

# Distributed Systems I

## *Software Architecture*

Brae Webb & Richard Thomas

March 24, 2025



**Mathias Verras**  
@mathiasverraes

There are only two hard problems  
in distributed systems:

2. Exactly-once delivery
1. Guaranteed order of messages
2. Exactly-once delivery

Lecture Goal: Balance a healthy love-hate relationship with dis-  
tributed systems

*Going forward*

Investigating architectures that are *distributed*.

## *Distributed Systems Series*

Distributed I    *Reliability* and *scalability* of  
*stateless* systems.

Distributed II    *Complexities* of *stateful*  
systems.

Distributed III    *Hard problems* in distributed  
systems.

*What are the benefits?*

- Improved *reliability*
  - Improved *scalability*
  - Improved *latency*
- Some systems are inherently distributed.

*What are the drawbacks?*

- Increased *complexity*
- Increased *attack surface*
- Increased *latency*
- Introduce *consistency* problems

We'll look at a few reasons that distributed systems are *fundamentally* quite challenging

## *§ Fallacies*

*A few reasons for complexity*

## Fallacies of *Distributed Computing*

Sun Microsystems in 1994, primarily accredited to Peter Deutsch  
(doy-ch)

*Fallacy #1*

The network is reliable



Success: Send request to add item to cart.



Failure 1: Request not received by server (CartService).



Failure 1: *Solution* resend request?



If the service goes down and all clients are re-trying,  
the service is in for a shock when it comes back,  
we solve this with *exponential backoff*.

## Exponential Backoff

```
1  retries = 0
2  do:
3      status = service.request()
5
5      if status != SUCCESS:
6          retries += 1
7          wait(2 ** retries)
8  while (status != SUCCESS and retries < MAX_RETRIES)
```



Failure 2: Response not received.



- Failure 1's *solution* of resending request leads to:
- *Duplicate actions*, problem for ordering/payments.



*Fallacy #2*

Latency is zero

## *Network Statistics*

Home to UQ

Home to us-east-1

EC2 to EC2

## *Network Statistics*

Home to UQ 20.025ms

Home to us-east-1

EC2 to EC2

## *Network Statistics*

Home to UQ 20.025ms

Home to us-east-1 249.296ms

EC2 to EC2

## *Network Statistics*

Home to UQ 20.025ms

Home to us-east-1 249.296ms

EC2 to EC2 0.662ms

- Be mindful when designing distributed systems.
- Network calls are *much* slower than local calls.

*Fallacy #3*

Bandwidth is infinite

Similar to previous fallacy, be mindful,  
distributed calls clog up network.

*Definition 0.* Stamp Coupling

Components which share a composite data structure.

Particularly an issue if components are on different compute nodes.

*Fallacy #4*

The network is secure



Authentication only occurs when entering Sahara data centre.



- Bad actor gets access via one insecure node.
- Network is *compromised*.
- Practice *defence in depth*.

### *Fallacy #5*

The topology never changes

- Topology changes all the time, cloud makes this *easier*.
- Don't rely on static IPs.
- Don't assume consistent latency.

### *Fallacy #6*

There is only one administrator

- Things spontaneously break.
- Who can help you?

*Fallacy #7*

Transport cost is zero

*Remember*

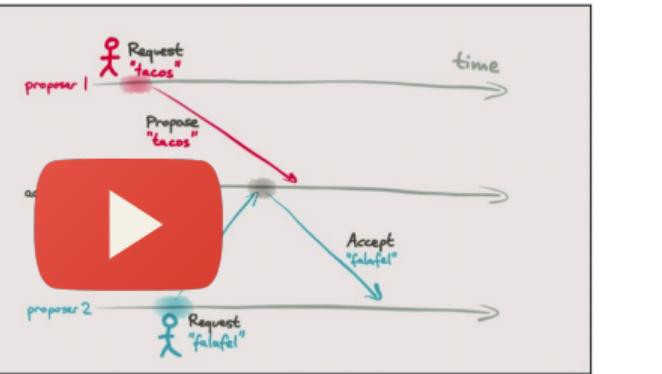
Distributed systems are *hard*. Choosing to use them should be *well considered*.

Can often introduce more problems than they solve.

*When you need to, maybe prove it?*



Sept 13-14, 2019  
[thestrangeloop.com](http://thestrangeloop.com)



- Presentation: Proving correctness of distributed systems.
- by Dr. Martin Kleppmann, University of Cambridge.
- Using Isabelle – See CSS research sub-group.

*Or, more realistically,*

Use *existing* algorithms and software

## *Distributed Systems Series*

Distributed I    *Reliability* and *scalability* of  
*stateless* systems.

Distributed II    Complexities of stateful  
systems.

Distributed III Hard problems in distributed  
systems.

## *Stateless vs. Stateful Systems*

Stateless Does *not* utilise *persistent data*.

Stateful Does utilise *persistent data*.

*Question*

What makes software *reliable*?

*Definition 0.* Reliable Software

Continues to work, even when things go wrong.

*Definition 0.* Fault

Something goes wrong.

Death, taxes, and computer system failure are all inevitable to some degree.

*Plan for the event.*

– Howard and LeBlanc

*Reliable software is  
Fault **tolerant***

John von Neumann built fault tolerant hardware in the 1950s.

### *Problem*

Individual computers fail *all the time*

- 10-50 years hard-drive lifetime.
- 10,000 disks will fail daily.
- Google last had 2.5 million servers.
- ...

### *Solution*

Spread the risk of faults over *multiple computers* or *nodes*

## *Spreading Risk*

If you have software that works with *just one* computer,  
spreading the software over *two* computers *halves* the  
risk that your software will fail.

## *Spreading Risk*

If you have software that works with *just one* computer,  
spreading the software over *two* computers *halves* the  
risk that your software will fail.

Adding *100* computers reduces the risk by *100*.



- Why is this software somewhat reliable?
- Any individual service can go down and the rest still work.
- Can we do better?
- Can a service go down but still have that service available?

*Question*

Who has used *auto-scaling*?

## Auto-Scaling Terminology

Auto-scaling group A *collection of instances* managed by auto-scaling.

## Auto-Scaling Terminology

Auto-scaling group A *collection of instances* managed by auto-scaling.

Capacity Amount of instances *currently* in an auto-scaling group.

## Auto-Scaling Terminology

Auto-scaling group A *collection of instances* managed by auto-scaling.

Capacity Amount of instances *currently* in an auto-scaling group.

Desired Capacity Amount of instances *we want to have* in an auto-scaling group.

## Auto-Scaling Terminology

**Auto-scaling group** A *collection of instances* managed by auto-scaling.

**Capacity** Amount of instances *currently* in an auto-scaling group.

**Desired Capacity** Amount of instances *we want to have* in an auto-scaling group.

**Scaling Policy** How to determine the desired capacity.

## Auto-Scaling Terminology

**Auto-scaling group** A *collection of instances* managed by auto-scaling.

**Capacity** Amount of instances *currently* in an auto-scaling group.

**Desired Capacity** Amount of instances *we want to have* in an auto-scaling group.

**Scaling Policy** How to determine the desired capacity.

**Minimum/Maximum Capacity** *Hard limits* on the minimum and maximum number of instances.

### *What we really want*

Desired Capacity Amount of *healthy* instances  
we want to have in an auto-scaling group.

## *Health check*

Mechanism to determine whether an instance is *healthy*.

## *Auto-scaling*

An example



Product Browsing service keeps going down



We might expect product browsing to have a much higher load than other services



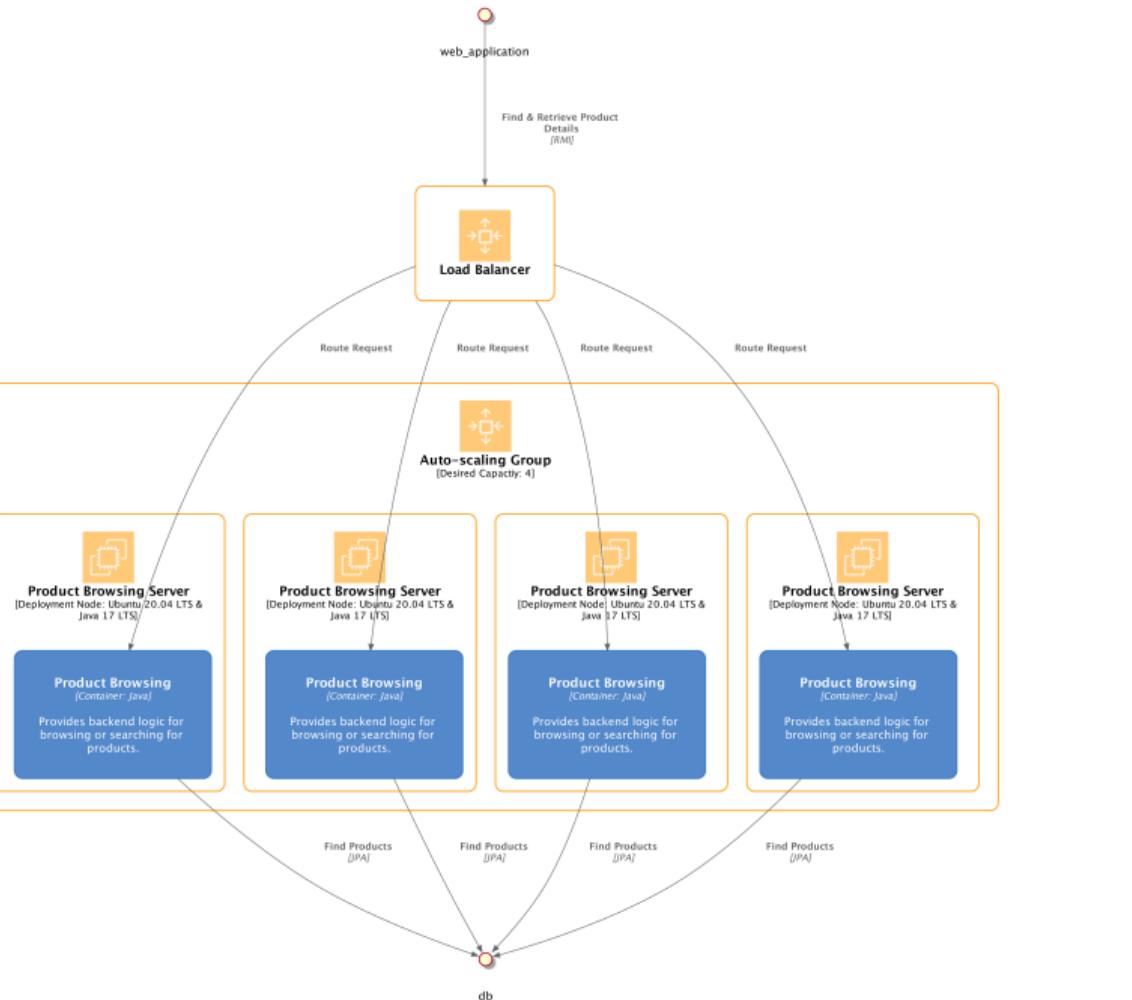
Deploy product browsing on an EC2 instance



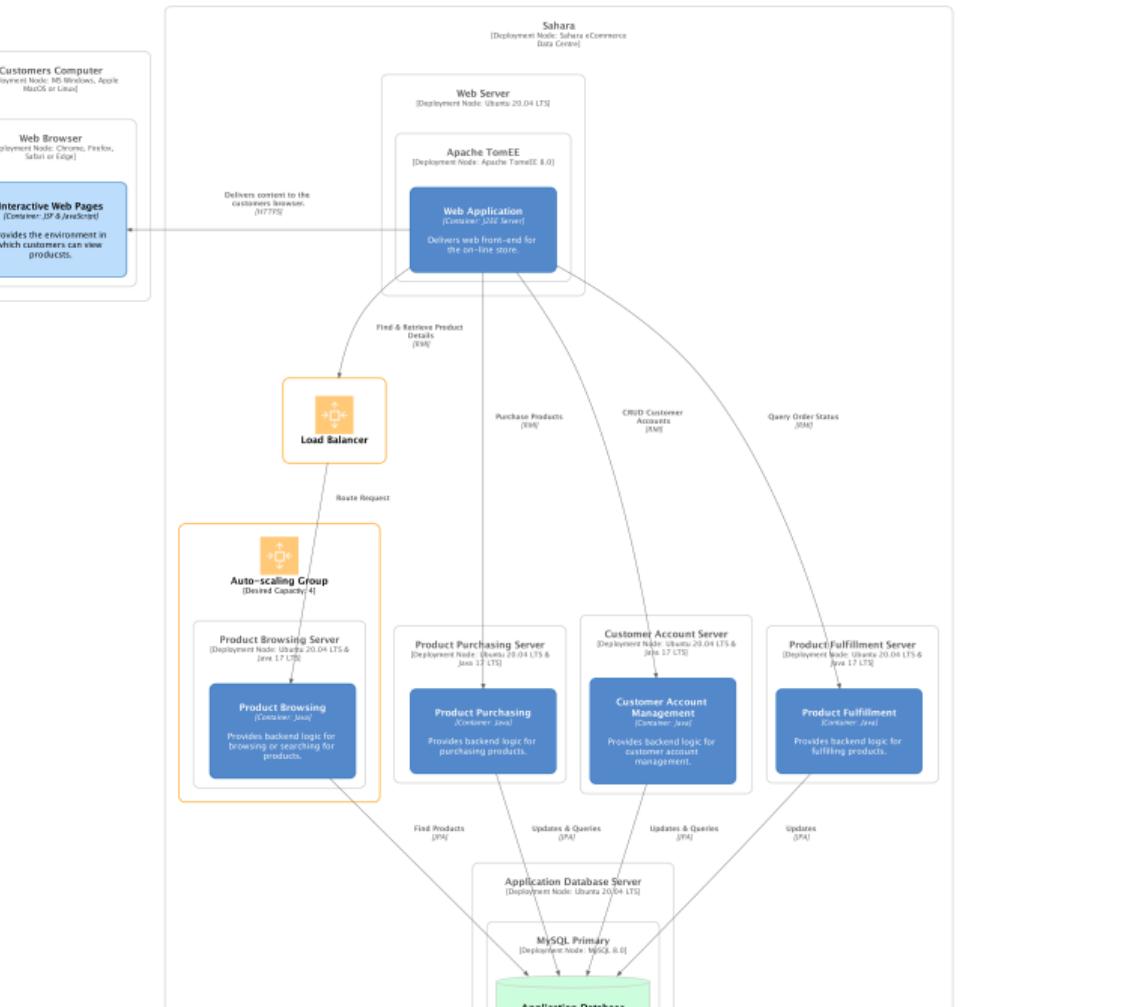
Use an auto-scaling group to replicate the service



What's the problem?



- Traffic was all sent through *one* instance
- Load balancer routes to *all* instances



Load balancing & scaling integrated into overall Sahara deployment diagram

## In Summary

Simplicity

Reliability

Scalability

## In Summary

Simplicity *Minimal network communication* (compared to other distributed systems), less impacted by fallacies.

Reliability

Scalability

## In Summary

Simplicity *Minimal network communication* (compared to *other* distributed systems), less impacted by fallacies.

Reliability Traffic is spread to various services, still *partially operational* if one goes down. Auto-scaling allows for *basic replication*.

Scalability

## In Summary

Simplicity *Minimal network communication* (compared to other distributed systems), less impacted by fallacies.

Reliability Traffic is spread to various services, still *partially operational* if one goes down. Auto-scaling allows for *basic replication*.

Scalability Auto-scaling and load balancing allows *individual services to scale*. However, the *database is a bottleneck*.

*Database is a bottleneck* is foreshadowing