Carnegie Mellon

Deployment and Operations for Software Engineers 2nd Ed

Chapter 11 – Deployment pipeline

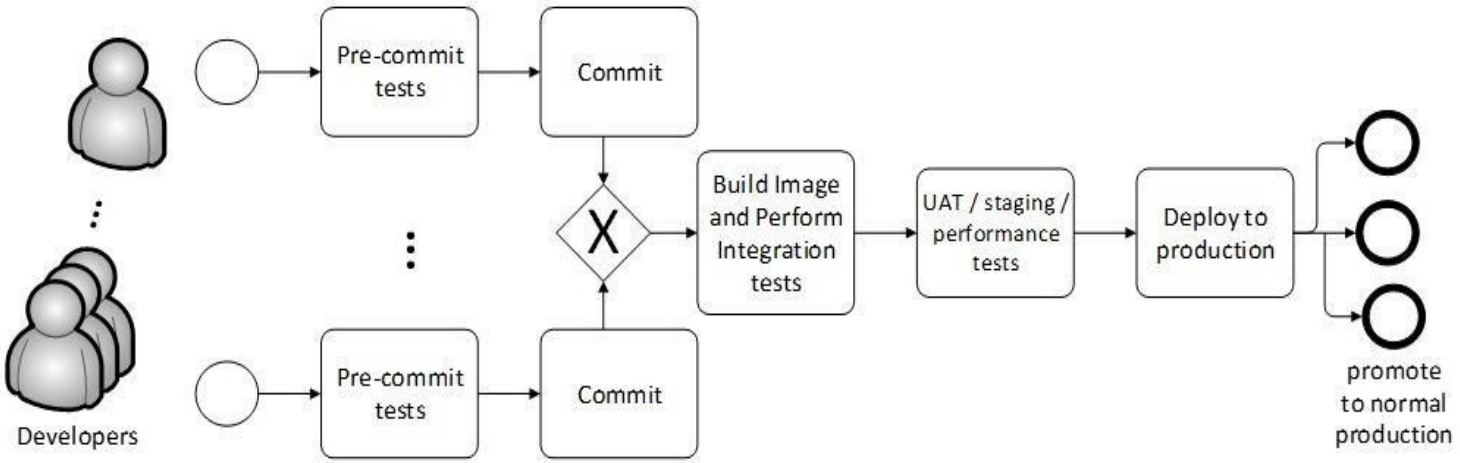
Outline

- **Overview**
- Environments
- Development environment
- Integration environment
- Staging environment
- Deployment

Overview of a deployment pipeline

- The deployment pipeline is the sequence of actions that move the code you write into production.
- We will present it as four stages – development, integration, staging, production.
- Integration is also called build.
- Organizations vary these stages
  - Some have no staging
  - Some hage multiple staging
- Each stage has a separate environment—separated from other environments

Deployment pipeline



Types of tests

- One purpose of each environment is to perform tests. Two types of tests can be used
  - Runtime tests
  - Static Analysis

Runtime tests

- The portion of the system being tested is executed and a set of tests are run on it. This requires a test harness.

Static analysis

- This type of testing is applied to the source code. Using techniques similar to those used in compilers, a static analyzer looks for problematic code constructions. Static analyzers have both false positives and false negatives. The type of problems flagged by a static analyzer are those that can be found in the Common Weakness Enumeration CWE).
- The CWE is a list of software weakness types..

Development environment overview

- In the development environment, you develop your modules.
- Unit tests are run on each module.
- The module is checked into the version control system. This triggers the integration stage.

Build/integration environment overview

- The build server compiles your new or changed code,
- It also compiles the code of other portions of your service.
- If you are working in an interpreted language such as Python or Javascript, there are no compilation steps.
- The build server constructs an executable image for your service.
- This executable image is then tested for functional correctness.
- After the executable image passes your service's functional tests, the staging environment is triggered.

Staging environment overview

- Tests the quality of the service
  - performance under load,
  - security vulnerabilities,
  - Compliance with regulations and licenses.
- User-acceptance testing (UAT) may occur at this stage depending on the business context of your service.
- Once the service passes its quality tests it is
  - Placed into production if you are doing continuous deployment
  - Marked as ready for deployment if you are doing continuous delivery

Production environment overview

- Once in production, the service is monitored closely until there is some confidence about 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.

Qualities of the pipeline

- Cycle time
- Traceability
- Repeatability
- Security

Cycle time

- Cycle time is the time taken to move through the pipeline.
- If humans are involved, cycle time is much increased.
- Continuous deployment depends on automated testing.

Traceability

- Traceability is the ability track all of the elements of a service in production.
- This includes
  - the code and dependencies that are included in that service.
  - The test cases that were run on that service
  - The tools that were used to produce the service.
- Traceability information is kept in an artifact database. I
  - code version numbers,
  - dependency version numbers,
  - test version numbers,
  - tool version numbers.

Repeatability

- Repeatability means that if you perform the same action with the same artifacts, you should get the same result.
- This is not as trivial as it sounds. For example,
  - Your build process fetches the latest version of a dependency. The next time you execute the build process, a new version of the dependency may have been released.
  - One test modifies some values in the database. If the original values are not restored, subsequent tests will not produce the same results.

Flaky tests

- Sometimes tests are not repeatable for a different reason. So called *flaky tests*.
- One cause could be parallelism in the system. Two different executions of a test could take different paths because of timing differences with parallel threads.
- Log examination is a method for resolving flaky tests.

Security

- 20-25% of security breaches are caused by insiders.
- Techniques to reduce the possibility of an insider attack on the pipeline are
  - Limiting access to the pipeline tools,
  - keeping a record of changes and the originator of the changes,
  - Notifying team members that a pipeline tool has been updated or modified.

Discussion questions

1. What are the tradeoffs between doing quality testing in the integration step and doing it in a separate staging step?
2. What is the relation between cycle time and the number of deployments per day?

Outline

- Overview
- **Environments**
- Development environment
- Integration environment
- Staging environment
- Deployment

Purpose of an environment

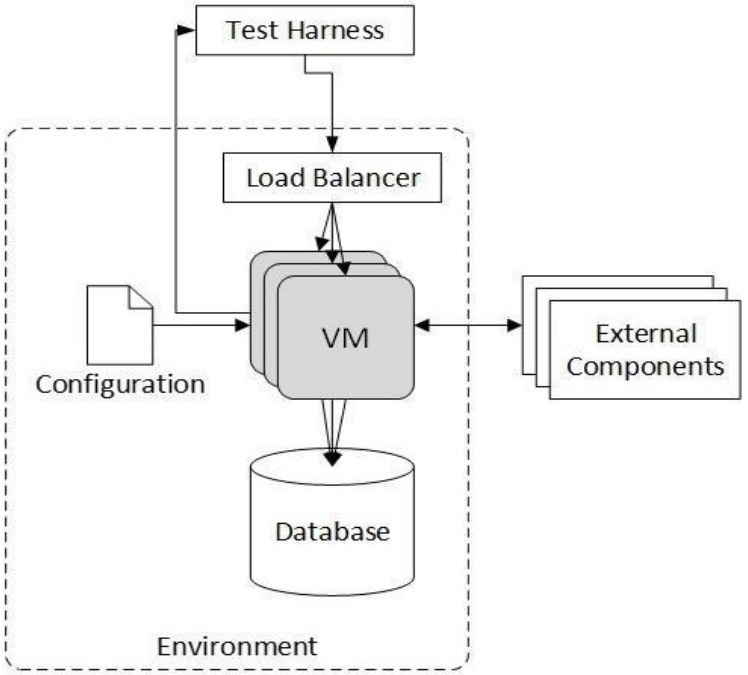
- The purpose of each environment, except for production, is to provide a place to perform testing that is isolated from other development or testing activities that might be ongoing.
- The production environment is treated differently.

Requirements for an environment

1. The environment is isolated from other environments. Including
   - address spaces
   - inputs
   - modifications to any database
   - other dependencies on the processes being executed in the environment.

Elements of the environment

- collection of virtual machines or containers
- infrastructure services (for example, a load balancer)
- a source of input for runtime tests
- a database
- configuration parameters
- external services



Collection of Virtual Machines or Containers

- The modules, services, or system being executed in this environment.
- The collection of services in the environment grows as the system moves through the pipeline.
  - In a development environment, a single module is being tested;
  - in an integration environment, the service is being tested;
  - in a staging environment, the service plus other services is tested.

Infrastructure Services

- Your service depends on infrastructure services.
  - In the development environment, you may need only a load balancer and logging,
  - In other environments you may need services such as registration and discovery.

Source of input

- The purpose of each environment is to run tests, and tests require a source of input.
- The input can come from
  - a test harness or dynamic workload generator
  - from live users
  - by replaying a previously captured input from live users.

Database

- The database must be restored after each test.
- The tests must be repeatable and give the same results every time they are run.
- Any given test may modify data in the database. Restoring the database after each test ensures that each execution of each test begins from the same state.

Configuration Parameters

- A configuration parameter is a value that is bound at runtime,
- Each environment will have a set of configuration parameters
- The configuration parameters will vary from environment to environment.

External Services

- External services 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 service is in and whether the external service is read only or read/write.
- If your service is not in the production environment and the external service is read/write, it must be stubbed or mocked.

Life cycle of an environment

- **A**n environment has a finite lifetime
  - Created
  - Used
  - Cleaned up
- Creation and clean up can be automated

Create

- The creation of an environment is triggered by some event.
- Which event depends on the purpose of the environment

Actions of the create step

- Create a VM or container loaded with your software.
- Create a load balancer. The load balancer is a separate VM or is allocated from a PaaS or service mesh.
- Create the test harness.
- Initialize the database for the environment.
- Create environment-dependent configuration parameters

Usage

- Usage will vary depending on the purpose of the environment.

Clean up

- Save any persistent artifacts created by the environment.
- Adjust any necessary data.
- Remove any resources created.

Variations

- Other variations on the pipeline stages include
  - Test a complete service in the development stage (rather than just your own module), and then test the entire system (your service integrated with other services) in the integration stage.
  - Quality testing that we describe in staging is performed in the integration step
  - The integration stage might test “typical” customer use cases
  - Multiple staging environments exist for different purposes

Discussion questions

1. Would UAT testing be done in a separate staging environment or the normal staging environment? How would you do UAT testing with real users?
2. Where does the data in the test database come from?

Outline

- Overview
- Environments
- **Development environment**
- Integration environment
- Staging environment
- Deployment

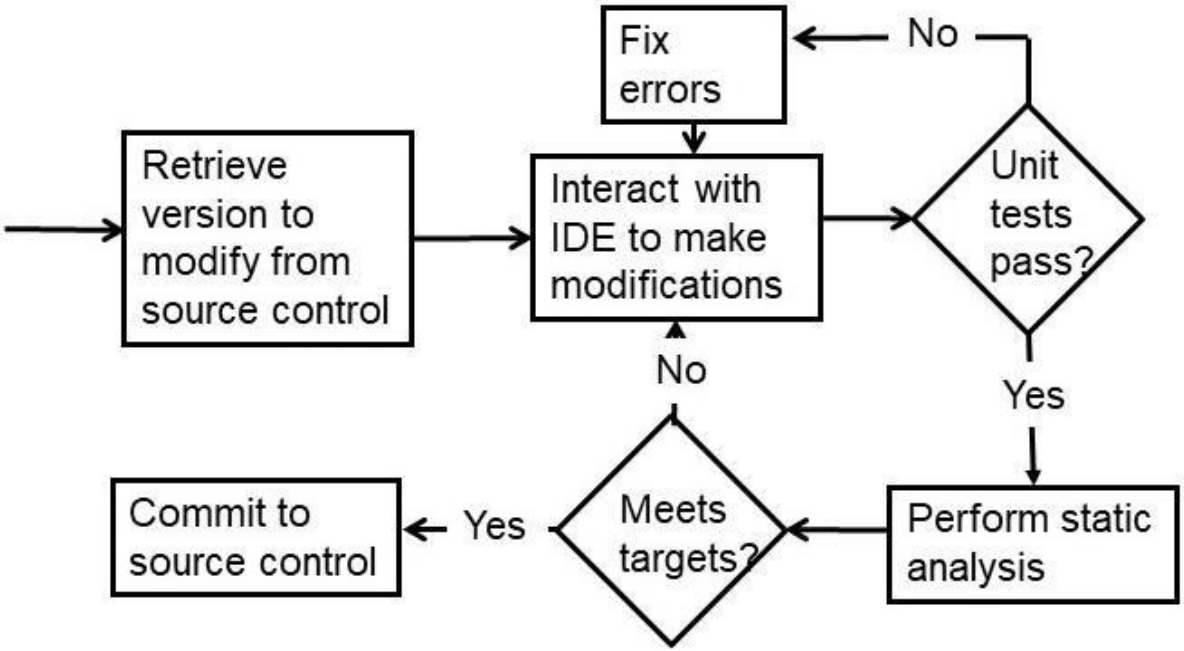
Development environment

- You are working on a single module.
  - It may represent a new service
  - It may be the maintenance of an existing service.
- You will interact with the version control system and an IDE
- There is a branch of the version control system where you keep the code you are developing.
  - It may be newly created
  - It may have checked out code from an existing branch.
- Either one of these activities is the trigger to execute the create step

Development environment – create

- Create loads the VM or container with the software you need for your module, which includes operating system, libraries, and dependent modules.
- Your IDE should be set up to use this environment as the destination for its activities:
- When you compile your new or changed code into an executable form, the IDE should place the executable artifacts into the development environment so you can begin testing.

Workflow in development environment



Unit tests

- Execution tests include
  - sunny-day tests conditions
  - rainy-day tests
  - regression tests
- Tests should be version controlled and saved in the version control repository.

Static analysis

- Checks code quality
- Looks for particular types of patterns that may be problematic.
- Generates false positives
- Has severity level settings

Security

- A static analyzer can look for known security weakness patterns.
- If you package your module as a container, a scanning tool can produce a bill of materials for your container and check it against known vulnerabilities.

Development environment – clean up

- Save the VM or container image
- Check your module into the version control system.
- Record information in the artifact database
  - The version number of the module being checked in and the version numbers of the tests.
  - The configuration parameters,
  - The version of any tools used for testing or static analysis,
  - The version of the IDE and any plugins used.
- All resources used in the development environment should be released.

Discussion questions

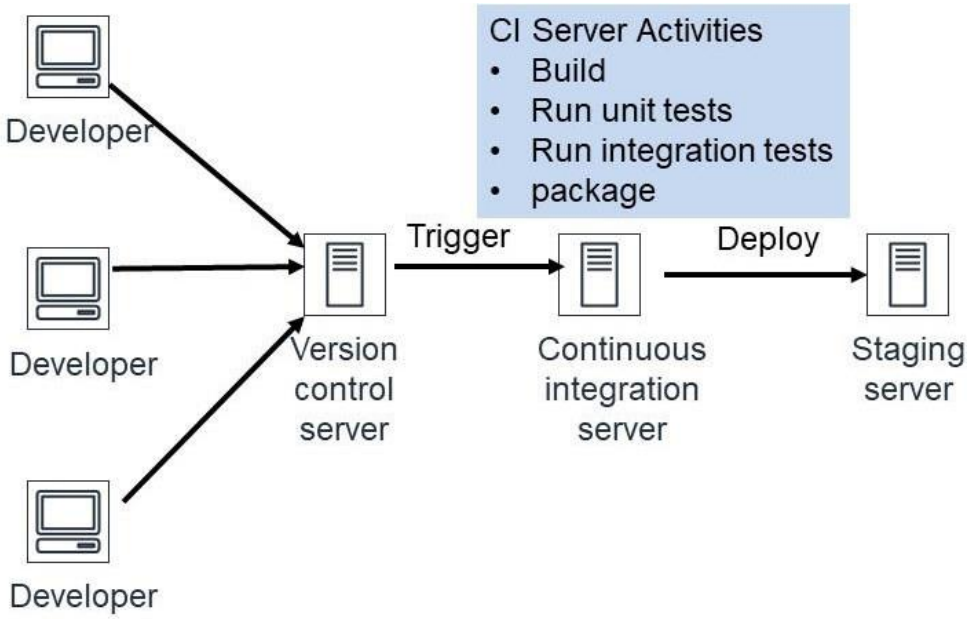
1. Why must all of the unit tests be passed but not all of the static analysis tests?
2. Sketch a script that performs the “create” step.

Outline

- Overview
- Environments
- Development environment
- **Integration environment**
- Staging environment
- Deployment

Integration environment – context

- The integration environment is where the continuous integration server operates
- It is triggered by a commit to the VCS
- It builds an executable, runs unit tests, integration tests, and packages a deployable image



Integration environment—create

- The Continuous Integration (CI) server requires two virtual machines.
  - One for the CI server itself
  - One is the workspace for the CI server.
- Populates the test database,
- Creates the test harness,
- creates a VM with a load balancer,
- Sets up the configuration parameters for integration,
- Links either to external services or to mocks of the external services.

Integration environment—usage

- Build a service. This is performed by the CI server.
  - Compile and link all the source code for the service
  - Load all dependencies required by the source ode
- Test for functional correctness
  - Unit tests from all included modules
  - System wide tests – sunny day, rainy day, regression
- Enough tests for sufficient coverage but not so many that the testing takes too much time

SBOM

- When the executable is being built, an SBOM (Software Bill of Materials) can be created.
- An SBOM enumerates all the dependencies included in your service.
- This allows for scanning for vulnerabilities once you service is in production.

Test database

- Data in the database should be
  - Representative of real data
  - Not too voluminous
- Personally Identifiable Information (PII) should be obscured
- Data base should be restored to its original state after each test.

Integration environment – clean up

- The VM or container with that service is saved for future use.
- Record information in the artifact database, including
  - the location of the service-specific package saved,
  - The version numbers and source of all the included modules,
  - The version numbers of all the tests run.
  - The version of the test tool and the continuous integration server
  - Configuration parameters
- Release resources used by the integration environment.

Discussion questions

1. How do you obscure PII?
2. Why are all of the unit tests rerun in the integration step.

Outline

- Overview
- Environments
- Development environment
- Integration environment
- **Staging environment**
- Deployment

The staging environment--overview

- The staging environment tests the whole system for its qualities.
  - performance under load,
  - Security
  - other nonfunctional qualities.
- The staging environment should be as close to the production environment as practical.
- Triggered by the clean up step of the integration environment.

Staging environment—create

- The integration environment will create all the VMs or containers for your service that exist in the production environment, so they do not need to be created for the staging environment
- Your service in the staging environment should not differ from that created in the integration environment.
- The configuration parameters created for the staging environment should be the same as the production environment, with a few exceptions:
  - Use a separate staging test database
  - Use credentials appropriate to the staging environment
  - Mock any external services that your system writes to.

Database

- The database for the staging tests should be a copy of the production database, or a large extract, if the full production database is too large.
- When using production data, you must obscure any sensitive or restricted data.

Staging environment—usage

- Testing types
  - Load testing
  - Security testing
  - Compliance testing
  - User testing
- The final step in the staging environment is deployment to production

Load testing

- Load testing has three portions
  - Defining the load
  - Applying the load
  - Measuring the system's throughput and/or latency.

Defining the load

- The contents of the oad should reflect the distribution of values that you will see in production.
  - It may be that your system receives mostly one type of request in production,
  - there may be hotspots in your database
  - Requests in production is may come in bursts
  - The number of simultaneous users may increase very slowly, or rapily

Sources for the load

- There are three possible sources for the load:
  1. *A load-testing tool.* A load-testing tool, such as Artillery, generates synthetic loads for systems. 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 to the production version of the system and use the recording as a source of test input
  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.

Applying the load

- Two approaches to applying the load
  - Load-testing tools, which combine load synthesis with load response.
  - Build a custom test harness.
- Ensure that the software applying the load does not introduce any limits on the request rate during test execution – you want to be sure that you are measuring the performance of your system and not your test harness.

Measuring the system

- Challenges to measurement
  - Accurately handling long-tail latency. This leads to running tests that execute so many requests that it is not practical to save the results of every request, and so the measurement framework must build latency histograms and calculate metrics on the fly.
  - The heterogeneous performance delivered by the underlying hardware in the cloud. Some of the physical computers may be slow because of disk or network-hardware issues, or some may have newer processors that deliver better performance. Your performance testing should cover enough time and enough physical hardware to ensure that you are accurately predicting your system's performance in production.

Runtime security testing

- .OWASP (Open Web System Security Project) is one tool vendor that focuses on vulnerabilities in web systems.
- pen (penetration)-testing tools. They test not only for web-facing vulnerabilities but also for vulnerabilities in other portions of the stack. These types of tools are used during the staging environment to perform runtime security testing on your system.

Static analysis

- Static analyzer tools examine the source code and look for insecure patterns of usage.
- Model checking is a technique that involves symbolically testing all possible paths of a system.
  - Specify an error condition and the model checker will determine whether that condition can ever occur in your service.
  - Model checkers suffer from state explosion caused by trying to symbolically execute systems that are too large. Model checking has been applied on systems up to 10,000 lines of code and, thus far, have primarily been used for operating-system and device-driver functions.

Compliance testing

- Test for conformance to regulations and license provisions
- This is also done using static analyzers.
  - regulation compliance
  - license compliance

Regulation compliance

- Different types of regulations impose different requirements on how your system handles data.
  - For example, HIPAA (Health Insurance Portability and Accountability Act) imposes a requirement that sensitive data be protected.
  - Other domains have their own regulatory requirements and specialized static analyzers can help determine whether the particular requirements are being met.
- A static analyzer can examine the code, see if regulated data has been properly identified and highlight how it is protected. An analyst can then examine the highlighted code and determine whether the HIPAA security requirement is being met.

License compliance

- Each type of open-source license allows you to do different things.
- A static analyzer can look at all the source code included in your system and check the system's license against a set of rules established, probably, by the legal department of your organization.

User testing

- The user-facing portion of your service must be tested to be sure it is acceptable to users.
- This may involve having actual users execute the system – a form of canary testing.
- It may also involve testing how the user interface is displayed on different devices.
- Common problems are inaccessible links and buttons and poor color contrast.

Deployment to production

- Once your system 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.

Staging environment—clean up

- The final step in the staging environment is to release the resources used.
- 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.

Discussion questions

1. What are the top three OWASP security vulnerabilities?
2. Sketch the design of a license compliance tool.

Outline

- Overview
- Environments
- Development environment
- Integration environment
- Staging environment
- **Deployment**

Feature toggles

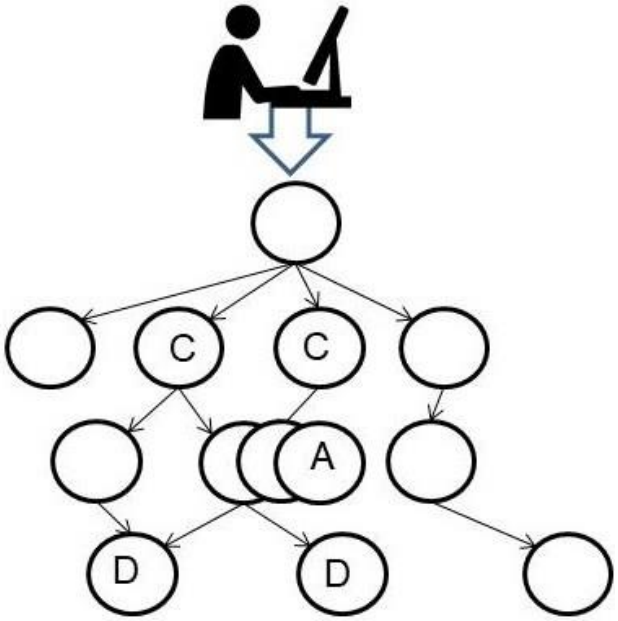
- Feature toggles are a mechanism to allow different versions of the same service to be simultaneously active.
- Feature toggles are “if” statements that make the new code for a service version conditional so that the new code executes only when the feature toggle is on. If the feature toggle is off, then the old code of the microservice is executed.
- Feature toggles should be removed from the code when no longer needed.

Installing vs activating a service

- When using feature toggles a new version is installed with the feature toggled *off* and when it is time to activate the feature, it is toggled *on*.
- The value of the feature toggle, whether toggled on or off, is maintained by a distributed coordination system so that all the instances of the microservice are toggled on–or off–at the same time.
- We will see several possible uses for feature toggles

Service graph for your system

- Updating Service A
- Clients C
- Dependencies D



Deployment

- Assumptions
  - You are performing continuous deployment of an upgrade to a service.
  - Image to be deployed is on a staging server.
  - The current version of the service (Service A) is to be replaced by a new version of the service (Service A').
  - There are N instances of Service A currently executing.
  - There is to be no reduction of service to the clients during the upgrade.
  - The deployer has the appropriate credentials for installing new version.

Goal of deployment

- The goal is to have N instances of Service A' executing and no instances of Service A

Basic deployment strategies

- Blue/Green.
  - Create N instances of Service A'.
  - Route incoming messages to Service A' instances
  - Drain and delete instances of Service A.
- Rolling Upgrade
  - Allocate a new instance.
  - Install Service A'.
  - Begin to direct requests to new instance of Service A'.
  - Drain Service A from one instance and then destroy that instance.
  - Repeat above steps until all instances have been replaced.

Tradeoffs

- *Financial.* The peak resource utilization for a blue/green approach is 2N instances, whereas peak utilization for a rolling upgrade is N+1 instances.
- *Responding to errors.* Suppose you detect an error in Service A' when you deploy it. If you are using blue/green deployment, by the time you discover an error in Service A', all the instances of Service A may have been deleted and rolling back to Service A could take some time. In contrast, a rolling upgrade may allow you to discover an error in Service A' while instances of Service A are still available.

Version skew

- You are updating Service A
- Services C (clients) and D (dependencies) may or may not have been updated.
- This results in inconsistent versions of the services.
- *Version skew* is the term given to inconsistent versions of services simultaneously being in production.
- Two types
  - *temporal inconsistency*
  - *interface mismatch*.

Version skew example

- Assume that Service A is a shopping-cart service that allows clients to add items to a basket and calculates the total price including discounts.
- You are updating the service to shift from calculating the discount item-by-item to calculating the discount based on the complete purchase.
- This change requires changes to Services A, its clients (C), and its dependent services (D).

Temporal inconsistency

- A request by client C to your service may be served by an instance running Service A.
- The return to client C will be a response that is based on Service A.
- Service C then makes a second call to your service using the response data, which may be served by an instance running Service A'.
- In the shopping cart example, the result is a cart in which some items are discounted twice–once as a single item by the Service A instance, and again when the Service A' instance discounted the entire cart.

Interface mismatch

- The interface in Service A' has changed from the interface in Service A
- If Service C calls Service A' with an interface designed for Service A, an interface mismatch would occur.
- Interface mismatch can occur whenever the update from Service A to Service A' involves changing the interface of Service A.

Interface mismatch

- Avoiding interface mismatch requires that any client of Service A be able to call Service A' and get a correct response, regardless of whether the request is serviced by Service A or Service A'.
- Service A must handle the cases
  - Client C assumes that Service A has been upgraded but it has not,
  - Client C does not assume Service A has been upgraded with an interface change.

Avoiding version skew

- Two techniques
  - Tagging messages
  - Feature toggles

Tagging messages

- Tag messages with the expected version of the recipient.
- It is the responsibility of the recipient to respond appropriately.
- If Service A' received a message tagged with A, it should interpret it as conforming to the interface of Service A.

Interface compatibility

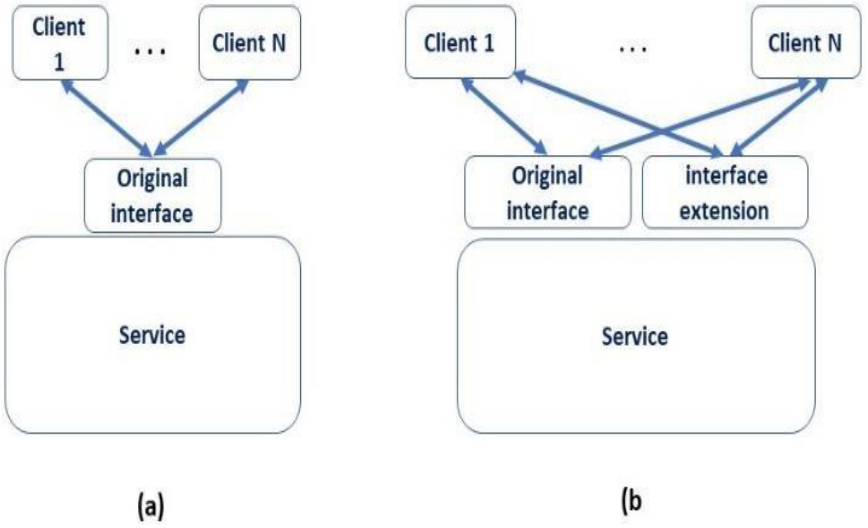
- Backward compatibility is supporting an outdated interface.
- Forward compatibility is being graceful when receiving a message tagged with a future interface version.
- The service should respond with an error message indicating that it does not recognize calls to this version
- The client then must be able to handle such an error message.
  - The client can try again hoping to be routed to an instance that supports the version it expects
  - fall back to a different method of achieving its goal,
  - report failure to its invoker.

Using feature toggles

- Write new code for Service A' under control of a feature toggle.
- Install N instances of Service A' using either the Blue/Green or the Rolling Upgrade approach. When a new instance is installed, begin sending requests to it without introducing any version skew, as the new code is toggled off.
- When all 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

A different approach to interface mismatch

- Extending an interface can also be used to manage interface mismatch.
- fields in an interface extension have different names than the fields in the original interface.
- (a) shows original interface and (b) shows an extended interface



Database schema evolution

- All writers and all readers must agree on the schema whether the schema is explicit (e.g. relational) or implicit (e.g. key, value)
- Well known and difficult problem,
- One approach is to treat a schema as you treat an extended interface.
  - The schema can be extended, but existing data remains valid.
  - Any version of a service will access data using field identifiers that it knows are supported,

Tool support for database evolution

- In some cases, you can use tools that convert data from one schema to another while leaving the database online.
- Thisautomatic conversion requires you to write specific translation routines to derive the new elements of the schema from the existing form.

Partial deployments

- You may not wish to replace all of the instances of Service A
  - Canary testing
  - A/B testing

Canary testing

- Canary testing means to designate a set of testers who will use a new release.
- Sometimes, these testers are so-called power users or preview-stream users from outside your organization.
- 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.

Implementing canary testing

- The testers get access to the canaries through DNS settings or through discovery-service configuration.
- After testing is complete, either the system is returned to its original version, or the new version is rolled out to all users.

A/B testing

- A/B testing is used by marketers by performing an experiment with real users to determine which of several alternatives yields the best business results.
- A small but meaningful number of users receives a different treatment from the remainder of the users.
- The two categories are compared based on a business metric specific to the organization.

Implementing A/B testing

- The implementation of A/B testing is the same as the implementation of canary testing.
- Discovery services 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.
- Feature toggles can also be used to control which version users see.

Roll back

- Use in production may uncover functional or quality issues that require a version to be replaced. Two options exist for replacing a release: roll back and roll forward.

Roll back

- *Roll back* means replacing the current version with an earlier version. This may involve discontinuing the deployment of the new release and redeploying a previous release that is known to meet your quality goals. It could also be accomplished by turning off the feature toggle used to activate the new release.

Roll Forward

- *Roll forward* means fixing the problem and generating a new version. This generally requires you to debug the problem and then be able to test and deploy the new version quickly.

Discussion questions

1. How many past interfaces should a service support?
2. During a rolling upgrade, how does the system decide which instance to replace?
3. How would a tool that automatically converts from one database schema to another be used?
4. What is the cost difference between canary testing and A/B testing?