DevOps

Continuous Integration

The Importance of Time To Market

- Nowadays, delivering products and features **fast** is crucial / vital for companies.
- Business has new ideas and needs every day/week.

Observations

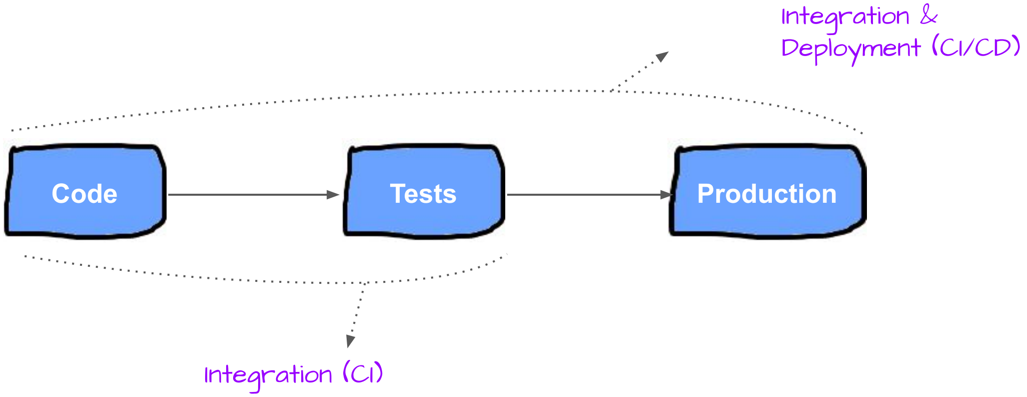
- Amazon, Google, Netflix, ... deploy applications thousand times a day in Production.

- No regression, no downtime.

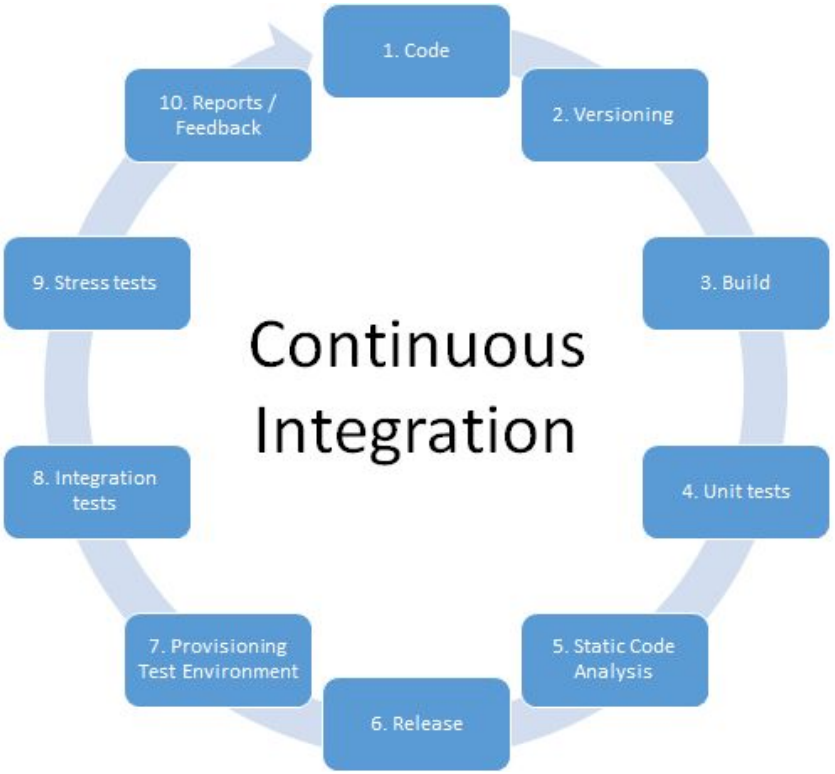
- Google’s example:
    - 18,000+ developers
    - Billions lines of code

How to Achieve that ?

- Deliver small increments.
- Re-use proven code whenever possible
- Test every single piece of programs.
- Automate every step.
- Delivering must be designed as a reliable pipeline.



Continuous Integration



1. Code

2. Versioning

3. Build

4. Unit tests

5. Static Code Analysis

6. Release

7. Provisioning Test Environment

8. Integration tests

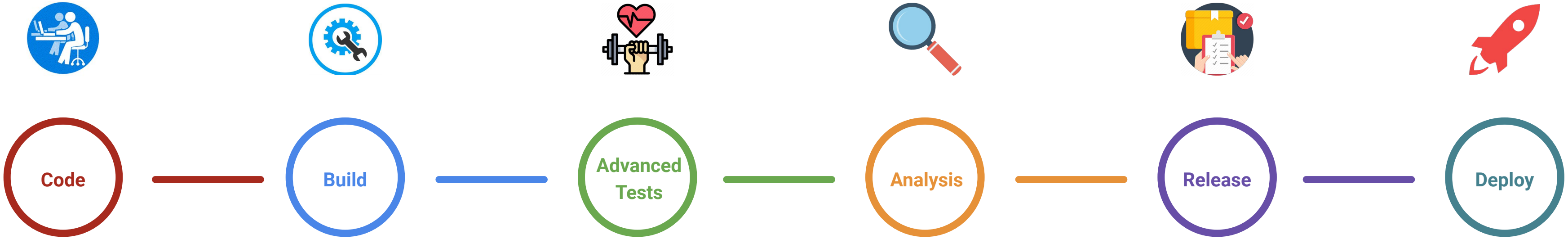
9. Stress tests

10. Reports / Feedback

Designing a CI/CD Pipeline

1. All the operations should be repeatable.
   - Code driven
   - Apply the same operations on each environment.
2. Easy to trigger
   - Either by clicking or creating a Pull Request
3. Checks the application builds correctly
   - Compilation
   - Unit tests
   - Packaging
4. Checks the application works with other services
   - Integration tests
5. Checks the application’s performance
   - Stress tests
6. Publish static code analysis reports
7. Versions and saves each “stable” component
8. Prepare the runtime
   - Installs required dependencies on the targeted environment
9. Deploys the application in the targeted environment

Designing a CI/CD Pipeline



Code

Versioned Code

Includes the application code, tests and configurations

Build

Compilation & Unit tests

Checks the application’s global integrity and that specific function return expected results

Advanced Tests

Integration & Stress Tests

Checks services interoperability and performance.

Analysis

Static Code Analysis Code Review

Checks code for vulnerabilities, bad practices, generate reports.

Release

Version & Save Artifacts

Tag the built artifact. Save it in a repository.

Deploy

Environment Provisioning

Install & configure the environment.

The importance of testing an application

- Process suite to make sure we don’t deliver applications containing (big) bugs, regressions, security issues, ...

- Many kind of complementary tests :
    - Unit tests
    - Integration tests
    - Stress tests & Load tests
    - QA

- The cost of fixing a bug is larger if testing is not done in early stages

Testing - Unit Tests

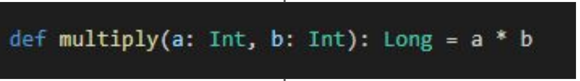
- Small piece of code, running very fast and checking a specific function’s behavior.

- May use mocks (ie: testing a function which accesses a third party service).

- Must cover all cases, even unexpected but possible scenarios.

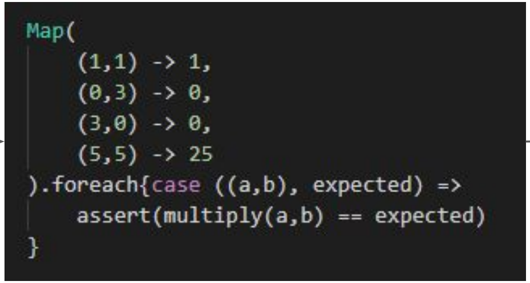
- In order to test specific methods, it is very important to design modular applications.

Testing - Unit Tests



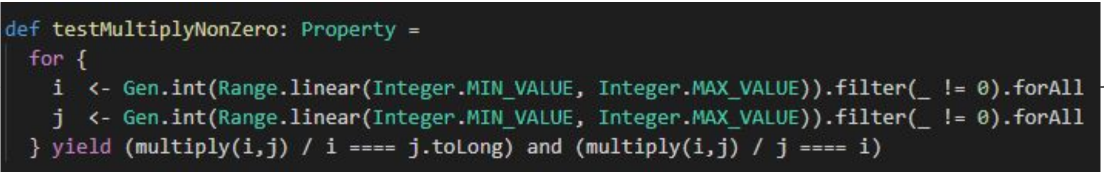
```
def multiply(a: Int, b: Int): Long = a * b
```

Example-based Testing



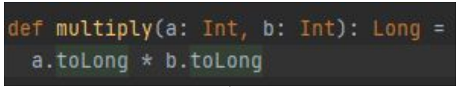
```
Map(
    (1,1) -> 1,
    (0,3) -> 0,
    (3,0) -> 0,
    (5,5) -> 25
).foreach{case ((a,b), expected) =>
    assert(multiply(a,b) == expected)
}
```

Property-Based Testing



```
def testMultiplyNonZero: Property =
  for {
    i  <- Gen.int(Range.linear(Integer.MIN_VALUE, Integer.MAX_VALUE)).filter(_ != 0).forAll
    j  <- Gen.int(Range.linear(Integer.MIN_VALUE, Integer.MAX_VALUE)).filter(_ != 0).forAll
  } yield (multiply(i,j) / i ==== j.toLong) and (multiply(i,j) / j ==== i)
```

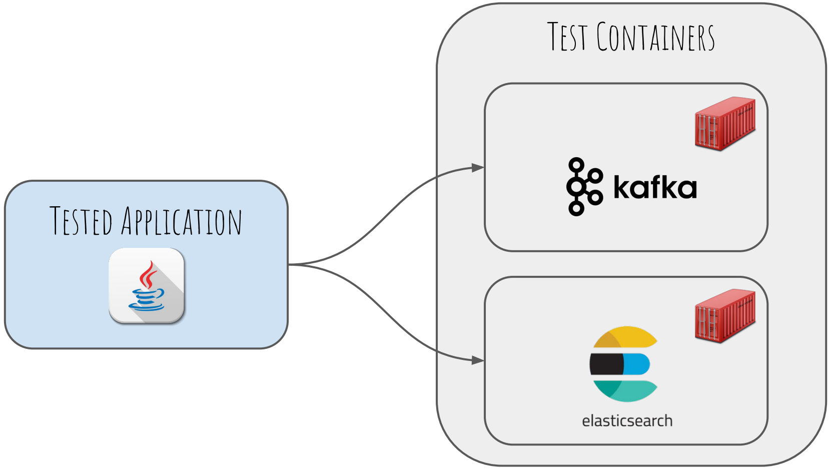
Overflow Bug discovered and fixed



```
def multiply(a: Int, b: Int): Long =
  a.toLong * b.toLong
```

Testing - Integration Tests

- Test end-to-end features / user experience scenarios.
- Run on localhost.
- Should not exceed a few minutes.
- May rely on ephemeral containers



1. Create Containers
2. Prepare Environment
3. Run the program
4. Check assertions
5. Stop Containers

Testing - Performance Tests

- The aim is to benchmark an application :
  - Measure response times
  - Ensure the system behaves correctly even when many users are connected
  - Ensure there are no memory leak
- Scenarios describing users behavior
  - Multiple requests in the same time
- May take time to run. Usually run when releasing a significant increment.

Testing - Performance Tests

- Multiple ways to address the problem:
  - Baseline Testing: You have technical specifications => Ensure they are respected (ie: N parallel users with expected performance)
  - Load Testing: You want to measure your system's performance. You should start small and add sessions gradually.
  - Stress Testing: You want to find the breaking point of the system.

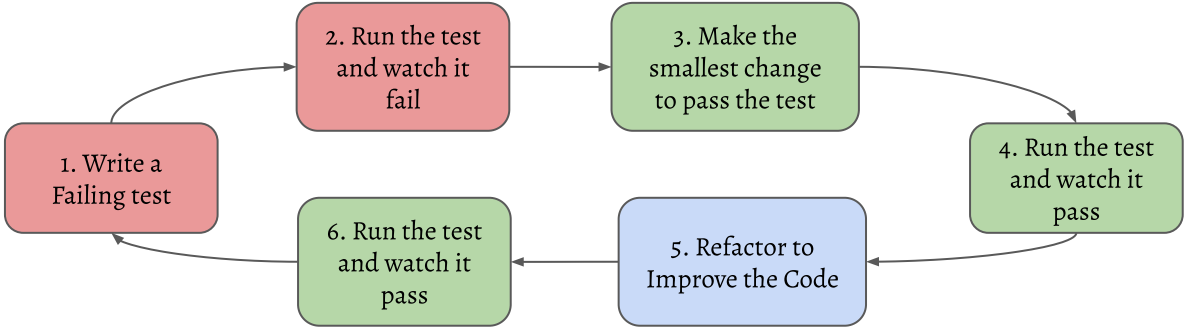
- You have to understand how the application is used (by real users) and inject data ideally from production (if any).

- The stress conditions (environment) should be realistic. Configure servers with Production specifications and build "real" users sessions.

- No need to absolutely break the systems ! There are tools for that.

A word about TDD

- Modern approach consisting in **writing tests before implementing** the Application code (Test First Design)
- 3 laws:
  - You must write a failing test before implementing Production code.
  - You must write one assertion at a time.
  - You must write a minimal code to make the test succeed.


1. Write a Failing test
2. Run the test and watch it fail
3. Make the smallest change to pass the test
4. Run the test and watch it pass
5. Refactor to Improve the Code
6. Run the test and watch it pass

Testing - Bias

Packaging

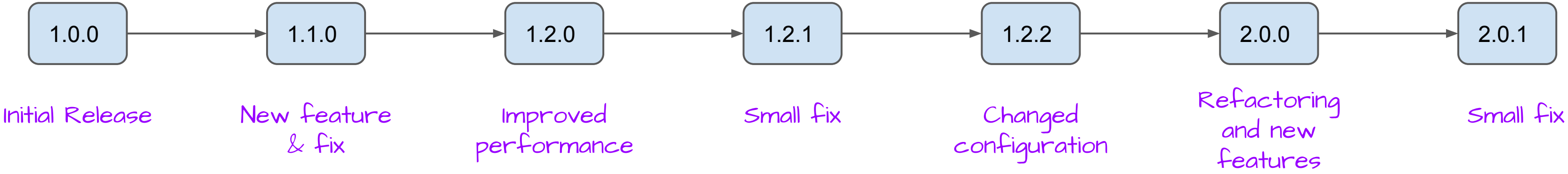
- Once a program is validated (ie: built and tested), we want to **assemble** it in order to make a **deliverable**, ready to deploy and run.

- Usually :
    - a simple file (bash, python script)
    - an archive (tar, jar, zip …)
    - an image (docker, VM)

Versioning artifacts

- Versioning (aka Tagging) assembled artifacts allows us:
  - To easily refer to a specific snapshot (commit) of the application.
  - To use it multiple times if needed, without re-building it each time.

- Usually, we choose understandable names by convention, composed with:
  - A Major number, incremented on big releases (breaking changes, lots of new features).
  - A Minor number, incremented when adding a few functionalities.
  - A Patch number, incremented when fixing bugs.



1.0.0 → 1.1.0 → 1.2.0 → 1.2.1 → 1.2.2 → 2.0.0 → 2.0.1

Initial Release

New feature & fix

Improved performance

Small fix

Changed configuration

Refactoring and new features

Small fix

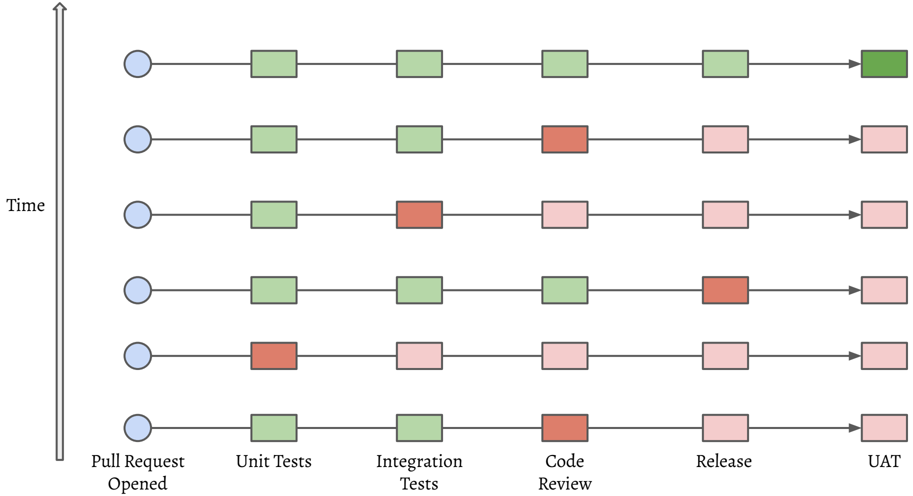
- Versioned application artifacts are then stored in repositories / registries

Quality & Frequency

- How to define efficiency ?
  - Number of commits per day ?
  - Number of releases / unit of time ?

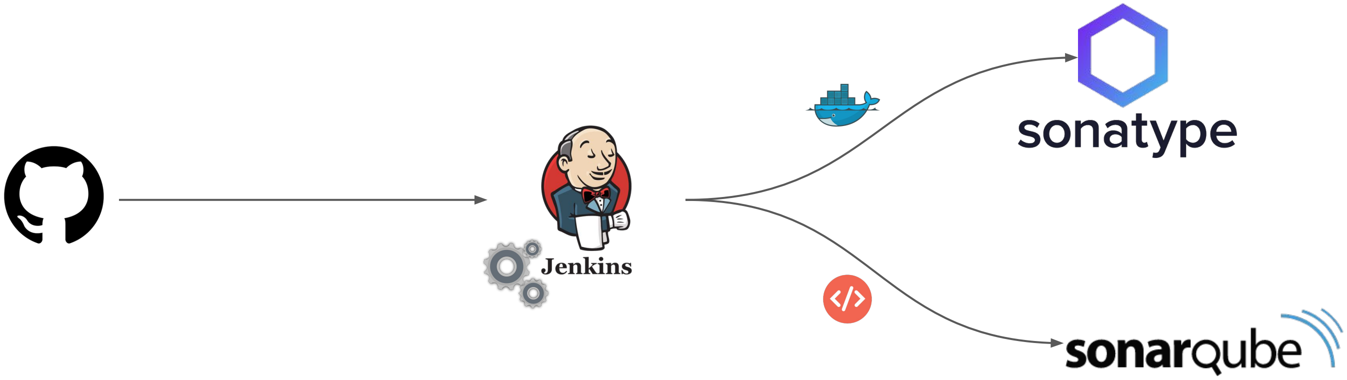
- Driven by speed and quality.

  A good process is expected to :
  - Be reliable : We don't push low quality code to production.
  - Be clear for all parties.
  - Compress lead times.
  - Bring value to each step.



Exercices

Let’s Build CI Pipelines


Jenkins
sonatype
sonarqube