Anti-Pattern

Reinventing the wheel

Dare To Be Different

Monolith database

Hardcoded hell

Manual Configurations Management

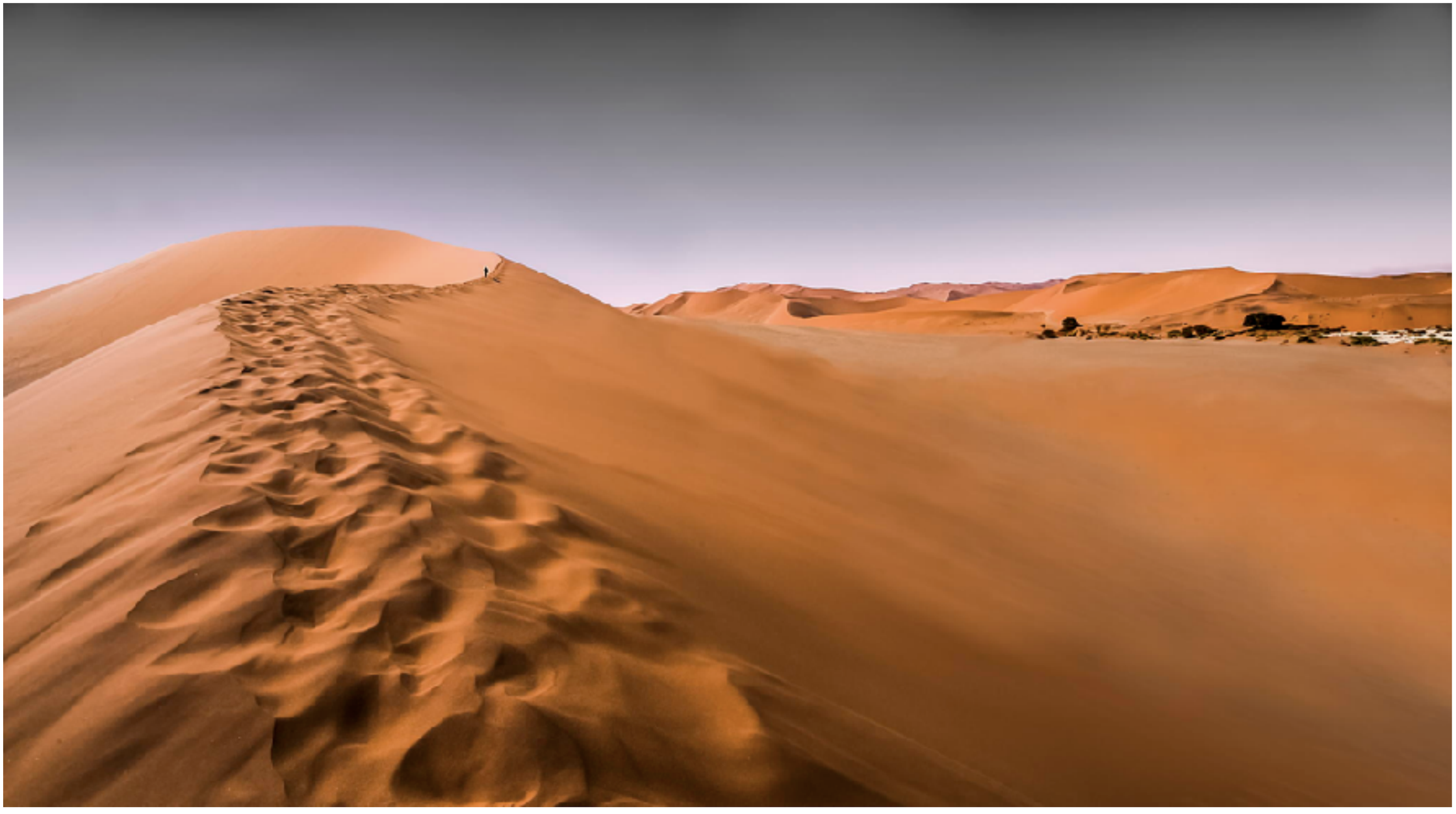
Flying Blind

Unknown caller

Dogpiles

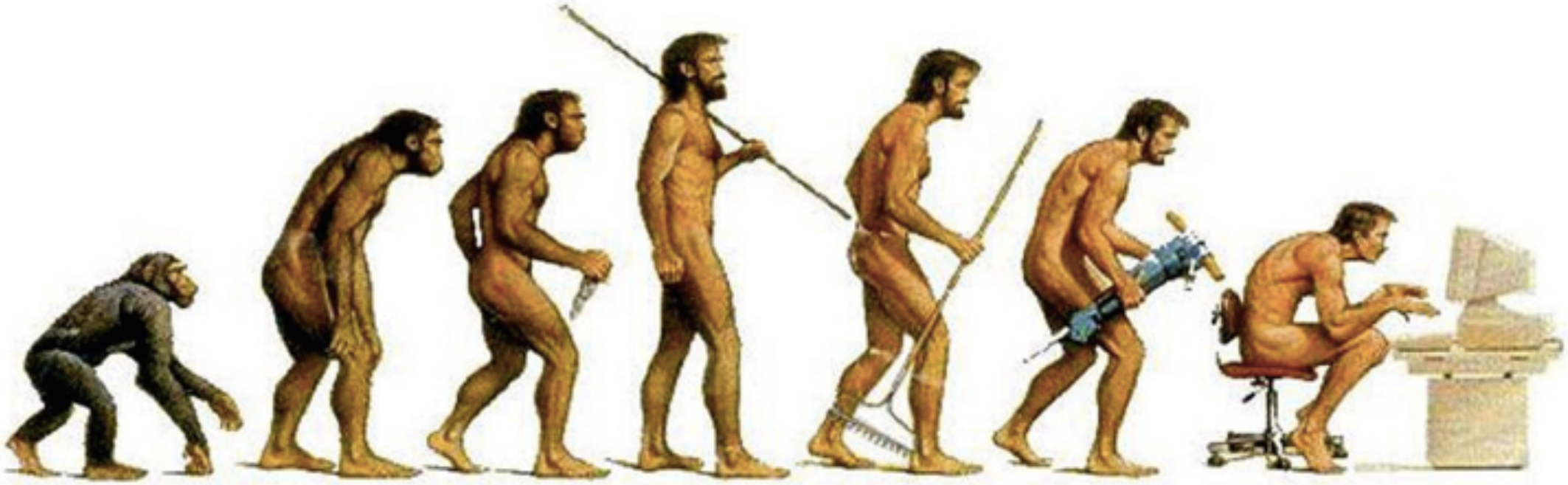
Synchronous world

Grains of Sand

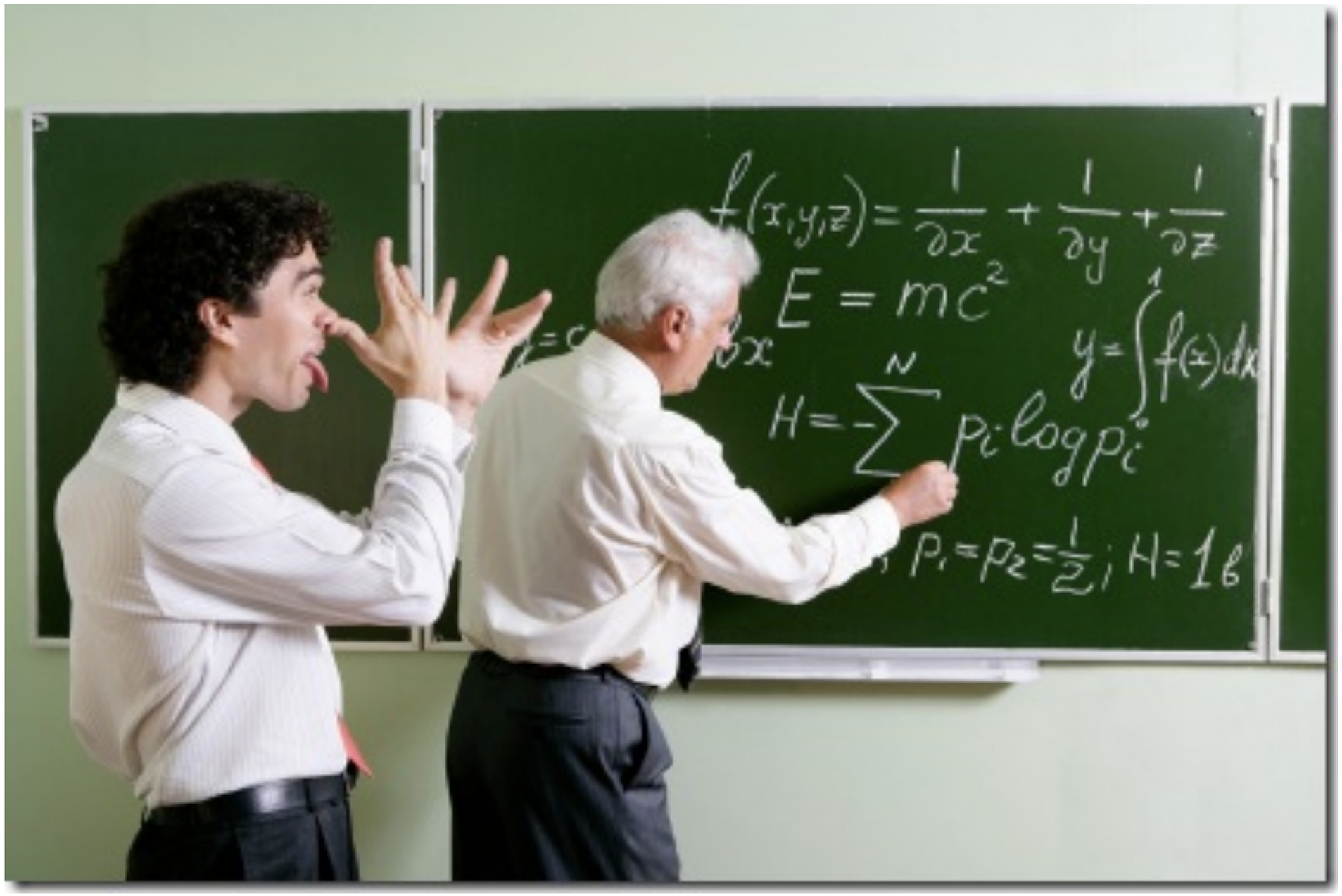


Jump On The Bandwagon

Versioning Avoidance



Missing mock servers



Give It A REST

Broken Gate

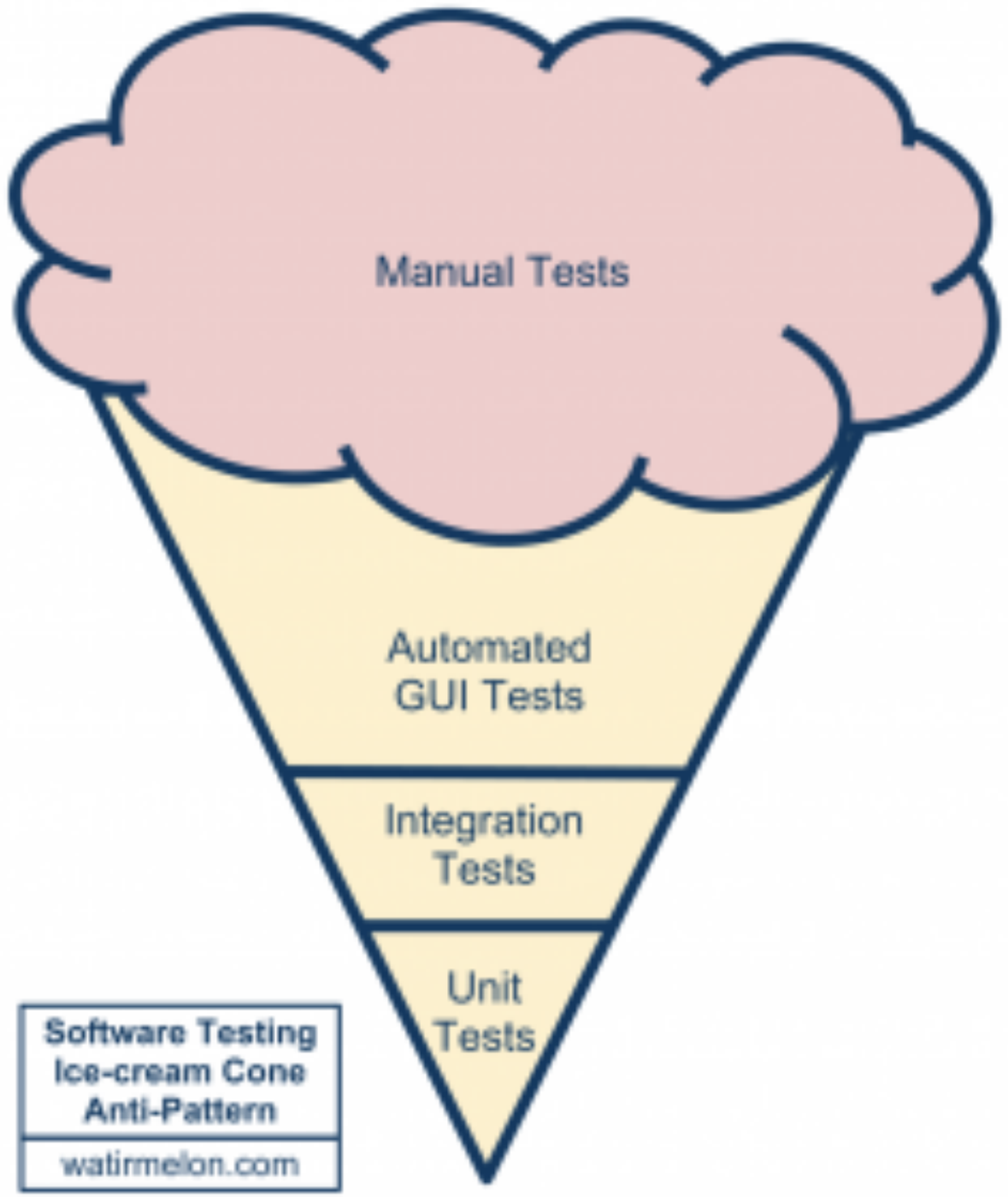
Layered Services Architecture

Lipstick on the Pig

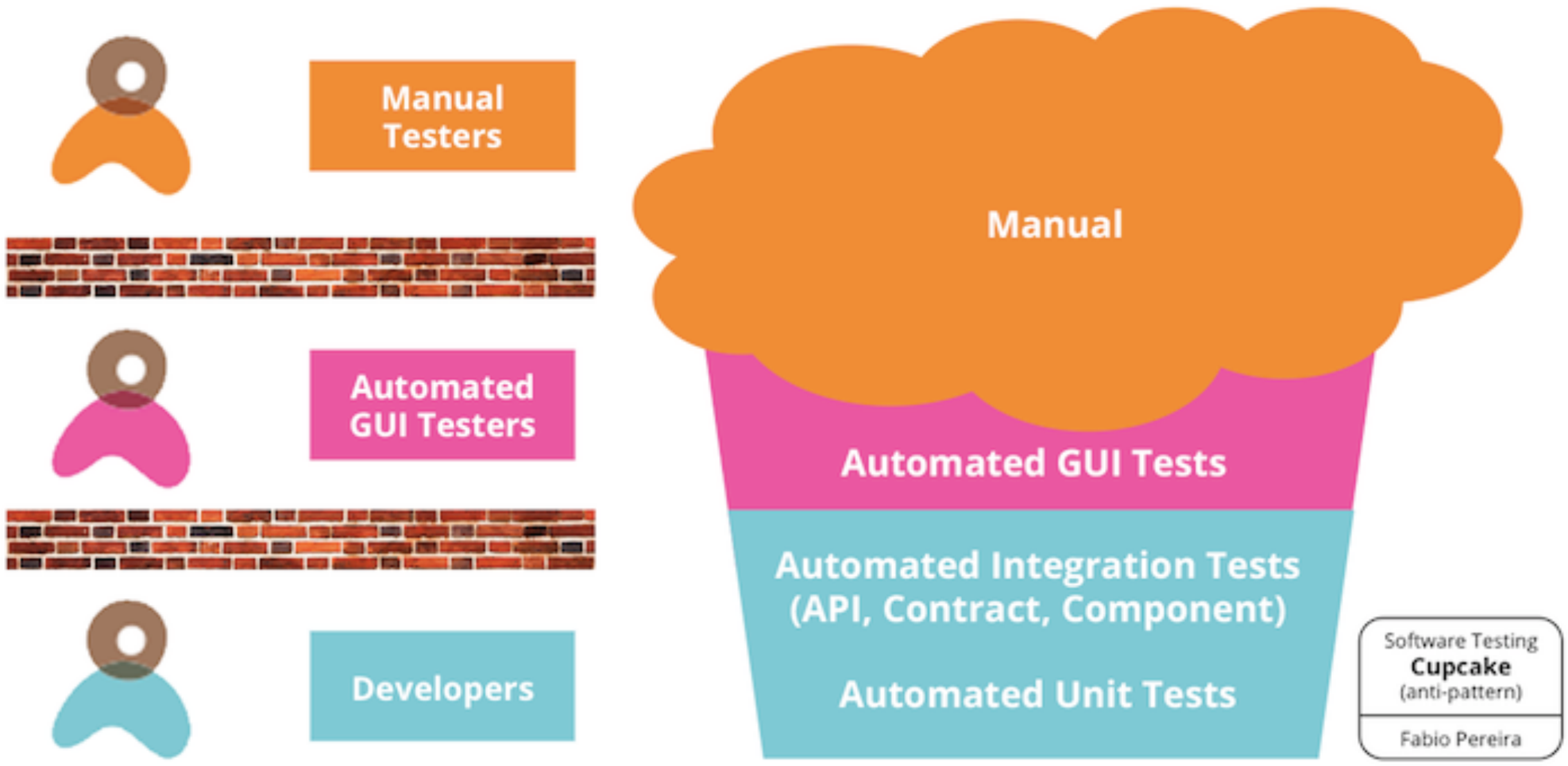
Backend Micrservice

NETFLIX

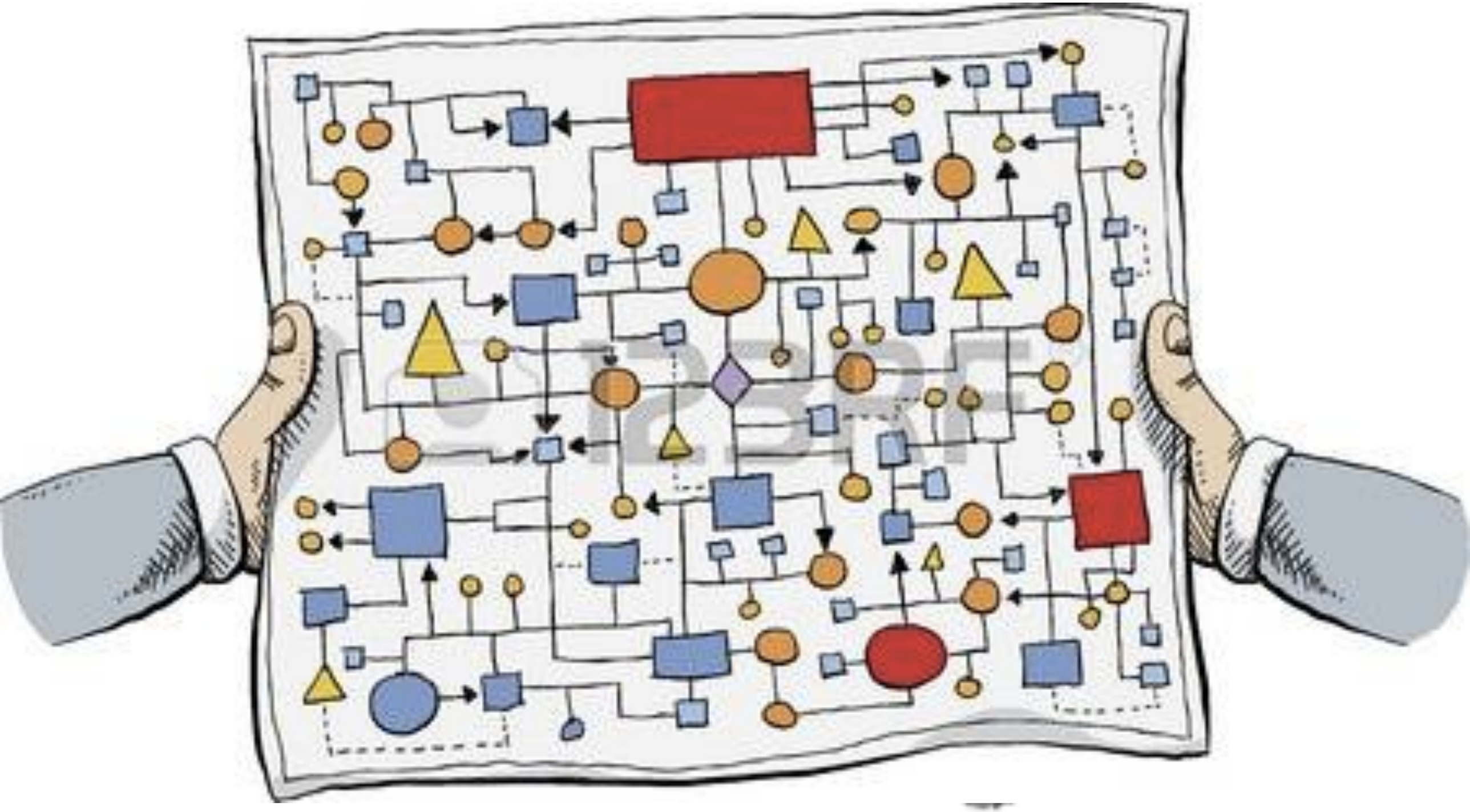
Ice Cream Cone Test



Cup Cake Anti Pattern



Addicted to complexity



Politics Oriented Architecture

The Unteachables

"If you can't understand it, don't invest in it.“

- Warren Buffet

Lust

Do you really need it ?

Appendix

Flying Blind

there are no healthchecks, no monitoring and no metrics, so you assume everything is well in your systems, all the time.

- you need each service to be able to produce a self healthcheck and metrics, possibly trough an HTTP endpoint, and surround it by some external monitoring.

- drop wizard, spring boot

Monolith database

microservices expose nice different REST endpoint, but are are sharing the same database

- moving on from the big SQL joins and welcoming asynchronous completion of tasks, a user interface that progressively updates itself

Unknown caller

there’s no correlation between calls, so each call is a new one, completely unrelated to the source of it.

- each server must inject a call id if missing , each server must propagate the call id if present, each server should log each call including the id. Thus providing a clear chain of invocations between the services

Hardcoded hell

All the address (endpoints) of services are hardcoded somewhere, sometimes directly in the code.

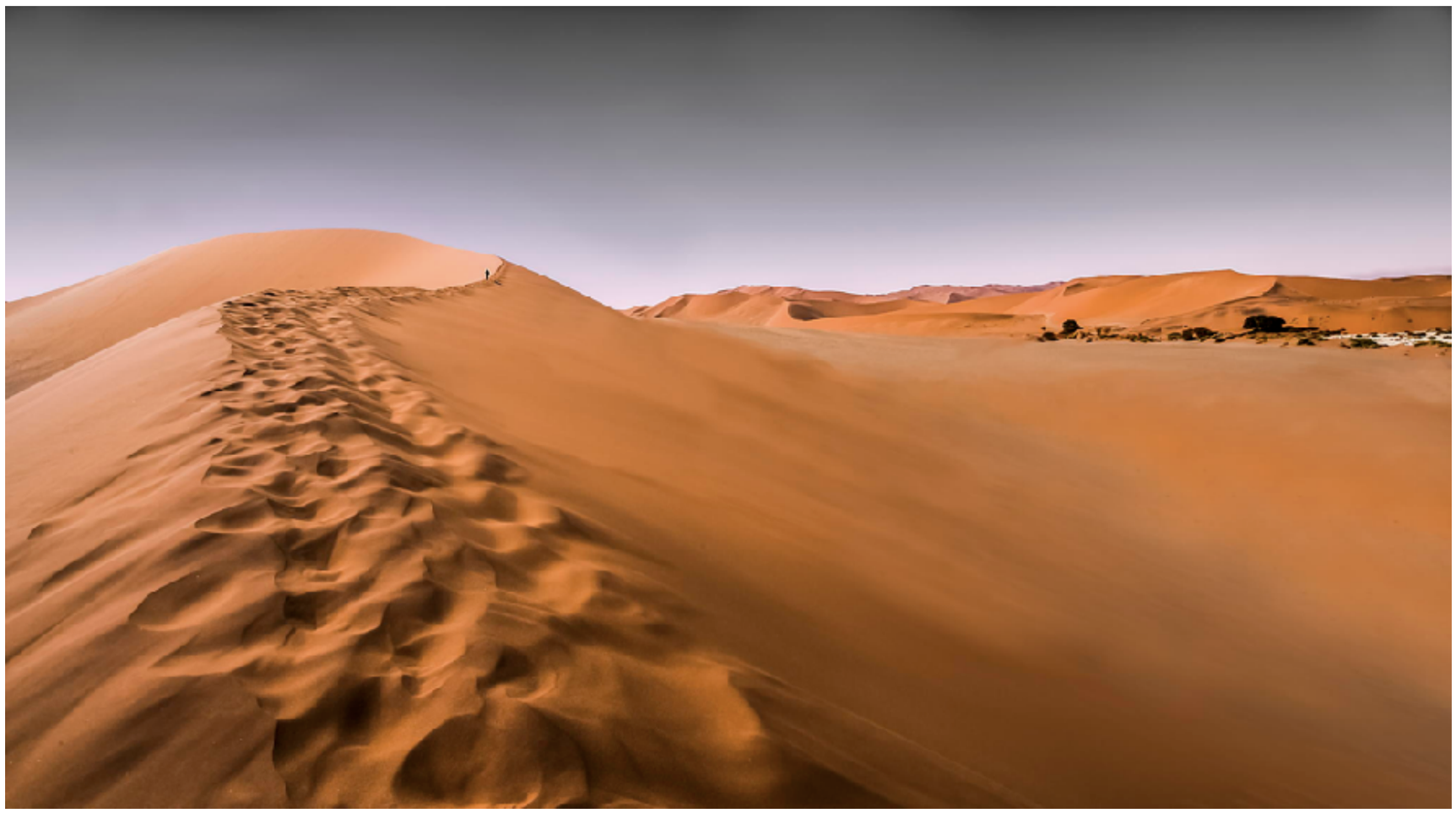
- introduce a discovery mechanism like eureka, zookeper, consul, msnos

Synchronous world

Every call in your systems is synchronous, and you wait for things to actually happen before returning to the caller

- using queues on top of your receivers and implementing asynchronous protocols from the start.

Grains of Sand



defining the size of a microservice using strict line of code counts

- As a rule of thumb you should start out with bigger, more coarse-grain services. It will always be easier to further decompose a single service than it will be to combine multiple services.

Jump On The Bandwagon

starts building microservices before the tools and processes are in place to support them.

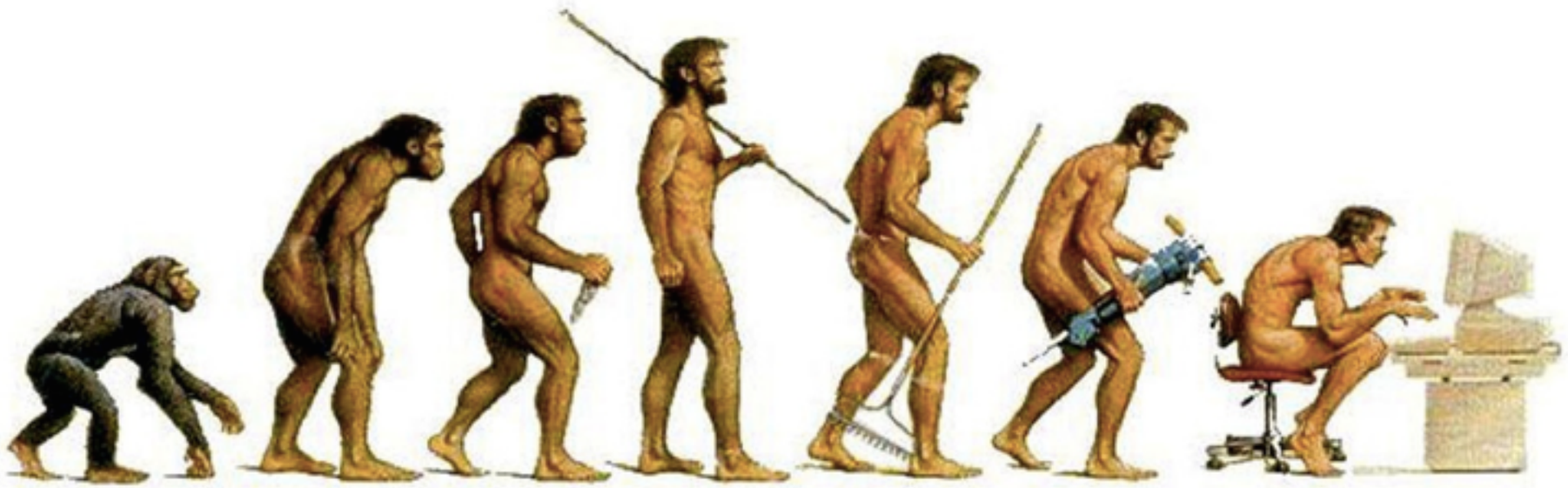
- if you’re organization is not ready or willing to change it’s testing, monitoring and deployment mentality, then you likely won’t get all the benefits of microservices.

Dare To Be Different

- Each of these frameworks provide their own mechanisms for security, logging, configuration, embedded HTTP server, UI template engine, dependency injection, and more.

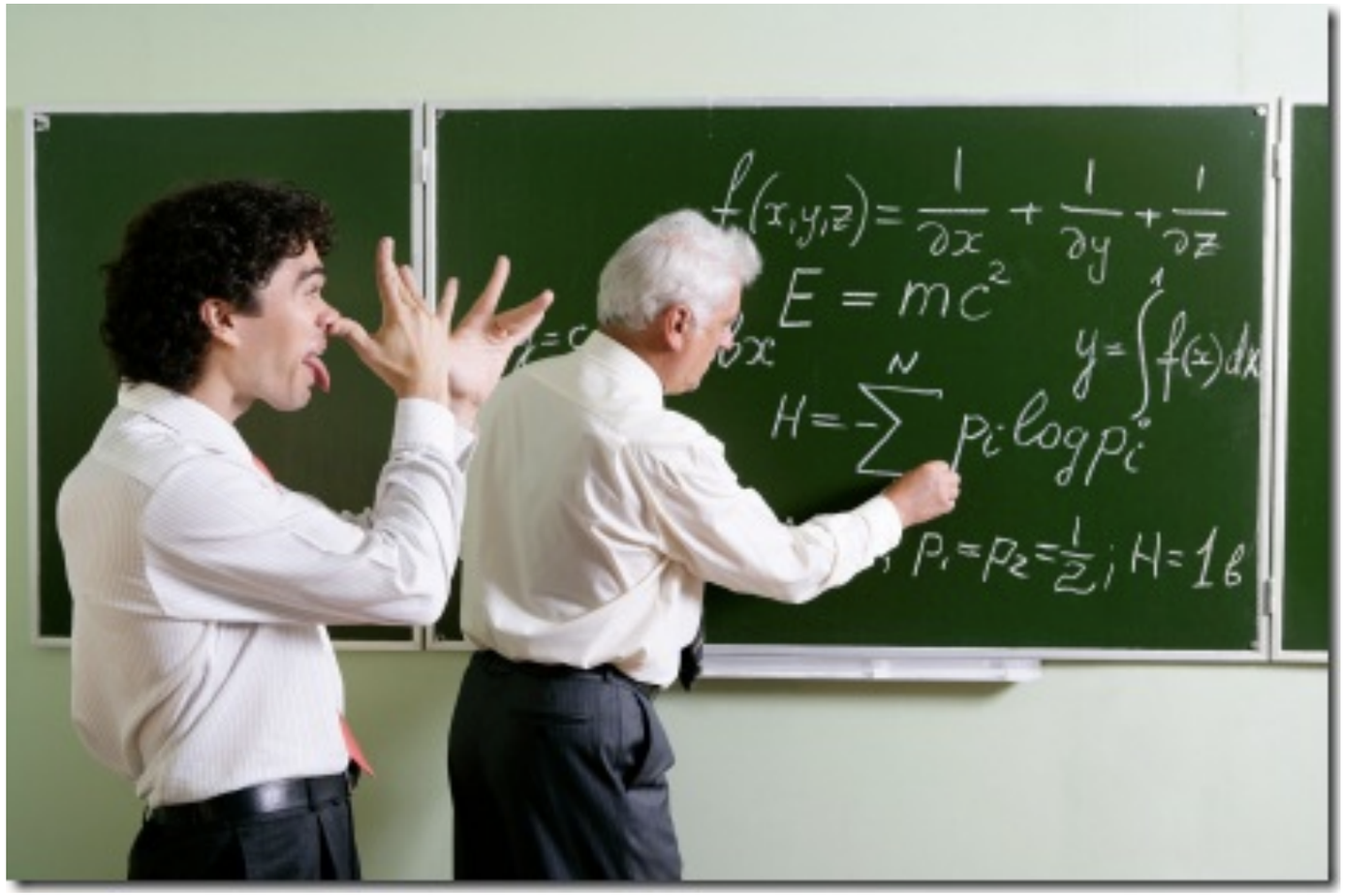
- one of your first tasks, before you build a single microservice, should be to standardize the technology stack for everyone.

Versioning Avoidance

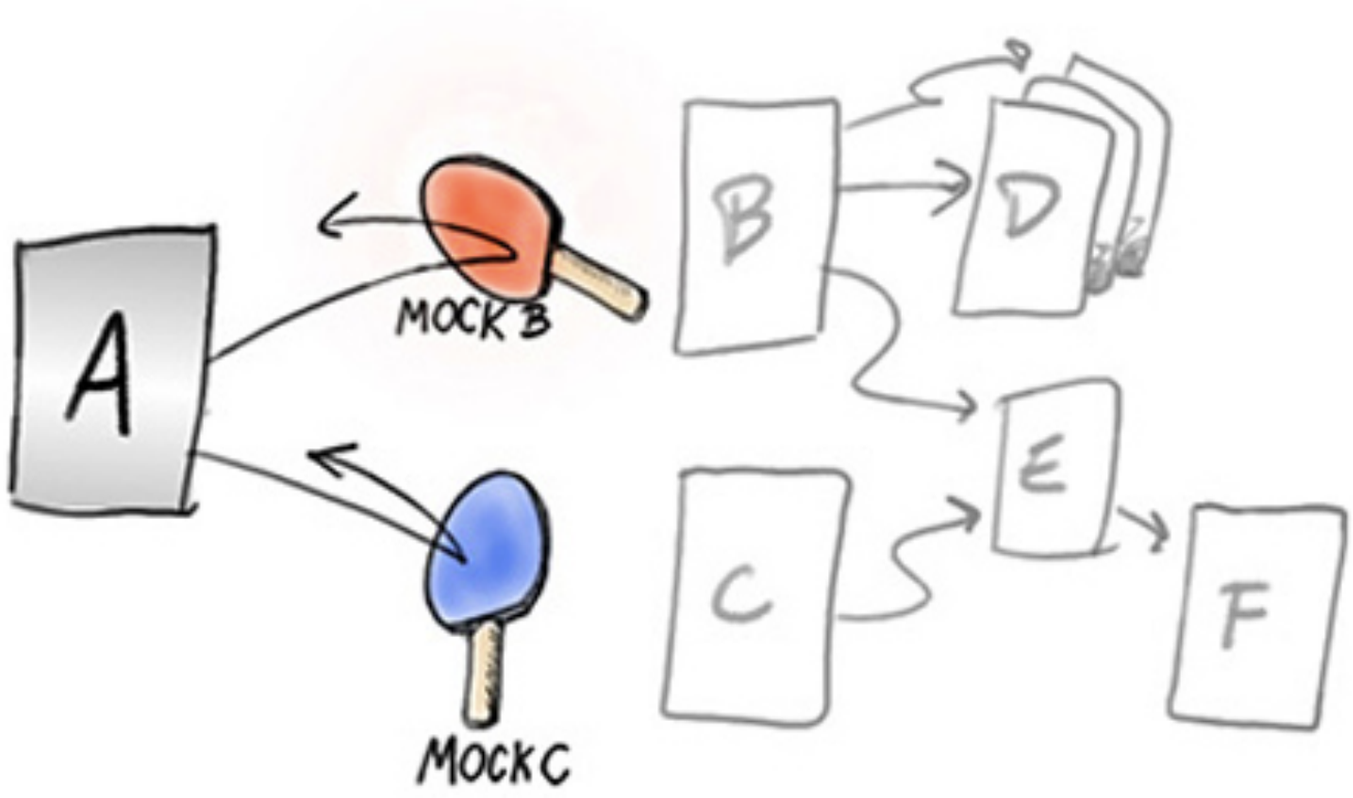


Thought it would be only need one version of the service.

Missing mock servers

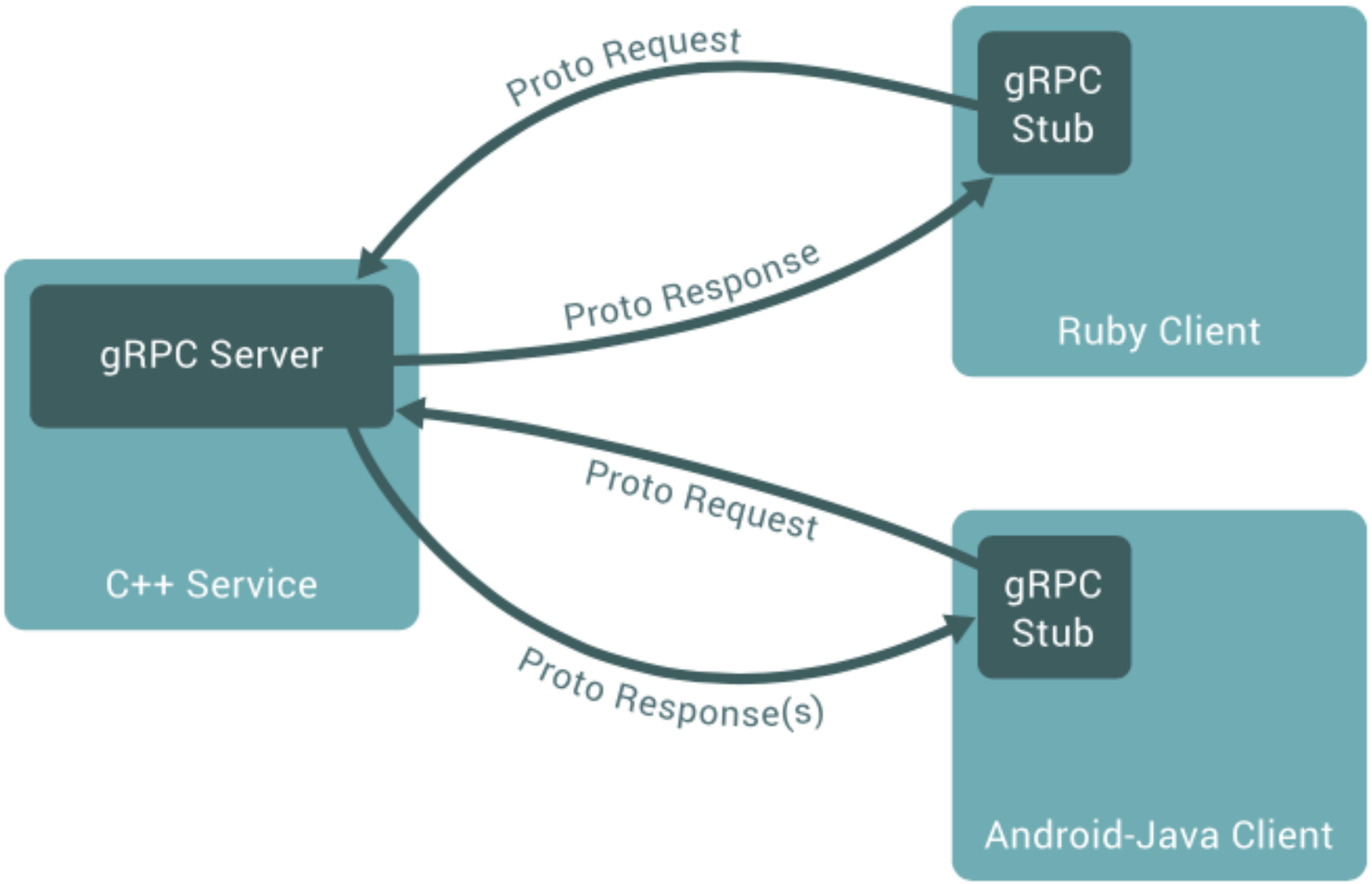


each consuming team of a service has to write their own mocks and stubs, which is wasteful.


A
MOCK B
MOCK C
B
D
E
C
F

Give It A REST

Relying heavily on RESTful APIs when other messaging paradigms may be more appropriate.



REST
gRPC
Apache Thrift

- Have a versioning strategy that can allow the consumers a graceful migration and assure providers can transparently deploy changes without affecting anyone. Limit the number of side-by-side major versions in the production and govern them.

Manual Configurations Management

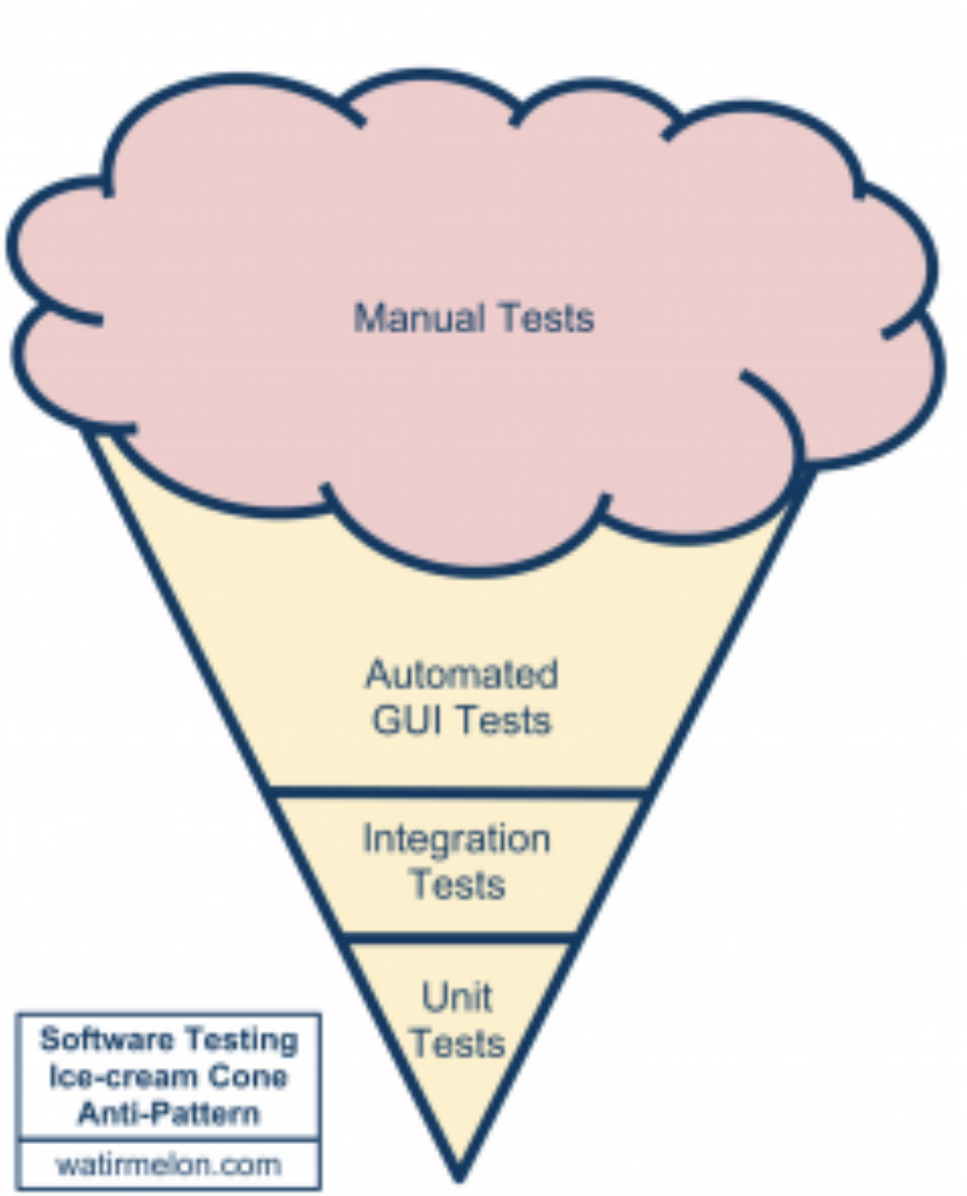
managing the configurations for each service

Layered Services Architecture

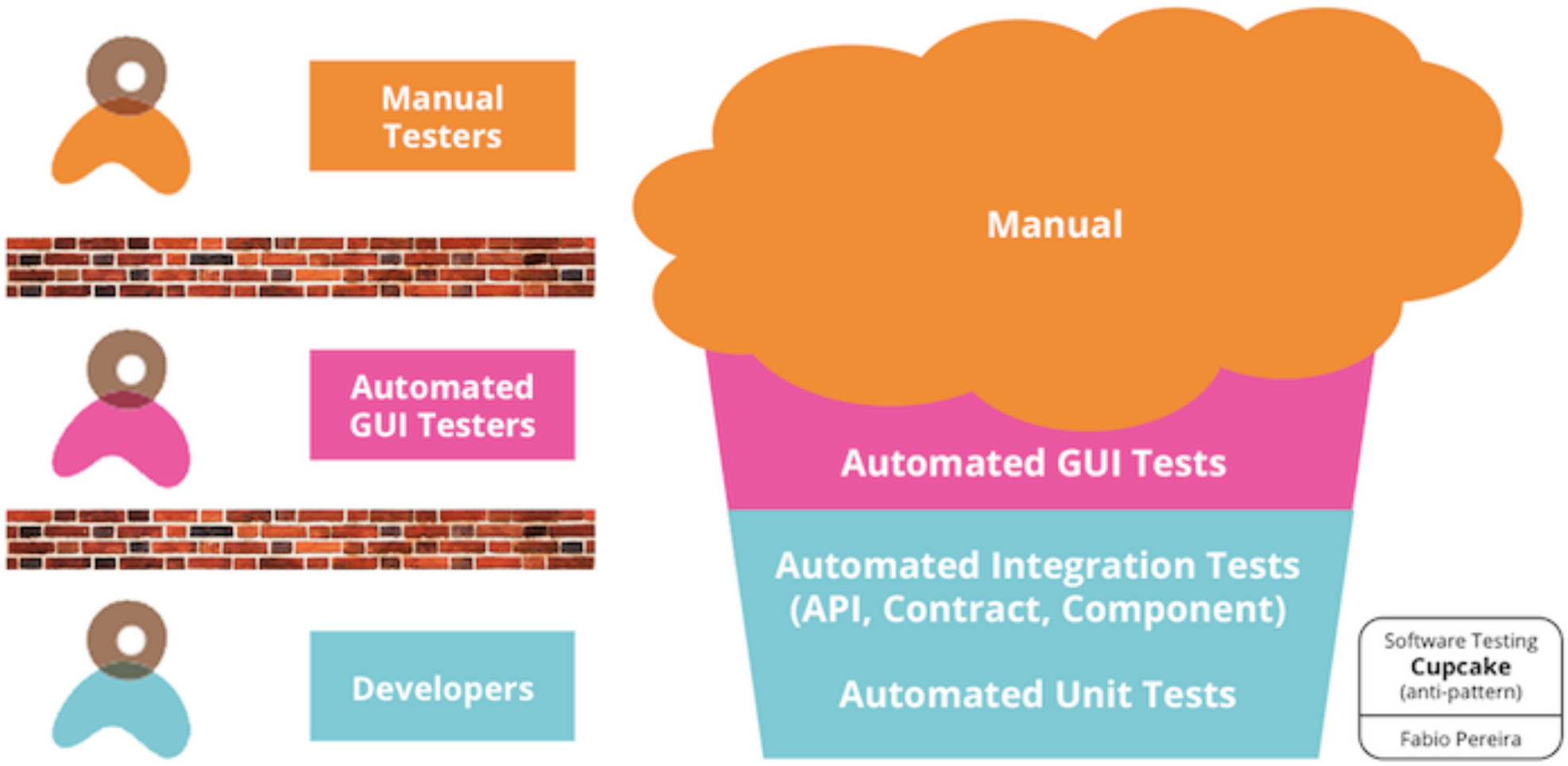
Teams mostly focused on technical cohesion rather than functional regarding reusability. For example, several services functioned as a data access layer to expose tables as services;

Not using a Gate

Ice Cream Cone Test



Cup Cake Anti Pattern



Appendix

- use application configuration management tools

- Logical separation of layers within a service is fine, however, there should not be any out of process calls. Try to look at a service as one atomic business entity, which must implement everything to achieve the desired business functionality.

Invest in API Management solutions to centralize, manage and monitor some of the non-functional concerns and which would also eliminate the burden of consumer's managing several microservices configurations.

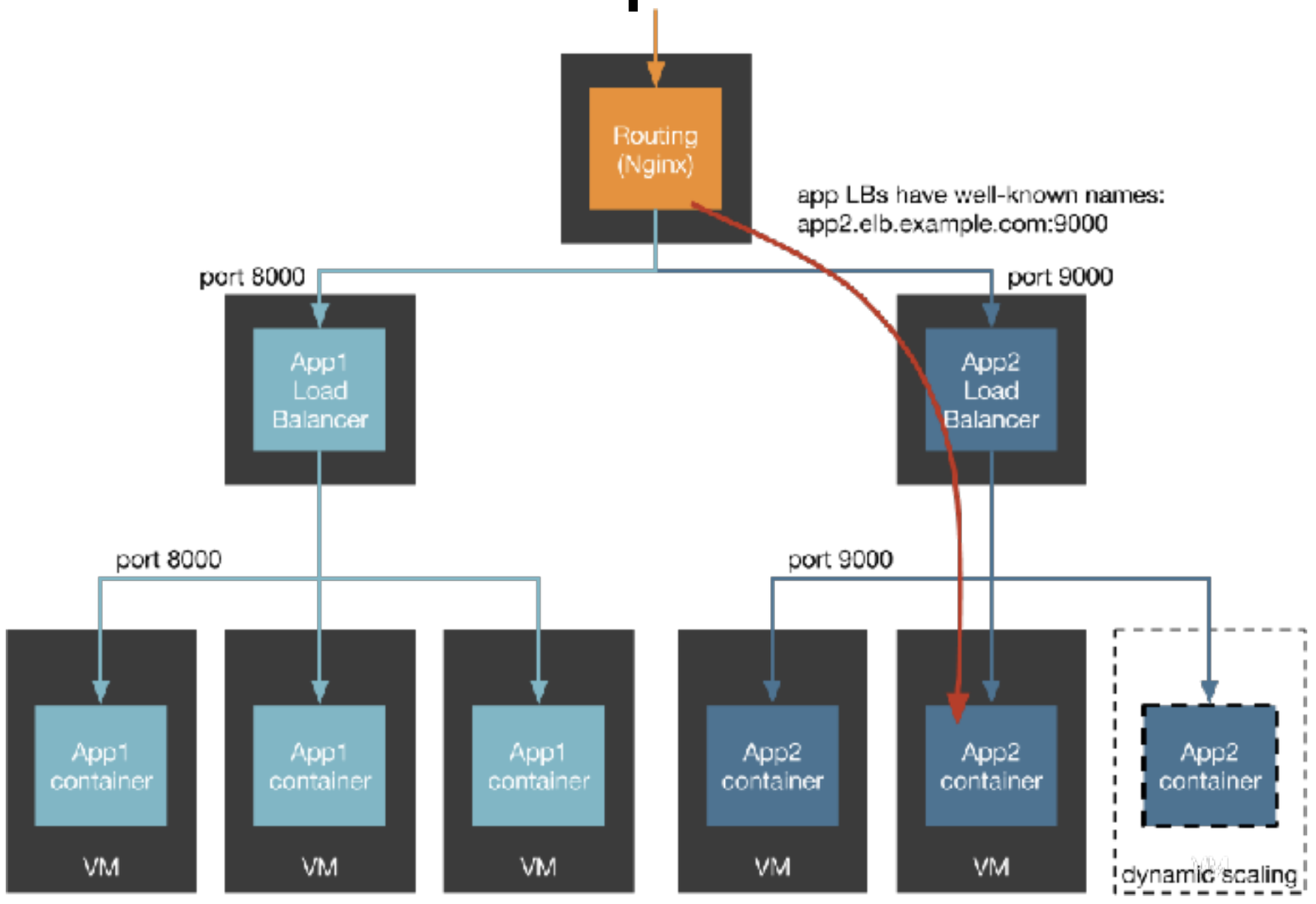
Netflix Architecture

Google Architecture

- happens when there is not enough low-level testing (unit, integration and component), too many tests that run through the Graphical User Interface (GUI) and an even larger number of manual tests.

- Writing tests usually spans 3 teams:
  - Developers tests.
  - UI Automation.
  - Manual testers.
- This results in duplication.

Local proxies



a load balancing service fronting each stateless microservice.

- If an application instance is unable to respond to remote health checks, then the load balancer will remove that application instance from the pool.

Local proxies - Anti pattern

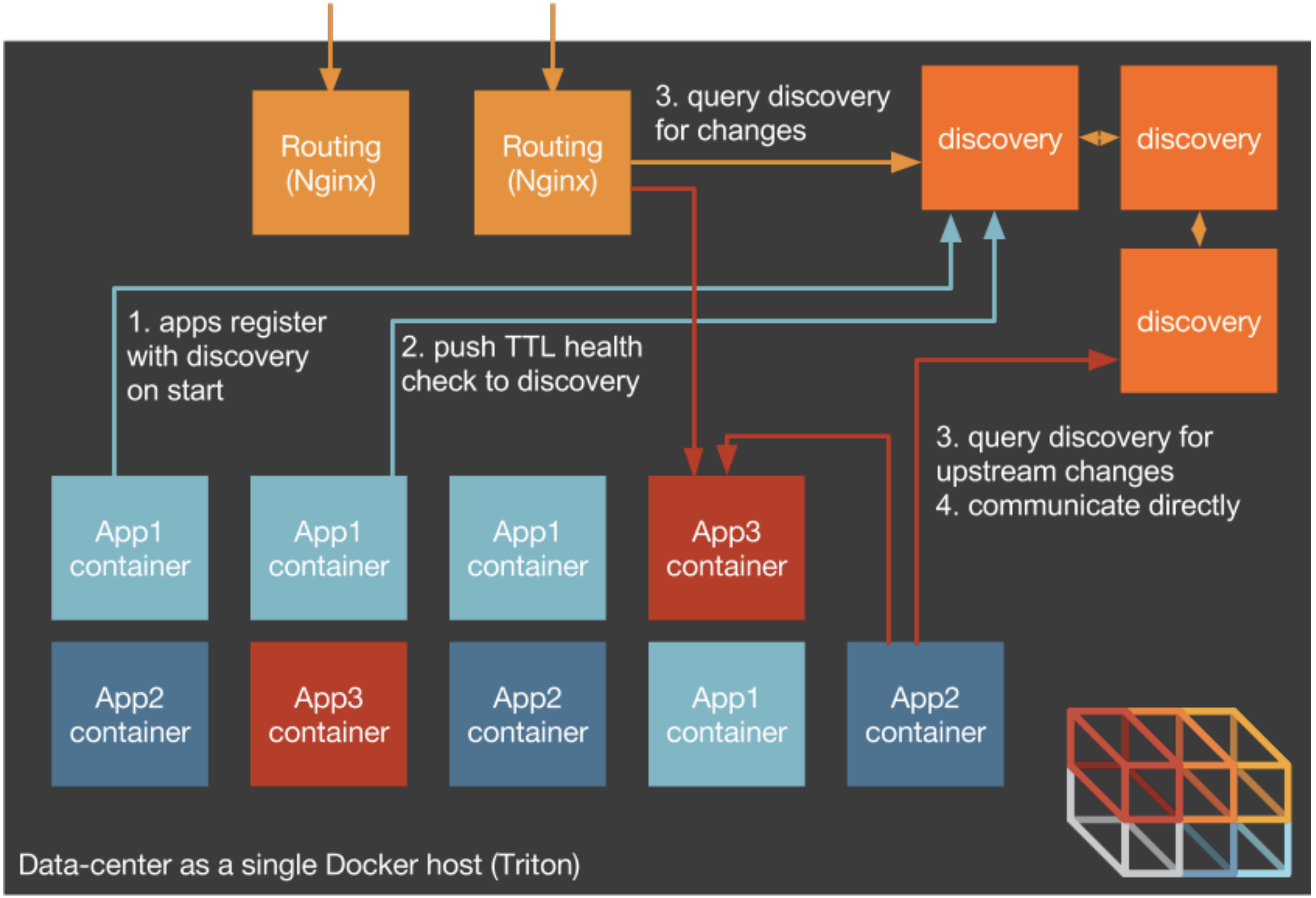
- Health checking: If an application instance is unable to respond to remote health checks, then the load balancer will remove that application instance from the pool. But if the application instance is unable to respond because it is under heavy load or is in a blocking state (ex. with Node.js performing computationally-heavy work or Rails unicorn workers waiting on long-running queries), then the load balancer may be removing instances just when the service needs more instances in the pool. Scaling up additional VM capacity takes on the order of minutes, and any new instances will be coming live in a state where there are still not enough nodes marked as healthy to serve queued requests. There is actually enough capacity available to run the service but the LB is "helpfully" not leaving instances marked healthy long enough to take up the load and the service is dead in the water until the health checks are turned off temporarily by a bleary-eyed operator at 2AM.

- Networking Inefficiency: We've introduced an extra network hop for every single internal request.

- Utilization: The hidden assumption of this architecture is that we have one container per VM. If we provision multiple containers per VM to improve utilization, then we need to configure the LB to listen for multiple ports and the question of how health checks are supposed to work becomes complicated. Do we remove the entire node because one of its services dies? Do we need an instance of every service on each VM or can we route some ports to only some instances? Do we even have that option if we're using a managed LB?

- pushes the responsibility for the application topology away from the network infrastructure and into the application itself where it belongs -- only the application can have the full awareness of its state necessary to manage itself

- https://www.joyent.com/blog/container-native-discovery


Routing
(Nginx)
Routing
(Nginx)
3. query discovery
for changes
discovery
discovery
discovery
1. apps register
with discovery
on start
2. push TTL health
check to discovery
3. query discovery for
upstream changes
4. communicate directly
App1
container
App1
container
App1
container
App3
container
App2
container
App3
container
App2
container
App1
container
App2
container
Data-center as a single Docker host (Triton)