COMPSCI4106/5118 Cloud Systems

University of Glasgow

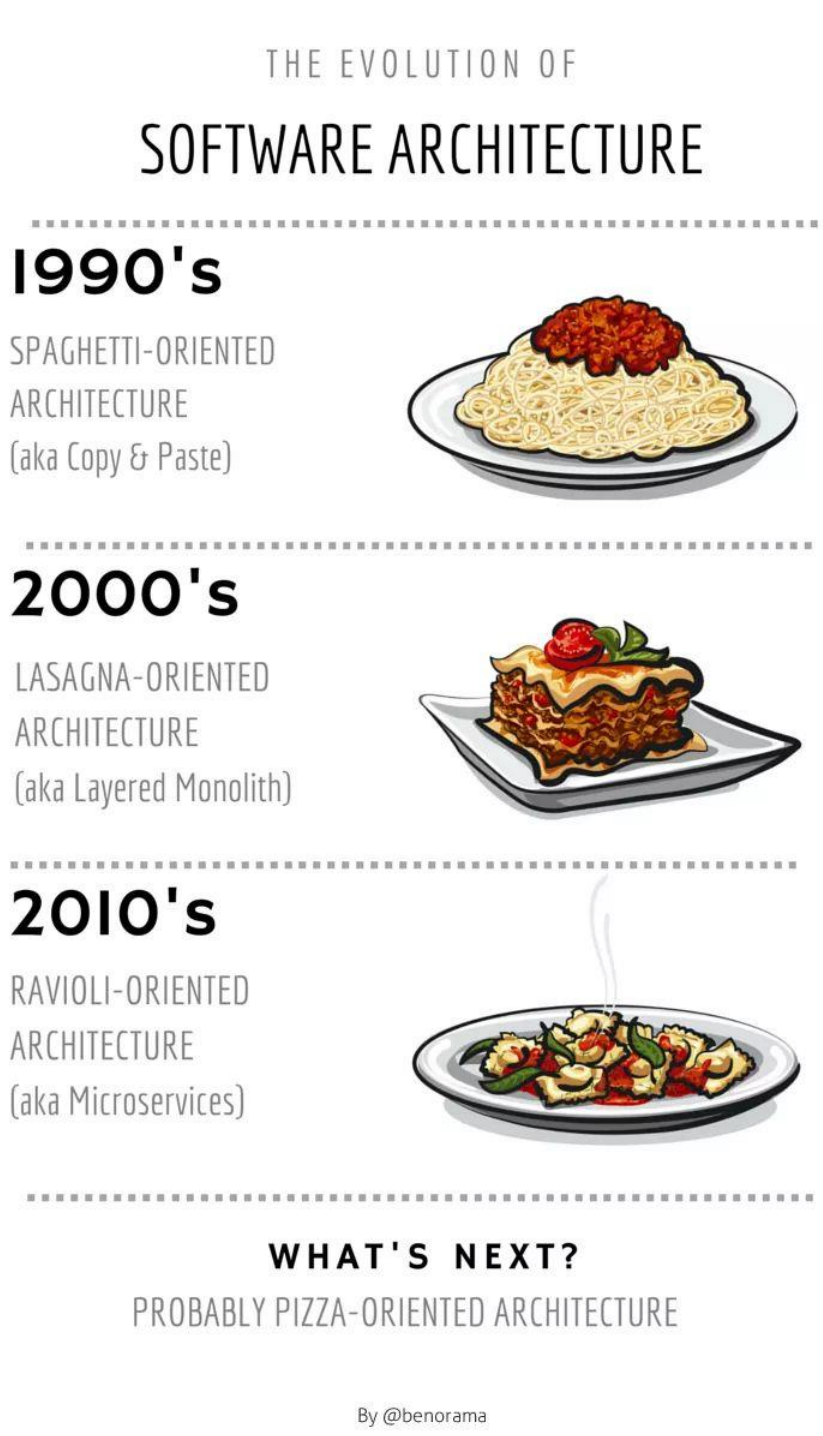
Intro to Cloud Computing and Virtual Resources
Virtual Machines
Containers and Container Management
Cloud System Operation and Management
Cloud System Sustainability
Cloud System Design
▶ **Modern Cloud Architectures**
Flavours of Cloud
A Wider Lens on Sustainability
Emerging Topics and Research Trends

Yehia Elkhatib

Overview

- Architectural Approaches
  - Redundancy by Replication
  - Cloud Scaling
- Advanced Architectures & Technologies
  - Microservices
  - Service Mesh
  - Cloud-native Technologies
- Designing Dependable Data Centres
  - Hardware, Networks, Power, Cooling

Architectural Approaches



reddit.com/user/yehiaserag

Architecture?

- Foundation Matters: just as buildings need to support their structure, cloud architectures require robust infrastructure layers (compute, storage, networking) to support higher-level services
- Scalability & Modularity: modular construction techniques allow for easier expansion and modification
- Resource Efficiency: through auto-scaling, serverless, and efficient resource allocation
- Evolution: cloud architectures must evolve to accommodate new technologies and requirements while ensuring system stability

Redundancy by Replication

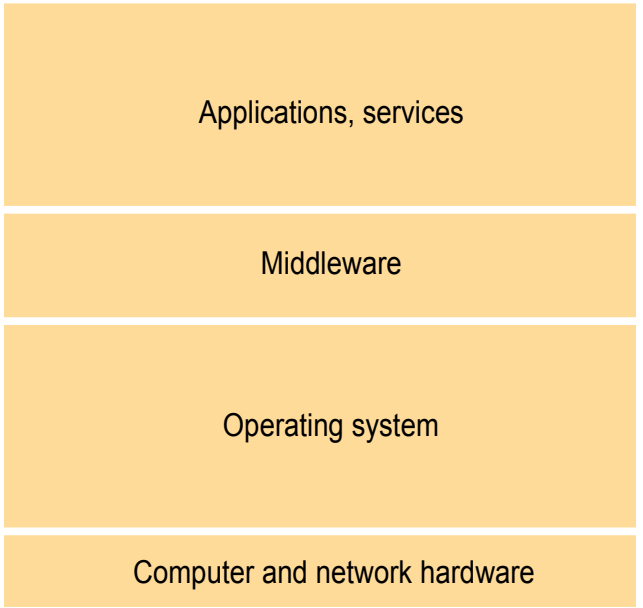
- Why replicate?
  - Performance, e.g. of heavily loaded servers
  - Allows error detection when replicas disagree
  - Allows backward error recovery
- Importance of **fault-model**
  - Effect of replication depends on *how* individual replicas fail
  - Crash faults → availability = $1\text{-}p^n$
    where n = no. of replicas, p = probability of individual failure
  - E.g. 5 servers each with 90% uptime → availability = $1\text{-}(0.10)^5$ = 99.999%

Architectural Concepts – Monolith

- A single, tightly coupled block of code with all application components
- Advantages:
  - Simple to develop and deploy initially
  - Easy to test and debug in the early stages
- Disadvantages:
  - Complexity – Becomes very difficult to maintain application growth / evolution
    - Changes to one part could affect the entire system
  - Scaling challenges – Difficult to scale individual components independently
  - Limited agility – Slow and risky deployments due to the large codebase
  - Technology lock-in – Difficult to adopt new technologies or frameworks

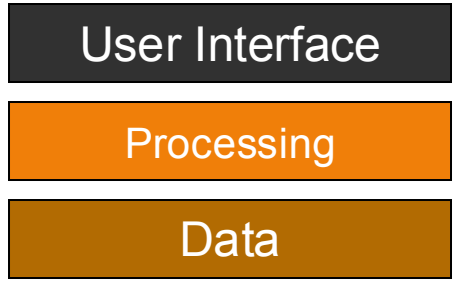
Architectural Concepts – Layering

- It is common to partition services vertically into layers
  - Lower layers provide services to higher ones
  - Higher layers being unaware of the underlying implementation details
  - Low inter-layer dependency
- Examples:
  - Network protocol stacks, e.g. OSI model
  - Operating systems – kernel, drivers, libraries, GUI, …
  - Games – engine, logic, AI, UI



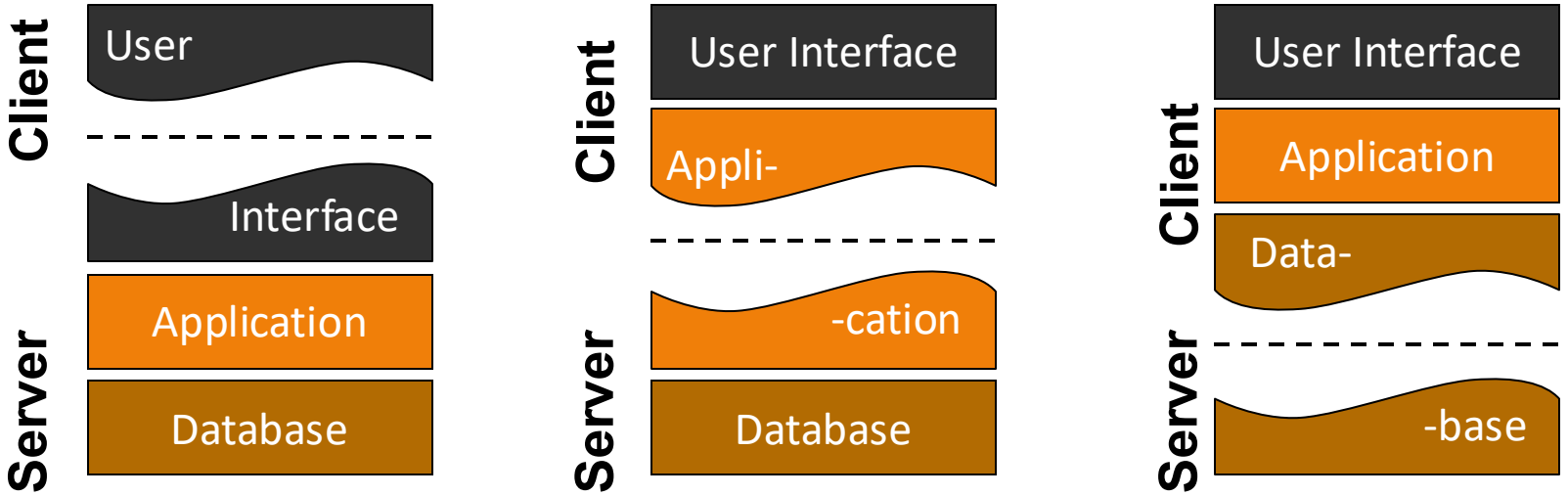
Architectural Concepts – Layering

- Distributed applications commonly feature 3 layers
  - User interface / Presentation: Maintaining interaction between user & system
  - Processing: Often referred to as the business logic of the application
  - Data: Manage persistent state; can comprise Access & Management sublayers
- Advantages:
  - Abstraction
  - Reusability
  - Loose coupling
    - isolated management
    - independent testing
    - software evolution



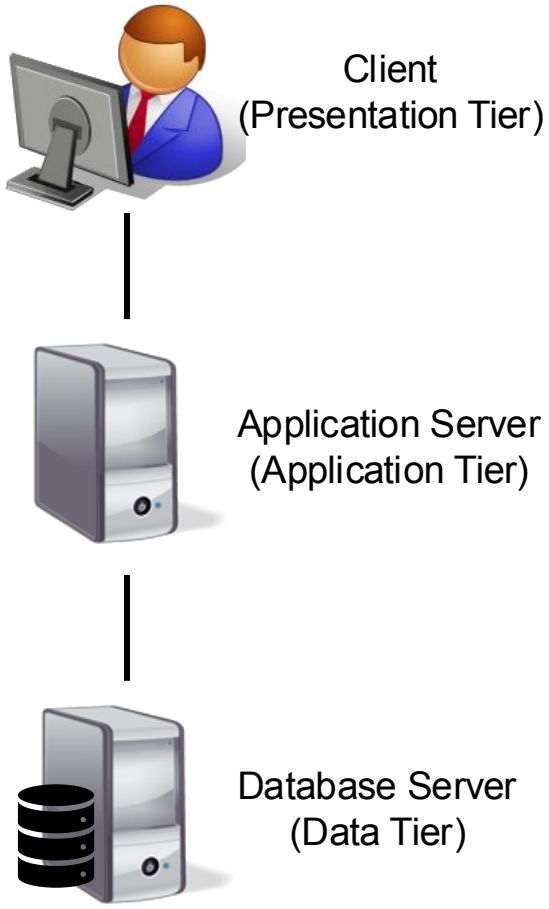
Architectural Concepts – Tiering

- Tiering complements layering by mapping the organisation *of and within* a layer to **devices**
  - implication about physical location
- Classic 2-tier architecture – *client-server* model
  - split layers between a client and a server in various ways:



Architectural Concepts – Tiering

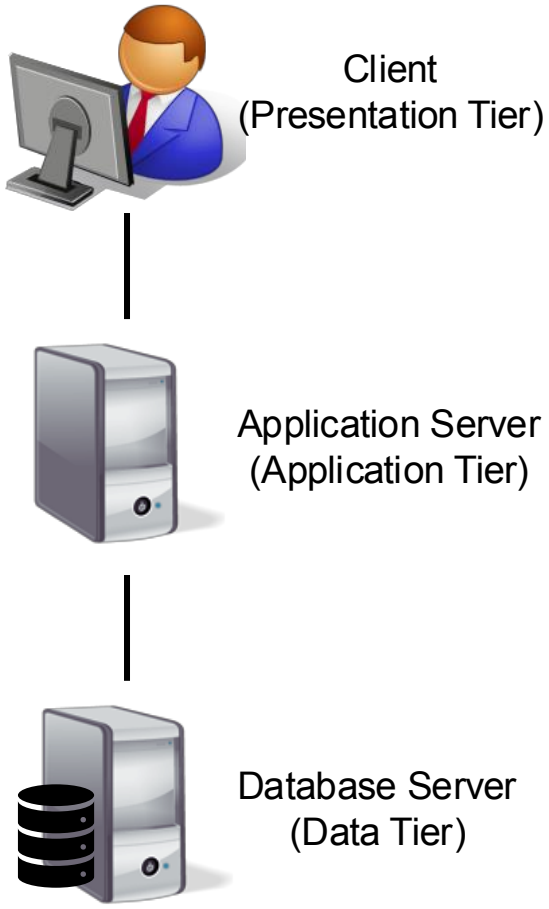
- 3-tiered architecture:
  - Same layers mapped on to three tiers
- 4-tiers … n-tier (or multi-tier) approaches
  - e.g. microservices
- Advantages:
  - Scalability
  - Availability
  - Flexibility
  - Easy management



How to increase availability?

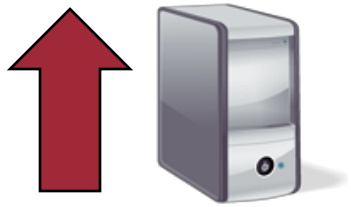
- Classic 3-tier architecture
  - What happens if we increase the number of clients?
  - What happens if one of the components goes down?

- The "cloud approach" – *Scaling*
  - Adjusting the amount of allocated resources
  - When a tier experiences change in load
    - ⬆ scale up if the load is increasing
    - ⬇ scale down is the load is decreasing



Cloud Scaling

- ✓ Error detection
- ✓ Failover (Error recovery)
- ▪ Load balancing
- ▪ Redundancy / Replication
- ○ Auto-scaling



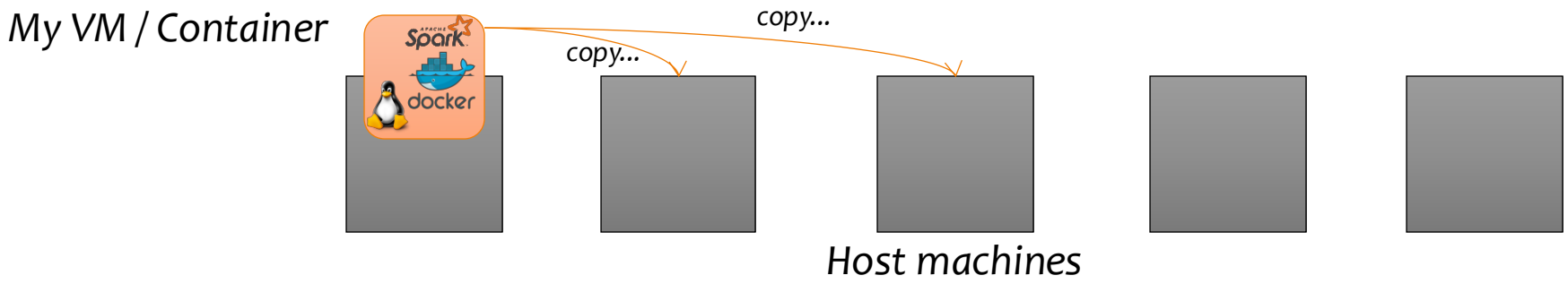
Vertical Scalability (Scaling Up)	
Idea	Increase performance of a single node (more cores, memory, …)
Pro	Good speedup up to a particular point; No change to appl. arch.
Con	Beyond that point, speedup becomes very expensive

Horizontal Scalability (Scaling Out)	
Idea	Increase number of nodes
Pro	Cheap to grow total amount of resources
Con	More difficult to efficiently utilize the resources; Need coord. sys.

Cloud Scaling – Out more than Up

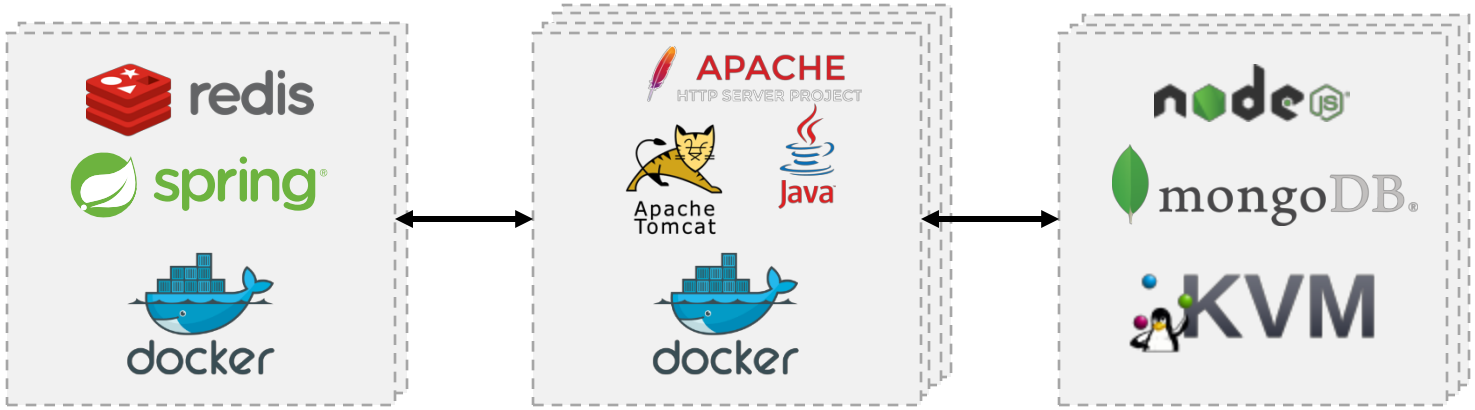
- ✓ Error detection
- ✓ Failover (Error recovery)
- ▪ Load balancing
- ▪ Redundancy / Replication
- ○ Auto-scaling

- Hardware trend: CPUs are not getting faster as fast as they used to
  → Datacentres today house large sets of inexpensive commodity servers (that fail eventually)
- Scaling applications typically means scaling-out across many nodes
  - Datacentres offer virtually unlimited scale-out services
- VMs and containers make it easy to replicate services across many nodes, especially with IaC and automation / orchestration tools



Cloud Scaling – Modern Cloud Architecture

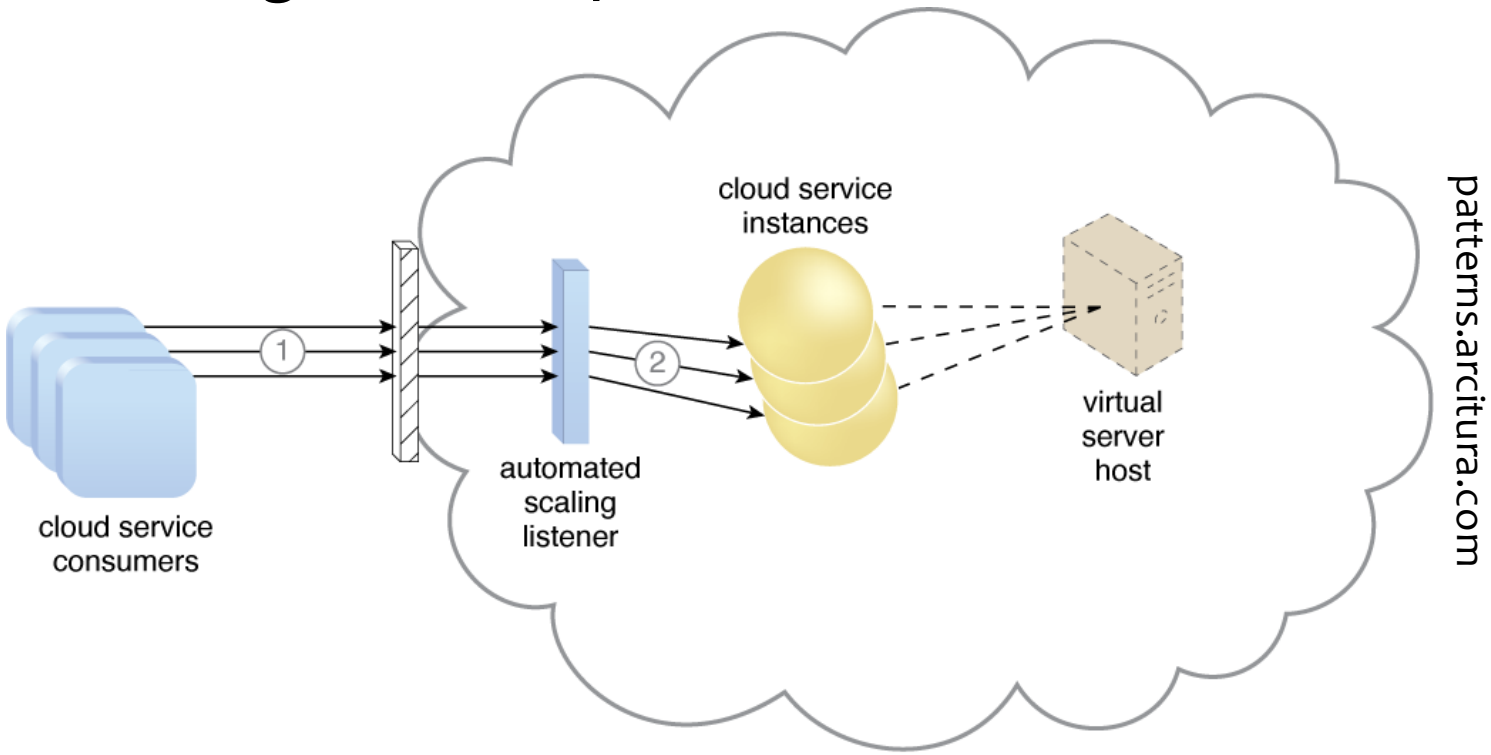
- Today's computer systems are built upon 2 architectural pillars:
  - vertical integration – doing more with each individual tier (service)
  - horizontal scaling – using many commodity computers working together



- RIP monolith applications
- Exploiting this created big data applications
- Expanding this created the microservice architecture

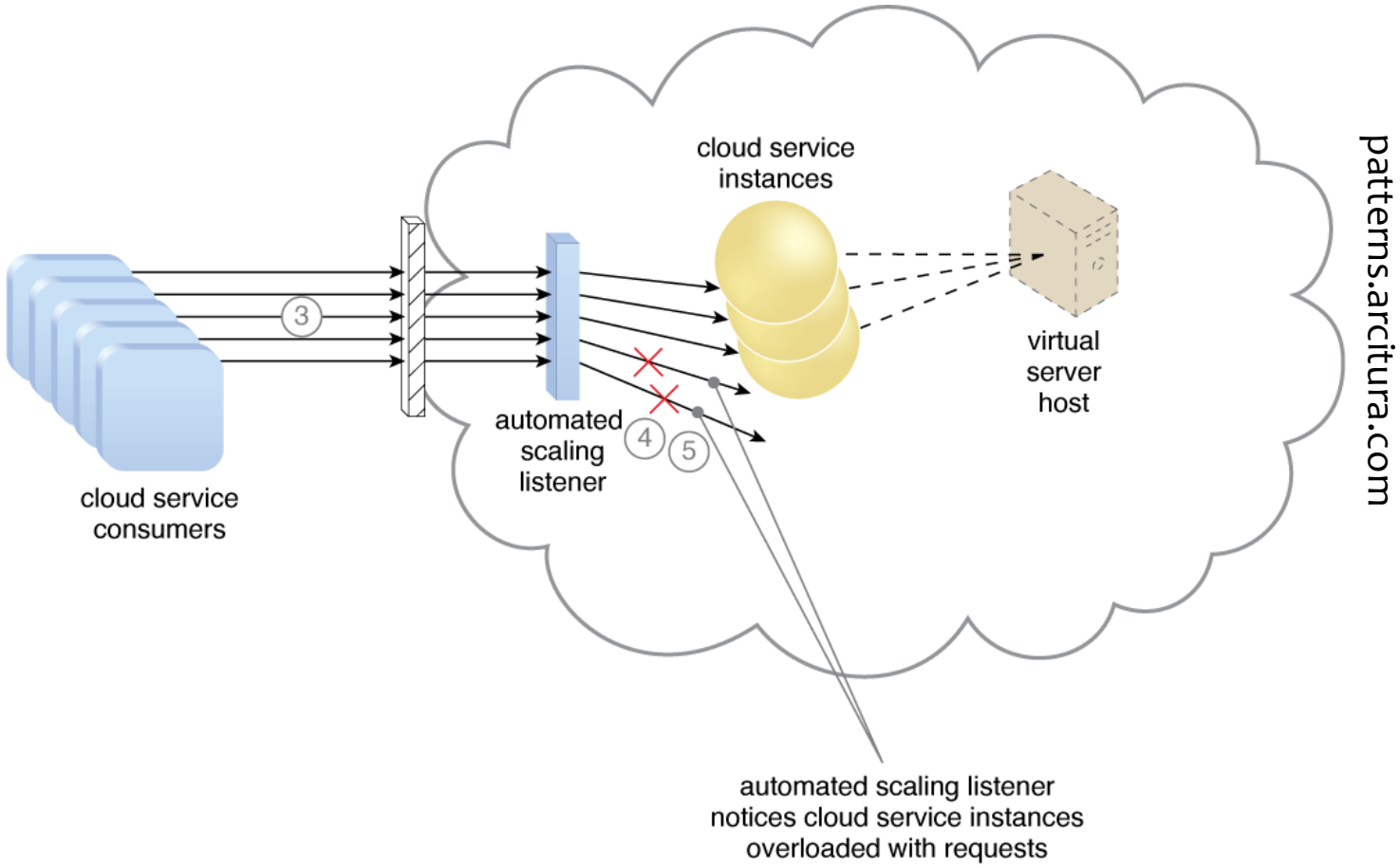
Simple Dynamic Scaling Architecture

- Consumers sending more requests



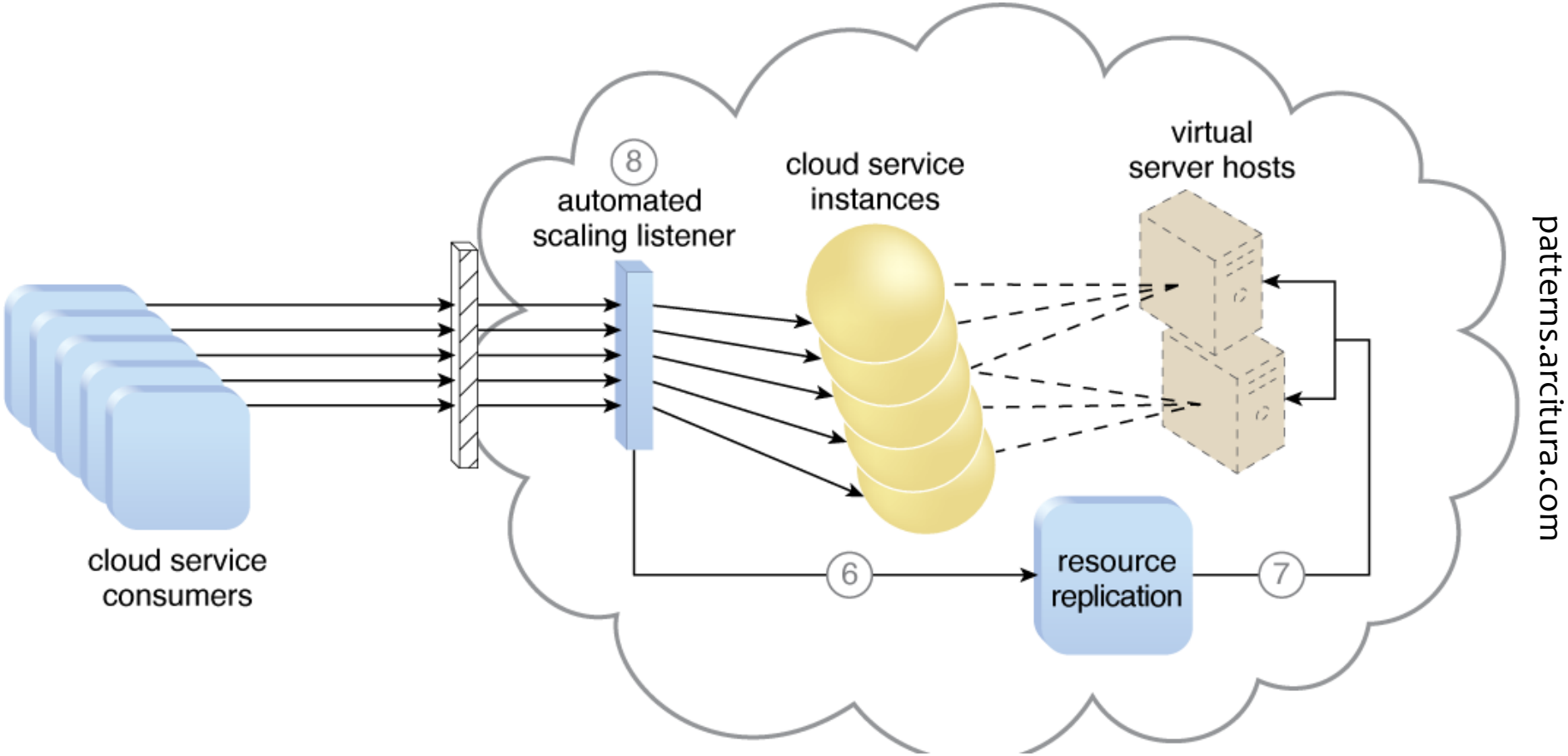
Simple Dynamic Scaling Architecture

- Requests overload the existing resources, timeouts exceeded



Simple Dynamic Scaling Architecture

- Scaling up by deploying additional resources



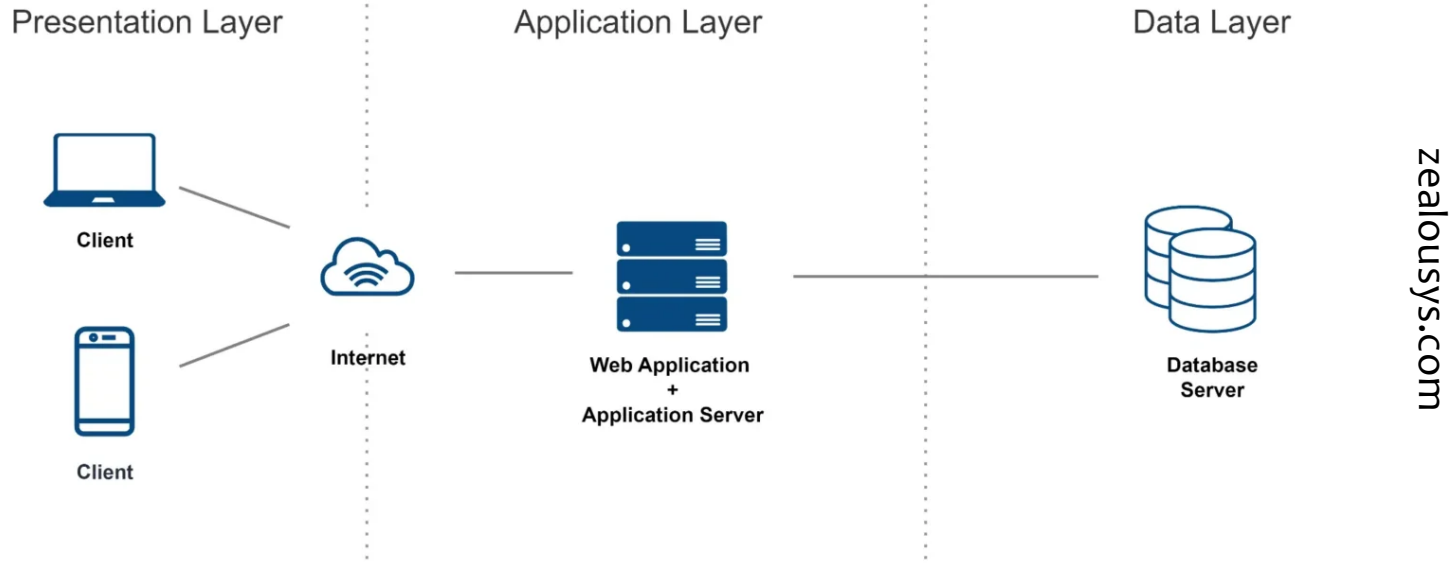
Scaling and State

- Multi-tier application scalability entails two questions
  - Where is the bottleneck in the architecture?
  - Is the bottleneck component *stateless* or *stateful*?
- Stateless components
  - Maintain no internal state beyond a request
  - Examples: DNS server, web server with static data, mathematical calc., …
- Stateful components
  - Maintain state beyond request
    - i.e. prior state is required to process future requests
  - Examples: mail server, firewalls, stateful web server, DBMS, …

Scaling Stateless Components

- ✓ Error detection
- ✓ Failover (Error recovery)
- ○ Load balancing
- ▪ Redundancy / Replication
- ✓ Auto-scaling

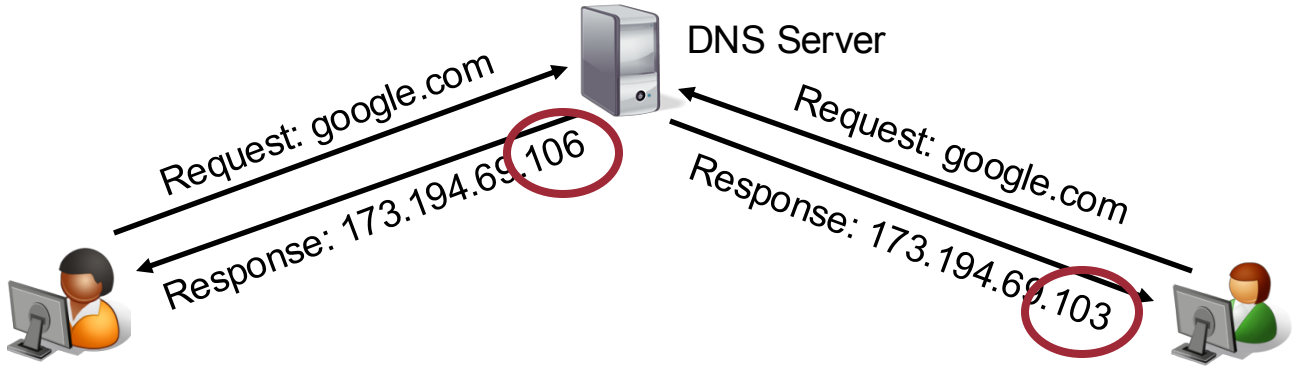
- Let's assume a bottleneck is a stateless service
  → Create more instances of said service



- How do clients figure out which server to contact?

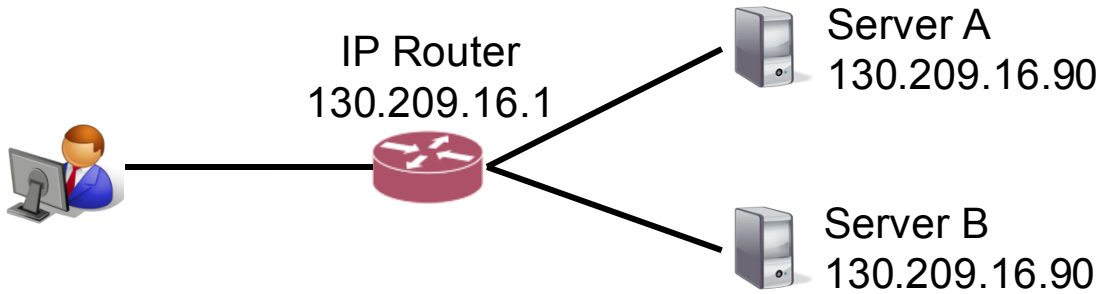
Stateless Load Balancing – DNS Level

- Balancing is implemented by DNS servers
  - DNS servers resolve domain names to IP addresses
  - Can use geo. location for also complying with sovereignty regs. (GeoDNS)
- Advantages:
  - Simple to implement
  - Cost-effective
  - Can handle high traffic volumes
- Disadvantages:
  - Less control over traffic distribution
  - Can be slow to react to server failures due to DNS caching
  - Limited health check capabilities



Stateless Load Balancing – IP Level

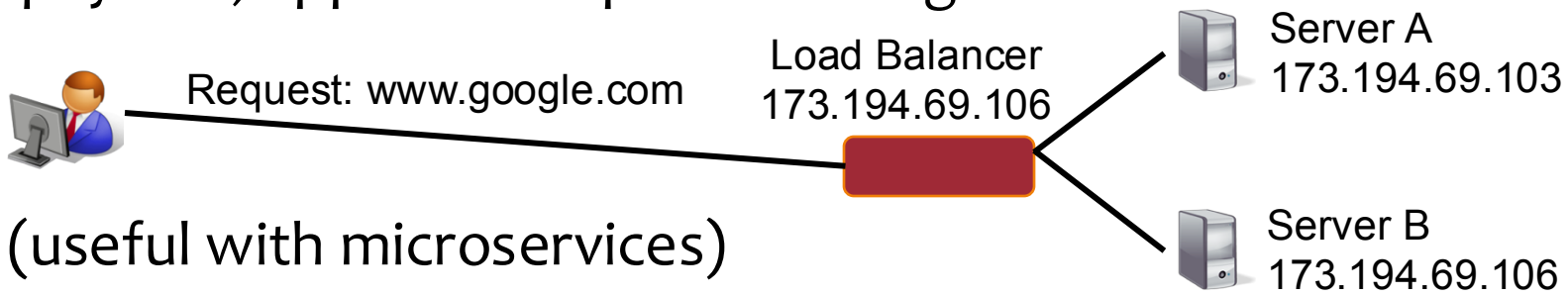
- Balancing is implemented by routers at the network layer
  - Multiple devices share one IP address (IP anycast)
  - Routers direct clients to different locations (round robin, hashing algorithms)
- Advantages:
  - Relatively simple
  - Fast response to server failures
  - Basic health checks and monitoring



- Disadvantages:
  - Less granular; assumes all requests create equal load
  - More expensive than DNS load balancing (routing configuration)
  - Does not handle application-layer optimisations (e.g. SSL termination)

Stateless Load Balancing – Application Level

- Bespoke load balancer acting as a temporary front end
  - Distributes requests among machines based on various features
  - HTTP headers, cookies, payload, application-specific configuration
- Advantages:
  - Granular control
  - Content-based routing (useful with microservices)
  - Can reduce load on servers (e.g. SSL offloading)
- Disadvantages:
  - Increased complexity; requires deep knowledge of the application
  - Performance overhead, higher latency
  - Resource intensive, costly



Stateless Load Balancing – Others

- Load balancing on different levels can be combined
  - e.g. distribute network load among distinct LBs first through DNS

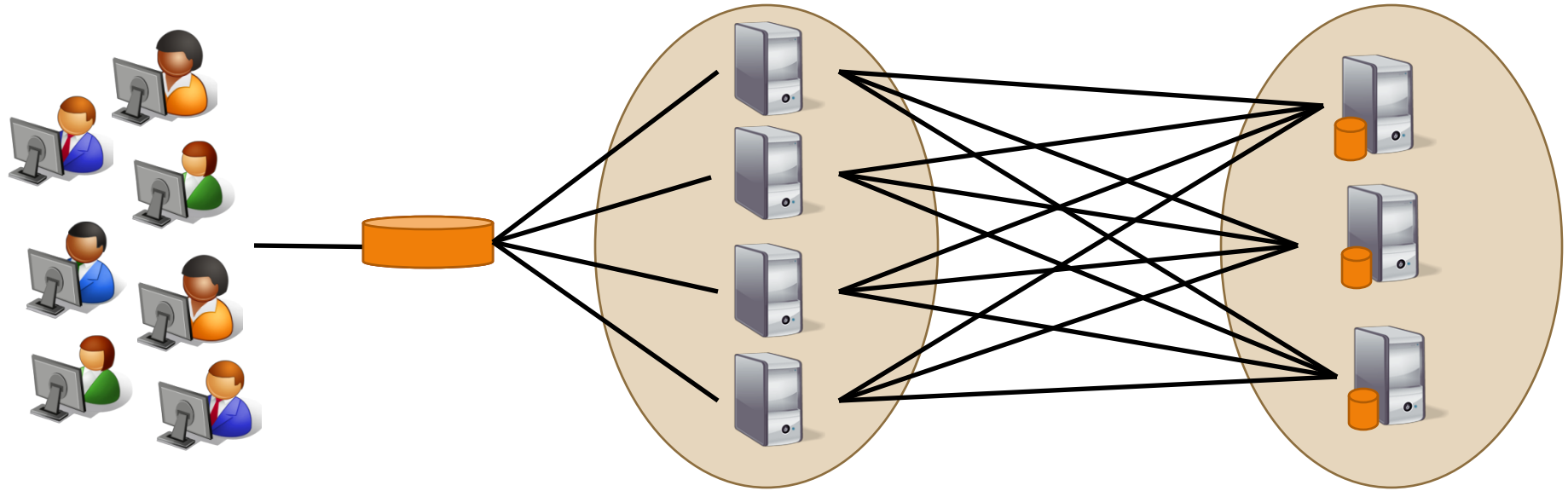
- Other load balancing strategies

1. Feedback-based – servers report their load levels back to the LB
2. Client-based – choose the server with the least network latency

Stateful Scaling

- ✓ Error detection
- ✓ Failover (Error recovery)
- ✓ Load balancing
- ○ Redundancy / Replication
- ✓ Auto-scaling

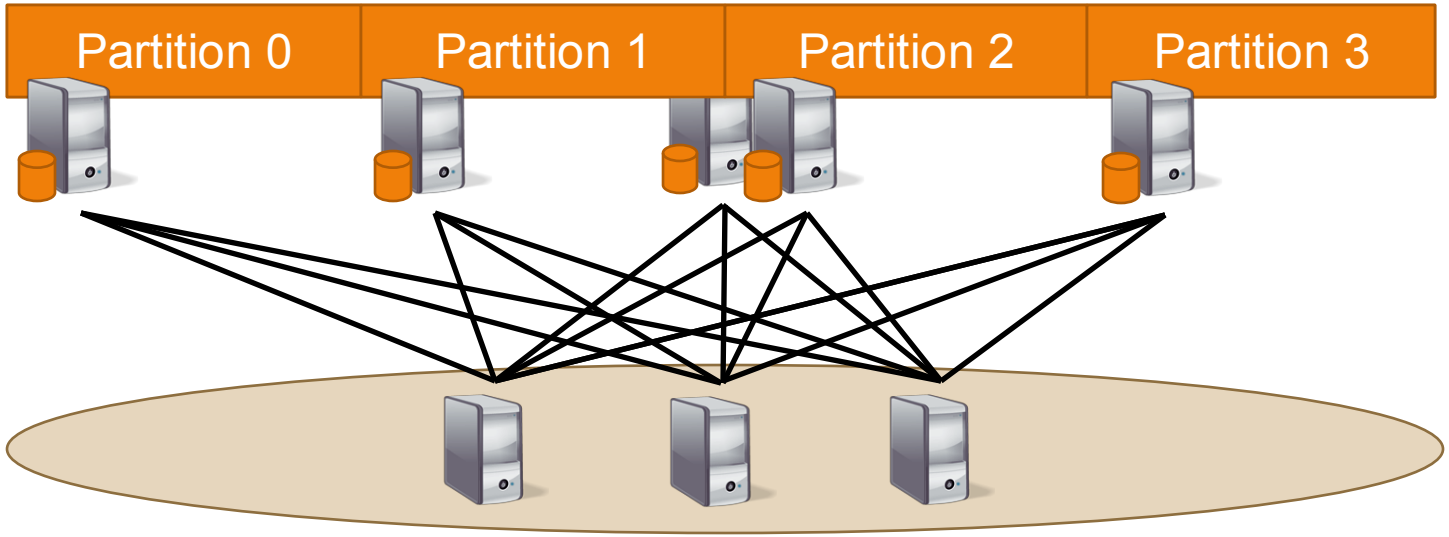
- ○ Now assume that the data tier is the bottleneck



- ○ A database (DB) is stateful; retains information about previous interactions
- ○ Requests from a client must be handled by the same DB instance

Partitioning

- Divide data into distinct, independent parts
- Each server is responsible for one or more parts



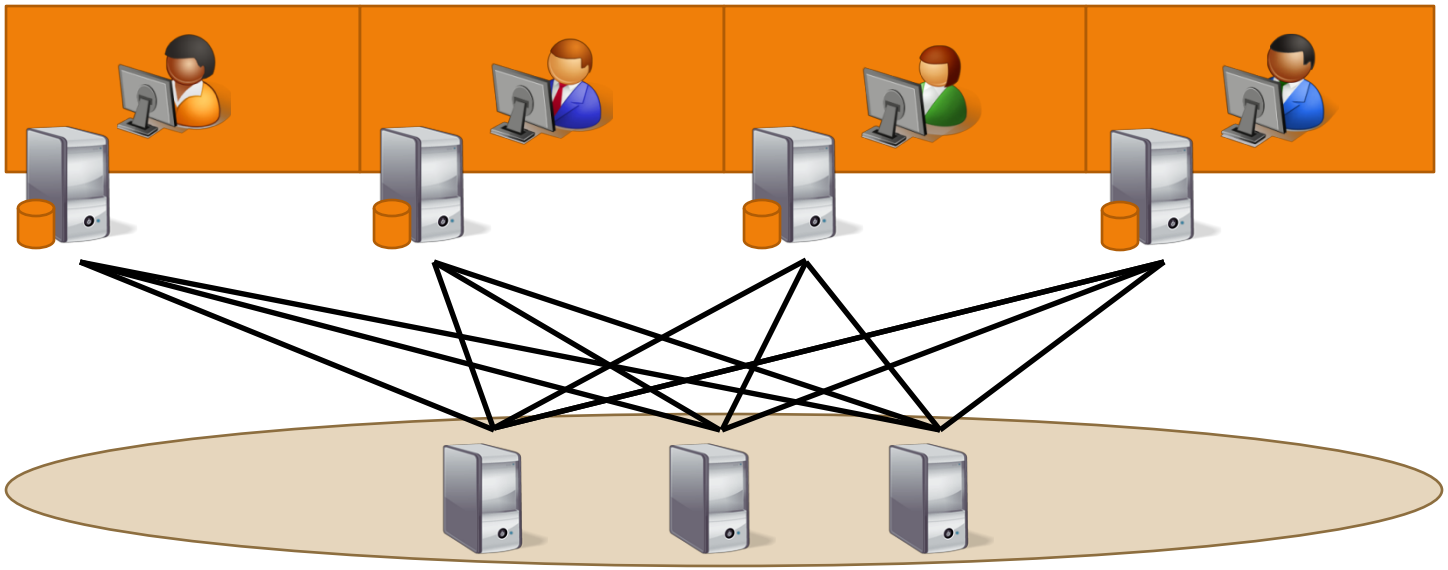
- Partitioning improves just scalability (performance), not availability
  - Each data item is only stored in one partition

Partitioning – General Considerations

- Data is spread across several machines
- Particular tasks might require reading/writing data from multiple servers
  - causes additional network traffic
- Challenge
  - Network is a scarce resource with limited scalability
  - Becomes scarcer the more machines you add
- For good scalability, the goal of every partitioning scheme is typically to reduce network communication
- However, this is highly application-specific…

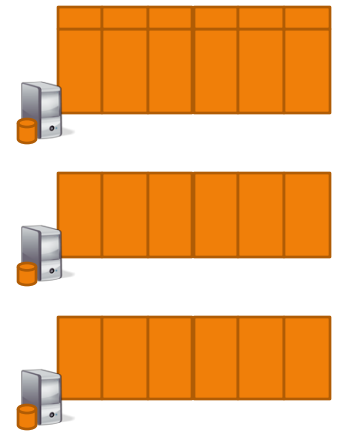
Partitioning Schemes – Per Tenant

- Put different tenants on different machines
  - Isolation is typically a requirement in cloud environments
  - No network traffic between machines, good scalability
  - Con: Scaling a tenant beyond one machine becomes complex



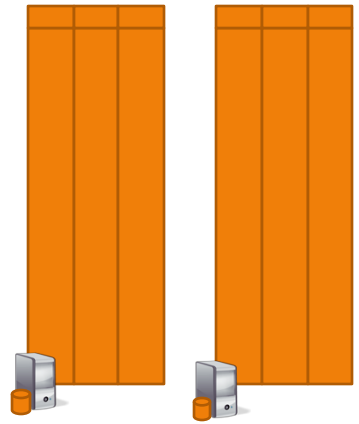
Partitioning Schemes – Horizontal / Sharding

- Split table by rows across different *shards* distributed across servers
- Each shard has the same schema, but contains only a subset of rows
  - Based on a partitioning key, e.g. user ID, region, etc.
- Reduced number of rows, reduced indices
- Easy to scale out (and up)
- If one shard fails, only that data is affected
- Con: cross-shard queries can be slow; requires balancing
- Examples: Google BigTable and MongoDB



Partitioning Schemes – Vertical

- Split table by columns, grouping related columns together
- Can improve performance of specific queries (e.g. routine analytics)
- Cons:
  - Does not inherently support scaling across multiple servers
  - Cross-partition queries may require additional joins, increasing latency



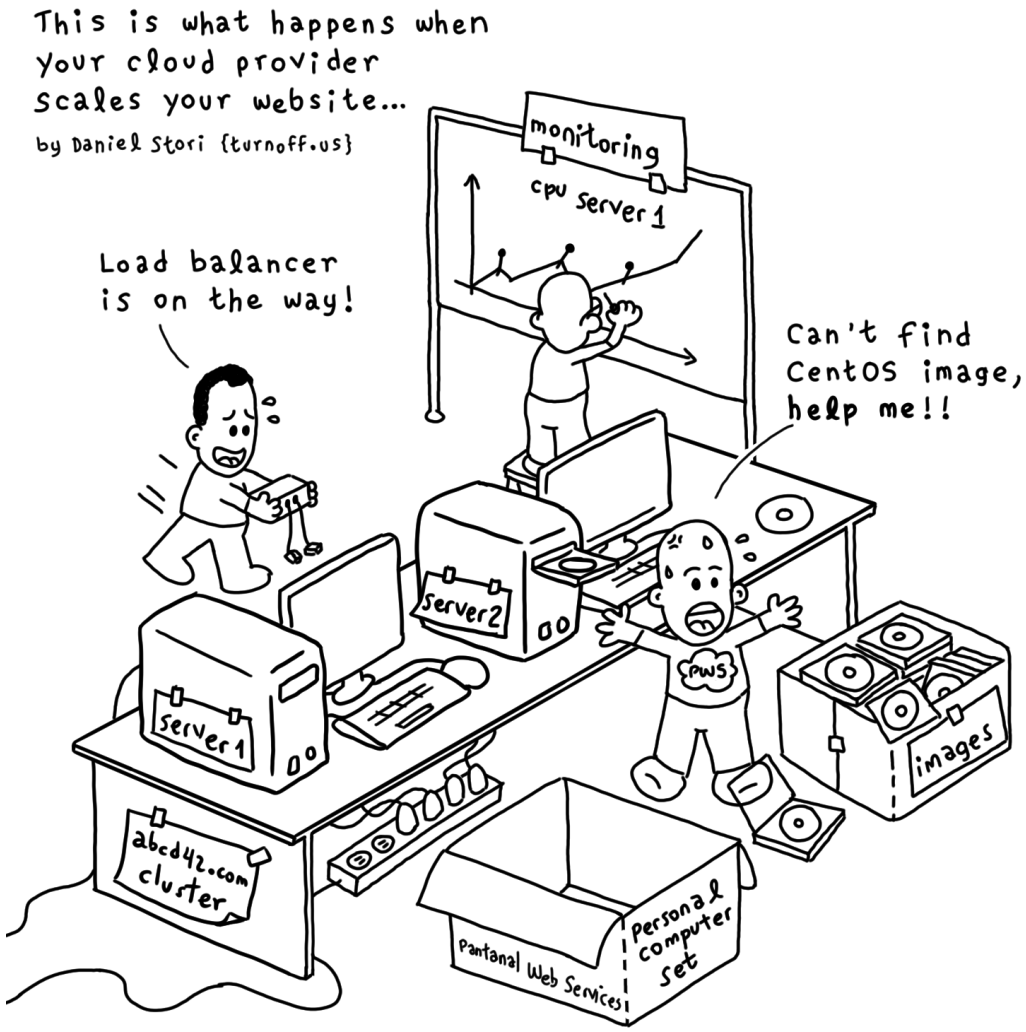
Partitioning – How to Distribute Partitions?

- Range partitioning
  - Goal: Close DB attributes will also be stored close to each other
  - Pro: Easy; efficient for range queries; simple to scale individual partitions
  - Con: Poor load balancing properties; requires manual adjustment
- Hash partitioning
  - Goal: Uniform distribution
  - Pro: Good load balancing characteristics
  - Con: Inefficient for range queries; requires reorganization when number of partitions changes

Further Reading

- Eric Brewer, "*CAP twelve years later: How the 'rules' have changed*", IEEE Computer, vol. 45, no. 2, 2012.
- James C. Corbett et al., "Spanner: Google's globally distributed database", ACM TOCS, Volume 31, Issue 3, 2013.
- Giuseppe DeCandia et al., "*Dynamo: Amazon's highly available key-value store*", ACM SIGOPS OS Review, Volume 41, Issue 6, 2007.
- Martin Klepmann, "*Designing Data-Intensive Applications*", chapter 6, O'Reilly, 2017.

Break!


This is what happens when your cloud provider scales your website...
by Daniel Stori {turnoff.us}
monitoring
cpu server 1
Load balancer is on the way!
Can't find CentOS image, help me!!
server2
server1
abcd42.com cluster
PWS
images
Pantanal Web Services
personal computer set

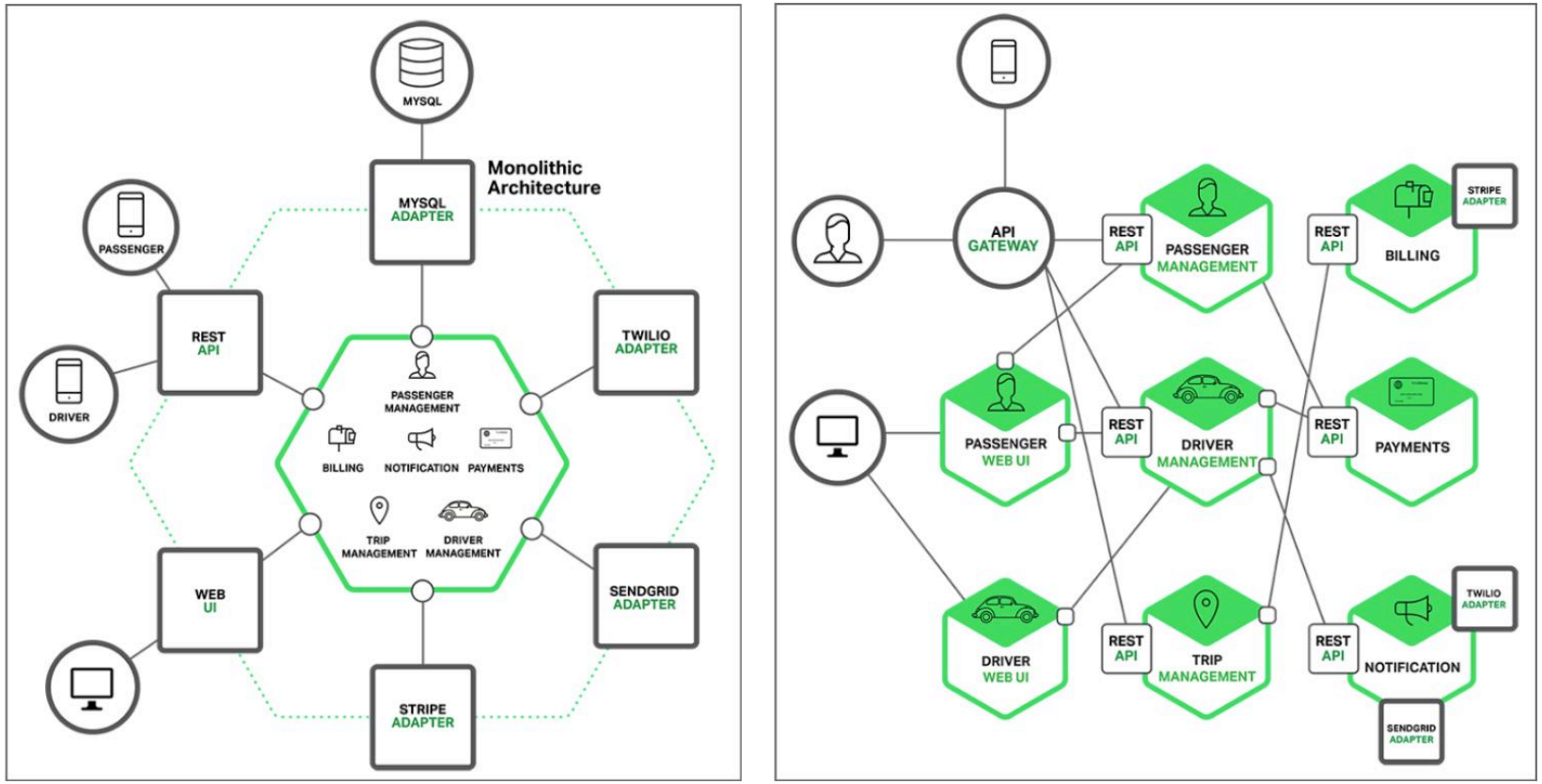
Advanced Architectures & Technologies

The Microservice Architecture

- Monoliths are incredibly difficult to maintain and scale
- Even layered architectures can be too complex to evolve
- n-tiered architecture
  - Fragment logic into loosely coupled elements (services)
  - Each service focuses on one job
  - Services can have completely different tech stacks
  - Services communicate through lightweight APIs (e.g. REST)
- The dominant architectural pattern
  - Top performers adopt microservices in 95.5% of applications

Humantiec, "*DevOps Setups – A Benchmarking Study*". Technical Report, 2021

The Microservice Architecture



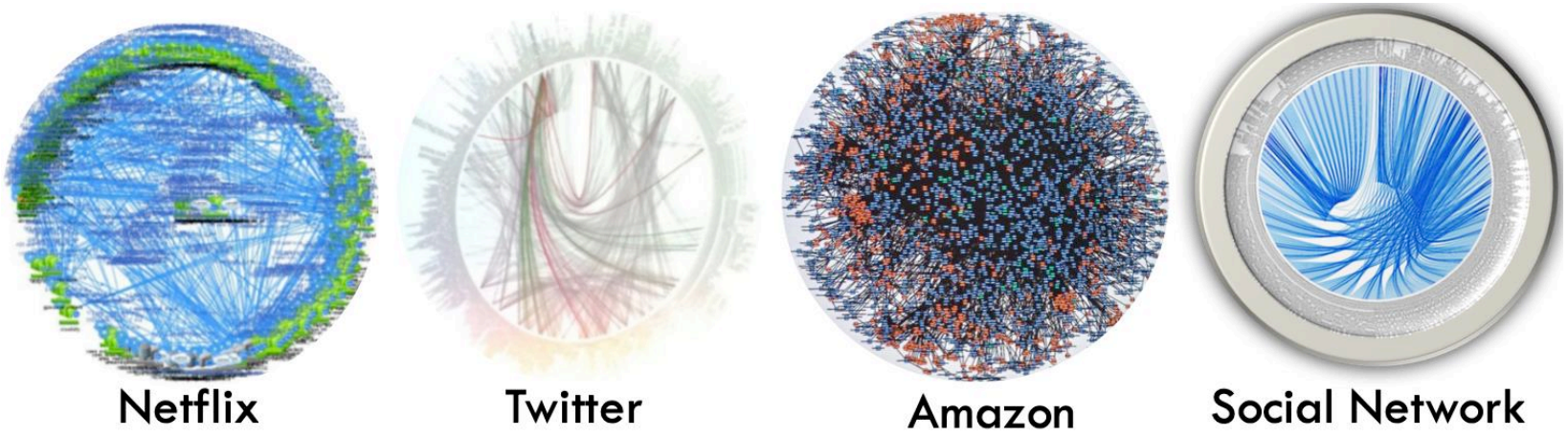
Christina Delimitrou, "*The Hardware & Software Implications of Microservices and How Big Data Can Help*", Stanford Oct 2018.

Microservices – Advantages

- Loose coupling
- Independent deployment, fault isolation (not affecting other services)
- Development flexibility, technology diversity
- Agility – faster development cycles
- Scalability – only scale parts that need it
- Maintainability – smaller codebases are less complex
- Easy reuse in different projects
  - e.g. Netflix Asgard, Eureka

Any cons?

The Microservice Architecture



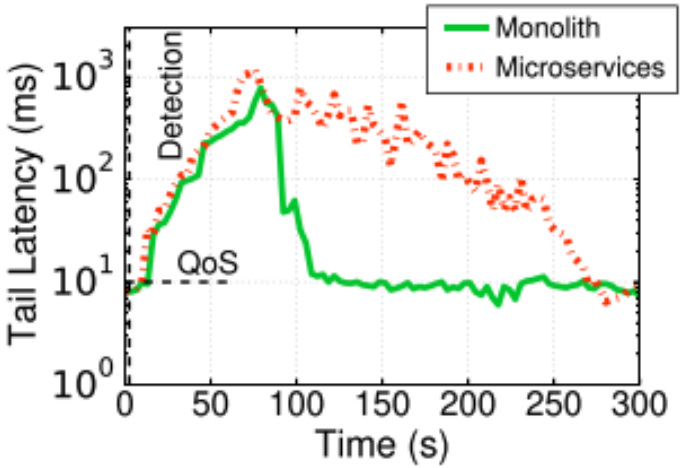
Y Gan et al. “An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems”, ASPLOS 2019

Microservices – Disadvantages (1/2)

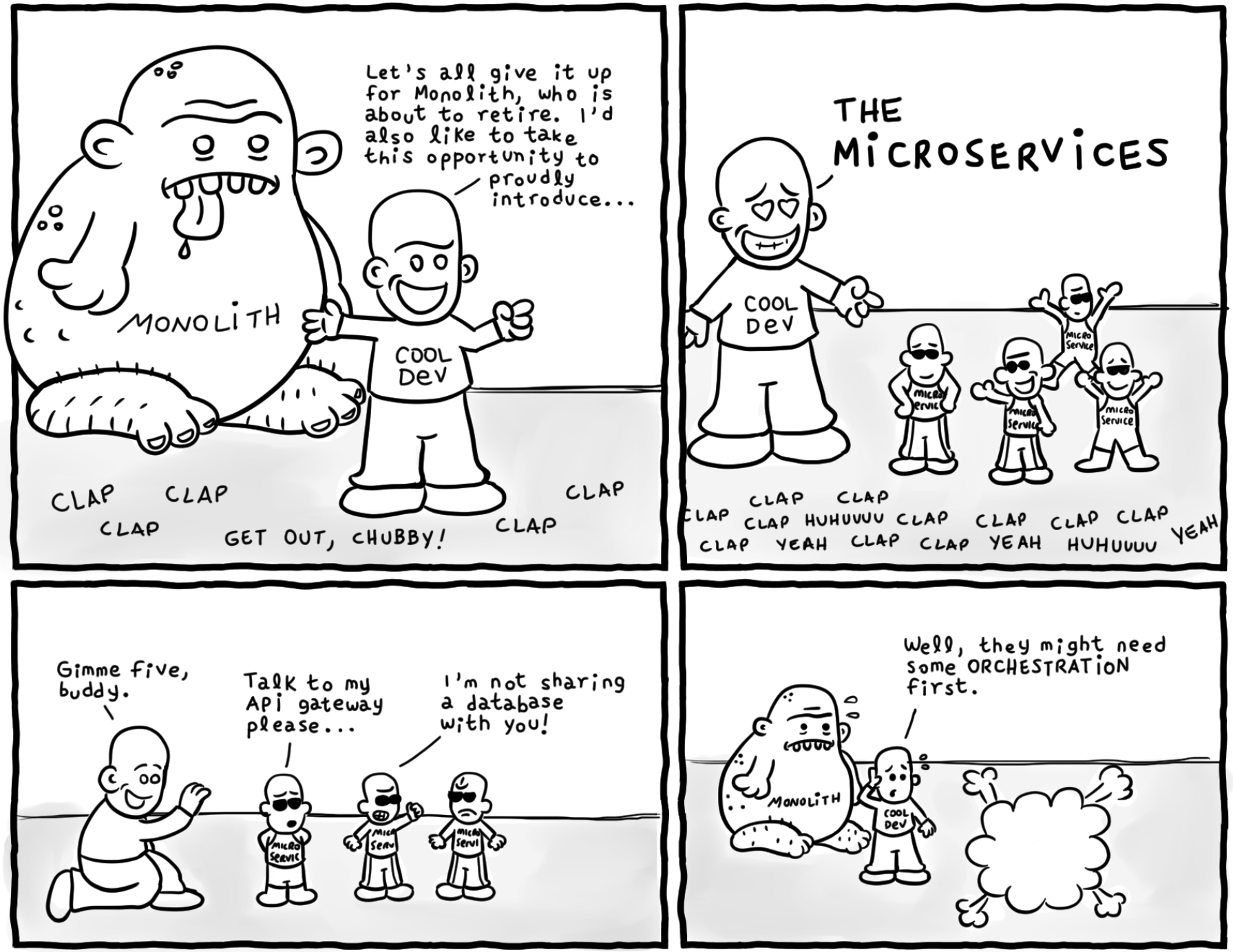
- Complexity
  - Increased operational overhead – more services to manage and monitor
  - Distributed debugging – tracing issues across multiple services
- Latency due to communication between services
- Microservice sprawl – could balloon into 100s or 1,000s of services
- Operational overhead – manage CI/CD pipelines for multiple services
- End-to-end testing can be difficult

Microservices – Disadvantages (2/2)

- Interdependency chains can cause chaos
- Failure of one service can trigger:
  - failures in dependent services, creating a domino effect (**Cascading failures**)
  - failures in other containers of the same service (**Death spiral**)
  - wasted resources on failed calls, degrading the calling service (**Retry storm**)
- Cascading QoS violations, due to bottleneck services
- Failure recovery could take longer than monoliths



Christina Delimitrou, "*The Hardware & Software Implications of Microservices and How Big Data Can Help*", Stanford Oct 2018.



Daniel Stori {turnoff.us}

Microservices – *Glue*ware

- Software to support monitoring, consistency, load balancing, coordination, service discovery, message passing, resilience, etc.
- In 2022, glueware outweighed core microservices

Cloud Native Computing Foundation, "*CNCF Annual Survey*", Technical Report, 2022.

Microservice Communication

Synchronous Communication

- REST APIs (HTTP/HTTPS)
- gRPC (Bidirectional streaming)
- GraphQL (Query-based)

Pros:

- Immediate response
- Simpler to implement
- Easy to debug

Cons:

- Tight coupling
- Higher latency
- Lower fault tolerance

Asynchronous Communication

- Message queues (RabbitMQ, Kafka)
- Event streaming
- Pub/Sub pattern

Pros:

- Loose coupling
- Better scalability
- Higher fault tolerance

Cons:

- More complex to implement
- Harder to debug
- Eventually consistent

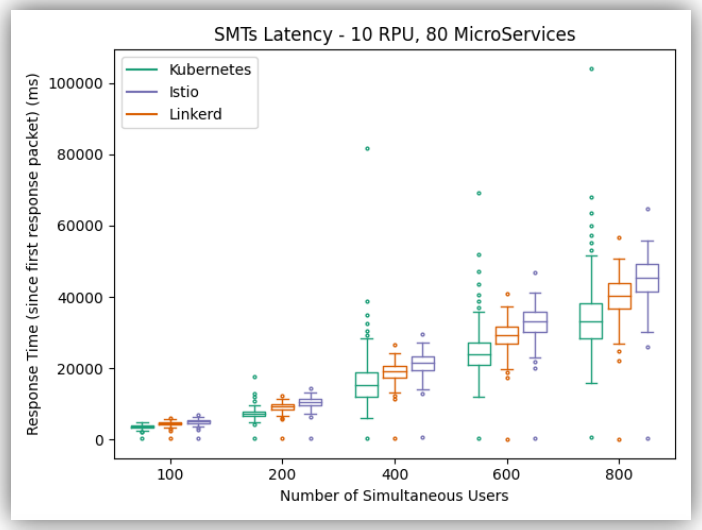
Service Mesh Technologies (SMTs)

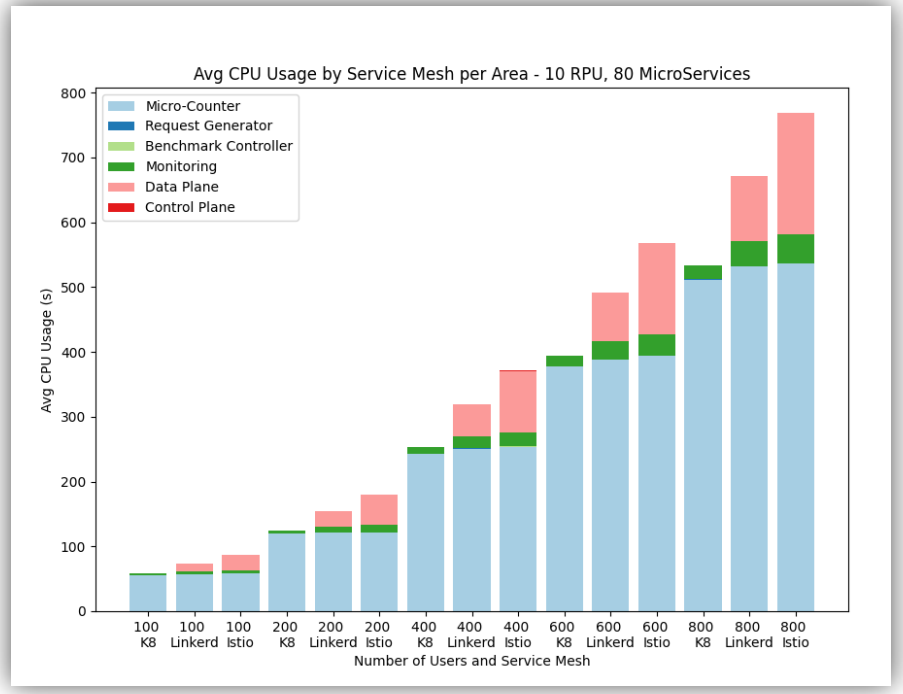
- A dedicated infrastructure layer for managing and securing comms
- Rely on the pattern of sidecar proxies
  - Microservice = a container with core business logic and a sidecar container
- The sidecar acts as the ingress and egress point, with auxiliary logic
  - Feature-rich data plane (low latency, secure)
  - Control plane (traffic management, service and traffic observability, certs)
- Programmatically define policies for s2s interactions
  - extract this logic out of the core business logic
  - version and apply consistently at different levels
- SMT use increased from 27% (2020) to 47% (2022)

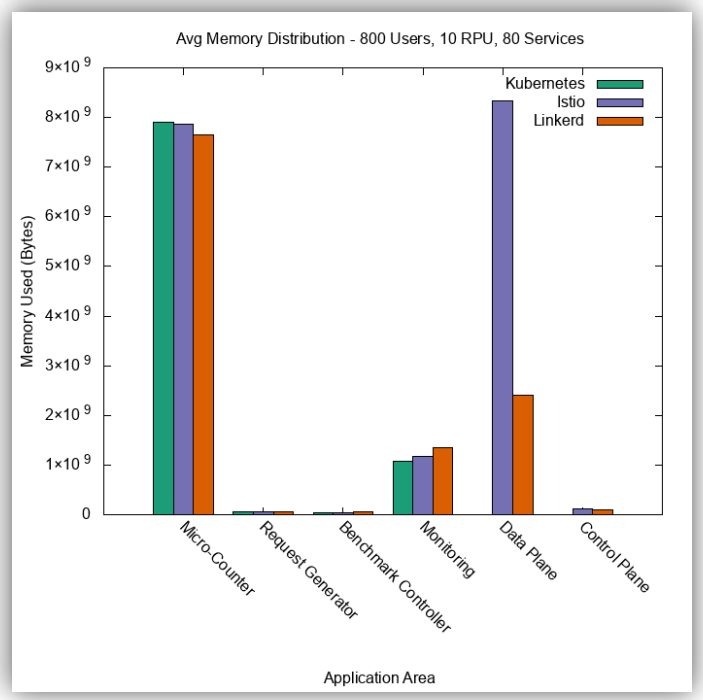
Cloud Native Computing Foundation, "*CNCF Annual Survey*", Technical Report, 2022.

SMT Comparison

- Linkerd is 6%–12% faster than Istio
- Istio uses 20%–43% memory > Linkerd
- Istio uses 16%–20% cpu > Linkerd







Yehia Elkhatib, and Jose Povedano Poyato, "*An Evaluation of Service Mesh Frameworks for Edge Systems*", EdgeSys, 2023

API Gateways


- A reverse proxy that sits between clients and backend services
- Acts as a single-entry point for all API calls
- Key responsibilities
  - Route incoming API requests to backend microservices
    - Based on the request path, method, etc.; Might also do load-balancing
  - Authentication and authorization
    - Verify the identity of clients
    - Ensure they have the necessary permissions to access specific resources
  - Protocol transformation and data format conversion
    - Translate between different protocols (e.g., REST, gRPC, GraphQL) when needed
  - Monitor API usage and rate-limiting

Microservices – Avoiding the Sprawl

- Start with a monolith design
  - Gradually break it down into microservices as needed
  - Identify natural boundaries and avoid over-decomposition
- Focus on business capabilities
  - Design around clear purposes rather than technical functions
- Establish clear governance
  - Define guidelines and best practices for microservice development
  - e.g. naming conventions, comm. protocols, deployment procedures
- Follow fault-tolerant microservice design practices
  - e.g. timeouts, bounded retries, circuit breakers

V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. K. Reiter and V. Sekar,
"*Gremlin: Systematic Resilience Testing of Microservices*", IEEE ICDCS, 2016

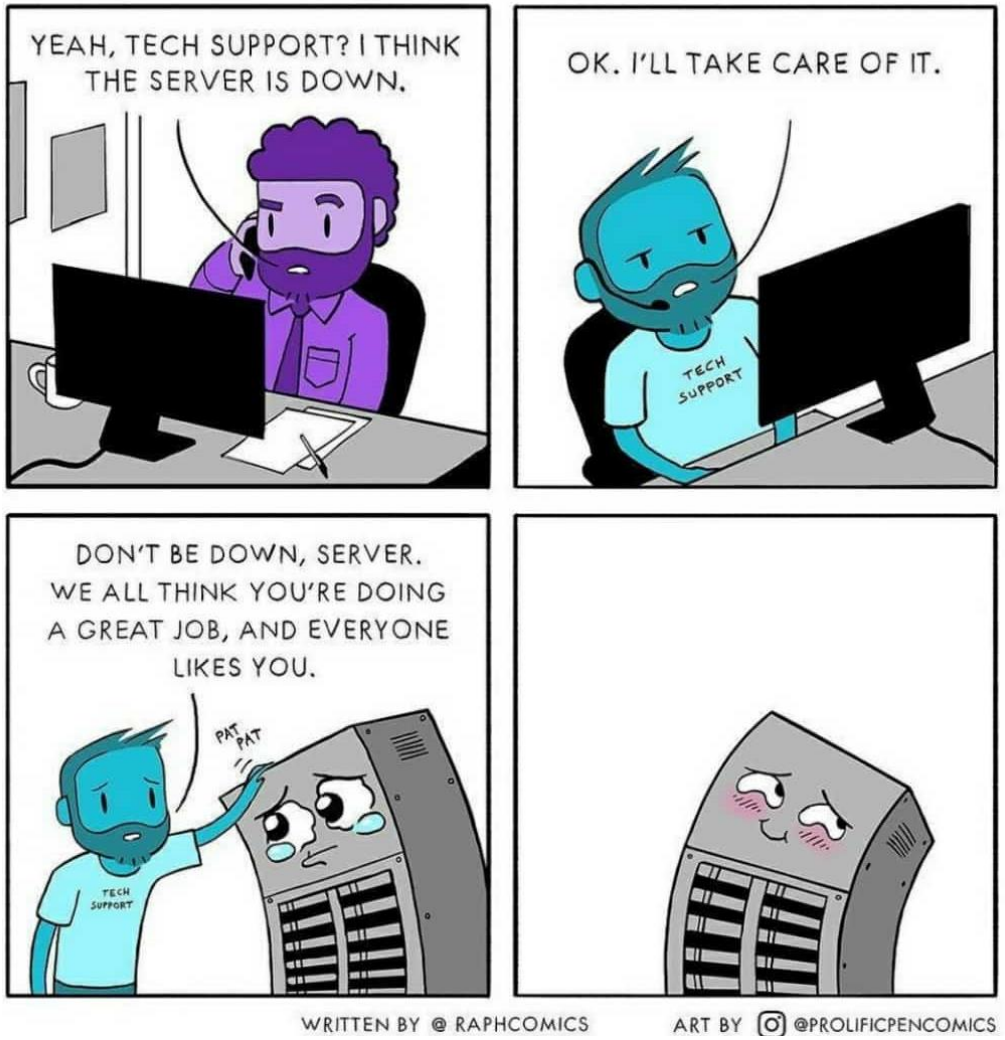
Cloud-native Technologies

- A collection of tools and techniques to build, deploy, and manage applications designed for cloud computing environments
- Embrace the cloud's scalability, elasticity, and dynamic nature
- Key Principles: Microservices, Containers, DevOps, Automation
- Benefits:
  - Increased agility, faster development cycles
  - Improved scalability
  - Enhanced resilience, due to distributed architectures and redundancy
  - Greater efficiency, through optimised resource utilization
  - Increased portability

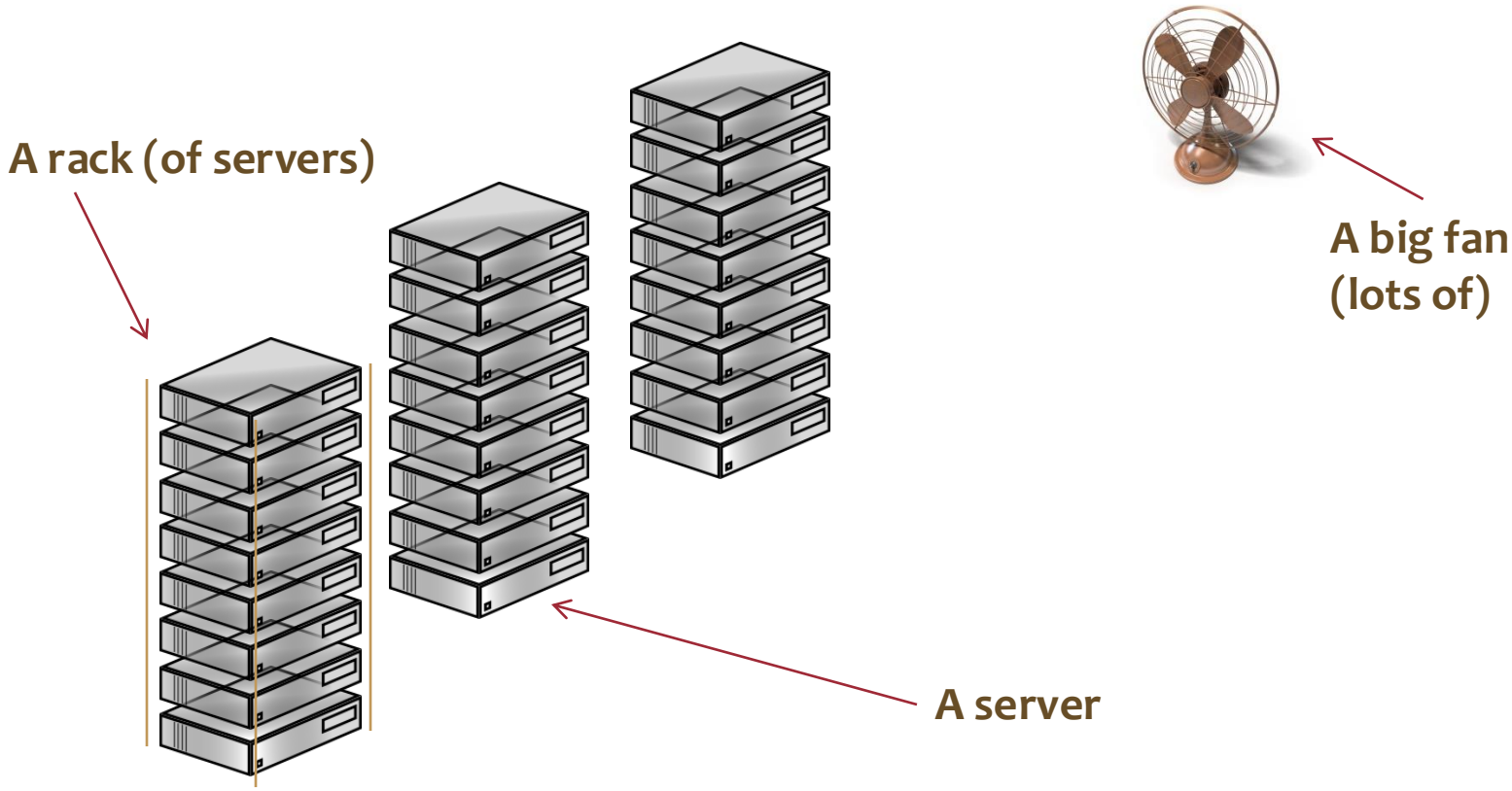
Cloud-native Technologies – Examples

- Containers: Docker, Containerd, Podman
- Orchestration: Kubernetes, Docker Swarm
- Microservices: Spring Boot, Node.js with Express
- Serverless Computing: AWS Lambda, Google Cloud Functions, Knative
- Message Queues: Azure Queue Storage, Google Cloud Pub/Sub
- Service Mesh: Istio, Linkerd, Consul Connect
- Container Registry: Docker Hub, Amazon ECR, Google GCR, ACR
- API Gateways: Kong, Apigee, Ambassador, Gloo
- Monitoring and Logging: Prometheus, Grafana, Jaeger, Fluentd
- …

Designing Dependable Data Centres



Data Centre Infrastructure Basics



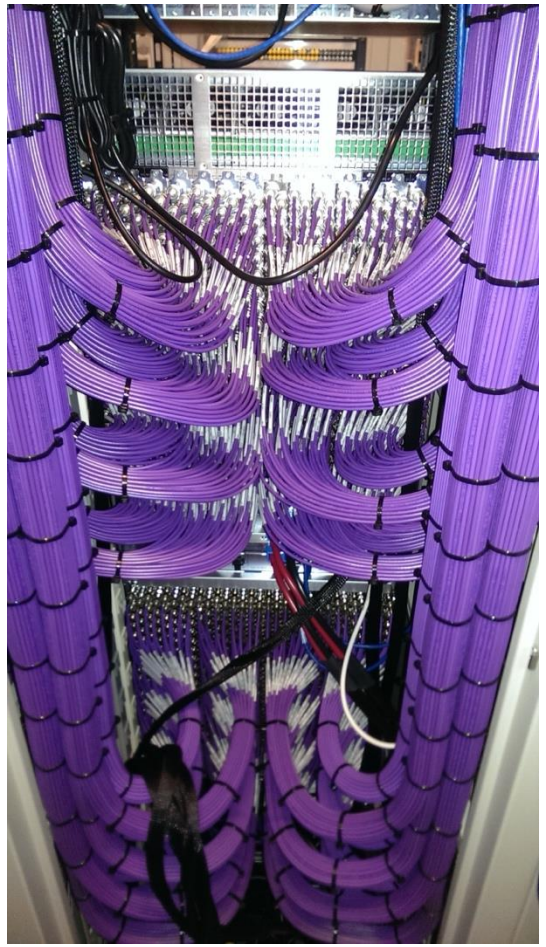
Hardware Redundancy

- Geographic redundancy
  - DCs are distributed across multiple geographic regions
  - Mitigate the impact of regional outages, e.g. natural disasters, power grid failures
  - Data is typically replicated across regions to ensuring high availability
- Server redundancy
  - Servers are deployed in clusters with automatic failover mechanisms
  - If one server fails, another takes over seamlessly (managed by virtualization tech.)
- Storage redundancy
  - Data is replicated across multiple storage devices and technologies
  - Within the same DC and across other DCs
  - RAID configurations to protect against data loss due to disk failures

Network Redundancy

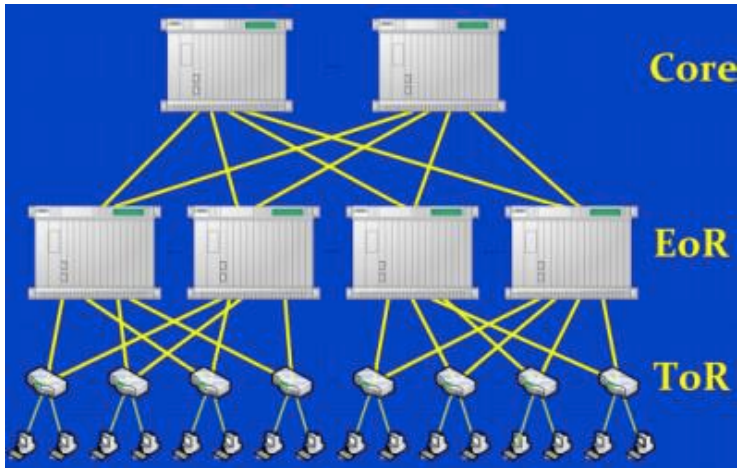
- Challenge
  - Reliably connect 100,000s of computers
  - Strict requirements of high bandwidth and low latency
  - Expect hardware-level failure at any and all points

1. Server-level
   - Redundant Network Interface Cards (NICs)
   - Dual (or more) power supplies
2. Network-level
   - Redundant switches, routers, firewalls, load balancers

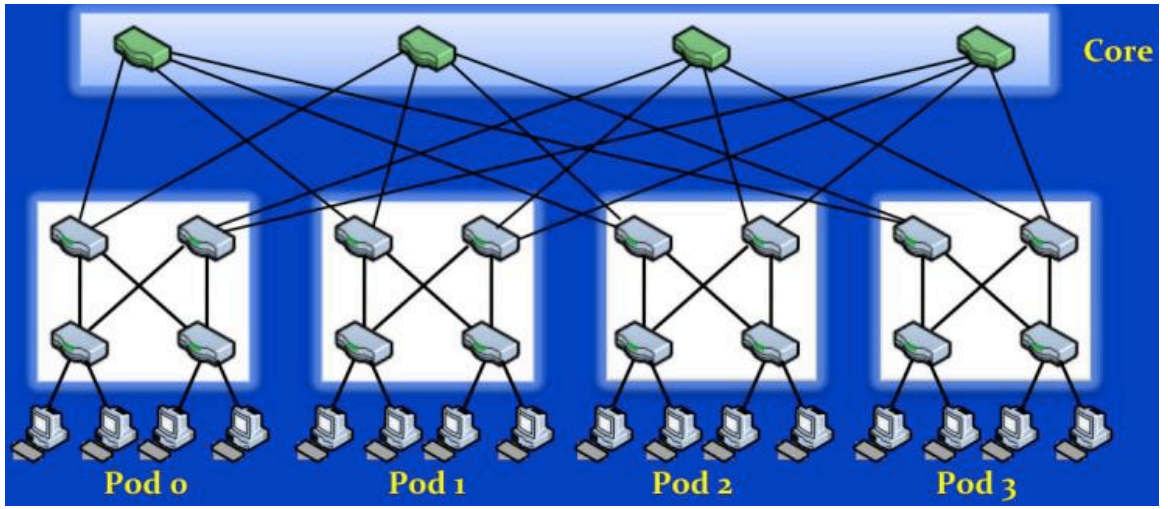


Network Redundancy

3. Link- and path-level
   - Link aggregation: multiple links between network devices
   - Spanning Tree Protocol (STP) variants to prevent network loops
   - Distribute traffic across multiple paths, i.e. load balancing traffic



Hierarchical / 3-tier topology



Fat-tree / clos topology

Power Redundancy

- DCs require a constant and reliable supply of power
- Power redundancy measures
  - Multiple power feeds from different utility substations
  - Uninterruptible Power Supplies (UPS) as backup for temporary outages
  - Backup diesel / gas generators for medium/long-term outages
  - Power Distribution Units (PDUs) with dual power inputs to ensure continuous rack power

Cooling Redundancy

- DCs generate a significant amount of heat
- Cooling redundancy measures
  - N+1 – having one extra cooling unit than required to handle the load
  - Multiple cooling technologies to mitigate single points of failure
    - e.g., CRAC units, free cooling, in-row cooling
  - Redundant cooling loops (pipes, heat exchangers, and pumps)
  - Hot/Cold aisle containment – prevent hot and cold air from mixing to improve efficiency

Further Reading

- Lee et al., "*The Tale of Errors in Microservices*", ACM Measurement and Analysis of Computing Systems 8.3: 1-36, 2024.
- DORA, "*Loosely Coupled Teams*"
- DgtlInfra, "*Data Center Racks, Cabinets, and Cages: An In-Depth Guide*"
- Reboot Monkey, "*Step-by-Step Guide to Designing Your Data Center Layout*"