Decentralized Systems Engineering

CS-438 – Fall 2025

DEDIS

Pierluca Borsò-Tan

EPFL

Credits: Netflix, DynaTrace, Wikimedia Commons

Software Quality for Decentralized and Distributed Systems

Or: How I Learned to Stop Worrying & Love the Tests

Roadmap for the (intense) day

- Why You Should Care
- Managing Quality & Software Dev Lifecycle
- Software testing – basic principles
- Testing distributed & decentralized systems
- Chaos engineering
- Developing a testing & evaluation strategy (worked out examples)
- Testing & Evaluation Tools in Go

Why You Should Care

What does a bug look like to you?

Say, an integer overflow

June 4, 1996

Of the Need to Manage Defects
A $370 million example: Ariane 5 (1996)

Converting a 64-bit float into a signed 16-bit integer.

Of the Need to Manage Defects
A deadly example: Therac-25 (1980s)

A race condition caused patients to receive 100x radiation dosage.

Of the Need to Manage Defects

A sinister example: Fujitsu Horizon (2000-2020)

Fujitsu's *Horizon* software said UK Post Office staff were stealing money.
Bugs led to lawsuits, 700+ people found guilty.
Dozens were sent to jail.
Some faced bankruptcy, others committed suicide.

https://en.wikipedia.org/wiki/British_Post_Office_scandal

Quality – a definition

What does it mean in business, engineering & manufacturing ?

According to Wikipedia:

- the non-inferiority or superiority of something
- being suitable for its intended use (fitness for purpose)
  while satisfying customer expectations.

Quality – a simpler view

- Building the right thing, satisfying customer needs

  → partially out of scope today

- Building the thing right, to specification & within tolerance

- Preventing defects and keeping costs under control

When should you start thinking about quality in a project?

Software Quality – How ?

- Tools

  IDE autocompletion, static analysis / linters, testing, CI/CD, etc.

- Processes

  Code review, pair programming, test-driven development, etc.

- Measurements

  Software complexity, code coverage, etc.

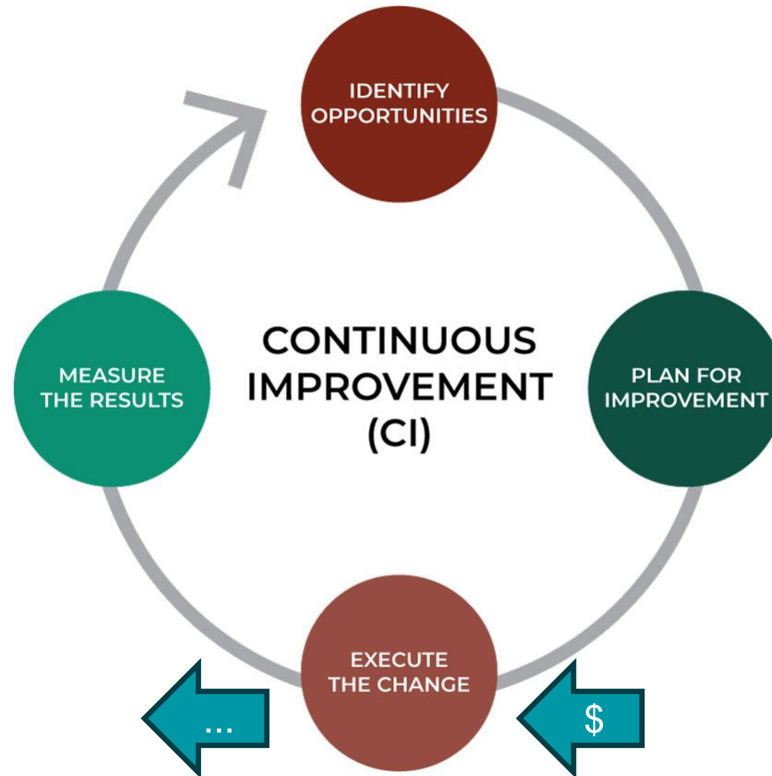
- Documentation

  Requirements documents, code style, traceability matrix, etc.

Software Quality

As continuous improvement

And investment



Software Quality – Summary

- A way to ensure engineering leads to a “good” solution
    - Functionally
    - Structurally

- A consideration throughout the project lifecycle

- An (empowering) constraint on the development

- An investment to ensure success

Managing Software Quality

Managing Software Projects

Like any project, need to:

- Manage limited resources
- Keep costs under control
- Manage risks (i.e. bad stuff happening)
    - Reduce probability
    - Reduce impact
- Quality supports this !

Software Development Lifecycle
A reminder

Quality and Cost – A Balancing Act

- Quality assurance measures are not free !
- Defects are even more expensive !
- What’s the trade-off ?

Quality and Cost
A balancing act

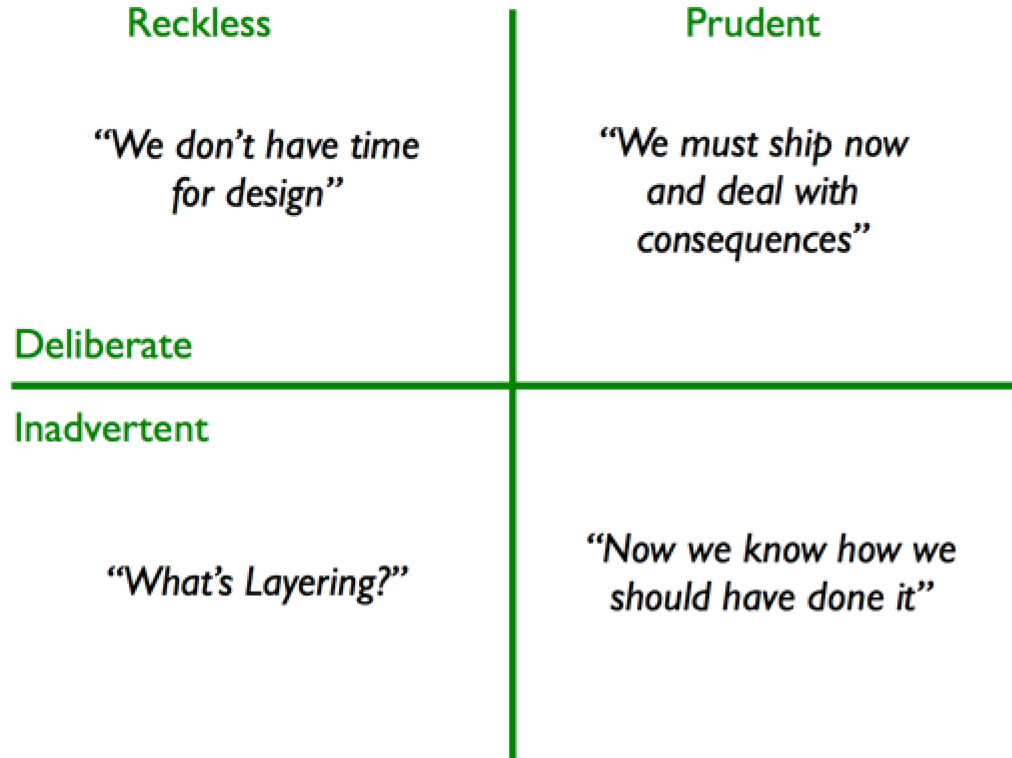
Cost of fixing bugs at various stages of software delivery

Relative cost of bug to fix

Requirements	Design	Development	Testing	Staging	Production
< 1x	1x	10x	25x	50x	150x

Quality and Cost
Technical debt

Quality and Cost
Technical debt



Requirements & Specifications

- Functional requirements

  e.g. “The user can reset a forgotten password”

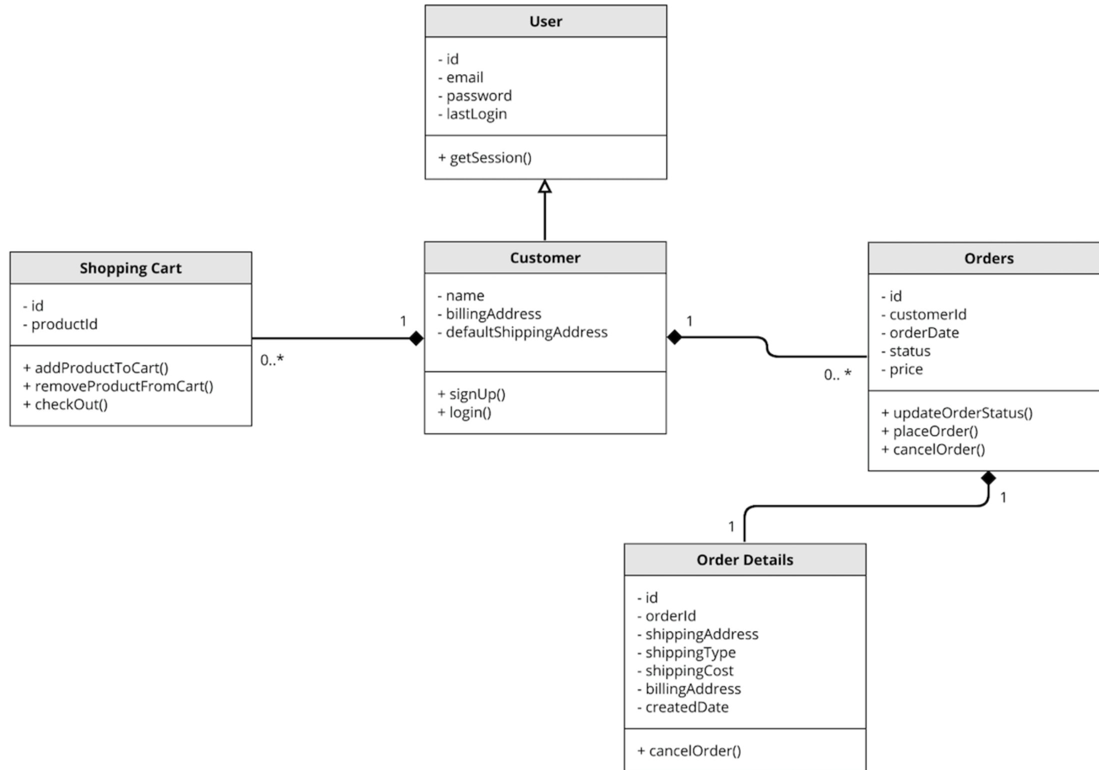
- Non-functional requirements

  e.g. “The system should respond to 99% of the requests within 100ms”

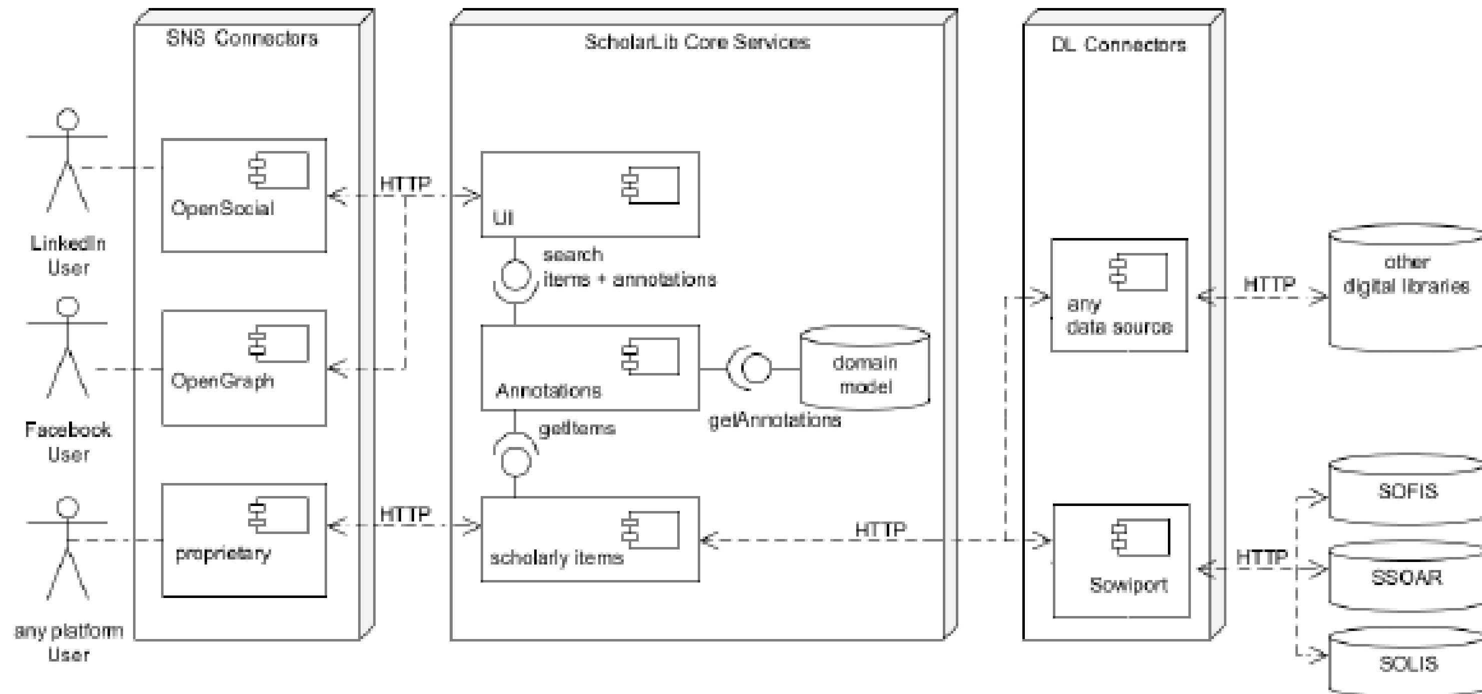
- Quality impact:
  - Shared understanding of the product
  - Allow detection of “specification” bugs
  - Informs the system design
  - Informs the security of the system (threat model)

Architecture & Design

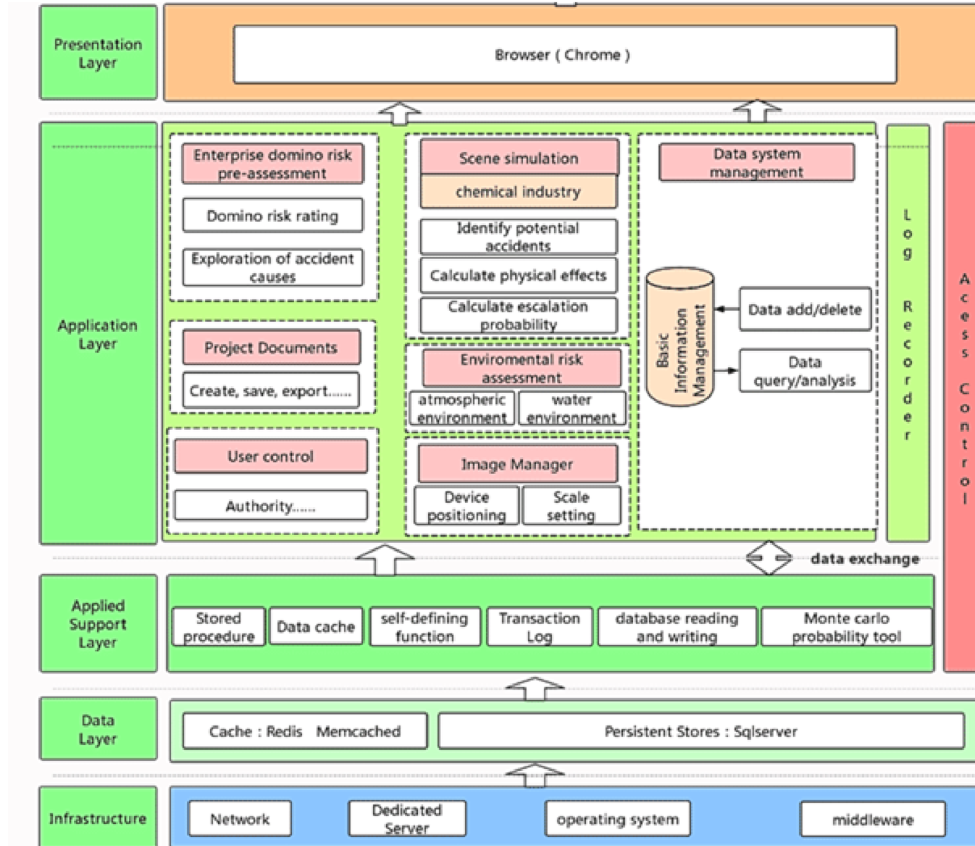
Architecture & Design
UML ?



Architecture & Design
Component diagram?



Architecture & Design
Layered ? Ad-hoc ?



Why & When is architecture important for quality ?

Architecture & Design

- Shared understanding of the software
- Easier to reason about than code
- Reduces complexity
- Divide & conquer
- *Manage risk (throughout the lifecycle)*

Implementation – Why Clean Code Matters

- Code should be written for readability
- Reviewers will detect bugs more easily
- Automatic tools will understand it better
- Maintainers will find it easy to modify
- Quality impact:
  → ***A bad implementation will kill your project.***

Testing – Because your code is buggy (& mine too)

- Detect bugs at all levels (unit, integration, system, etc.)

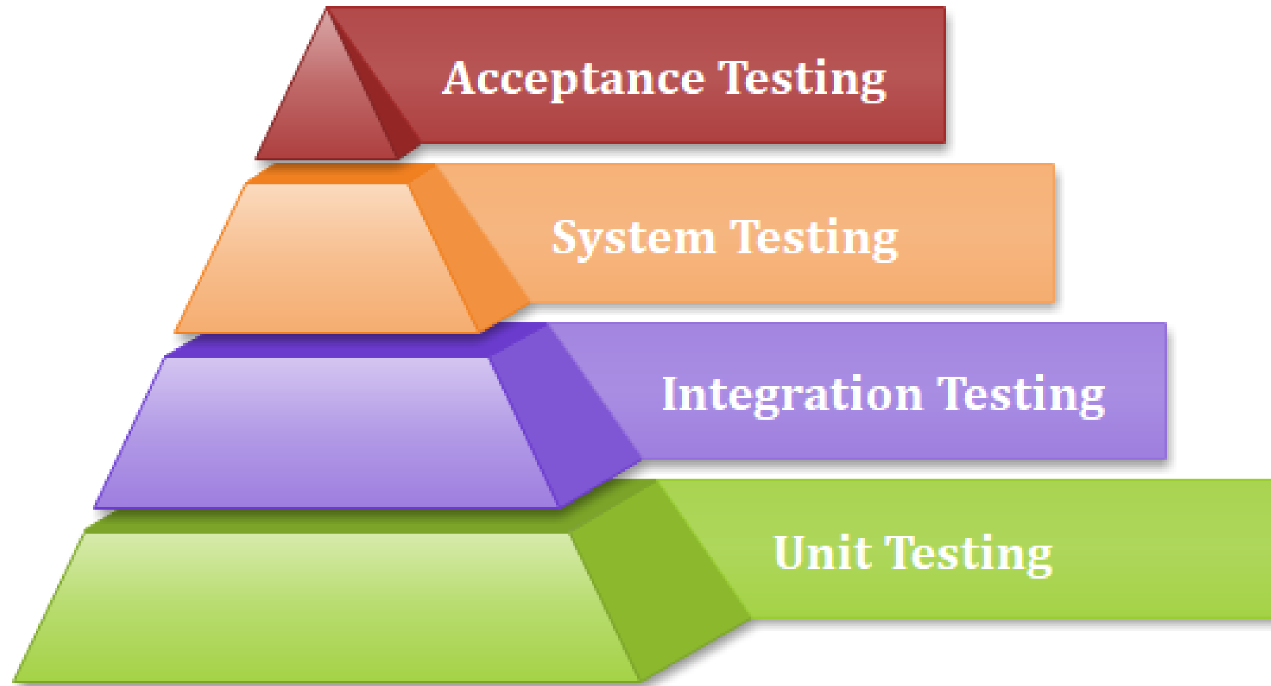
- Detect defects with respect to specification

- What else is testing good for ?
  System documentation !

- Quality impact:
  → ***Limited, poor or absent testing will cost you down the road.***

... and it will cost A LOT !

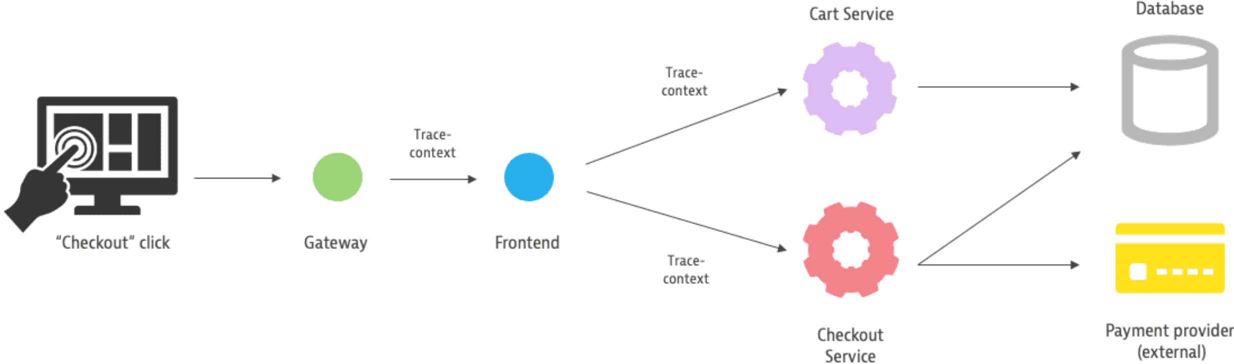
Testing
The Pyramid

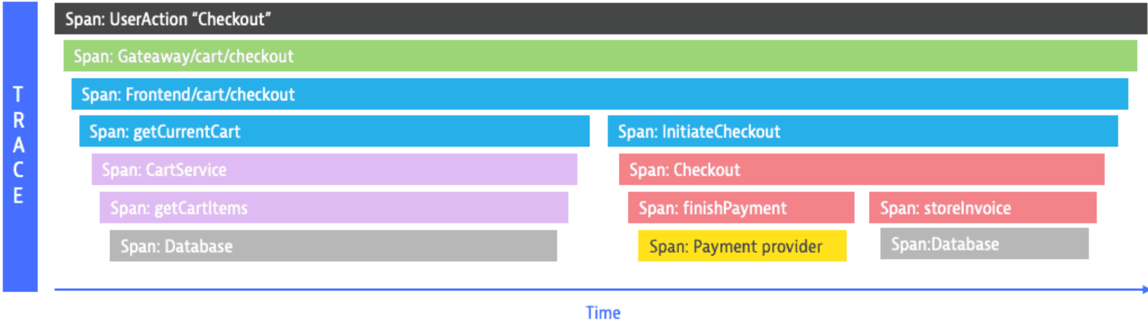


Deployment – Because your code is still buggy !

- Ensure repeatability and consistency
- Detect defects in production, ideally before the users !
- Observability – making the application behaviour “visible” in production
  - Metrics
  - Logs
  - Stack traces
  - Traces
- If something goes wrong, this is your debug information !

Observability – Traces





Managing Software Quality – Summary

- Each software lifecycle stage impacts the subsequent ones

- Prevent defects whenever possible
- Detect them as soon as possible

- Perverasive concerns: traceability, repeatability

- Nobody likes documentation, but it’s fundamental

- It’s an investment, *always* evaluate its return

Testing and Breaking Stuff !

Testing basics

- A test is a way to determine if an artifact meets its requirements
  - A function's API
  - A system's specification
  - A given user experience
- Along a specific axis:
  functionality, performance, security, resilience, etc.
- Types
  Unit, integration, system (end-to-end), user acceptance, exploratory, smoke, staging (pre-production), A/B testing (in production), etc.

How are developers and testers different ?

Psychology of testing – mindset matters

- Developer

  wants to see things work, build them

  focuses on what the *system* should be doing

  → *solution*-oriented work

- Tester

  wants to break things

  focuses on what the *user* expects

  → *problem*-oriented work

The (Manual) Test Case

Documenting the *what* with the *how*

Test Scenario ID	Login-1	**Test Case ID**	Login-1A
Test Case Description	Login – Positive test case	**Test Priority**	High
Pre-Requisite	A valid user account	**Post-Requisite**	NA

Test Execution Steps:

S.No	Action	Inputs	Expected Output	Actual Output	Test Browser	Test Result	Test Comments
1	Launch application	https://www.facebook.com/	Facebook home	Facebook home	IE-11	Pass	[Priya 10/17/2017 11:44 AM]: Launch successful
2	Enter correct Email & Password and hit login button	Email id : test@xyz.com Password: ******	Login success	Login success	IE-11	Pass	[Priya 10/17/2017 11:45 AM]: Login successful

Automated Tests – the “right” way

- Reproducible (not “flaky”)
- Isolated (testing one thing only)
- Independent from each other
- Self-contained
- Same code quality as production code

→ Tests are an investment, aim for a good return-on-investment !

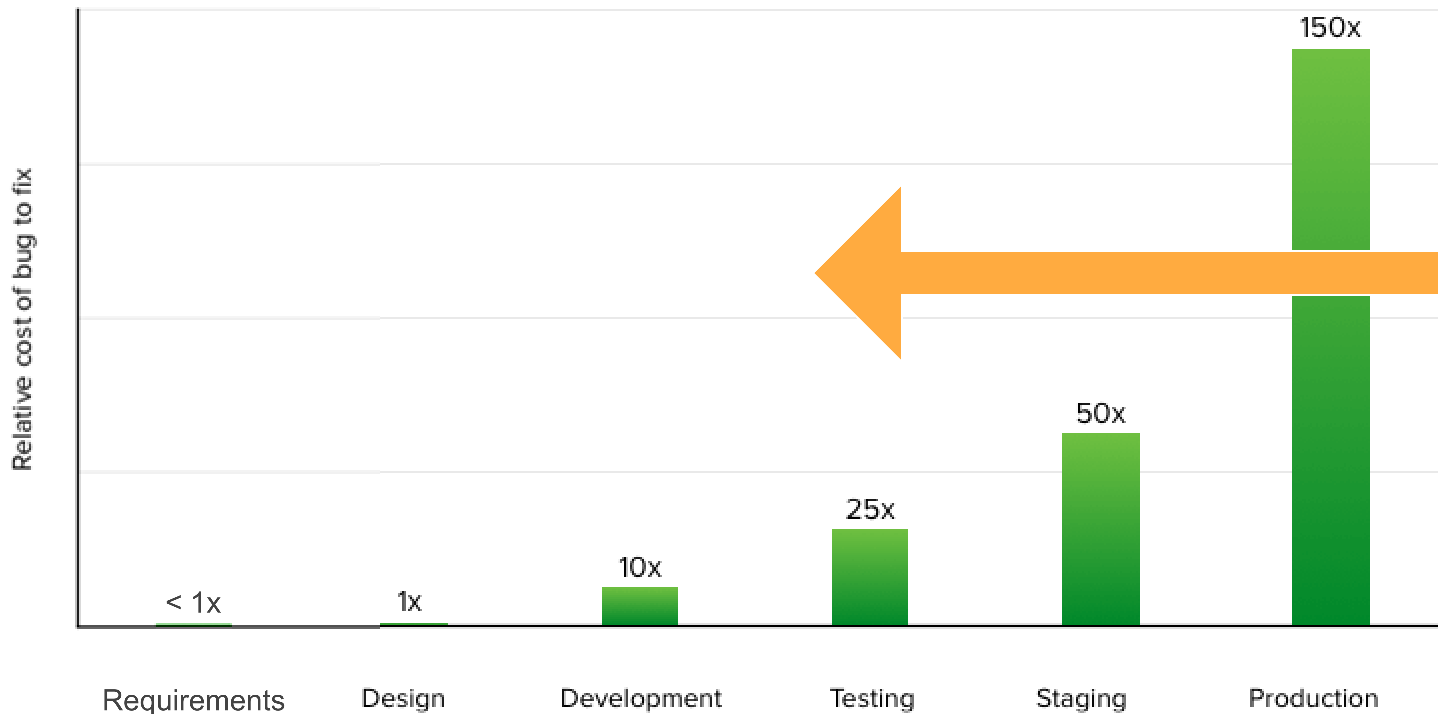
Manual vs Automatic testing
A clear trade-off

	Manual testing	**Automatic testing**
Investment (creation)	Low	Medium to Significant
Investment (maintenance)	Very low	Depends on code quality, tooling cost, requirements change, ...
Investment (execution)	Massive	Minimal, unattended
Flaws	Lower reliability, human errors	Sometimes nearly impossible, test code can also have bugs,
Unique advantages	Human adaptability, user perspective	Immediate results, reusability,
Practical for	What do you think?	

Naive test case automation
What if replaced testers with computers?

- Hard to develop
- Hard to debug
- They come in too late in the development !

Quality and Cost
A reminder



Metrics

if you can't measure it, you can't improve it!

- Code coverage
- Mutation coverage
- Code complexity
- Historical bug location
- Automatic ratings
- Risk
- Derived metrics, e.g. complexity vs. coverage

Review of basic concepts

- Unit test
- Integration test
- End-to-end test

- Mocking
- Dependency Injection

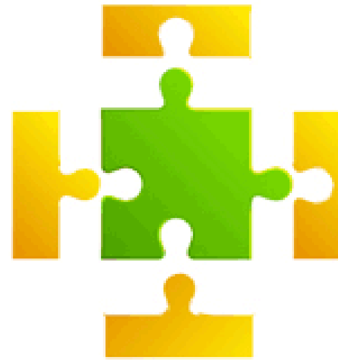
- Fuzzing
- Property-based tests
- Formal verification
- Model checking

Unit testing
Dealing with coupling

REAL SYSTEM

Green = class in focus
Yellow = dependencies
Grey = other unrelated classes

CLASS IN UNIT TEST



Green = class in focus
Yellow = mocks for the unit test

Unit testing – Outcomes

- Find defects early
- A failing unit test makes the defect obvious
- Better functional / class design
- Reduction of complexity (hard to test)
- Increased code coverage

Mocks, Stubs, Fakes
A word about terminology

A function/object/etc. could be mocked in a number of ways

- Stub - pure data
- Mock - data and calls
- Fake - fake implementation
- Spy - real object, extended with partial mocking
- Dummy - necessary object, unused in tests

Language Tooling
How to make this work

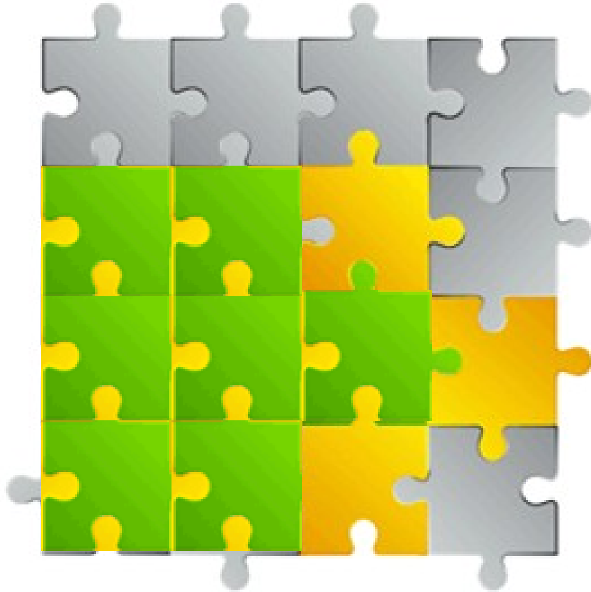
Concepts span all languages, but each has its own tooling.
A few selected tools:

1. Go — `testing`, `testify/require`, `gomock`, `gremlins`
2. Java — `JUnit`, `Mockito`, `JMock`, `PITest`, etc.
3. Scala — `ScalaTest`, `Mockito`, `Stryker`, etc.
4. JavaScript — `jest` … or `jasmine`, `mocha`, `sinon`, etc.

... and sometimes you might have to roll your own

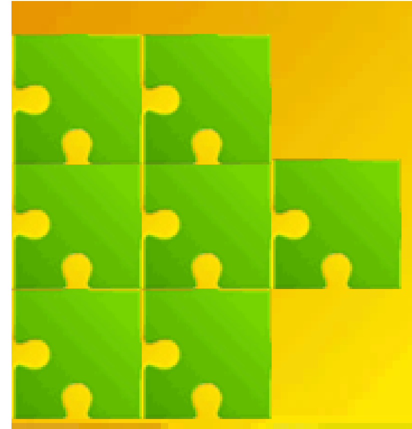
Integration testing
Dealing with coupling

REAL SYSTEM



Green = class in focus
Yellow = dependencies
Grey = other unrelated classes

INTEGRATION TESTS



Green = code under test
Yellow = test support (mocks?)

Integration Testing
Key points

- “Integration” can happen at all scales (classes, modules/packages, ...)
- Tools depend on the job: no silver bullet
    - like unit tests
    - like end-to-end (E2E) tests
    - custom
- Typical mocks when testing business logic:
    - database
    - network
    - external APIs
    - filesystem

Integration Testing
A few examples

1. Testing how 2-3 classes work together
2. Web: testing the user interface without a backend
3. Server: testing an API endpoint (without a database)
4. Distributed: testing interaction between a few systems
5. PoP: testing the processing of incoming messages

End-to-End Testing
Key points

- “End-to-end”, because it involves the whole system (-ish)

- Tools are completely dependent on the test objective, the system under test and the domain

    embedded? browser-based? server infrastructure ?

    dedicated testing tool? bash script?

- Automates “using the software”

- Generally:
    - the slowest tests
    - detect defects, but not their cause

Property-Based Testing

Define a test as:

- A given input data ***shape***
- An operation on the input
- A given set of expectations

The test runner will:

1. generate the data
2. try to prove the expectations wrong

Property-Based Testing
An arithmetic example

```
// function under test
int add(int a, int b) {
    return a+b;
}
```

```
@Property
boolean sumIsCommutative(
  @ForAll int a, @ForAll int b
) {
    return add(a,b) == add(b,a);
}
```

```
@Property
boolean sumIdentity(@ForAll int a)
{
    return add(a,0) == a;
}
```

```
@Property
boolean sumIsAssociative(
  @ForAll int a, @ForAll int b, @ForAll int c
) {
    return add(add(a,b),c) == add(a,add(b,c));
}
```

```
@Property
boolean sumTwiceIsMultiplication(@ForAll int a)
{
    return add(a,a) == 2*a;
}
```

```
@Property
boolean sumOneIncreases(@ForAll int a)
{
    return add(a,1) > a;
}
```

Question time !

So, what kind of tests shall we start with and why ?

Distributed & Decentralized Testing

Basic Good Practices

Mature Testing System

- At all levels: unit, integration, system

Separate Environments

- Development
- Staging
- Production

Full Observability in Production

- Centralized log
- Metrics
- Traces

Distributed System Tests
So many parameters ...

- What is the testing objective?
- What are the parameters we want to control?
- How do you make the tests reproducible?
- How would you do it in practice ?

Case Study – A distributed, embedded system

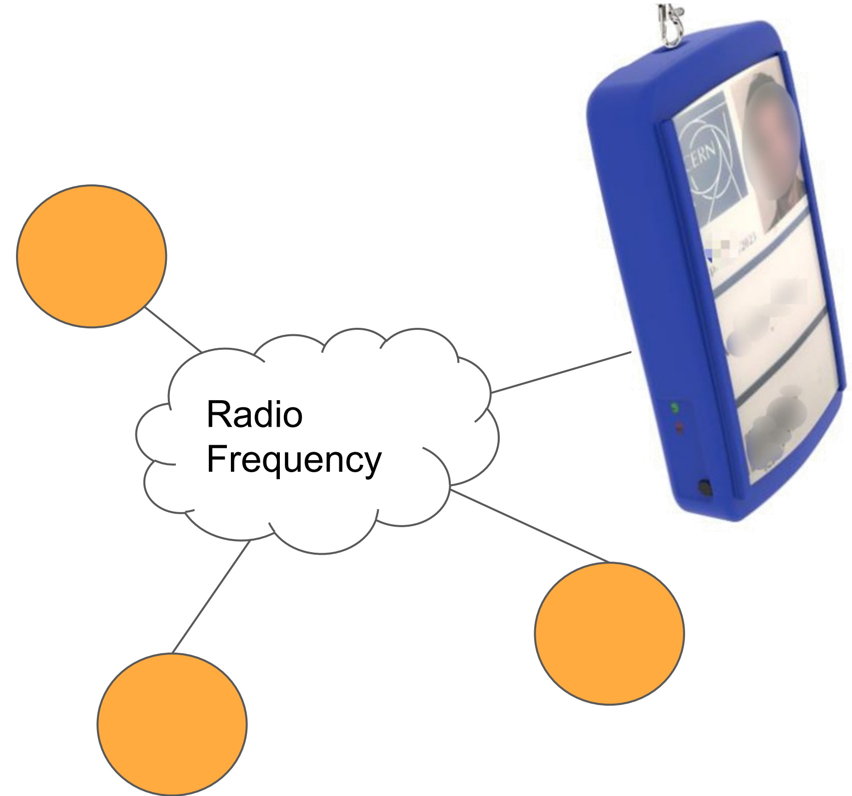
Covid Contact-Tracing System

Specification:

1. Radio-based distance measure
2. Support up to 16 devices

Implementation:

1. Separate HW boards
2. Finite State Machine-based communication protocol



Challenges in distributed / decentralized systems

- The network is an uncontrolled variable
- (Virtually) infinite number of states
- Failures can happen at any layer (network, HW, OS, application, ...)
- Many software versions may coexist
  - Backward compatibility
  - Forward compatibility
  - Application invariants across versions
- Subtle environment differences can mask issues

Jepsen – Testing Distributed Storage Systems

- Full suite of tools to evaluate distributed systems
  They've "broken" dozens of major, well-known systems

Approach:

- They test real systems, running on real clusters (1-4 months of work)
- They test under failure modes: faulty networks, unsync'd clocks, etc.
- They make abundant use of generative testing:
  - → apply (many, many) random operations to the system
  - → build a "concurrent history" of the results
  - → check history against a model to ensure correctness / verify invariants

https://jepsen.io/

Testing (Permissioned) Blockchain Nodes

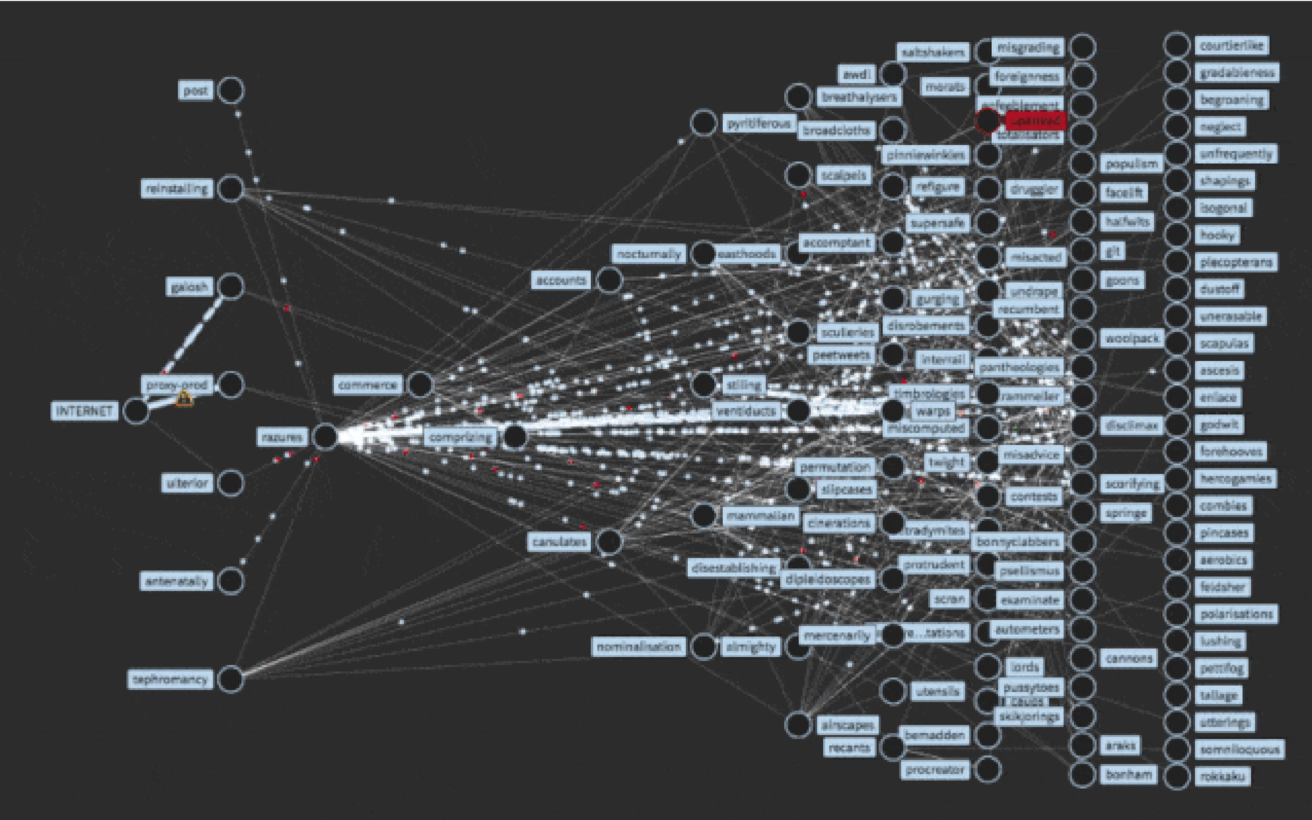
Unique challenges:

- Node robustness during upgrades ?
- Can old and new versions communicate without failures ?
- Smart contract determinism across versions ?
- Impact of failures ?

Solutions:

- Formal verification (where feasible)
- Testing latest version against baseline
- Testing mixed environments, including under failure scenarios
- Testing upgrade paths, including under failure scenarios

Case Study – Netflix



Case Study – Netflix

- Key metric: Stream “play” Per Second (SPS)
- Actual video data comes their content distribution network
- All the logic (incl. Stream play) comes from their microservices
- Hundreds of microservice clusters
  → how do you not break anything ?

Chaos Monkey & Chaos Kong

How can we ...

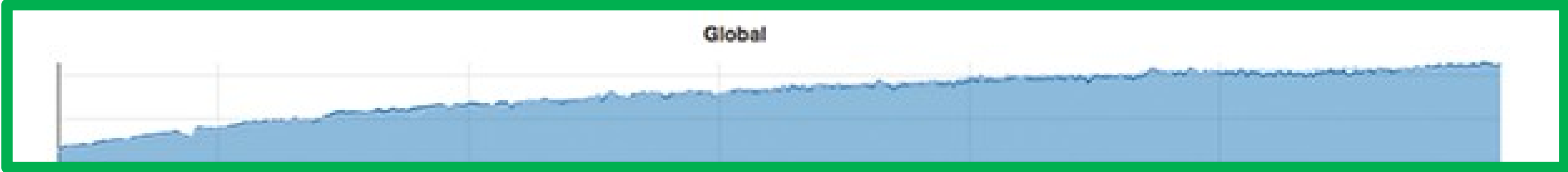
- test that a VM's unavailability has no consequences ?
  → Kill them !
  → Kill them at random !

- test that an application is resilient to a cloud region's unavailability ?

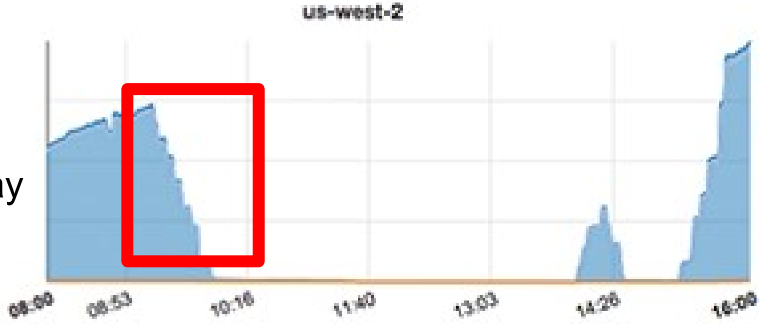
→ Kill it !
→ Run a "Chaos Kong" exercise !

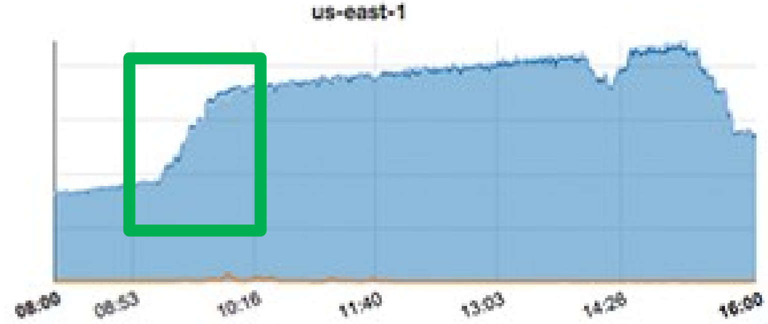
Chaos Kong – A Metrics Perspective

Video Play (SPS)



Video Play (SPS)





Beyond Chaos – Fault Injection Testing

- Chaos Monkey and Chaos Kong are limited by granularity

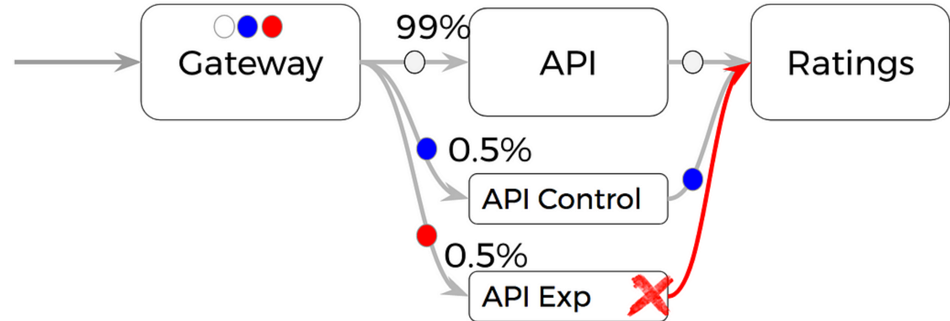
How can we test a hypothesis ?

- Redirect traffic to control/experimental group resources
- Choose % of traffic going to each

What kind of hypothesis ?

- Arbitrary failures !
  client, server, network, OS, ...

- Arbitrary scope !
  single VM, whole cluster, whole region, ...



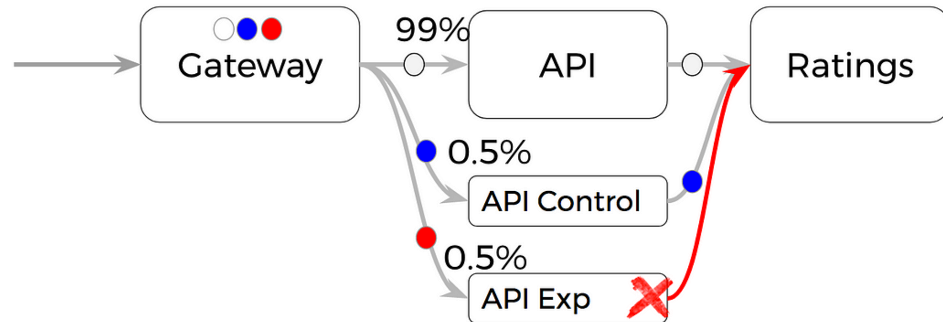
Beyond Chaos – Fault Injection Testing

How do you minimize the blast radius?

- If control/experimental group deviate, abort experiment automatically !
- Deviations are measured locally, upstream and globally !

Many non-trivial engineering considerations

- Assigning requests to groups
- Dealing with “retry” mechanisms
- Analyzing *only* experimental data
- How do we inject faults



Chaos Engineering

Goal:

- Run scientific experiments on production systems

Requirements:

- Strong observability of the system (= good monitoring)
- “Mature” testing environment (= moderately reliable software)
- Enough usage to measure a “steady state” in the system
  hypothesis: “steady state” will continue in both experimental & control groups
- Infrastructure-level separation between experimental and control groups

Then:

- Inject a failure in the experimental group and observe

Chaos Engineering – Advanced Principles

- Risk management: focus on likely / impactful events
- Run experiments in production
- Automate: experiments should run periodically / continuously
- Minimize blast radius: users should not be impacted !
- Want tools & resources ?
  https://github.com/dastergon/awesome-chaos-engineering