AWS Services

1. **Be a Better Dev** - Best course on YouTube after reviewing dozens of non-specific videos with heavy accent: https://www.youtube.com/watch?v=FDEpdNdFglI&ab_channel=BeABetterDev. Summary provided below.
2. **Similar** course by the same author: https://www.youtube.com/watch?v=B08iQQhXG1Y&list=RDCMUCraiFqWi0qSIxXxXN4IHFBQ&start_radio=1&t=2s&ab_channel=BeABetterDev (this link shows the channel with other videos)

BOTH COURSES: LOOK FOR LINKS IN VIDEO DESCRIPTION (UNDER THE VIDEO) FOR LECTURES FOCUSING ON A SPECIFIC SERVICE

Drawback for 1 and 2: focus mostly on the web app architecture

3. **Certified Cloud Practitioner Course** (14 hours), very detailed, hands on with AWS consoles + theory: https://www.youtube.com/watch?v=NhDYbskXRgc&ab_channel=freeCodeCamp.org
4. **Certified Cloud Practitioner Course** (3 hours). The ppt file on the screen is more boring, but the course length is much more acceptable (better for summarization?): https://www.youtube.com/watch?v=JsmhEgIV1mQ&ab_channel=Learn2Cloud1017 **BUY COURSE** (incl. pptx): https://www.learn2cloud1017.com/product-page/aws-certified-cloud-practitioner-complete-study-guide-2023
5. **AWS Complete Tutorial 2023** (9 hours, better than most other AWS courses, Indian version of Certified Cloud Practitioner**)**: https://www.youtube.com/watch?v=B8i49C8fC3E&t=36s&ab_channel=SCALER, but it's not as good as Be a Better Dev or CCP - very colloquial talk, simplistic, a lot of water, not too many topics covered. 100K-word script saved in a separate file – a condensed version will be useful.
6. **Azure Fundamentals:** https://www.youtube.com/watch?v=5abffC-K40c&ab_channel=freeCodeCamp.org
7. **Google Cloud Digital Leader Certification Course:** https://www.youtube.com/watch?v=UGRDM86MBIQ&ab_channel=freeCodeCamp.org (6 hours)
8. **Google Cloud Associate Cloud Engineer Course**: https://www.youtube.com/watch?v=jpno8FSqpc8&ab_channel=freeCodeCamp.org (20 hours)

Introduction

Standard **three-tier application architecture** used in the video.

- Top layer - **UI or Presentation Layer** (author of the video calls it web backend layer) - built using HTML, CSS, JavaScript for web apps, UI dev libraries (e.g., React, Angular, Vue.js for web, and Swift for iOS, Kotlin for Android).
- Middle layer - **Application Layer** where app's functional **business logic** resides (backend services): intermediary between UI and data layer, processing user commands, making logical decisions, performing calculations, and moving data between the presentation and data layers. Enforces rules of the app, transactions, and security, developed w/server-side langs (e.g., Java with Spring, C# with .NET, Python with Django or Flask).
- Third tier - **Database or Data [Access] Layer** for storing and retrieving data; independent from app servers or business logic, providing modularity and flexibility to change DBs and schemas as needed without affecting other tiers. Common technologies - RDBMS (e.g., MySQL, PostgreSQL, Oracle), NoSQL databases (e.g., MongoDB, Cassandra), and ORM (Object-Relational Mapping) tools.

Amazon Route 53 = *DNS service* where you define all your DNS configuration for defining externally facing APIs or endpoints, including where to route traffic to from the internet; supports other things like health checks of endpoints and any traffic shaping. Your DNS would typically point to a load balancer endpoint.

Load Balancers - top-level category = Elastic Load Balancer:

- *Application Load Balancer* operates at the L7 level - if you want to use content from HTTP headers to route your traffic
- *Network Load Balancer* operates at the L4 level - lower level, more cost-effective, supports higher throughput limits, lower latency.

Note: levels associated with the OSI (**Open Systems Interconnection**) model.

- **L1 (Physical Hardware Layer)**: *physical components* of network communication (e.g., cables, switches).
- **L2 (Data Link Layer)**: Manages *protocols*, *devices*, and mediums to effectuate the transfer of data.
- **L3 (Network Layer)**: Handles *packet forwarding*, including routing through intermediate routers.
- **L4 (Transport Layer)**: reliable **data transfer across a network** including error correction, data flow control, establishment of end-to-end connections.
- **L5 (Session Layer) & L6 (Presentation Layer)**: less commonly referenced in AWS, designed for session establishment, management, and data presentation formats between apps.
- **L7 (Application Layer)**: top layer, closest to the end-user, *facilitates the interaction between software apps and lower-level network services*, deals with HTTP requests, load balancing based on content type, routing decisions based on headers, etc.

Compute options are the same for both web backend and application layer:

- **Amazon EC2** (Elastic Compute Cloud) - you rent *virtual machines* by the hour; you can use them for whatever purposes: host backend DBs, host a WordPress blog, deploy an app for a REST API. Drawback – u r responsible for the setup and configuration.
- **AWS Lambda** - serverless compute infra, more hands-off than EC2; you *write and deploy functions* (simple or complex snippets of code) and don't worry about any infra - *AWS deploys your app onto a container and scales it when # requests increases*. Attractive because it pay-per-invocation - cost-effective for app workloads w/bursty traffic or for apps w/traffic only during the day. Getting more and more popular. Check out the Lambda Udemy course by the author.
- **Amazon ECS / EKS** - Elastic Container Service / Elastic Kubernetes Service, helps you manage containers by *setting up servers with integrated load balancing and auto-scaling + facilitate the deployments to these containers*, kind of in the middle between EC2 and Lambda. Check out the author's video on the three compute options.
- **ECS** - proprietary *AWS-native container orchestration service* that supports Docker containers, but not Kubernetes; thus, *not portable* to other clouds. *Simple to use*, less management overhead - more straightforward container management platform. No community. No Kubernetes compatibility
- **EKS** - *managed service to run Kubernetes on AWS*, no need to operate your own Kubernetes control plane or nodes. *Portable* - runs the same way on any cloud provider vast ecosystem. More management overhead – *complex system*. Large community. Compatible with general Kubernetes - you can use existing tools and workflows that you are already using with Kubernetes
- **Kubernetes** - *open-source container orchestration platform* that automates the deployment, scaling up and down, load balancing, and management of containerized applications, groups containers into logical units for easy management and discovery, manages the lifecycle of containers. Docker provides the containerization technology (to create and package containers), and Kubernetes provides the means to manage containers at scale once they are created across a cluster of machines.

API Gateway - facilitates the *creation and hosting of REST APIs*. Super powerful service in addition to load balancer offering API throttling or authorization on an API for a private API to be accessible to a limited user pool - *API Gateway validates user tokens, does model validation* after defining what types of models your API supports, etc. A bunch of different combinations here – e.g. DNS pointing to API Gateway, which points to load balancer, which points to infrastructure layer. That is if you want to take advantage of some of those features that I just described.

Amazon Cognito - very powerful but underrated service, *very useful for apps requiring user registration: allows to create user pools* which are similar to what you'd have on any login and registration website - you create a user, they provide a login, a username, a phone number, a recovery option. With Cognito, you can have *users sign up for accounts directly within Cognito using the hosted UI, or you can integrate with other third-party identity providers* ("Log in with Amazon" or "Log in with Google", etc.).

Combined that with API Gateway, you can do things like ensuring that a user is part of a certain user group before the request can be validated and that flows through to the backend layer.

Database Layers

- **Elasti Cache** - caching service: a lot of apps have caching enabled to increase performance by looking up common items or in some other ways. Based on key-value lookups, relatively hands-off service, but there are nuances in terms of maintenance and alarming and you must worry about the infra - you own a cluster of memory-optimized nodes, but you still have to *handle maintenance* of the cluster, node replacement, hardware failures, etc. Flavors: *Memcached or Redis* (most popular) – both are *in-memory data stores* enhancing web app performance through caching. **Redis** supports a *broad array of data structures* like strings, lists, sets, hashes, and more, making it versatile for complex applications beyond simple caching. It also offers *data persistence* and built-in support for *replication and clustering*, enabling use in scenarios requiring *durability and high availability*. **Memcached** offers straightforward key-value storage, excelling in basic caching to reduce database load without the complexities of advanced data types or persistence. *Redis scales horizontally*, *Memcahed vertically*. Redis supports *six eviction policies* (no eviction, least recently used (LRU), random, if expiration is set on key-value pairs – removing shortest time to live (TTL)), while *Memcached - LRU only*. Choice between Redis and Memcached depends on the app's specific needs for data complexity, persistence - Redis = more feature-rich env, Memcached = simpler, focused solution.
- **Amazon Aurora** – relatively popular, in-house, compatible w/MySQL and Postgres (the latter in preview mode now): *fully managed RDS database*, you don't worry about administration, monitoring, storage / compute auto-scaling. It also offers *Data API* – calling RDS database using a REST API (vs. traditional database connection) + *a lot of other features* that are coming with Aurora.
- **Aurora serverless** - you don't worry about provisioning hardware, like an auto-scaling database, *provisions infra whenever the request rate requires it*: if you have a bursty workload, *it'll add more nodes and scale up so that your database can handle more volume* ==> it does for RDS what Lambda did for EC2.
- **Amazon RDS** - Relational Database Service similar to Aurora allows you select DB configuration: *MySQL*, *Postgres*, *Microsoft SQL Server*, *Oracle*, *Cassandra*, etc. This is *unmanaged* – you have to worry about manual scaling, replication, etc. Many customers still opt for Aurora – it's more hands-off and makes life easier.
- **Amazon Neptune** – graph database, Facebook-style nodes and connections between them.
- **DynamoDB -** most popular *fully managed*, powerful *NoSQL DB* optimized for *key-value lookups*: you don't worry about infra or hardware. You handle scaling configuration, and DynamoDB handles the *auto-scaling* for you behind the scenes. Used as the building block for much of the internet (used by a large part of the internet, including common services that we all love like Netflix and others). *The heart of many AWS services* behind the scenes.
- **Open Search -** more flexible querying at scale, new name for AWS's version of Elasticsearch, performs *queries that are more fuzzy in nature*: "Find all records where value one is in([X, Y, Z]), value two = m, and value three = n." Allows powerful grouping features, dynamic grouping. Comes with Kibana - open-source dashboard to review your data inside the OpenSearch DB. This has been used effectively as a replacement to RDS in some cases.

- **DocumentDB** - fully managed DB compatible w/*MongoDB*.
- **MongoDB** - popular open-source, *document-oriented NoSQL database* designed for *ease of development, high performance, and scalability*, robust indexing, replication for high availability, sharding for horizontal scaling, and a powerful query language. It stores data in *flexible JSON docs w/varying schema* from doc to doc - easily accommodates different data types and structures. Go-to choice for apps requiring rapid development and handling diverse sets of data models. Wide range of use cases *from simple CRUD apps to complex analytics at scale*. CRUD - software apps that enable to create, read, update, and delete (CRUD) data - four basic functions of persistent storage,

Pre-packaged infrastructure services

Make developer's life easier. Building a three-tier app architecture - a lot of moving pieces: different compute option, load balancing, API gateways, DBs, etc. Packaged services offer a *combination of different elements as a single product*. You don't worry about building blocks because pre-packaged infra services bundle this functionality together and *abstract away some complexity* at the sacrifice of configuration and control.

Elastic Beanstalk – PaaS that automates the deployment, scaling, and management of apps (health monitoring), allows to *set up any web app* w/different components incl. containerized app. Comes with a lot of components, but you *manage them in one spot: Elastic Beanstalk console*, e.g. backend layer, load balancer, auto-scaling, monitoring => you deploy apps quickly without managing the underlying infra but having the flexibility to customize the AWS resources - you fully control the infra via an *orchestrator service*. Supports Java, .NET, PHP, Node.js, Python, Ruby, Go, and Docker. It'll go out to all these different services here and provision what it needs for the type of application that you're trying to deploy onto it. Elastic Beanstalk orchestrates the deployment of EC2 machines, load balancers, and other services, but you still have visibility into the infra and must maintain it.

App Runner – PaaS, ***ECS** + Fargate*. Unlike Elastic Beanstalk, App Runner *abstracts away the lower-level components* of infra - you just specify the app configuration and deployment, and App Runner will *deploy that onto your infra and scale it* if required. Designed for quick automatic deployment, scaling, and management of **containerized apps** and APIs.

Amazon Fargate - *serverless compute engine for containers that works w/ECS and EKS*. Simplifies deployment by allowing users to run containers *without having to manage infra* (servers or clusters). Users specify and pay for resources per app. *Automatically scales* the underlying infra in response to demand, optimizing resource utilization, and manages the lifecycle of containers. Seamlessly *integrates with AWS services* for logging, monitoring, and security

Amazon Lightsail – IaaS, pre-packaged service that provide a simplified, cost-effective cloud platform to *launch and manage virtual private servers (VPS) with pre-configured software stacks*, designed as a straightforward, simplified, low-cost solution for *deploying web apps, websites, or dev environments* without the complexity of managing detailed cloud configurations and includes everything needed to jumpstart a project—a virtual machine, SSD-based storage, data transfer, DNS management, and a static IP + predictable monthly pricing plan. I use it for my AWS blog (beabetterdev.com), similar to other cloud vendors like *GoDaddy* or *DigitalOcean* where you select the stack for your app. You don't worry about underlying infra (e.g. what instance to pick as

with EC2 machines) - just pick different **pre-packaged options** for compute (reasonably priced). *You can deploy a WordPress blog, LAMP stack, MEAN stack, Ruby app, your own containers, etc. on Lightsail + you can add load balancing, auto-scaling, and other components*. A lot of features are built into Lightsail, but things are much more streamlined (**simplified, beginner-friendly**) in Lightsail – you add all the extra stuff within a safe zone of the AWS console. **Not recommended for production-grade apps**, but something great **for smaller apps** or a WordPress blog.

In summary, Elastic Beanstalk is best for developers who want to deploy applications without managing infra while retaining full control over AWS resources. AppRunner is focused on simplicity and is ideal for users who want to quickly deploy containerized applications without dealing with infra. Lightsail is designed for users who need a simple, low-cost virtual server solution, a simple and quick way to get their app or websites off the ground, with an easy-to-understand pricing model without needing to configure or manage the details of AWS resources.

Appsync - *fully managed GraphQL as a service* for GraphQL apps. Can be integrated w/other backend AWS services e.g. DynamoDB or Lambda functions. It *scales really well* and transparently if traffic increase.

Amazon CloudFront (CDN) – many web apps require *serving cached static content*: images, JavaScript, HTML, CSS, anything to cache and put close to the end user for better performance. If your app source is deployed in North America, but you have customers in Europe, Asia, or Australia, and if they have to go through a North America server, they will experience performance degradation. CloudFront allows to deploy a CloudFront distribution by *replicating some of your content from general object storage to regional nodes located all across the world, close to your end users* => *better performance* for much of this static content, optimizing the user experience.

Serverless - with Lambda, you are charged zero dollars if you deploy a function and never invoke it. Recent product launches like *Aurora Serverless, Redshift Serverless, and OpenSearch Serverless* use a cost model that deviates from this – there is a *minimum cost that you must spend even when the service is not in use*. Minimum # of Aurora Capacity Units (ACUs) = 0.5 units => minimum spend is $44.83 per month. Minimum # Redshift Processing Units (RPUs) = 8 => minimum spend is $44.83 per month. Minimum # of OpenSearch Capacity Units (OCUs) = 4 => minimum spend is $700!

Deployment

Now about how we actually *deploy and monitor* these applications. *Deployment - four different smaller AWS services* working hand in hand.

CodeCommit - storing source code either directly inside *CodeCommit as a service* or *integrating it* with third-party providers such as *GitHub* if you have a private GitHub repo. CodeCommit's integration w/other services makes it powerful.

CodeBuild - a) *build source code* located in CodeCommit or elsewhere into artifacts, b) *create and run tests* in a test env for your source code. The you combine this with other components listed below, and you can *build a sophisticated CI/CD pipeline* w/multiple steps here, *running unit and integration tests*.

CodeDeploy - deploy code to the: *deploy the built code artifacts onto different types of services* / compute infrastructure (knowing how to integrate the code with them).

CodePipeline - *deployment orchestration service* to chain individual building blocks together, orchestrate a *sophisticated CI/CD deployment pipeline*. You define a workflow of different stages that your app runs through, e.g. source code + build step + test step + deploy to a test env + another set of tests + deploy to prod env. There is a separate video on these services.

Monitoring

Applies to ALL services, very important concept esp. for prod workload, determines when things are going wrong and you need to step in. Two key services:

Amazon CloudWatch - *umbrella service* w/ a lot of different features, monitors the state of apps in your AWS account. Most useful ones:

a) *Evaluating metrics on many AWS services*: on CloudWatch, you can view in a chart format different metrics for EC2 machines over time, like CPU/memory utilization OR concurrent # invocations per day for Lambda.
b) *Logging* - application logs: what your app is doing. Or administration level events for managed services whenever you're using that infra.
c) *CloudWatch Insights* - searching over very large volumes of CloudWatch data using a SQL style language: convenient to find certain log lines in a giant collection of log files.

CloudTrail - audit trail of *what operations are being performed on your infra and who is performing them* (*another app*, such as Lambda f(x) calling a DB, or *another user*, who is accessing different services and what they are doing on those services). Events can be at the control or administration level - when your *infrastructure gets provisioned, deleted, or modified*. *Data level events*, if you configured it on a DynamoDB table, logs every single request to that table (not advisable, but can be done if needed).

Insights, if configured, can *automatically monitor* your account for *anomalies like security threats* using ML.

Identity and Access Management (IAM) - security management service for AWS. You can *create high-level entities* - users or roles and associated policies = you're explicitly denied access to everything AWS unless there is an *IAM policy* - you must create IAM policies, attach them to users, *assign new users to predefined groups with predefined set of policy permissions* applied to anyone in the group => then you create *individual user log in accounts* => Developer A has their own account, Developer B has their own account, etc.

Identity management is used all the time, every day if you're working with AWS - you're trying to get access to something. If you're experimenting with *a new service or feature, you're going to need to give yourself access*. Watch a separate video on IAM.

CloudFormation – *infra as code* is the preferred way to create and manage your infra. No one uses the console anymore unless you are new and experimenting. Everyone *writes infra in a code format or configuration format to easily pick up and deploy to a new environment*.

CloudFormation is a service that allows you to write *JSON* or *YAML* config files, then you upload them to CloudFormation who is responsible for *calling other AWS services* to provision your infra.

E.g. you write a template with a DynamoDB table and a Lambda function, upload your change set into CloudFormation, and CloudFormation will create the Lambda function and DynamoDB table (convenient and quick). However, *writing your infra as YAML or JSON sucks* - that's where CDK comes in

Cloud Development Kit (CDK) - a method of *writing the infra as code* that is more fluent for developers: like actual code *with loops and primitive functions*. You can be more expressive with your infra definition files, more dynamic, structure your code in a much simpler way. Unlike in YAML files, CDK has *autocomplete*. What's cool about CDK - it has easy to use *higher-level constructs* containing entire app specification => you can have a construct with an entire serverless architecture: a Lambda function, DynamoDB table, load balancer with API Gateway, Cognito user pool. It's just one line to write in CDK code to use this construct. Behind the scenes, CDK generates the code for CloudFormation who deploys it into AWS.

Other options for infrastructure as code – **Terraform** or **Pulumi**, but if you're looking to do everything **native in AWS, use CDK (and learn CloudFormation)**.

Two important services in the **rapid development** category:

AWS Amplify - a toolkit-style service that allows to *rapidly build and deploy entire apps*. It's primarily a *CLI tool* + it's a *great abstraction* when you don't know or don't care about individual AWS service - *it focuses on functionality and not the infra*. E.g. with Amplify, you can run a very simple command to add an API: "add API," and behind the scenes, Amplify will deploy maybe a Lambda function with an API Gateway. If you add user authentication and authorization, behind the scenes, Amplify gives you a Cognito user pool. You can add a relational DB, and it'll give you an Aurora, probably serverless.

One con is that it's an abstraction - it's great when everything is working, but if something breaks and you don't know the independent services, it may be challenging. So, if you want to stay within a well-defined box, use Amplify. If not, write your own CDK code and understand these other AWS services.

Serverless Application Model (SAM) - common *infra setup templates* written in CloudFormation, similar to higher-order constructs in CDK. SAM can *handle much of the complexity of the setup* for you – you just fill in a couple of specifications or a couple of fields, and it gives an intelligent default. Also great for building and *testing Lambda functions locally before deploying on AWS*.

Let's say we have a search query submission, an app similar to Google. A user searches, the result is stored in a DB, you probably want to send a notification to other microservices (who care or for analytics) that someone searched for this thing. What to use for event coordination, pub/sub, notifications to other services that something has changed in our app?

Simple Notification Service (SNS) – to be used if you want to tell others about changes in your data; pub/sub service => *publishes notifications to a SNS topic which can have many subscribers*. Idea – the main domain model service (publisher) receives an entry into its DB made by e.g. a search query service and needs to notify other microservices about it - it publishes this to an SNS topic. *Other AWS services can be subscribers*, e.g. a Lambda function, an HTTP endpoint on an EC2 instance, SQS, etc.

Simple Queue Service (SQS) – to be used if you want to be notified of changes in someone else's data; holds messages to be processed later. In SQS *you define queues connected to different types of compute infrastructure*: a Lambda function, EC2 machine, ECS task. These pieces of infra poll your queue for new messages and *perform some type of action when it finds new messages in the queue*. Often an SNS (publisher) is connected to an SQS (subscriber).

There is a separate video about the difference between SNS and SQS.

Eventbridge - very similar to SNS, but offers distinct benefits. Instead of SNS topics, EventBridge uses *event buses* which you can *integrate with many different application actions* all across AWS. E.g. you integrate EventBridge with whenever an EC2 machine gets terminated, or another operation you're interested in, or when a Lambda function gets updated, or when the configuration of Dynamo table gets changed. *You integrate these events into EventBridge, and define rules that specify who to deliver these events to*.

EventBridge also has *subscribers*, and you *define the rules and target groups* of who to deliver the messages to, depending on the type of event. But advantages of EventBridge over SNS:

a) **schema discovery** – doesn't exist in SNS or SQS: EventBridge allows you to *define the schema / format of the message*: JSON, XML, "foo" as a key and "bar" as a value, are there nulls, is it an array, etc. => helps subscribers get access to the models that are going to be delivered from the EventBridge event bus + it also allows you to *search through different schemas* to find the one required for your application.

b) **third-party integrations** - you can *work with e.g. Shopify*, PagerDuty, and many others (natively integrated w/EventBridge) – if an order is placed on your Shopify e-commerce website, it is integrated into EventBridge + you set specific rules to deliver this info to a microservice or a backend service that needs this update, or maybe to general object storage. This is not natively offered in SNS.

Also, events can be stored in S3.

Step Functions – like an orchestration service *define very sophisticated and large workflows with different steps* in them, like starting step, next step, X step, then Y step, etc. You can have conditional logic in your Step Function workflows.

Examples: customer ordering workflow: 1. Validate the details of the order. 2. Package the order in the warehouse. 3. Send out delivery notifications and send out a notification to the customer. You can have *fail-safe conditional logic*: if anything fails, take this different path. It offers direct integration with many other AWS services. So, you can use a service like *AWS Lambda to glue different parts of the workflow together*. And this is going to be completely *serverless*.

Amazon S3 (Simple Storage Service): is a large *object store, not a traditional file system*. It is designed to store and retrieve any amount of data from anywhere on the web, providing a *highly scalable, reliable, and low-latency data storage* infrastructure. S3 organizes data in a *flat structure*, using *buckets* (similar to root directories) and *objects* (files) instead of a hierarchical file system structure. Each object is identified by a *unique, user-assigned key* rather than by paths and directories. While S3 can be used to store files and can *mimic a file system* to an extent, its architecture and data access model are fundamentally different from traditional file systems. Very affordable, very scalable - you can store exabytes or petabytes of data + it scales really well. And you can also move your data over time into cold storage to get even better price points. E.g. you

can use S3 with CloudFront for caching - S3 can store any file types: images, CSS, video, other media, etc.

Amazon Elastic File System (EFS): as opposed to S3: EFS provides a simple, *fully managed, scalable, durable, elastic file storage* designed for Linux-based workloads for both Amazon EC2 / other AWS services and on-premises resources. *It scales on demand to petabytes* without disrupting applications, growing and shrinking automatically as files are added and removed (General Purpose, Max I/O performance, Bursting Throughput and Provisioned Throughput modes). EFS is designed to offer *high availability and durability, by redundantly storing data* across multiple Availability Zones and supporting the Network File System (NFS) protocol, which makes it suitable for a *wide range of applications* and use cases, including content management, web serving, data analytics, and more. EFS supports the *creation of multiple file systems*, each acting as a separate namespace accessible by EC2 instances, containers, and Lambda functions for shared access.

Unlike S3, EFS supports a *hierarchical structure of directories, subdirectories, and files* for organized data storage. EFS Integrates with AWS IAM for API access control and uses POSIX permissions for data access, alongside VPC security groups for network access control.

Elastic Block Storage (EBS): Unlike Amazon S3 and EFS, EBS provides highly available, reliable, and scalable block-level storage volumes for use with Amazon EC2 instances; they *automatically replicate* within their Availability Zone to protect users from component failure, ensuring *high durability*. They can be *dynamically scaled up or down* with minimal disruption. This is a type of data storage partition that *stores data in fixed-sized blocks*, each with a unique address: allows for efficient data storage, retrieval, and modification, as operations can target specific blocks rather than the entire storage volume. This makes it suitable for *intensive read/write operations*, frequent data updates or random data access.

EBS volume types tailored to different use cases: general-purpose or high-performance SSD, throughput optimized HDD for frequent access, cold HDD for less frequent access. Allow for snapshot functionality – creating *point-in-time backups* which can be used for long-term durability or to instantiate new volumes. EBS integrates with AWS IAM for access control and is encrypted at rest and in transit, providing robust security for sensitive data.

EMR - so, now let's say we dispatched an event, stored it in S3, and want to run some analytics on it. We can start with EMR - *large-scale distributed data processing system*. Run many different frameworks, incl. Spark clusters, Hive, Presto, or even in a serverless mode. This is a service for *massive number crunching* to perform some kind of analytics.

Amazon Athena - completely *serverless* big data processing or analytic service. Athena *connects directly to your S3 data*, crawl it, automatically detects the schema, and *creates tables* that you can query using *SQL*. Athena uses AWS infrastructure behind the scenes to *prioritize your request* - *you can run massive queries* on data stored in S3, really powerful analytics service and just number-crunching service, very viable option compared to EMR.

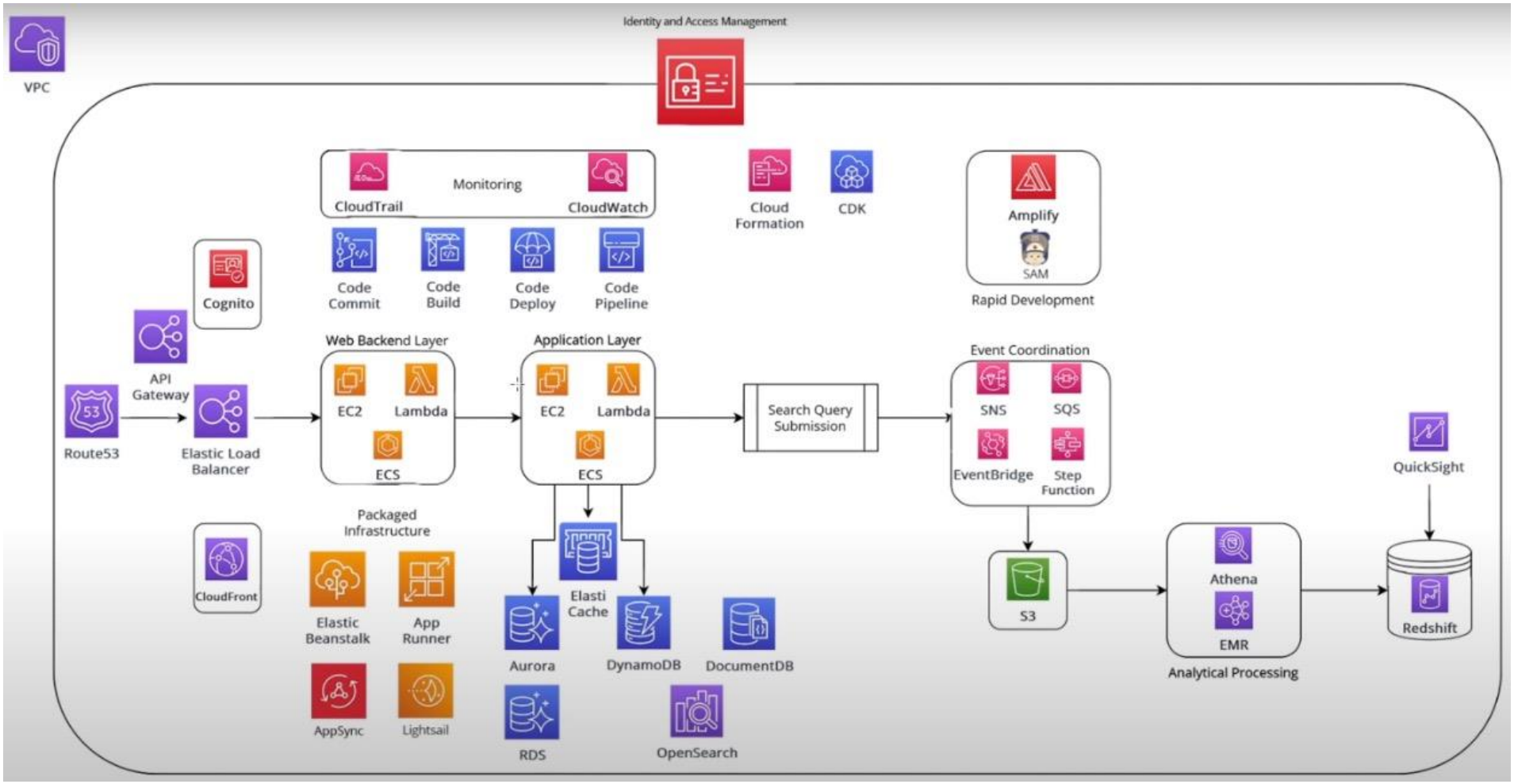
Amazon Redshift - data warehouse that stores data for *business intelligence and analytics*. DocumentDB, Dynamo, or any RDS are not really meant for this. A. r. is great for data engineers, business intelligence users, anyone that wants to *interact with data at scale using SQL*. *Columnar DB that performs very large concurrent OLAP queries* (columnar means quick operations on columns (not rows as in RDS) = it does Online Analytical Processing = slicing, dicing, drill down,

drill up, hierarchy, multidimensional trends, data consolidation, etc.;), supports many concurrent users, a bit expensive, but comes as a *serverless mode* (pay-per-use). You can set up an *automatic load job from S3* into Redshift.

Note: for some services, you can provision nodes in a distributed way.

QuickSight dashboarding - end users with independent logins *access data located in many different AWS services* (Redshift, S3, etc.) and *create business-facing dashboards* (similar to what Power BI did for Microsoft).

Amazon Virtual Private Cloud (VPC) – AWS is big on security => VPCs create a *network boundary and puts all your resources into an isolated network*. You create your own VPCs = *private networking spaces* for your infra, completely separate from other AWS users. You can connect your VPCs to other VPCs, e. g. to talk to other services in a different account, or you can make your VPC's infra callable from the public internet. VPCs can be very large hosting many different microservices or be service-oriented architectures. With VPCs, *you define the security setup and configuration* to isolate your resources from any other system and also allows you to define some security rules *to make sure that your infra is protected* from any outside actor.


VPC
Identity and Access Management
Monitoring
CloudTrail
CloudWatch
Cloud Formation
CDK
Amplify
SAM
Rapid Development
Code Commit
Code Build
Code Deploy
Code Pipeline
Cognito
API Gateway
Route53
Elastic Load Balancer
Web Backend Layer
EC2
Lambda
ECS
Application Layer
EC2
Lambda
ECS
Search Query Submission
Event Coordination
SNS
SQS
EventBridge
Step Function
QuickSight
CloudFront
Packaged Infrastructure
Elastic Beanstalk
App Runner
AppSync
Lightsail
Elasti Cache
Aurora
DynamoDB
DocumentDB
RDS
OpenSearch
S3
Athena
EMR
Analytical Processing
Redshift

Distributed Computing (copied to SCALABILITY)

Comprehensive *suite of services to build and deploy distributed apps* that are scalable, reliable, and highly available, catering to a wide range of use cases from web applications to complex ML and big data analytics:

1. **Distributed computing tasks** across multiple compute instances: *Amazon EC2* for scalable virtual servers, *AWS Lambda* for serverless computing, and *AWS Elastic Beanstalk* for deploying and managing applications. *Fargate* – autoscaling of containers.
2. **Distributed analytics and ML**: *Amazon EMR* for big data processing and Amazon *SageMaker* for distributed building, training, and deploying of ML models at scale. *Amazon Kinesis* offers real-time data streaming and analytics in a distributed manner. **Amazon Kinesis** - fully managed AWS service for *real-time data processing / analysis over large, distributed data streams*, ingests high volumes of data from various sources: app logs, website clickstreams, IoT telemetry with the capability for monitoring, analytics, machine learning, and other apps. Key Kinesis functionalities: K. Data Streams, K. Data Firehose (loading streaming data to data stores and analytics tools), K. Data Analytics, and K. Video Streams for streaming video.
3. **Distributed storage**, ensuring data durability, availability, and scalability: Amazon S3 for object storage, Amazon EBS for block storage, and Amazon EFS for file storage.
4. **Distributed databases** (storage, management, and retrieval): *Aurora (serverless)*, *Amazon RDS*, *DynamoDB* for NoSQL databases, and *Amazon Redshift* for data warehousing.
5. **Global Infrastructure**: *regions*, *availability zones*, and edge locations supports the deployment of *distributed apps and services close to end-users*, reducing latency and improving performance.
6. **Content Delivery Networks (CDN)**: *Amazon CloudFront*, a fast CDN service, *distributes content globally* with low latency and high transfer speeds.
7. **Security and Identity**: AWS Identity and Access Management (*IAM*) allows *distributed management of access to AWS services* and resources securely. AWS Shield and AWS WAF provide distributed *protection against DDoS attacks* and *application-layer attacks*, respectively.

Bedrock: a) *serverless*, API-driven access to foundation models (no infra management) *streamlining* the process of GenAI app creation, b) *easy* access to *quick experiments* and integration, c) *pay-per-use*, d) text, code, images, agents
JumpStart: a) *deploy flexibility* (greater control over model versions, endpoints, and security), pre-built solutions, algos, adapting to *unique project reqs*, b) initial setup *more complex*, but *broader capabilities* and customization, c) costs for deployed endpoints and data transfer, d) classification and regression for experienced ML users.

Sagemaker JumpStart

Foundation Models:

- Using foundation model in **Sagemaker Studio** (fine-tuning)
- Using **Python SDK**: fine-tune and deploy on an endpoint
- Models: Code Llama (7, 13, 34, 70), **Llama** (7, 13, 70), Falcon, **Gemma** (2, 7), **Mistral**, **Mixtral**, **Zephyr**

Task-Specific Models:

- **Text classification**, NER, summarization, Q&A, etc.
- **Image classification**, object detection, semantic segmentation

Customize a model: prompt engineering, RAG, fine-tuning

Solution Templates:

- **Credit rating prediction**, Demand forecasting, Financial pricing
- **Fraud** detection
- **Churn** prediction
- **Healthcare** and life sciences

Amazon Bedrock

Fully managed service that makes high-performing foundation models (FMs) available via a unified API. Features:

- Experiment with **prompts and configurations** of LLM for **inference** (purchase Provisioned Throughput for discounted rates)
- Adapt models to specific tasks or domains: **fine-tuning** or **continued-pretraining**.
- Augment with **knowledge bases** (to query and add to LLM prompts), build **agents.**
- **Evaluate** outputs of different models with built-in or custom prompt datasets.
- **Guardrails** - prevent inappropriate or unwanted content

Models: Titan, Llama, Claude, Cohere Command, Mistral, Stable Diffusion
Playgrounds for models
Access Amazon Bedrock API:

- AWS CLI
- Python SDK (Boto3)
- SageMaker Notebook (w/SDK)

Additional Requirement from Recruiter (Brian)

Fix Mac permissions for Chimes

Cloud infrastructure or system design – **services that support and being able to operationalize and host an LLM**: load balancing, CI/CD, databases, security. Anything that takes an LLM and supports it in a customer's infra. PLUS **distributed systems** in general.

There will be **broad questions** – ask for **clarifications**! Questions: please feel to ask lots of questions. We value **curiosity**. OK to say **IDK** – confident wrong answers! (IDK is what they are trying to find)

Experience around cloud infra

- DONE App Dev: Queues, JSON, CI/CD, etc.
- Storage: SAN, NAS, RAID, etc.
- Big Data / Analytics: Lifecycle of data, HDFS, basic definitions.
- Infrastructure: Containers (Docker), high availability, efficiency, etc.
- Networking: CIDR, MPLS, OSI model, latency, etc.
- Security: Certificates, Encryption, TLS, WAF, data at rest and movement, etc.
- 3-tier serverless app – how to identify a single point of failure
- **Bedrock**!
- Distributed systems – trade-offs, business value (how does it affect business?)
- Database: HA, Scalability, SQL, NoSQL
- Performance / Tuning: Latency, GPU, DoS, etc.
- Costs, scaling
- Layer 4 vs. layer 7
- DONE Systems: Tier application (beginning), design application (below), etc.

Designing Applications

1. **Architecture design** - cloud services (compute, storage, DB).
2. Ensuring app can **scale up or down** based on demand and remain **highly available**: load balancers, auto-scaling, and multi-zone or multi-region deployment.
3. **Optimize performance** by choosing the right storage (*SSD vs. HDD vs. read-write*) or the right compute instance (*CPU and memory*).
4. **Security** at all levels: *VPC* network access, *IAM* resource access, data encryption, security against threats.
5. **Cost efficiency**: right types and sizes of resources, using cost-management tools, leveraging reserved or spot instances for cost savings.

6. **Compliance** w/regulations and policies **and Governance**.
7. **Integration** with other systems and services: APIs, message queues, event-driven architectures.
8. **Disaster recovery and data integrity**: backup, restore, and disaster recovery to ensure data integrity and application uptime.
9. **Deployment automation**: Infrastructure as code (IaC), CI/CD pipelines to automate the deployment and updates.

Disaster recovery strategies

Ability to recover from a disaster with minimal data loss – two business strategies:

- **Recovery Time Objective (RTO)** - max amount of time your business can afford to be offline due to an incident without incurring a substantial loss
- **Recovery Point objective (RPO)** - max amount of data that can be lost due to an incident. Example: your DBs go offline, but your end users keep doing financial transactions on your e-commerce website - can you handle five minutes of transaction data loss (also measured in time, but relates to data).

These can be different in terms that your business can be offline from 11am to 12pm (RTO = 1 hour), but by the time the operations resume at 12pm, your data is restored only up to the 10:30am checkpoint – RPO is 0.5 hours (data from 10:30 until 11am is not recovered).

1. Backup and Restore

- Data and apps are periodically backed up to a storage service. If disaster - backups are restored.
- Trade-offs:
    - **Low cost**.
    - Low to **moderate complexity** (depending on scale and frequency of backups).
    - **Lengthy recovery time** if restoring large amounts of data.

2. Pilot Light

- Minimal version of env is always running in the cloud (core elements like DB at a min scale). If disaster - this env is scaled up to handle the load.
- Trade-offs:
    - **Higher costs** than backup, but still cost-effective.
    - **Moderate complexity** (more planning).
    - **Recovery time** is faster than backup - already running.

3. Warm Standby

- Full-scale duplicate of your prod env running at reduced capacity. Data synch periodic, but not real time. No live traffic or transactions.

- Trade-offs:
    - Higher costs than pilot light.
    - Moderate to high complexity (maintain a full standby env).
    - Quick recovery time (already running), but some delay vs. hot standby needed for scaling up.

4. Hot Standby

Same as Warm Standby, but the standby system is a fully operational, data synch is real-time, higher costs but time recovery – immediate.

4. Multi-Site Active/Active

- Two or more active envs in different geographical locations – each handles a portion of traffic under normal conditions. If one site goes down, the other one picks up the full load.
- Trade-offs:
    - Very high costs (multiple full-scale envs).
    - High complexity (data synch, traffic management).
    - Immediate recovery time.

Other Services

AWS Backup automates and centralizes backup across AWS services.

CloudFormation automates deployment of infra.

AWS DataSync - data transfer service

Data Replication

- High availability - cross-regional replication across AWS Regions and Availability Zones – minimizes downtime and data loss.
- Low latency - Aurora or RDS local read replicas that are geographically closer to end-users, etc.).
- Fast failover (RTO) - in case of regional outage etc. – quick failover to a secondary region.
- Makes it easier to scale.

CI/CD Pipeline

Continuous Integration (CI)

Before CI, developers worked independently in silos, and merging things together was chaotic. Solution: pushing changes to the same GIT repo. But with so many people constantly contributing code, things may not always run smoothly, and new errors can emerge. Solution: automate the **building / compiling and testing of code changes** (unit, functional, integration tests). If tests fail - team is alerted.

Continuous Integration ensures *seamless frequent code integration* into a shared repository and maintains *code quality*. It **improves collaboration** among developers by ensuring all *code changes are* **compatible** and **free of merge conflicts**. *Rapid feedback* for developers – **reduced risk of bugs**.

Tools: GitHub, GitLab, Jenkins, and others.

Continuous Deployment (CD)

Follows CI, automates deploy process, **auto-deploys all passing builds to prod w/out manual intervention** => rapid delivery of updates (features). Benefits: *faster release* cycles, *improved productivity*, and *enhanced customer satisfaction* – rapid delivery of improvements.

Tools: ArgoCD for Kubernetes.

Continuous delivery

Prepares code for prod by transitioning it through *multiple test / staging envs* (ensuring only stable code is deployed). Difference from continuous deployment - *requires a decision to deploy*.

JSON(L)

JavaScript Object Notation (language-independent) - lightweight data-interchange format, easy to read / write and parse by machines.

- **Infra as code in CloudFormation**.
- **Lambda functions**: **input / output.**
- **API Gateway**.
- **IAM policies**.
- **DynamoDB** - storing and querying data.
- Ensure **proper encoding to prevent injection attacks**, validate JSON schemas to maintain data integrity, and understand the performance implications of JSON parsing in large-scale systems.

Containers (Docker)

See ECS / EKS / Kubernetes above

By using Amazon Virtual Private Cloud (Amazon VPC) and AWS Key Management Service (AWS KMS) (reate, manage, and control cryptographic keys) all data transfers remain secure in transit and rest. Data protection involves securing data both in transit, using SSL (Secure Sockets Layer) and TLS (Transport Layer Security) => HTTPS, and at rest through server-side and client-side encryption

Content Delivery

CDN, Caching, see CloudFront

Additional Additional

MLOps

- **Increases productivity through automation** and standardization
- Ensures **reproducibility** through data and model versioning
- **Reduces costs** by minimizing manual tasks and improving model management
- **Improves monitorability** by allowing continuous model retraining and monitoring for data and model drift.

MLOps Tools

- **CICD**: Jenkins, TravisCI, and CircleCI - automate integration and deploy of code changes
- **Containerization and Orchestration**: Docker, Kubernetes, OpenShift - help package and deploy ML models.
- **Model Management and Versioning**: *MLflow*, DVC, Neptune - version control and collaboration on ML models.
- **Monitoring and Logging**: *TensorBoard*, *Grafana* (analytics and interactive visualization web app), ClearML, and ELK Stack - monitor performance and health of ML model.
- **Data Management**: Apache *Airflow* (workflow management), *AWS Glue* (data integration), Apache *Nifi* (data flow automation) - automate data collection, preparation, and storage.
- **Infrastructure Automation**: Terraform, Ansible, Puppet – automate provisioning and configuration of infrastructure.

Docker vs. Kubernetes

- **Docker** – containerization or package, distribute, and **run apps in isolated containers on indiv. hosts**. Providing lightweight env to encapsulate apps and their dependencies. You manually manage the host, and setting up multiple containers can be complex.
- **Kubernetes** – **orchestrates** the deployment, scaling, and management of **containerized apps** across **a cluster** of hosts / nodes, automates **load balancing**, **scaling**, and ensuring **the desired state** of apps.

Orchestration frameworks like **LangChain** and **LlamaIndex shine** - abstract away many details of prompt chaining; interface with external APIs; retrieve contextual data from vector databases; and maintain memory across multiple LLM calls; provide templates for common apps. Output – prompt(s) for LLMs. LangChain – leader. LangChain new (v. 0.0.201), but popular and in prod. Using raw Python for this is on decline.

Contextual data for LLM apps - text documents, PDFs, structured CSV or SQL tables. Data-loading is through ETL tools or using document loaders built into orchestrators like like LangChain and LlamaIndex. For embeddings, most developers use the **OpenAI API for text-embedding-ada-002** model: easy to use, good results, becoming increasingly cheap. Also **Cohere** - better performance in certain scenarios. In the open-source realm - Sentence Transformers **SBERT**.

From a systems standpoint, most important piece of preprocessing pipeline - **vector database**: efficiently storing, comparing, and retrieving up to billions of embeddings. Default go to in the market - **Pinecone**: fully cloud-hosted, easy to get started with, has many production features OOTB (performance at scale, SSO, uptime SLAs).

There's a huge range of vector databases available:

- ✓ Single-node: Weaviate, Vespa, Qdrant - tailored for specific applications.
- ✓ Local libs: Chroma and Faiss: easy to spin up for small apps and experiments. May not substitute for a full database at scale.
- ✓ OLTP extensions like pgvector: good option if you try to use Postgres, or for enterprises who buy most of their data infrastructure from a single cloud provider.

Most developers **start with gpt-4** (gpt-4-32k), then **switch to gpt-3.5-turbo** when project matures (~50x cheaper and significantly faster) OR **experiment w/other vendors** (Anthropic), triage **some requests to open-source** models, **fine-tuning open-source models**. Open-source models are closing the availability and performance gap (e.g. LLaMa 2). A **variety of inference options** are available for open source models, including simple API interfaces from **HuggingFace** and Replicate; raw compute resources from the major cloud providers.

LLM operational tooling is not developed deeply yet. **Caching** is common - *improves response times and cost*: usually based on *Redis*. Tools like Weights&Biases and MLflow (traditional ML) or PromptLayer and Helicone (built for LLMs) – **log, track, and evaluate LLM outputs**, usually to improve prompt construction, tuning pipelines, or selecting models. There are new tools to **validate LLM outputs** (e.g., Guardrails), **detect prompt injection attacks** (e.g., Rebuff).

Hosting LLMs:

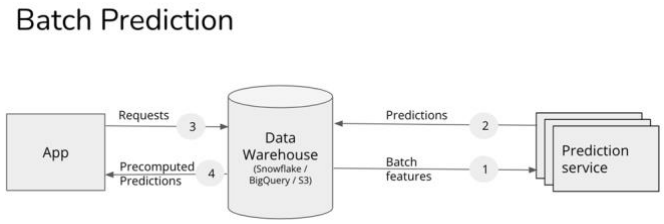
- ➢ **Cloud platform as a service** (Vercel)
- ➢ **Major cloud providers**
- ➢ Two new categories: a) **end-to-end hosting for LLM apps w/orchestration** (LangChain), multi-tenant data contexts, async tasks, vector storage, and key management (startups like Steamship); b) **hosting models and Python code in one place** (companies like Anyscale and Modal).

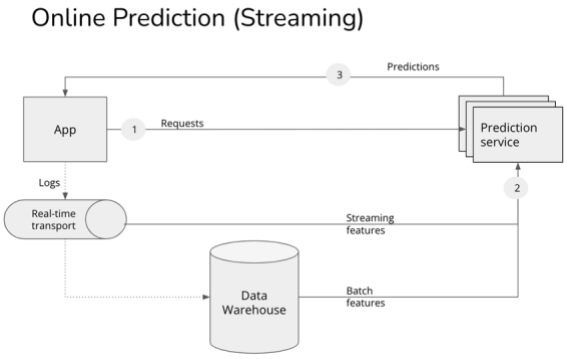
Special topic: **AI agent** (not covered above). **AutoGPT** – popular on Github. Developers are incredibly excited. Agents have a huge potential. But, agents are still in the **proof-of-concept phase**.

Batch vs. Online [Serving] Prediction

- o Batch p.: **periodical** (*hourly*, *weekly*), when *don't need immediate res*, saves resources (optimized for higher throughput), example – Netflix movie *recommender system*. Predictions stored somewhere (SQL or CSV files), retrieved as needed
- o Online p. - **immediate**, on-demand, low latency, example - Google Assistant *speech recognition*. Traditionally uses RESTful APIs.

Fundamental decision: online or batch? Term is confusing: both can make predictions for a batch of samples or one sample at a time ==> another term: **synchronous** or **asynchronous prediction**.





- ➢ **Batch features**: mean meal preparation time in restaurant in the past (historic data)

- **Streaming features**: # active orders and # delivery people at this particular time. Not always used, sometimes batch features are sufficient

Often, **side by side**, not mutually exclusive - a) **precompute predictions for popular queries**, online predictions for less popular queries, b) DoorDash / UberEats use **batch p. for restaurant recommendations** – takes long, too many restaurants, but use online p. for food item recommendations. Many believe that online p. less efficient (cost and performance) because you may not batch inputs together to leverage vectorization / other optimizations – this is not necessarily true. Online p. - no need to generate predictions for users who are not active (e.g. only 2% users log in daily – huge compute savings).

Model Compression

Low-rank Factorization

Replacing high-dimensional tensors with lower dimensional tensors – e.g. **compact convolutional filters**: over-parameterized convolution filters are replaced with compact blocks to reduce # params and increase speed. E.g. replacing 33 convolution w/11 convolution in **SqueezeNets** - AlexNet-level accu on ImageNet w/50 times fewer params. But, this method tends to be specific to certain types of models (CNN) and requires a lot of architectural knowledge.

Knowledge Distillation

Knowledge distillation - small student model is trained to mimic a larger teacher model or their ensemble. Example – **DistilBERT** reduced BERT size by 40%, but retained 97% of NLU capabilities and is 60% faster. This approach works regardless of the architectural differences betw. teacher and student (student can be RF, teacher - transformer). Disadvantage - if you don't have a teacher – its training requires a lot of data and time. Method also sensitive to application and model architecture – not widely used in production.

Pruning

Removal uncritical sections of decision tree (redundant for classification). In neural networks, pruning:

- **removal of entire nodes** of neural network - changing the architecture / reducing # params;
- **setting to 0 the least useful params** for prediction. Pruning != reducing total # params, but only non-zero params. Network architecture is same. Reduces size of model - **more sparse** (by over 90%!) and *improves computational performance w/out compromising accuracy*.

Some argue the value of pruning – why not train a dense model in the first place, but it was shown that a large sparse model after pruning can outperform a retrained small counterpart.

Quantization

Reduces memory footprint, improves computation speed (=reduces training time and inference latency): **using fewer bits to represent model's parameters:**

- ➢ Default - 32 bits per float number (single precision floating point) => 100M params = 400MB.
- ➢ 16 bits = half precision, cuts memory in half (popular: used NVIDIA, Google TPUs for training).
- ➢ 8 bits – fixed point or completely in integers (1 int = 8 bits). Not as popular for training, but is an industry standard for inference. Some edge devices support only 8 bits. Most frameworks for on-device ML inference (TensorFlow Lite, PyTorch Mobile) - offer free post-training quantization.
- ➢ Extreme case - 1-bit representation of each weight (binary weights).

Types:

- ➢ **Q during training** – training in lower precision (train larger models on same hardware).
- ➢ **Post-training** = trained in single-precision + quantized for inference.

Downsides: your range is smaller => you have to round up and/or scale the values outside the range => rounding errors, and small rounding errors => big performance changes. Risk of rounding/scaling to under-/over-flow and rendering to 0. But major frameworks have this built in.

Cloud vs. edge computing

- o **Cloud computing**: speedy data transfer over network - most queries to Alexa, Siri, Google Assistant
- o **Edge computing**: browsers, phones, tablets, laptops, smartwatches, etc. a) works w/out Internet, b) low latency, c) fewer privacy concerns, d) cheaper

Challenges in Training LLMs

This technology has a great potential for AI apps. But:

- ➢ requires **millions of dollars** for parallel training of the model.
- ➢ requires **months of training** + humans in the loop for fine-tuning.
- ➢ requires a **huge text corpus** + still accusations of using the scraped data illigealy.
- ➢ large **carbon footprint** (training one LLM = five cars in their whole lifetime).