17-423/723: Software System Design

Design for Scalability

Mar 11, 2026

Learning Goals

- Describe scalability as a QA of a software system and its relationship with other QAs, such as performance, availability, and reliability.
- Specify a scalability QA in terms of load and performance metrics.
- Identify dimensions of workload patterns in the system being designed.
- Describe the differences between vertical and horizontal scaling.
- Describe the benefits and downsides of replication and partitioning approaches to distributed data.
- Describe different strategies for load balancing to avoid overloading parts of the system.
- Identify a bottleneck in the workload and apply caching to improve the system performance.

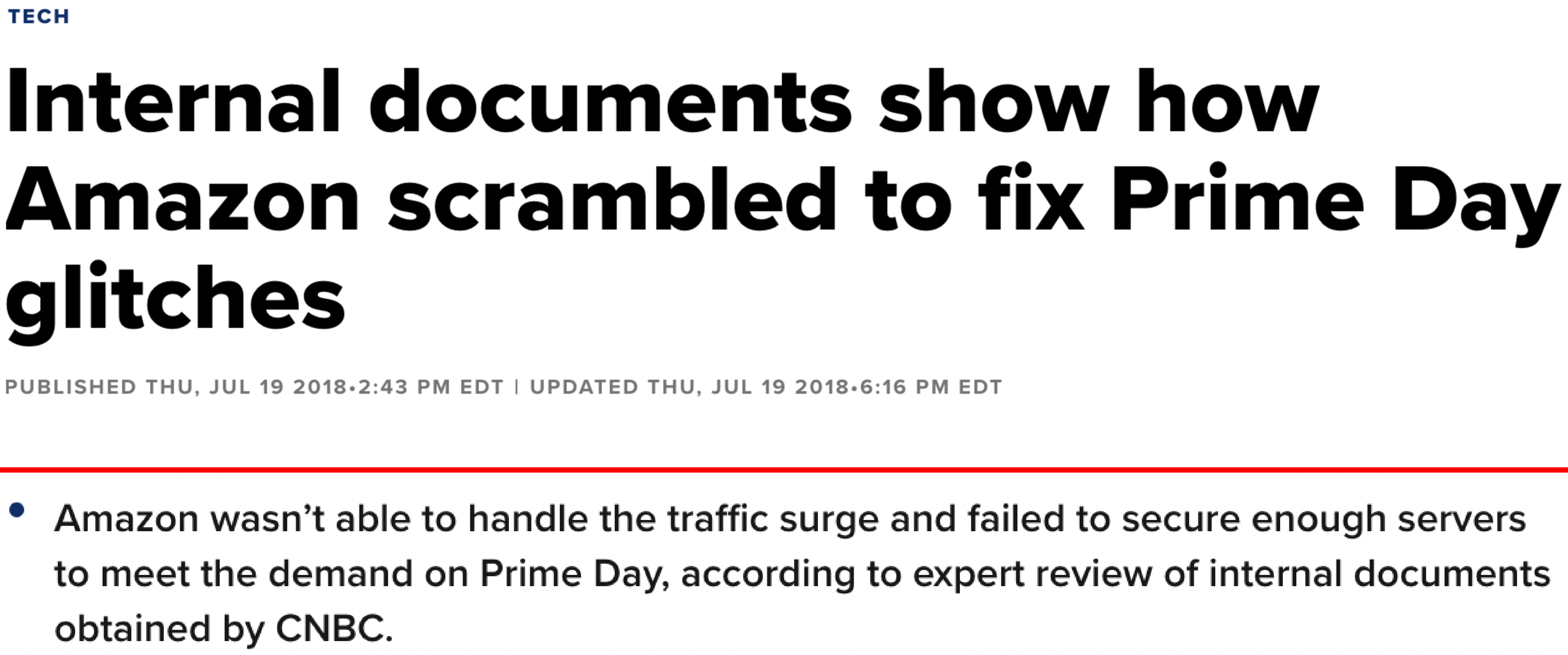
Scalability

What is Scalability?

- The ability of a system to handle **growth in the amount of workload** while maintaining an **acceptable level of performance**
- Why is scalability important?

Prime Big Deal Days

Included with a Prime membership

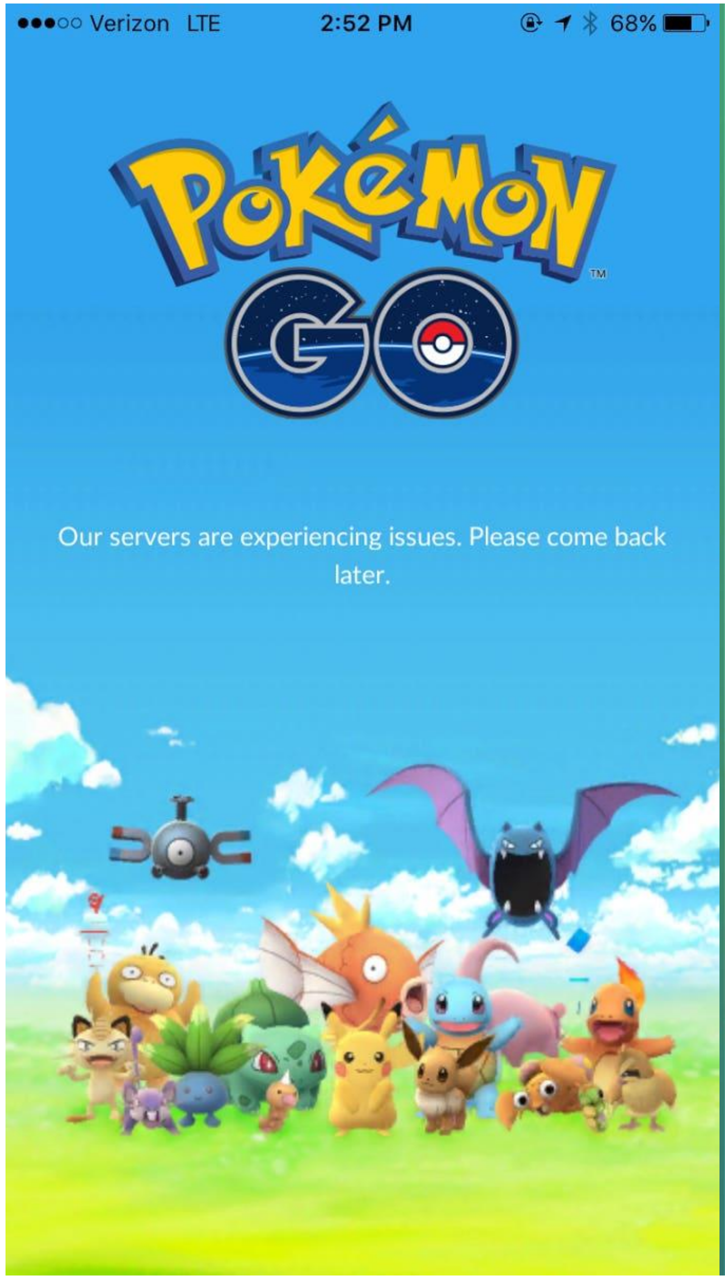


TECH

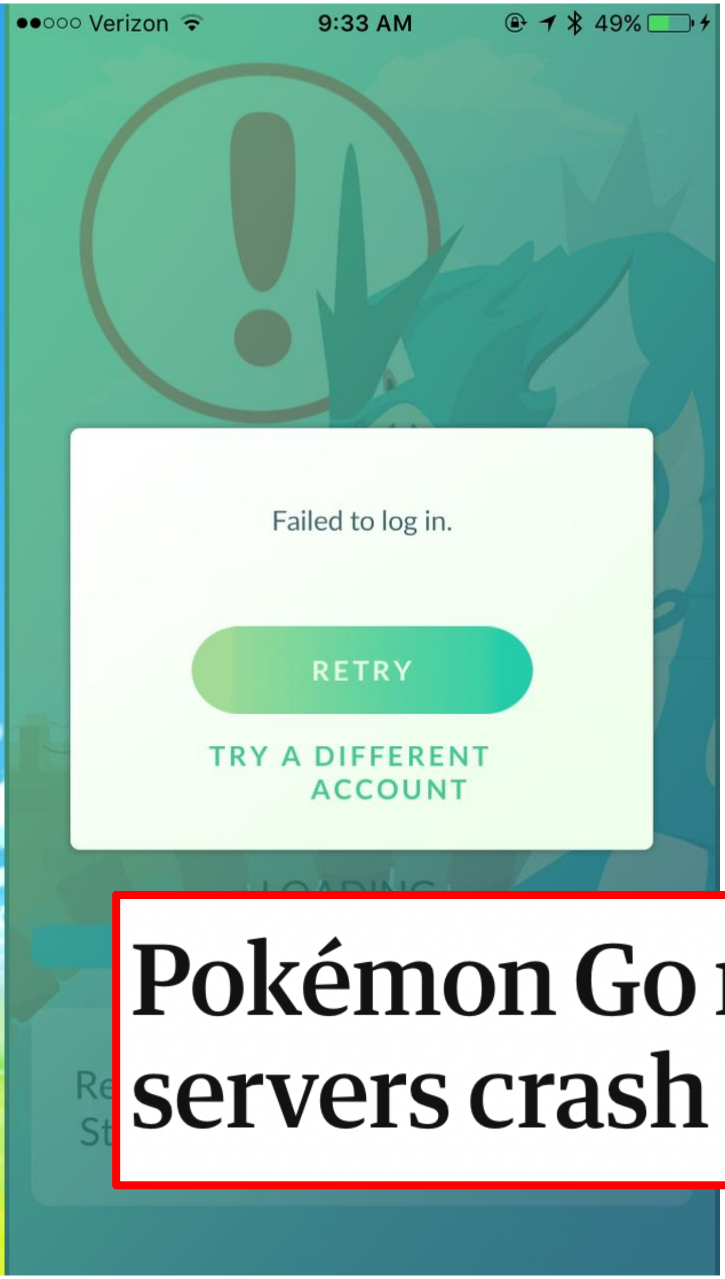
Internal documents show how Amazon scrambled to fix Prime Day glitches

PUBLISHED THU, JUL 19 2018•2:43 PM EDT | UPDATED THU, JUL 19 2018•6:16 PM EDT

- Amazon wasn't able to handle the traffic surge and failed to secure enough servers to meet the demand on Prime Day, according to expert review of internal documents obtained by CNBC.



Our servers are experiencing issues. Please come back later.



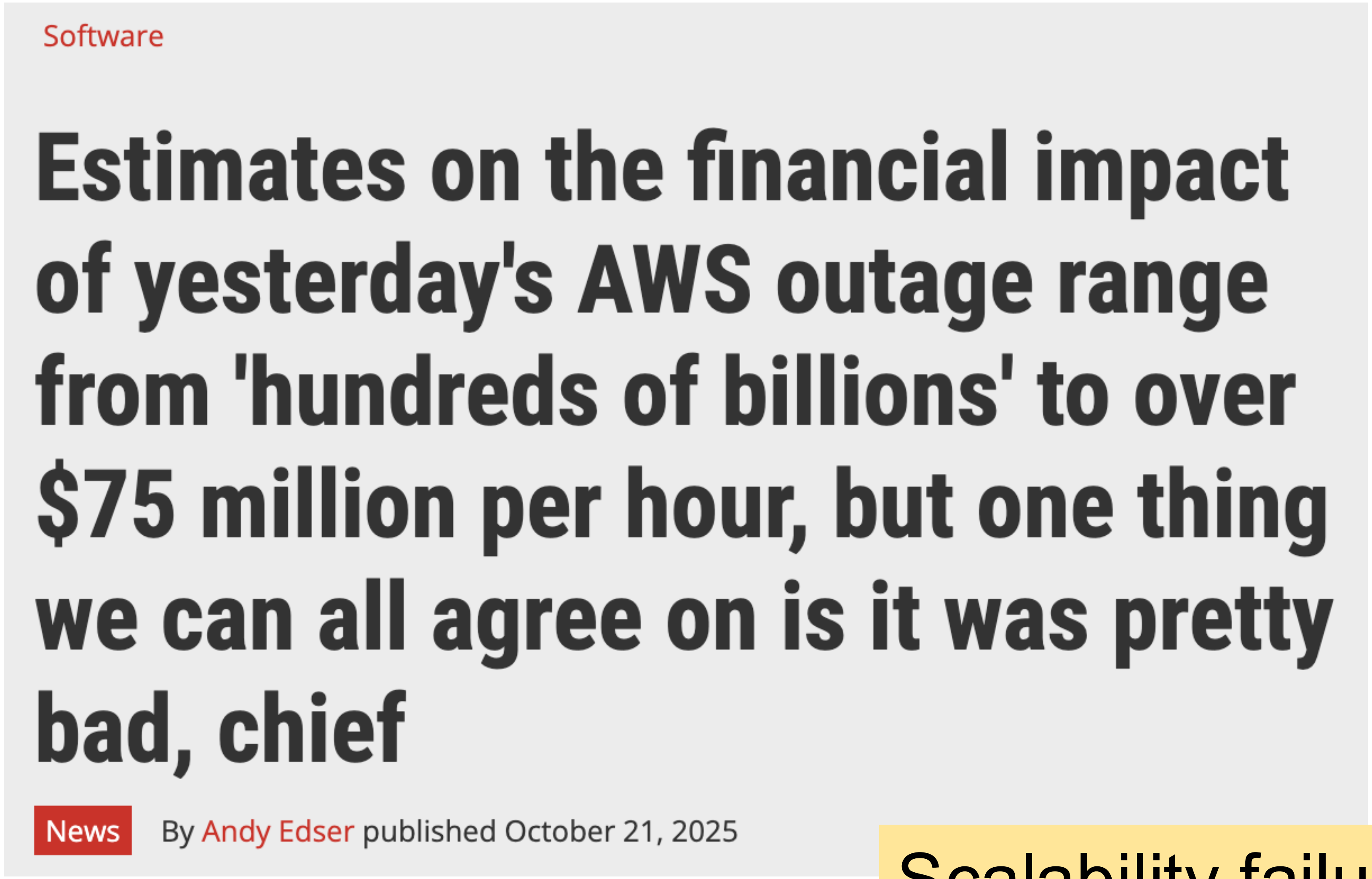
Failed to log in.

RETRY

TRY A DIFFERENT ACCOUNT

Pokémon Go makers call for calm as servers crash across Europe and US

Sat 16 Jul 2016 16.33 EDT

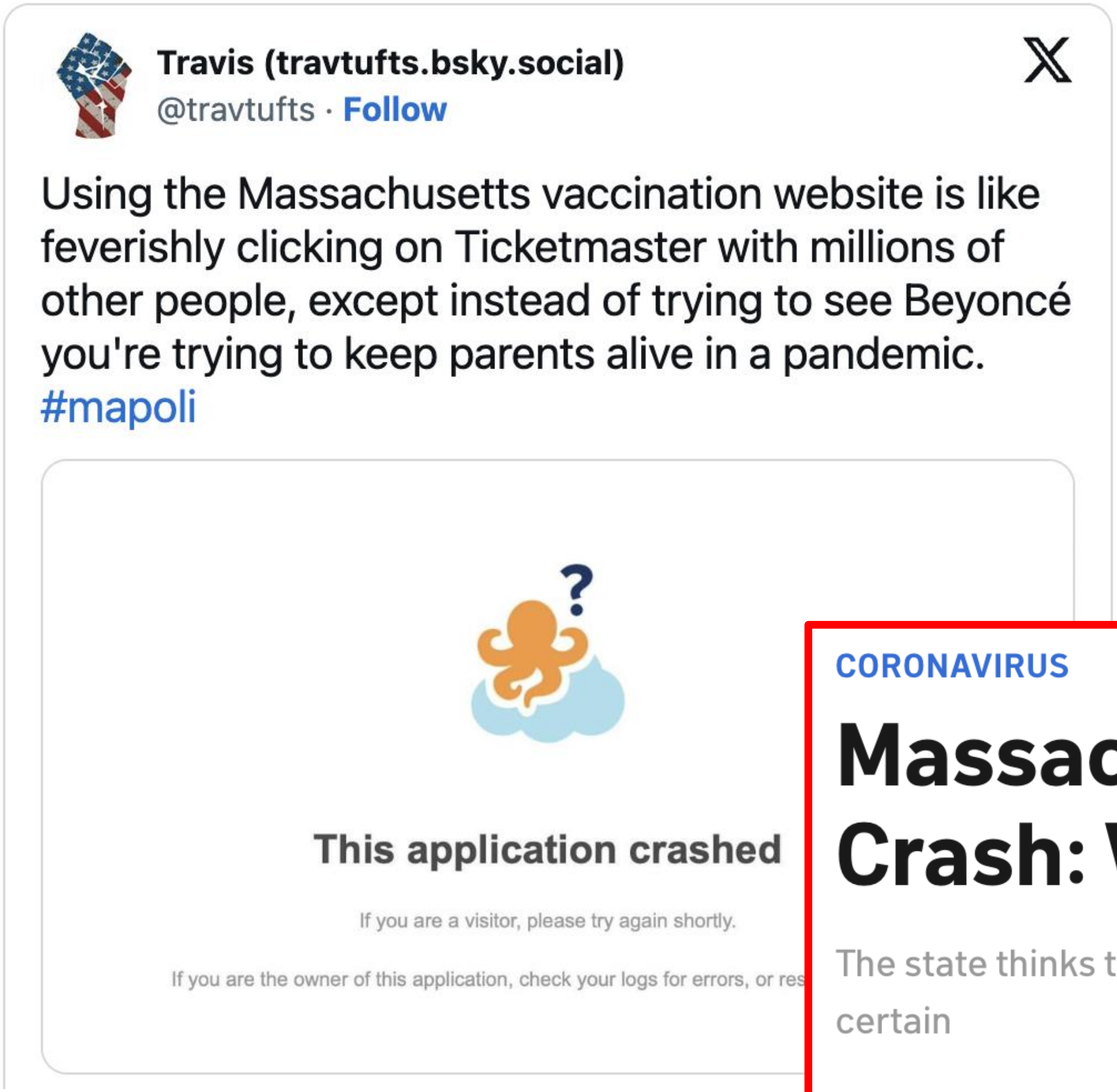


Software

Estimates on the financial impact of yesterday's AWS outage range from 'hundreds of billions' to over $75 million per hour, but one thing we can all agree on is it was pretty bad, chief

News By Andy Edser published October 21, 2025

Scalability failures are expensive!



Travis (travtufts.bsky.social)
@travtufts · Follow

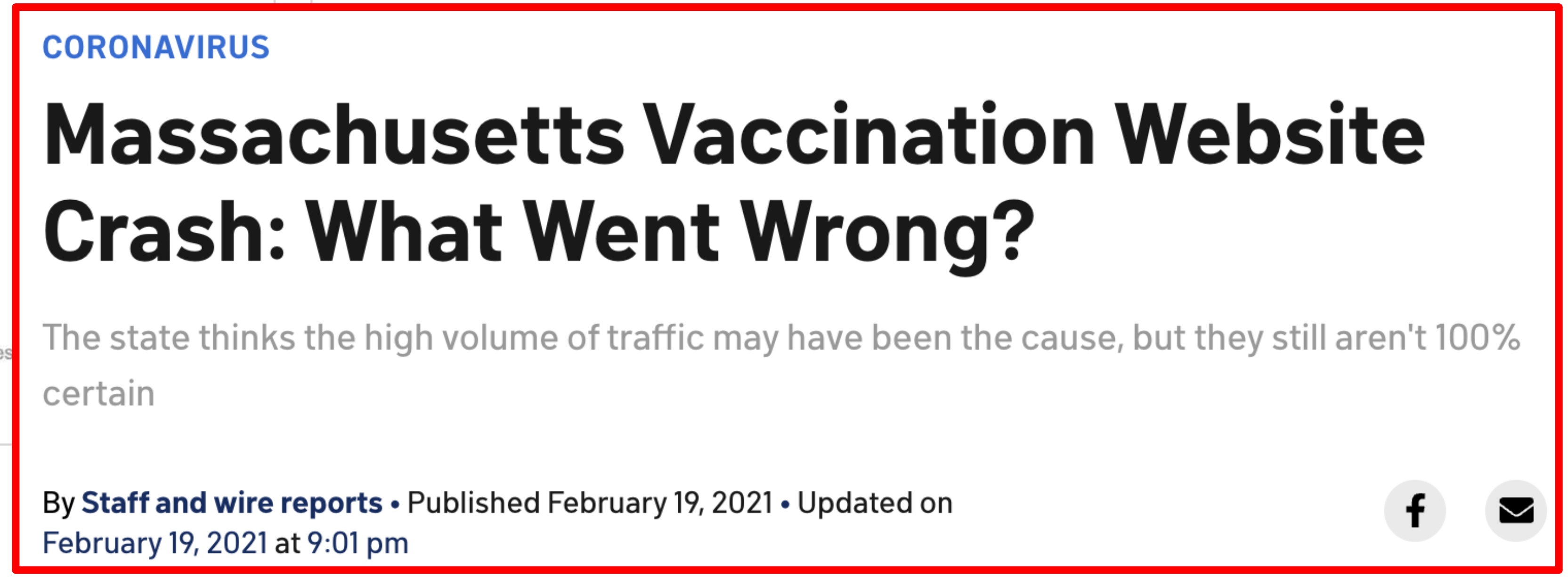
Using the Massachusetts vaccination website is like feverishly clicking on Ticketmaster with millions of other people, except instead of trying to see Beyoncé you're trying to keep parents alive in a pandemic. #mapoli

This application crashed

If you are a visitor, please try again shortly.

If you are the owner of this application, check your logs for errors, or res

Scalability failures can also cause harm to users!



CORONAVIRUS

Massachusetts Vaccination Website Crash: What Went Wrong?

The state thinks the high volume of traffic may have been the cause, but they still aren't 100% certain

By Staff and wire reports • Published February 19, 2021 • Updated on February 19, 2021 at 9:01 pm

Related Concepts

- **Performance**: Amount of resources (e.g., time, memory, disk space) that the system consumes to perform a function
  - It's not just about time or responsiveness (but in this class, we will mostly talk about time-related attributes)
  - It's part of a scalability QA, but not the same!
- **Availability**: Degrees to which the system is available to perform its function(s) at the request of a client
  - Usually expressed in terms of uptime over duration (99.99% available)
- **Reliability**: Degrees to which the system performs its functions **correctly**
  - e.g., Mean time between failures (1000 hours before a sensor failure)
  - Availability does not imply reliability!

Specifying Scalability

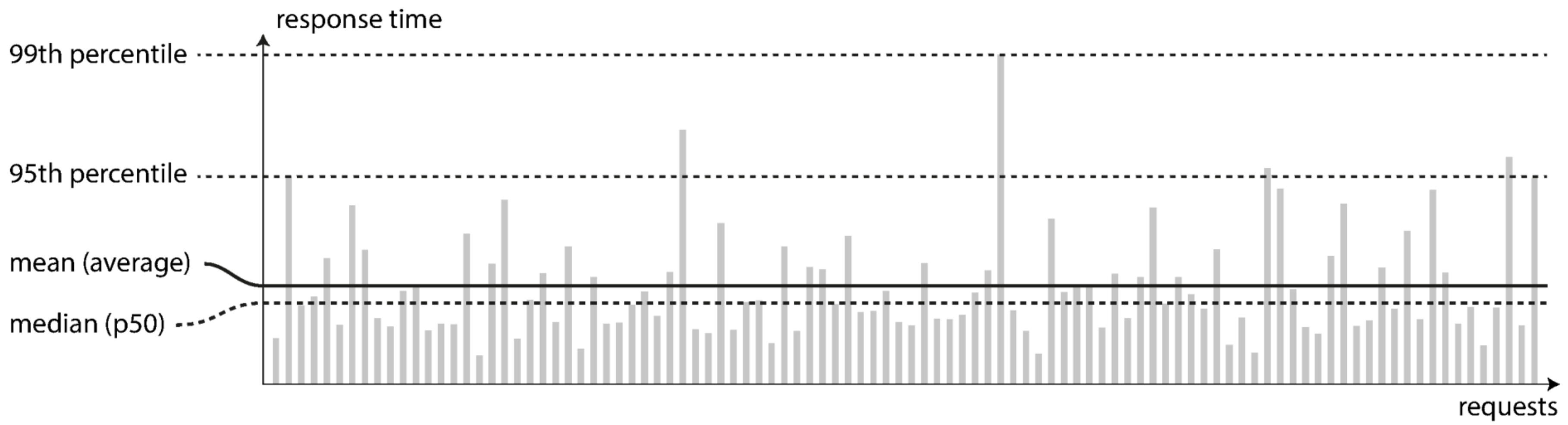
- The ability of a system to handle growth in the amount of **workload** while maintaining an acceptable level of **performance**
- **Workload** (or simply, **load**): Amount of work that the system is given to perform
  - Number of client requests per second, average size of input data, number of concurrent users, etc.,
- **Performance**: Amount of resources that the system consumes to perform a function
  - Average response time, average throughput (i.e., number of requests successfully processed per hour), peak response time, CPU utilization, etc.,

Scalability Specification: Good & Bad Examples

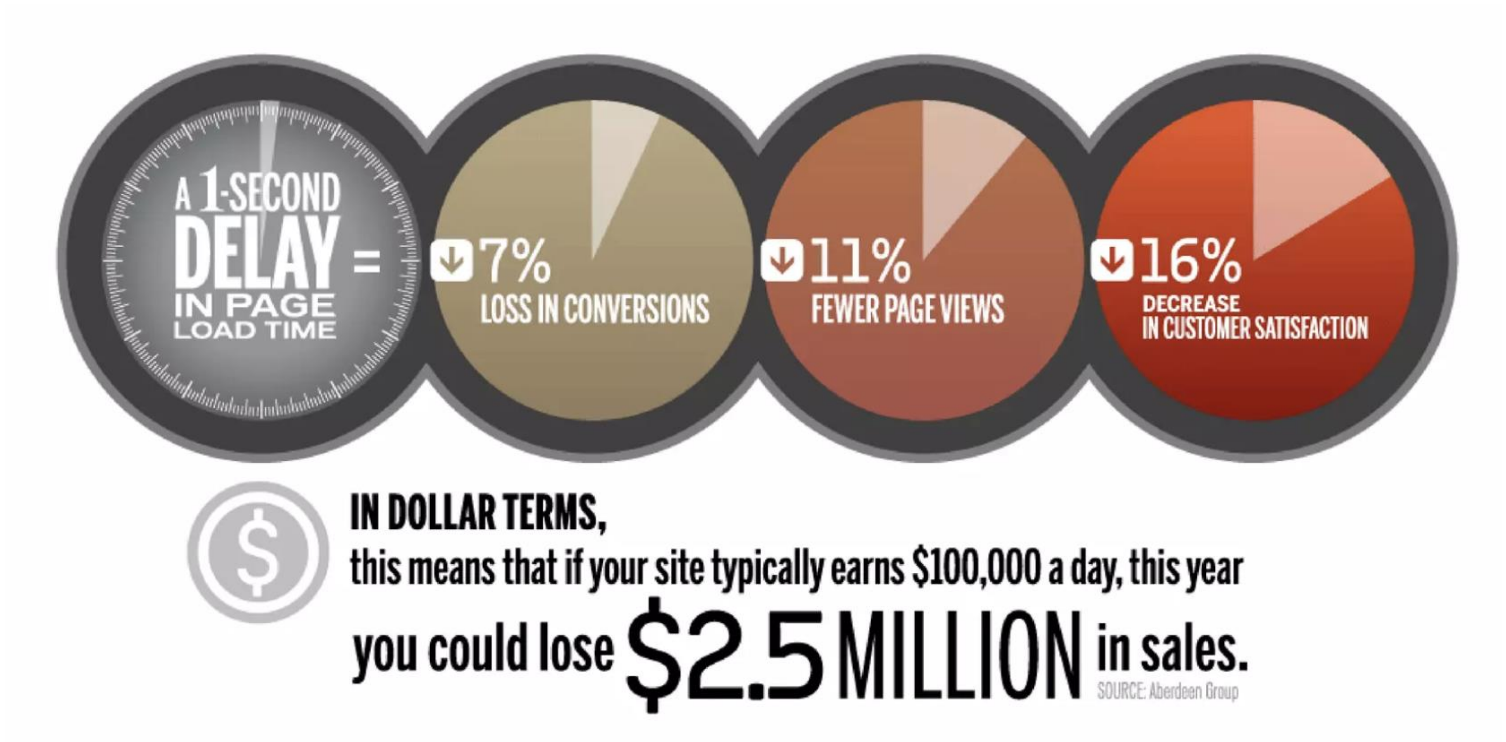
- “Twitter must be able to handle 100 million additional users in the next year” ❌
- “The average time to load a tweet must be no more than 100 ms” ❌
- “Twitter must be able to handle 1 million concurrent requests for viewing tweets with an average response time of 100 ms” ✅
- “On Prime Day, the Amazon storefront should be 100% up and available” ❌
- “Netflix should be able to process addition of 1000 shows per day in its catalog with no more than 2% increase in average latency” ✅
- “The company should achieve active daily users of 10 million by the end of 2024” ❌

Describing Performance: Common Metrics

- **Throughput**: Amount of work processed per time period
- **Response time (RT)/latency**: Time between a client's request & response received
  - Average RT: Commonly used, but not very useful (**Q. why not?)**
  - Percentile RT: "1.5s at 95th" means 95% of requests take < 1.5s
  - Median (50th percentile, or p50): How long users typically wait

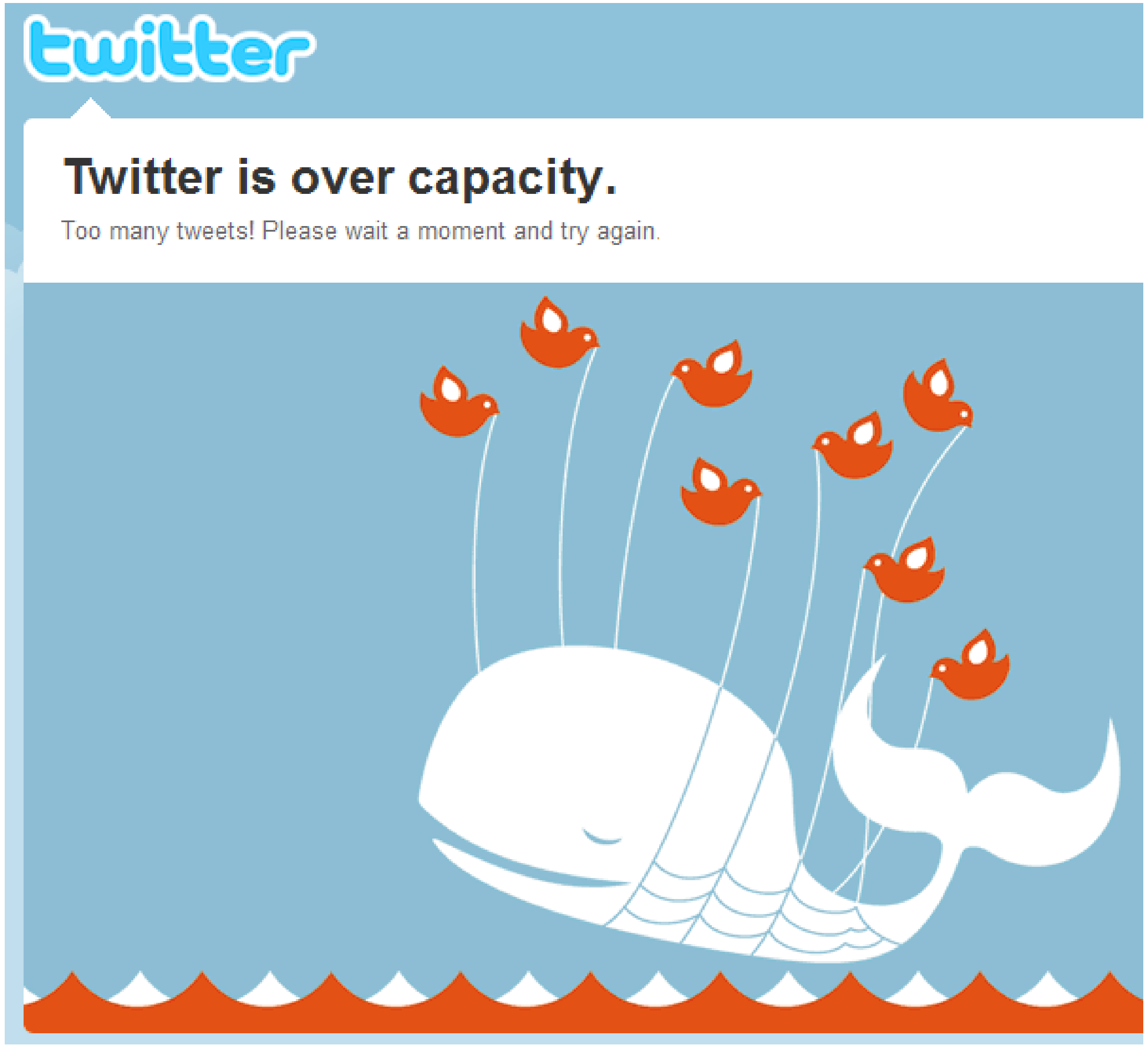


Performance Matters!



- Performance affects other business metrics, such as revenue, conversions/downloads, user satisfaction/retention, time on site, etc.,

Source: The Real Cost of Slow Time vs Downtime, Tammy Everts (2014)

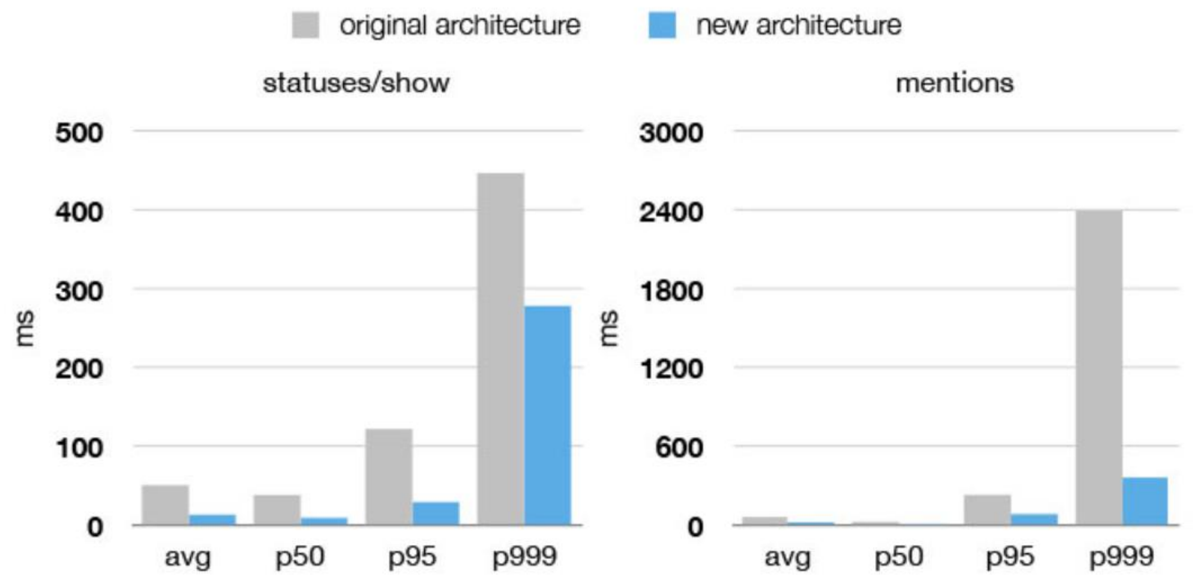

Twitter is over capacity.

Too many tweets! Please wait a moment and try again.

"Twitter fail whale"

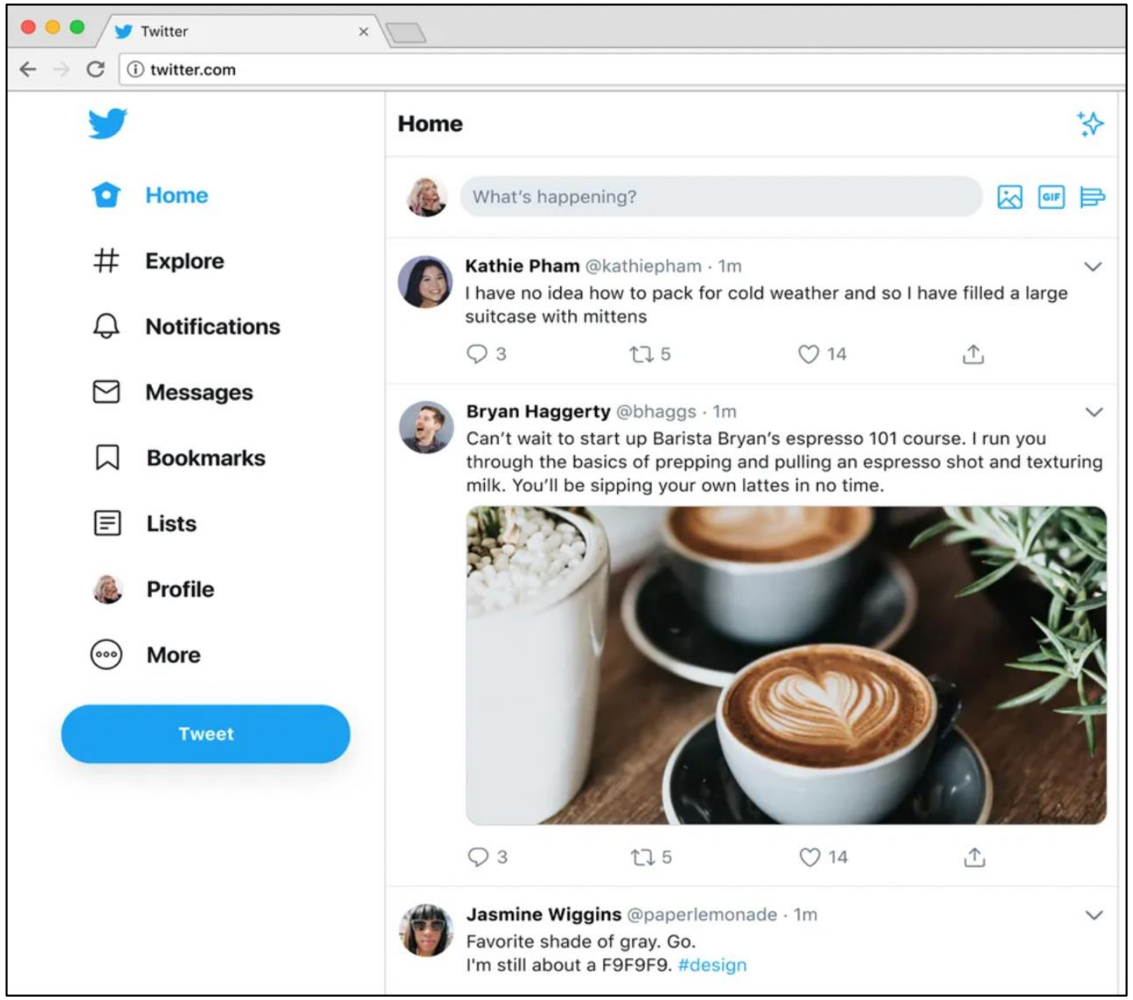
Twitter Redesign for Scalability

- Early Twitter architecture (~2010): Monolithic design running Ruby on Rails, connected to MySQL databases
- Difficulty handling traffic spikes during major events (World Cup, Super Bowl, etc.,)
- Redesign decisions
  - Ruby -> JVM/Scala
  - Monolith -> Microservices
  - Load balancing, continuous monitoring, failover strategies
  - New, distributed database solution



https://blog.twitter.com/engineering/en_us/a/2013/new-tweets-per-second-record-and-how

Workload Parameters Example: Twitter



Kathie Pham @kathiepham · 1m
I have no idea how to pack for cold weather and so I have filled a large suitcase with mittens
3 5 14

Bryan Haggerty @bhaggs · 1m
Can't wait to start up Barista Bryan's espresso 101 course. I run you through the basics of prepping and pulling an espresso shot and texturing milk. You'll be sipping your own lattes in no time.
3 5 14

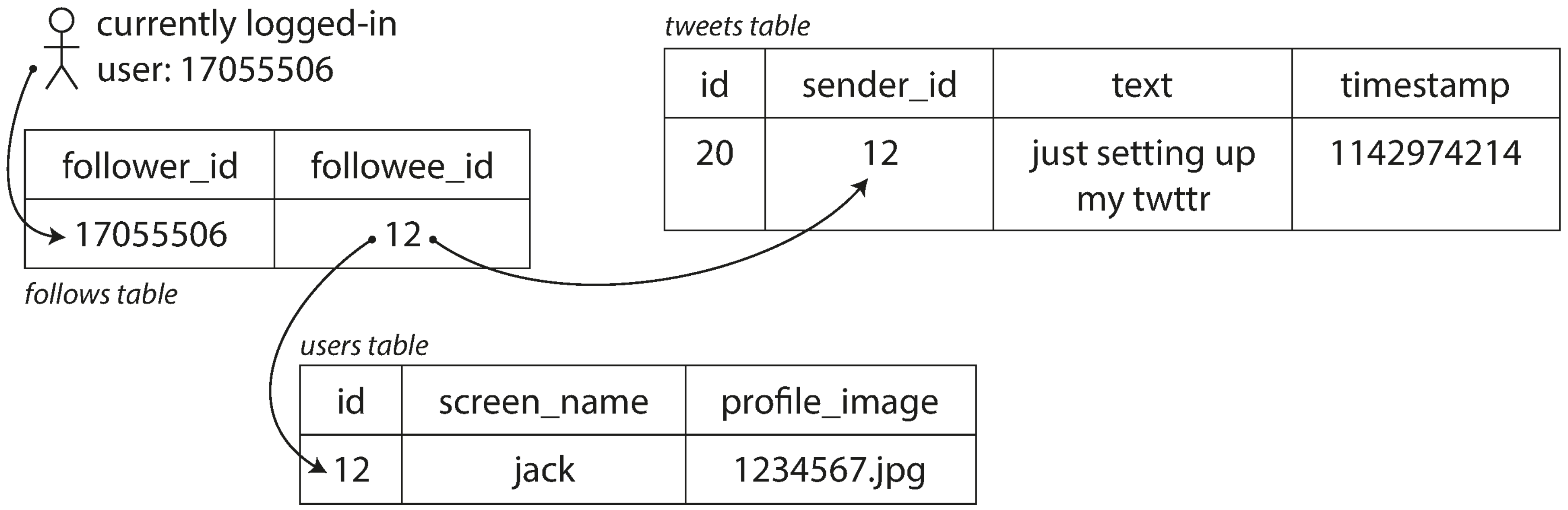
Jasmine Wiggins @paperlemonade · 1m
Favorite shade of gray. Go.
I'm still about a F9F9F9. #design

- Post tweet: Publish a tweet to followers
  - 4.6k requests/sec on average
  - 12k requests/sec at peak
- View home timeline: View tweets posted by the people that the user follows
  - 300k requests/sec on average
- Fan-out problem: Each user follows many people & each user is followed by many people
- **Q**. How would you design these two operations?

Example from: *Designing Data Intensive Applications,* Chapter 1, by M. Kleppmann

Design Option #1: Timeline Reconstruction

- **Post tweet**: Insert the new tweet into a global database of tweets.
- **View timeline**: Reconstruct the timeline for each request - (1) look up all the people the user follows, (2) find all the tweets for each of these people, (3) merge & sort by time



follows table

follower_id	followee_id
17055506	12

tweets table

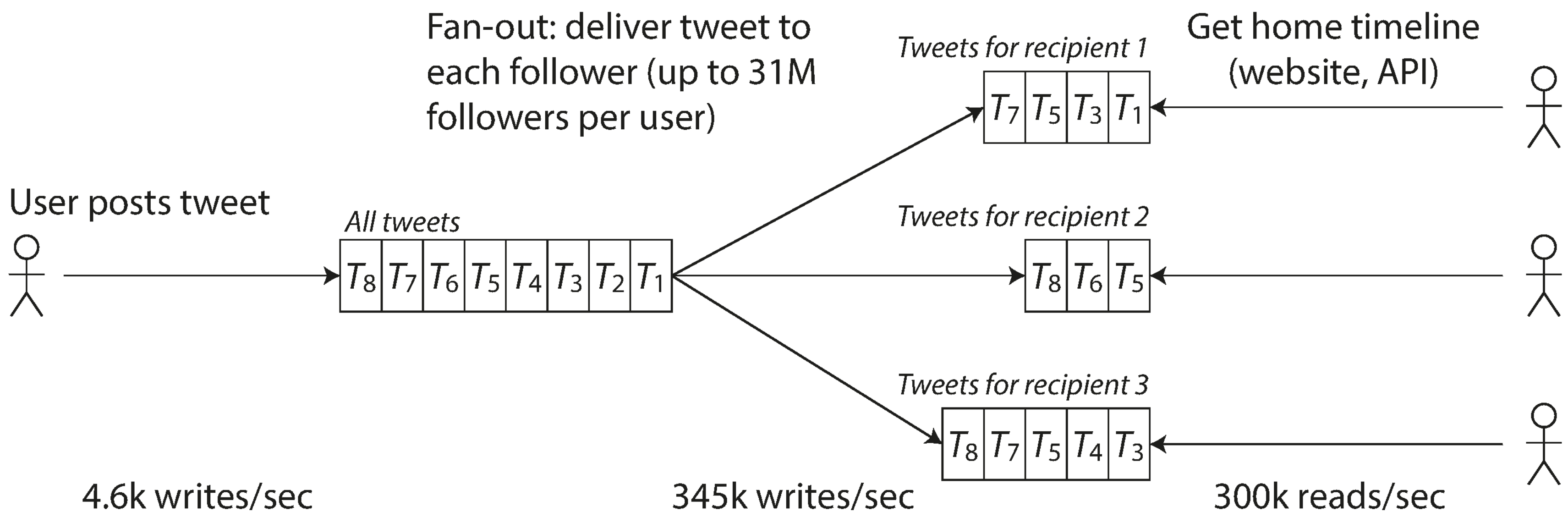
id	sender_id	text	timestamp
20	12	just setting up my twttr	1142974214

users table

id	screen_name	profile_image
12	jack	1234567.jpg

Design Option #2: Timeline Cache

- For each user, maintain the current view (cache) of their timeline
- **Post tweet**: (1) Look up all the people who follows the user and (2) insert the new tweet into each of their timeline
- **View timeline**: Inexpensive; no need to re-compute the timeline



Fan-out: deliver tweet to each follower (up to 31M followers per user)
Tweets for recipient 1
Get home timeline (website, API)
User posts tweet
All tweets
T_8 T_7 T_6 T_5 T_4 T_3 T_2 T_1
T_7 T_5 T_3 T_1
Tweets for recipient 2
T_8 T_6 T_5
Tweets for recipient 3
T_8 T_7 T_5 T_4 T_3
4.6k writes/sec
345k writes/sec
300k reads/sec

Discussion: Which Option?

- Design Option #1: Reconstruct the timeline at every request
- Design Option #2: Maintain & update a cache of timeline
- **Q. Which option would you prefer for better scalability, and under what assumptions?**
    - **What additional information about the workload do you need to make the decision?**

Discussion: Which Option?

- **Design Option #1**: Reconstruct the timeline at every request
  - Used in an older design of Twitter, but difficulty dealing with an increasing number of timeline requests
- **Design Option #2**: Maintain & update a cache of timeline
  - More scalable for timeline requests
  - For post tweet: 4.6k writes/sec * (**<u>average # followers</u>** ~ 75) = 345k writes/sec
  - But what if a user (e.g., celebrity) has a very large number of followers?
- In practice: **Hybrid solution**!
  - Option #2 for most users; Option #1 for users with very high # followers
- **<u>No one solution fits all!</u>** Design decisions for scalability must consider workloads under different scenarios and user interactions.

Workload Patterns

Workload Patterns

- Scalability is NOT about just adding more servers
- It's about understanding the **patterns of workloads** and making design decisions that align with them
  - Two systems that process the same volume of data can have completely different designs based on workload patterns!
  - Scalability failures often occur due to a wrong/missing assumption about workloads
- **<u>Dimensions of workloads</u>**
  - Read-heavy vs write-heavy
  - Hot vs. cold data
  - Burst traffic vs steady workload
  - Few large requests vs many small requests

Read-heavy vs. Write-heavy

Read-heavy

- Most operations involve retrieving data
- Reads far outweigh writes
- **Examples**:
  - Tweeter timeline view
  - Wikipedia (mostly reads; edits are comparatively rare)
  - Product catalog (browsing is much more frequent than update)

Write-heavy

- Most operations write or update data
- Writes are the major bottleneck
- **Examples**:
  - IoT sensor (e.g., temperature) logging
  - Payment processing
  - Real-time analytics (e.g., clickstreams)

Hot vs. Cold Data (Distribution of Data Access)

Hot data

- Small fraction of the application data is accessed frequently
- **Examples**: Top 100 Netflix titles, viral videos, trending search queries

Cold data

- Large fraction of data is accessed rarely
- **Examples**: Historical (i.e., old) tweets, archived Netflix titles, IoT sensor logs

Note: Many systems have a mix of both hot & cold data, while some may have mostly cold data

Burst vs. Steady Workload

Steady load

- Workload is consistent over time
- Easier to design for
- **Examples**: Internal enterprise tool with a fixed user base

Predictable burst

- Spikes occur on a known schedule
- Can pre-scale capacity before those events
- **Examples**: Amazon Prime Day, Super Bowl, tax season

Unpredictable burst

- Spikes occur without a warning
- Must handle surge in real time or degrade gracefully
- **Examples**: Pokemon Go launch, COVID vaccination

Few large requests vs. many small requests

Few large requests

- Small number of requests, each transferring large amount of data
- Bottleneck: Bandwidth and connection throughput
- **Examples**: Video streaming (e.g., 2GB file), batch ML training data upload

Many small requests

- High volume of requests, each small and fast
- Bottleneck: Connection management, concurrency, queuing
- **Examples**: Tweet lookups, web page requests

Combining Workload Dimensions

System	Read/Write	Hot/Cold	Burst/Steady	Request Size
Twitter timeline	Read-heavy	Hot & cold	Predictable burst	Many small
IoT sensor log	Write-heavy	Cold	Steady	Many small
Youtube videos	??	??	??	??
Ticketmaster sales	??	??	??	??
Google Docs collaboration	??	??	??	??

Combining Workload Dimensions

System	Read/Write	Hot/Cold	Burst/Steady	Request Size
Twitter timeline	Read-heavy	Hot & cold	Predictable burst	Many small
IoT sensor log	Write-heavy	Cold	Steady	Many small
Youtube videos	Read-heavy	Hot & cold	Predictable burst	Many large (!)
Ticketmaster sales	??	??	??	??
Google Docs collaboration	??	??	??	??

Combining Workload Dimensions

System	Read/Write	Hot/Cold	Burst/Steady	Request Size
Twitter timeline	Read-heavy	Hot & cold	Predictable burst	Many small
IoT sensor log	Write-heavy	Cold	Steady	Many small
Youtube videos	Read-heavy	Hot & cold	Predictable burst	Many large (!)
Ticketmaster sales	Write-heavy	Hot	Unpredictable burst	Many small
Google Docs collaboration	??	??	??	??

Combining Workload Dimensions

System	Read/Write	Hot/Cold	Burst/Steady	Request Size
Twitter timeline	Read-heavy	Hot & cold	Predictable burst	Many small
IoT sensor log	Write-heavy	Cold	Steady	Many small
Youtube videos	Read-heavy	Hot & cold	Predictable burst	Many large (!)
Ticketmaster sales	Write-heavy	Hot	Unpredictable burst	Many small
Google Docs collaboration	Mix	Hot & Cold	Predictable burst	Many small

Workload Patterns: Remarks

- Many systems provide multiple features or APIs with different clients/traffic sources
  - Need to identify corresponding workload patterns for these
  - **Example**: E-commerce site: Customer API (browsing & buying products) vs. seller API (inventory updates)
- Workload patterns are not fixed; they can evolve over time!
  - **Example**: Early-stage social media vs. after viral growth
- Workload patterns are difficult to predict before deployment
  - But we can often make a good guess based on similar systems and prior experiences
- Different workload patterns call for different scaling strategies!

Workload Challenges & Design Strategies

Workload	Challenges	Design Strategy
Read-heavy	Database read bottleneck	Replication, caching
Write-heavy	Database write bottleneck; replication lag	Partitioning
Hot data	Hotspots	Caching, content delivery network (CDN)
Burst traffic	Server overload; latency spikes; cascading failures	Load balancing, auto scaling, rate limiting
Large data volume	Bandwidth saturation; data storage limits	Replication, partitioning

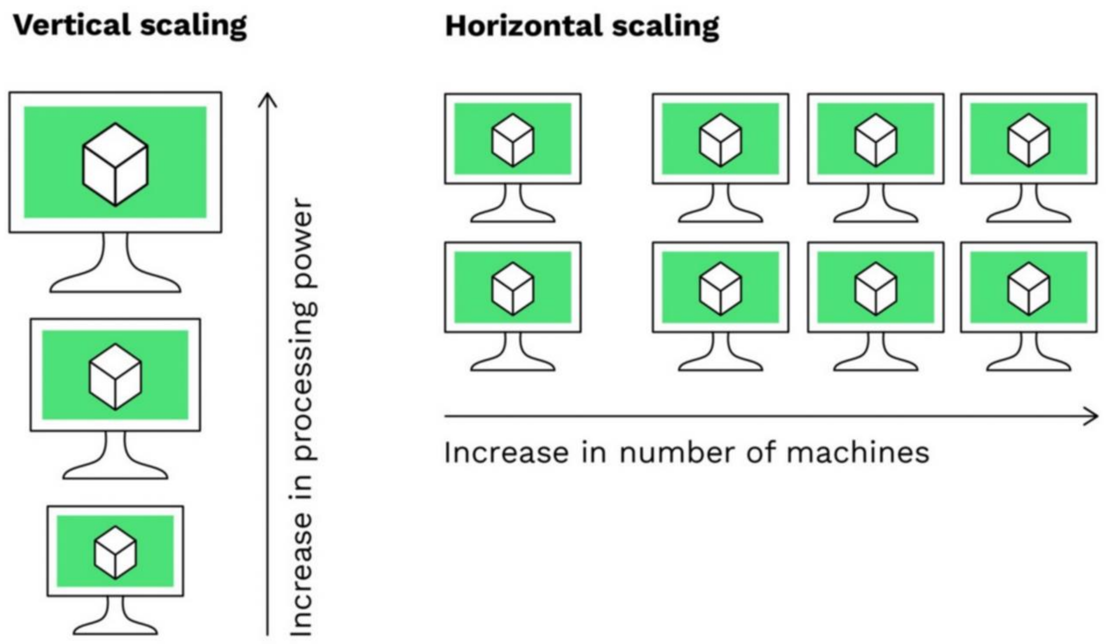
Design Strategies for Scalability

Common Design Problems for Scalability

- How do we increase capacity to handle additional workload? **Vertical & horizontal scaling**
- How do we avoid overloading one part of the system due to increased workload? **Load balancing**
- How do we reduce bottlenecks that arises from repeated data access? **Caching**

Vertical vs. Horizontal Scaling

- **Problem: How do we increase capacity to handle additional load?**
- **Vertical scaling** (*scaling up*): Get a machine with more capacity
- **Horizontal scaling** (*scaling out*): Distribute the workload across multiple machines



Vertical vs. Horizontal Scaling

- **Problem: How do we increase capacity to handle additional load?**
- **Vertical scaling** (*scaling up*): Get a machine with more capacity
- **Horizontal scaling** (*scaling out*): Distribute the workload across multiple machines
- **Q. What are the benefits & downsides of each approach?**

Vertical vs. Horizontal Scaling

- **Problem: Ho** **dditional load?**
- **Vertical scal** capacity
- **Horizontal s** ad across multiple mac
- **Q. What are** **oach?**
    - So does th r choice?

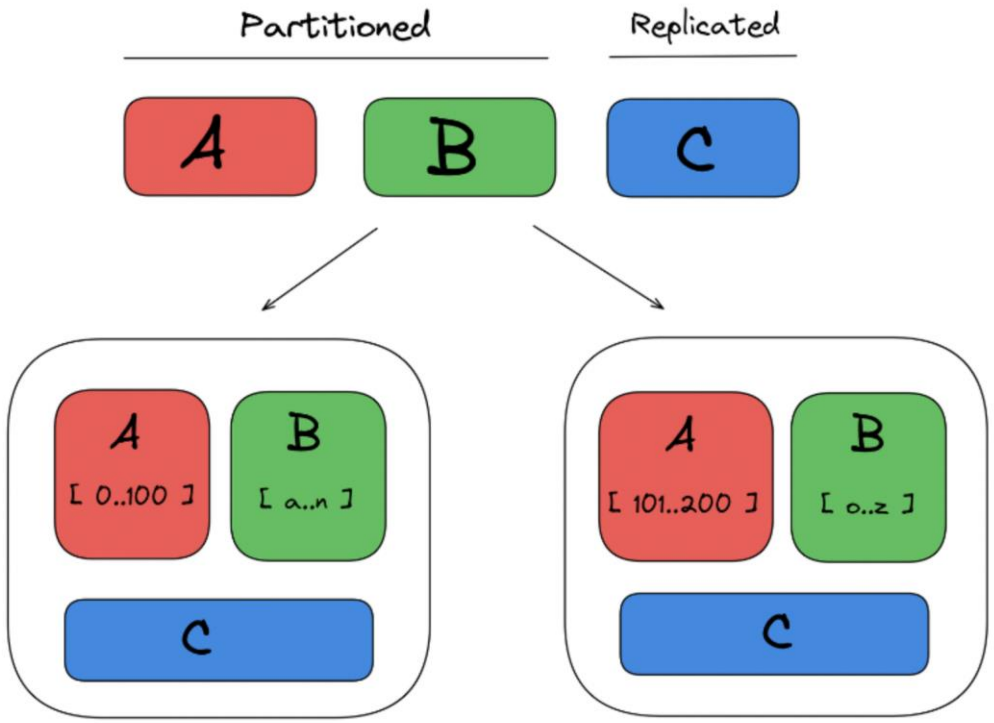
Vertical vs. Horizontal Scaling

- Vertical scaling is limited by machine capacities; horizontal scaling is often unavoidable for highly scalable systems
- But horizontal scaling introduces many complexities
  - **Load balancing**: Distribute requests
  - **Distributed data**: Partition or replicate data
  - **Consistency**: Keep replicas consistent
  - **Fault tolerance**: Handle machine failures
- Vertical scaling, where possible, is simpler and more efficient
- In practice, most systems use a hybrid approach
  - **Example**: StackExchange Architecture

Horizontal Scaling

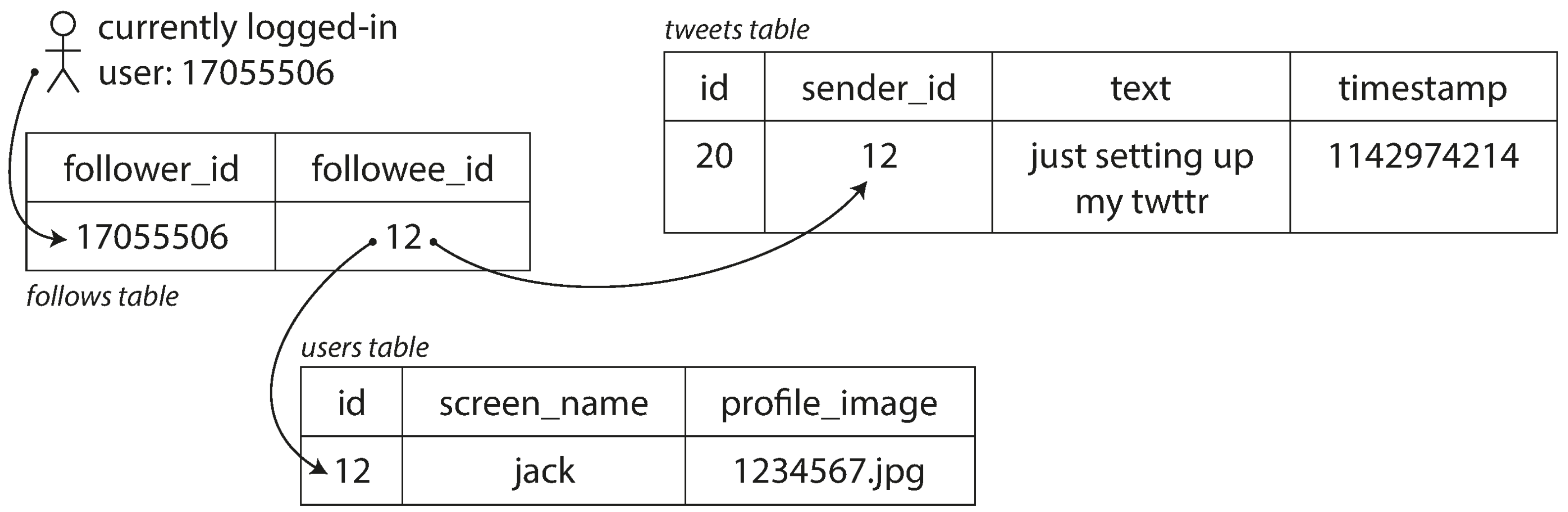
Horizontal Scaling through Distributed Data

- **Horizontal scaling**: Distribute workload across multiple machines
  - Typically involves distributing & storing data across those machines
- Two common ways to distribute data
  - **Replication**: Make copies of data
  - **Partitioning**: Split data and store them over multiple servers
- Many systems use a hybrid approach that combines both, depending on workload patterns on specific data



Relational vs. Document Model of Data

- **Relational data model**: Schemas, tables & queries (e.g., SQL)



tweets table

id	sender_id	text	timestamp
20	12	just setting up my twttr	1142974214

follower_id	followee_id
17055506	12

follows table

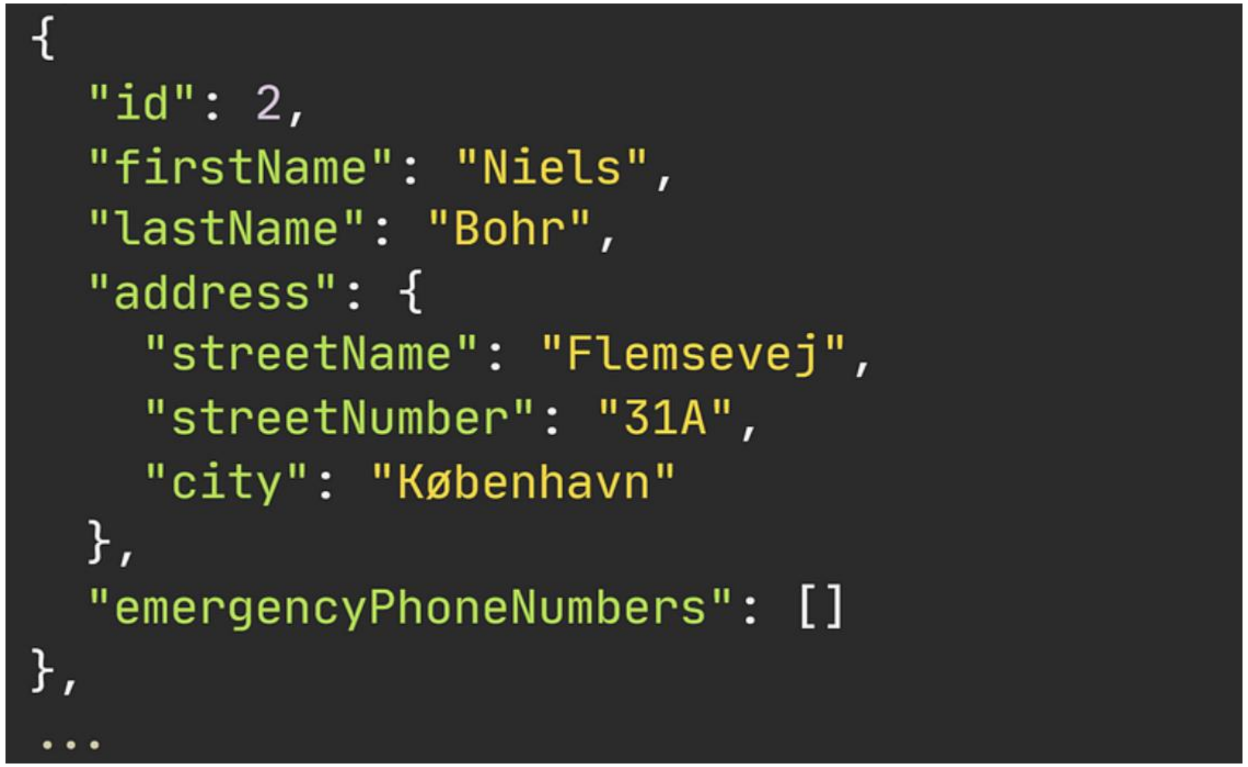
users table

id	screen_name	profile_image
12	jack	1234567.jpg

```
SELECT tweets.*, users.* FROM tweets
  JOIN users   ON tweets.sender_id    = users.id
  JOIN follows ON follows.followee_id = users.id
  WHERE follows.follower_id = current_user
```

Relational vs. Document Model

- **Relational data model**: Schemas, tables & queries
- **Document model**: No fixed schema, semi-structured (e.g., JSON/XML)

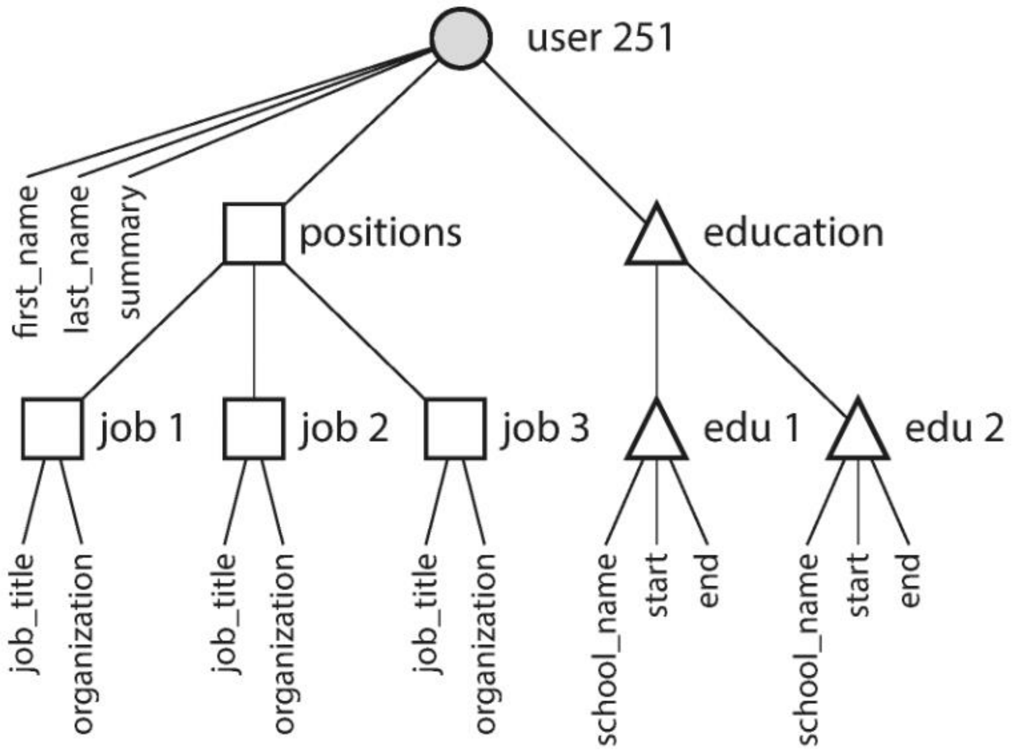


```
{
  "id": 2,
  "firstName": "Niels",
  "lastName": "Bohr",
  "address": {
    "streetName": "Flemsevej",
    "streetNumber": "31A",
    "city": "København"
  },
  "emergencyPhoneNumbers": []
},
...
```

Q. Benefits & drawbacks of each model?
When would you choose one over the other (in relation to scalability)?

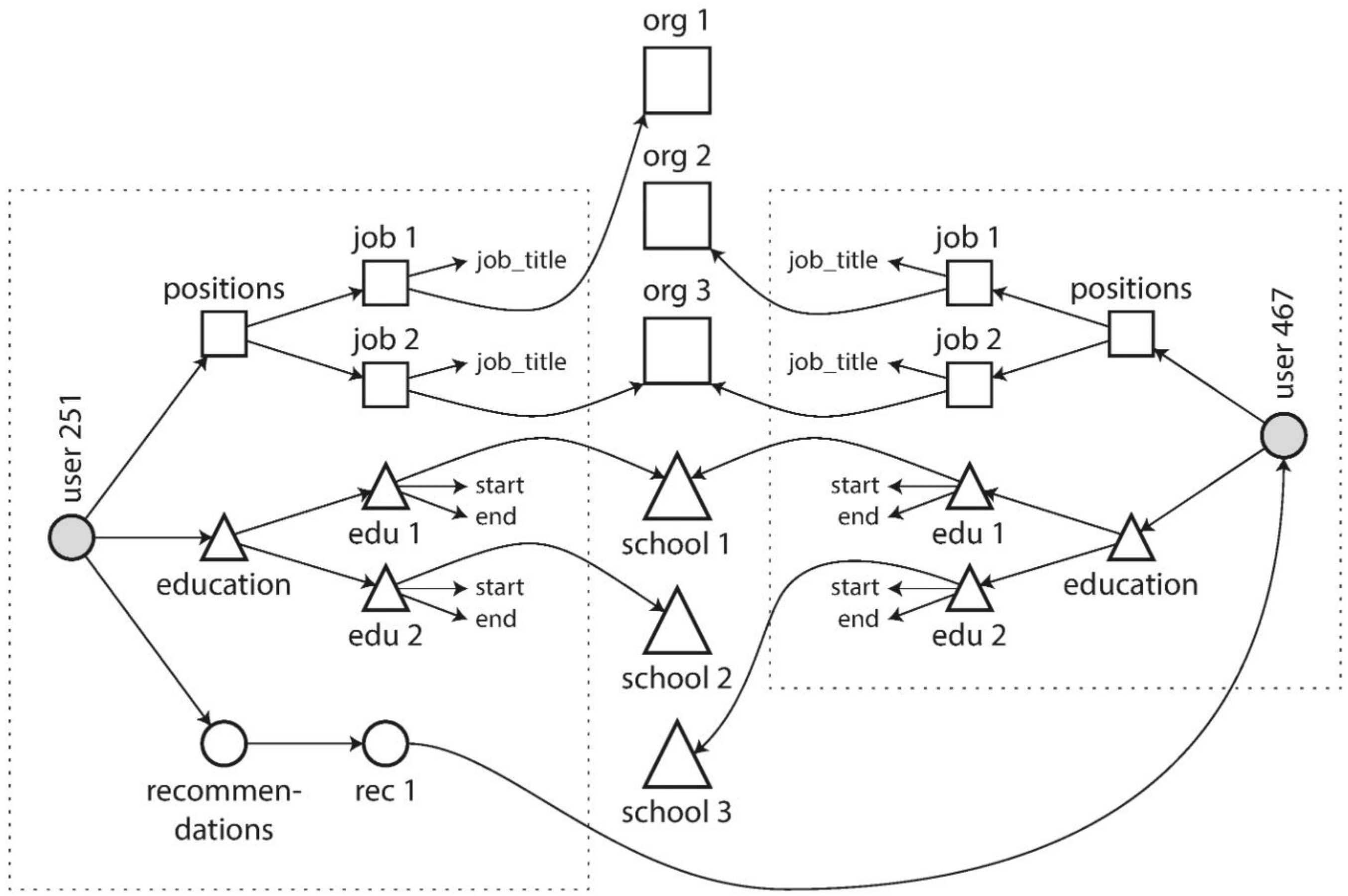
When to use document model?

- Which model to use? **Again, it depends!**
- **One-to-many** relationship between data elements
  - Forms a tree-like structure
  - Data query through indexing operations
    - e.g., positions[0]
- **Data locality**
  - A document (e.g., JSON) object is stored as a continuous string
  - Efficient when the entire object needs to be accessed at once
- More **lightweight** than relational model



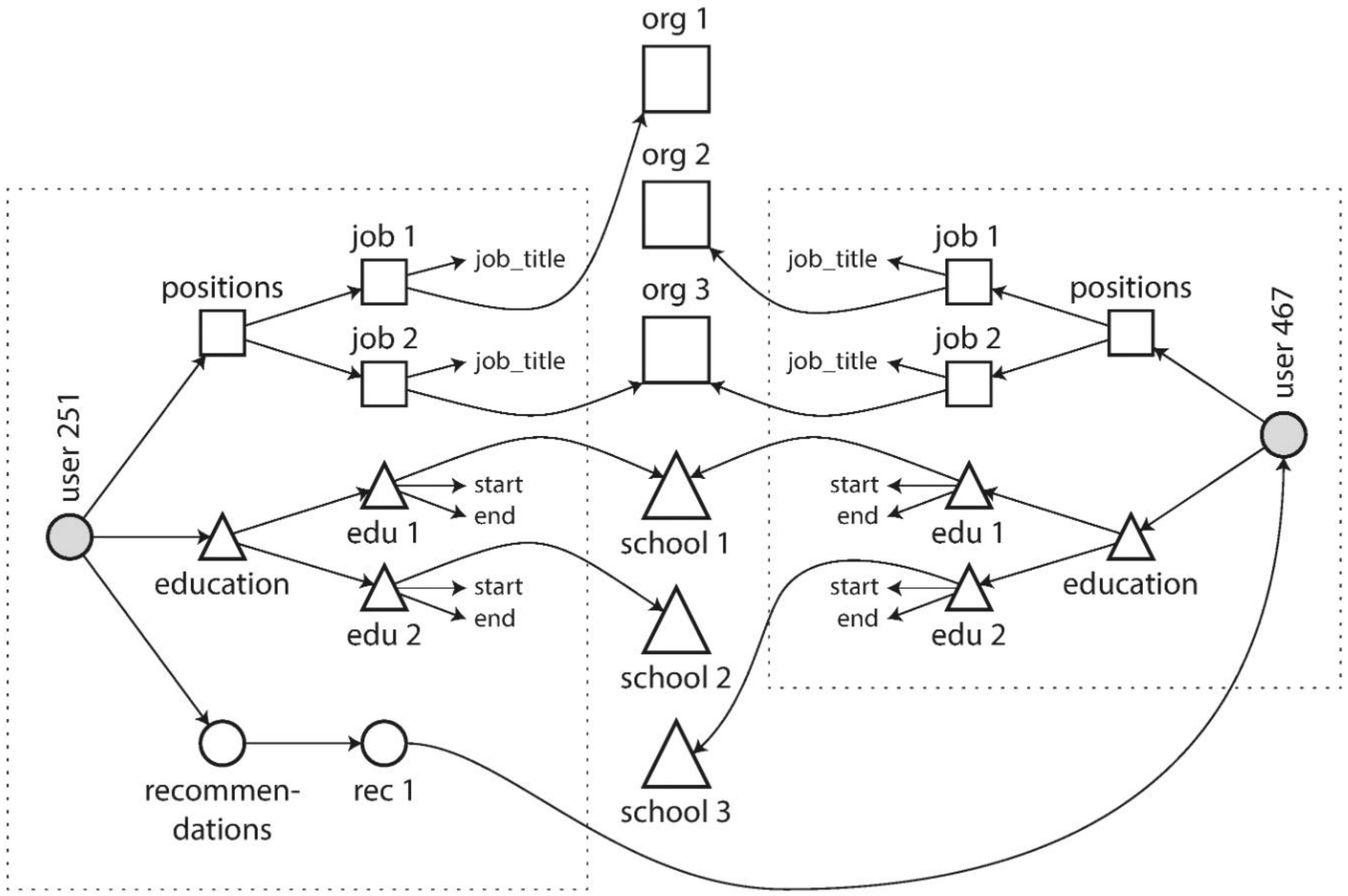
When to use relational model?

- **Many-to-many** relationships between data elements
  - Data query involves relational join
  - Awkward and inefficient in document model
- **Data integrity** and **transaction guarantees**
  - Schema and integrity constraints enforced
  - **Transaction**: When something goes wrong, roll back to previous state



When to use relational model?

- In general**, more heavyweight** than document model
- Modern DB engines are very efficient in query processing
- DB engines over distributed databases also exist
  - But these tend to be more complex and incur high performance costs
- Relational model is better suited for data that can fit on a single server



Distributed Data: Replication

- **Replication**: Copy & store data across multiple machines (or *nodes*), possibly in different geographical locations
  - **Fault-tolerant**: If some nodes become unavailable, data can be access from the remaining nodes
  - **Performance**: Requests can be directed to a node that is physically closer (reduced latency)
  - **Scalability**: Increased load can be handled by adding more nodes with replicated data

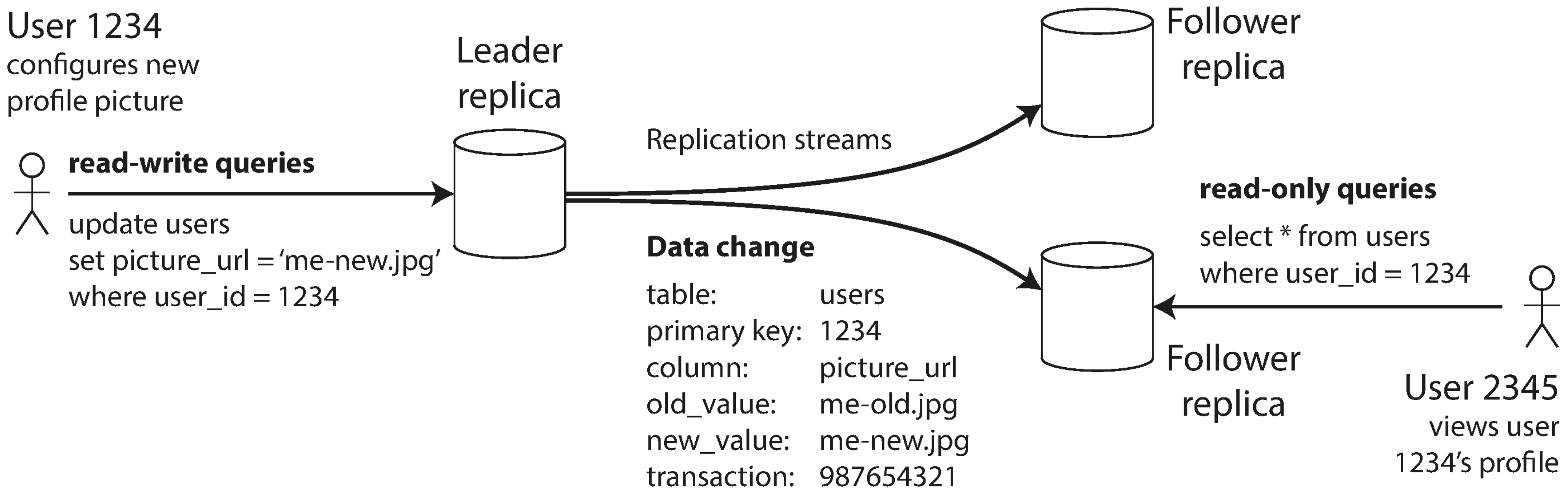
Q. This sounds great! What’s the catch?

Replication: Challenges

- **Consistency**: If data on one node changes, how do we ensure that all its replicas have the same, consistent data?
  - Clients may read **outdated** data from inconsistent nodes
  - **Node failures**: What if some of the nodes fail before updating its data?
- There are several different approaches to dealing with these challenges
- This is a mature and still active area of research (called *distributed systems*); you can take multiple courses on this topic alone
- We will cover one well-known approach: **Leader-follower model**

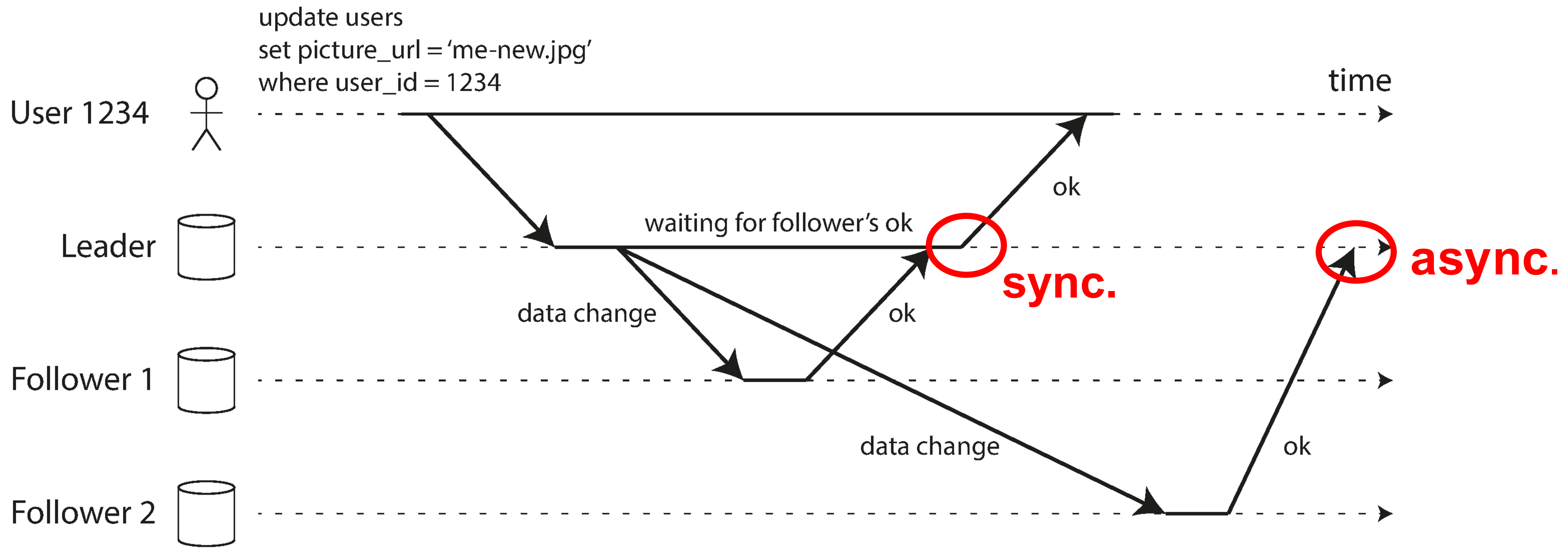
Leader-Follower Model

- Designate one of the replicas as the **leader**; the rest are **followers**
- Write operations are allowed only on the leader
- When data changes on the leader, send update to every follower



Synchronous vs. Asynchronous Replication

- **Synchronous**: The leader waits until the follower confirms that it has received the update
- **Asynchronous**: The leader sends the update and continues without confirmation



Synchronous vs. Asynchronous Replication

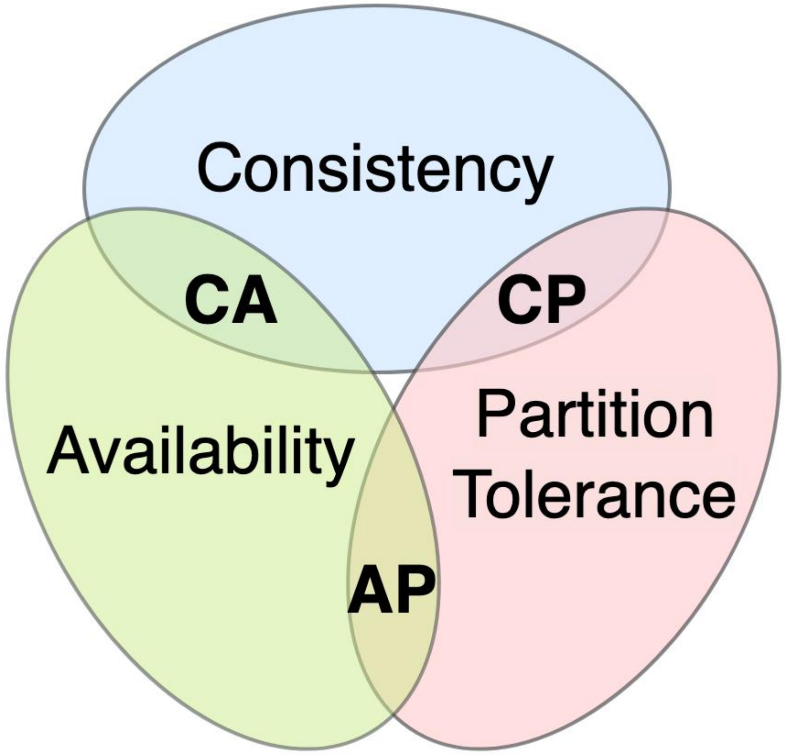
- **Synchronous**: The leader waits until the follower confirms that it has received the update
- **Asynchronous**: The leader sends the update and continues without confirmation
- **Q. What are the benefits & downsides of each design? In what types of applications does one approach make more sense over the other?**

Synchronous vs. Asynchronous Replication

- **Synchronous**: The leader waits until the follower confirms that it has received the update
  - Pros: Ensures that followers have updated data. If the leader fails, latest data can still be read from the followers
  - Cons: Higher latency for the client; some followers may fail and never return a confirmation
- **Asynchronous**: The leader sends the update and continues without confirmation
  - Pros: Higher performance; the leader can continue to process client requests
  - Cons: Weaker guarantees on consistency across replicas
- **Hybrid model:** Assign some followers to be synchronous, the others asynchronous

Digression: CAP Theorem

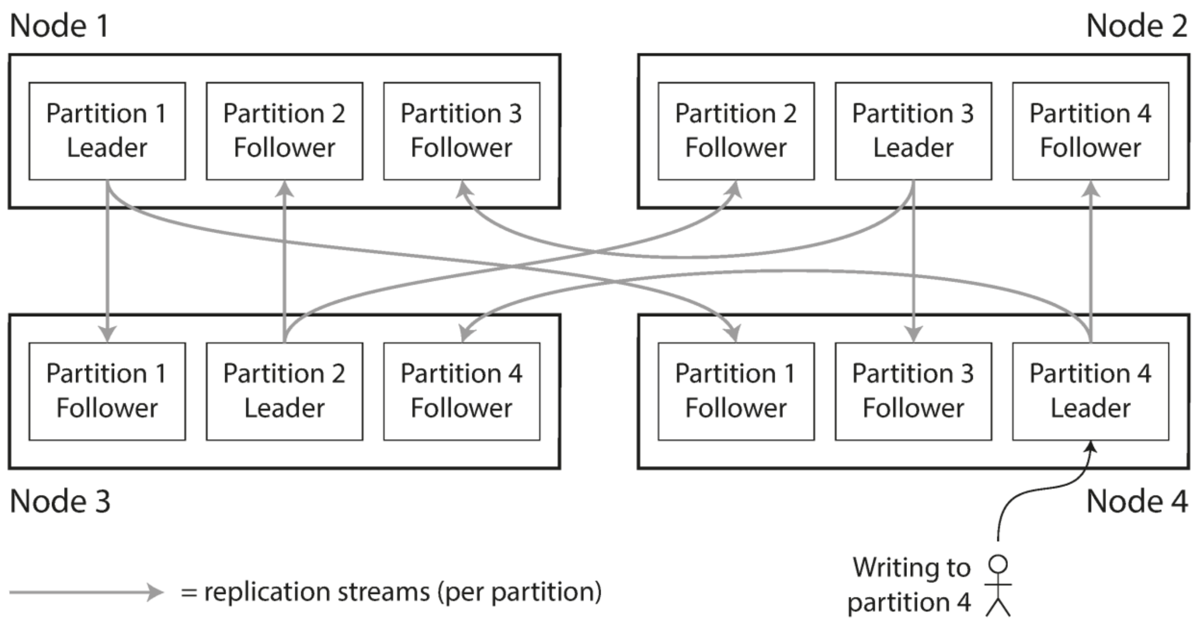
- **Consistency:** Clients always read the latest data
- **Availability:** Services are available for clients to access
- **Partition tolerance:** System continues to operate despite network failures
- **CAP theorem**: Choose two out of three
  - e.g., if a network failure occurs (and system tolerates it), choose consistency or availability



- Demonstrates trade-offs between different qualities of scalable systems
  - But somewhat controversial; some people argue it as being misleading

Distributed Data: Partitioning

- **Partitioning** (also called *sharding*): Split the data into smaller, independent units & distribute them across nodes
  - Useful and necessary when one dataset is too large to be fit onto a single node (i.e., replication alone is not sufficient!)
  - Usually combined with replication: Each partition is replicated stored across multiple nodes

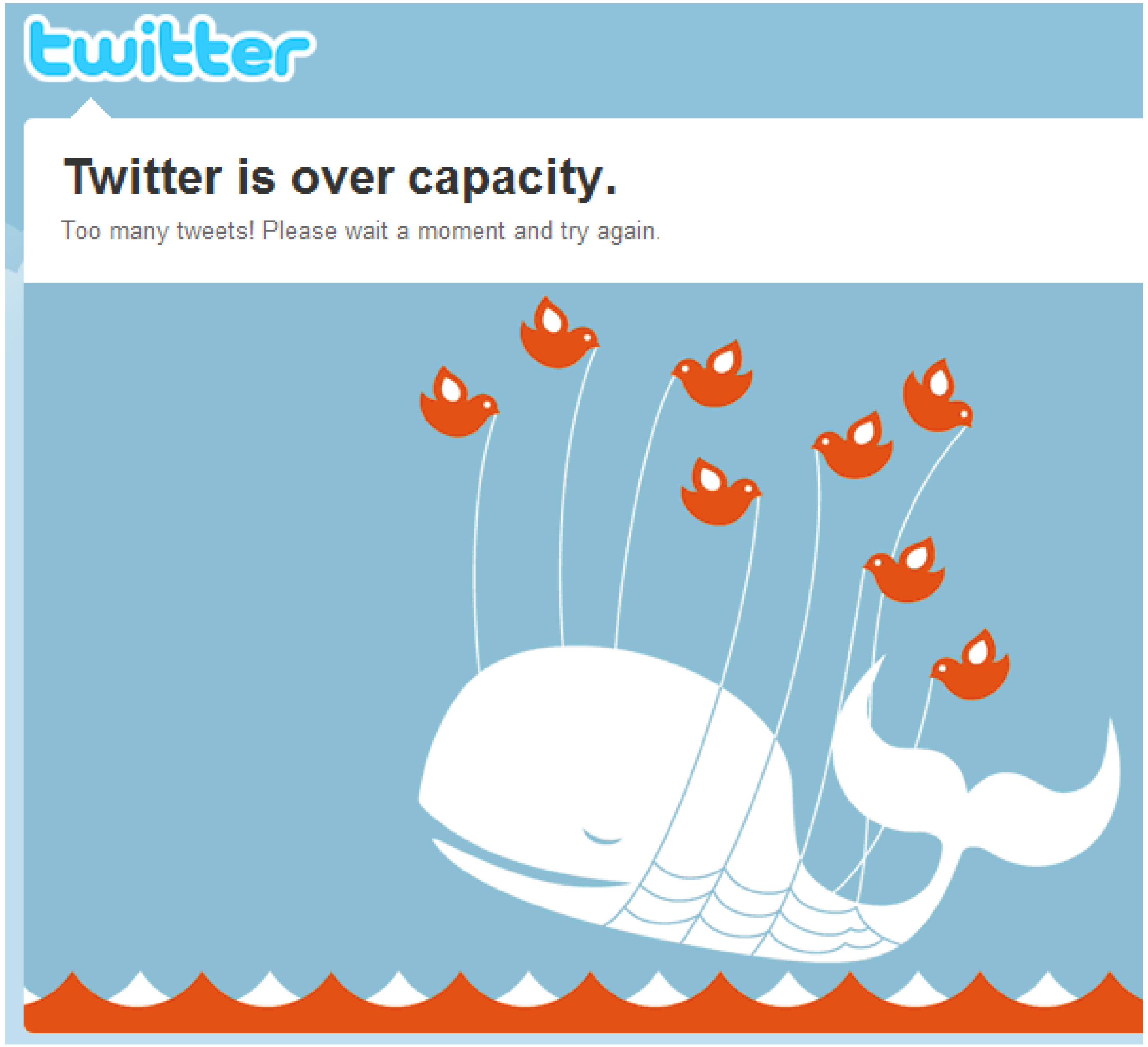


Distributed Data: Partitioning

- **Partitioning** (also called *sharding*): Split the data into smaller, independent units & distribute them across nodes
  - Useful and necessary when one dataset is too large to be fit onto a single node (i.e., replication alone is not sufficient!)
  - Usually combined with replication: Each partition is replicated stored across multiple nodes
- Types of partitioning
  - **Horizontal**: Partition a series of data by rows
    - e.g., Split user IDs into ranges ([A-G], [H-N],…)
    - Partition based on geographical regions, timestamp, logical categories
  - **Vertical**: Partition a series of data by columns
    - e.g., Split user table into (profile info, user images)
    - Partition based on data access frequency, data size, data sensitivity

Prime Big Deal Days

Included with a Prime membership



Twitter is over capacity.

Too many tweets! Please wait a moment and try again.

Distributed Data: Partitioning

- **Partitioning** (also called *sharding*): Split the data into smaller, independent units & distribute them across nodes
  - Useful and necessary when one dataset is too large to be fit onto a single node (i.e., replication alone is not sufficient!)
  - Usually combined with replication: Each partition is replicated stored across multiple nodes
- **Other design considerations**
  - How to rebalance partitions (when new data is added over time)
  - How to route client requests to the right partition (e.g., Zookeeper)
  - More details in the assigned reading (Chapter 6, Kleppman)

Summary: Vertical vs. Horizontal Scaling

- Vertical and horizontal scaling are two major ways of adding capacity to a system
- Horizontal scaling typically involves distributing data across multiple nodes, to allow load to be divided among the machines
- Replication and partitioning are two common ways of distributing data
- Despite multiple benefits (performance, scalability, fault-tolerance), distributing data introduces new challenges into the design task
- Start with vertical scaling if possible! It’s simpler and more efficient

Exercise: Designing Movie Streaming Service

- Sketch a design of a movie streaming service (e.g., Netflix), focusing on two operations: **Browsing recommended movies** and **playing a movie**
- **Questions to discuss**:
  - What data do we need store for the operations?
  - What type of data model (relation vs. document) for which data?
  - What type of scaling (vertical, horizontal, or both) do we apply?
  - How do we distribute data (replication, partitioning, or both)?
  - If replication is used, synchronous vs. asynchronous replication?

Load Balancing

Common Design Problems for Scalability

- How do we increase capacity to handle additional load? Vertical & horizontal scaling
- **How do we avoid overloading one part of the system due to increased load? Load balancing**
- How do we reduce bottleneck in the overall workload? Caching

Load Balancing (LB)

- The process of distributing workload across multiple machines, to avoid overloading parts of the system

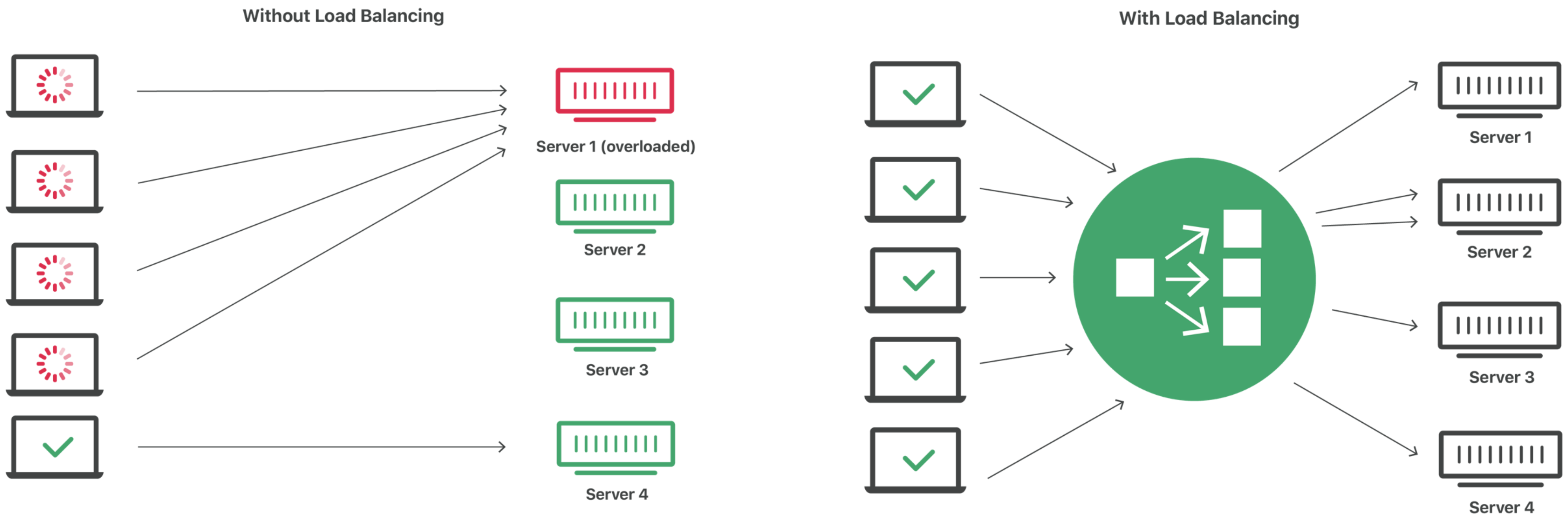


Image source: https://www.cloudflare.com/learning/performance/what-is-load-balancing/

Load Balancing (LB)

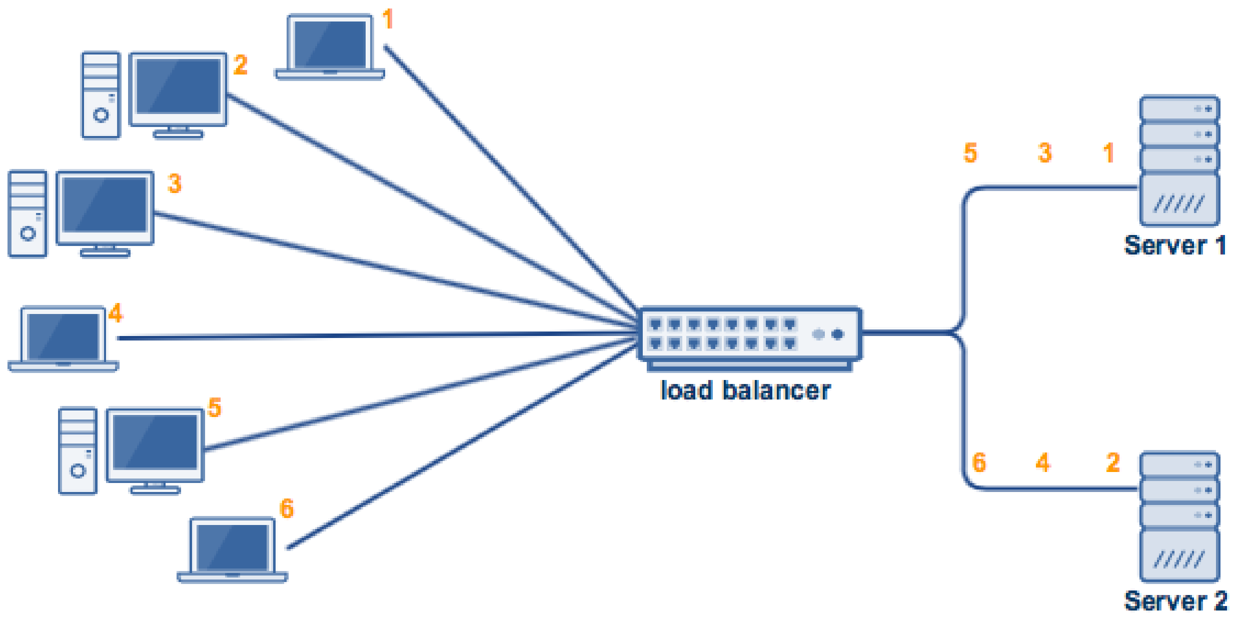
- The process of distributing workload across multiple machines, to avoid overloading parts of the system
- Benefits:
  - Scalability: Handle high workload by distributing them evenly
  - Availability: Run maintenance & upgrades without application downtime
  - Performance: Redirect client requests to a geographically closer node
  - Security: Monitor, identify, and block problematic traffic (e.g., denial-of-service attacks)

Types of LB Algorithms

- **Static LB methods**
  - Use fixed rules that are independent of the current state of the nodes
  - Easy to set up & can be made very efficient, but only if the actual workload matches the expected pattern
- **Dynamic LB methods**
  - Dynamically decide how to distribute traffic based on the current state
  - **State information**: For each node, amount of utilization, number of outstanding requests, average response time, etc.,
  - More complex to design & deploy, but also more robust to varying workloads

Static LB Algorithms

- Round-robin
  - Assign client requests to the nodes in the round-robin fashion
  - Q. Possible downside?



Static LB Algorithms

- **Round-robin**
  - Assign client requests to the nodes in the round-robin fashion
  - Downside: Disregards the capacity (e.g., processing power) differences between the nodes
- **Weighted round-robin**
  - Round-robin, but each node is also weighted based on its capacity; nodes receive amount of load in proportion to their weights
- **IP hash method**
  - Compute a hash of the client’s IP address & map requests to the node with the corresponding hash
  - Useful for ensuring consistent connection between a specific pair of client and machine

Dynamic LB Algorithms

- **Least connection**
  - Check the number of connections to the nodes & assign task to the least busy nodes (can also be weighted based on their capacity)
- **Average response time**
  - Monitor the average respond time (RT) for each node & assign task to the ones with the fastest RT
- **Resource-based**
  - Measure available CPU & memory on each node & assign task to the ones with the most available resources
- A hybrid of one or more of these

(Animation of LB methods)

Exercise: Which LB algorithm?

- For each of the following applications:
    - What are unique characteristics of the client request/load pattern?
    - Which LB to use: Round-robin, IP hash, or least connection?

1. News media site (e.g., NYTimes)
2. Video streaming app (Netflix)
3. Social media app (Tiktok)

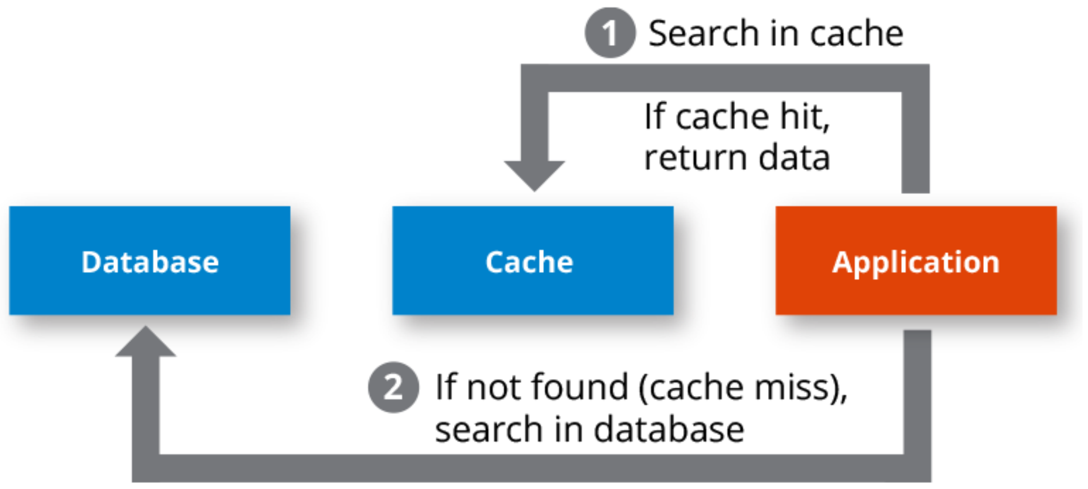
Caching

Common Design Problems for Scalability

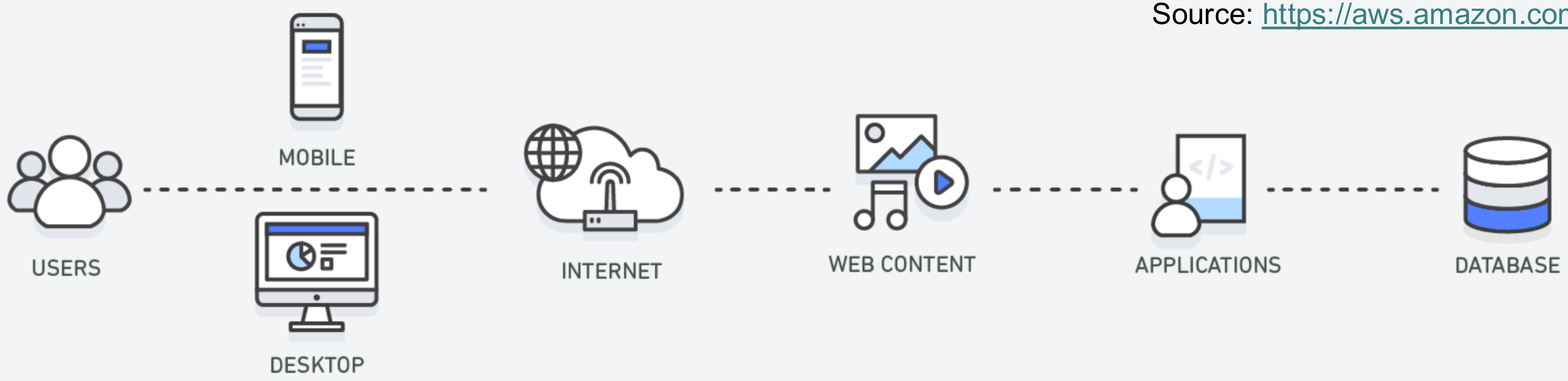
- How do we increase capacity to handle additional load? Vertical & horizontal scaling
- How do we avoid overloading one part of the system due to increased load? Load balancing
- **How do we reduce bottleneck in the overall workload? Caching**

Caching

- Store and serve a subset of data in a special storage area (e.g., RAM) that enables faster access
- Improve application performance & reduce the load on the backend
- Can be applied at different layers and locations within a system: Client-side, network, server-side, database, hardware, etc.,


1 Search in cache
If cache hit, return data
Database
Cache
Application
2 If not found (cache miss), search in database

Source: https://aws.amazon.com/caching/



Layer	Client-Side	DNS	Web	App	Database
Use Case	Accelerate retrieval of web content from websites (browser or device)	Domain to IP Resolution	Accelerate retrieval of web content from web/app servers. Manage Web Sessions (server side)	Accelerate application performance and data access	Reduce latency associated with database query requests
Technologies	HTTP Cache Headers, Browsers	DNS Servers	HTTP Cache Headers, CDNs, Reverse Proxies, Web Accelerators, Key/Value Stores	Key/Value data stores, Local caches	Database buffers, Key/Value data stores
Solutions	Browser Specific	Amazon Route 53	Amazon CloudFront, ElastiCache for Redis, ElastiCache for Memcached, Partner Solutions	Application Frameworks, ElastiCache for Redis, ElastiCache for Memcached, Partner Solutions	ElastiCache for Redis, ElastiCache for Memcached

Application-layer Cache

- **Design decision**: Which data do we serve through cache?
- Monitor & identify bottleneck in the workload
- **Determine**: What are some frequently requested or displayed data?

Prime Big Deal Days

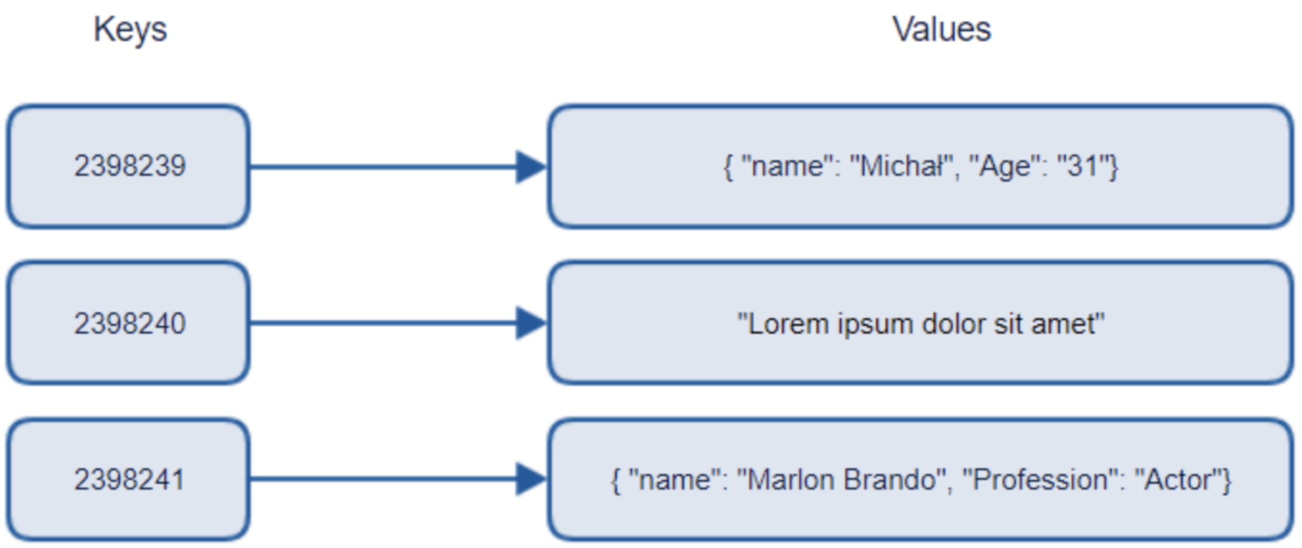
Included with a Prime membership

Application-layer Cache

- **Design decision**: Which data do we serve through cache?
- Monitor & identify bottleneck in the workload
- Determine: What are some frequently requested or displayed data?
- **Example: E-commerce site**
  - **Site-wide data**: Top selling products, promotions, pre-rendered HTML (for home page)
  - **User-specific data**: Recommended products, shopping cart status, recent order history
  - Computations based inventory analysis: Stock availability, price trends
- **Benefit**: Avoid redoing computation (e.g., database queries) and reduce response time

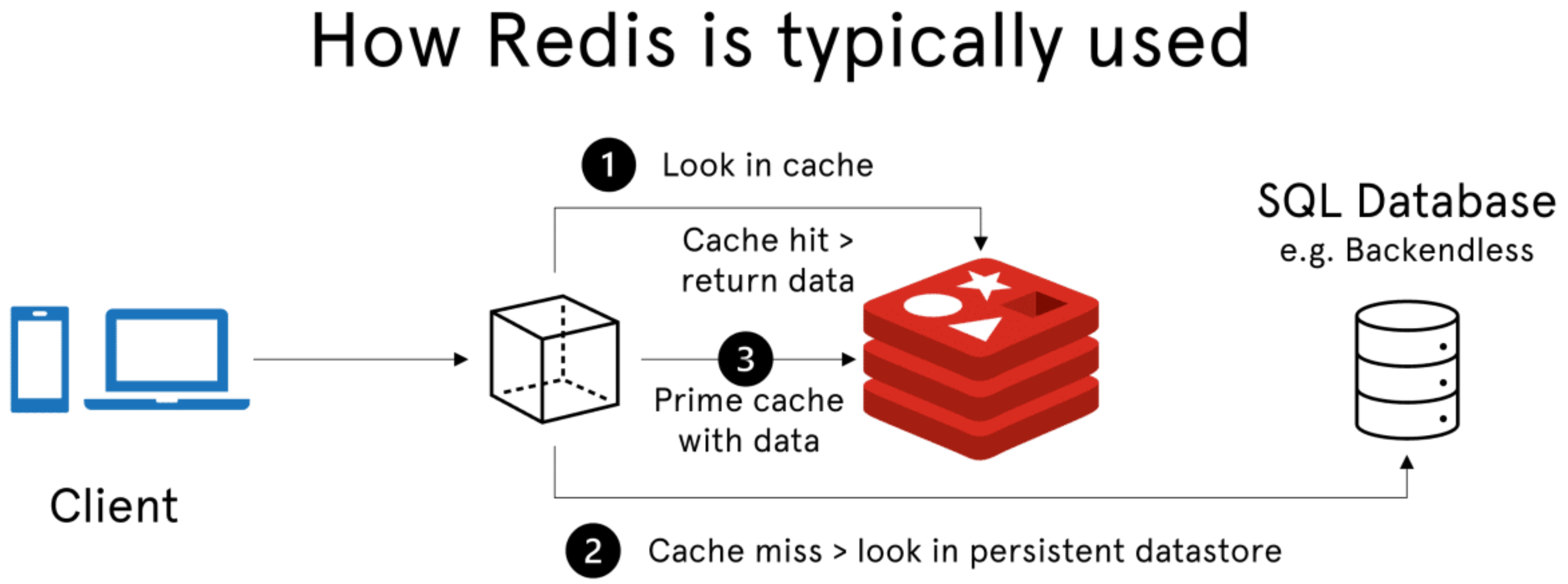
Key-Value Store

- Mapping from a key value (e.g., hash of a request) to a data object
- Simple structure (recall: document data model) & fast lookup; used to serve frequently accessed data
- Typically stored in-memory to optimize access time (e.g., Redis, memcached)


Keys
Values
2398239
{ "name": "Michał", "Age": "31"}
2398240
"Lorem ipsum dolor sit amet"
2398241
{ "name": "Marlon Brando", "Profession": "Actor"}

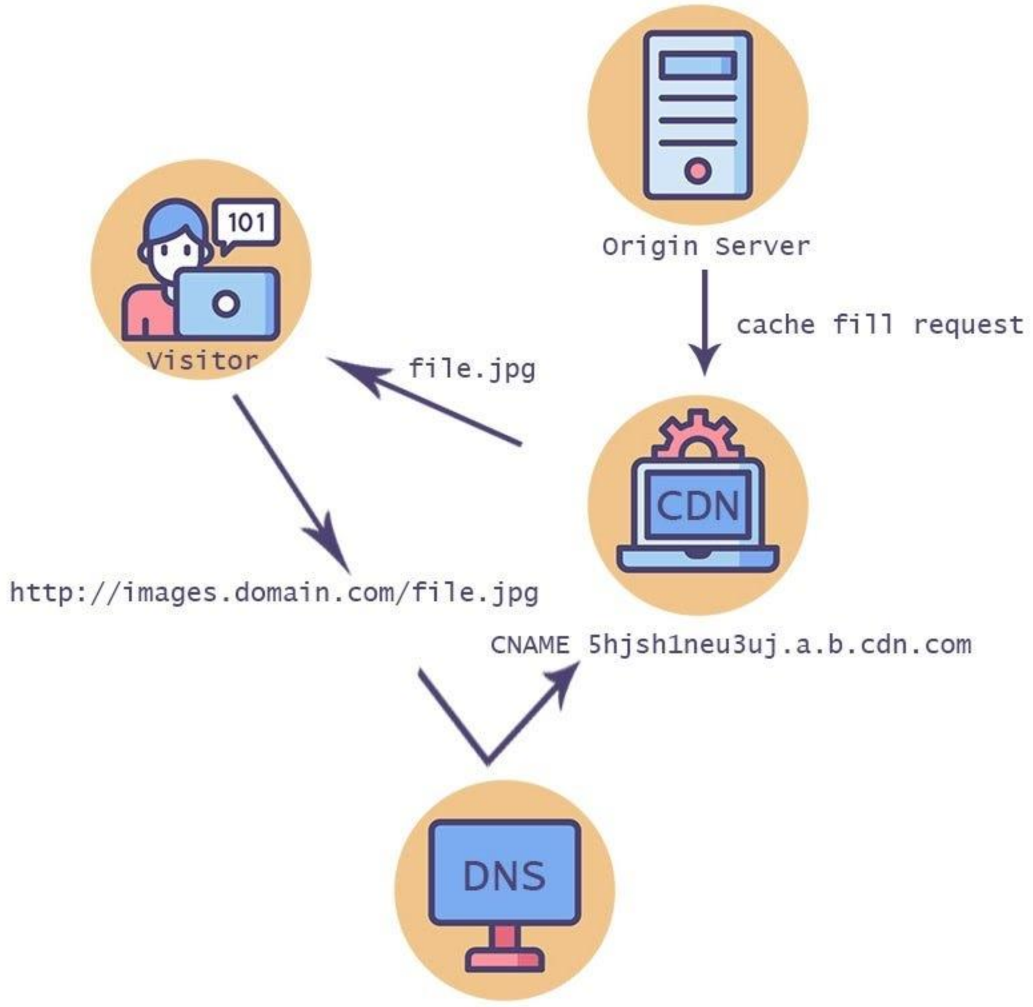
Key-Value Store

- Mapping from a key value (e.g., hash of a request) to a data object
- Simple structure (recall: document data model) & fast lookup; used to serve frequently accessed data
- Typically stored in-memory to optimize access time (e.g., Redis, memcached)



Content Delivery Network (CDN)

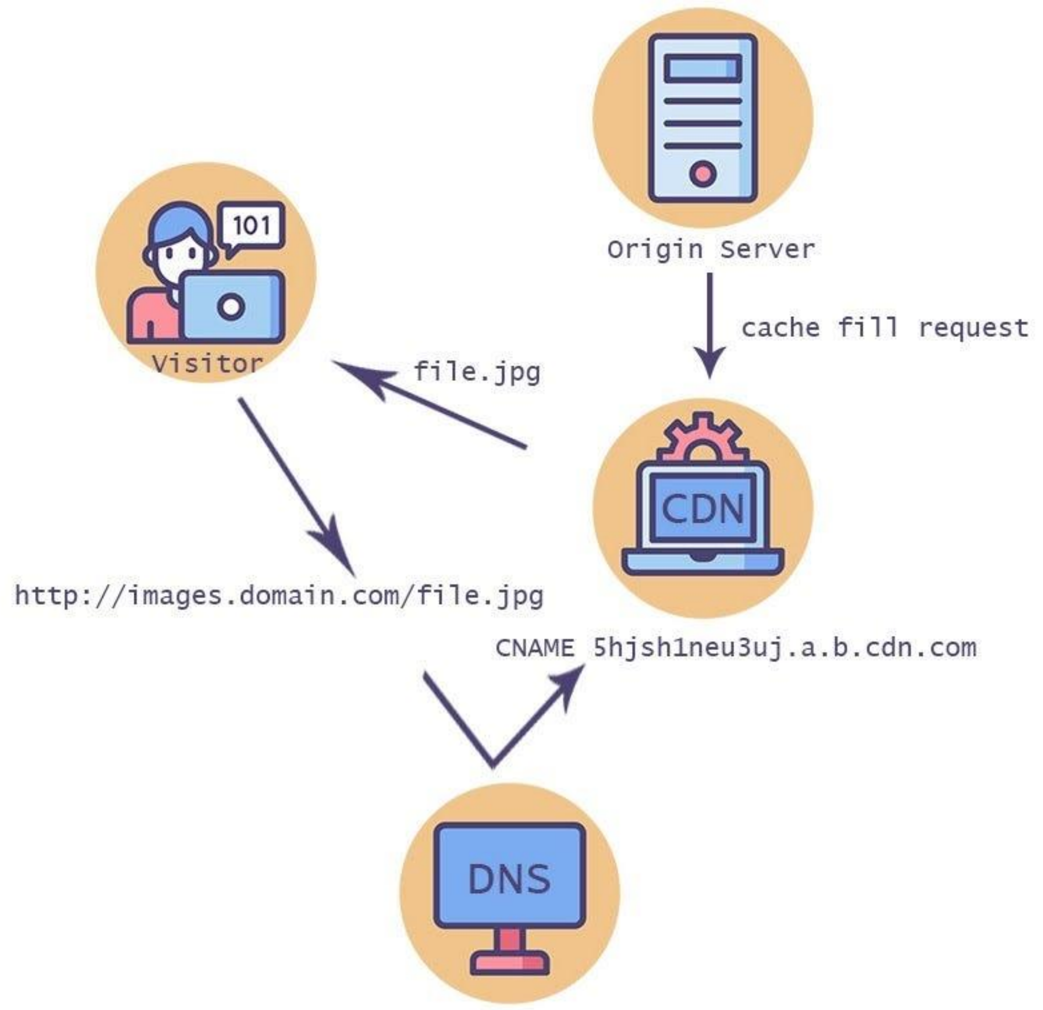
- A set of network nodes distributed across geographical locations
- A third-party service that allows an application to deliver a cache of data (e.g., videos, webpages, images, etc.,) to its users
- Application uploads data to be served by a CDN provider
- The provider handles delivery of content to customers from nearby nodes


Origin Server
cache fill request
101
Visitor
file.jpg
CDN
http://images.domain.com/file.jpg
CNAME 5hjsh1neu3uj.a.b.cdn.com
DNS

Source: How the Cloud and CDN Architecture Works for Netflix

Content Delivery Network (CDN)

- Benefits
  - Improves performance and scalability, without need to build own infrastructure
  - Global reach; improved search engine rankings
- Drawbacks
  - Costly (> $0.10 per GB)
  - Data stored on third-party nodes; potential privacy & security issues
  - Dependency on another network; additional point of failure


Origin Server
cache fill request
101
Visitor
file.jpg
CDN
http://images.domain.com/file.jpg
CNAME 5hjsh1neu3uj.a.b.cdn.com
DNS

Source: How the Cloud and CDN Architecture Works for Netflix

BUSINESS

Major Internet outage along East Coast causes large parts of the Web to crash — again

By Timothy Bella

July 22, 2021 at 2:02 p.m. EDT

“…the websites of UPS, USAA, Home Depot, HBO Max and Costco were also among those affected. The websites of British Airways, GoDaddy, Fidelity, Vanguard and AT&T were among those loading slowly.

The cause of the outage, the latest major Internet outage this summer, was linked to Akamai Technologies, the global content delivery network based in Cambridge, Mass.”

Exercise: Designing Movie Streaming Service

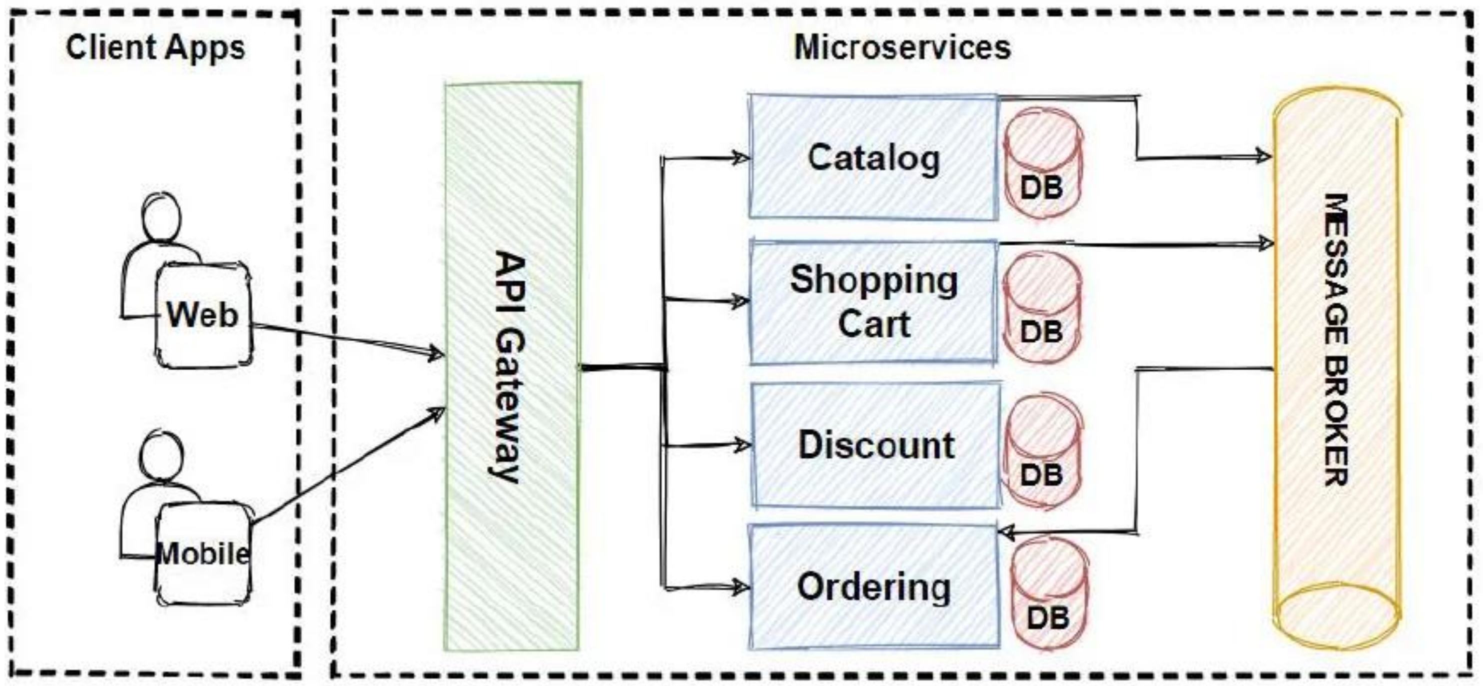
- Sketch a design of a movie streaming service (e.g., Netflix), focusing on two operations: **Browsing recommended movies** and **playing a movie**
- **Questions to discuss**:
  - What data do we need store for the operations?
  - What type of data model (relation vs. document) for which data?
  - What type of scaling (vertical, horizontal, or both) do we apply?
  - How do we distribute data (replication, partitioning, or both)?
  - If replication is used, synchronous vs. asynchronous replication?
  - **Which client requests/results/data do we cache?**

Summary: Load Balancing & Caching

- Load balancing: Avoid overloading by distributing workload across nodes
- Caching: Serve frequently accessed data from a special storage for faster delivery
- Essential part of most modern large-scale systems!
- You probably won’t need to (and shouldn’t) implement your own LB or caching solutions; many web/DB frameworks support these features
- But application-specific decisions about what data to cache & where to place load balancer are important!

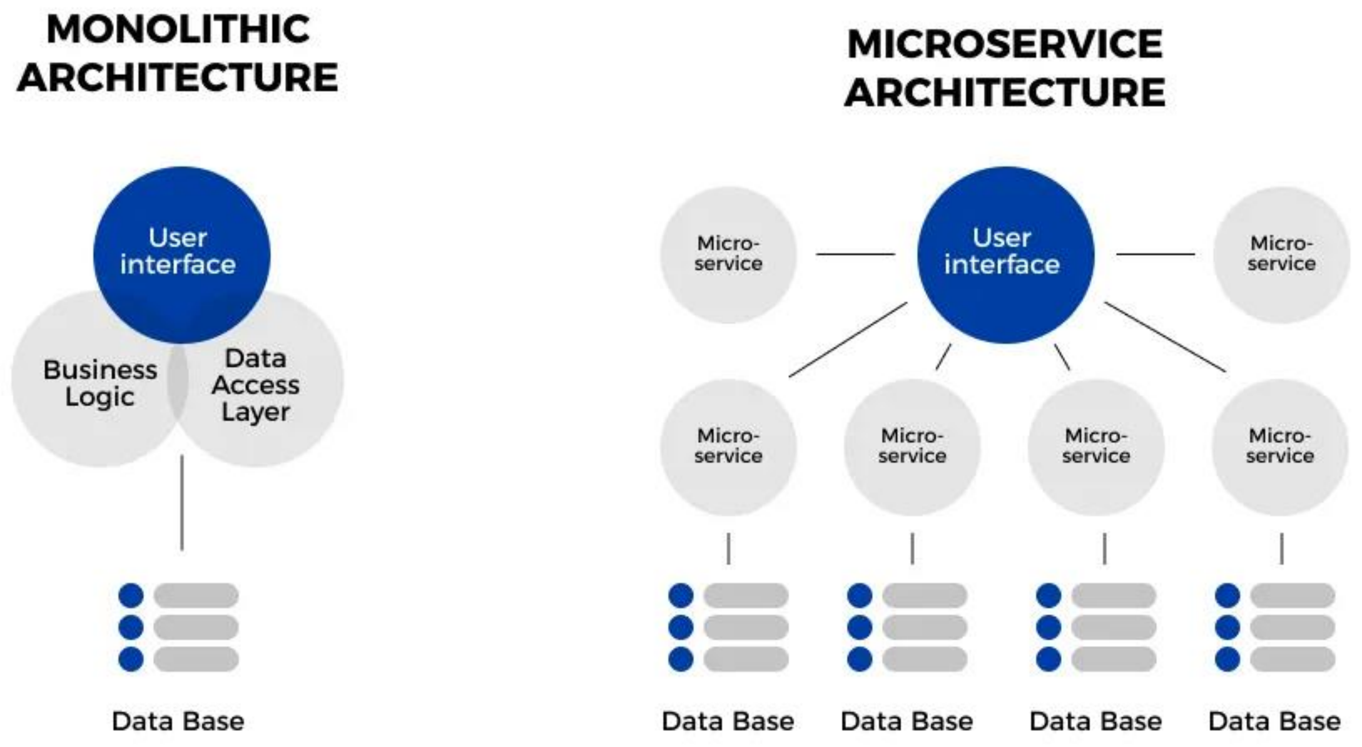
Microservices & Scalability

Microservice Architecture



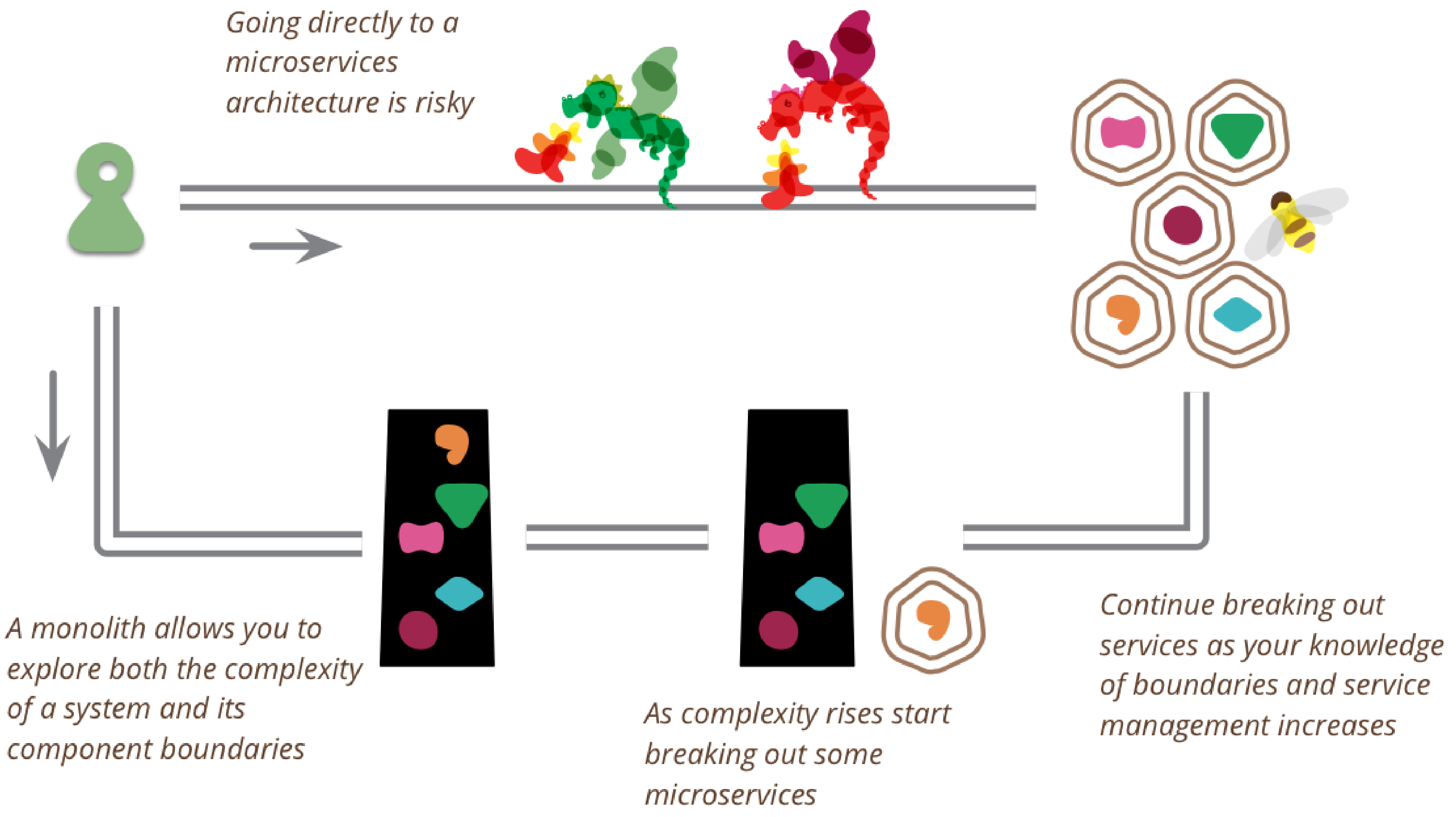
- Decompose system into multiple, deployable units of services, typically developed by independent teams
- User requests are routed to the appropriate service
- Services communicate directly or through a message broker

Microservice Architecture



- Q. What are the benefits of a microservice architecture **with respect to scalability**?
- Easier to scale a specific service(s) instead of the entire system

Recall: “Monolith First”


Going directly to a microservices architecture is risky
A monolith allows you to explore both the complexity of a system and its component boundaries
As complexity rises start breaking out some microservices
Continue breaking out services as your knowledge of boundaries and service management increases

https://martinfowler.com/bliki/MonolithFirst.html#footnote-typical-monolith

Design for Scalability: Closing Thoughts

- No generic, one-size-fits-all scalable design (“magic scaling” sauce).
- Many factors to consider: Volume of reads & writes, complexity of data to store, response time requirements, access patterns, or some mix of these
  - Handling 100,000 requests per second, each 1 kB in size vs. 3 requests per minute, each 2 GB in size – same data throughput, but very different design!
- An architecture that scales well for a particular application is built around assumptions about load patterns (e.g., which operations will be common and which will be rare, which data will be frequently accessed, etc.,)
  - If those assumptions turn out to be wrong, the engineering effort for scaling is at best wasted, and at worst counterproductive.
- In an early-stage startup or an unproven product, it’s more important to iterate quickly on product than to scale to some hypothetical future load!

Summary

- Exit ticket!