Architecting Online Services

SCC.306 Internet Applications Engineering

November 2024

Agenda

1

- Purpose of architecting web services
- Traditional Monolith Web Architecture
- Rapid growth of online services
- Challenges with Monolith Web Architecture

2

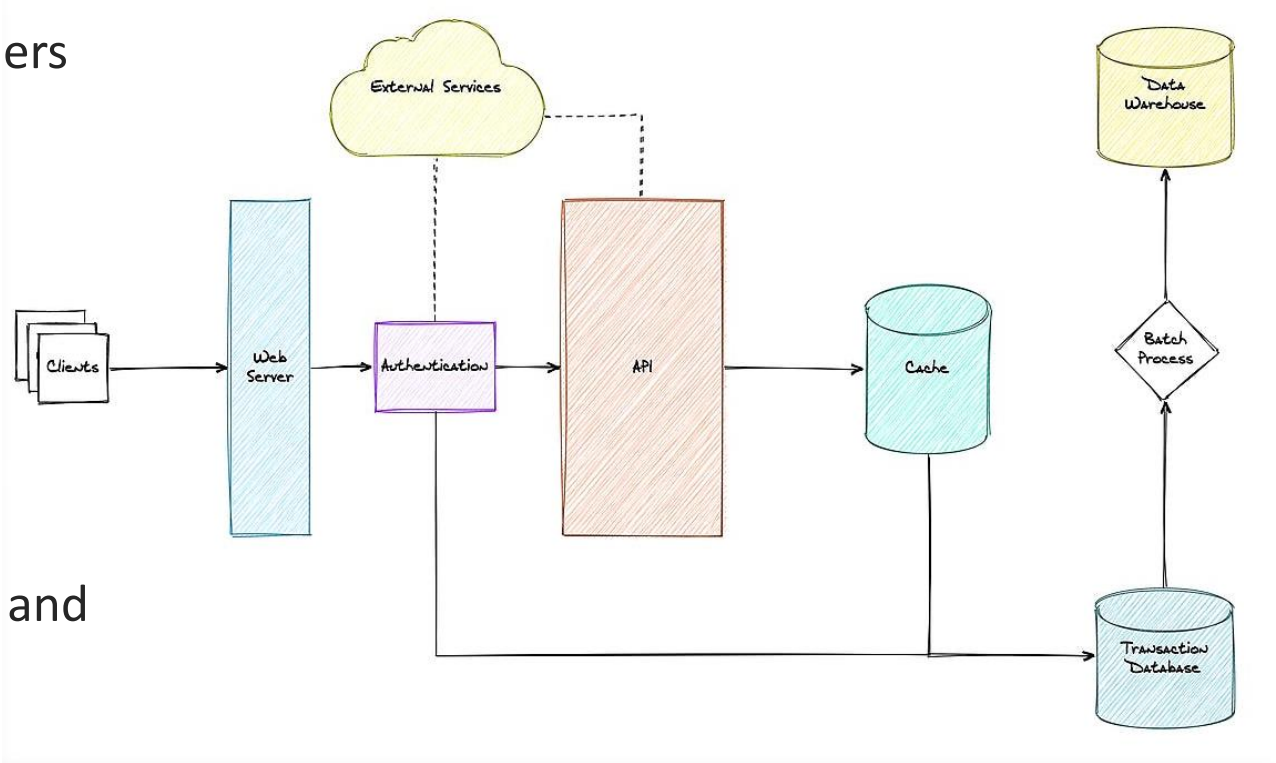
- The Amazon.com Story
- SOA & Microservices at Amazon
- Amazon’s Infrastructure
- How Amazon created AWS
- The state of AWS today

3

- System Design (Case Study)
- UI Options
- Compute Options
- Database Options
- Caching Options
- Summary

Purpose of architecting web services

- Identify components, processes, stakeholders and interactions
- Differentiate between mission-critical components and non-core capabilities (e.g. build v/s buy)
- Identify fit-for-purpose tools, frameworks, and programming environments



Architecture decisions will impact...

Timeline & methodology

Project management, delivery goals and timeline

Team & skills

Team structure, composition and required expertise

Budget & costs

Project budget, upfront expenses, the cost of services and support

Flexibility & agility

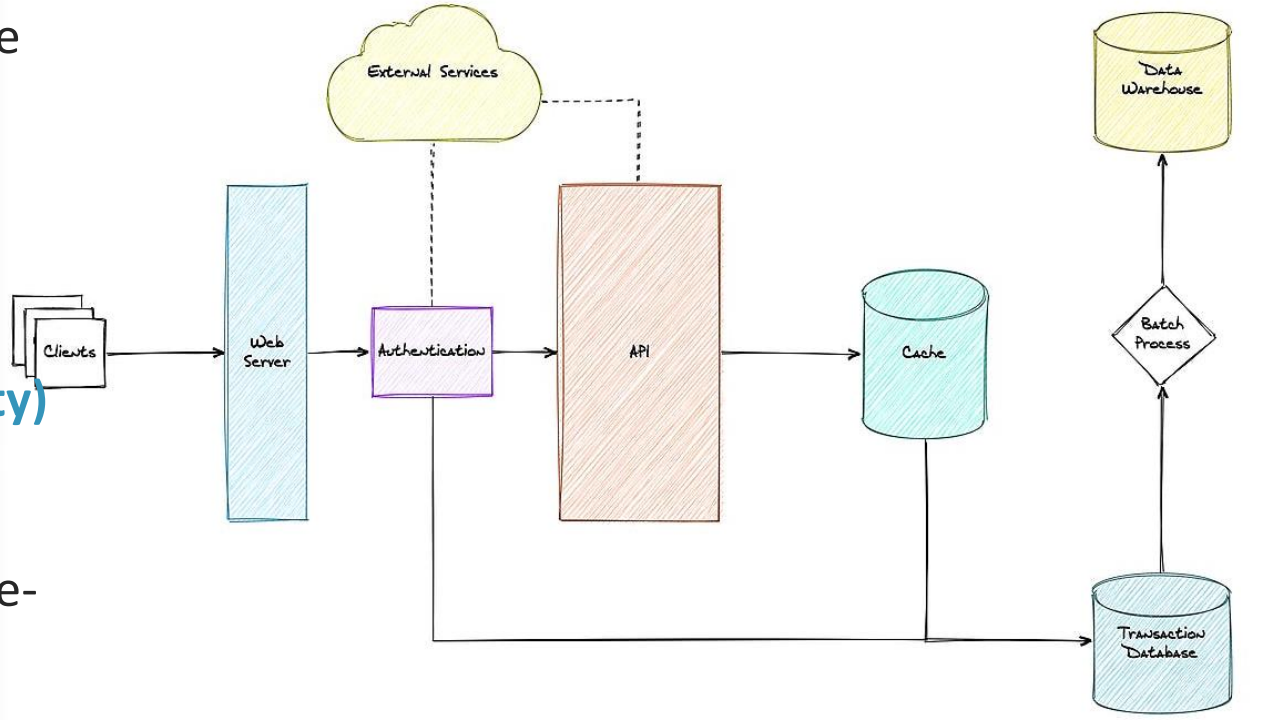
How fast and easy you can add new functionality and modernize stack

Performance & scalability

Speed and ability to cope with the growing load

Benefits of architecting web services

- Establish standards and conventions for the Dev-Test-Deploy process to achieve faster Time-to-Market **(Agility)**
- Reduce single points of failure to increase system reliability and resilience **(Availability)**
- Identify components that need to be future-proofed to accommodate growth in user traffic **(Scalability)**

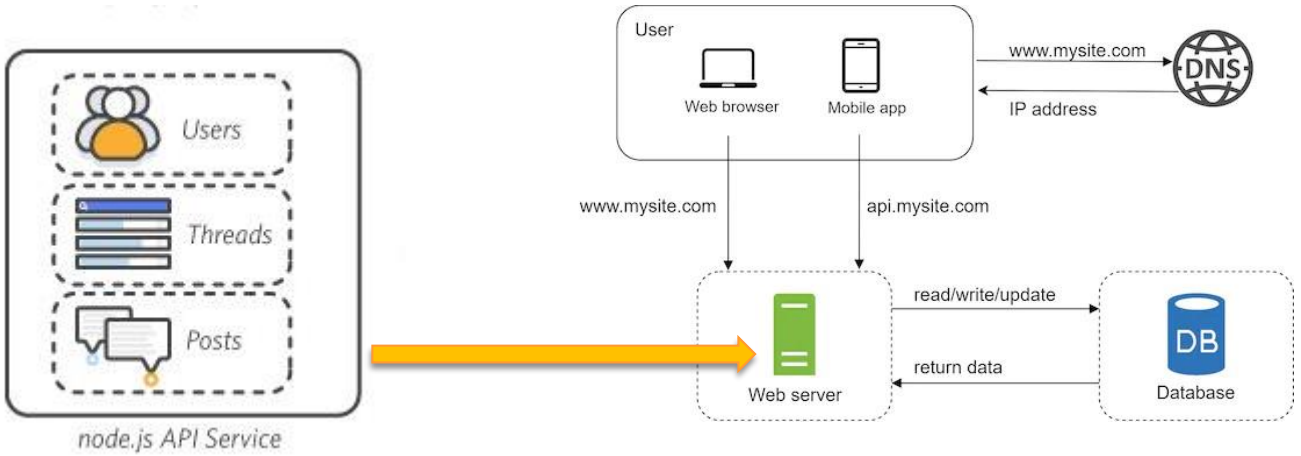


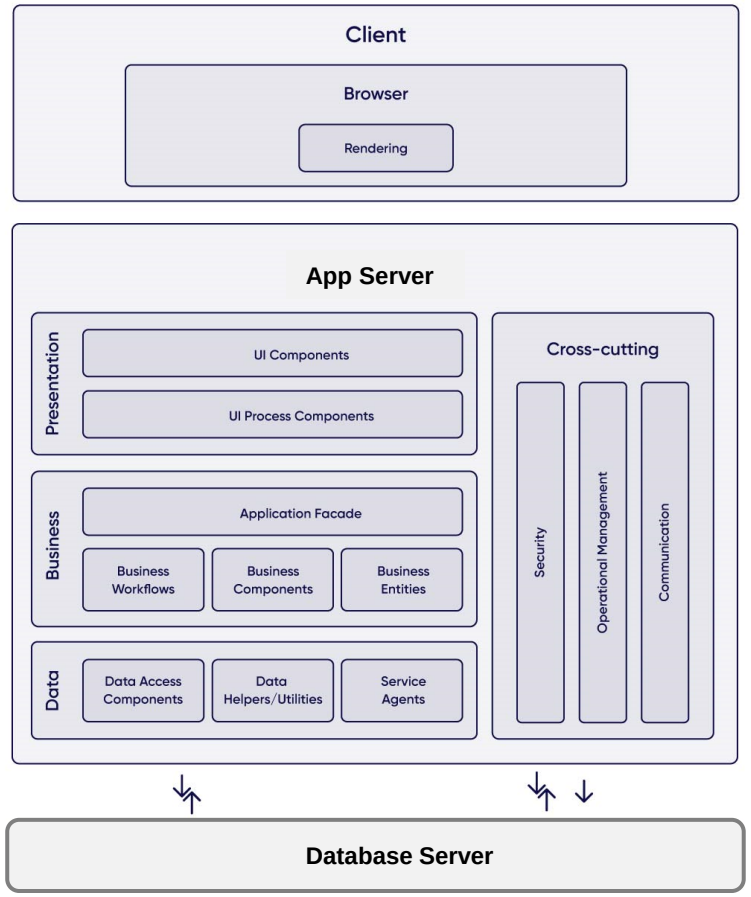
Architecture Evolution

Technology trends that impacted web architecture

Traditional Monolith Web Architecture (1)

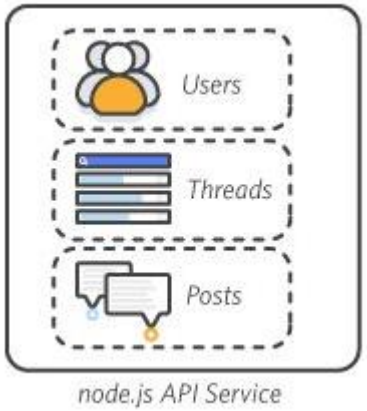
- All application components are bundled into a single codebase. This includes server-side functionality and client-side UI code.
- End-to-end testing, deployment, monitoring and troubleshooting a monolithic app is simpler compared to distributed systems.
- Suitable for rapid-prototyping and reducing time-to-market.

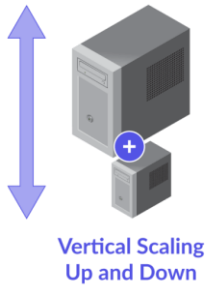


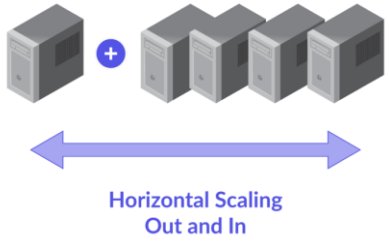


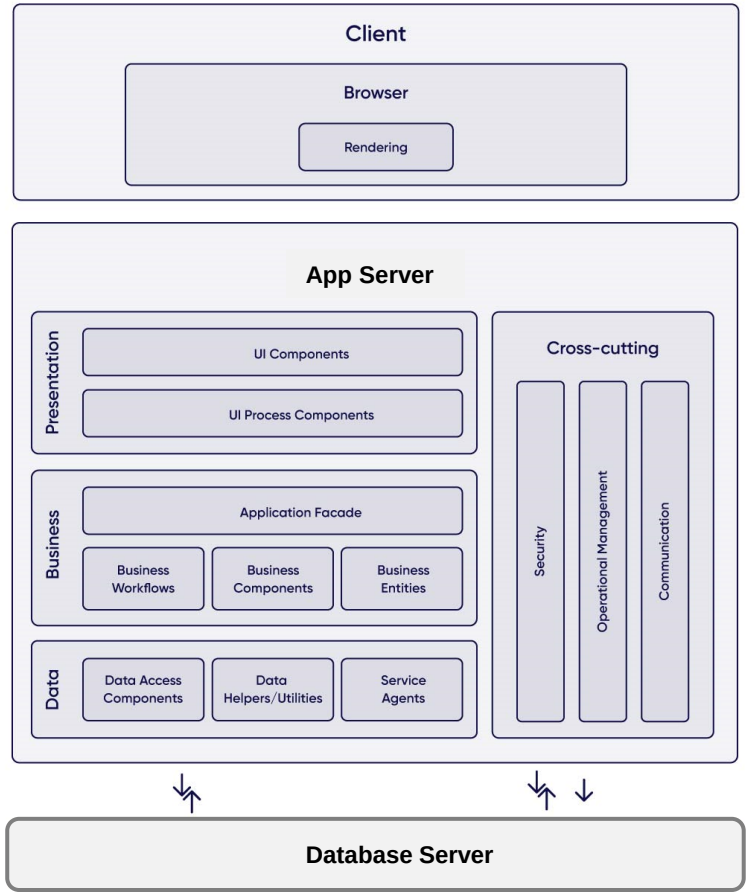
Traditional Monolith Web Architecture (2)

- Vertical scaling (e.g. increasing CPU or memory) is used to accommodate higher user traffic with a single app server
- Horizontal scaling can be achieved with a cluster of app servers and a load-balancer
- Monoliths tend to be convenient early on in a project's life due to easier code management, homogenous tooling, and lower cognitive overhead for smaller teams



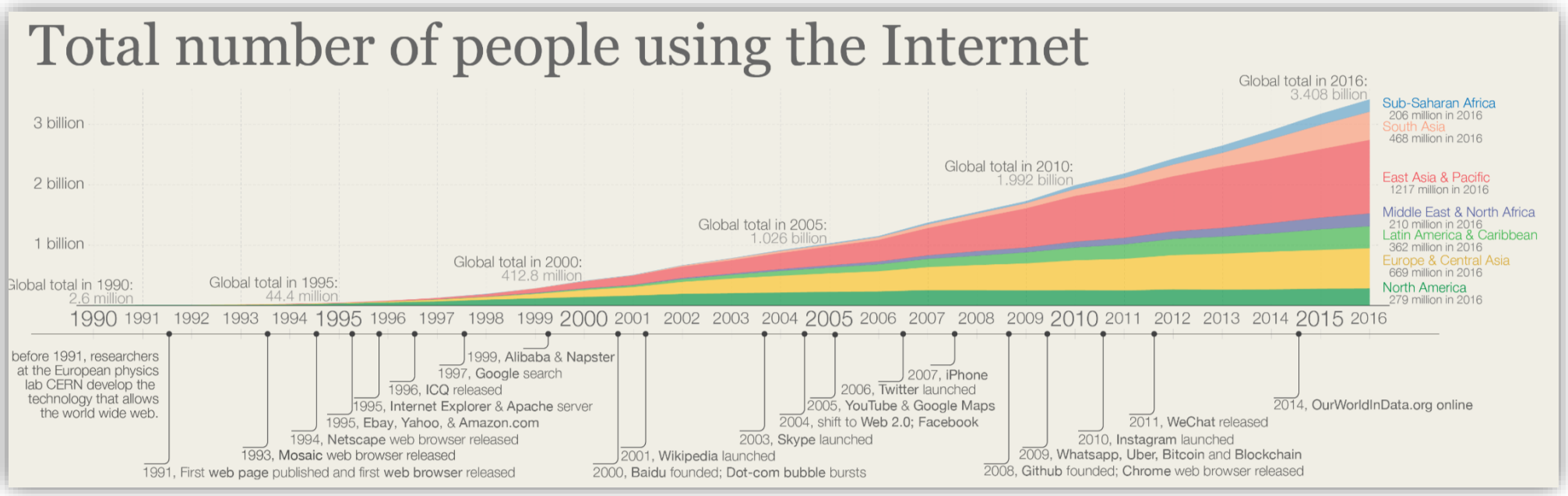






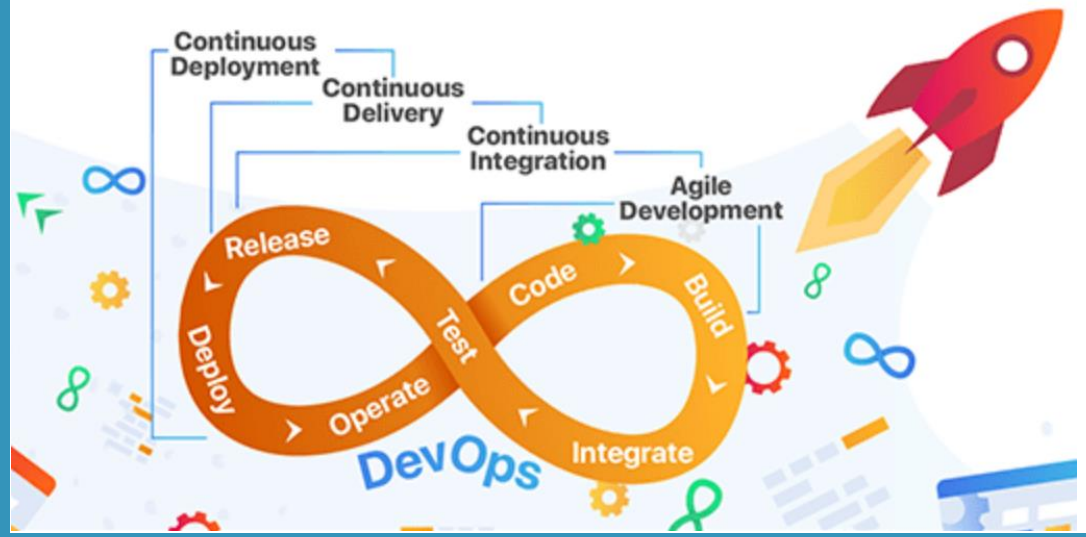
Rapid growth & scale (1)

- Transition from Web 1.0 (static, read-only) to Web 2.0 (dynamic, interactive) websites
- Unpredictable traffic spikes and exponential user growth (e.g. hockey stick growth, virality)
- Rapid adoption of Social-Location-Mobile services



Rapid growth & scale (2)

- Rise of open source tooling, frameworks, and reusable code libraries
- Horizontal scaling had become cheaper due to commoditisation of compute/storage infrastructure
- Acceleration of infrastructure automation e.g. software-driven networking, infrastructure as code, DevOps practices etc.



GitHub

docker

ANSIBLE

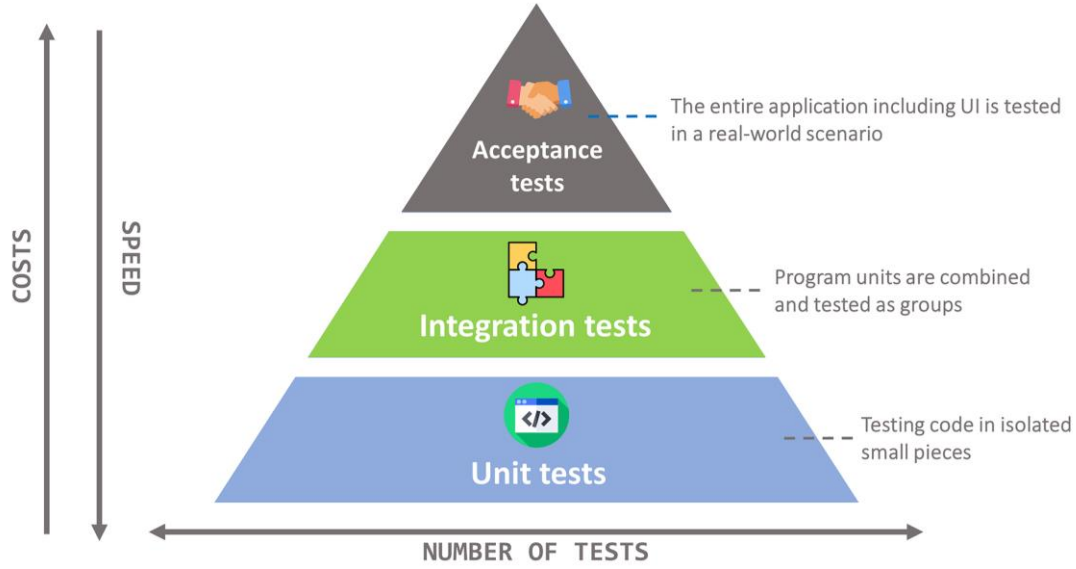
HashiCorp Terraform

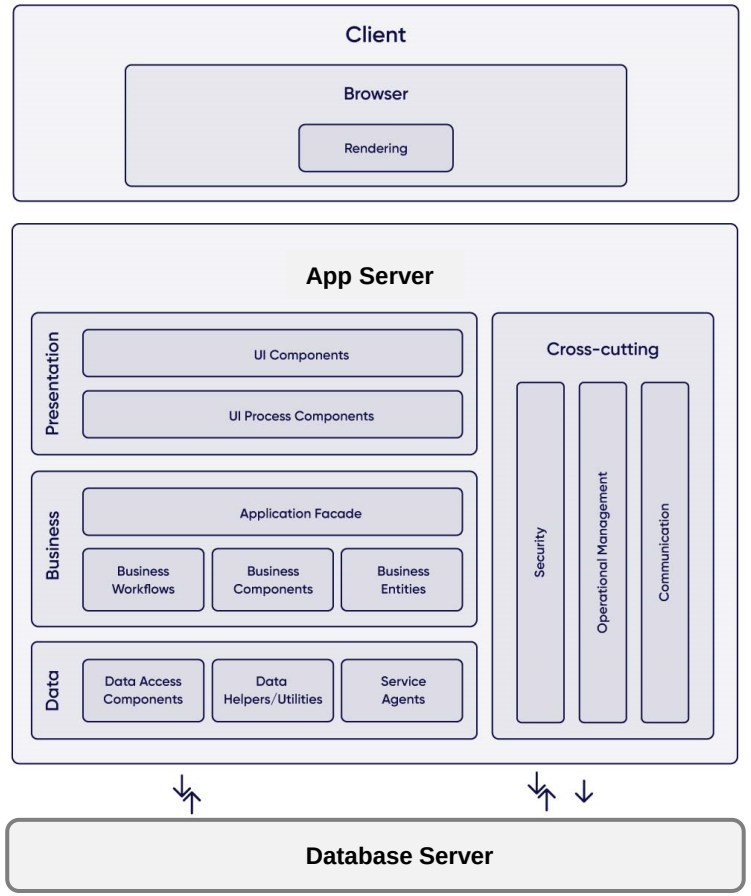
GitLab

kubernetes

Challenges with Monolith Architecture (1)

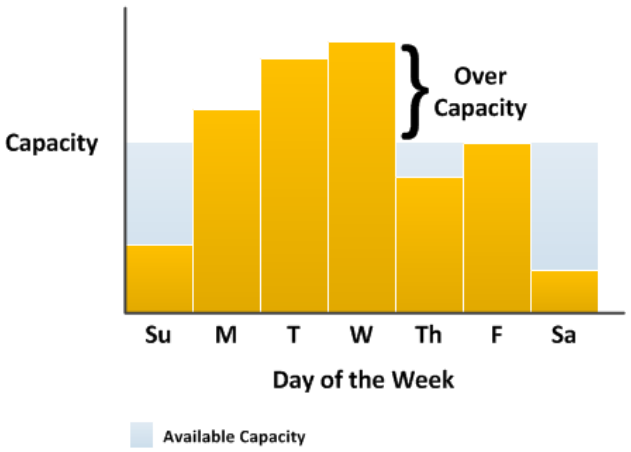
- Codebase can become unwieldy over time as presentation code, business logic code and data access code is intertwined. Code complexity tends to increase with time and team size.
- Making a small code changes to even one application component/function requires updating the entire app and a new deployment



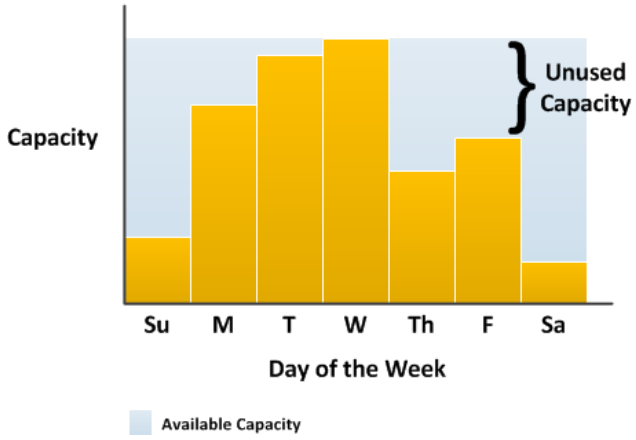


Challenges with Monolith Architecture (2)

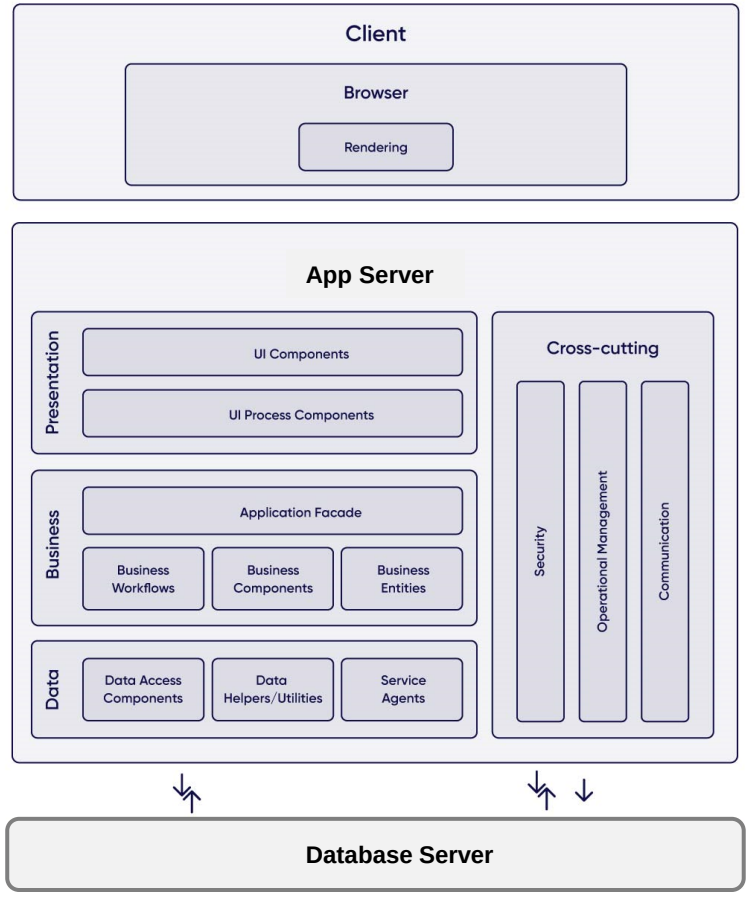
- Vertical scaling is inefficient and tends to underutilise hardware resources such as CPU, RAM etc. during off-peak periods.
- If the server goes down, then the entire web service fails. This can result in lower reliability due to a single point-of-failure.



Under provisioning infra capacity

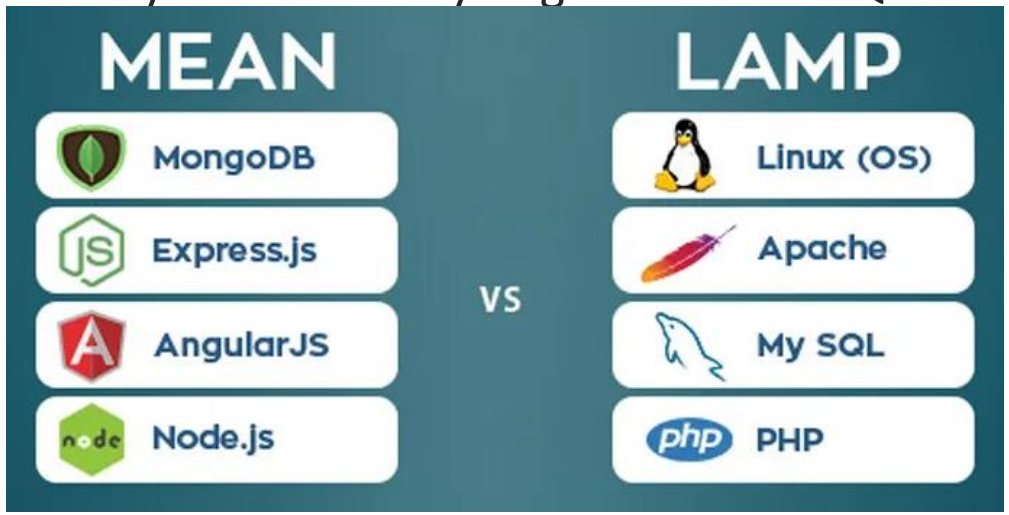


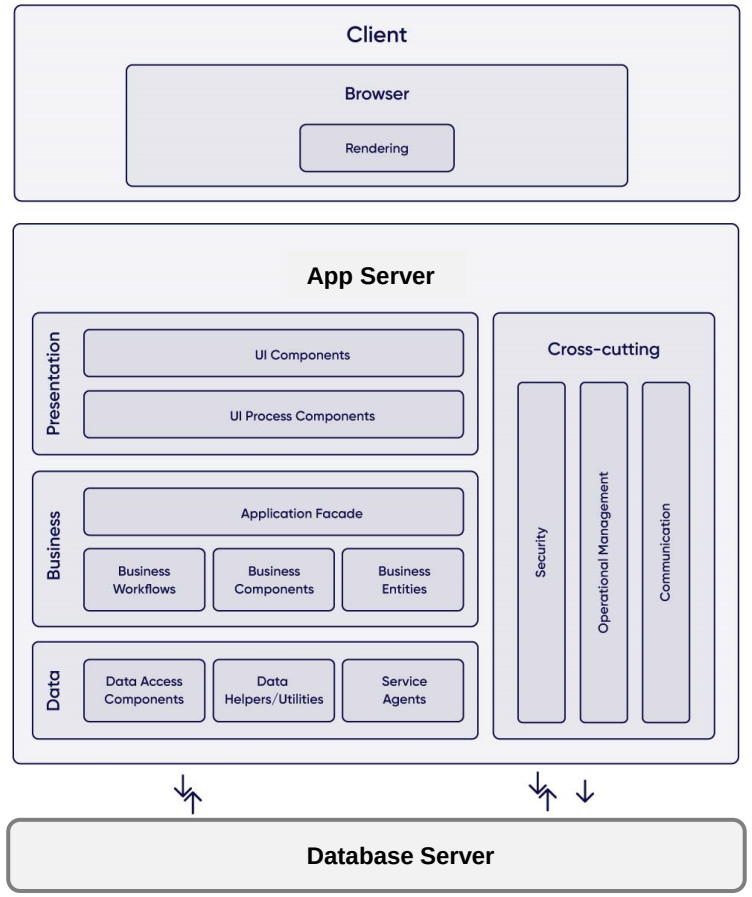
Over provisioning infra capacity



Challenges with Monolith Architecture (3)

- Barrier to adopting new technologies including languages, frameworks, tools etc.
- For example, if your monolith app is written in Java, then you can't easily migrate to Python.
- Similarly, if the app is written to use a relational database, then you can't easily migrate to a NoSQL database.





Q&A

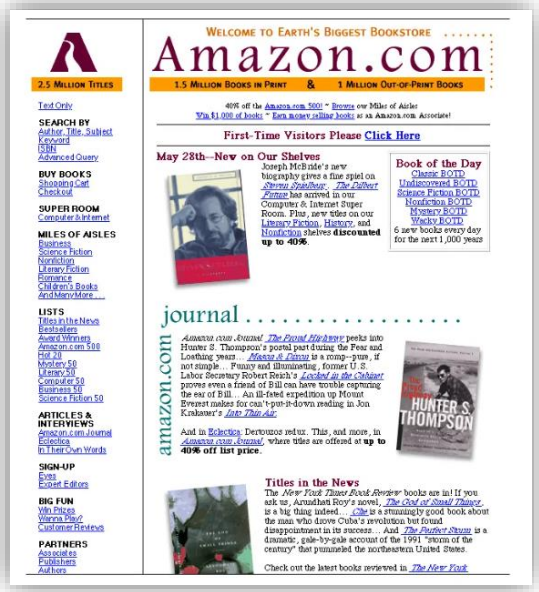
Time Check

Amazon & AWS

The story behind Amazon's architecture evolution

The Amazon.com Story (1)

- Amazon.com started in 1995 as one page rendering application (Obidos) written as C++ binary.
- By 2001, it became clear that the frontend application couldn't scale anymore.
- Amazon made a big architectural change in early 2000s to move from a two-tier monolith to a fully-distributed, decentralized, services platform serving many different applications.



1995

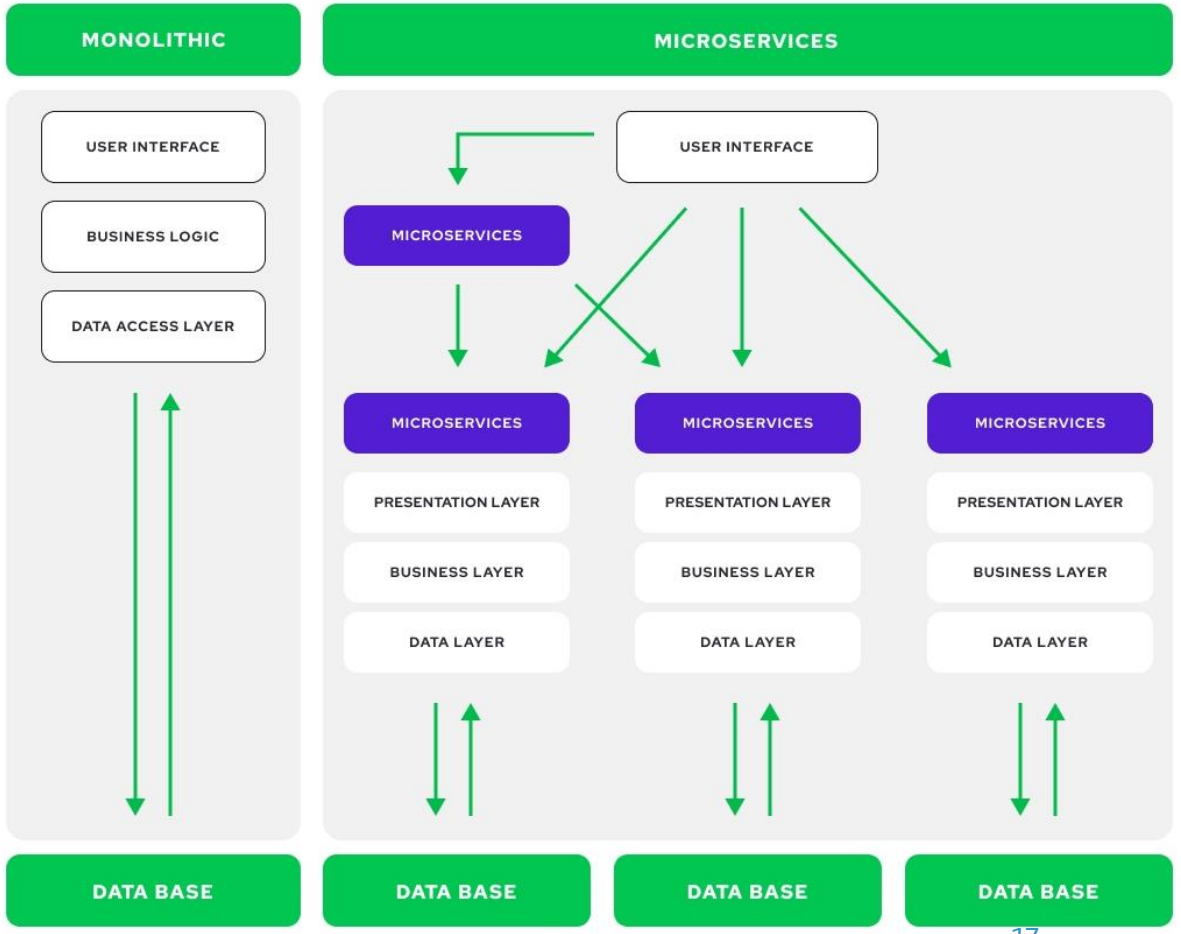
1998

2000

The Amazon.com Story (2)

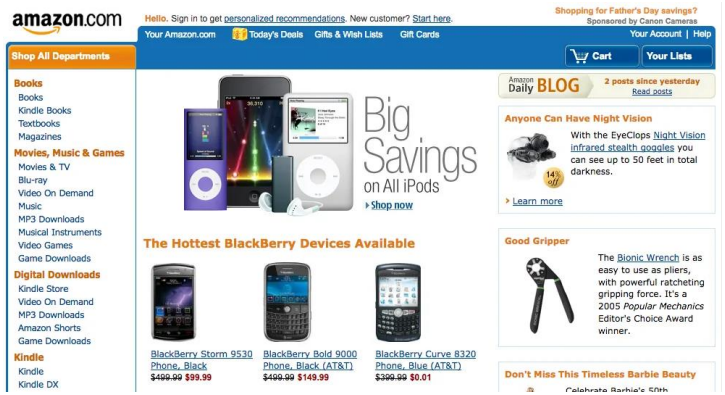
- Starting 2002, the databases were split into small parts and around each part and created a services interface that was the only way to access the data.

- The architecture evolved from a monolith application to a few core applications that aggregate data from smaller downstream services.



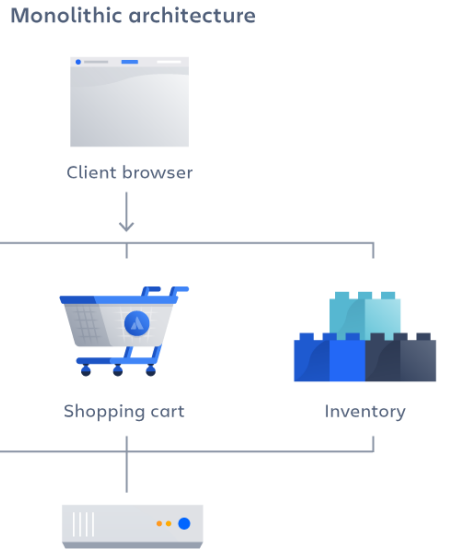
The Amazon.com Story (3)

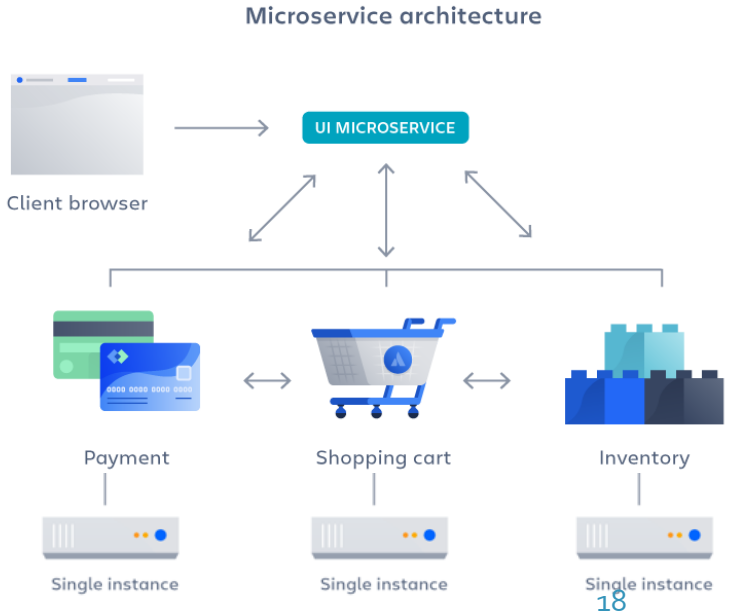
- The application that renders Amazon.com web pages is one such core application.
- Other core applications include the shopping basket, customer service portal, and the seller interface.



2009

2015





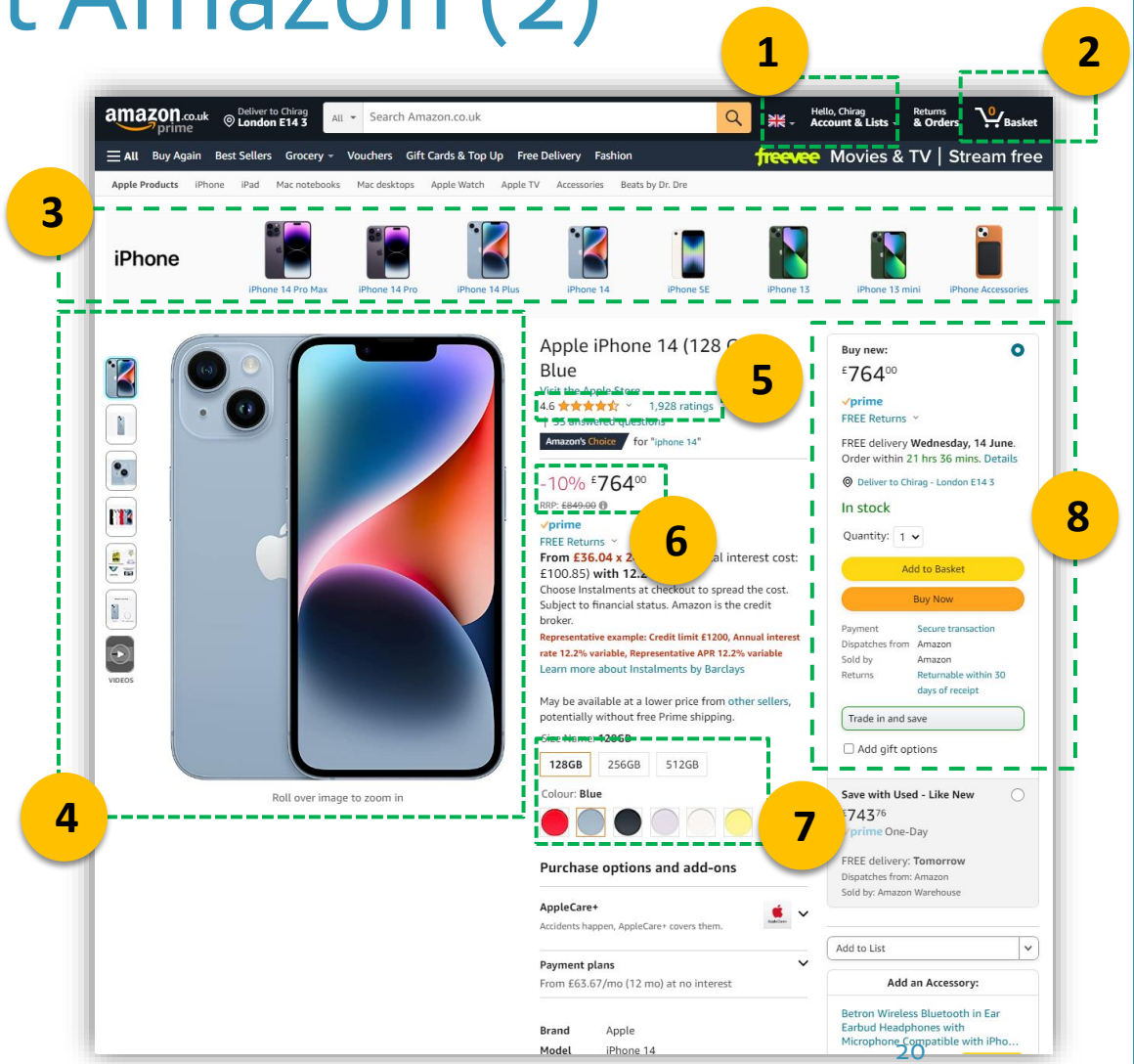
SOA & Microservices at Amazon (1)

- Amazon adopted service oriented architecture (SOA) which is a practice of making software components reusable via service interfaces.
- Microservices architecture goes further to make reusable components smaller and simpler.
- For example, to deliver a **Product Page** on Amazon.com, dozens of microservices are invoked to build discrete portions of the page.

SOA & Microservices at Amazon (2)

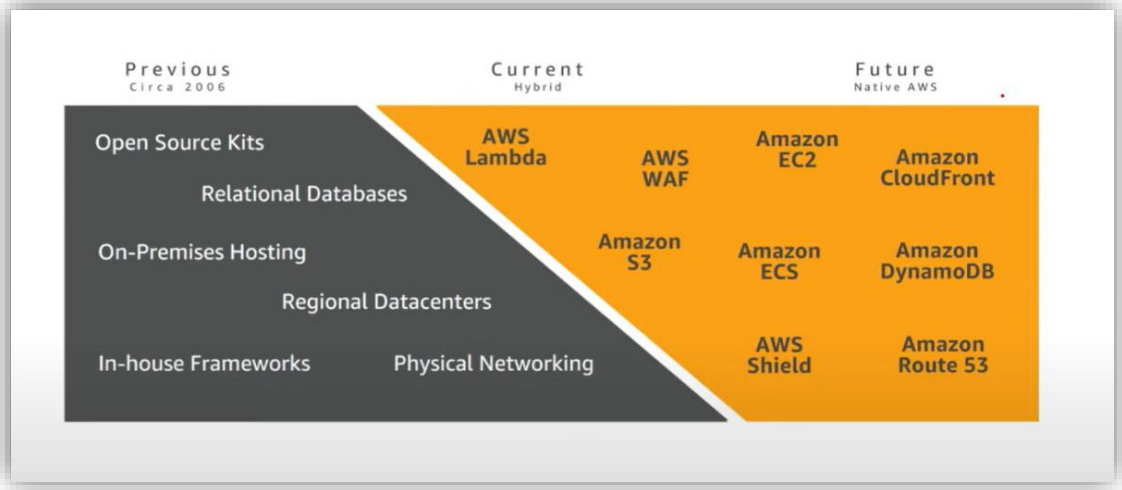
- While there are a few microservices that must be available to provide the price and the product description (must have), the vast majority of content on the page can simply be excluded if the microservice isn't available.

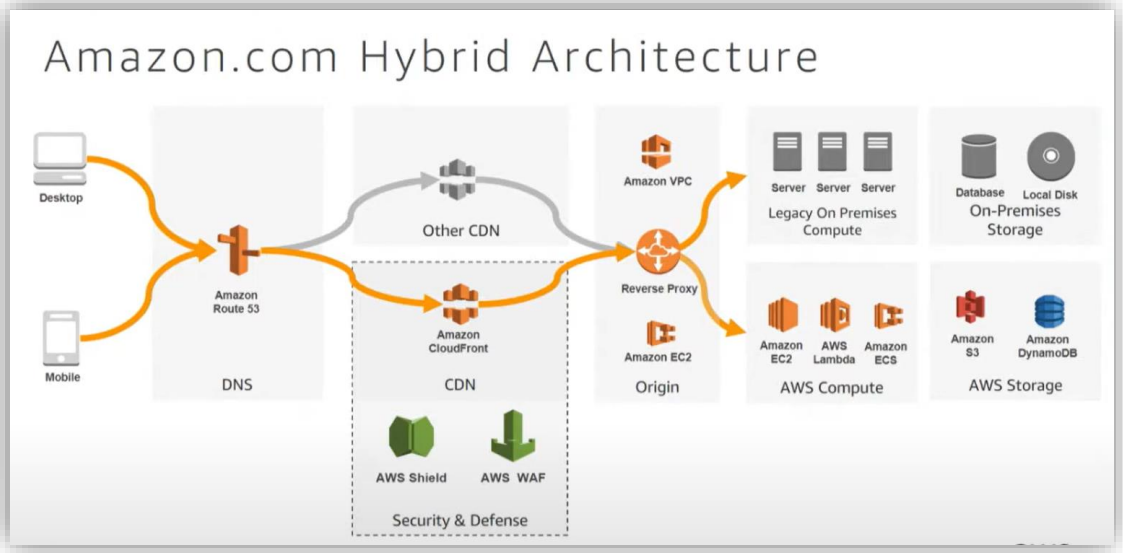
- One could argue that the photo catalogue and customer reviews are not blocking features for the customer to buy a product (nice to have).



Amazon's Infrastructure

- Amazon.com can be classified as an "*Internet Scale*" business
- 45+ billion requests per day
- 18+ petabytes of content served per month
- Hosted in ~20 countries and delivers to 100+ countries via 150+ fulfilment centres





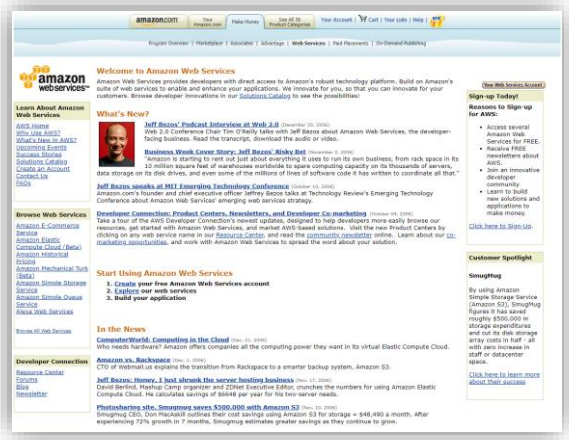
How Amazon created AWS

- **2003:** In 2003, Chris Pinkham and Benjamin Black (Amazon.com South Africa) presented a paper on how Amazon's own internal infrastructure should look like. They suggested to sell it as a service and presented a 6 page narrative (business case) to Jeff Bezos.

- **2004:** Amazon SQS (Message Queues) was launched as a beta service.

- **2006:** Amazon Web Services was officially launched with EC2 (Virtual Machines) and S3 (Storage).

- **2007:** Over 180,000 developers had signed up for AWS.



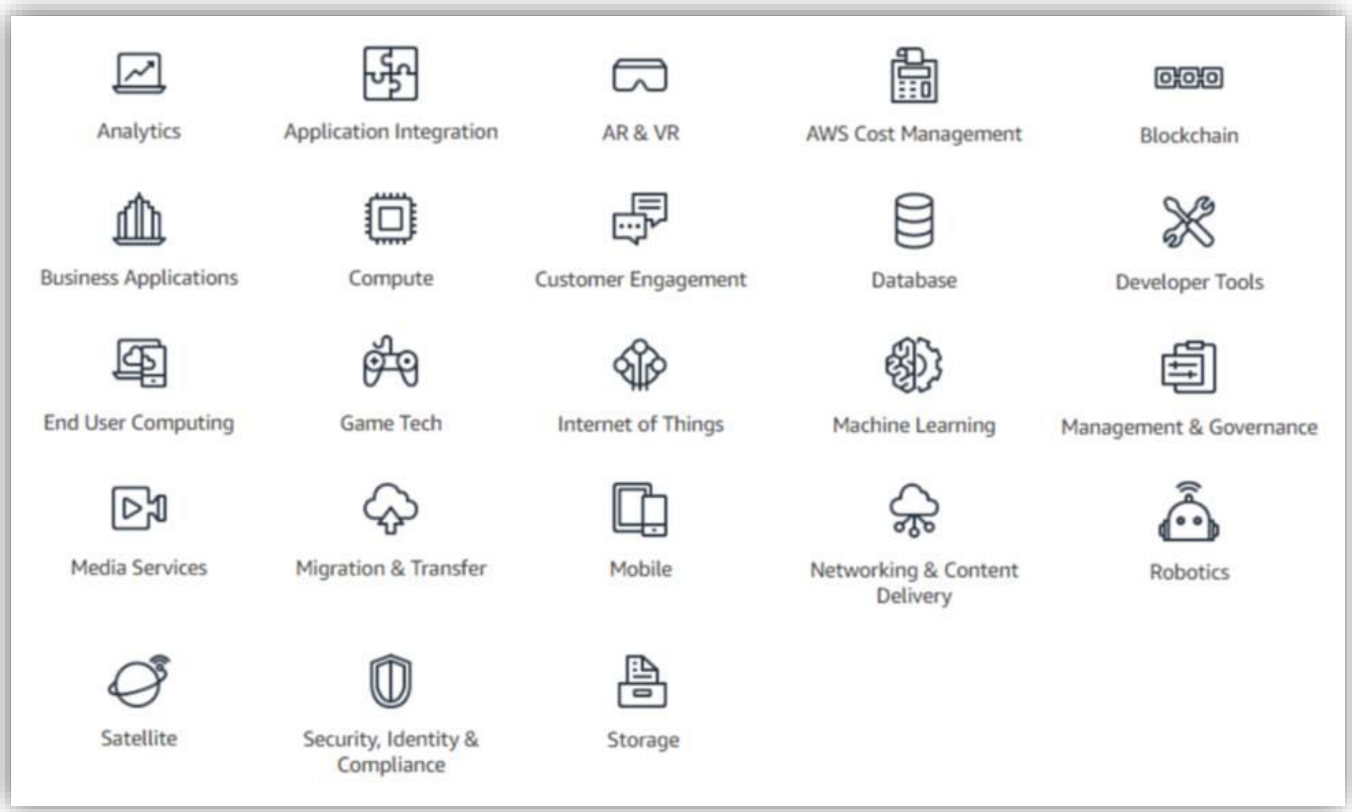
2006

2012

The state of AWS today

As of 2024, AWS comprises of 250+ cloud services across 23 categories

During the 2015 re:Invent keynote, AWS disclosed that they have more than a million active customers every month in 190 countries, including 2000+ government agencies, 5000+ education institutions and 17,500+ non-profits.



34 launched Regions
each with multiple Availability Zones

108 Availability Zones

600+ CloudFront POPs
and 13 Regional edge caches

AWS Global Infrastructure Map

The AWS Cloud spans 108 Availability Zones within 34 geographic regions, with announced plans for 18 more Availability Zones and six more AWS Regions in Mexico, New Zealand, the Kingdom of Saudi Arabia, Thailand, Taiwan, and the AWS European Sovereign Cloud.

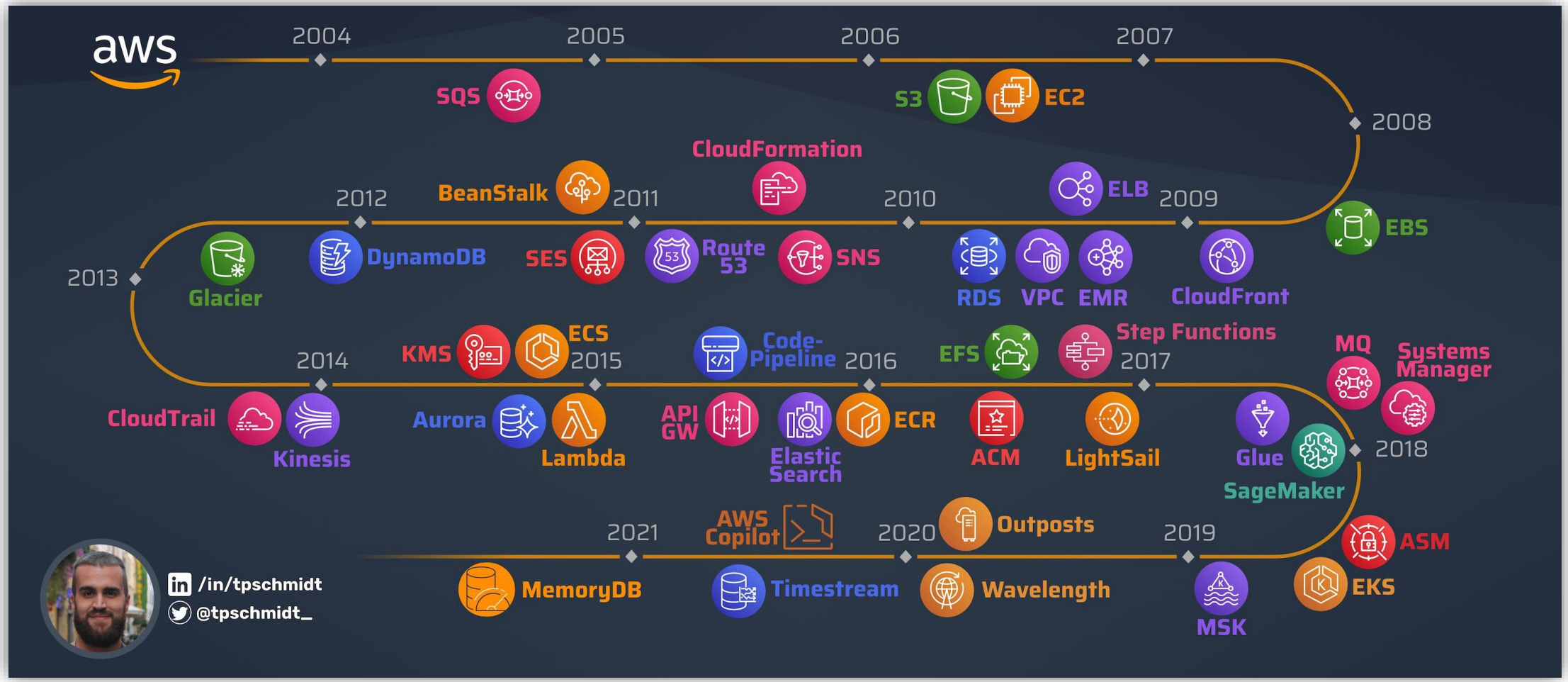
List view

Regions Coming soon

Early adopters

- Netflix
- Snap
- Pinterest
- Zoom
- NASA
- Apple
- LinkedIn
- Instagram
- Deliveroo
- Airbnb
- Uber

Timeline of AWS service releases



Q&A

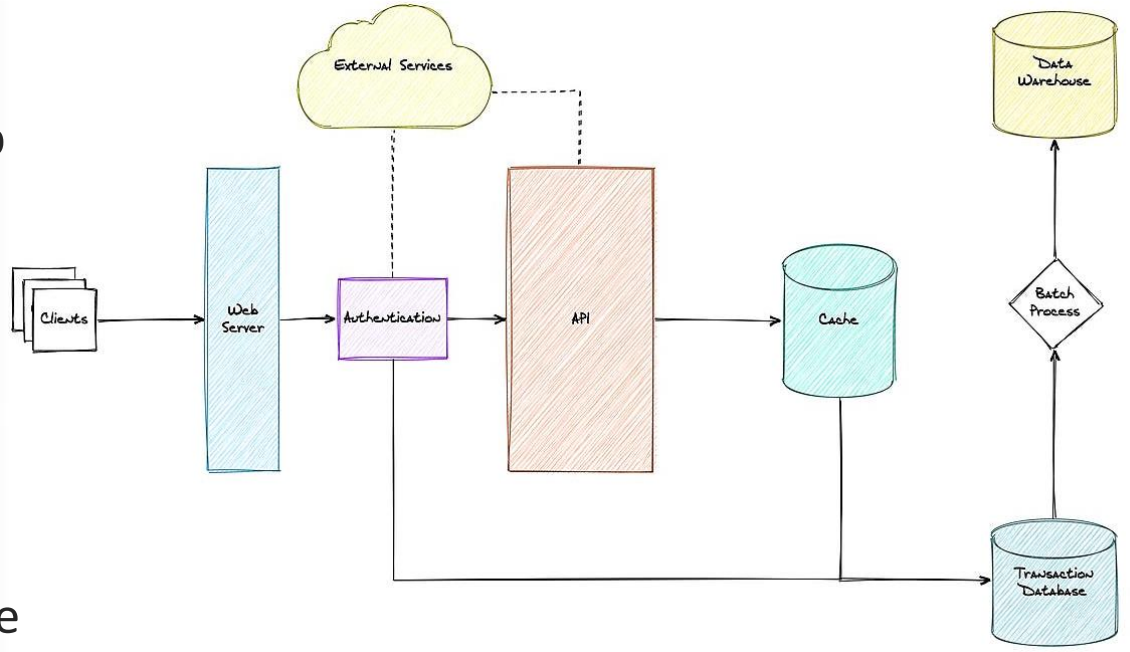
Time Check

System Design Case Study

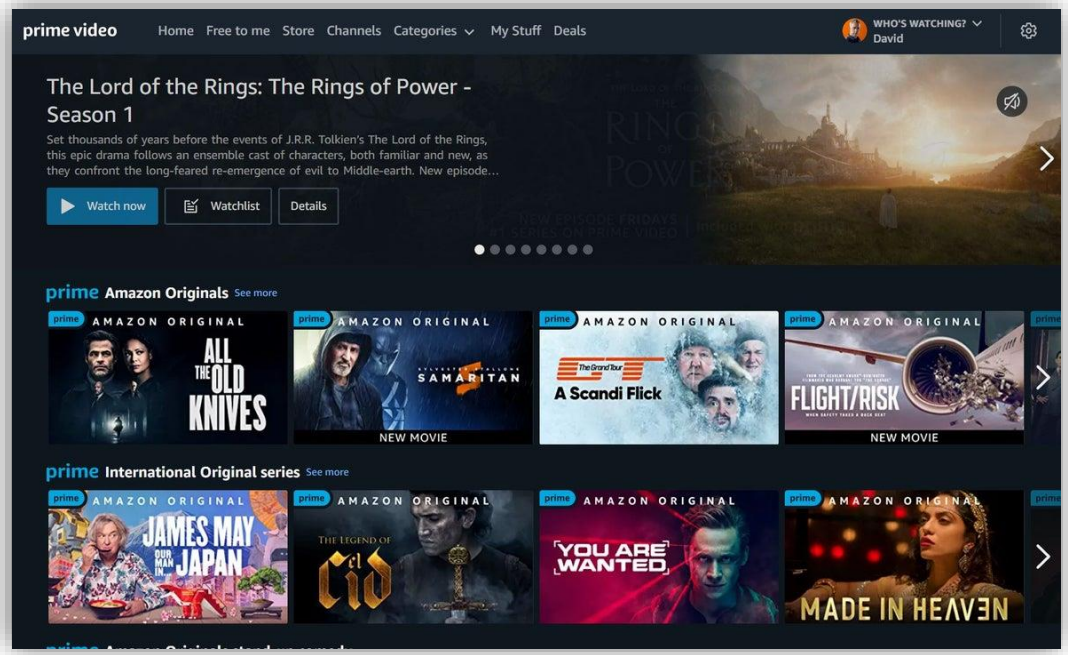
Techniques and steps for developing architecture

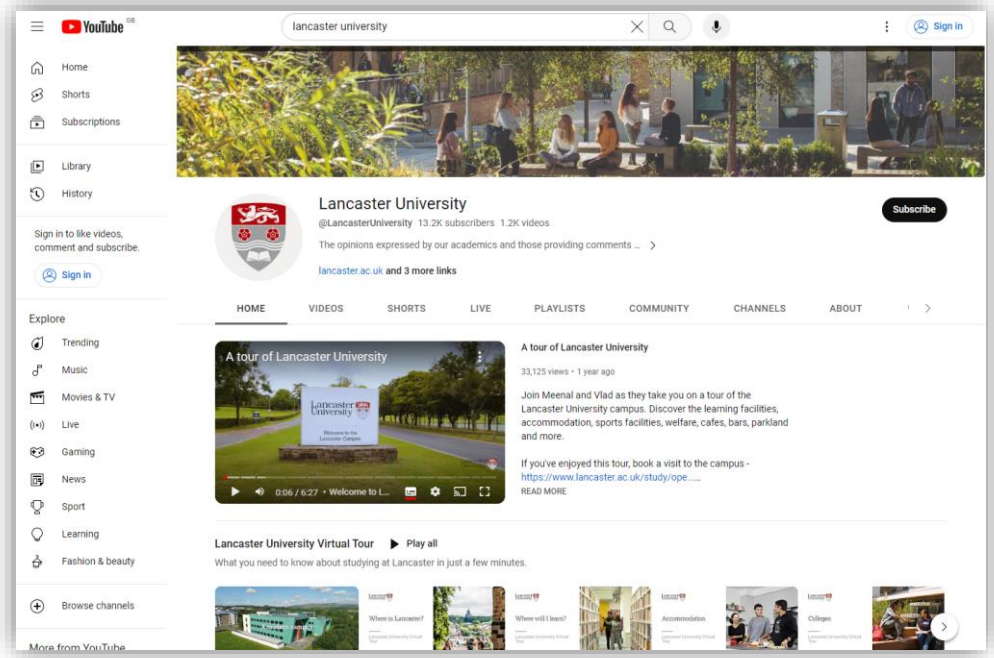
System Design (Recap)

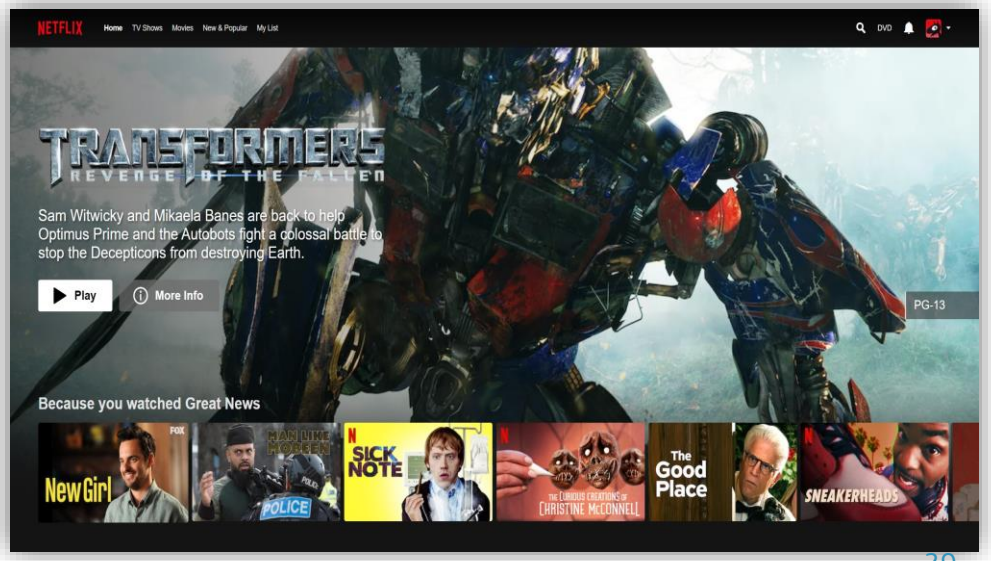
- System design is the process of defining the components, interfaces, and data for a new web service
- After requirements are determined, we can begin to build them into a system design that addresses the functional and non-functional needs of users.
- A good system design requires us to think about everything in an architecture, from the hardware and software, all the way down to the data and how it's stored.



Video Streaming Service







Video Streaming Service (Requirements)

Functional Requirements

- View videos and video catalogue on multiple devices
- Upload videos and transcode videos in the background
- Videos must be available 24x7

Design Decisions

- Cross-device UI built as a Single Page Application (SPA)
- Asynchronous processing / background workers
- Highly reliable storage for videos (i.e. no data loss)

Non-functional Requirements

- Low latency streaming (i.e. negligible buffering)
- High availability (i.e. eventual consistency)

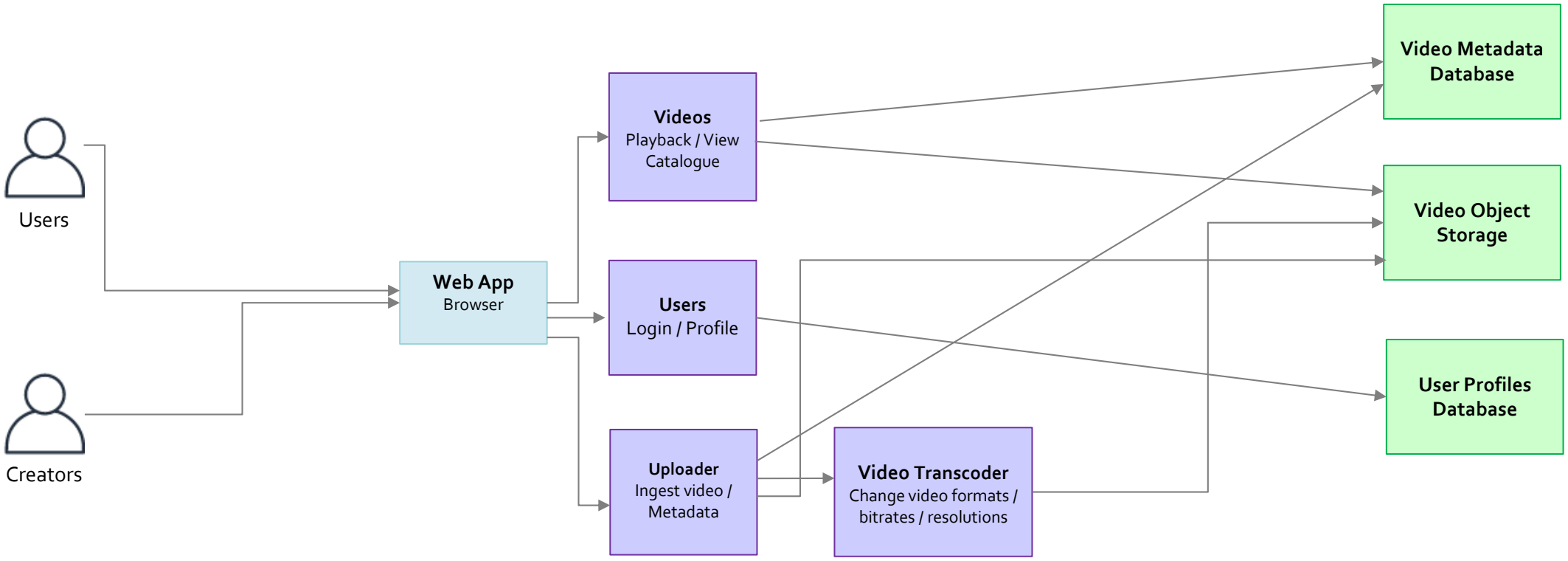
- CDNs and caching will be important to reduce video buffering
- Load balancer and auto-scaling horizontally to achieve fault tolerance (no single points of failure)

Assumptions

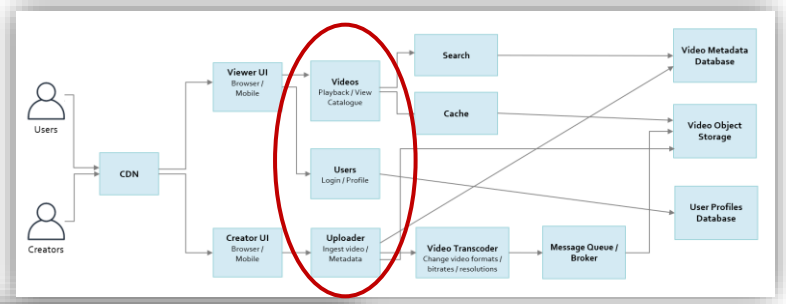
- 50k to 75k Monthly Active Users
- Average Monthly uploads: 1k to 2k
- More video views than uploads (reads >> writes).
- Average video runtime will be >15 minutes

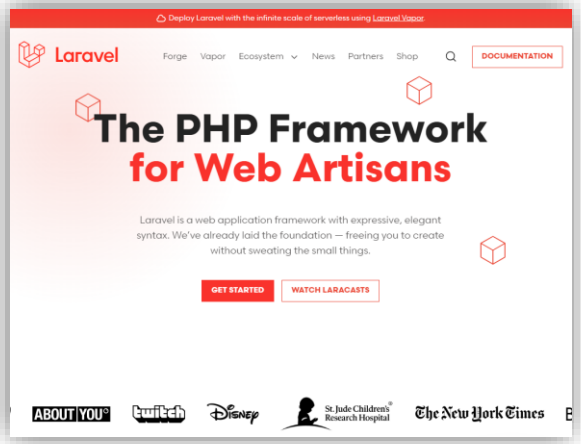
- Few primary databases. Many read-replicas.
- Live streaming not required
- Key value database to store metadata (e.g. video title, description, genre, tags, cover image).

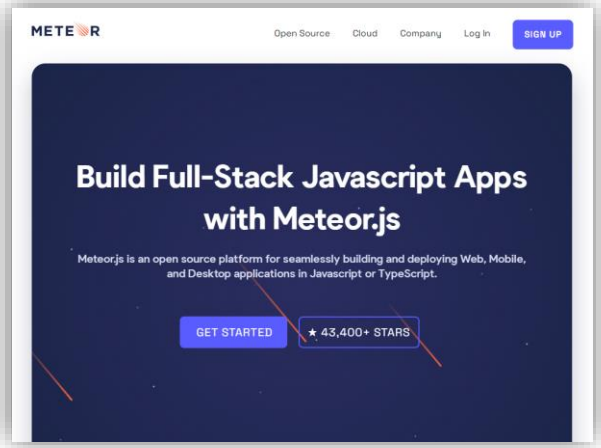
Video Streaming Service (Level 100)

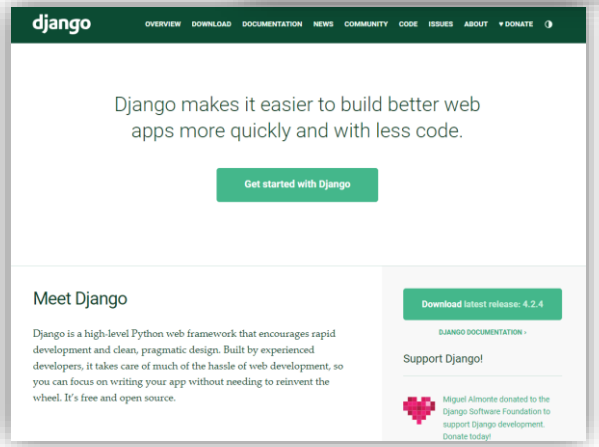


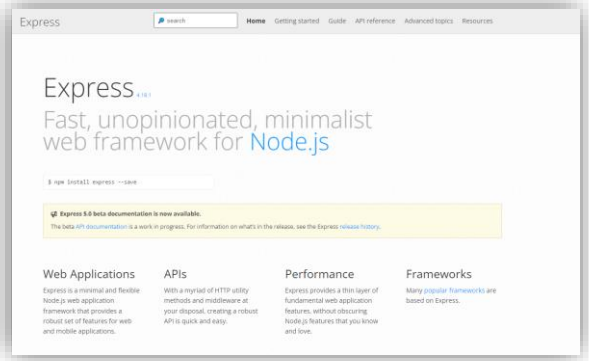
Web Application Frameworks



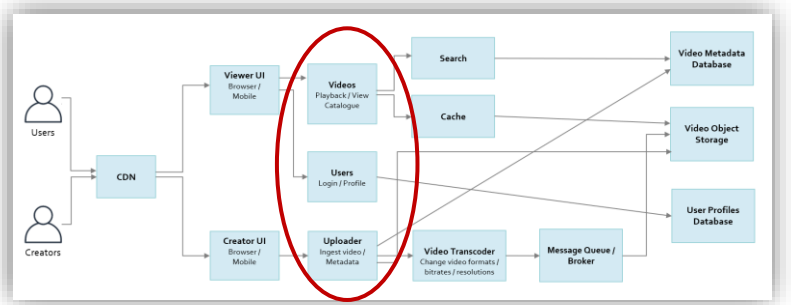




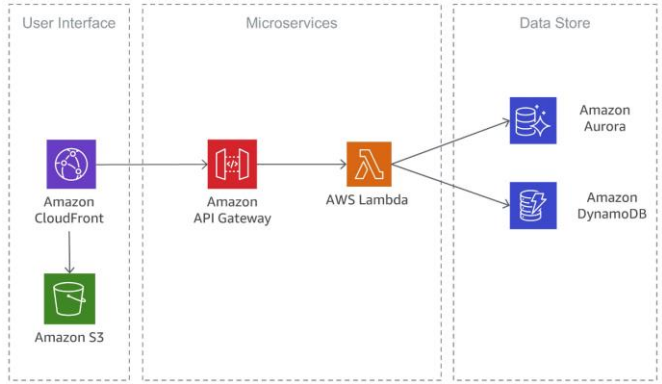


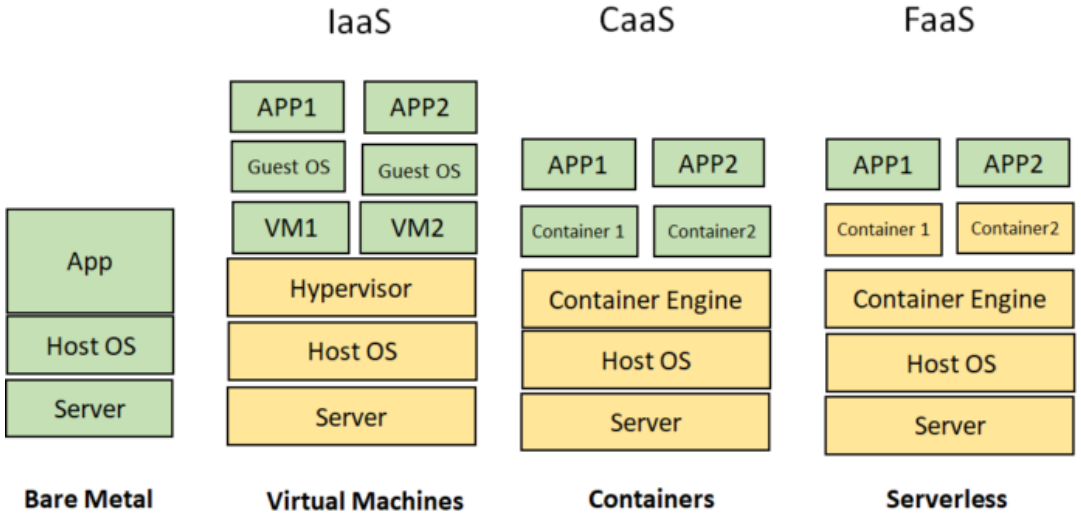


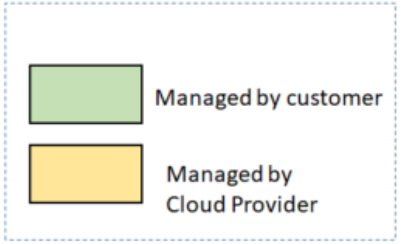
Compute Options (1)

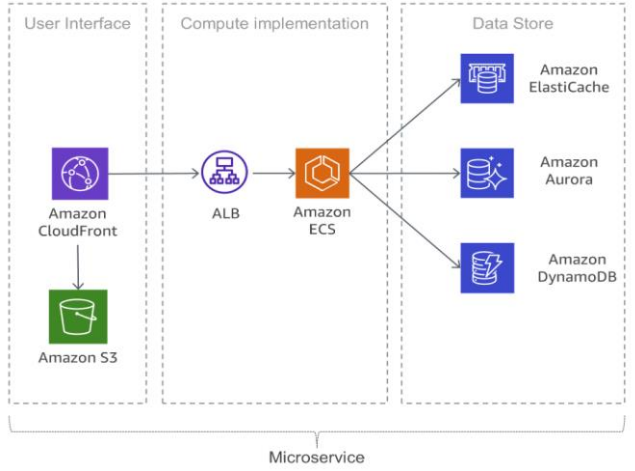


- Microservice application components can be deployed to Virtual Machines (e.g. KVM), Containers (e.g. Docker) or as Serverless (e.g. Cloud Functions)
- Container orchestration (e.g. Kubernetes) is useful for managing a complex microservice architecture with hundreds of containers.

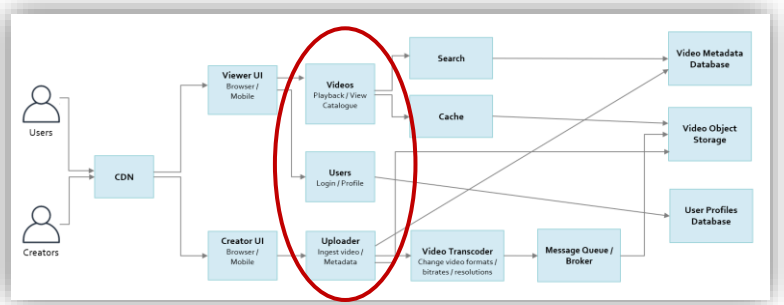






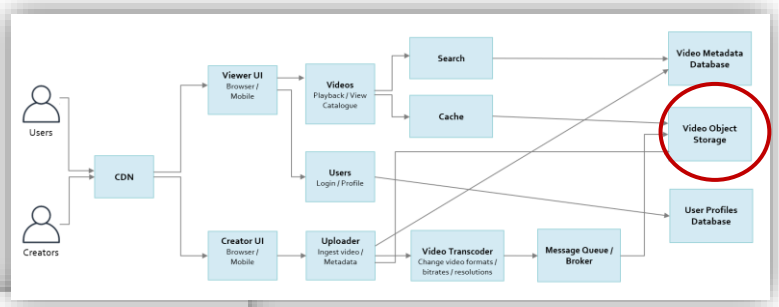


Compute Options (2)



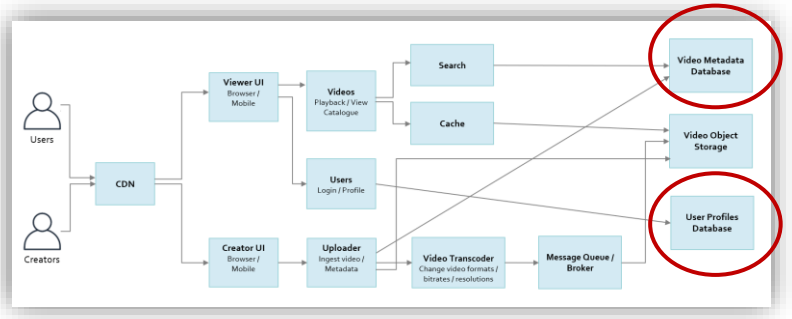
Category	AWS service
Instances (virtual machines)	• **Amazon Elastic Compute Cloud** (Amazon EC2) — Secure and resizeable compute capacity (virtual servers) in the cloud • **Amazon EC2 Spot Instances** — Run fault-tolerant workloads for up to 90% off • **Amazon EC2 Auto Scaling** — Automatically add or remove compute capacity to meet changes in demand • **Amazon Lightsail** — Easy-to-use cloud platform that offers you everything you need to build an application or website • **AWS Batch** — Fully managed batch processing at any scale
Containers	• **Amazon Elastic Container Service** **(Amazon ECS)** — Highly secure, reliable, and scalable way to run containers • **Amazon ECS Anywhere** — Run containers on customer-managed infrastructure • **Amazon Elastic Container Registry** (Amazon ECR) — Easily store, manage, and deploy container images • **Amazon Elastic Kubernetes Service** (Amazon EKS) — Fully managed Kubernetes service • **Amazon EKS Anywhere** — Create and operate Kubernetes clusters on your own infrastructure • **AWS Fargate** — Serverless compute for containers • **AWS App Runner** — Build and run containerized applications on a fully managed service
Serverless	• **AWS Lambda** — Run code without thinking about servers. Pay only for the compute time you consume.
Edge and hybrid	• **AWS Outposts** — Run AWS infrastructure and services on premises for a truly consistent hybrid experience • **AWS Snow Family** — Collect and process data in rugged or disconnected edge environments • **AWS Wavelength** — Deliver ultra-low latency application for 5G devices • **VMware Cloud on AWS** — Preferred service for all vSphere workloads to rapidly extend and migrate to the cloud • **AWS Local Zones** — Run latency sensitive applications closer to end-users
Cost and capacity management	• **AWS Savings Plan** — Flexible pricing model that provides savings of up to 72% on AWS compute usage • **AWS Compute Optimizer** — Recommends optimal AWS compute resources for your workloads to reduce costs and improve performance • **AWS Elastic Beanstalk** — Easy-to-use service for deploying and scaling web applications and services • **EC2 Image Builder** — Build and maintain secure Linux or Windows Server images • **Elastic Load Balancing** (ELB) — Automatically distribute incoming application traffic across multiple targets

Storage Options

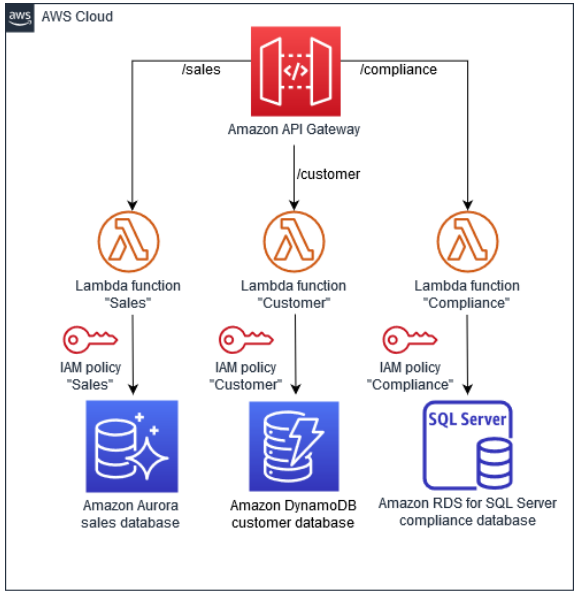


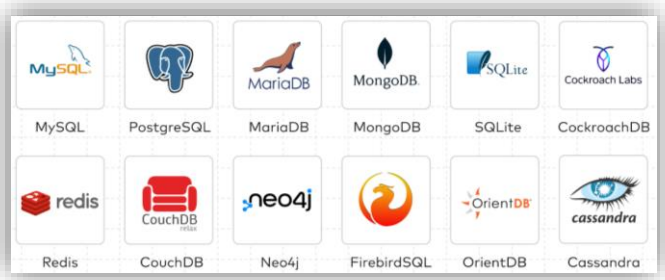
Storage type	What is it optimized for?	Storage services or tools
Block	Applications requiring low-latency, high-performance durable storage attached to single EC2 instances or containers, such as databases and general-purpose local instance storage.	Amazon EBS Amazon Elastic Compute Cloud (Amazon EC2) instance store
File system	Applications and workloads requiring shared read and write access across multiple EC2 instances/containers or from multiple on-prem servers, such as team file shares, highly-available enterprise applications, analytics workloads, and ML training.	Amazon Elastic File System (Amazon EFS) Amazon FSx Amazon FSx for Lustre Amazon FSx for NetApp ONTAP Amazon FSx for OpenZFS Amazon FSx for Windows File Server AWS Storage Gateway
Object	Read-heavy workloads such as content distribution, web hosting, big data analytics, and ML workflows. Well-suited for scenarios where data needs to be stored, accessed, and distributed globally over the internet.	Amazon S3

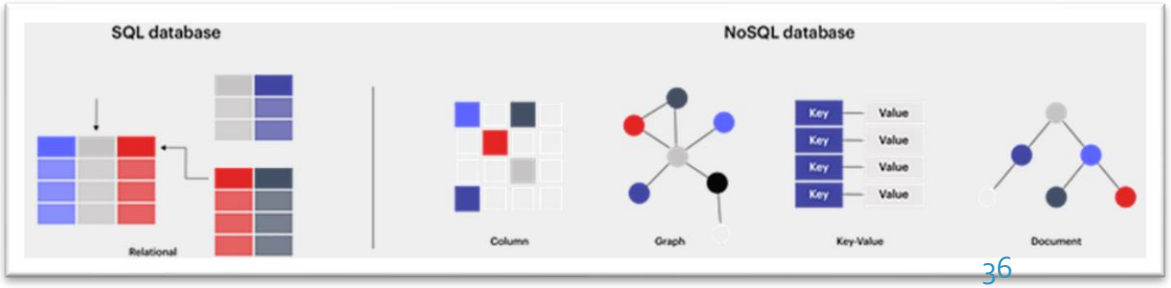
Database Options (1)



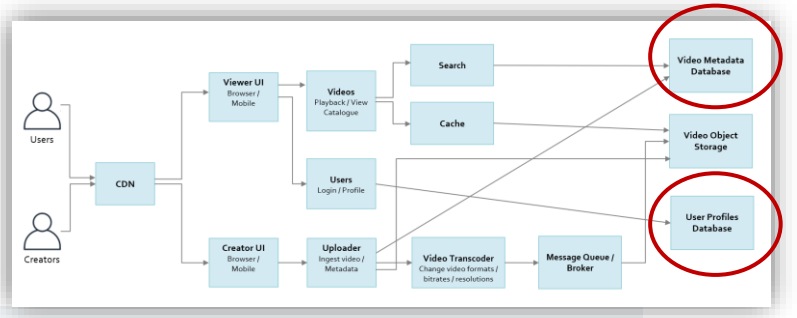
- Databases are used to persist data needed by the microservices. Each microservice can use a purpose-built database as databases are not typically shared outside microservice boundaries.
- Relational databases are still very popular to store structured data and business objects.
- Popular stores for session data are in-memory databases such as Memcached or Redis.
- NoSQL databases have been designed to favour scalability, performance, and availability over the consistency of relational databases.





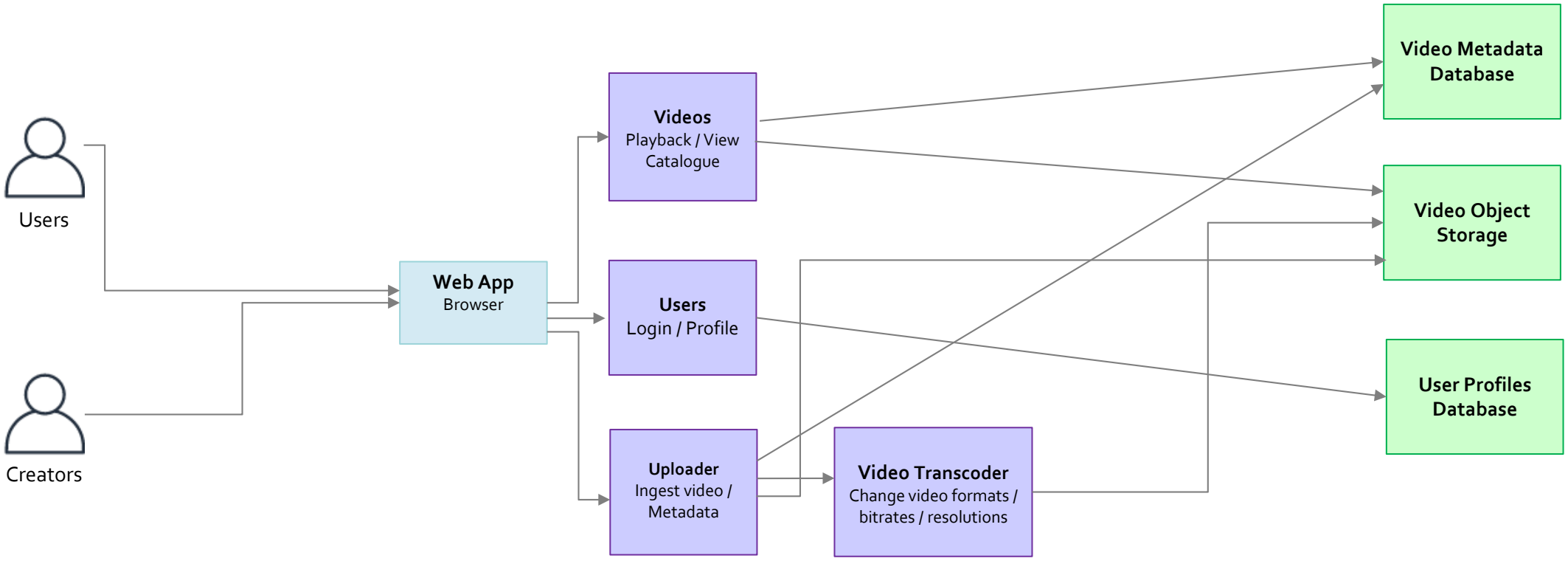


Database Options (2)

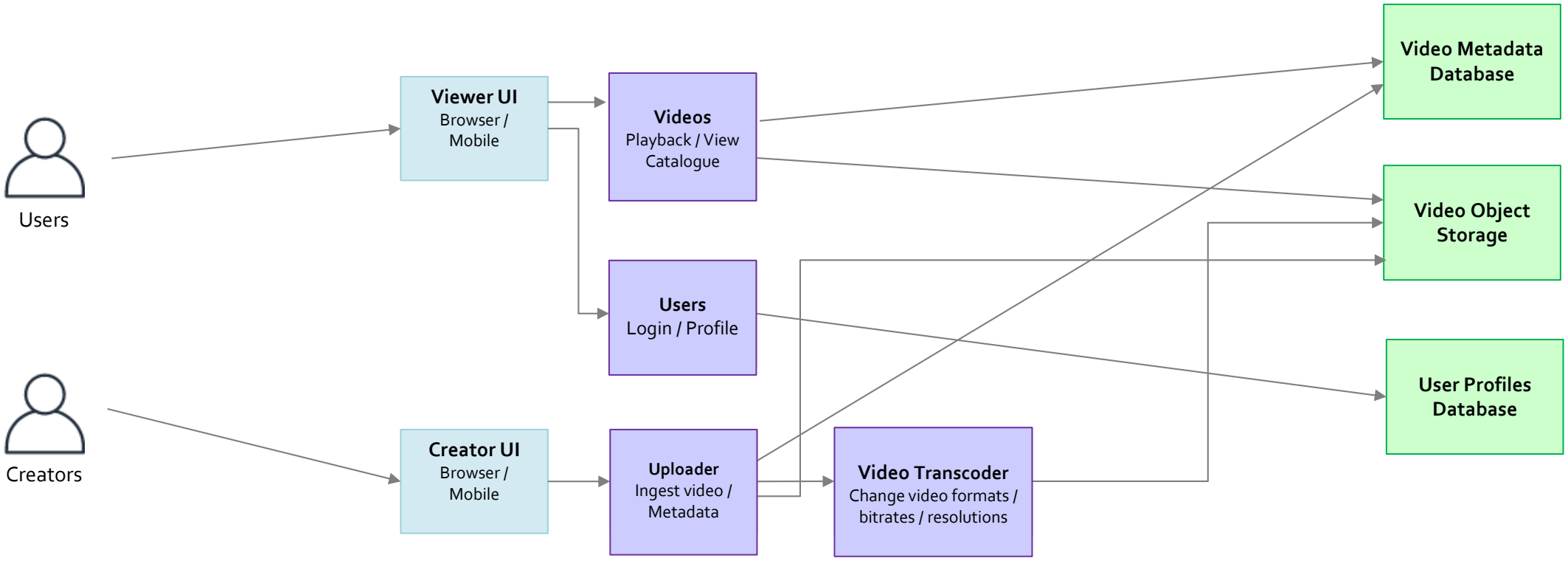


Database type	Examples	AWS service
Relational	Traditional applications, enterprise resource planning (ERP), customer relationship management (CRM), ecommerce	Amazon Aurora, Amazon RDS, Amazon Redshift
Key-value	High-traffic web applications, ecommerce systems, gaming applications	Amazon DynamoDB
In-memory	Caching, session management, gaming leaderboards, geospatial applications	Amazon ElastiCache, Amazon MemoryDB for Redis
Document	Content management, catalogs, user profiles	Amazon DocumentDB (with MongoDB compatibility)
Wide column	High-scale industrial apps for equipment maintenance, fleet management, and route optimization	Amazon Keyspaces
Graph	Fraud detection, social networking, recommendation engines	Amazon Neptune
Time series	Internet of Things (IoT) applications, DevOps, industrial telemetry	Amazon Timestream
Ledger	Systems of record, supply chain, registrations, banking transactions	Amazon Ledger Database Services (QLDB)

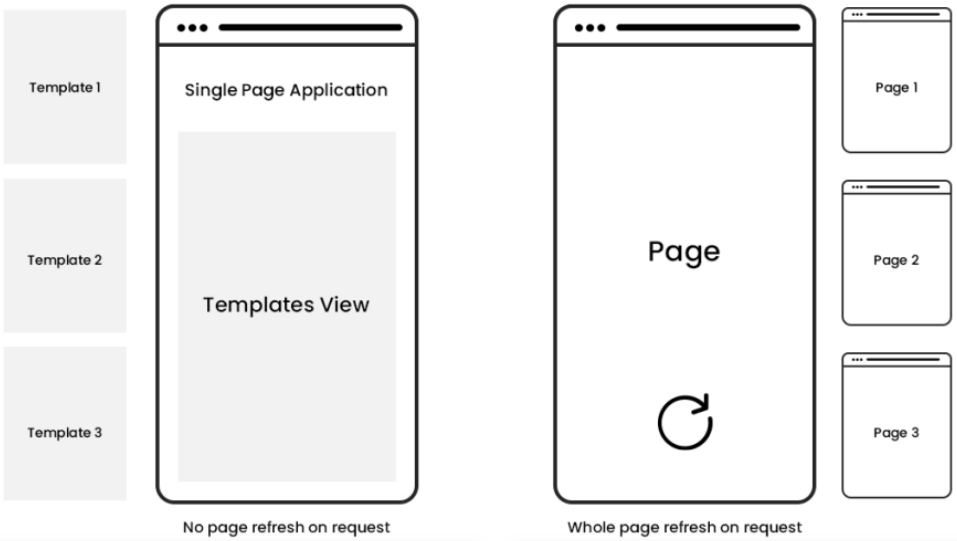
Video Streaming Service (Level 100 Recap)

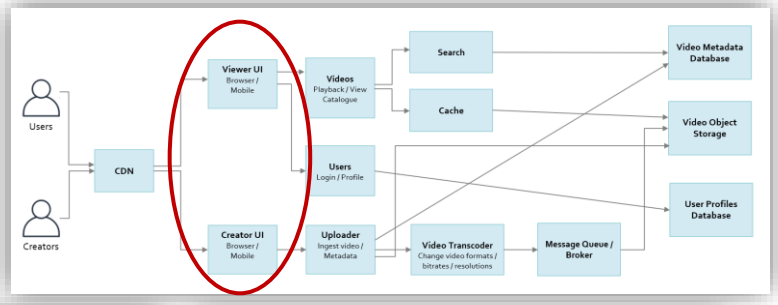


Video Streaming Service (Level 200)

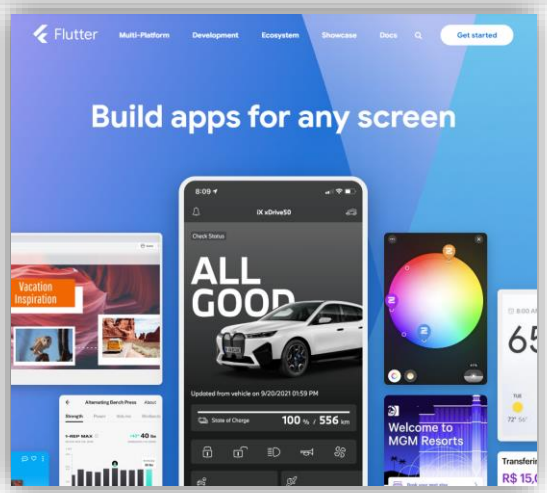


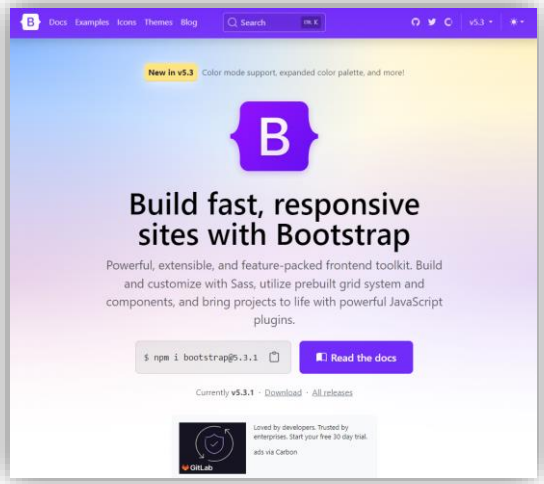
UI Options


Template 1
Template 2
Template 3
Single Page Application
Templates View
No page refresh on request
Page
Page 1
Page 2
Page 3
Whole page refresh on request

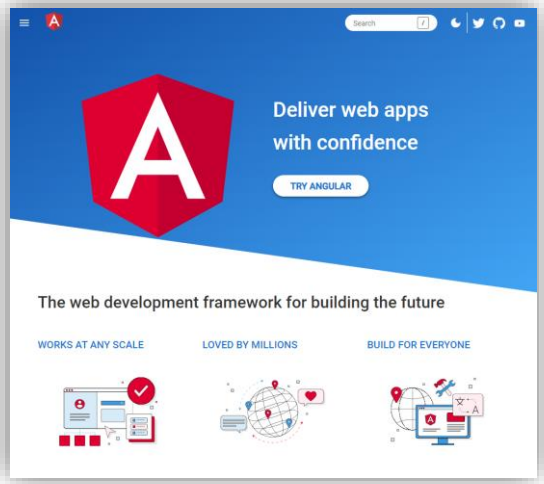

Users
Creators
CDN
Viewer UI
Browser / Mobile
Creator UI
Browser / Mobile
Videos
Playback / View Catalogue
Uploader
Ingest video / Metadata
Users
Login / Profile
Search
Cache
Video Transcoder
Change video formats / bitrates / resolutions
Message Queue / Broker
Video Metadata Database
Video Object Storage
User Profiles Database

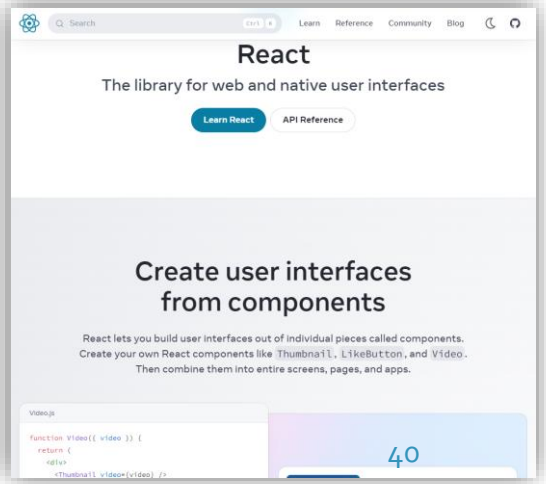

React Native
Development
Contributing
Community
Showcase
Blog
React Native
Learn once, write anywhere.
Get started
Learn the basics ›
Welcome to React
Create native apps for Android, iOS, and more using React
React Native combines the best parts of native development with React, a best-in-class JavaScript library for building user interfaces.
Use a little—or a lot. You can use React Native today in your existing Android and iOS projects or you can create a whole new app from scratch.


Flutter
Multi-Platform
Development
Ecosystem
Showcase
Docs
Get started
Build apps for any screen
ALL GOOD
Vacation Inspiration
Welcome to MGM Resorts

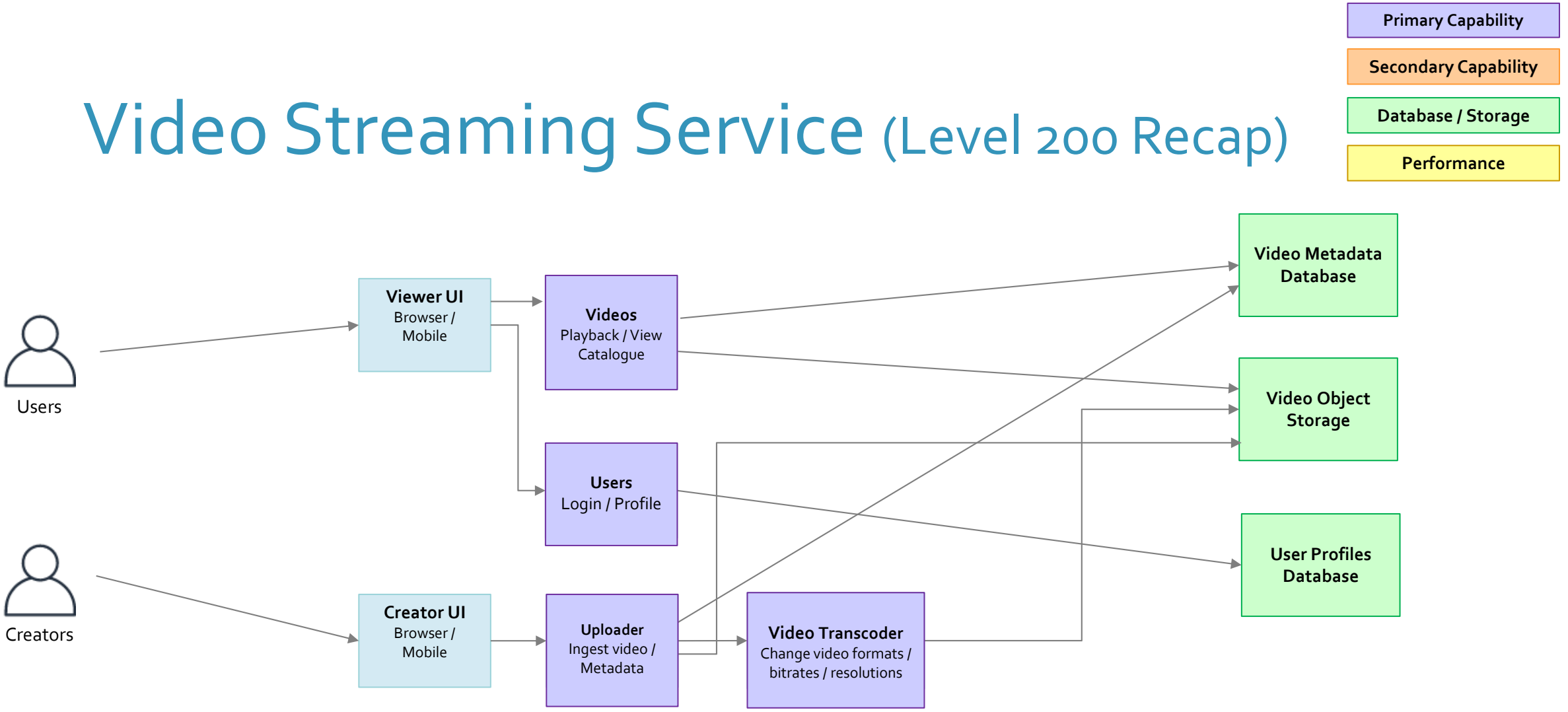

Docs
Examples
Icons
Themes
Blog
New in v5.3
Color mode support, expanded color palette, and more!
Build fast, responsive sites with Bootstrap
Powerful, extensible, and feature-packed frontend toolkit. Build and customize with Sass, utilize prebuilt grid system and components, and bring projects to life with powerful JavaScript plugins.
$ npm i bootstrap@5.3.1
Read the docs


Vue.js
Docs
API
Playground
Ecosystem
About
Sponsor
Partners
The Progressive JavaScript Framework
An approachable, performant and versatile framework for building web user interfaces.
Why Vue
Get Started
Install
Approachable
Performant
Versatile

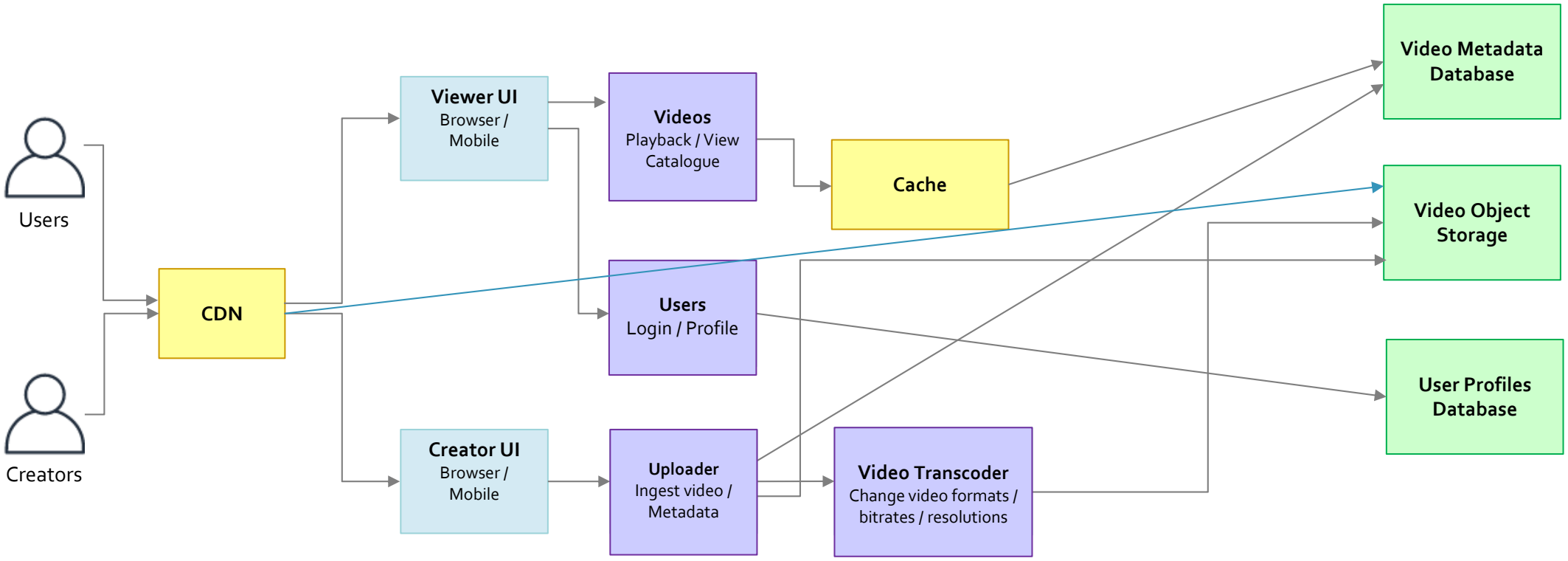

Deliver web apps with confidence
TRY ANGULAR
The web development framework for building the future
WORKS AT ANY SCALE
LOVED BY MILLIONS
BUILD FOR EVERYONE


Learn
Reference
Community
Blog
React
The library for web and native user interfaces
Learn React
API Reference
Create user interfaces from components
React lets you build user interfaces out of individual pieces called components.
Create your own React components like Thumbnail, LikeButton, and Video.
Then combine them into entire screens, pages, and apps.

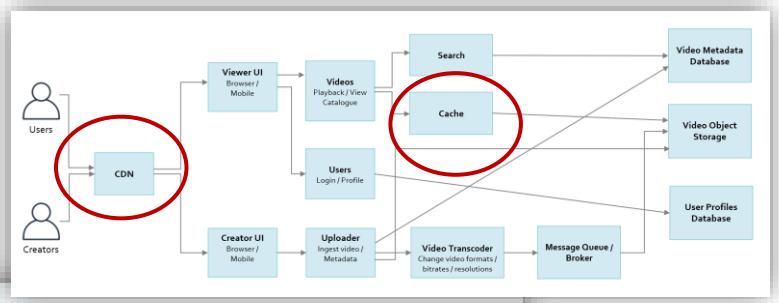
Video Streaming Service (Level 200 Recap)

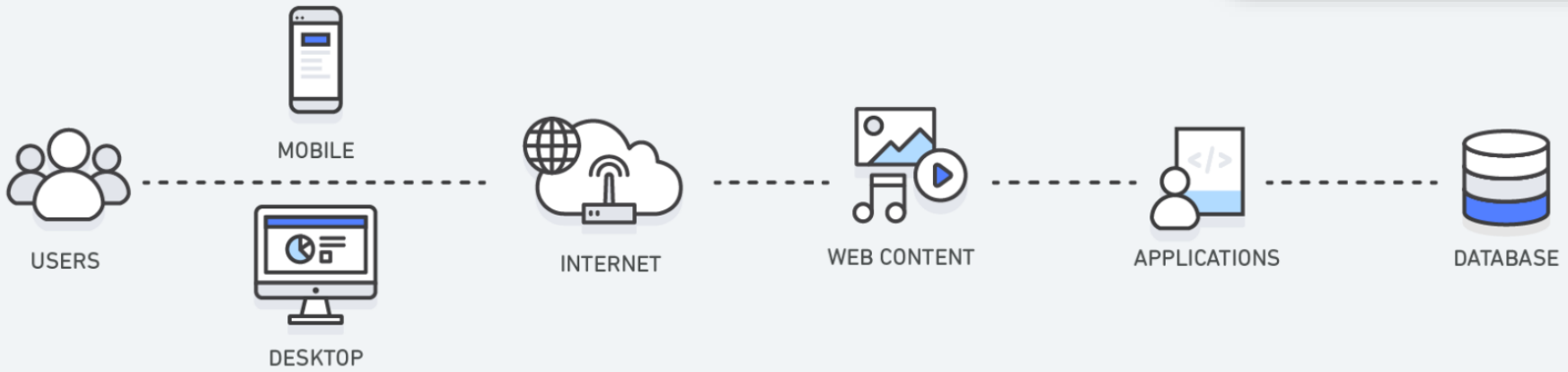


Video Streaming Service (Level 300)



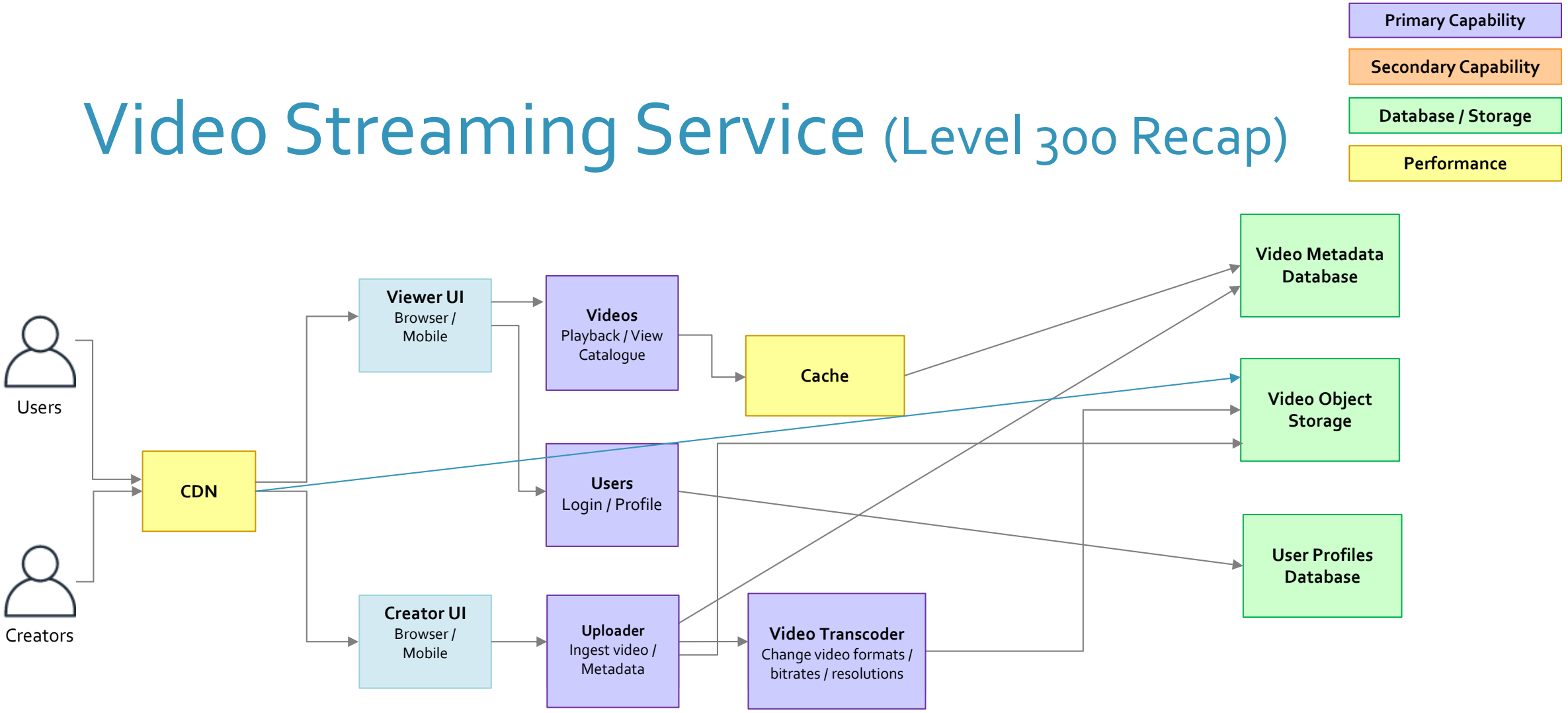
Caching Options



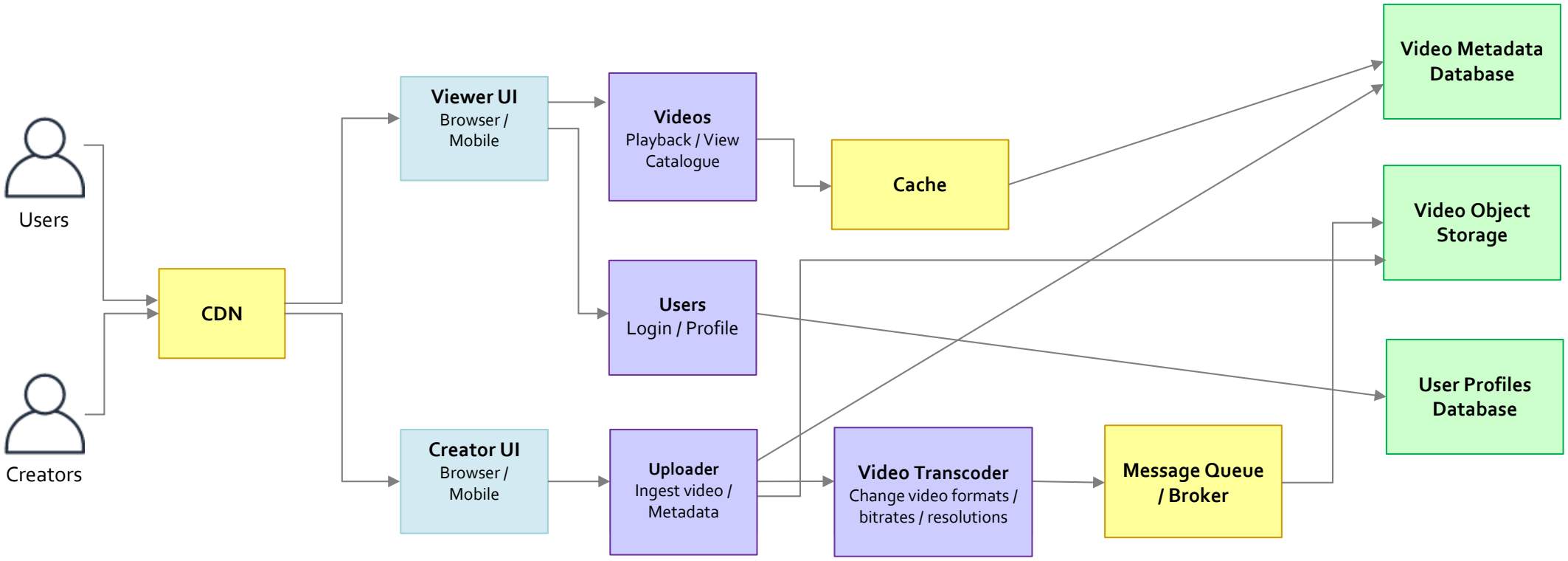


Layer	Client-Side	DNS	Web	App	Database
Use Case	Accelerate retrieval of web content from websites (browser or device)	Domain to IP Resolution	Accelerate retrieval of web content from web/app servers. Manage Web Sessions (server side)	Accelerate application performance and data access	Reduce latency associated 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

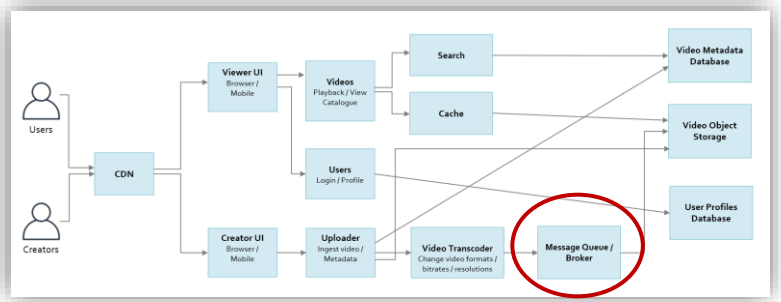
Video Streaming Service (Level 300 Recap)



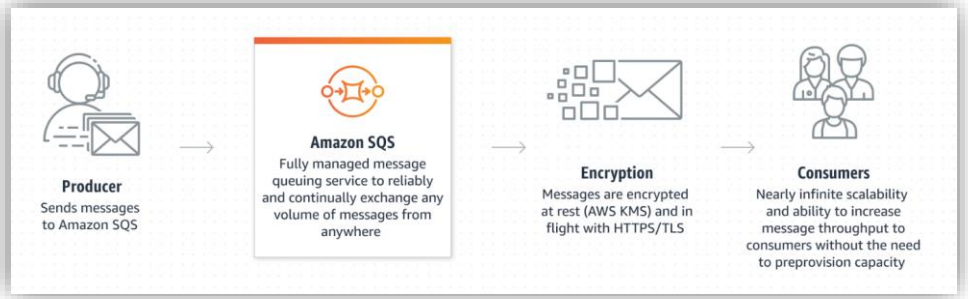
Video Streaming Service (Level 400)



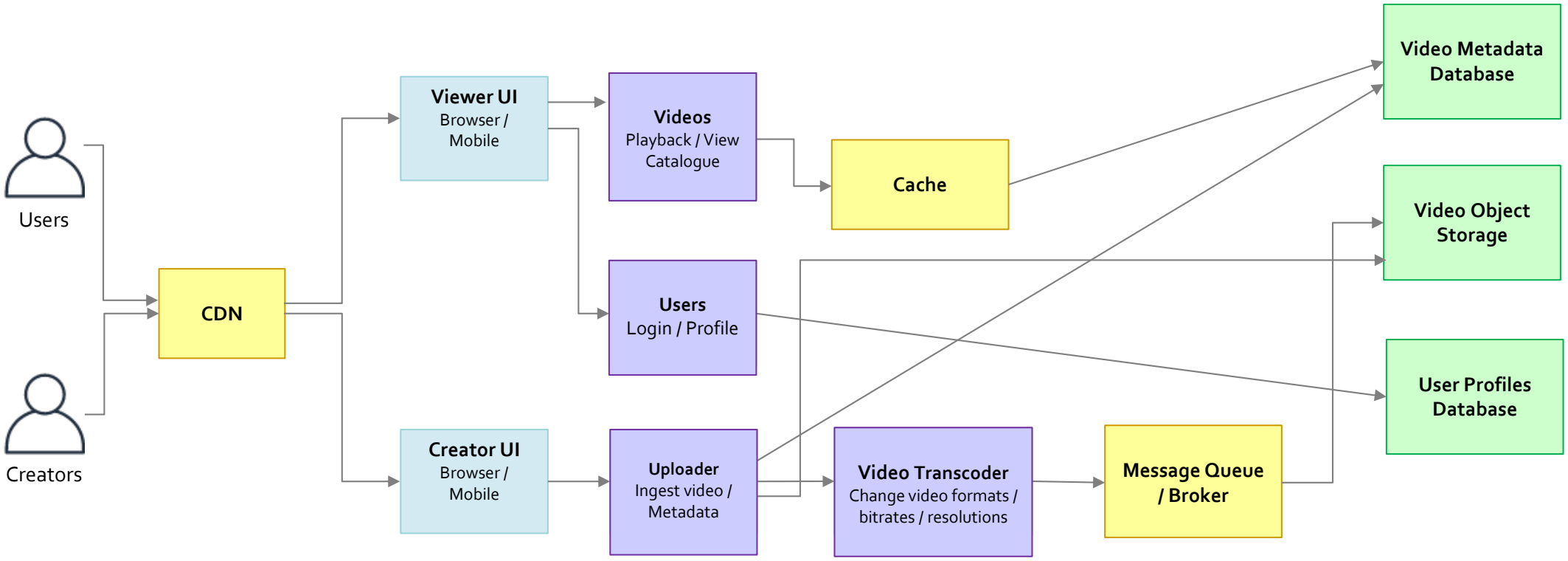
Asynchronous Messaging



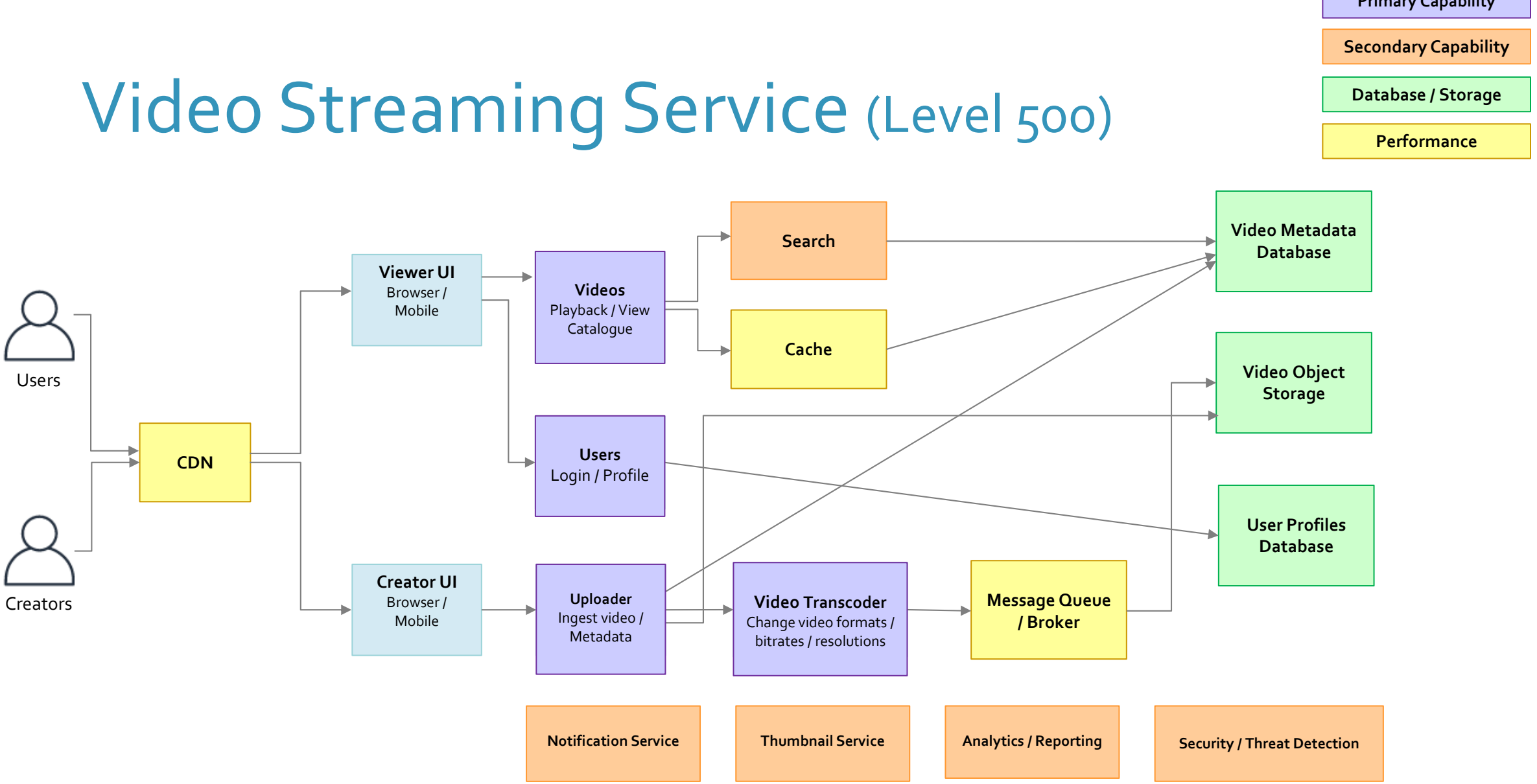
Service type	When would you use it?	What is it optimized for?	Associated services
Events	Use when you need to decouple publishers and subscribers and send events to multiple subscribers simultaneously.	Optimized for asynchronous, loosely coupled communication between publishers and subscribers. Events provide flexibility in message routing and delivery and are well-suited for event-driven architectures where events play a central role in initiating actions or workflows.	Amazon EventBridge Amazon SNS
Messaging	Use when you need either pub/sub messaging to broadcast messages to multiple recipients simultaneously, or point-to-point messaging when you need reliable and asynchronous communication between components.	Optimized for high-throughput, scalable, and reliable asynchronous pub/sub and point-to-point messaging between distributed components.	Amazon SNS Amazon SQS Amazon MQ



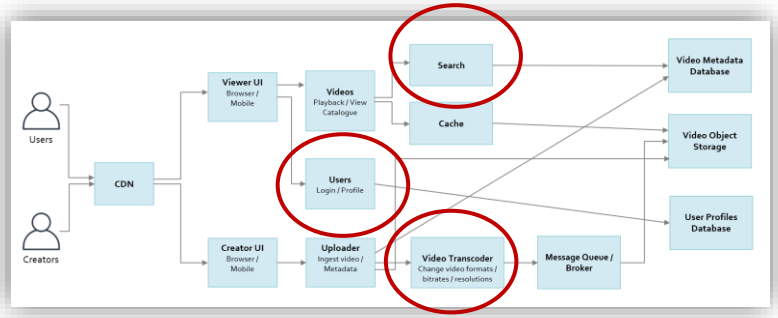
Video Streaming Service (Level 400 Recap)

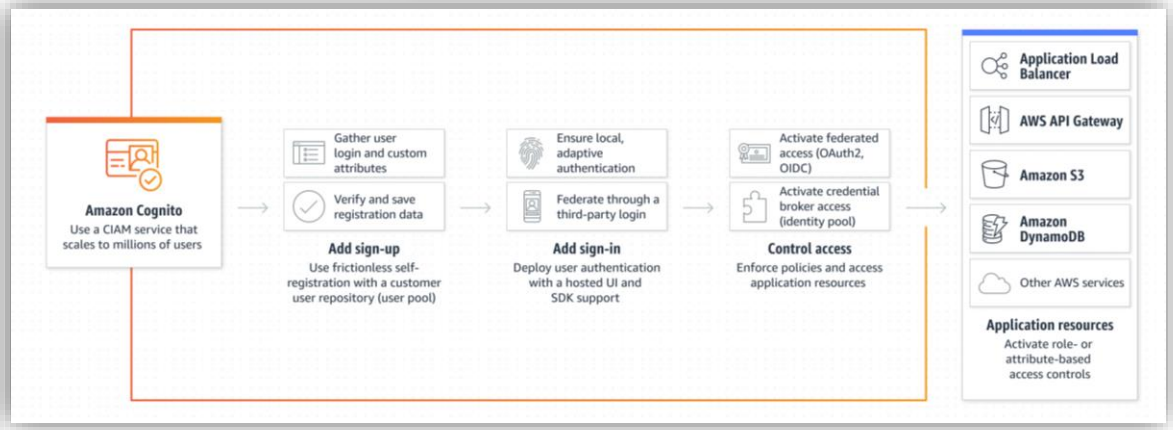


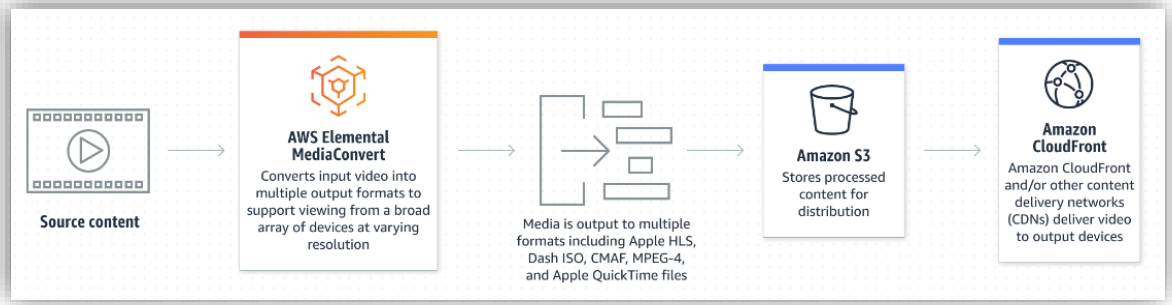
Video Streaming Service (Level 500)



Other Components







Summary

- Architecture is expected to evolve as a project scales from hundreds to thousands to millions of users
- Architecture used in venture-backed, hypergrowth internet-scale organisations can be instructive but not a good fit for everyone
- A monolith can allows us to explore the complexity of a system and its component boundaries before breaking them in to independent microservices
- Going directly to a fine-grained microservice architecture can be risky unless the application domains are well understood
- Refactor monoliths to coarse grained microservices before refactoring them to fine-grained microservices
- Adopt best of breed tools, frameworks and optimise infrastructure for each microservice independently

Q&A

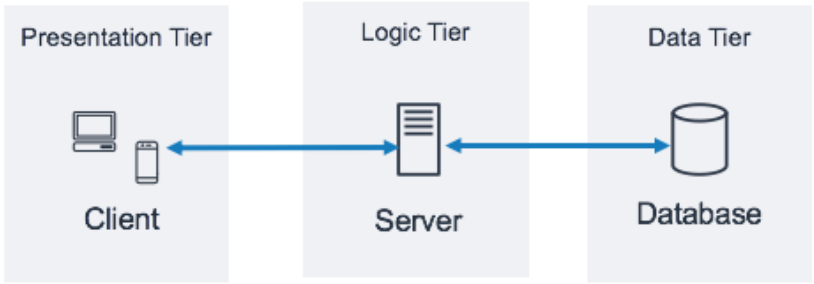
The End

APPENDIX

Additional content & reading resources

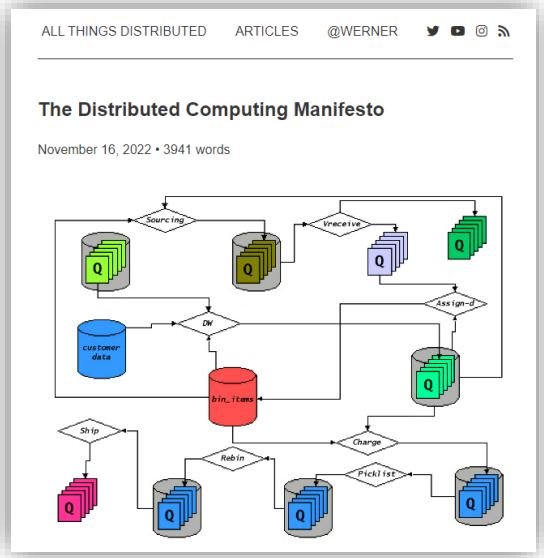
3-Tier Architecture

The architecture of a web application can be grouped in 3 layers. The key principle is “separation of concerns’

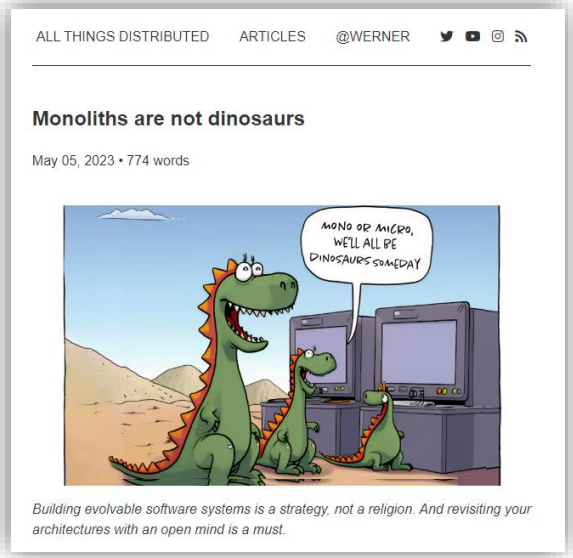


- **Presentation**: This layer collects input data, sends the request to downstream systems, and finally presents the server’s output (e.g. HTML in the client browser, JSON data).
- **Application**: This layer consists of application logic and business rules encapsulating all functional capabilities. It can also contains multi-step workflows that need to be performed to accomplish tasks.
- **Data**: This layer enables us to access and modify data residing in databases, file-stores, caches etc. Typical operations include generating, reading, updating, and removing stored data (CRUD)

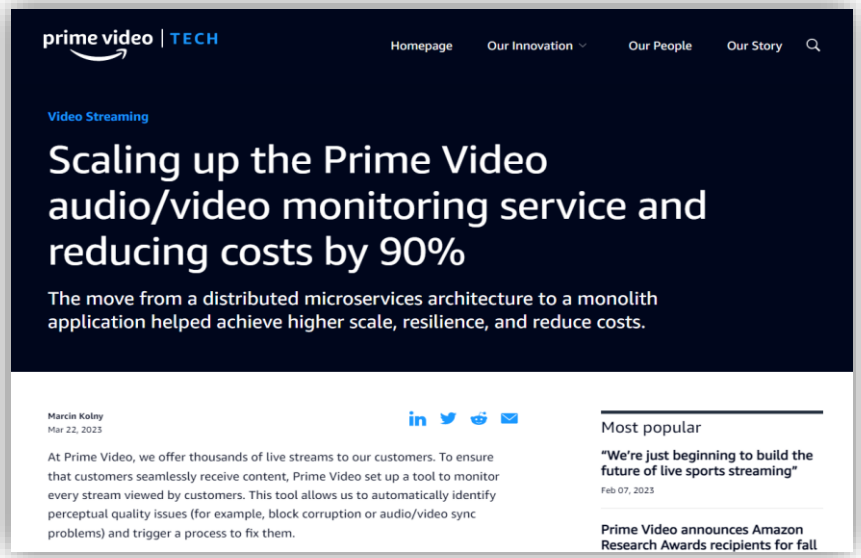
Blogs from Amazon Engineering Teams



https://www.allthingsdistributed.com/2022/11/amazon-1998-distributed-computing-manifesto.html



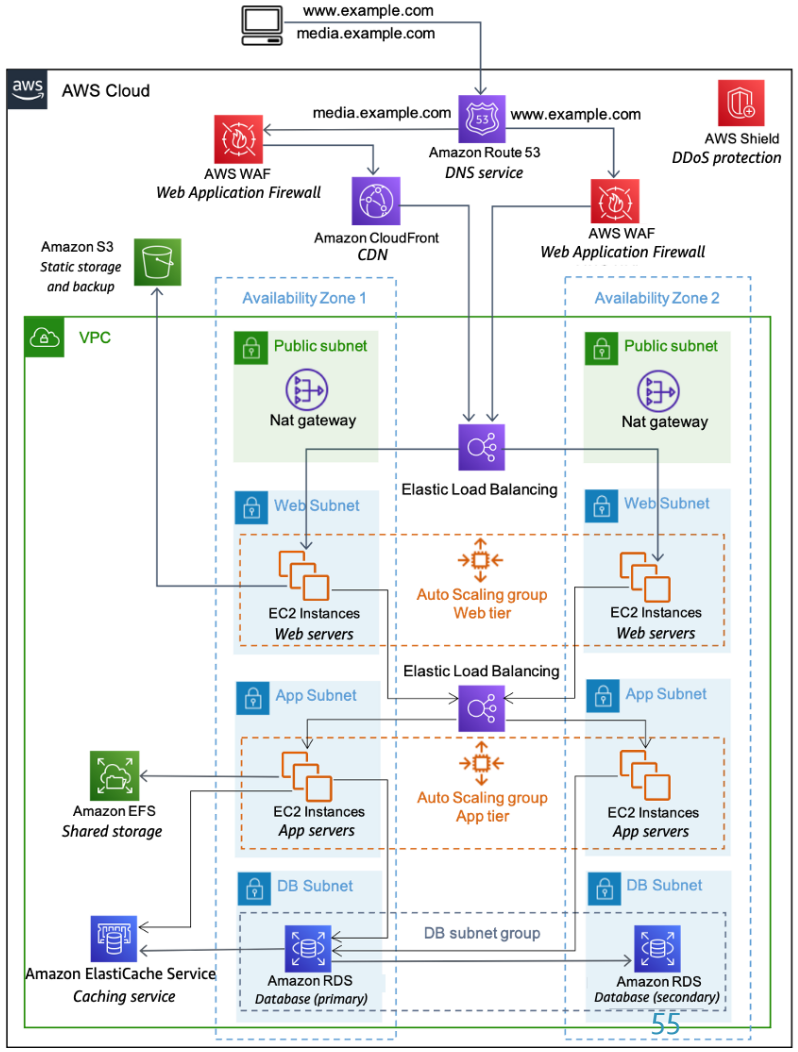
https://www.allthingsdistributed.com/2023/05/monoliths-are-not-dinosaurs.html



https://www.primevideotech.com/video-streaming/scaling-up-the-prime-video-audio-video-monitoring-service-and-reducing-costs-by-90

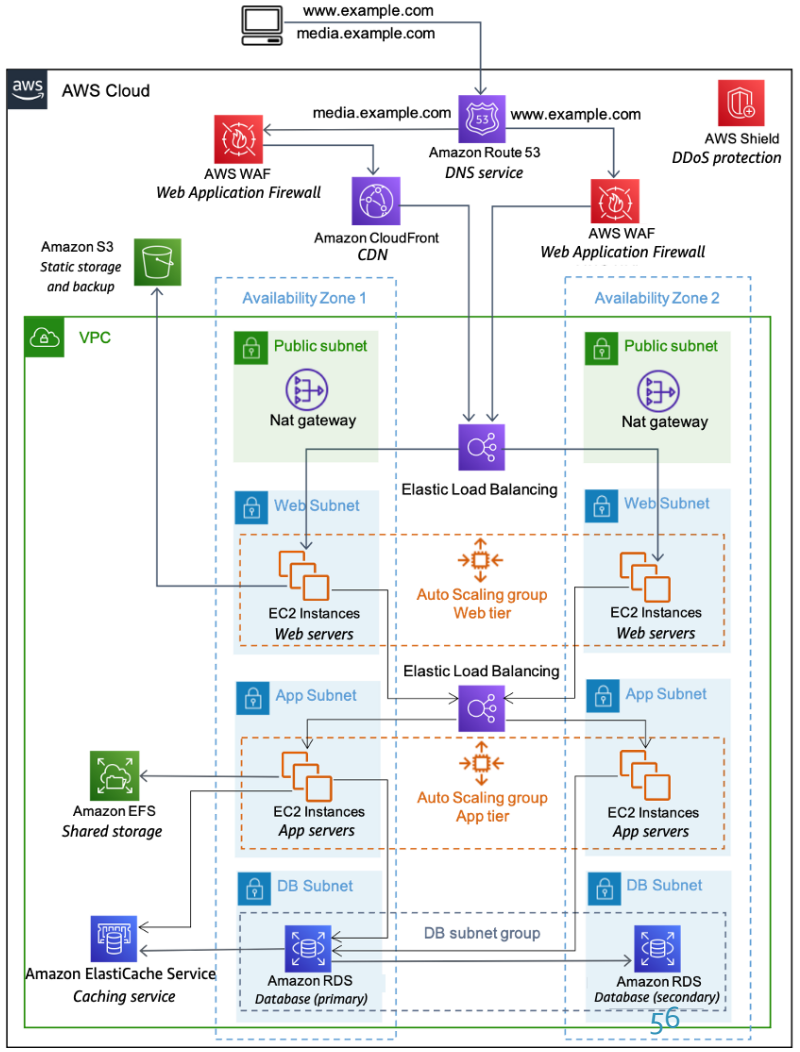
Reference Architecture (Monolith Implementation)

- Minimum viable architecture for 3-tier Web Application
- Deployed across minimum of two availability zones (AZs) to achieve High Availability
- Load balancers automatically distributes your incoming traffic across multiple targets i.e. instances
- HTTP Servers (e.g. NGINX, Apache) deployed in Public Subnet.
- App Servers and databases deployed in Private Subnet
- NAT Gateway provides outbound Internet connectivity for Instances in private subnets
- Bastion host in Public Subnet can act as a jump-box for Developer Access (e.g. SSH) to instances in private subnets (e.g. web subnet, app subnet)



Reference Architecture (Monolith Implementation)

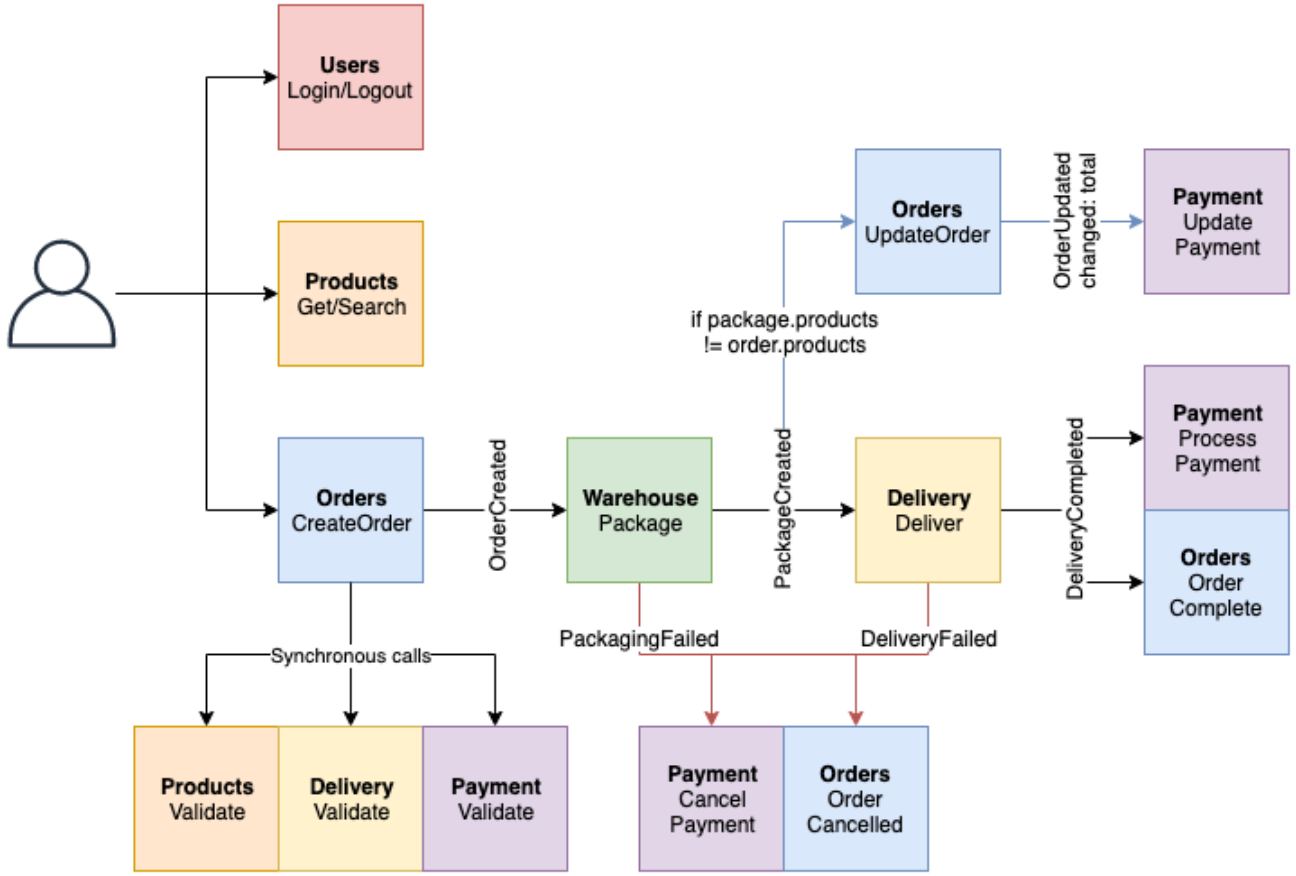
- HTTP servers and App servers in Auto-scaling Groups provide Horizontal scaling
- Databases deployed in Multi-AZ / Active-Passive Mode. Primary database for Reads/Write. Secondary for only Reads. Async synchronization across two AZs
- Application code is deployed to Linux VMs. Programming language for code can be Python, JavaScript, PHP, Java etc.
- App framework can be Django, Express, Laravel, Spring etc to enable rapid Prototyping.
- Database can be relational (e.g. MariaDB, PostgreSQL), document based (e.g. MongoDB), key-value store (e.g. Redis), columnar (e.g. Cassandra)



Ecommerce Backend

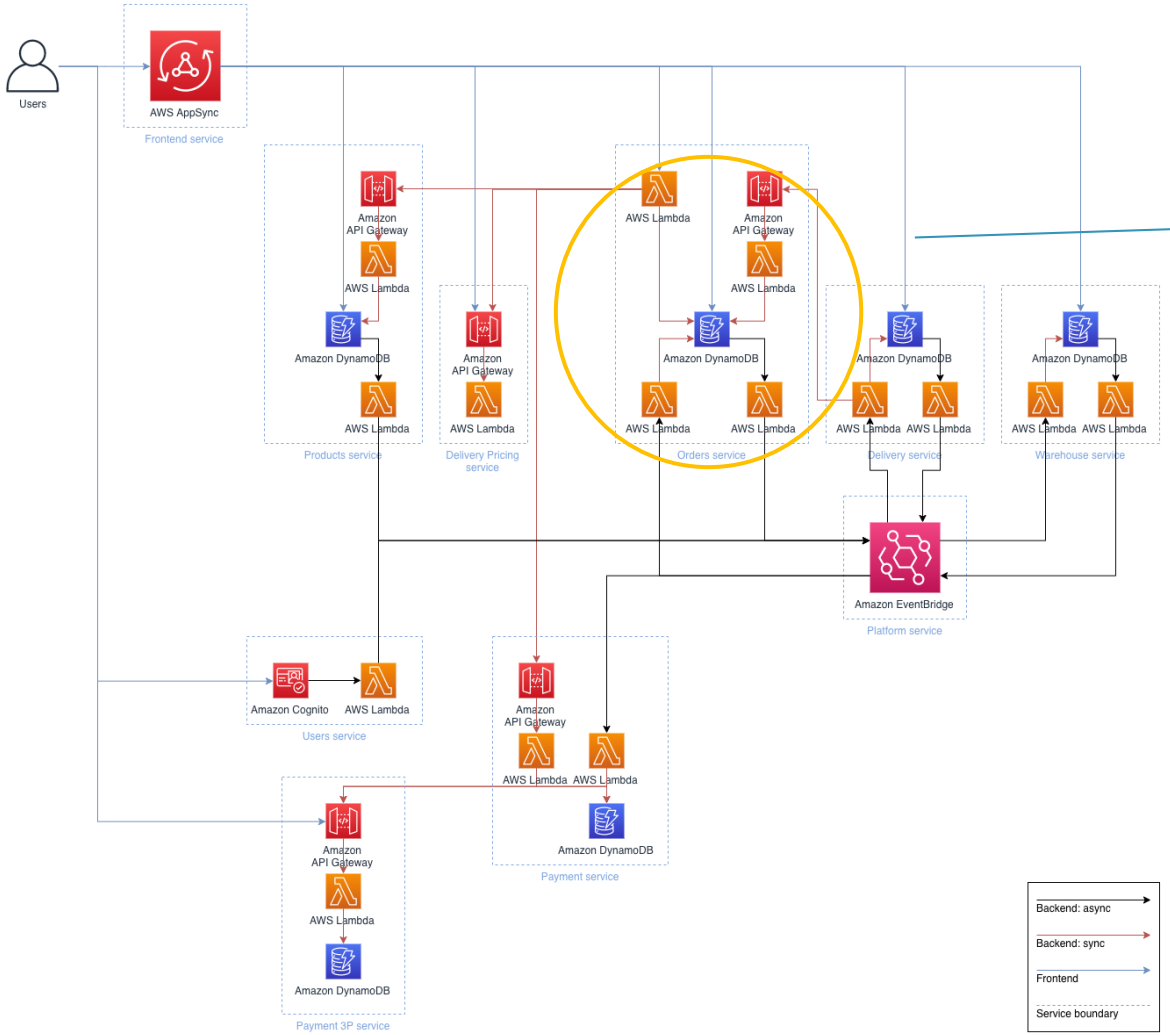
AWS Reference Architecture & Solution Design

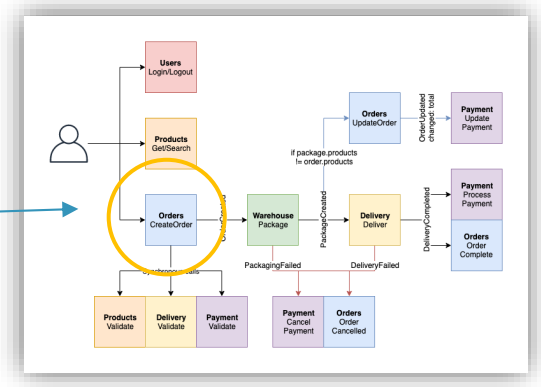
Ecommerce Backend (Components & Interactions)



Services	Description
users	Provides user management, authentication and authorization.
products	Source of truth for products information.
orders	Manages order creation and status.
warehouse	Manages inventory and packaging orders.
delivery	Manages shipping and tracking packages.
delivery-pricing	Pricing calculator for deliveries.
payment	Manages payment collection and refunds.

Ecommerce Backend (Microservices Implementation)





This is a sample AWS implementation of a serverless backend for an ecommerce website. Functionalities are split across multiple microservices that communicate through asynchronous messages.

https://github.com/aws-samples/aws-serverless-ecommerce-platform