Aalto University
School of Electrical
Engineering

Extending to multiple nodes

14.2.2019

Santeri Paavolainen

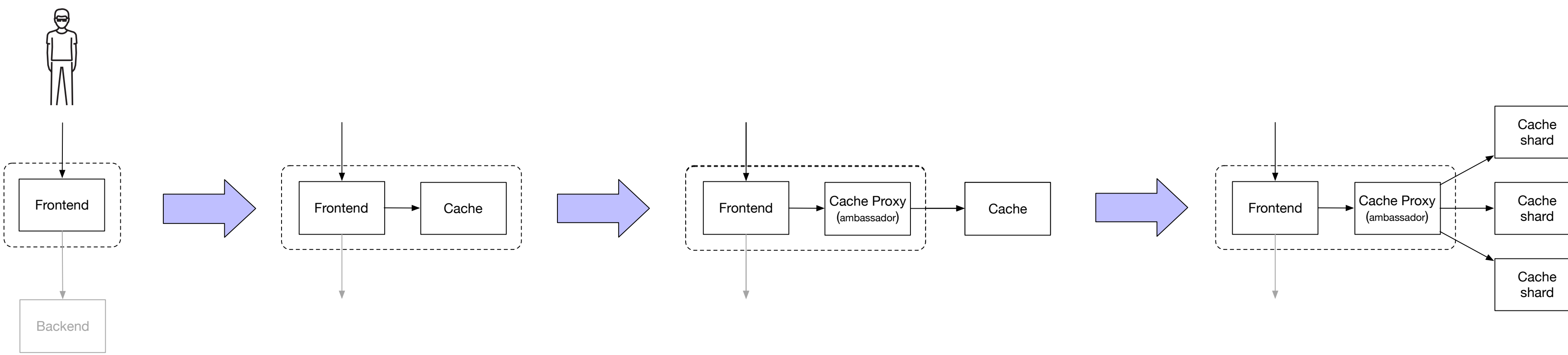
Overview

- **Previously “single-node” patterns**
- **Extend now to consider multiple nodes**
  - Multiple instances, multiple pods
  - Within the context of a single service with unified interface
  - … although ambiguous where boundary across services lies …

- **Covers a wide variety of scalable, fault-tolerant and resilient patterns**
  - Load balancing, replication, sharding, stateless and stateful service scaling, etc.

Single node vs. multi-node patterns

- **Single node patterns retain applicability**
  - Ambassadors and adapters abstracting away system-wide changes



Stateless and stateful services

- **Perhaps easiest to understand:**
  - Service is stateless if <u>you do not lose data</u> if service disappears
    - *(never ever — backups don't help, they have latency and can fail too)*
  - Contrary, service is stateful if losing data <u>causes some form of loss</u> (performance, consistency, monetary, data …)
- **Really applies only to persistent state: systems have lots of <u>transient state</u>**
  - Results from downstream microservice before passed upwards

State in distributed systems

- **Even discounting all failures, state has its problems**

- **Often infeasible to replicate data to all nodes**
  - Partitioning, sharding, partial replication— lead to data locality problems
- **Synchronization when changed**
  - Ensuring persistence
  - Access to stale data

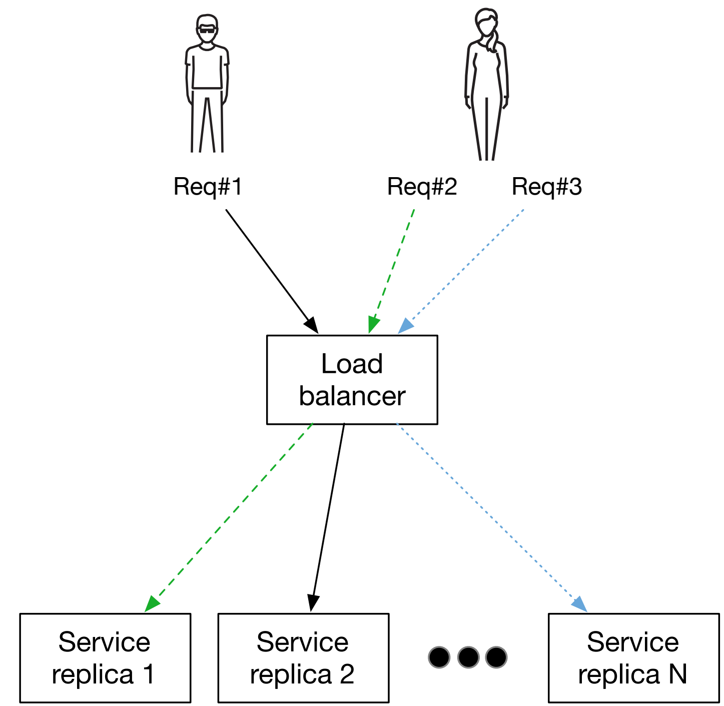
Sessions

- **“Session” is state tied to specific requestor**
  - Any form of identifier token passed from client works
  - Often stored as signed tokens in HTTP cookies (but not always)
  - We are interested in <u>server-side sessions</u>

- **Should differentiate between “transient session state”**
  - Loss of transient state not problematic (e.g. cached data)
  - We are interested in <u>persistent session state</u> (cart, UI state)
    - *Remember — loss of cart data can be linked to monetary losses*
    - *Differing priorities — user’s geolocation $\prec$ shopping cart $\prec$ purchase history (think about data persistence SLA!!!)*

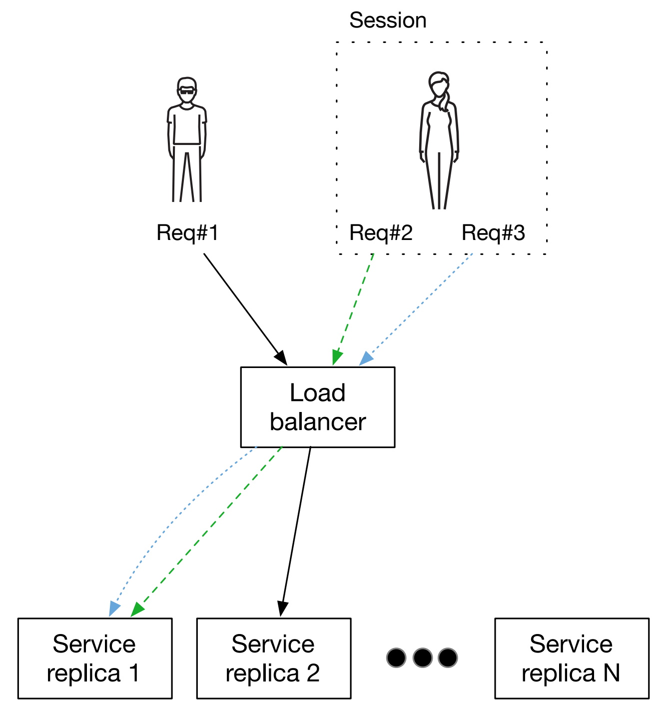
Load-balanced services

- **Multiple identical stateless services**
  - Send requests according to some policy (RR, random, LRU, …)
  - Service is <u>replicated</u>, functionally identical portions duplicated



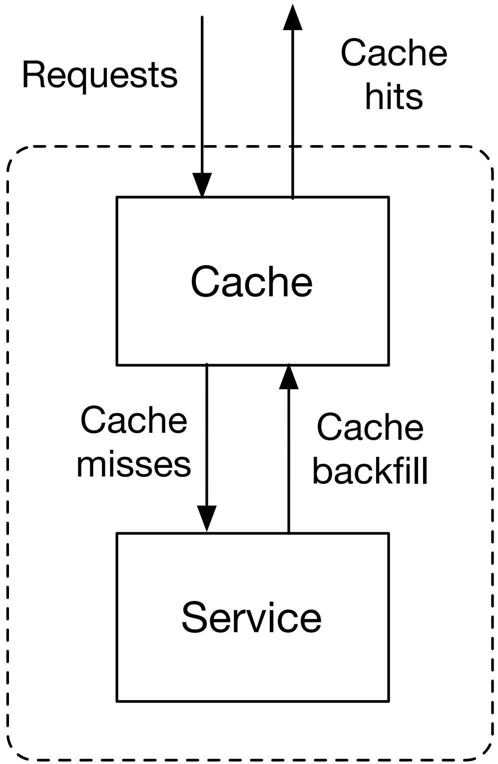
Load-balanced services

- **Multiple identical <u>stateful</u> services**
  - Identify a session key
  - Send request to backend identified by the session key
  - If not identified, use some policy (like before)
- **Problems**
  - Hot replica
  - Key redistribution



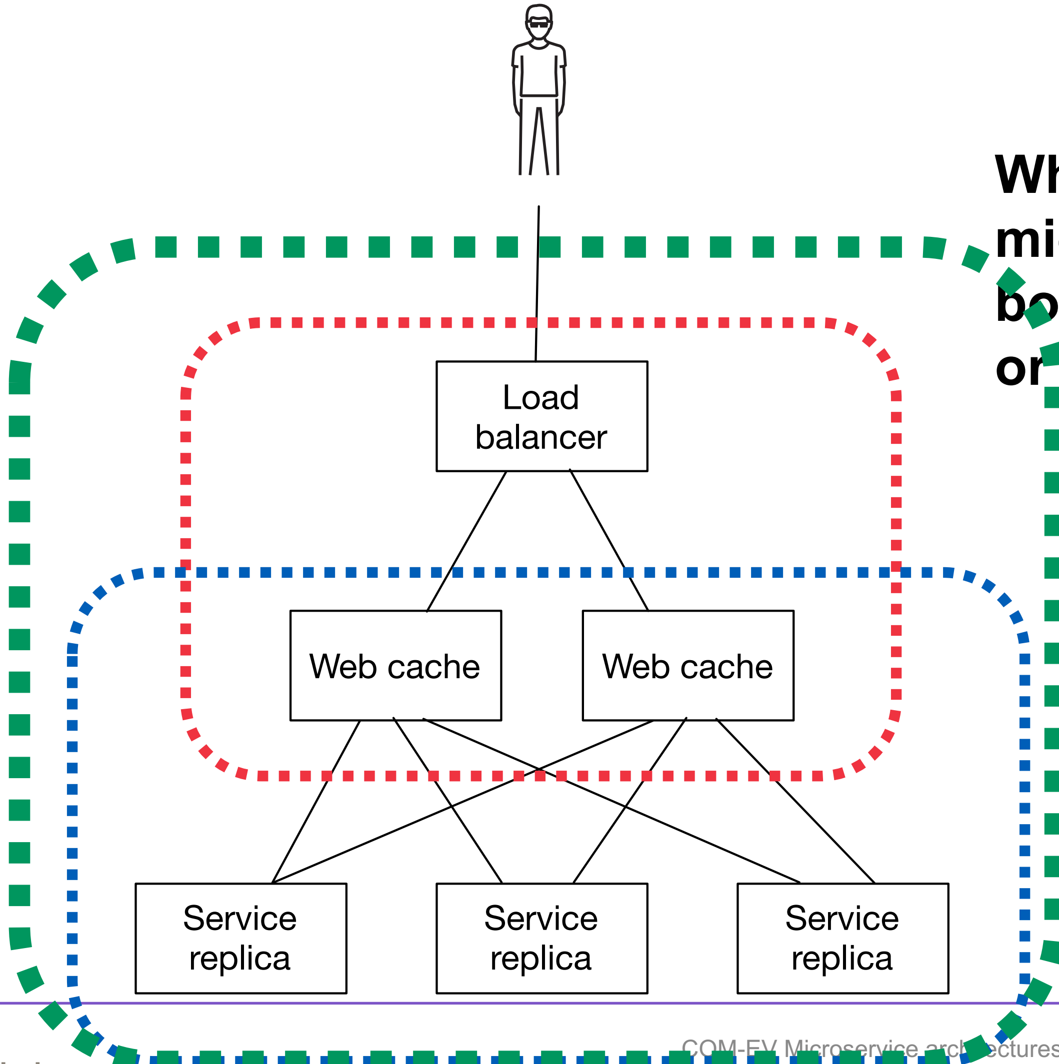
Caching

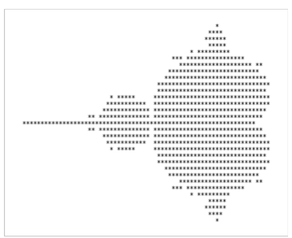
- **Stateless data may still be expensive**
- **Borrow from dynamic programming: don't recompute**
- **Algorithm: 1. If already computed, skip to 4. 2. Compute result. 3. Store result in cache. 4. Return result.**
  - Lots of details omitted: how to identify a specific computation; how long to hold to a result; how to avoid storing data indefinitely; what to do if space runs out; security considerations; …



Which is the microservice boundary? Red or blue?

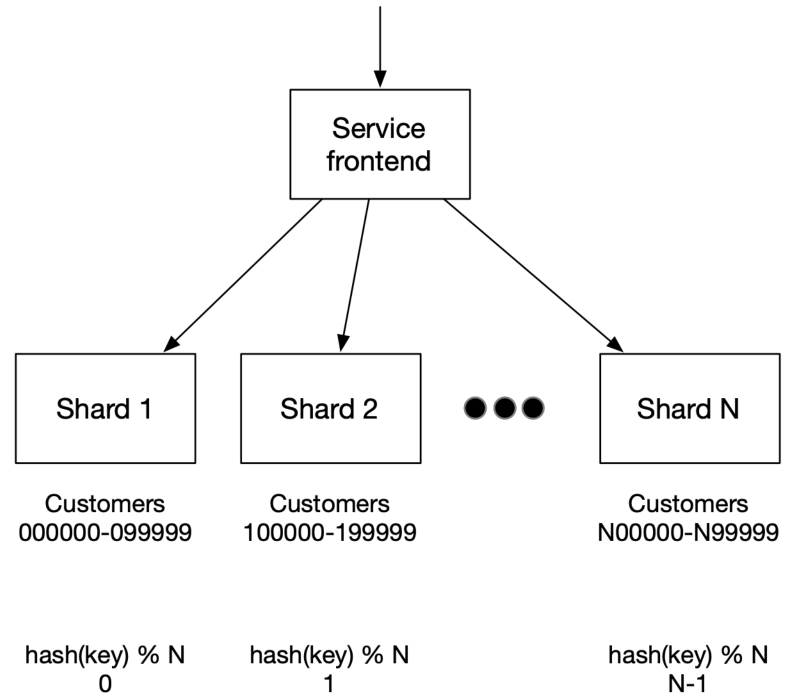
Why?





Sharding

- **Distribute requests to <u>specific</u> backend**
  - Use <u>sharding function</u> mapping a sharding key to shard index
  - Non-sequential keys hashed
  - Consistent sharding functions (why modulo is not?)

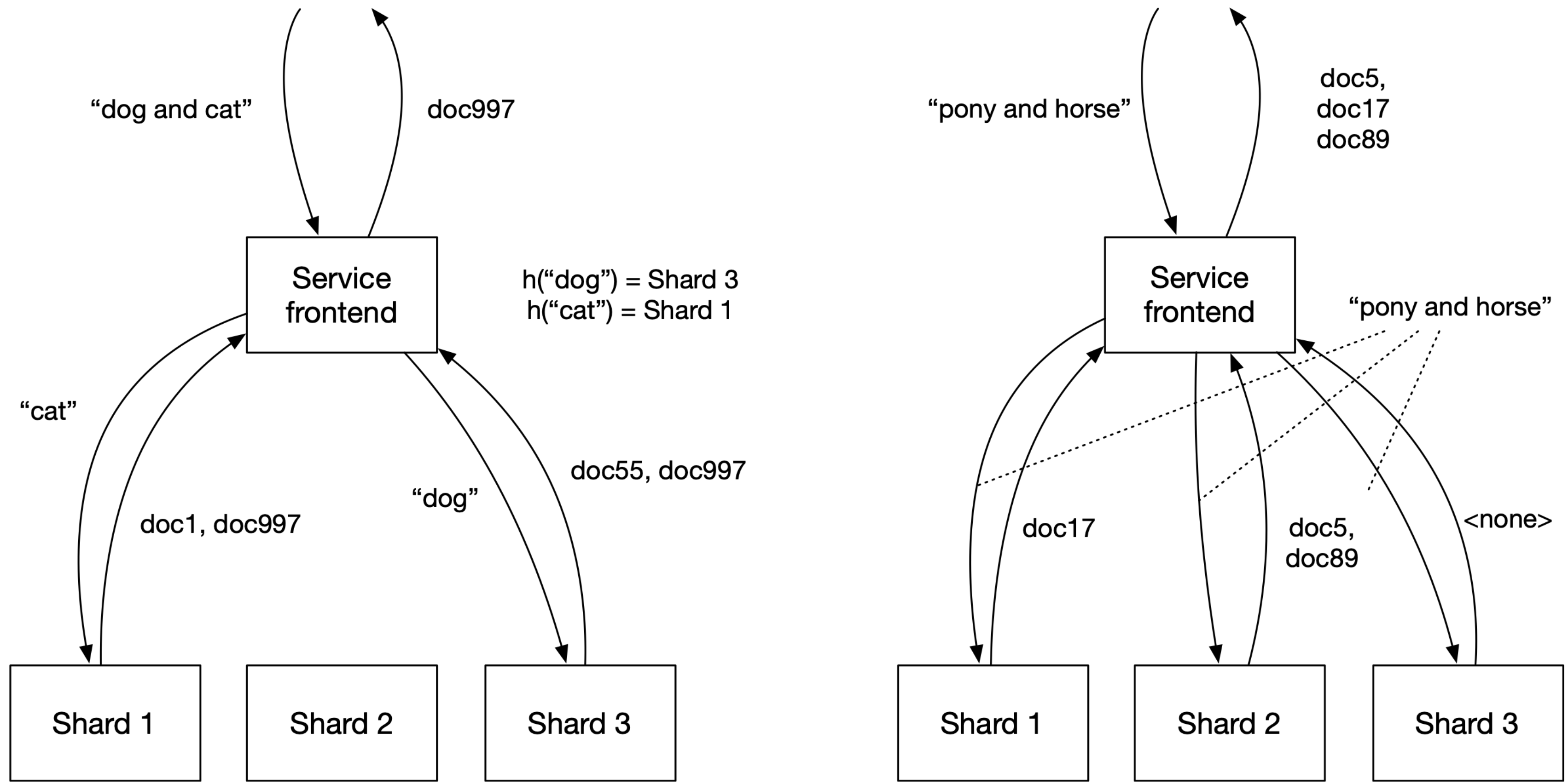


Sharded services

- **A category of <u>service brokering</u>**
- **Usually used for sharding of data**
  - 100 servers x 100 TB = 10 PB
  - Contrast with replication
- **Problems**
  - Hot keys and hot shards
  - Keyspace changes (need for consistent sharding function)
  - Persistence and reliability (shard replicas or replicate shards? → development leads to systems such as Cassandra)

Scatter-Gather

- Specific type of sharding & replication
- Distributed searching one of first large-scale applications (at Google)



"dog and cat"
doc997
Service frontend
h("dog") = Shard 3
h("cat") = Shard 1
"cat"
doc1, doc997
"dog"
doc55, doc997
Shard 1
Shard 2
Shard 3
"pony and horse"
doc5,
doc17
doc89
Service frontend
"pony and horse"
doc17
doc5,
doc89
<none>
Shard 1
Shard 2
Shard 3

Scatter-Gather

- **Specific type of sharding & replication**
- **Distributed searching one of first large-scale applications (at Google)**

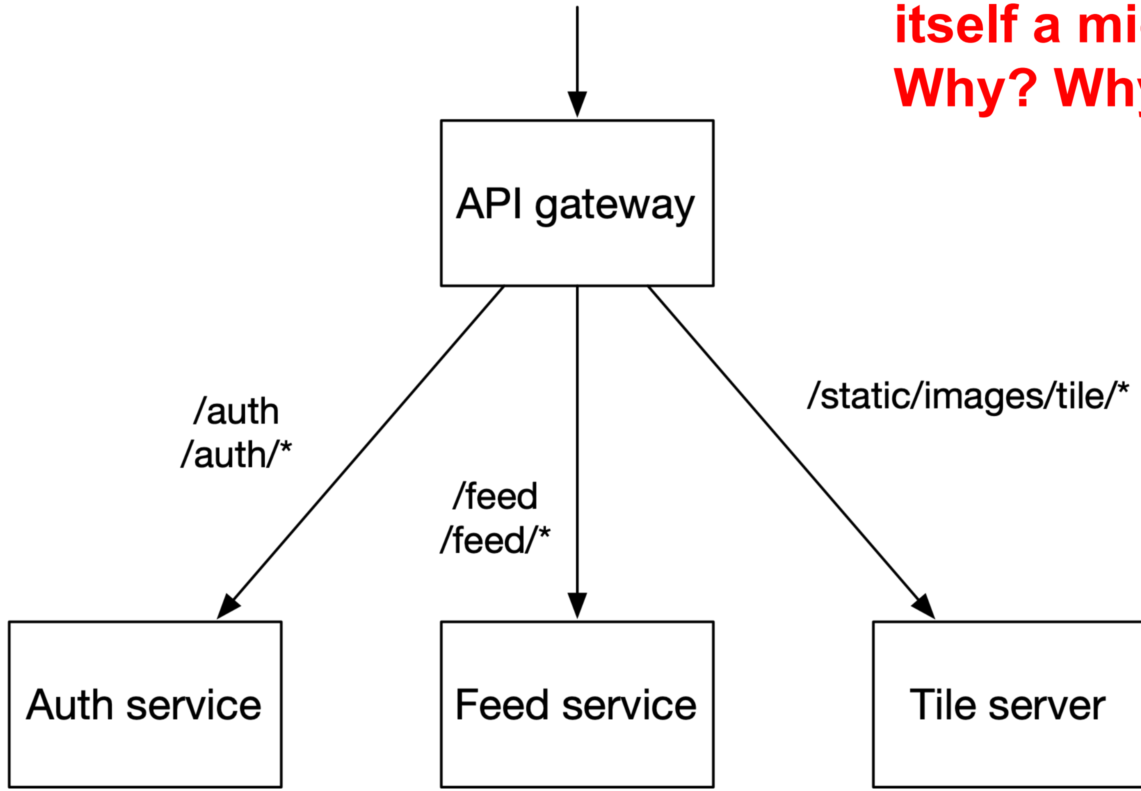
- **Contrasted to replication and sharding, the request is split into <u>multiple</u> sub-requests (scatter) whose results must be processed into final result (gather)**
- **Suitable for "embarrassingly parallel" problems**

Service broker

- **General category for solutions where**
  - Requests are forwarded only to one target
  - Target is defined by request context (session, URL, …)
- **Sharding is one example**
- **Other examples**
  - Reverse proxy
  - API gateway

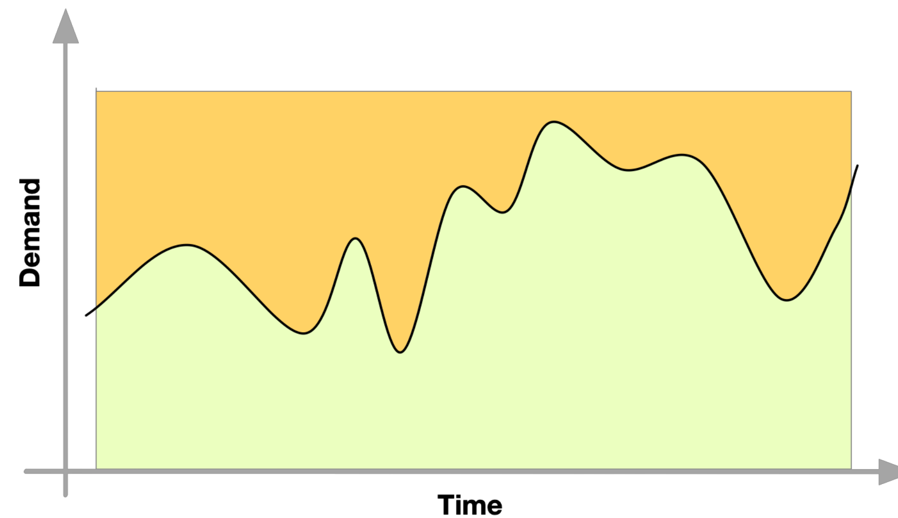
- **Why not UI frontends? Why not scatter-gather?**

Is the API gateway itself a microservice? Why? Why not?



Scaling

- **Statically resourced systems applicable if**
  - Load pattern is predictable and not highly variable
- **Conversely many real-world problems don't fit this**
  - Daily variation (night / day)
  - Weekly variation (weekday / weekend)
  - Spikes and dips (black Friday, Christmas)
  - Long-term patterns (increased popularity, viral effects)
- **➔ Unused capacity ➔ $£€ lost**



Scaling

- **Two different problems to solve: capacity and application**
- **Capacity problem:**
  - How to add (and remove) or alter physical capacity (cpu, disk)
  - Manual process with physical servers, hence cloud services and machine-friendly management APIs
- **Application problem:**
  - How the application adapts to capacity changes
  - Node additions / removals — on-line or off-line process?
  - Problem primarily for stateful services

Horizontal and vertical scaling

- **Vertical scaling (going up!): bigger box**
  - Increase instance size, increase disk allocation, …
- **Horizontal scaling (going sideways!): more boxes**
  - Add 1 box … add 1 box … add 1 box … repeat
- **Of course it is possible to use both simultaneously**

- **”Blast radius” describes area of impact of an failure**
  - “Larger instance” (vertical scaling) >> Lots of boxes
  - SPOF database’s blast radius is easily the whole system
    1-out-of-N stateful customer service affects 1/N customers

Scaling automation

- **A whole problem field in itself …**
- **Autoscaling solutions (AWS, GCP, Azure)**
  - Upscale and downscale triggers (CPU, network, request rate, ...)
  - Scaling actions (instance selection, termination policy etc.)

- **Never ever use autoscaling in production without monitoring and alerting!**

Moarrr patterrrrns!

- Microsoft Azure: Design Patterns
- Microservices.io: A pattern language for microservices

- Of course GoF, C2 Wiki etc. for more on patterns in general

Summary

- **Single-node patterns useful for abstracting and extending applications**
  - Without application code changes
  - Exchange tight coupling at code level to tight coupling on interfaces
- **Multi-node patterns are the toolbox for scalable and distributed services**
  - We'll come back to more but those in this lecture are the most common building blocks

- **Remember: DRY & NIH**