down all of the information recorded in the artifact database for the integration step.
3. Determine all of the licenses used in the .jar package created in exercise 4. What restrictions are imposed by these licenses.