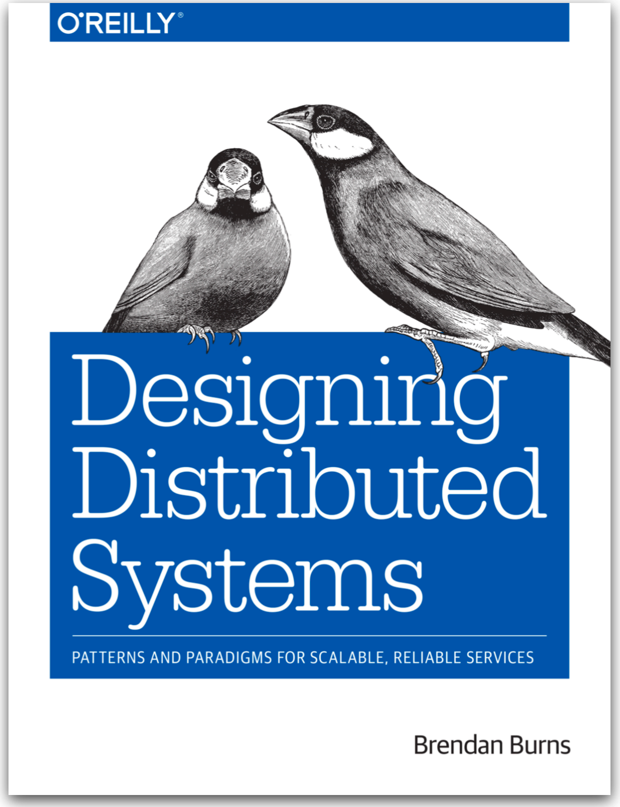
Cloud Native
#4 - Cloud Application Design

ERICSSON

This Session

- Challenges in designing and deploying scalable, efficient and safe cloud applications in an agile fashion

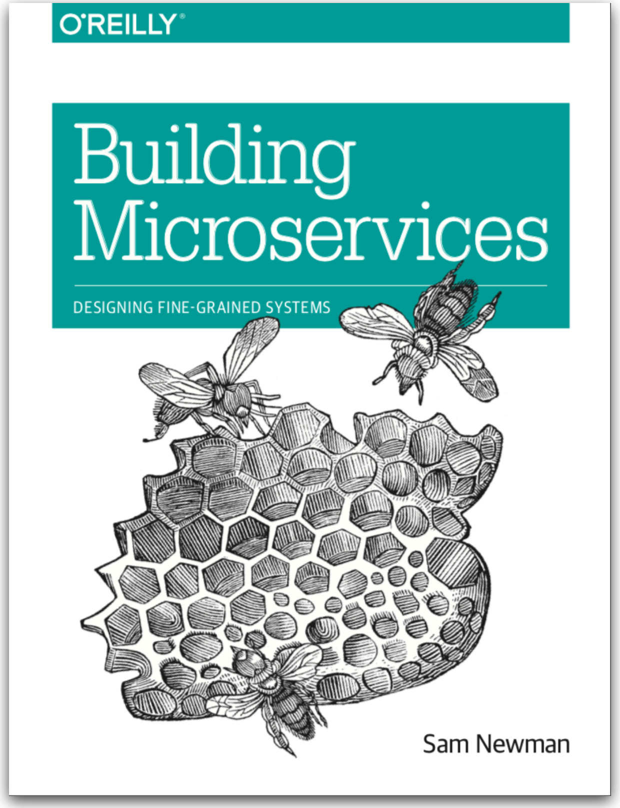


O'REILLY

Designing Distributed Systems

PATTERNS AND PARADIGMS FOR SCALABLE, RELIABLE SERVICES

Brendan Burns

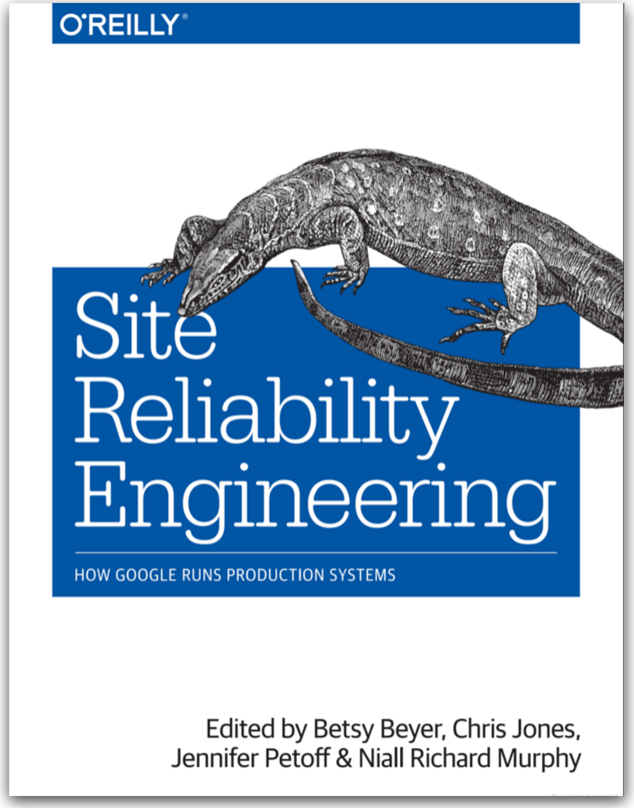


O'REILLY

Building Microservices

DESIGNING FINE-GRAINED SYSTEMS

Sam Newman



O'REILLY

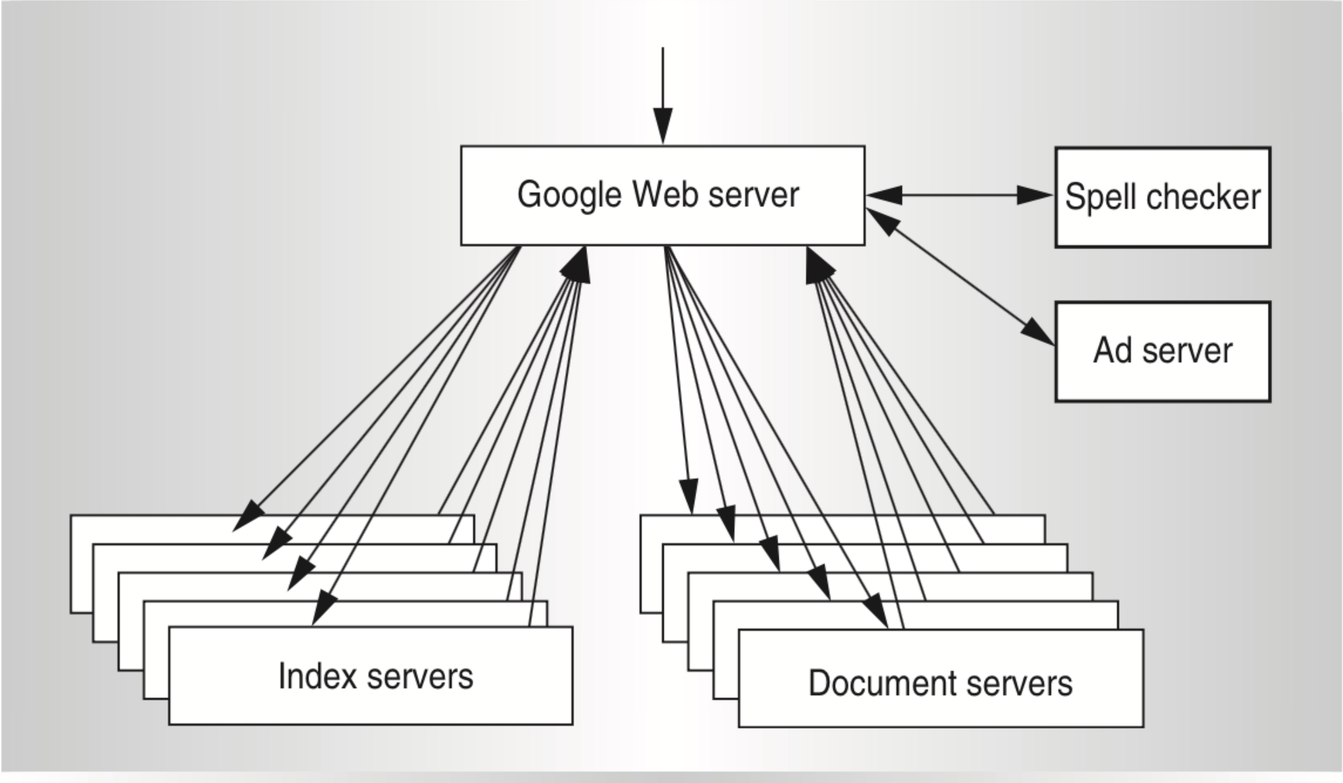
Site Reliability Engineering

HOW GOOGLE RUNS PRODUCTION SYSTEMS

Edited by Betsy Beyer, Chris Jones,
Jennifer Petoff & Niall Richard Murphy

Case Study: Google Search

- A web search touches 50+ separate services, 1000s machines
  - Searching is highly parallelizable
  - Map-Reduce
  - *Massive amounts* of data
- *Data gathering* ongoing background process
- *Latency* is important for user-facing services
- Data and responses are *heavily cached*

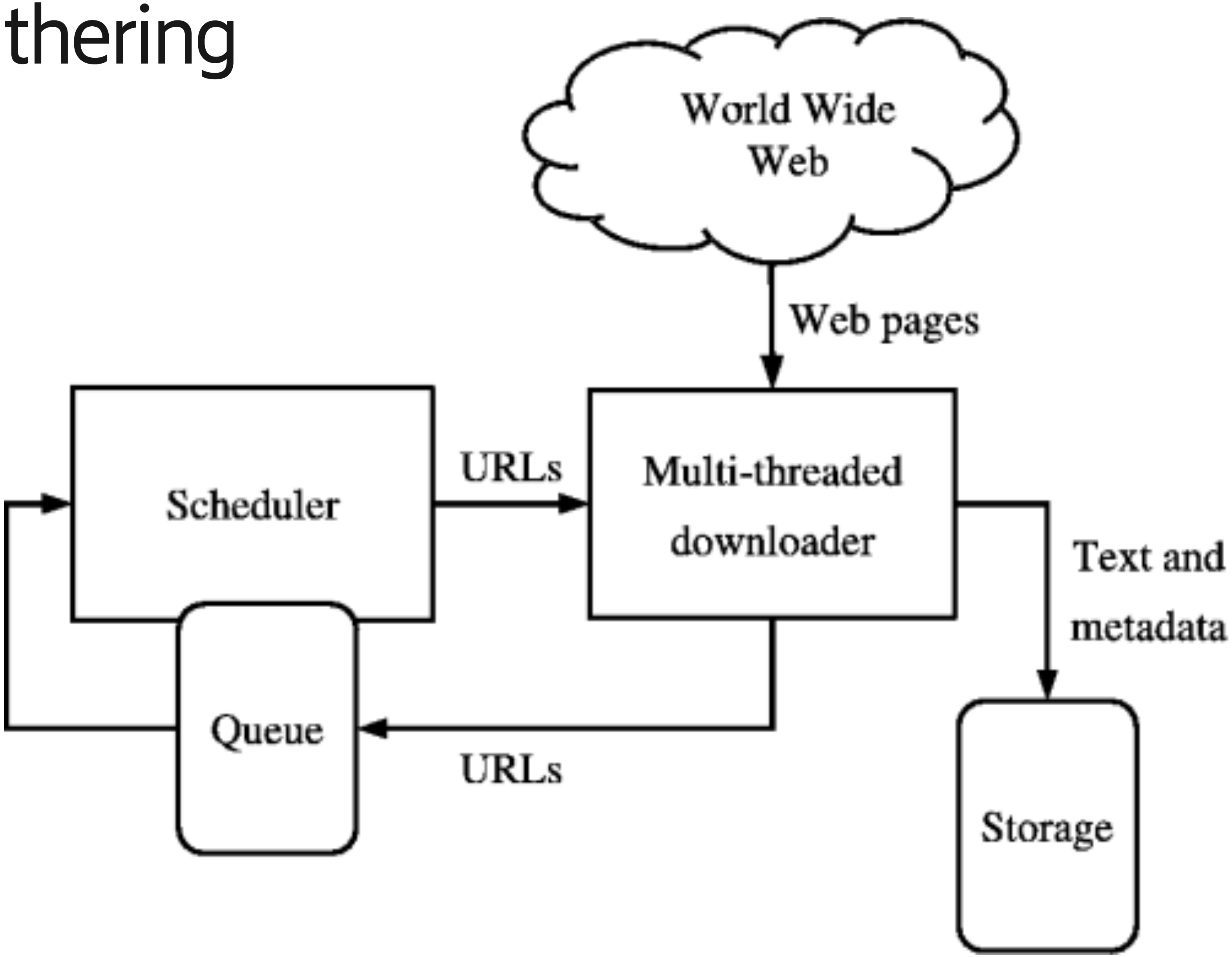


Massive amounts of data

- Large data sets
  - ~50 billion web pages index by Google
  - Average size of webpage = 20kB
  - 50 billion * 20KB = 1 PB (1^15 B)

- Network
  - NW bandwidth = 1 Gbps => Moving 10TB takes ~1 day
  - Solution: Push computation to the data
  - Round trip between Europe & US ~200 ms

Data gathering



Performance = High throughput

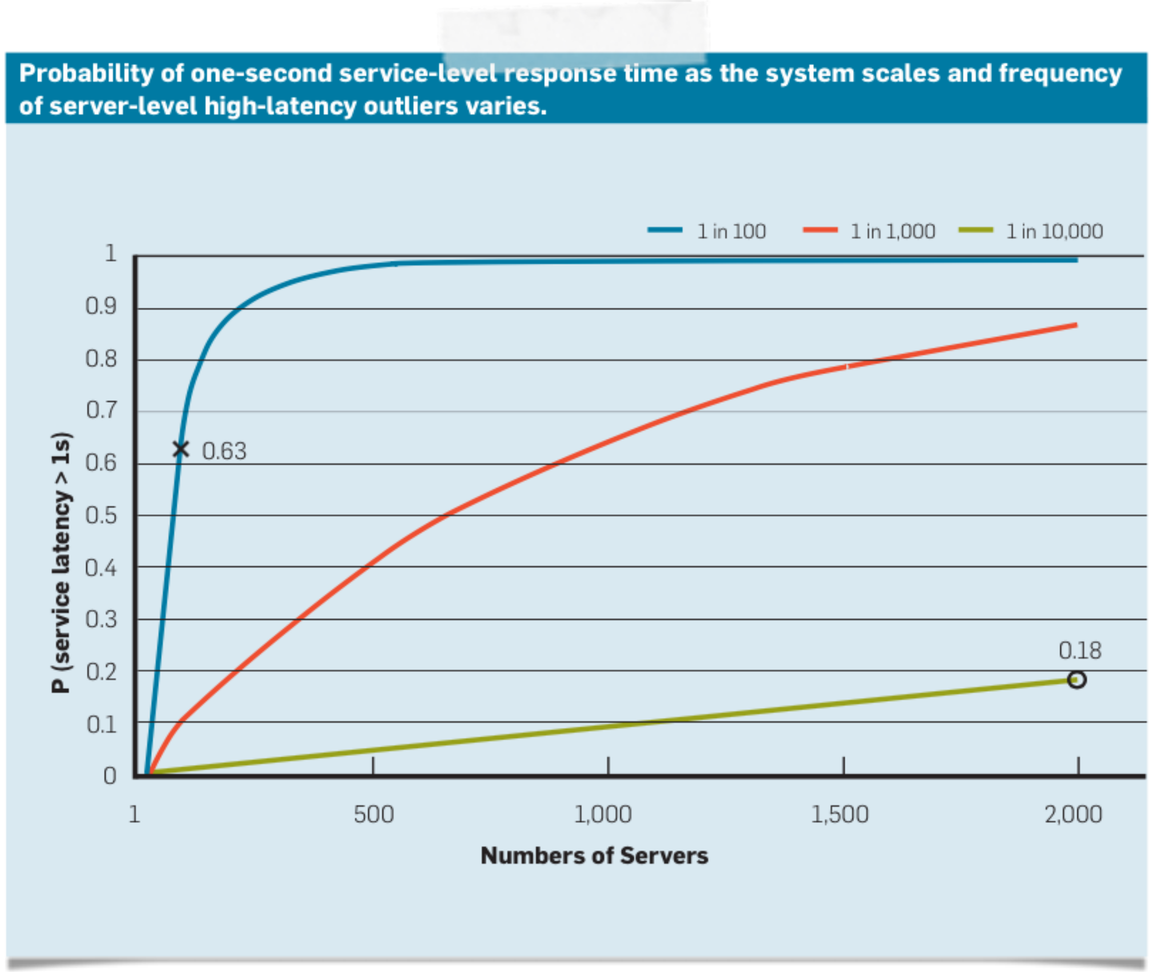
Latency

- Quality = low latency

- Server response
  - Typical: 10 ms
  - One out of x: 1000 ms

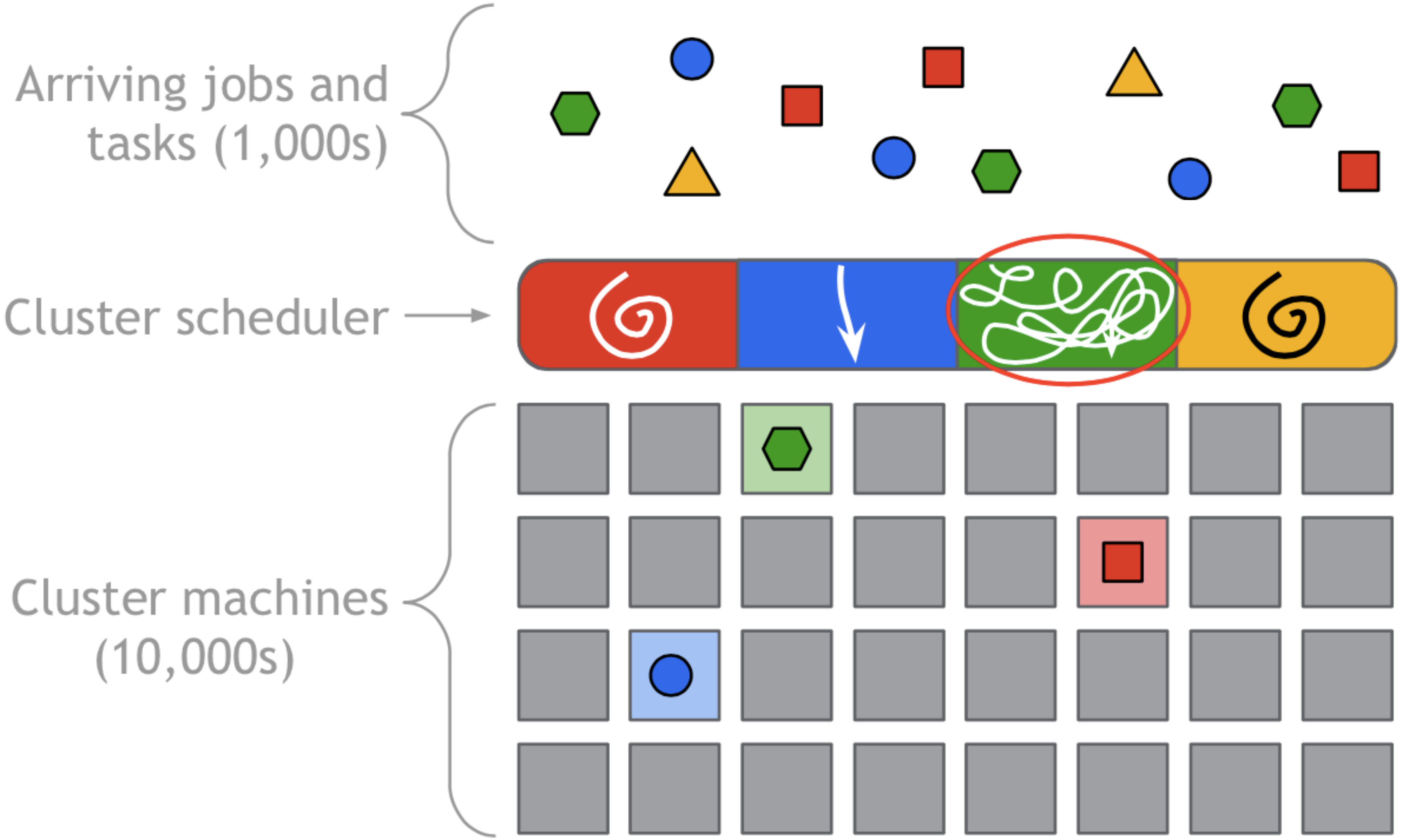
- Latency sources
  - Resource sharing
  - SSD GC & compactations

- Mitigate by running several request copies in parallel and use earliest response



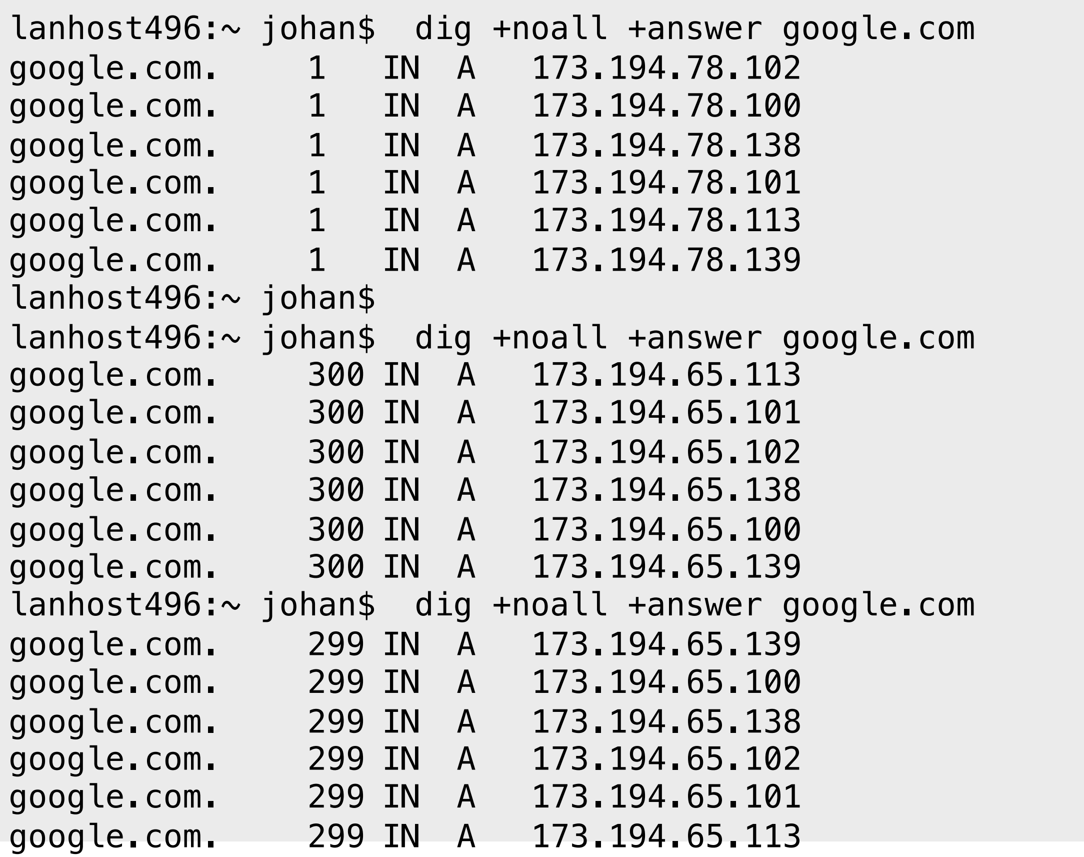
Probability of one-second service-level response time as the system scales and frequency of server-level high-latency outliers varies.

The Cluster Scheduler

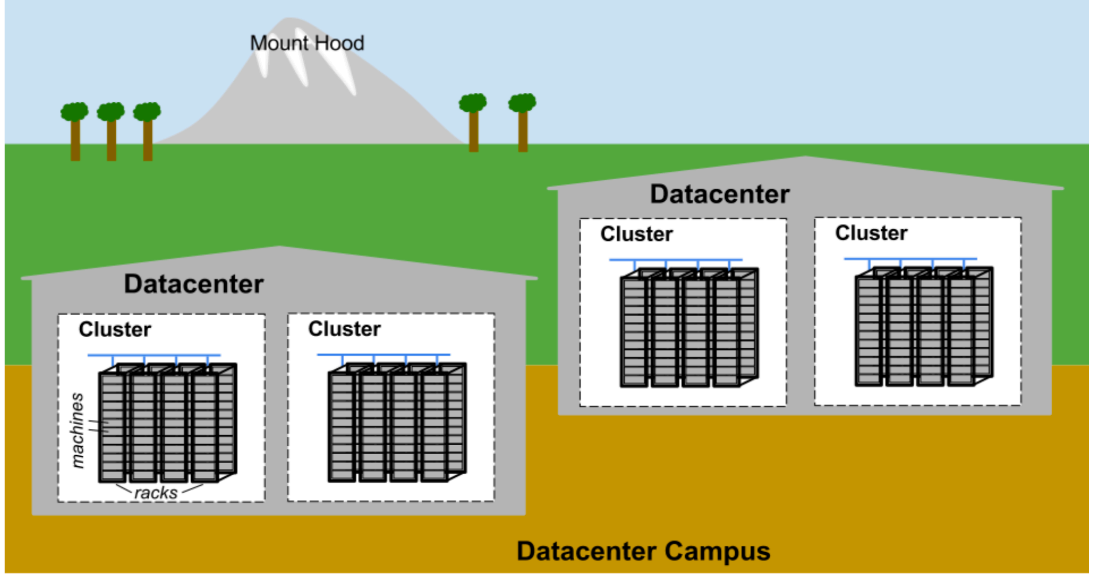


Getting there

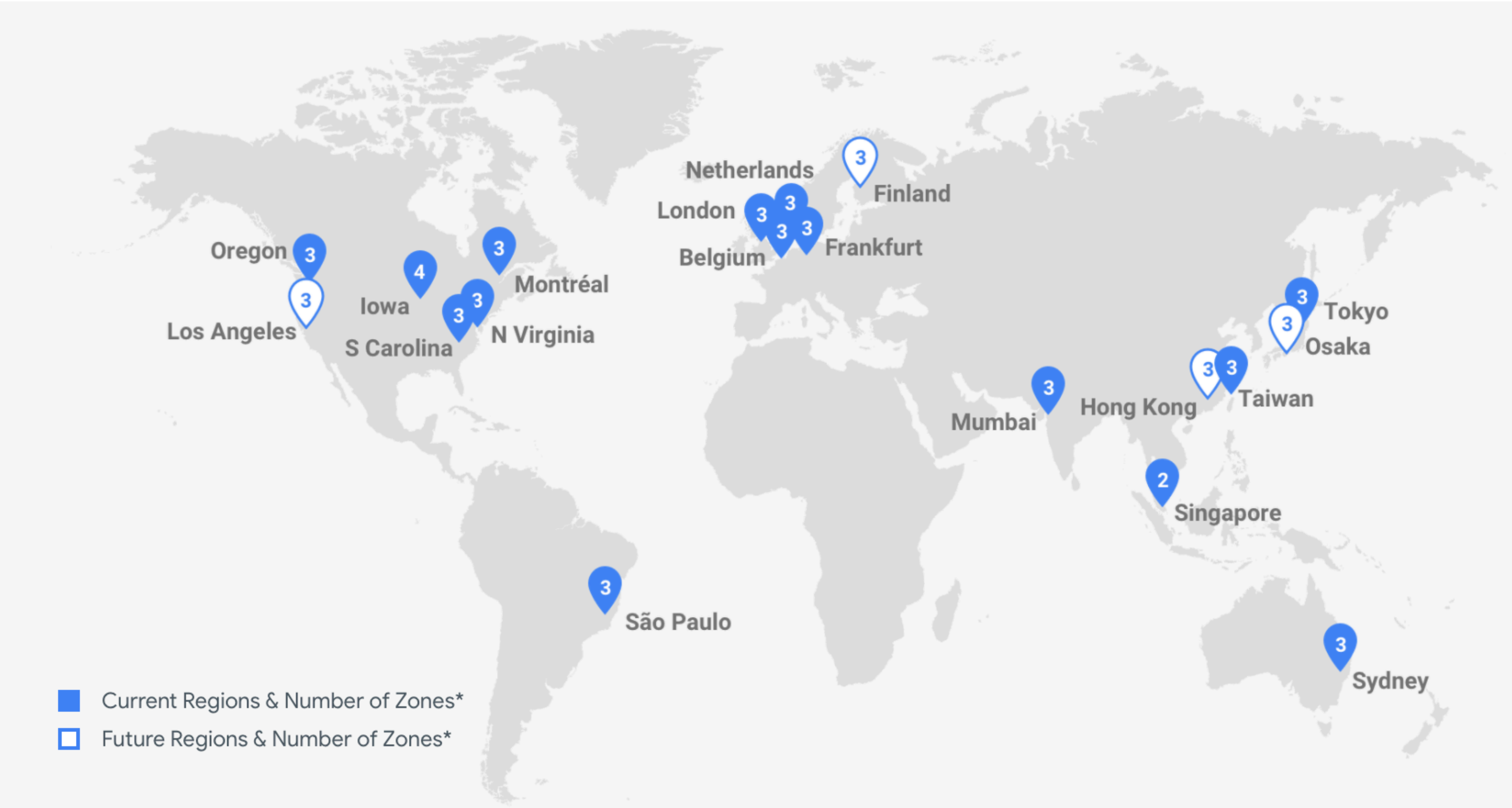
- DNS load balancing
- TTL < 5 minutes
- 500+ IP addresses for ‘Search’



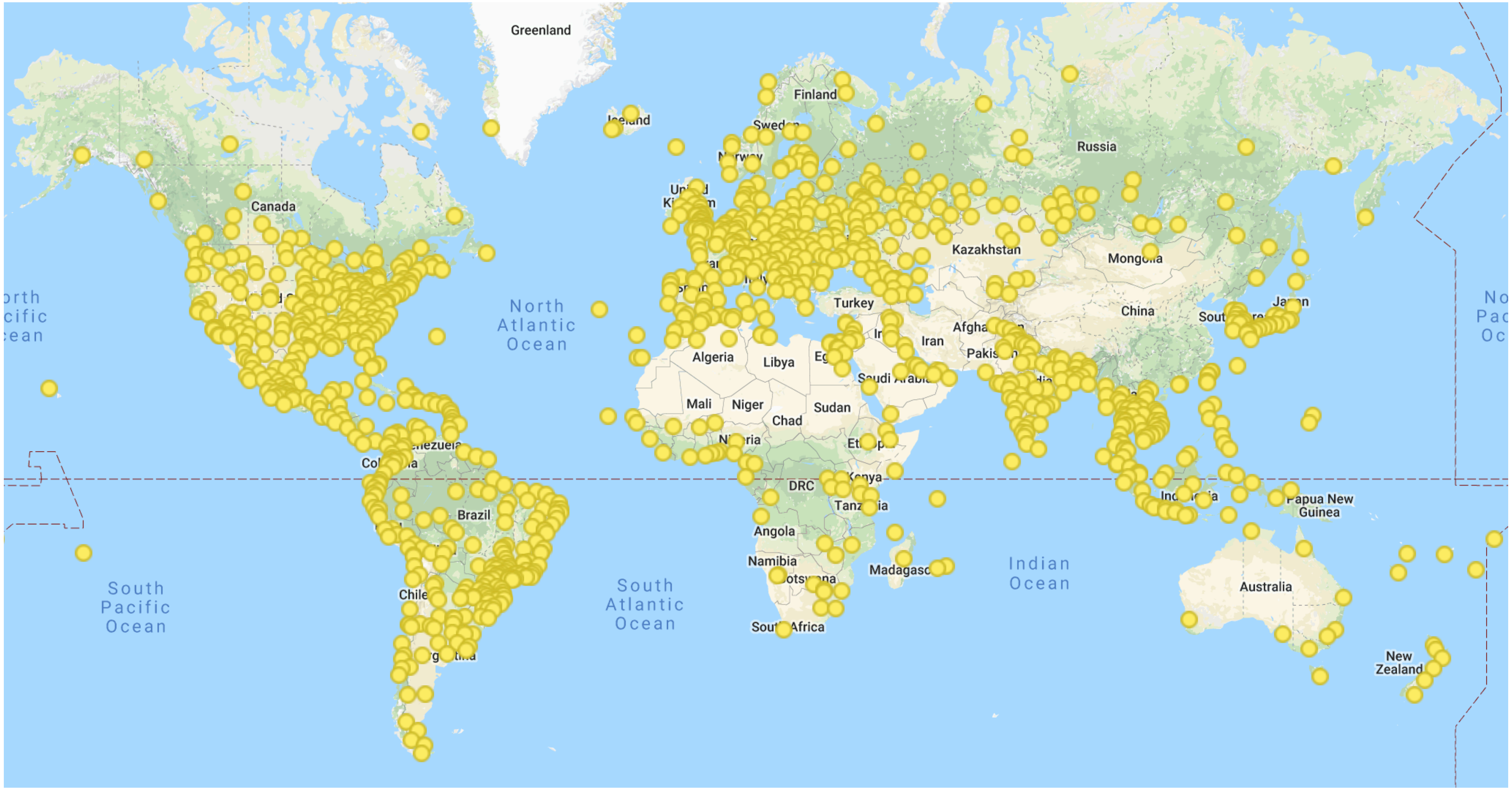
```
lanhost496:~ johan$  dig +noall +answer google.com
google.com.     1   IN  A   173.194.78.102
google.com.     1   IN  A   173.194.78.100
google.com.     1   IN  A   173.194.78.138
google.com.     1   IN  A   173.194.78.101
google.com.     1   IN  A   173.194.78.113
google.com.     1   IN  A   173.194.78.139
lanhost496:~ johan$
lanhost496:~ johan$  dig +noall +answer google.com
google.com.     300 IN  A   173.194.65.113
google.com.     300 IN  A   173.194.65.101
google.com.     300 IN  A   173.194.65.102
google.com.     300 IN  A   173.194.65.138
google.com.     300 IN  A   173.194.65.100
google.com.     300 IN  A   173.194.65.139
lanhost496:~ johan$  dig +noall +answer google.com
google.com.     299 IN  A   173.194.65.139
google.com.     299 IN  A   173.194.65.100
google.com.     299 IN  A   173.194.65.138
google.com.     299 IN  A   173.194.65.102
google.com.     299 IN  A   173.194.65.101
google.com.     299 IN  A   173.194.65.113
```



The Data Centers

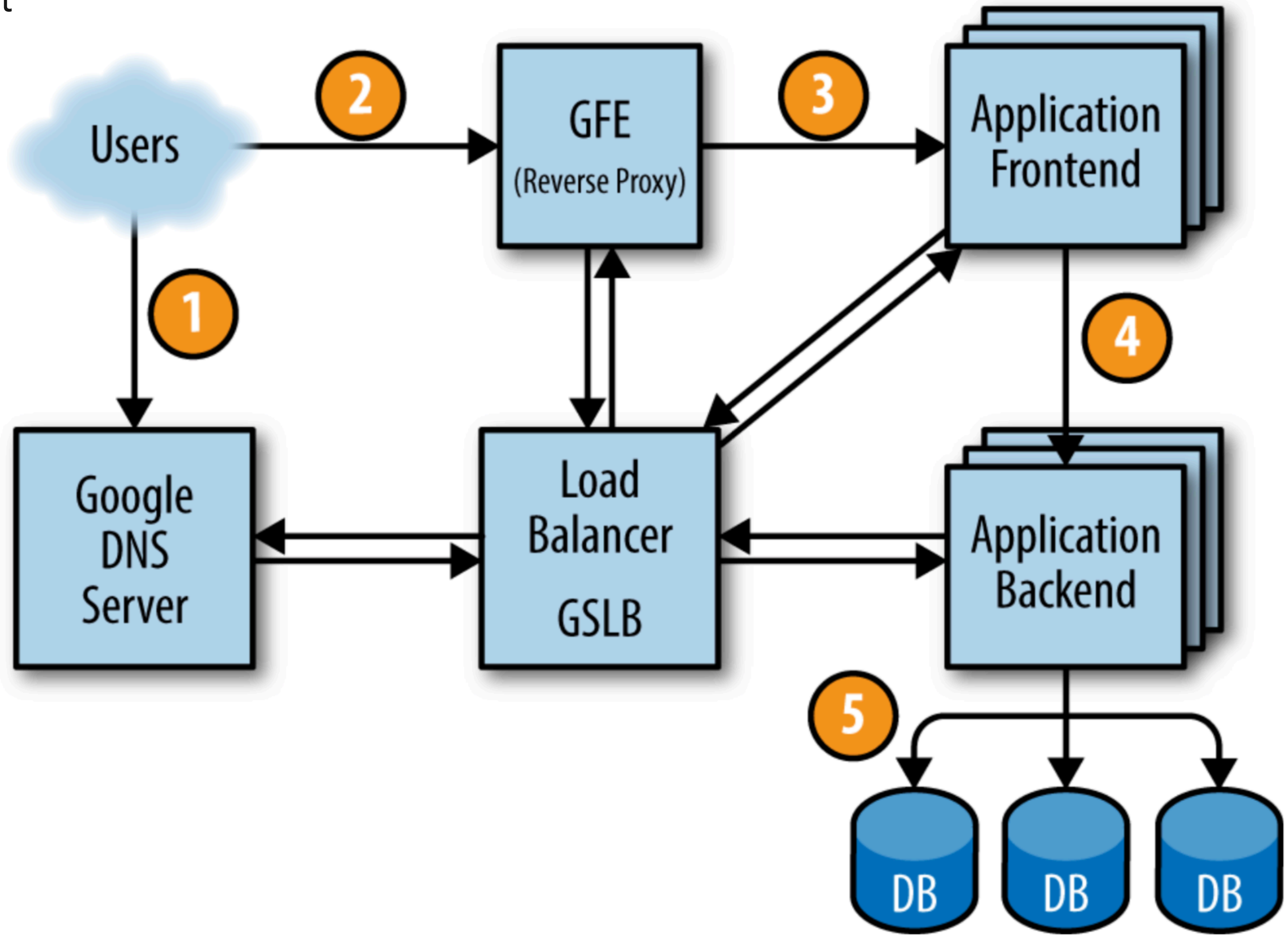


Google Global Cache (GGC) — Edge Nodes



Once you are inside

The life of a request



First generation cloud applications

- Typically a monolithic application
  - Database, mail server, web site, etc.

- Replicated a physical compute environment in the cloud

- Still gives many advantages over physical hosting
  - Host upgrade with zero-downtime using VM migration
  - Fail-over support on host failure
  - Better monitoring possibilities
  - "Somebody else's problem"

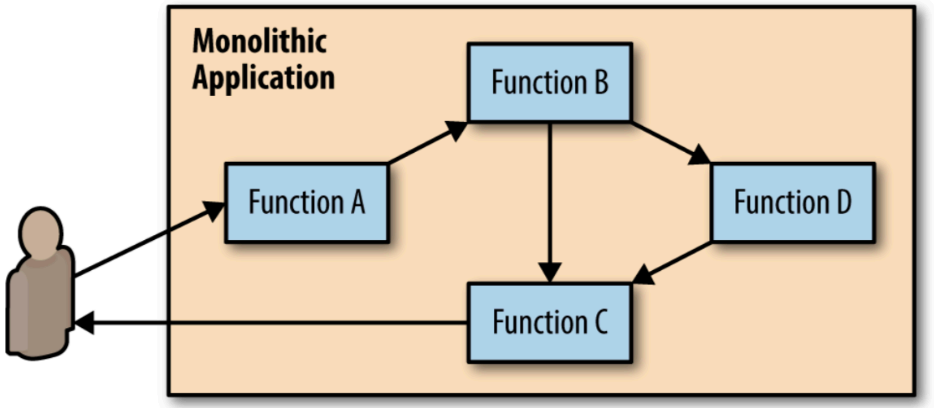
Second generation cloud applications

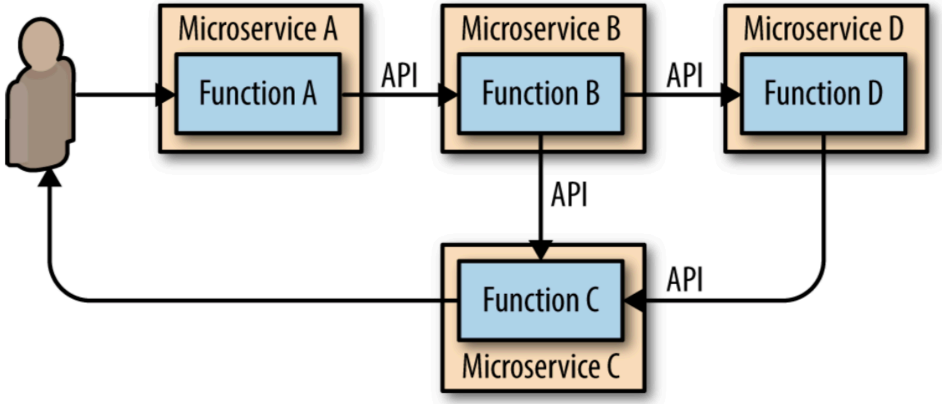
- Cloud native
  - Designed to run in a cloud environment
  - Benefit from a software-defined infrastructure
  - Fault-tolerant and distributed
  - Horizontal scaling of virtual resources
    - "Infinite scalability"
  - Configuration and orchestration

Micro-services

- Applications made up of independent building blocks
  - Not a monolith

- Services designed, deployed, and operated independently





Jeff Bezos Mandate

“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 interprocess communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network.

It doesn't matter what technology they use. HTTP, Corba, Pubsub, custom protocols -- doesn't matter. Bezos doesn't care.

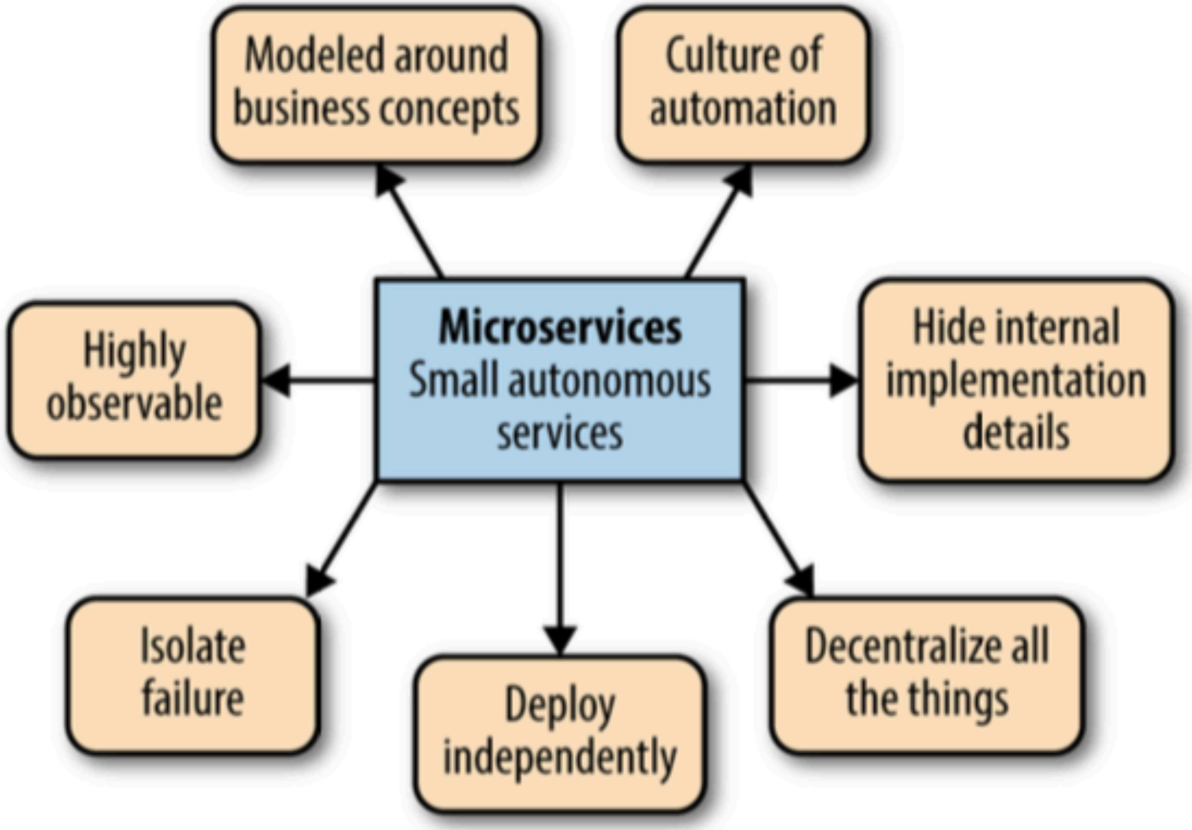
All service interfaces, without exception, must be designed from the ground up to be externalizable. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world. No exceptions.

Anyone who doesn't do this will be fired.

Thank you; have a nice day! “

Microservices Basics

- Small, focused and doing one thing well
- Several connected microservices replaces one monolithic service
- All communication via network calls
  - Expose service via API
  - Minimize coupling between services
  - Minimize coupling between users and services
- All services have a clear business goal


Modeled around business concepts
Culture of automation
Highly observable
Microservices
Small autonomous services
Hide internal implementation details
Isolate failure
Deploy independently
Decentralize all the things

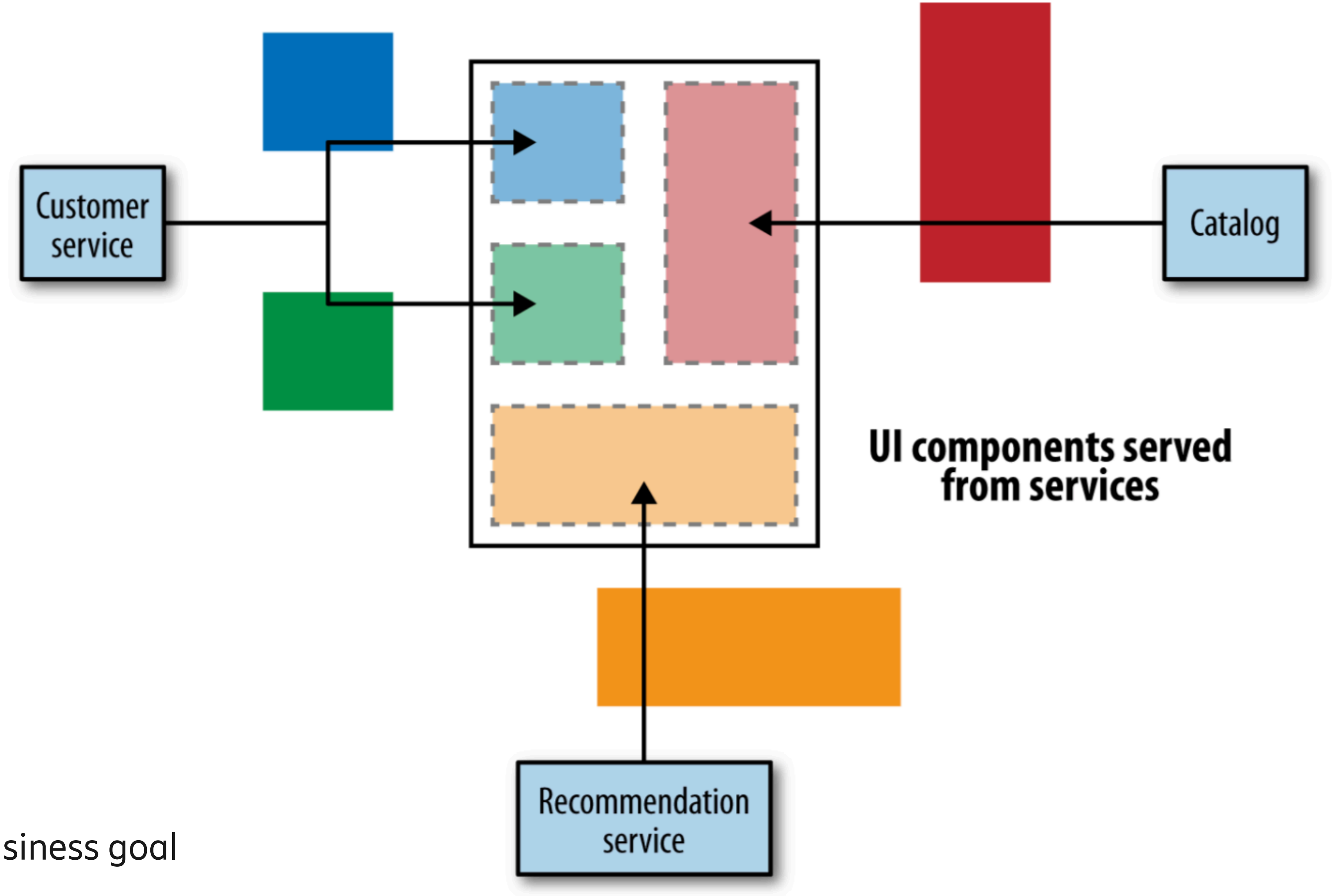
Micro-service benefits

- Technological heterogeneity (use right tool for the job)
- Distributed systems more resilient to failure of single nodes
- Monitor and scale each component independently
- Composability
- Aligns well with focused teams organizationally
- Bug isolation and fixing easier, only affects single micro-service

Micro-service drawbacks

- Distributed systems difficult to reason about and debug
  - Network calls between micro-services slower than local in-process calls
- APIs must be versioned and never broken
- Harder to make sweeping refactoring changes across them all
  - (This is considered bad practice, but let's be realistic…)

Microservices as UI components



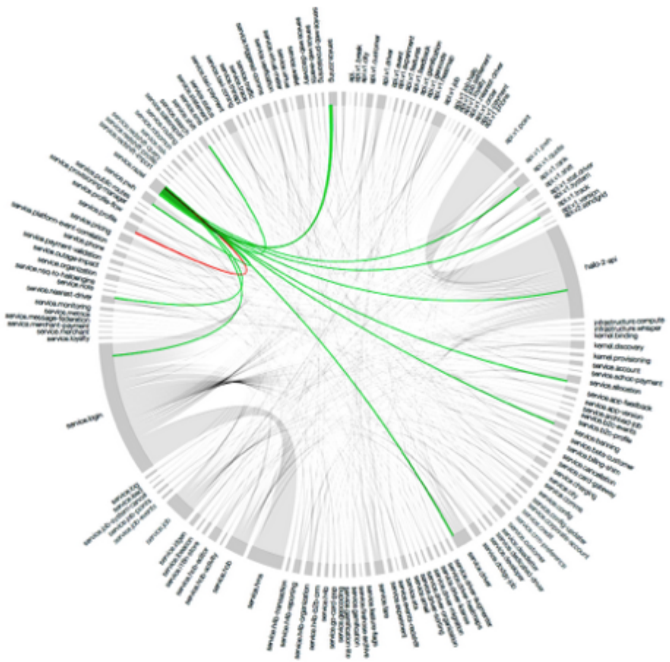
Each service has a business goal

Microservices are clean and simple.

At least in theory

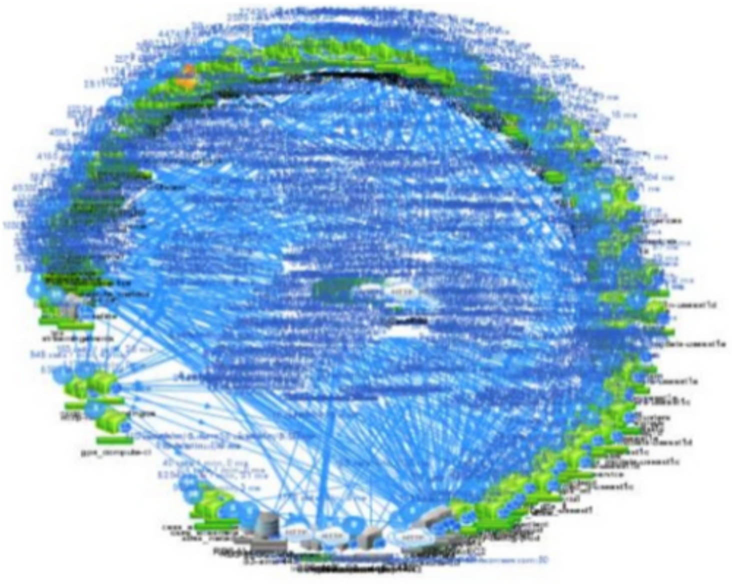
In reality, not so much...

450+ microservices



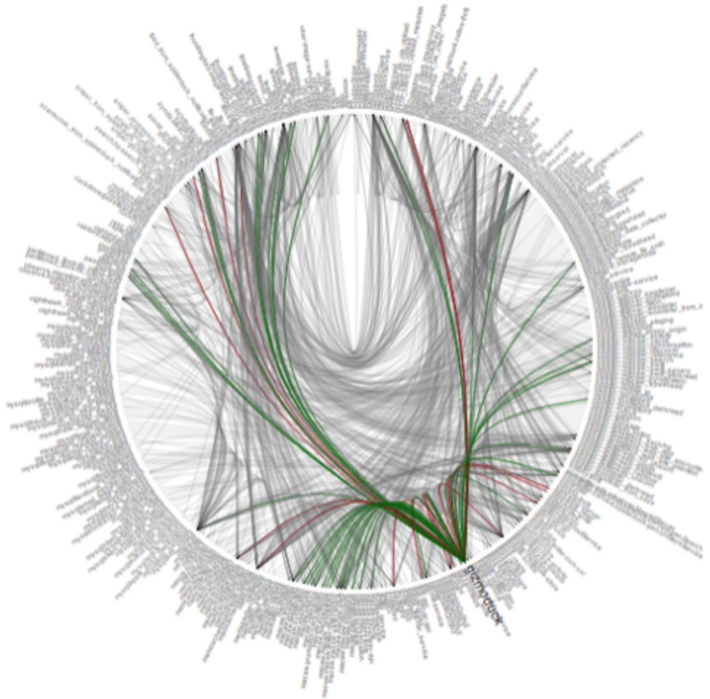
HAIL O

500+ microservices



NETFLIX

500+ microservices



Slide: Adrian Cockroft

What makes a good microservice?

- Loose coupling
  - Easy to replace a service
  - Often means to we need to limit the ways a service may be used

- High cohesion
  - Related behaviour sit together, avoid updating several services to fix one issue
  - Identify boundaries for problem domains

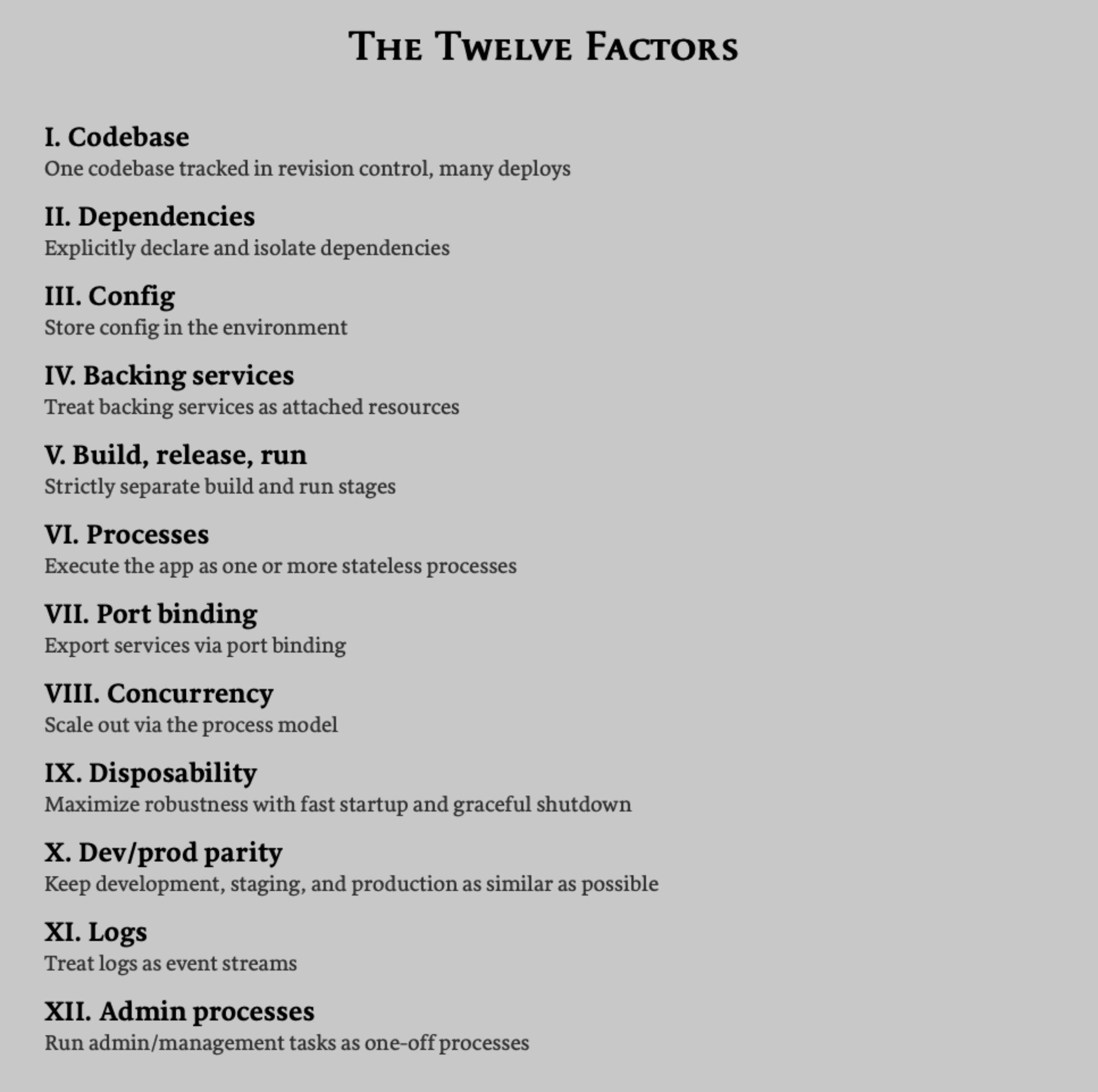
HEROKU

- Heroku is a PaaS and "app" is similar to a microservice

- Twelve rules for good design
- https://12factor.net

Rules are for the obedience of fools and guidance of wise men.

Douglas Bader



The Twelve Factors

I. Codebase
One codebase tracked in revision control, many deploys

II. Dependencies
Explicitly declare and isolate dependencies

III. Config
Store config in the environment

IV. Backing services
Treat backing services as attached resources

V. Build, release, run
Strictly separate build and run stages

VI. Processes
Execute the app as one or more stateless processes

VII. Port binding
Export services via port binding

VIII. Concurrency
Scale out via the process model

IX. Disposability
Maximize robustness with fast startup and graceful shutdown

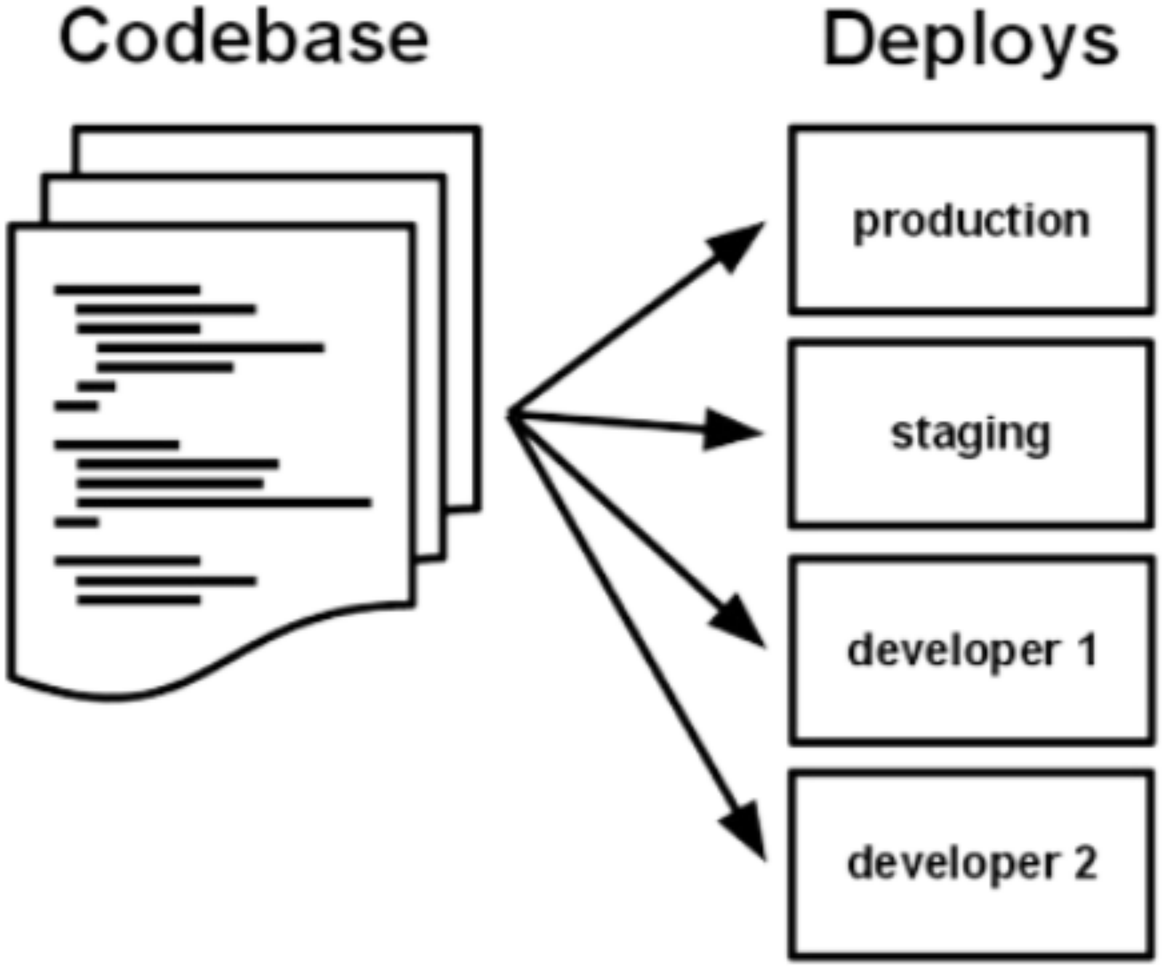
X. Dev/prod parity
Keep development, staging, and production as similar as possible

XI. Logs
Treat logs as event streams

XII. Admin processes
Run admin/management tasks as one-off processes

I. Codebase

- One repo, one code base
- One code base, many deployments.



II. Dependencies

- Never rely on implicit existence of system-wide packages or tools
- Declare all dependencies using a dependency manifest
- Utilize tools for dependency isolation
- Python: `pip` and `virtualenv`

- This simplifies deployment and reduces faults

III. Config

- Config is everything that may vary between different deployments (staging, production, development, etc.)
  - Location and access to storage and databases and other backing services
  - Credentials
  - Deployment names
- Store config in the environment, not in the code
  - Environment variables populated by your deployment tools

- A litmus test is if can release the code base as open source without sharing sensitive information

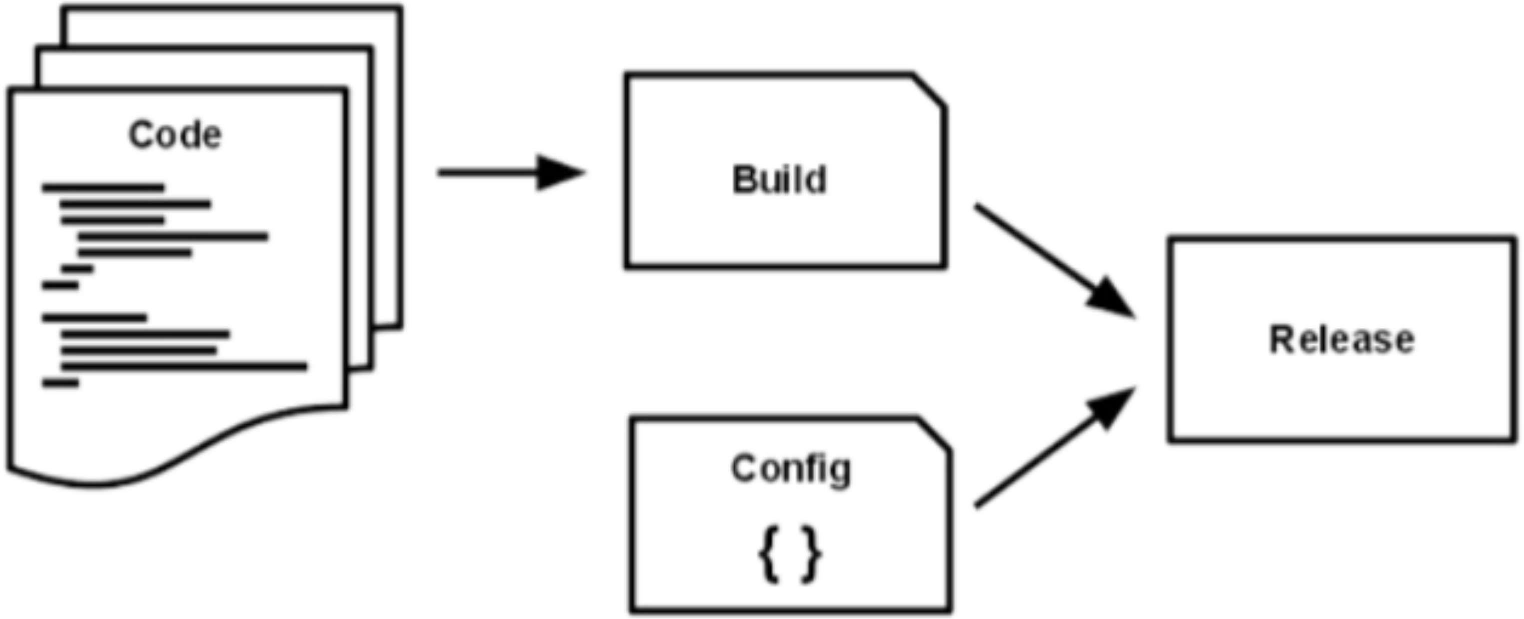
IV. Backing Services

- A backing service is any service that is used as part of the normal operation
  - Datastores (MySQL, MongoDB), messaging systems (RabbitMQ), SMTP (Postfix), Caches (Memcached)

- Treat all services in the same fashion. Make no distinction between local and global services
  - Allow for services to be switched out (and this should be done in the config)

V. Build, release, run

- Strictly separate build and run stages
- Builds are initiated whenever code is added or modified
- Every release have a unique ID
- The release version can be automatically deployed for testing and or production

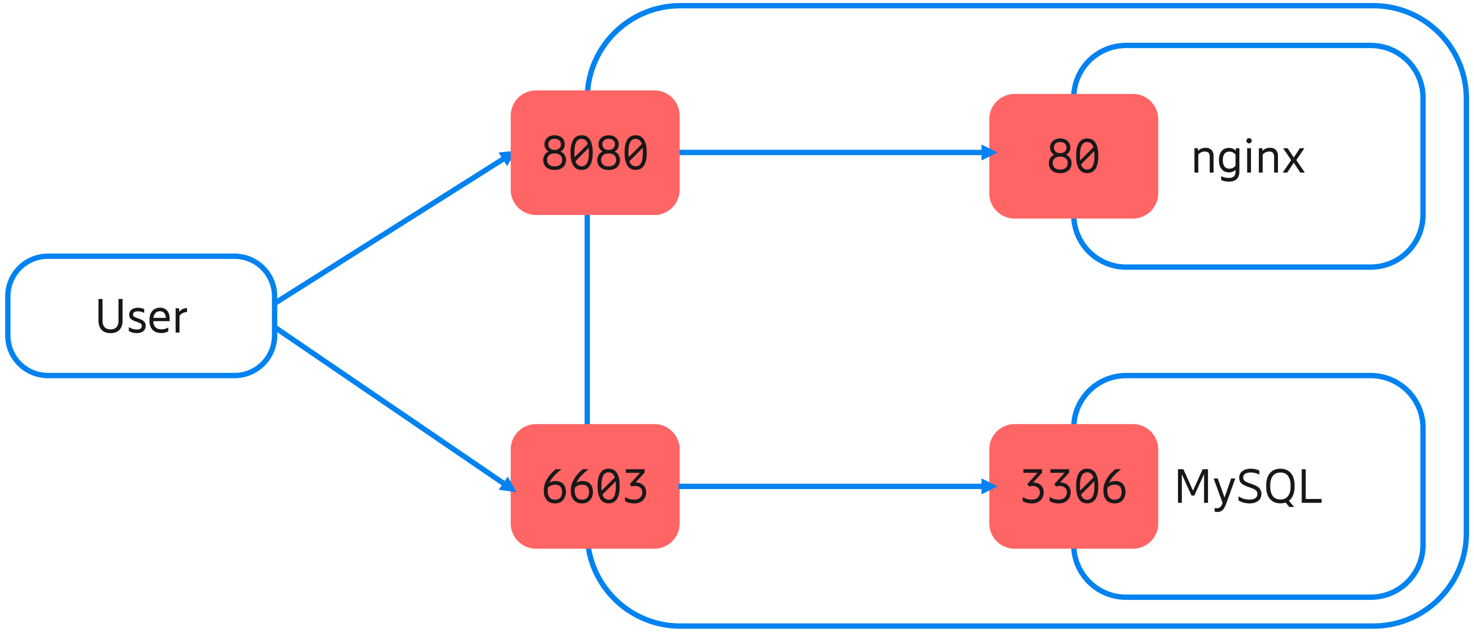


VI. Processes

- Implement the service as one or more stateless processes
- Processes are stateless and share nothing. Any data that needs to be persistent must be stored on a stateful backing service, e.g. a database
- The memory of a process may be used for example for caching, but must not be critical for correct operation
- Assume that process can be terminated at any point in time

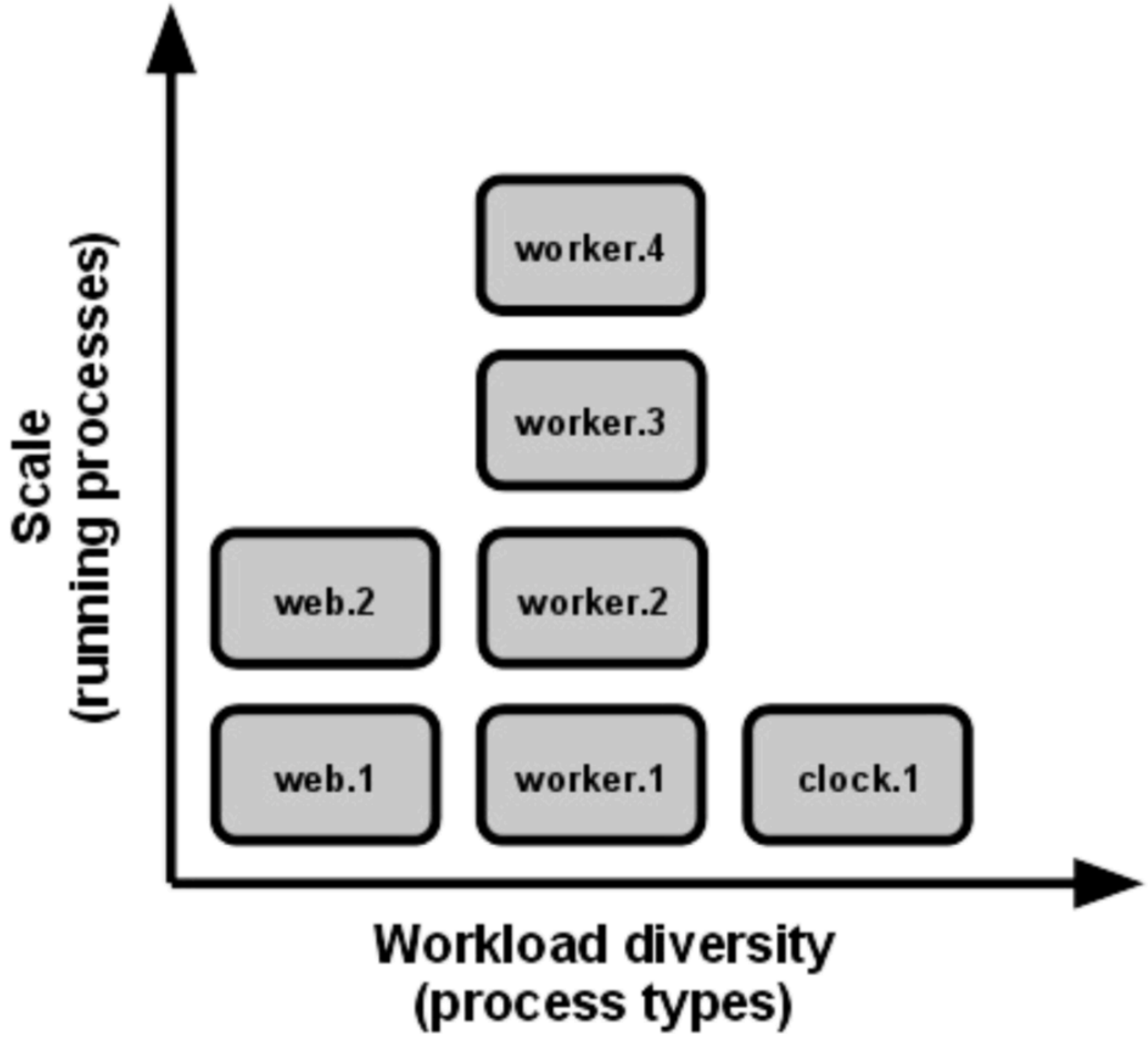
VII. Port binding

- Expose your service by binding to a port. That is how your service integrates.
- Locally the service may be available as http://localhost:5001/
- In deployment, a routing layer handles requests from users of the service



VIII. Concurrency

- A VM is limited in how large it can become (vertical scaling). Instead make processes the unit of scalability.
- If the processes are stateless, scaling becomes almost trivial.



IX. Disposability

- Services are disposable, i.e. the can be started or stopped at a moment's notice. This facilitates fast elastic scaling, rapid deployment of code or config changes.
- Important to minimize startup time in order to provide more agility.
- Support graceful shutdown when a SIGTERM is received.
- Be robust against sudden death.

X. Dev/Prod Parity

- Keep development, staging, and production as similar as possible. Historically, there have been a substantial gap between development and production:
  - The time gap: development is a long term activity
  - Personnel gap: Developers write code, ops engineers deploy it
  - The tools gap: the stack for development and deployment typically differs

- Make the time gap small:
  - A developer may write code and have it deployed hours or even just minutes later.
  - Make the personnel gap small: developers who wrote code are closely involved in deploying it and watching its behavior in production.
  - Make the tools gap small: keep development and production as similar as possible.

XI. Logs

- Treat logs a event streams, not only writing to file locally.

- Logs are the stream of aggregated, time-ordered events collected from output streams of all running processes and backing services.

- In a deployed system each process' stream will be captured by the execution environment, collected together will all other streams from the app, and then analyzed (maybe triggering alarms) and finally archived.

XII. Admin Processes

- Run admin/management tasks as one-off processes
  - Patching, database migration, fixing errors, terminal (REPL), etc.

- Use the same execution framework and environment as the services themselves
  - Identical environment as the regular long running jobs
  - The admin code should be shipped with the rest of the code to avoid synchronization issues.

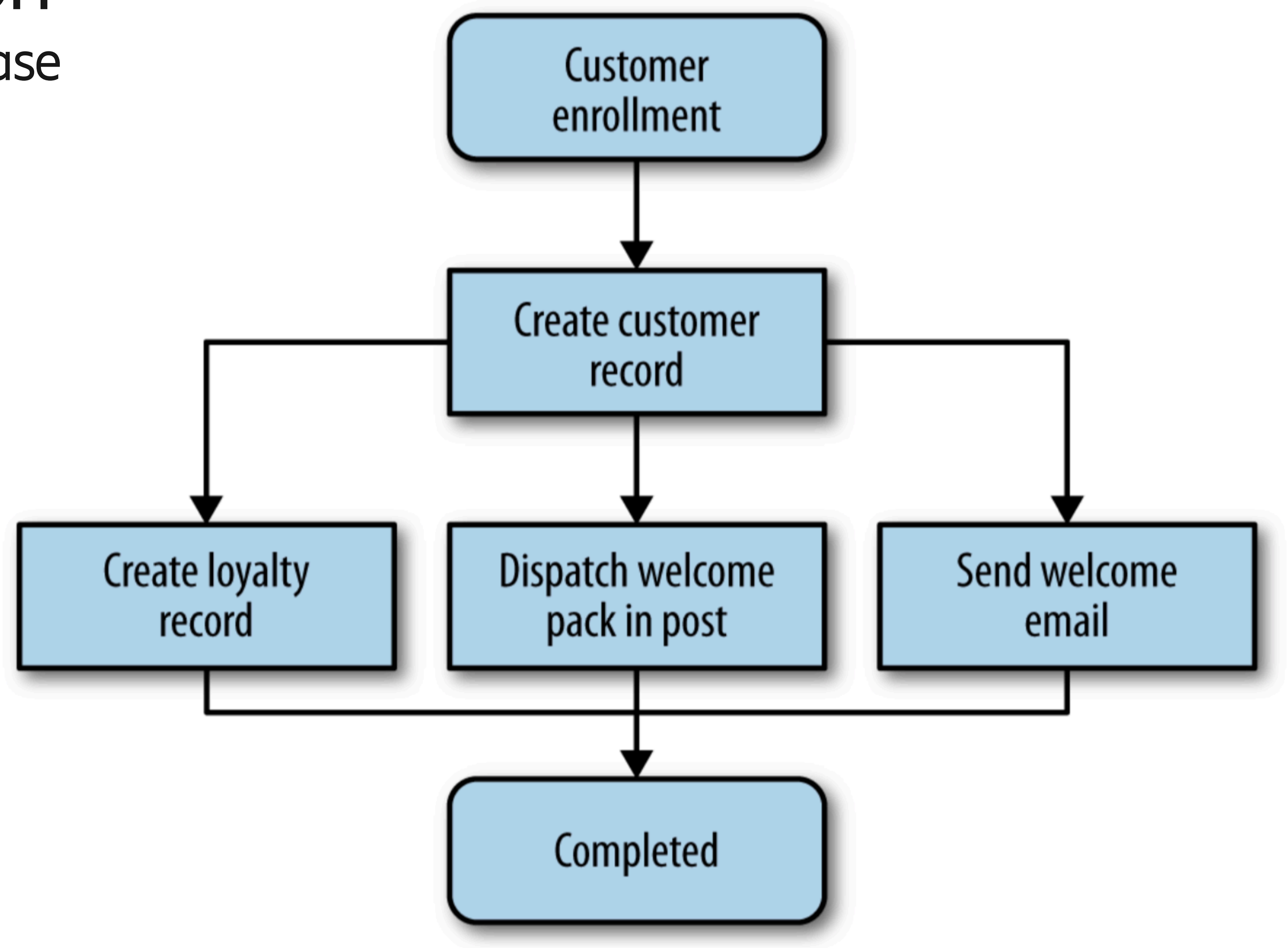
Fallacies of Distributed Computing

Peter Deutch (and James Gosling) around 1995 or so:

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn't change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous.

Integration

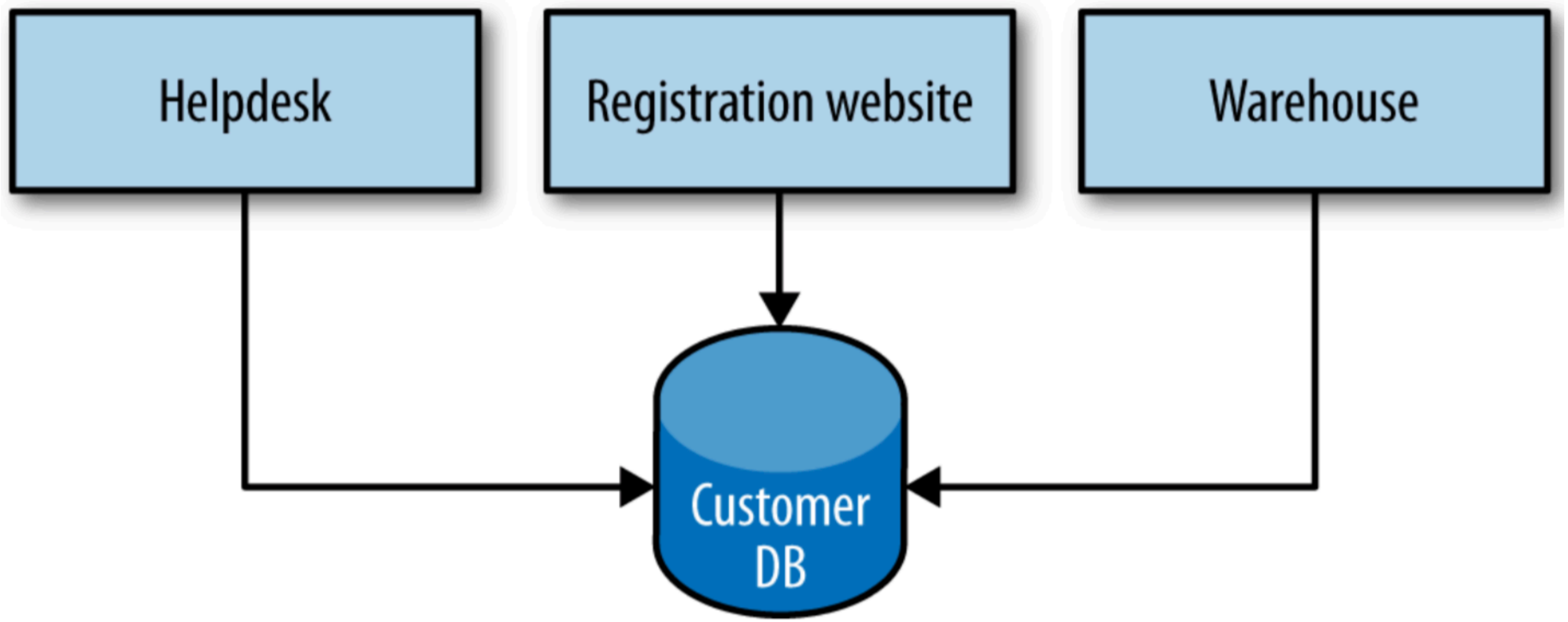
Example use case



How to best design loosely coupled components?

Integration

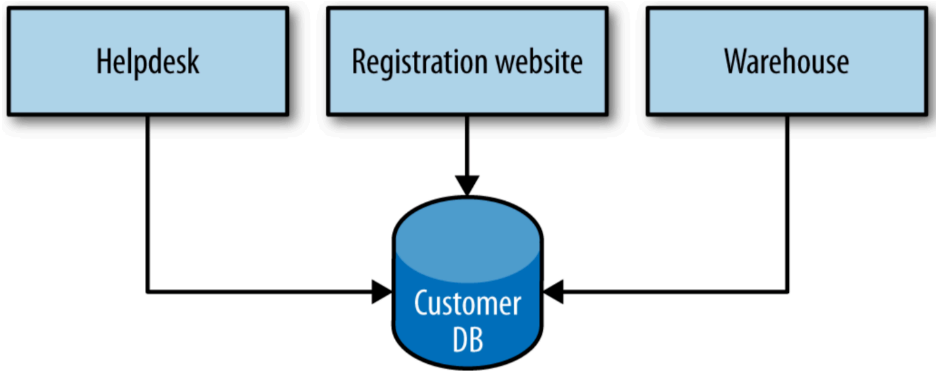
The naive approach: shared storage



Why is this a bad design?

Integration

The naive approach: shared storage

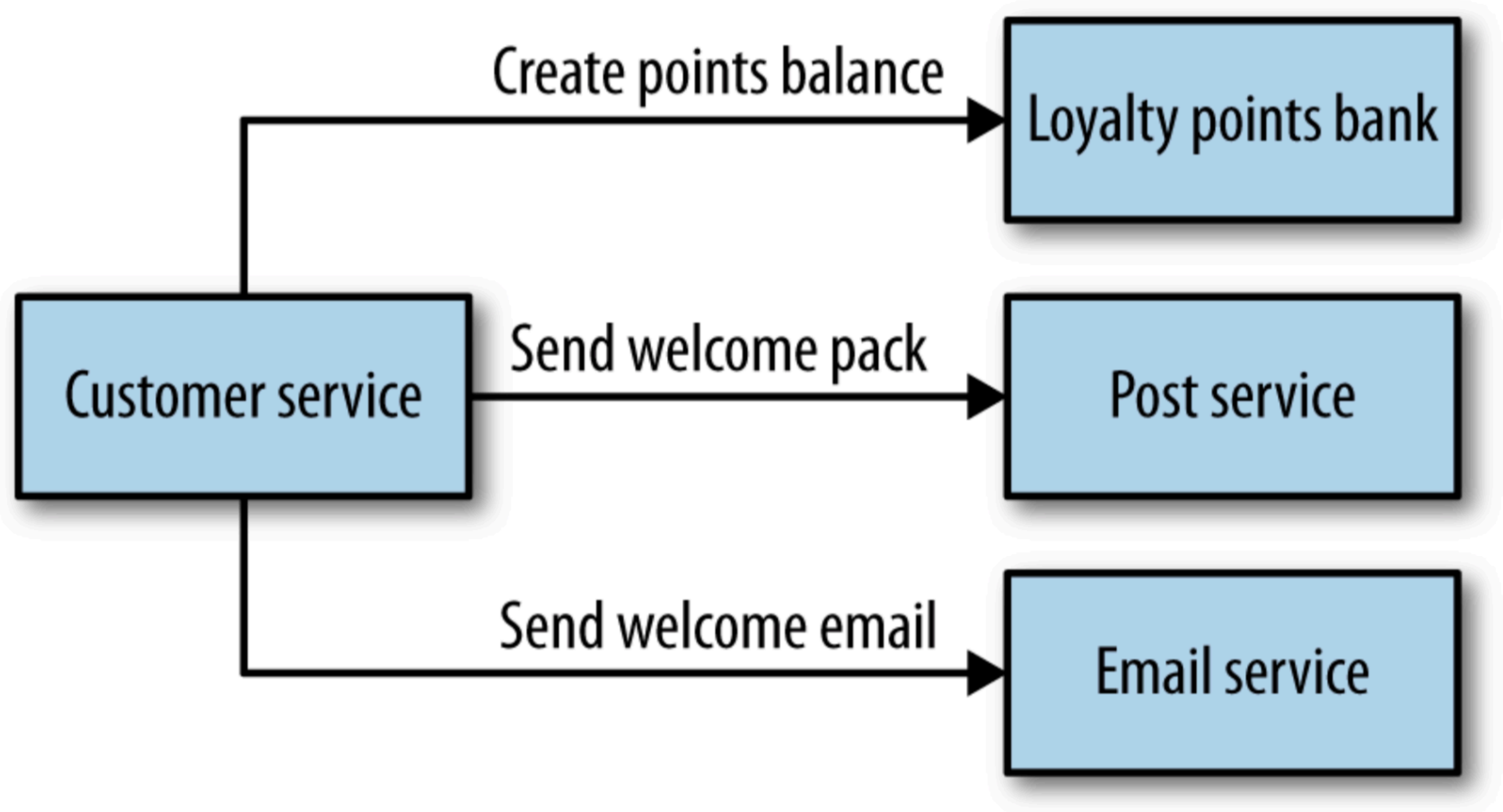


- Exposes internal details of the database, which ties consumers to a specific technology
  - Goodbye loose coupling!

- Logic around database organisation is spread into several places
  - Goodbye cohesion!

Integration

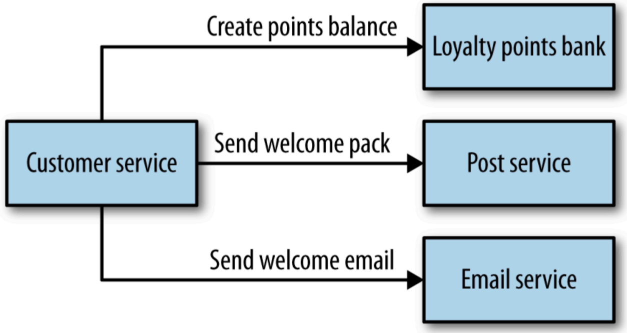
The synchronous approach: Request/response



Why are the pros & cons?

Integration

The synchronous approach: Request/response with RPC

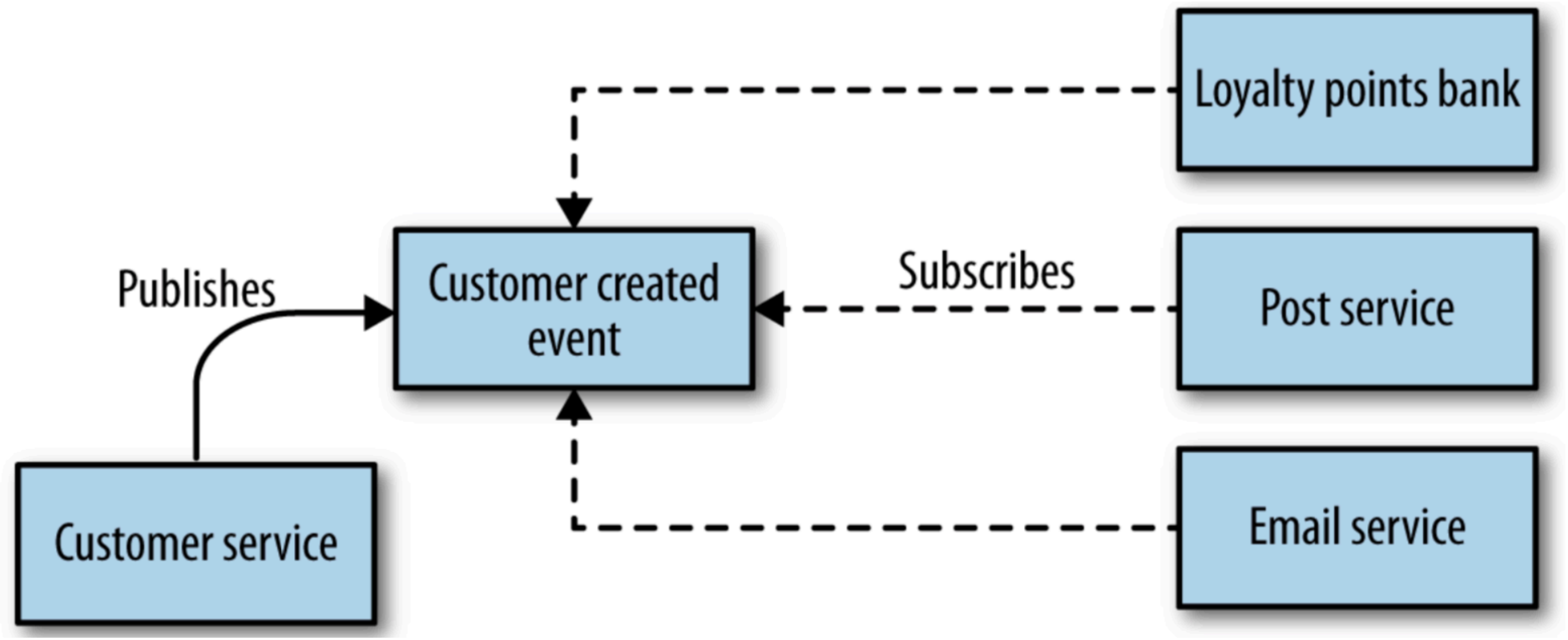


- Remote calls are really different from local calls...

- Performance
  - Cost of marshalling/unmarshalling
  - Must be careful in API design to avoid overhead
- Failures are hard to detect or understand. Difficult to distinguish between faults.
  - Network failing (or worse, just slow)
  - Servers failing
  - Service failing
  - Call is incorrect
- Brittleness
  - Changes to server requires changes to clients
  - This leads to lock-step releases (not trivial)

Integration

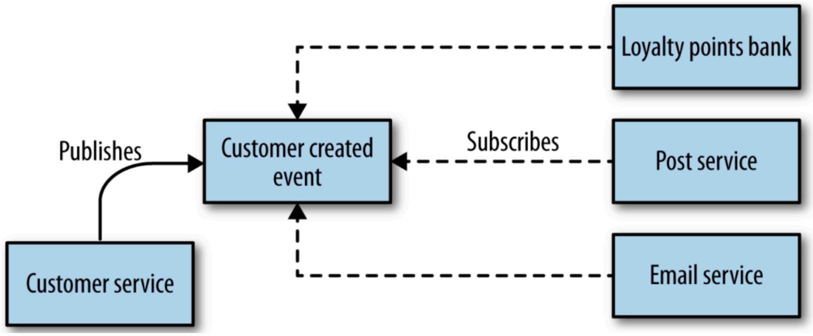
The asynchronous approach: Event based



Why are the pros & cons?

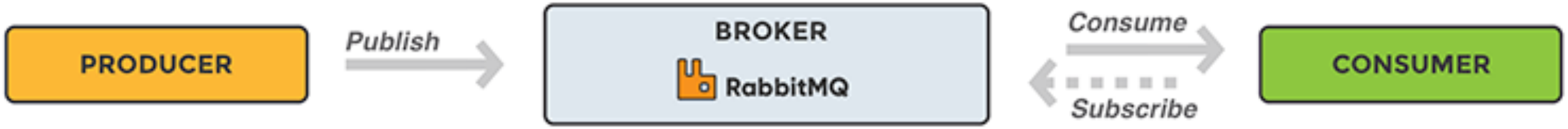
Integration

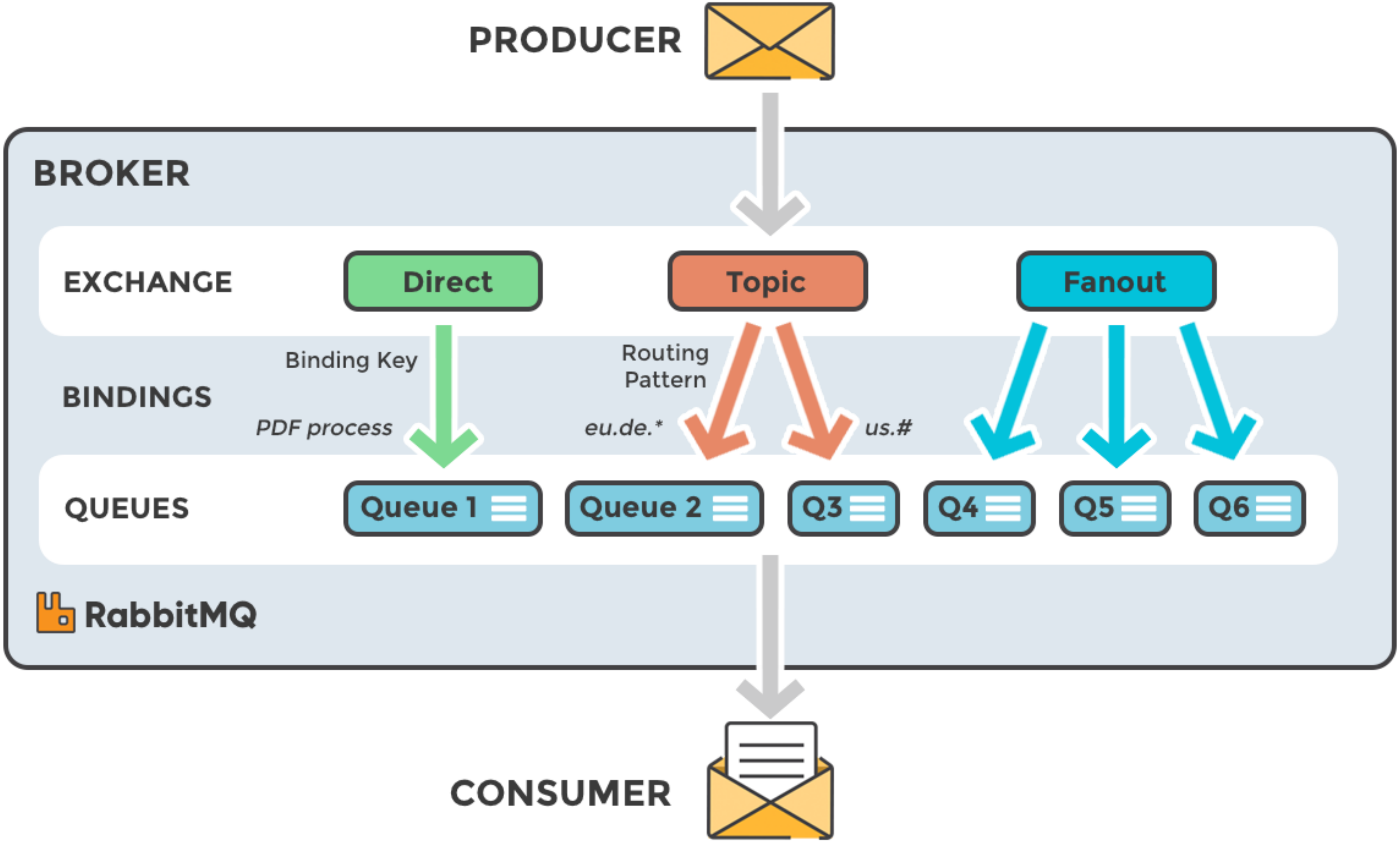
The asynchronous approach: Event based



- Message brokers, e.g. RabbitMQ, Kafka, etc.
  - Message formats like JSON, Thrift, Protocol Buffers , etc.

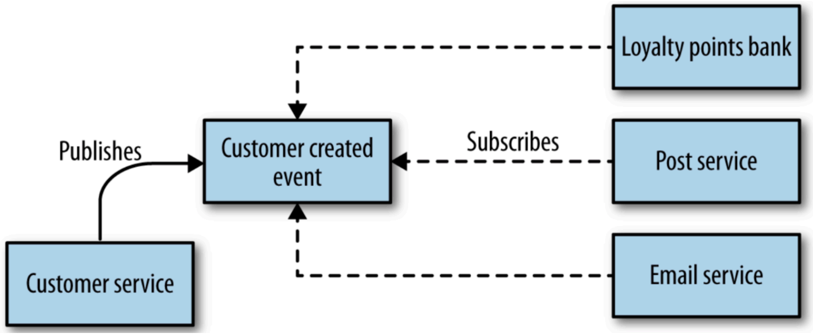
- Will guarantee on delivery, ordering, resilience, at most once delivery, etc.




PRODUCER
BROKER
EXCHANGE
Direct
Topic
Fanout
BINDINGS
Binding Key
PDF process
Routing
Pattern
eu.de.*
us.#
QUEUES
Queue 1
Queue 2
Q3
Q4
Q5
Q6
RabbitMQ
CONSUMER

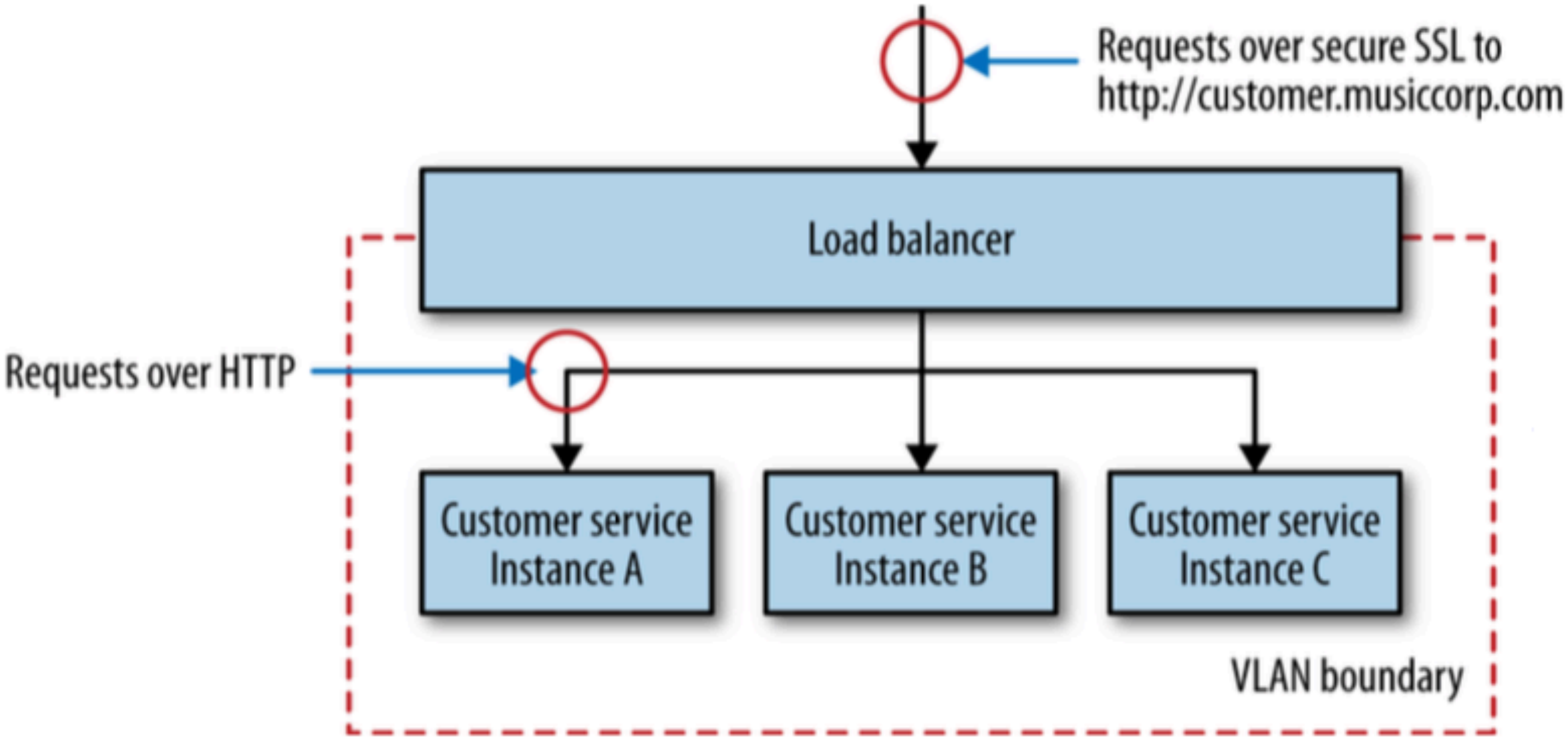
Integration

Asynchronous adds complexity



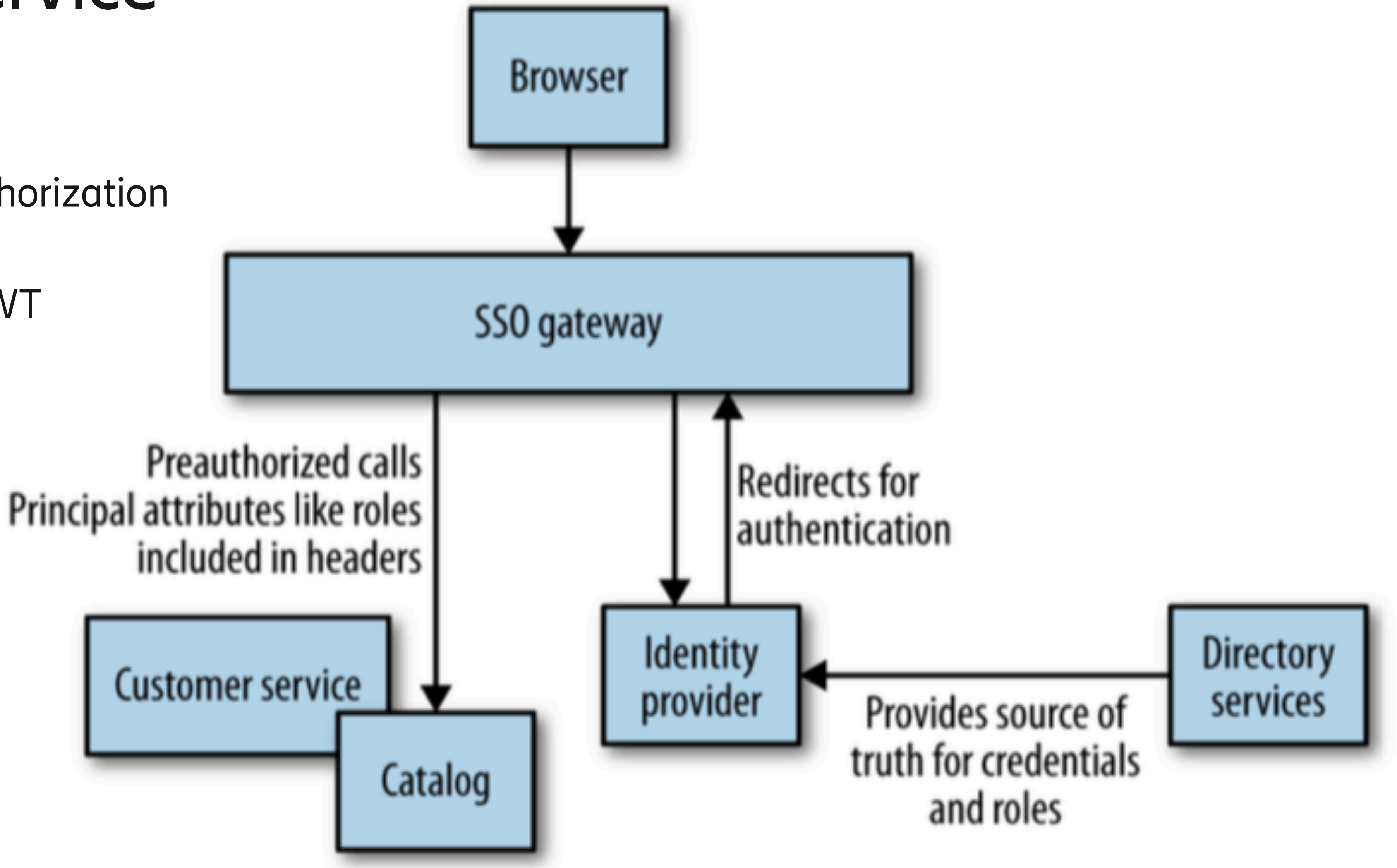
- The message broker is a non-trivial piece of software. Needs to be operated and maintained, and of course contains bugs.
- RPC/REST can also be implemented as asynchronous (does not make it any easier)
- Long running asynchronous requests and response becomes complicates in large systems where the client might not be around anymore due to failure or scaling.
- May require clients to check in every once in a while, check if status has updated

Secure your service

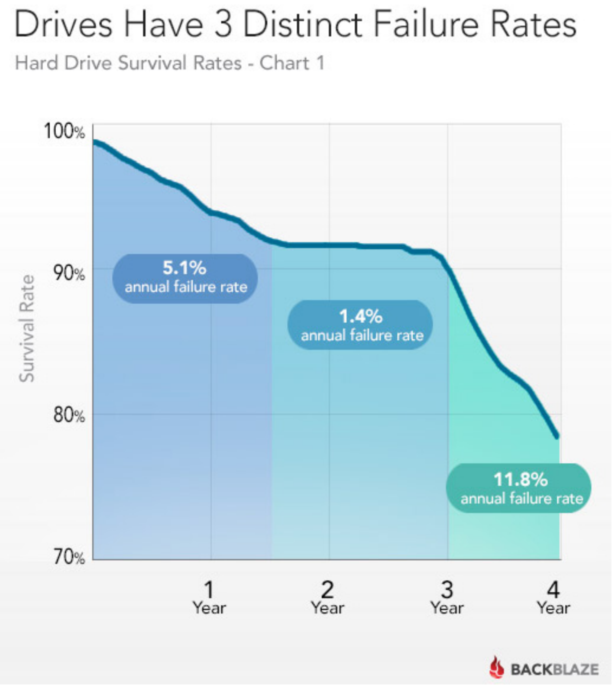

Requests over secure SSL to http://customer.musiccorp.com
Load balancer
Requests over HTTP
Customer service Instance A
Customer service Instance B
Customer service Instance C
VLAN boundary

Secure your service

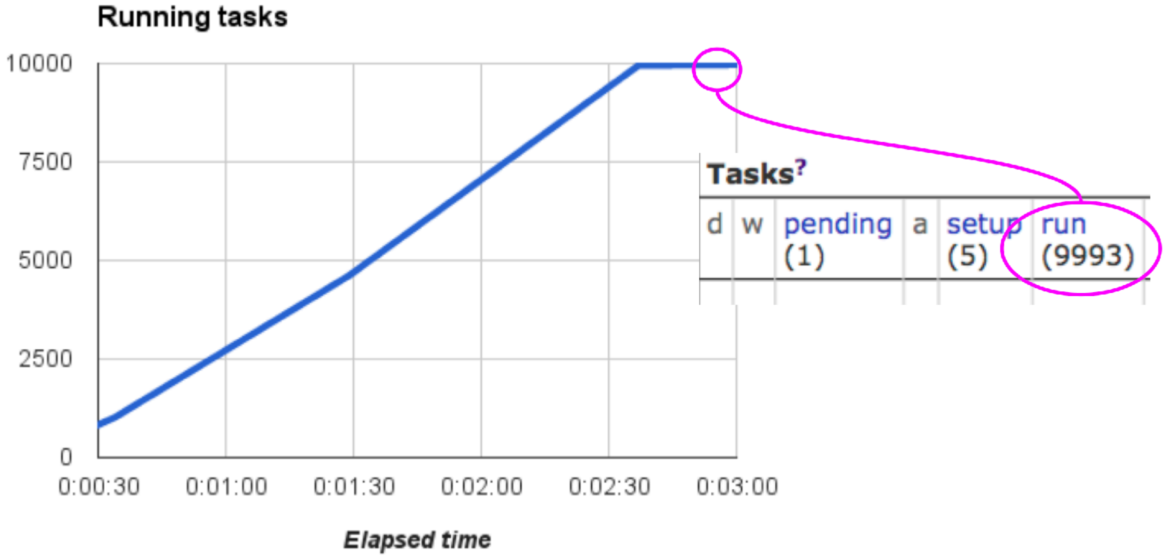
- Authentication and Authorization
- Single-sign on gateway
- JSON Web Tokens -- JWT



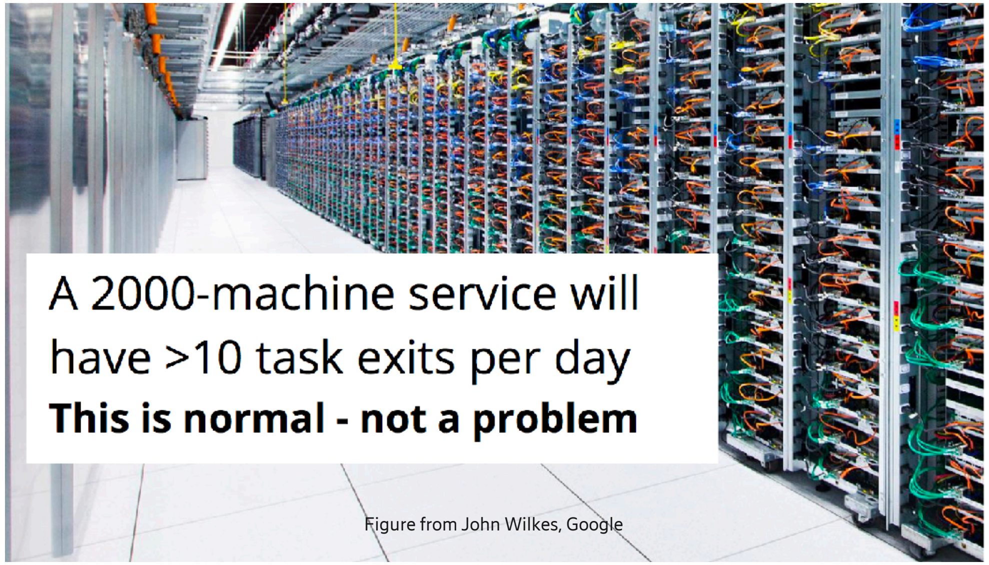
Failures - Live with it



“This means that 50% of hard drives will survive until their sixth birthday.”



- Disk failures:
  - Assume: 1 server has 6 disks,
  - Disk MTBF ≈ 4.5 years
  - 1000 servers in cluster => ~3.5 failures/day



A 2000-machine service will have >10 task exits per day

This is normal - not a problem

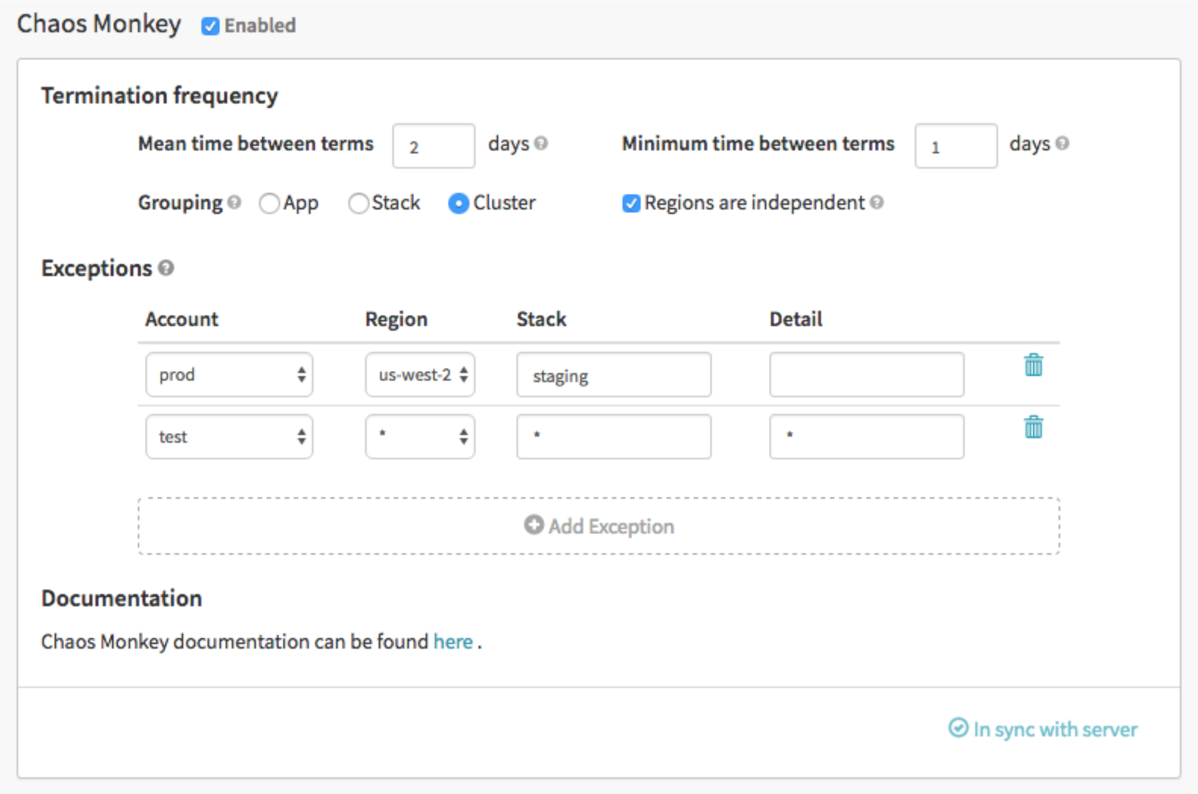
Figure from John Wilkes, Google

Create Havoc yourself

PRINCIPLES OF CHAOS ENGINEERING

> *“The discipline of experimenting on a distributed systems in order to build confidence in the system’s capability to withstand turbulent conditions in production.”*

Open source tool from Netflix



Chaos Monkey ☑ Enabled

Termination frequency

Mean time between terms 2 days

Minimum time between terms 1 days

Grouping ○ App ○ Stack ● Cluster

☑ Regions are independent

Exceptions

Account	Region	Stack	Detail
prod	us-west-2	staging	
test	*	*	*

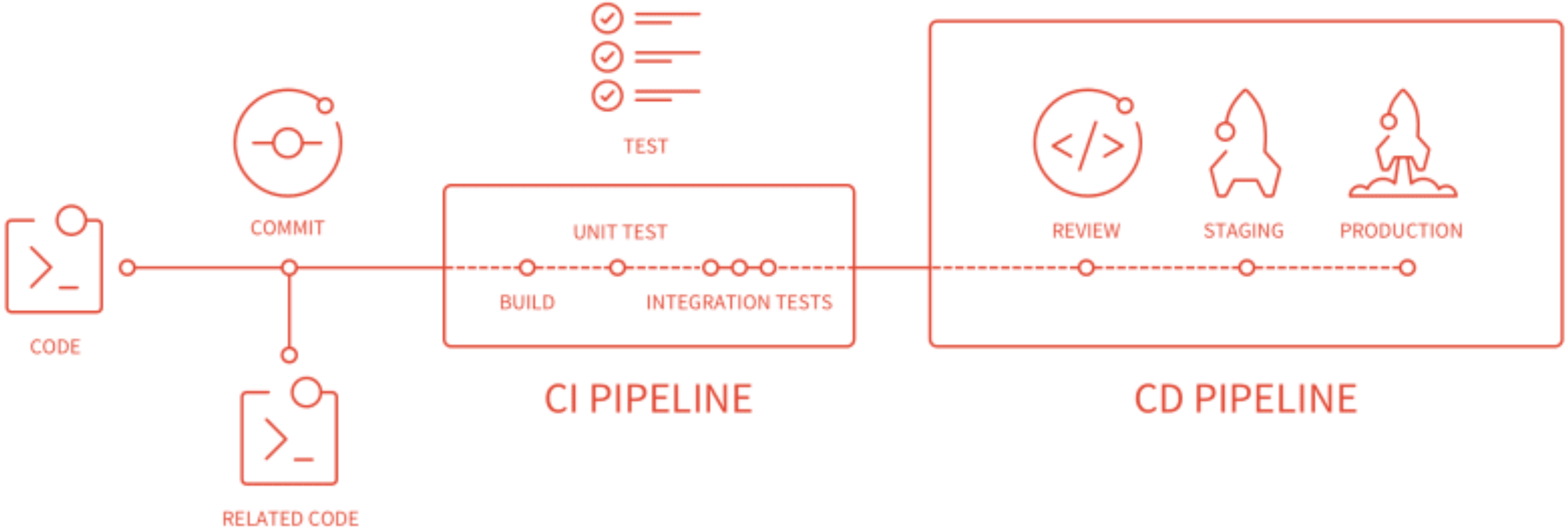
Add Exception

Documentation

Chaos Monkey documentation can be found here .

In sync with server

CI/CD - Continuous Integration/Continuous Deployment



https://gitlab.datahub.erdc.ericsson.net/root/demo

Scaling up or Scaling out?

- Scale up/down (or vertical scaling)
  - Add more memory and CPUs to a single server
  - Add more threads
  - Ideal for legacy stateful components such as databases
  - "Quickly" runs into an upper bound

- Scale out/in (or horizontal scaling)
  - Add more servers
  - Requires load balancing
  - **Cloud-native approach**

Cloud Native Application Design

- Design for failure, assume everything fails, and design backwards
  - Make sure to avoid single points of failure
- Goal: Applications should continue to function even if the underlying physical hardware fails, is removed, or replaced
- Couple loosely to make applications scale
  - Independent components
  - Design everything as a black box
- De-couple interactions
- Load-balance applications and clusters

fin

Back-up slides

Course project

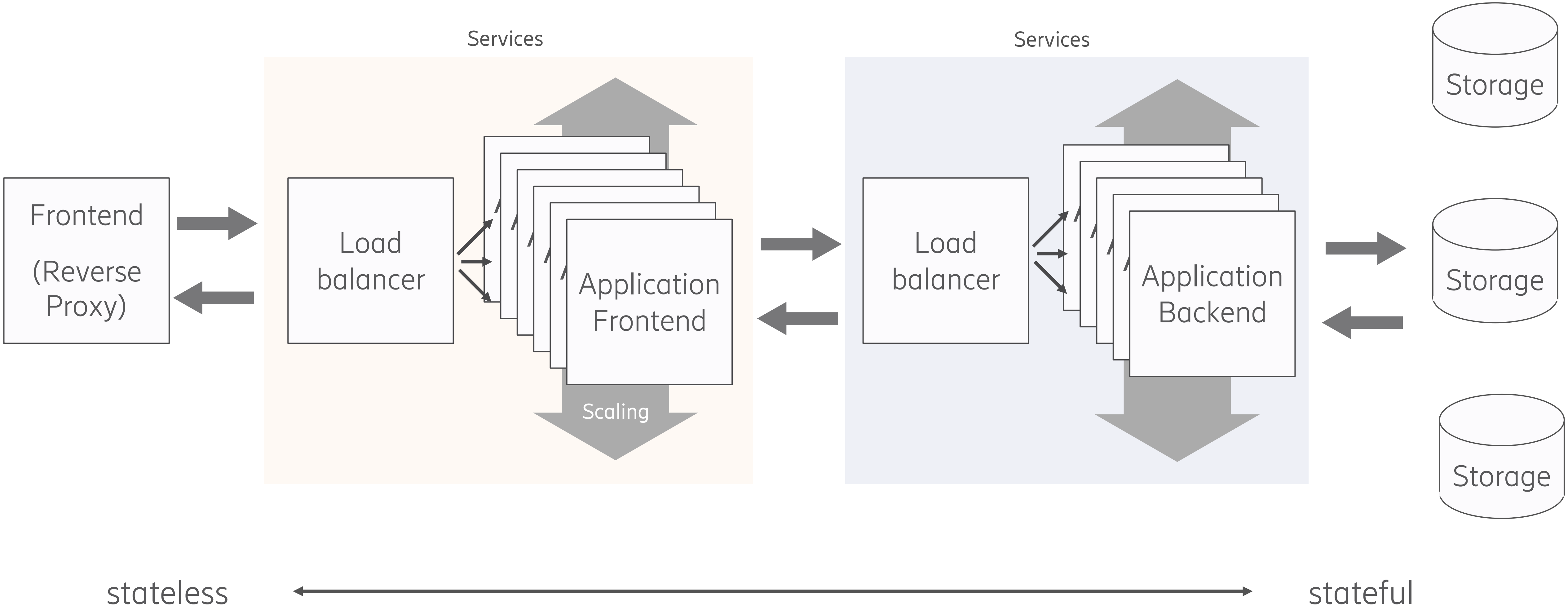
Cloudification of your favourite tool or application. It’s your choice, but make it scalable, multi-tenant, and resilient to faults.

Start by paper design. Provide and architecture as well as interaction patterns (sequence diagrams, etc.). Discuss how to handle faults and when and how to scale.

Once we have discussed the design individually per group, you should go ahead and implement it. But that is not enough. You should also provide a workload generator as well as a chaos animal.

The final project will be presented Nov 1.

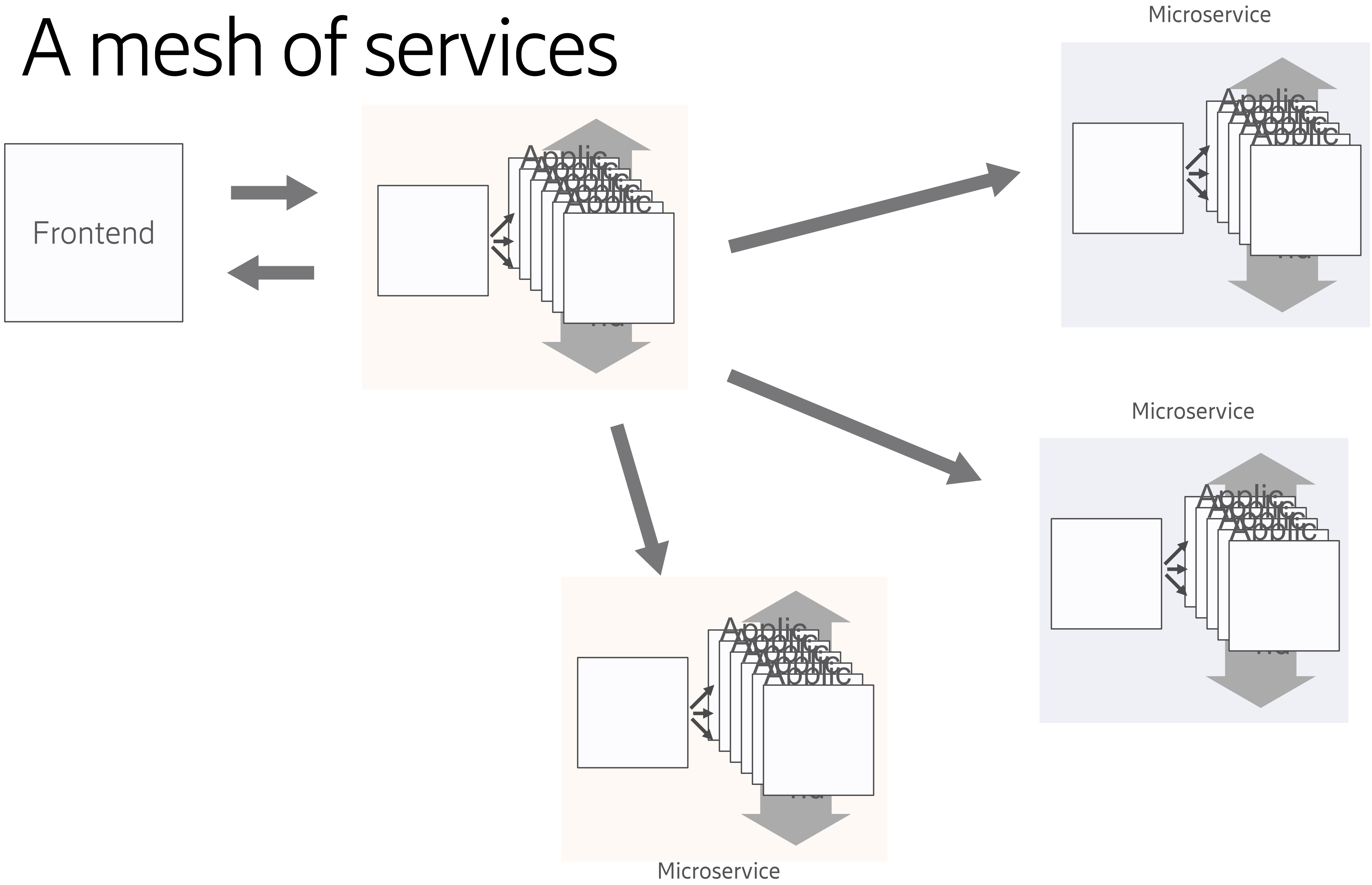
A typical cloud application architecture



Services

Services

A mesh of services


Microservice
Frontend
Applic
Microservice
Microservice