Engineering Cloud Applications 3

Scalability, Elasticity, Availability, and Resilience

Dr. Philipp Leitner

University of Zurich, Switzerland

Universität Zürich[UZH]

s.e.a.l.
software evolution & architecture lab

Organisatorial

- **Date for Oral Exam:**

  June, 16th, 2017 (Office Prof. Gall, 2.D.02)

- **Please use Doodle poll to register (first come, first served)**

  http://doodle.com/poll/4s469mdwpqm5rmgt

Outline

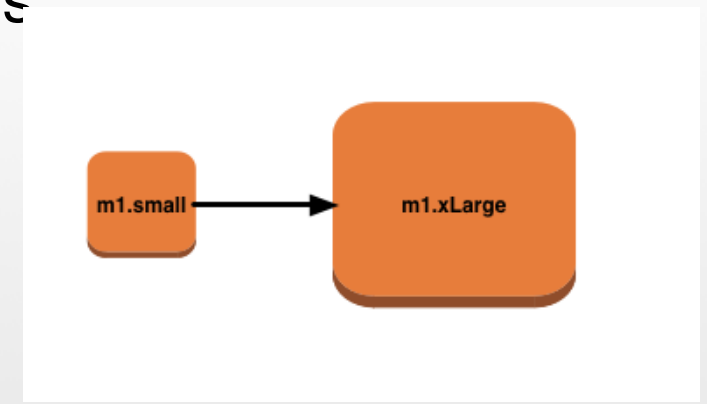
- Basic Concepts of Scalability and Availability
- Performance and Performance Variation in Clouds
- Microservices as a Common Way to Build Cloud Apps


Foundations of
Scalability and Elasticity

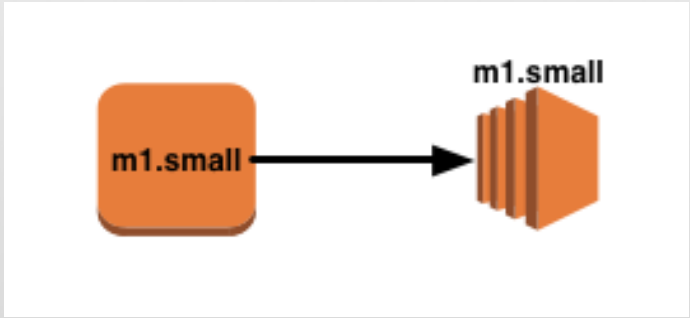
Increasing Resources

In order to increase the amount of resources available for an application, we may do 2 different things:

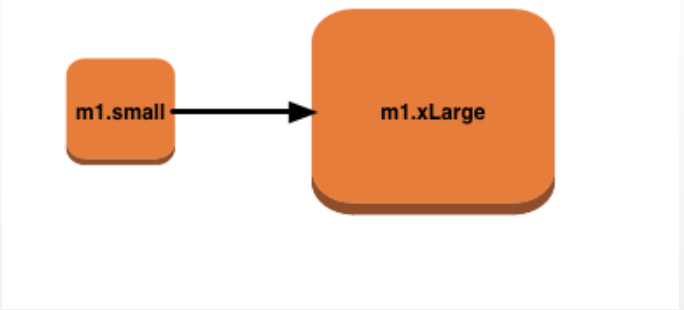
- **Variant 1:** Scale up (or: scale vertically)



- **Variant 2:** Scale out (or: scale horizontally)



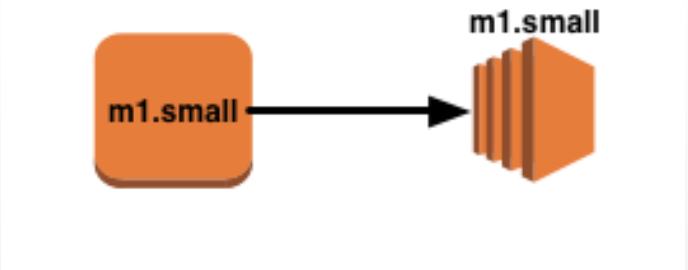
Scaling Up and Out (1)



Scaling up:

- Inverse: scaling down
- Advantage: easy to implement, works for many applications
- Disadvantages: often cannot be done without interuption, there are natural limits **how far** one can scale up (there is no infinitely fast machine)

Scaling Up and Out (2)

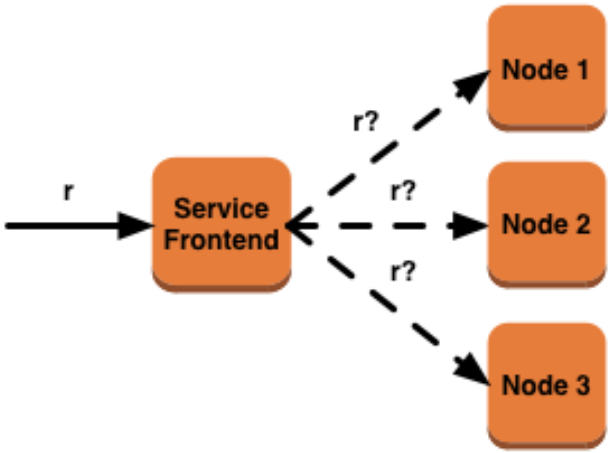


Scaling out:

- Inverse: scaling in
- Advantage: technical limits **much higher**, can be done without interruption (in well-built systems)
- Disadvantages: requires some load balancing, speedup not the same for each application

Load Balancing

- In a scaled-out system, whenever a new request comes in, it needs to be scheduled (mapped) to one of the nodes
- This request scheduling algorithm should optimally balance the load between nodes so that the average processing time for requests is minimal



Request Scheduling Algorithms

- **Generic:**
  - Round Robin
  - Random
- **With heterogeneous nodes:**
  - Weighted Random
- **Based on load monitoring:**
  - Least loaded
  - “The supermarket model”
- **Predictive**
  - Bin packing

Scaling Out

For which systems does it make sense to scale out?

For systems that are scalable

Scalability

- Scalability is a measure for how much *"better" a system becomes if it gets more of a given resource*
- **Speedup**:
  - How much faster does a system become if we add more CPU time?
  - E.g, if I go from one CPU to two (identical) CPUs, and my system now takes 80% of the time to finish, the speedup is 0,4

$$Speedup = \frac{N_{orig}\ t_{new}}{t_{orig}\ N_{new}}$$

$$e.g.$$

$$Speedup = \frac{1 * 80}{100 * 2} = 0.4$$

Amdahl's Law (1)

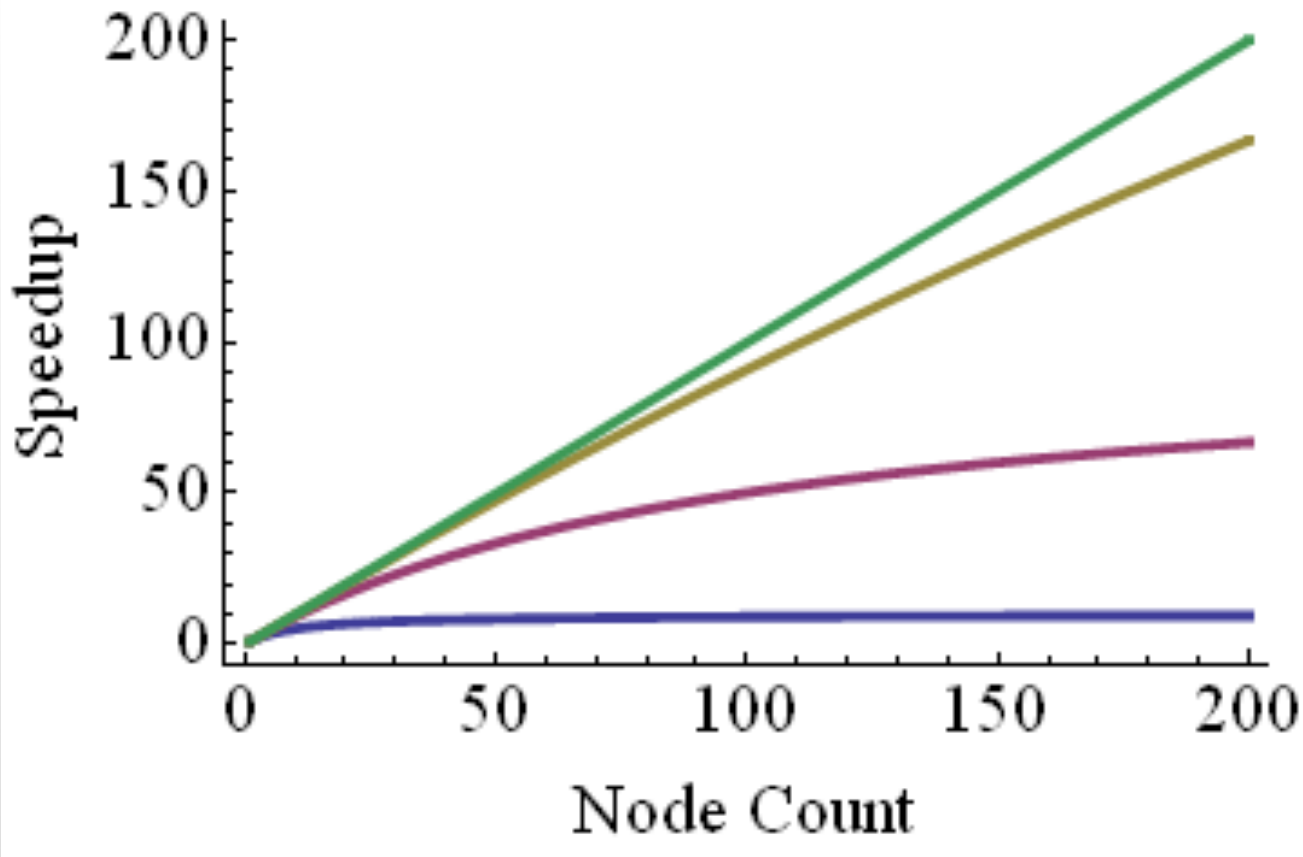
- Many systems have a speedup < 1
  - (the system does not scale perfectly, there are "diminishing returns" on adding more CPUs)
- Amdahl's Law:

$$t_{total} = t_{serial} + \frac{t_{parallel}}{N}$$

$$Speedup = \frac{1}{(1 + (N-1)a)},\ with\ a = \frac{t_{serial}}{(t_{serial} + t_{parallel})}$$

  - For a > 0, the Speedup converges against 0

Amdahl's Law (2)



Source: https://www.cac.cornell.edu/VW/Optimization/doitfaster.aspx

Special Cases

- **Special case 1:** a = 0
  - The system is fully scalable, there is no serial part
  - "**Embarassingly parallel**" application

$$\forall N \in \mathbb{N} : Speedup(N) = 1$$

- **Special case 2:** a = 1
  - The system is not parallelizable at all
  - E.g., a single-threaded application

$$\forall N \in \mathbb{N} : Speedup(N) = 0$$

Embarassingly Parallel Problems (1)

- In practice, a large set of problems turn out to be embarassingly parallel

  - Handling Web requests (if stateless)
  - Many rendering problems
  - Rainbow table attacks on hashes
  - Optimization via genetic algorithms
  - …

Embarassingly Parallel Problems (2)

- Characteristic of all embarassingly parallel problems:
  - Individual small work items (handling a web request, rendering a frame, etc.) can be handled **entirely independently**
  - Any node that handles a work item needs to know nothing about any other work item
  - There are no cross-references or interactions between work items

— Statelessness —

Limitation of Amdahl's Law

- Amdahl's law gives us a good feeling for whether a system will be scalable
- However:
  - Even an embarrassingly parallel system may have some amount of parallelization overhead
  - **Time required for data transfer between nodes**
  - (not captured at all in Amdahl's law)

Extension of Amdahl's Law (1)

$$t_{total} = t_{processing} + t_{messaging} + t_{dataTransfer}$$

$$t_{processing} = \frac{t_{parallel}}{N} + t_{serial}$$

$$t_{messaging} = M * latency$$

$$t_{dataTransfer} = \frac{|M|}{bandwidth}$$

M … number of messages

IMI … total size of those messages

Source: https://www.cac.cornell.edu/VW/Optimization/estimate.aspx

Extension of Amdahl's Law (2)

- With the previous definitions, we can assume the following model for total time incl. message transfer

$$t_{total} =$$

$$\frac{t_{parallel}}{N} + t_{serial} +$$

$$k_0 * N^{\alpha} * latency +$$

$$k_1 * \frac{N^{\beta}}{bandwidth}$$

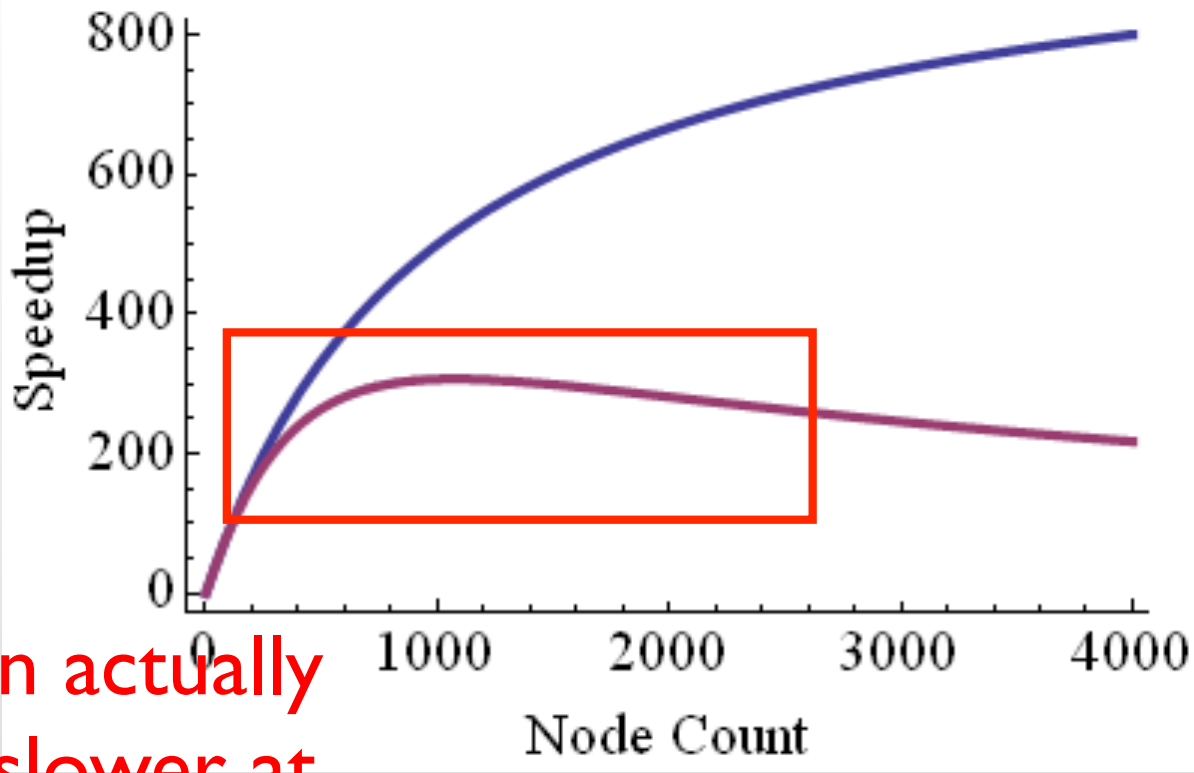
Amdahl's Law

Overhead of Message Init

Overhead of Message Transmission

Source: https://www.cac.cornell.edu/VW/Optimization/estimate.aspx

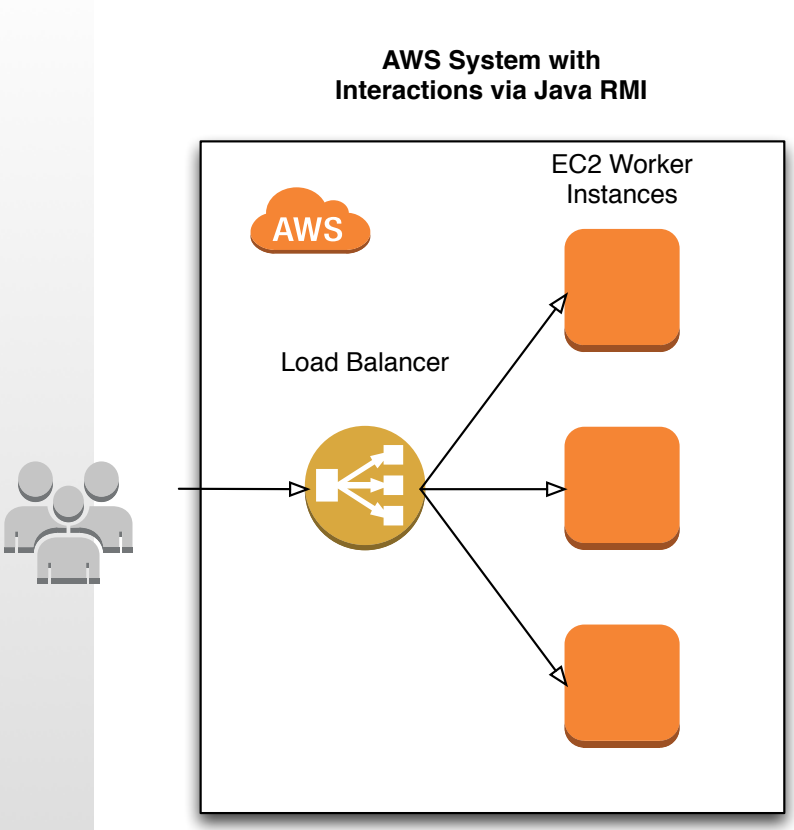
Shape of Speedup With Messaging

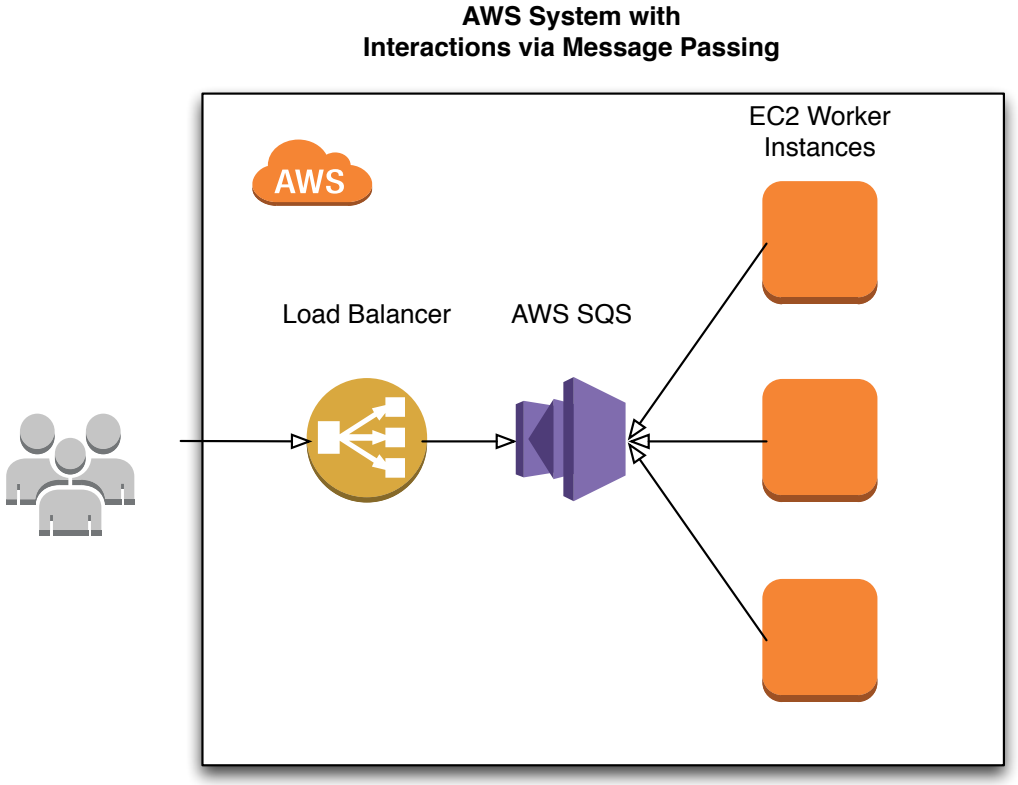


Application actually becomes slower at some point

Source: https://www.cac.cornell.edu/VW/Optimization/estimate.aspx

Message Queueing





Advantages of Message Queueing

- **When architecting a scalable system, (asynchronous) message queueing has many advantages:**
  - Loose coupling — it is easy to add more instances to the system, as other components (e.g., the load balancer) never talk directly to instances
  - Reduced number of open connections — the load balancer does not need to keep connections open to all other components
  - Fault tolerance — individual faulty instances do not break the system

Fault Tolerance (1)

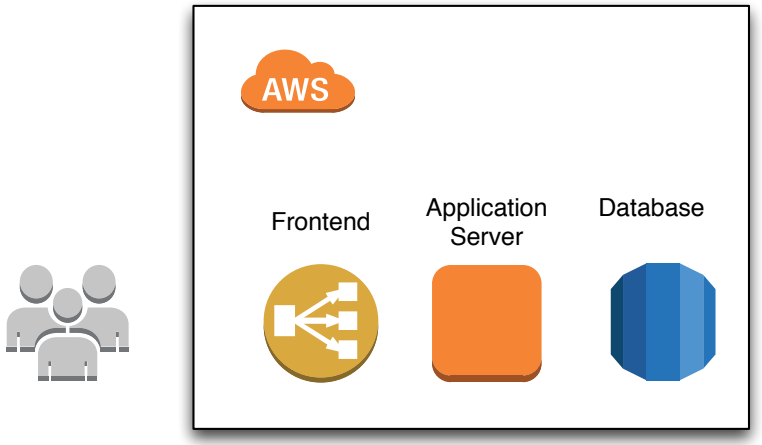
- In addition to improving performance, message passing and replication improves **availability**

- Availability:
  - The probability that a system is online at any point in time

$$av(S) = 1 - \frac{downtime}{uptime}$$

Fault Tolerance (2)

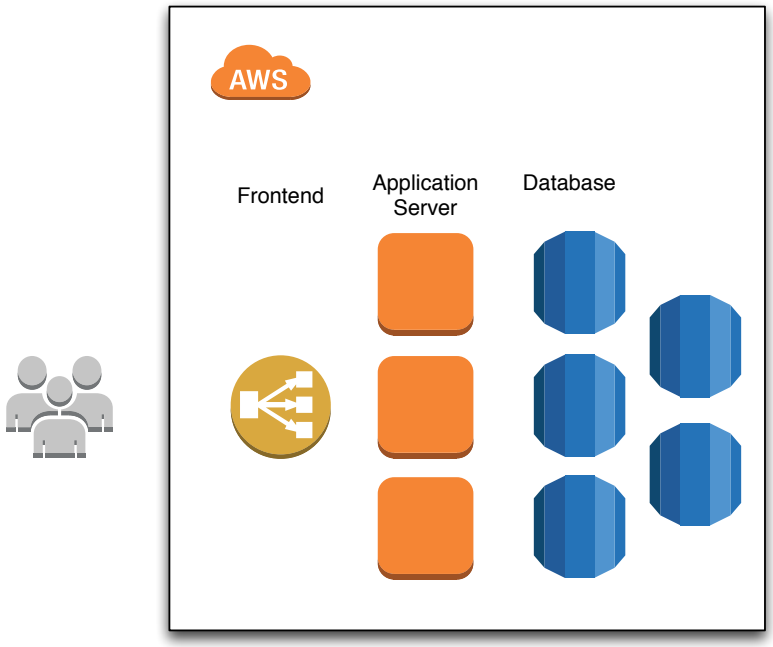
- Assume a system consisting of a frontend, an application server, and a database
- The availability may be:
  - av(frontend) = 0.99
  - av(appserver) = 0.99
  - av(db) = 0.99
  - av(system) = 0.99^3 = 0.97



The overall system availability can be interpreted as the probability that all necessary components are online at the same time ...

Fault Tolerance (3)

- Now assume we have 3 app servers (each with a bad availability of 0.9) and a cluster of 5 crappy database servers with an availability of 0.85
- The availability is now:
  - av(frontend) = 0.99
  - av(appserver) = 1 - 0.1^3 = 0.999
  - av(db) = 1 - 0.15^5 = 0.9999
  - av(system) = **0.9889**



(this is under the assumption that one database / app server alone is able to handle a request)

Testing for Fault Tolerance

- Big cloud customers tend to be fanatic about testing for fault tolerance

Remember that cloud providers may turn off instances at any time (volatility)

- E.g., Netflix has the **Chaos Monkey**

"Chaos Monkey is a service which runs in the Amazon Web Services (AWS) that seeks out Auto Scaling Groups (ASGs) and terminates instances (virtual machines) per group."

http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html

Regions and Availability Zones (1)

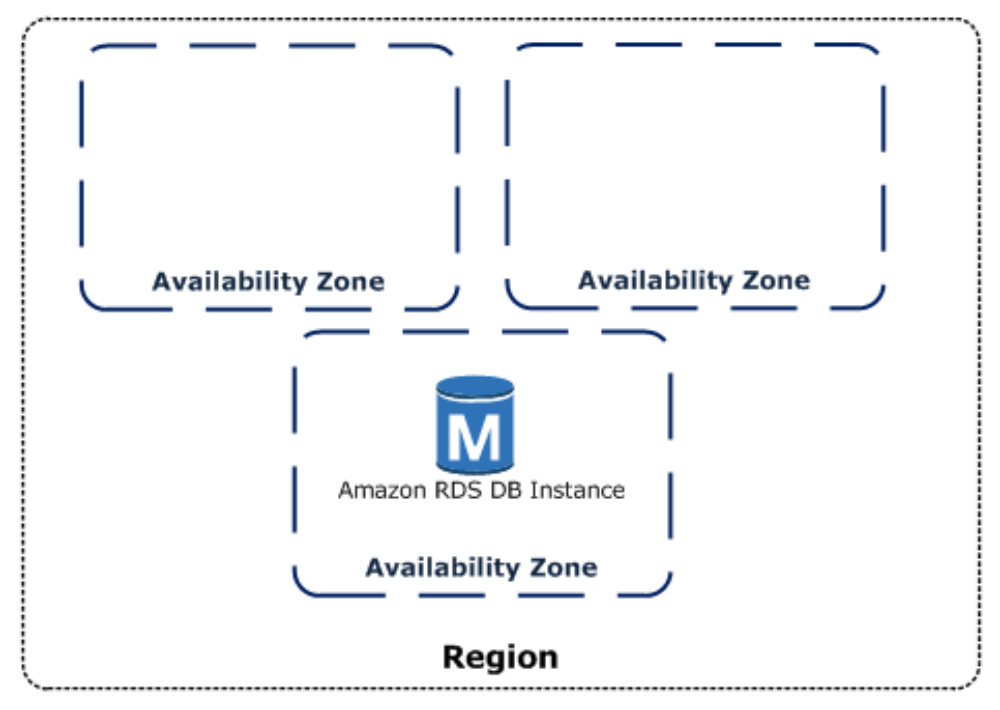
Most big public clouds have data centers all around the world (*regions*)

(as of March 30th, 2017)

Region	Name	Endpoint
US East (N. Virginia) Region	us-east-1	https://rds.us-east-1.amazonaws.com
US East (Ohio) Region	us-east-2	https://rds.us-east-2.amazonaws.com
US West (N. California) Region	us-west-1	https://rds.us-west-1.amazonaws.com
US West (Oregon) Region	us-west-2	https://rds.us-west-2.amazonaws.com
Asia Pacific (Mumbai) Region	ap-south-1	https://rds.ap-south-1.amazonaws.com
Asia Pacific (Seoul) Region	ap-northeast-2	https://rds.ap-northeast-2.amazonaws.com
Asia Pacific (Singapore) Region	ap-southeast-1	https://rds.ap-southeast-1.amazonaws.com
Asia Pacific (Sydney) Region	ap-southeast-2	https://rds.ap-southeast-2.amazonaws.com
Asia Pacific (Tokyo) Region	ap-northeast-1	https://rds.ap-northeast-1.amazonaws.com
Canada (Central) Region	ca-central-1	https://rds.ca-central-1.amazonaws.com
China (Beijing) Region	cn-north-1	https://rds.cn-north-1.amazonaws.com.cn
EU (Frankfurt) Region	eu-central-1	https://rds.eu-central-1.amazonaws.com
EU (Ireland) Region	eu-west-1	https://rds.eu-west-1.amazonaws.com
EU (London) Region	eu-west-2	https://rds.eu-west-2.amazonaws.com
South America (São Paulo) Region	sa-east-1	https://rds.sa-east-1.amazonaws.com
AWS GovCloud (US)	us-gov-west-1	https://rds.us-gov-west-1.amazonaws.com

Regions and Availability Zones (2)

In each region, there are multiple (e.g., 3) *availability zones*. Your instances typically exist in exactly one of those.



Redundant deployments for high availability should deploy *in different AZs*!

Two resources in the same AZ are much more likely to fail at the same time.

Rules of Thumb for Region and AZ Selection

- **Select the *region* based on:**
  - Where your customers are (latency)
  - Where you want to store your data (privacy)
  - If none of those matters, choose the cheapest
- **As for selecting the *AZ*:**
  - If you want to reduce latency, make sure that all resources are in the same AZ
  - If you want to increase availability (e.g., have backups), make sure that they are in *different* AZs

Unpredictability in Clouds

Coarse-Grained QoS Guarantees in Clouds

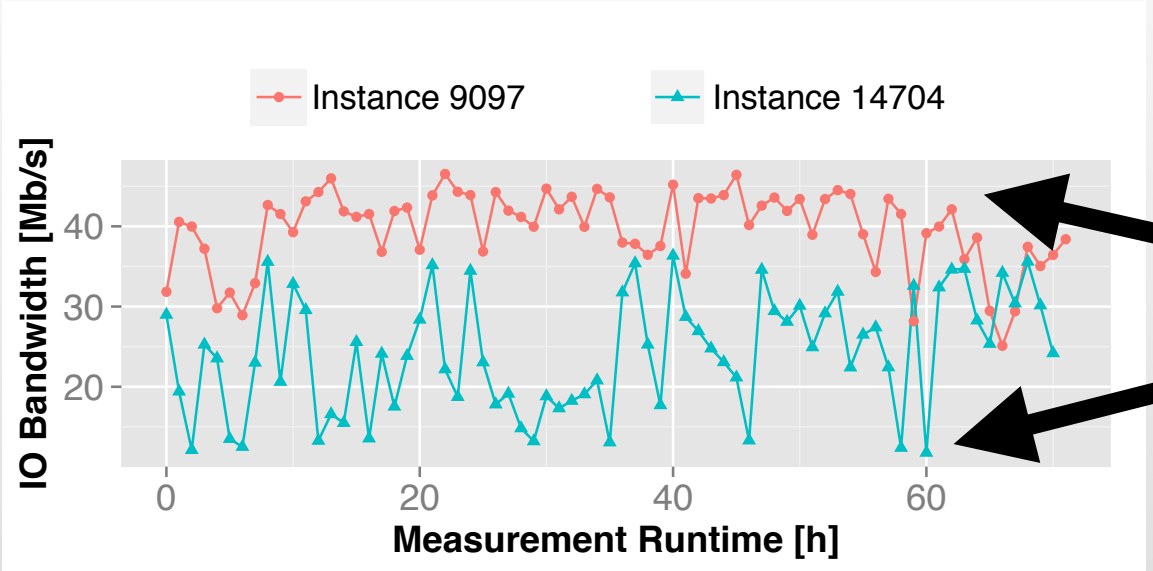
amazon web services™ | EC2

Instance Types Matrix

Instance Type	vCPU	Memory (GiB)	Storage (GB)	Networking Performance	Physical Processor	Clock Speed (GHz)	Intel® AES-NI	Intel® AVX†	Intel® Turbo	EBS OPT	Enhanced Networking
t2.micro	1	1	EBS Only	Low to Moderate	Intel Xeon family	2.5	Yes	Yes	Yes	-	-
t2.small	1	2	EBS Only	Low to Moderate	Intel Xeon family	2.5	Yes	Yes	Yes	-	-
t2.medium	2	4	EBS Only	Low to Moderate	Intel Xeon family	2.5	Yes	Yes	Yes	-	-
m3.medium	1	3.75	1 x 4 SSD	Moderate	Intel Xeon E5-2670 v2*	2.5	Yes	Yes	Yes	-	-
m3.large	2	7.5	1 x 32 SSD	Moderate	Intel Xeon E5-2670 v2*	2.5	Yes	Yes	Yes	-	-
m3.xlarge	4	15	2 x 40 SSD	High	Intel Xeon E5-2670 v2*	2.5	Yes	Yes	Yes	Yes	-
m3.2xlarge	8	30	2 x 80 SSD	High	Intel Xeon E5-2670 v2*	2.5	Yes	Yes	Yes	Yes	-

Performance Unpredictability (1)

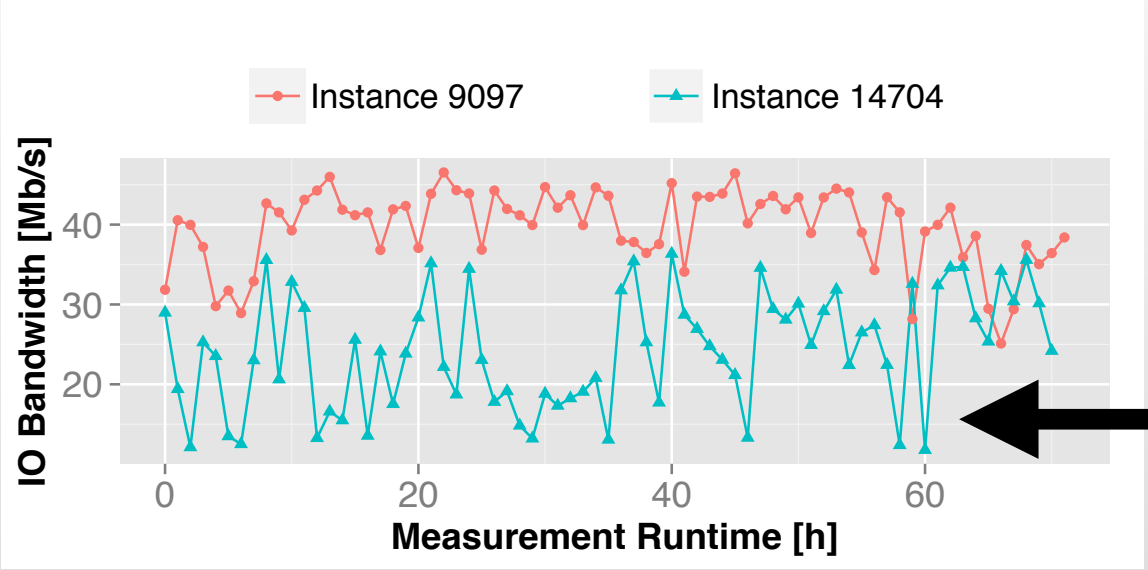
amazon web services™ | EC2



Two identical instances
-
very different performance

Performance Unpredictability (2)

amazon web services™ | EC2



Same instance over time
-
also very different performance

Two Kinds of Predictability

- **Inter-Instance Predictability**

  "How similar is the performance of **multiple identical instances** *?"*

- **Intra-Instance Predictability**

  "How **self-similar** *is the performance of a single instance over time?"*

Inter-Instance Predictability

Relative Standard Deviations

			CPU-Bound			IO-Bound	
		Type	CPU	MEM	Java	IO	OLTP
EC2	eu	t1.micro	12.14	17.67	30.63	71.33	30.66
		m1.small	3.19	3.77	3.17	88.49	13.02
		m3.large	0.13	2.07	7.22	35.53	21.26
		c3.large	0.21	8.60	6.42	58.88	21.31
		i2.xlarge	0.12	11.92	8.44	20.07	12.28
	na	t1.micro	20.28	26.40	59.32	70.08	32.18
		m1.small	12.81	26.18	5.34	94.47	15.68
		m3.large	0.16	4.46	9.23	49.02	37.10
GCE	eu	f1-micro	5.28		8.36	3.06	
		n1-standard-1	2.54		6.99	3.36	
		n1-standard-2	1.71		6.96	1.33	
	na	f1-micro	5.13		7.17	9.47	
		n1-standard-1	2.05		8.31	10.39	
		n1-standard-2	1.16		9.53	4.88	
Azure	eu	ExtraSmall	18.38		16.88	61.92	
		Small	18.23		8.37	59.01	
		Medium	17.81		11.91	47.14	
	na	ExtraSmall	18.13		15.96	49.01	
		Small	19.11		6.62	44.01	
		Medium	18.28		10.96	48.31	
SL	na	1 CPU / 2048 MB	0.11		6.65	13.01	
		2 CPUs / 4096 MB	0.11		7.14	6.27	

Intra-Instance Predictability

Relative Standard Deviations of Benchmark Runs Within the Same Instance

		Type	CPU	MEM	IO
EC2	eu	t1.micro	40.17	40.86	22.78
		m1.small	1.69	1.66	27.28
		m3.large	0.89	0.49	17.92
GCE	eu	f1-micro	3.24	6.85	2.65
		n1-standard-1	0.76	2.43	4.42
		n1-standard-2	0.86	2.01	1.46
Azure	eu	ExtraSmall	2.41	3.39	27.08
		Small	1.65	2.33	93.47
		Medium	1.30	1.98	0.14
SL	na	1 CPU / 2048 MB	0.22	0.39	2.94
		2 CPU / 4096 MB	0.13	0.14	2.25

Reasons

- **For IO-Bound Workloads**

  Multi-tenacy (noisy neighbours)

- **For CPU-Bound Workloads**

  Hardware Heterogeneity

Hardware Heterogeneity

CPU Models

(for m1.small and Azure Small in North America)

	Model	#
EC2	Intel E5-2650	1962
	Intel E5430	364
	Intel E5645	293
	Intel E5507	84
	Intel E5-2651	32
	AMD 2218 HE	9
Azure	AMD 4171 HE	782
	Intel E5-2673	595
	Intel E5-2660	189

Basic Alleviation Strategies

- **Pay**

  e.g., for baremetal servers or dedicated instances

  Larger instance types also tend to be more predictable

- **“Play the numbers game”**

  If you scale out, individual instance performance matters less

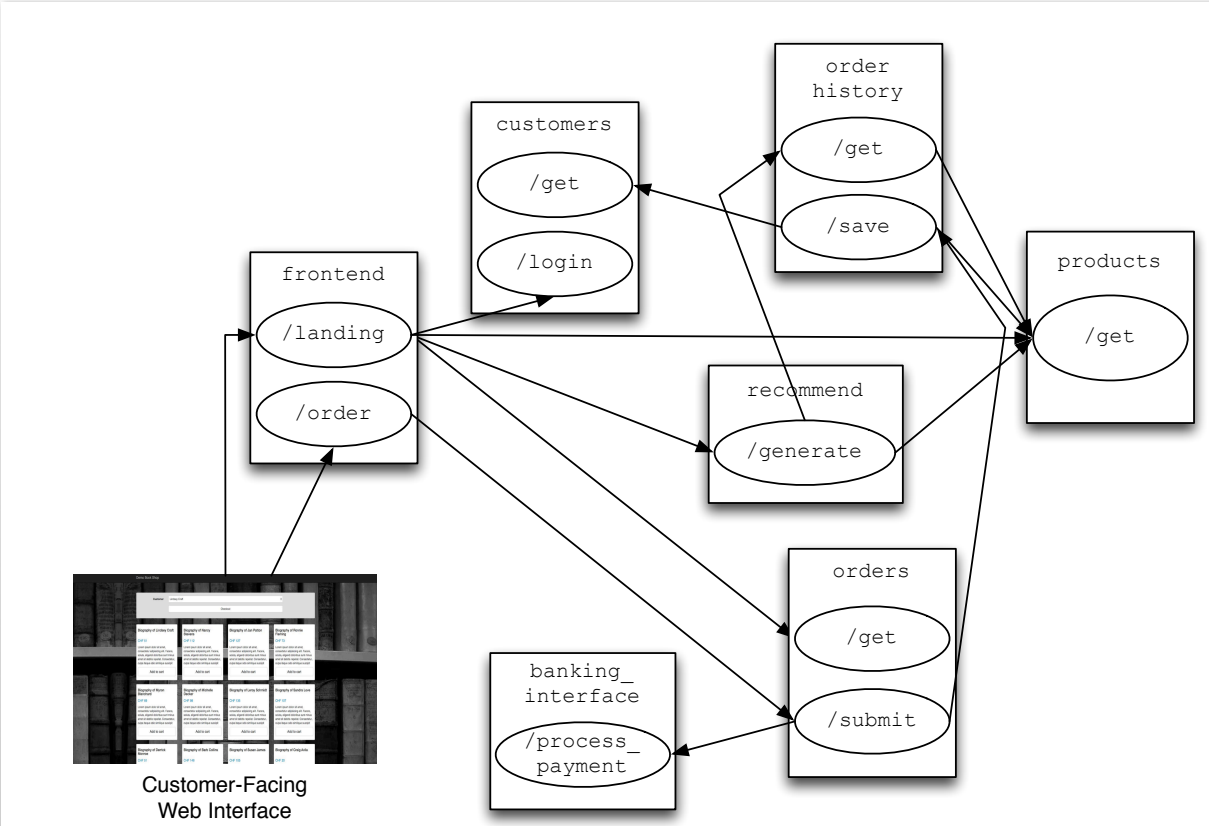
- **Benchmark-and-Discard**

  Start more instances than you need, benchmark them, and keep only the ones that were the fastest for you


Microservices

Reminder: Service-Based System

Applications are built as composition of services (lego principle)



What Are Microservices?

- **Subset / implementation strategy for service-based systems**
- **Sometimes called an architectural style, but has more to do with team organization than software**
- **Technology-agnostic**
- **Does *not* necessarily mean “very small services” :)**

Basic Ideas (1)

- **Smart endpoints and dumb pipes**
  - See also: pipes-and-filters architecture
- **Conway's Law**
  - Software architecture follows organizational structure
- **"Two-Pizza Rule"**
  - Services can be built and operated by a team that can be fed by two large pizzas

Basic Ideas (2)

- **“You build it, you run it”**
  - Embrace the DevOps concept
  - Every service is run as a little product of it’s own
  - “Startup mode”
- **Developer-on-call**
  - Developers are on call to fix issues in production
  - There is no “hand-over” to a prod team

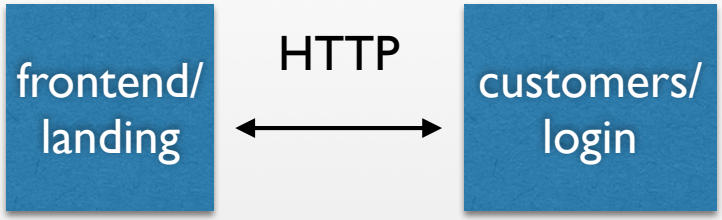
Conway's Law

Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.

http://www.melconway.com/Home/Conways_Law.html

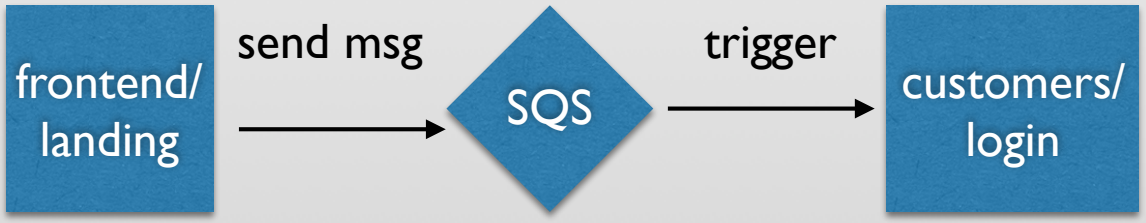
Integration Strategies

RPC services



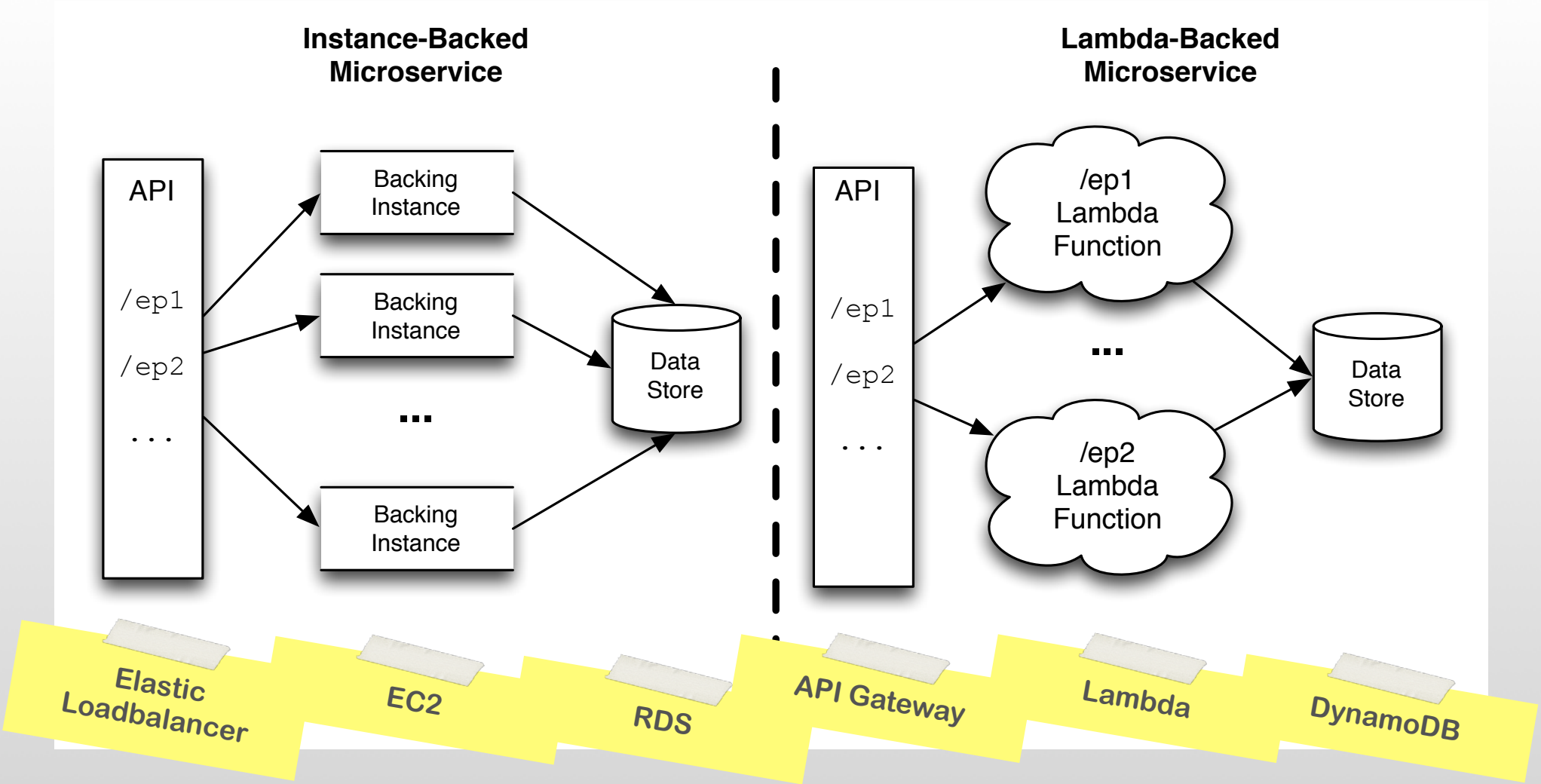
Often implemented via REST

Event-driven services



E.g., via AWS Lambda

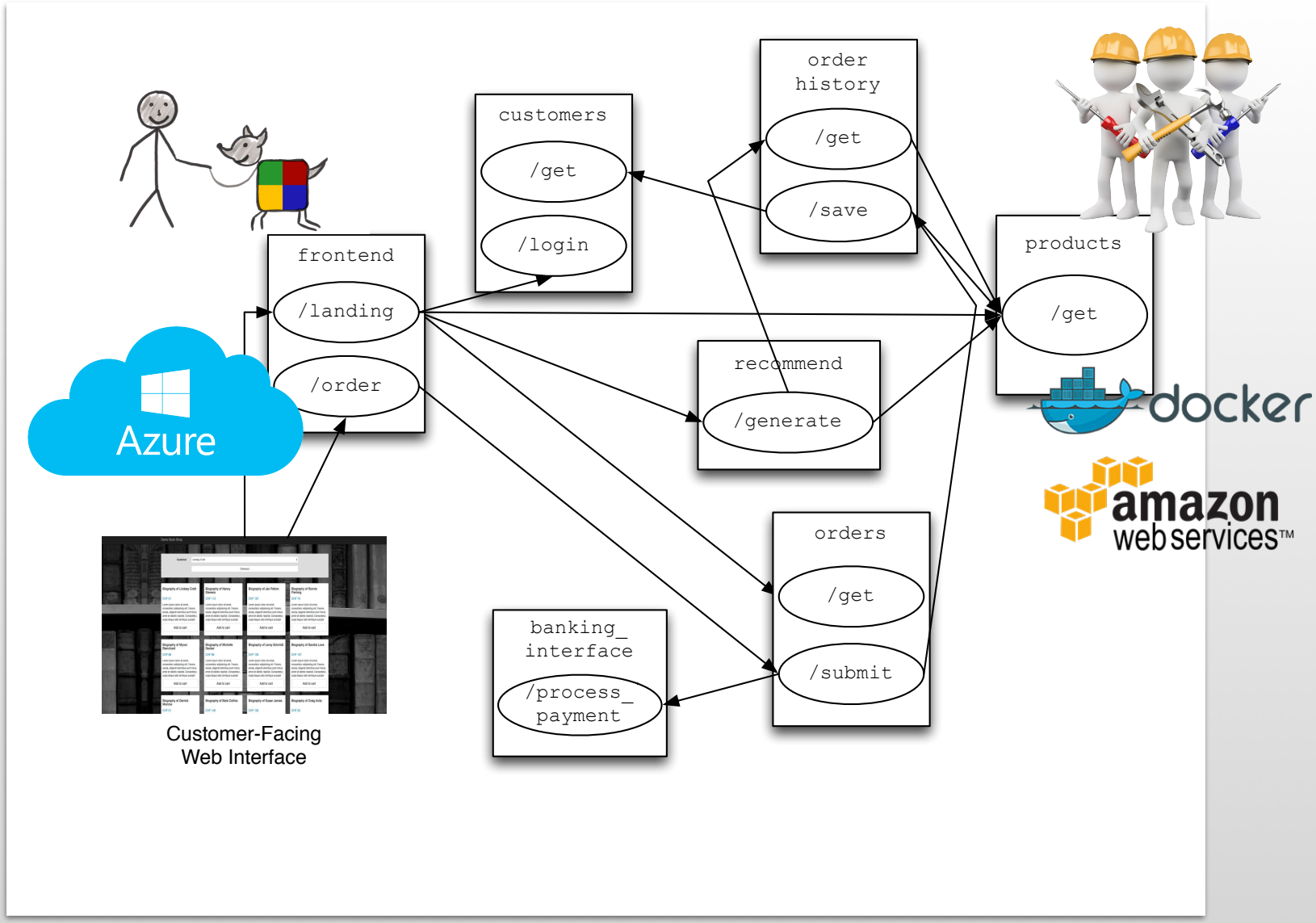
Basic Conceptual Service Model



Technological Advantages (1)

- **Technology-neutral**
  - Every team / service chooses the stack that makes sense for them
  - “Polyglott persistence”
  - Disadvantage:
    - Maintenance may be difficult
    - Knowledge transfer often hard

Technological Advantages (2)

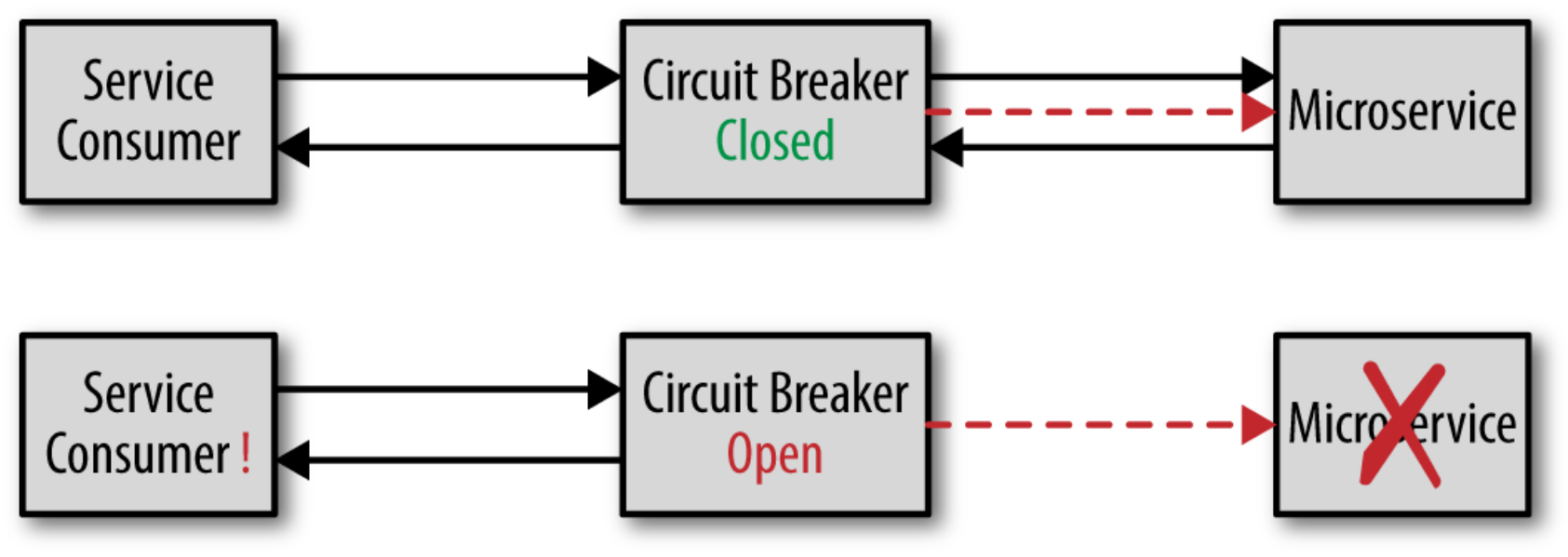


Technological Advantages (3)

- **Fine-Grained Elasticity**
  - Every service can scale out on it's own.
  - Allows to avoid overspending
  - Disadvantage:
    - Lack of global planning
    - "Friendly DoS"

Technological Advantages (4)

- **Resilience and Circuit Breakers**
  - Assume that any microservice may be slow or even fail at any time, and build your software accordingly



Summary

- **Scalability and elasticity is (also) a question of architecture**
  - Avoid state and keep your app embarrassingly parallel
  - Redundancy helps with availability (but keep in mind that more components —> more data transfer overhead)
- **Performance in cloud is *unreliable***
  - Inter-instance vs. intra-instance predictability
- **Microservices are an *architectural* and *organizational* approach to build *cloud-native* applications**