# Desmistificando Microsserviços e DevOps: Projetando Arquiteturas Efetivamente Escaláveis

Prof. Vinicius Cardoso Garcia
vcg@cin.ufpe.br :: @vinicius3w :: assertlab.com

[IF1004] - Seminários em SI 3
https://github.com/vinicius3w/if1004-DevOps

# Licença do material

Este Trabalho foi licenciado com uma Licença

Creative Commons - Atribuição-NãoComercial-CompartilhaIgual 3.0 Não Adaptada


Mais informações visite

http://creativecommons.org/licenses/by-nc-sa/3.0/deed.pt

AssertLab
Advanced Software and Systems Engineering Research Technologies

# The Deployment Pipeline

# Overall Architecture

A distributed system is one in which the failure of a computer you didn’t even know existed can render you own computer unusable.
— Leslie Lamport

# Introduction

- What are the structural implications of the DevOps practices?
  - both the overall structure of the system and techniques that should be used in the system's elements

- DevOps achieves its goals partially by replacing explicit coordination with implicit and often less coordination
  - the architecture of the system being developed acts as the implicit coordination mechanism

# Do DevOps Practices Require Architectural Change?

- If you must re-architect your systems in order to take advantage of DevOps, a legitimate question is "Is it worth it?"

- Some DevOps practices are independent of architecture,
- whereas in order to get the full benefit of others, architectural refactoring may be necessary

# Recall the 5 categories of DevOps practices

1. Treat Ops as first-class citizens from the point of view of requirements
   - Operations have a set of requirements that pertain to logging and monitoring
2. Make Dev more responsible for relevant incident handling
3. Enforce the deployment process used by all, including Dev and Ops personnel
   - Ensure a higher quality, avoids errors and the resulting misconfiguration
4. Use continuous deployment
   - Shorten the time between a developer committing code to a repository and the code being deployed
5. Develop infrastructure code, such as deployment scripts, with the same set of practices as application code

# Overall Architecture Structure

# Overall Architecture Structure

- Warm up
  - a module is a code unit with coherent functionality
  - a component is an executable unit

- Development teams using DevOps processes are usually small and should have limited inter-team coordination
  - integration and acceptance tests are mandatory

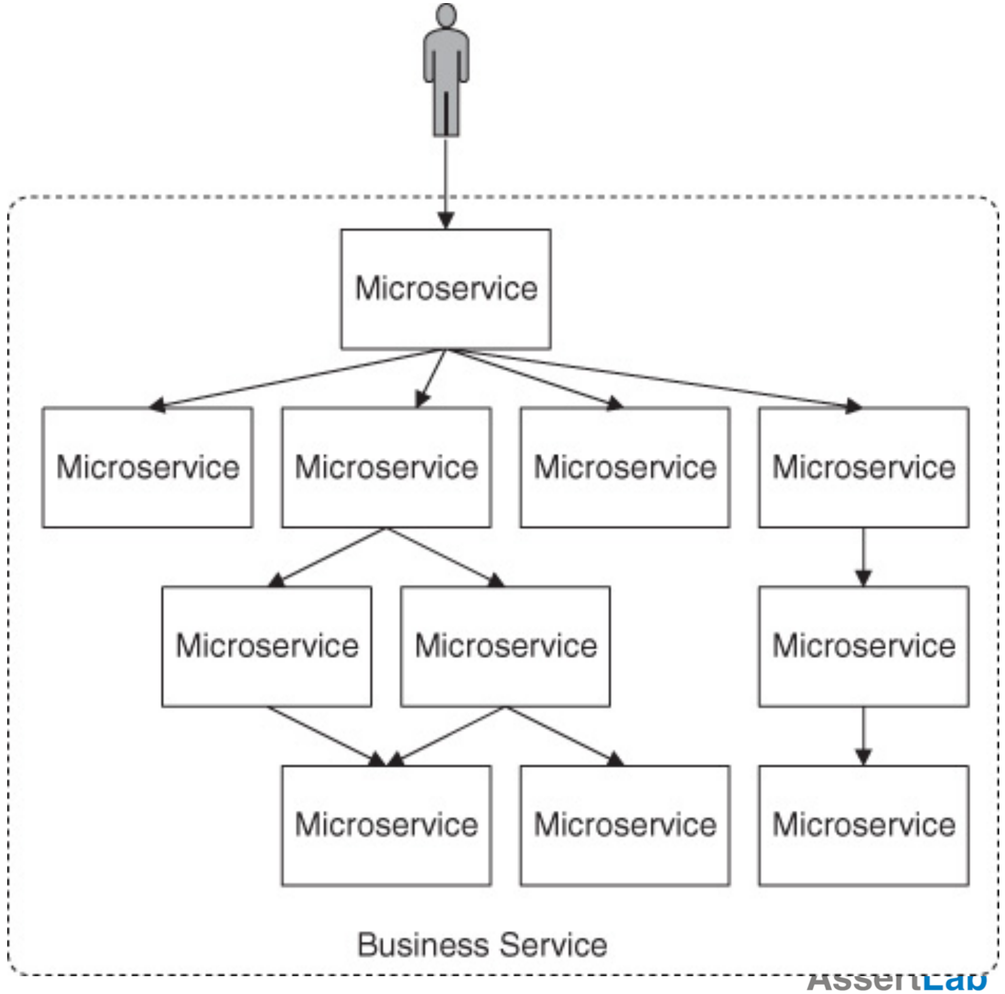
# Overall Architecture Structure

- An organization can introduce continuous deployment without major architectural modifications
  - Deploying without the necessity of explicit coordination with other teams reduces the time required to place a component into production.
  - Allowing for different versions of the same service to be simultaneously in production leads to different team members deploying without coordination with other members of their team.
  - Rolling back a deployment in the event of errors allows for various forms of live testing

- Microservice architecture is an architectural style that satisfies these requirements

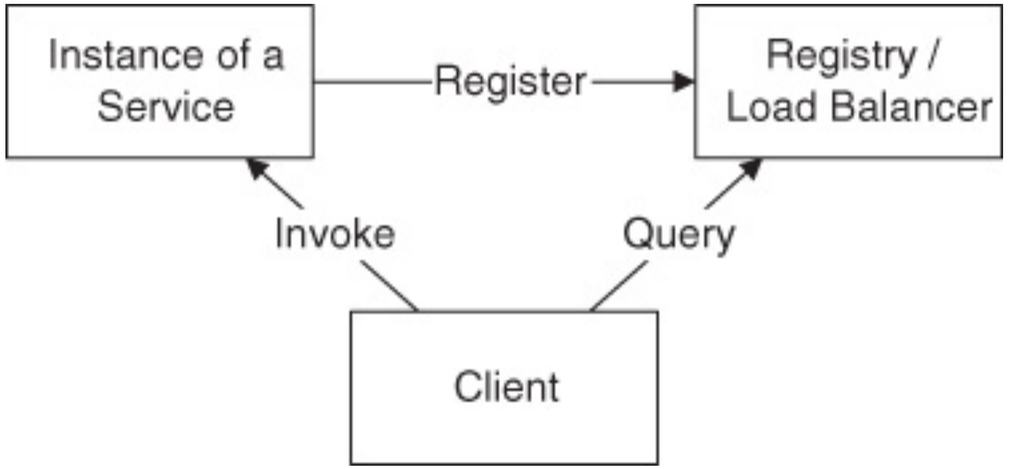
# Microservice Architecture

"A microservice architecture consists of a collection of services where each service provides a small amount of functionality and the total functionality of the system is derived from composing multiple services"



# Coordination Model

- If two services interact, the two development teams responsible for those services must coordinate in some fashion
  - How a client discovers a service that it wishes
  - How the individual services communicate?




- Netflix Eureka is an example of a cloud service registry that acts as a DNS server.
  - The registry serves as a catalogue of available services, and can further be used to track aspects such as versioning, ownership, service level agreements (SLAs), etc., for the set of services in an organization.

# Management of Resources

- Two types of resource management decisions can be made globally and incorporated in the architecture
  - provisioning/deprovisioning VMs
  - managing variation in demand.

# Provisioning & Deprovisioning VMs

- New VMs can be created in response to client demand or to failure
  - If the instances are stateless, a new instance can be placed into service as soon as it is provisioned
  - Similarly, if no state is kept in an instance, deprovisioning becomes relatively painless

- An additional advantage of a stateless service is that messages can be routed to any instance of that service, which facilitates load sharing among the instances.

# Provisioning & Deprovisioning VMs

- This leads to a global decision to maintain state external to a service instance (see lecture #3)
- Determining which component controls the provisioning and deprovisioning of a new instance for a service is another important aspect
  - A service itself can be responsible for (de)provisioning additional instances
  - A client or a component in the client chain can be responsible for (de)provisioning instances of a service
  - An external component monitors the performance of service instances (e.g., their CPU load) and (de)provisions an instance when the load reaches a given threshold

# Managing Demand

- The number of instances of an individual service that exist should reflect the demand on the service from client requests
  - Monitor performance
  - Use SLAs to control the number of instances

# Mapping Among Architectural Elements

- The final type of coordination decision that can be specified in the architecture is the mapping among architectural elements
  - Work assignments
  - Allocation

# Quality Discussion of Microservice Architecture

# Dependability

- Three sources for dependability problems are:
  - the small amount of inter-team coordination
  - correctness of environment, and
  - the possibility that an instance of a service can fail

# Small Amount of Inter-team Coordination

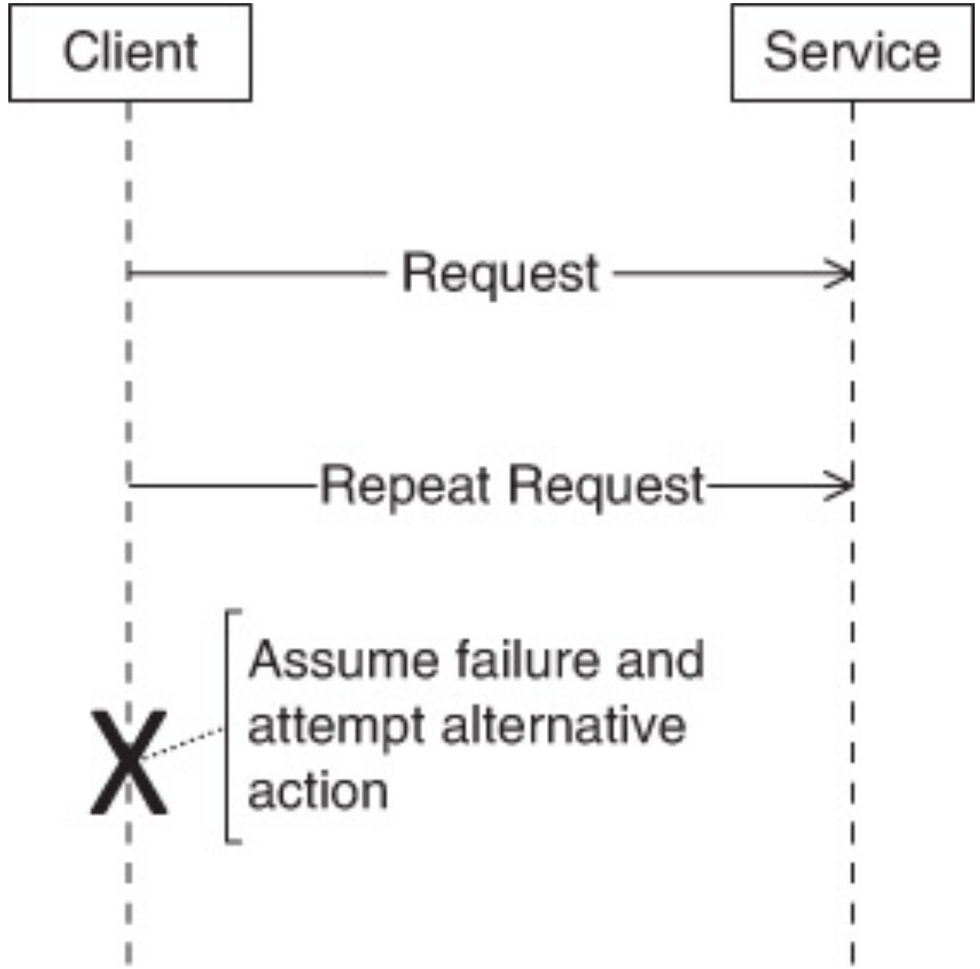
- May cause misunderstandings between the team developing a client and the team developing a service in terms of the semantics of an interface
  - Unexpected input to a service or unexpected output from a service can happen
    - defensive programming
    - integration and end-to-end testing [expensive to run these tests frequently]
    - Consumer Driven Contract (CDC)
      - The test cases for testing a microservice are decided and even co-owned by all the consumers of that microservice
      - Any changes to the CDC test cases need to be agreed on by both the consumers and the developers of the microservice

# Correctness of Environment

- A service will operate in multiple different environments during the passage from unit test to post-production
  - Errors in code and configuration parameters are quite common
  - Inconsistent configuration parameters are also possible
  - Due to a degree of uncertainty in cloud-based infrastructure, even executing the correct code and configuration may lead to an incorrect environment
- Thus, the initialization portion of a service should test its current environment to determine whether it is as expected
- An important trend in DevOps is to manage all the code and parameters for setting up an environment just as you manage your application code, with proper version control and testing

# Failure of an Instance

- Failure is always a possibility for instances
- Services should be designed so that multiple invocations of the same service will not introduce an error
- Idempotent is the term for a service that can be repeatedly invoked with the same input and always produces the same output —namely, no error is generated



# Modifiability

- Making a service modifiable comes down to making likely changes easy and reducing the ripple effects of those changes
- In both cases, a method for making the service more modifiable is to encapsulate either the affected portions of a likely change or the interactions that might cause ripple effects of a change

# Identifying Likely Changes

- The environments within which a service executes
  - A module goes through unit tests in one environment, integration tests in another, acceptance tests in a third, and is in production in a fourth
- The state of other services with which your service interacts
  - If other services are in the process of development, then the interfaces and semantics of those services are likely to change relatively quickly
- The version of third-party software and libraries that are used by your service
  - Third-party software and libraries can change arbitrarily, sometimes in ways that are disruptive for your service

# Reducing Ripple Effects

- Once likely changes have been discovered, you should prevent these types of changes from rippling through your service
- This is typically done by introducing modules whose sole purpose is to localize and isolate changes to the environment, to other services, or to third-party software or libraries
- The remainder of your service interacts with these changeable entities through the newly introduced modules with stable interfaces

# Amazon’s Rules for Teams

# Two pizzas

- All teams will henceforth expose their data and functionality through service interfaces
- Teams must communicate with each other through these interfaces
- There will be no other form of inter-service/team communication allowed:
  - no direct linking, no direct reads of another team's datastore, no shared-memory model, no backdoors whatsoever
  - The only communication allowed is via service interface calls over the network
- It doesn't matter what technology they [other services] use
- All service interfaces, without exception, must be designed from the ground up to be externalizable
  - That is to say, the team must plan and design to be able to expose the interface to developers in the outside world

# Microservice Adoption for Existing Systems

# Microservice Adoption for Existing Systems

- Operational concerns are considered during requirements specification
- The overarching structure of the system being developed should be a collection of small, independent services
- Each service should be distrustful of both clients and other required services
- Team roles have been defined and are understood
- Services are required to be registered with a local registry/load balancer
- Services must renew their registration periodically
- Services must provide SLAs for their clients
- Services should aim to be stateless and be treated as transient
- If a service has to maintain state, it should be maintained in external persistent storage
- Services have alternatives in case a service they depend on fails
- Services have defensive checks to intercept erroneous input from clients and output from other services
- Uses of external services, environmental information, and third-party software and libraries are localized (i.e., they require passage through a module specific to that external service, environment information, or

# For Further Reading

- For more information about software architecture, we recommend the following books:
  - Documenting Software Architectures, 2nd Edition [Clements 10]
  - Software Architecture in Practice, 3rd Edition [Bass 13]
- Service description, cataloguing, and management are discussed in detail in the Handbook of Service Description [Barros 12]
- The microservice architectural style is described in the book Building Microservices: Designing Fine-Grained Systems [Newman 15]
- The Netflix implementation of Eureka - their open source internal load balancer/ registry - can be found at https://github.com/Netflix/eureka/wiki/Eureka-at-a-glance
- Consumer Driven Contracts (CDCs) are discussed in Martin Fowler's blog "Consumer-Driven Contracts: A Service Evolution Pattern"