Content

AWS Hierarchical structure

- AWS Data Centers
  - → "small" isolated buildings
  - → separated by <250 µsec
  - → "local" storage
- AWS AZ (availability zone)
  - → 1 or more data centers
  - → separated by <2ms. miles apart within an AWS Region
  - → Sync. replication between AZ (Aurora HA, Dynamo DB, S3, EFS, ...)
- AWS Regions
  - → several AZ
  - → Async. replication between Regions
  - → You enable, control, pay async data replication across regions (using AWS backbone)
  - * NOT all service available
  - * Cost
  - * Latency
  - * throughput
  - * compliance, governance

LOW ↓ HIGH

- AWS Edge Locations
  - → CloudFront with Security @ Edge (WAF, Shield, etc) → CDN / Cache
  - → S3 transfer acceleration
    - → route S3 traffic through closest Edge Location
  - → Direct Connect (Private wired to AWS)
  - → Route 53 → DNS → Lambda @ Edge

Adding elasticity to architecture

↳ ability to scale-out/scale-in as performance needs change

Monitoring & Log → know what's happening & react

security pillar

Cost Explorer
↓
where are money spent (reports & forecasting)

Flow logs
↓
EC2 network traffic

Cloud Watch

- **Metrics** ↓ health & performance metrics, or trigger alarms
- **Logs** ↓ collect, store, analyze logs from services and applications
- **Events** ↗ trigger process on a schedule or in response to a change state (event routing service)

Cloud Trail → record activity in an AWS account

* Elastic load balancing (balance multi AZ)

Network LB vs Application LB

Network LB ← Layer 4, Static IPs (TCP)
(network & transportation layer)
↳ NOT AWARE of app availability !!
↳ NOT DIFFERENTIATE between 2 applications on same host sharing IP

Application LB → Layer 7, flexible (HTTP/HTTPS)
(application = content based routing)

* Auto Scaling EC2 instances + ELB
↳ billing starts when AMI boot sequence initiate

= fault-tolerant
highly-available

- Easier → target tracking scaling
- Fine tune → step scaling
  - ↳ scale on >2 CloudWatch metrics
  - ↳ "steps adjustments"
    - ↳ not linear scaling

* Scaling RDS

→ vertical (Push-Button scaling)
- ↳ Storage layer (IOPS)
  - EBS storage → online
  - Aurora → storage always 'no limits'
- ↳ Compute layer (instance size)
  - Aurora serverless → auto-scale instance size, set limits or 'no limits', online
  - other db → manually choose instance size, offline

→ horizontal
- → Read-heavy workloads → Read replicas
- → Write-heavy → Sharding

* Scaling Dynamo DB
- provisioned → default, auto-scaling (set min. & max. limits) define how much $ is spent
- on-demand → serverless, pay-per-request

* Route 53 (= DNS look up, route between Regions)

Routing:
- → Latency → have resources in multiple Regions, route traffic to region with best latency
- → Geoproximity → based on location of resources, shift traffic from 1 to another resource
- → Geolocation → based on user location

Use automation

Infrastructure as Code = ↓ error prone
↓ cost ($, people, effort)

→ build infrastructure (stack) = CloudFormation
- use JSON / YAML template to design, document, build & update stacks
  - good template → quick starts
  - manual change → drift detection
  - maintain fleet after stack creation → System Manager

Can combine together

→ deploy application layer = OpsWorks
- fully managed Chef Automate Server
  ↳ opsworks for Chef Automate
- managed Puppet Master server
  ↳ opsworks for Puppet Enterprise
- deploy Chef recipes for free → opsworks Stacks

→ run web app = BeansTalk
- auto deploy & scale web app → Java, .NET, PHP, Node.js, Python, Go, Docker
  ↙ Apache, Nginx, Passenger & IIS servers

Disaster Recovery Planning

→ recovery time & recovery point objective

- recovery time: time takes to restore business process to its service level
- recovery point: the acceptable amount of data loss measured in time

→ KMS is region specific
! cross region encryption of data between primary and DR env

→ Duplicate data in another Region

pro : + no need to store tape backup (except Tape Gateway)
+ restore EBS snapshot in seconds
+ replicas can be accessed directly

↳ AWS storage :

S3 Glacier no rule to place delete marker.
↳ DO IN S3 LIFECYCLE RULE

- S3 → cross region replication
- EFS → AWS FileSync, AWS DataSync
- EBS → copy snapshot to another Region

EBS data lazy load, so up on initialization, there's performance hit

- RDS MS SQL, Oracle, Aurora PostgreSQL
  → in disaster, auto switch to standby replica in another AZ, if multi-AZ enabled.
  fail-over = 60 - 120s. auto change DNS record of DB to point to standby DB.
  ⇒ need to re-establish connection to DB.
- RDS PostgreSQL, MySQL, MariaDB
  ↳ cross region read replica
- Aurora MySQL → Aurora global database
- DynamoDB → DynamoDB Global Table

↳ on-premises Storage:

→ AWS Storage Gateway (Virtual Appliance)

- SMB 2, SMB3, NFS v3, NFS v4.1 } File Gateway
- iSCSI LUN → Volume Gateway
  - ↳ primary in S3, frequent access local
  - Gateway-Cached Volumes → upload buffer
  - Gateway-Stored Volumes → mirror
    - ↳ primary still local, back up in S3
- iSCSI VTL → Tape Gateway

require public virtual interface ↑ connection between on-premise & VPC

⇒ Performant & secure
↳ LAG link aggregation group = aggregate multi DX connection = max 40GB bandwidth

→ Network & Compute recovery strategies

- Pilot Light → Active/Passive
  - ✱ Compute resources are quickly recreated after the disaster using automation
- Fully working Low-Capacity standby
  - ↳ Warm standby

* a scaled-down version of a fully functional environment is always running

* can be used for non-prod e.g. QA, data warehouse, reporting, internal use, ...

* in a disaster, the system is scaled-up quickly using auto-scale to handle the production load.

- Multi-Site Active-Active

* a fully functional system running in AWS at the same time as the on-premises systems.

* Active / Active config, with a % of traffic go to AWS, remainder go on-site infrastructure.

* in a disaster, the system is scaled-up using auto-scale to handle full prod load.

Running Micro-services → decoupled architecture

- small
- stateless functions
- communicate over well-defined API

Architecture pattern	Service-Oriented (SOA)	micro services
Program size	interdependent business task	stateless small functin, API
Persistence	shared DB	different data stores
Communicat	enterprise service bus	queues, topics
execution time	~minutes	~0.1 sec
execution platform	VMs, Containers	Containers, Lambda

- Docker Containers
- AWS λ — run code serverless
- AWS Fargate — docker container serverless

→ Containers & Docker

- ↳ boot in seconds
- → small (storage)
- → popular runtime environment for running Win/Linux containers
- operating system virtualisation
- lightweight (RAM)
- No noticeable performance overhead (CPU)

→ container deployment?

→ Orchestration services

- ECS
  - elastic container service
- EKS
  - ECS for Kubernetes

→ distribute container images?

→ ECR (Elastic Container Registry)

free fully managed Docker image registry hosted by AWS

↳ deploy, manage, store container image in S3

→ where to run Docker containers?

→ EC2 instances

choose instance type, AMI & pricing model

provision & manage EC2 instances

→ pay by hour / seconds if using Amazon Linux AMI

→ Fargate

Serverless service for running (and orchestrating) Docker containers

→ pay by seconds

choose amount of vCPU (max 4)
RAM (max 30GB)

→ Serverless platform

↳ no deploy / manage of servers

↳ pay per requests (not paying capacity ahead of time)

e.g. Aurora Serverless, Dynamo DB on-demand

Fargate (containers), Lambda (code)

→ AWS Lambda

→ serveless computing service to run stateless code.
- → on schedule
- → event-driven

→ choose RAM (max 3GB), pay for compute time per 0.1 second

- default timeout = 3 s → can ↗ 15 min

→ can run at edge location

→ Lambda@Edge use case: segregate different types of users

expose to public → API Gateway

coordinate λ calls in visual workflow → Step Functions

Using caches

pro: + ↘ latency ↗ throughput
- → proximity to user
- → in-memory data

+ ↗ backend performance
- → lower load on servers, DB, ...

+ ↘ cost
- → less load → smaller & cheaper resources
- → pay to read data only once

!! Implement caching @ multiple layers of an architecture

* Edge caching → CloudFront
  - Content delivery network (CDN)
    - → use nearest edge location to customer to distribute static/dynamic web content
  - supports WebSocket protocol
    - → persistent connection
  - provides additional layer of security
    - → prevent DDoS

→ To expire stale contents
- Time to Live → easiest if reading stale data is tolerable
- rename object
- invalidate object → last resort, system must forcibly interact with all edge locations

* DB cache layer & session management

↳ in-memory NoSQL DB

- DynamoDB Accelerator (DAX)
  → in-memory, in-line cache for DynamoDB
- Elasticache Redis
  → managed in-memory data store based on Memcached or Redis

* Web tier caching

→ session management

each web server will remember the authentication, access control, state data

- ELB sticky session
  ↳ route a request to the specific server managing the user's session

(elastic load balancing)

⚠ when scaling, it's possible traffic might unequally spread across servers as active sessions prevent routing traffic to increased capacity

⇒ Better to store session state in a very fast DB

* DB caching

- RDS Aurora already integrates a cache
- Configure Elasticache for RDS EBS side cache
  - ↳ rewrite the app for Elasticache look-ups, population, and invalidation

most common design pattern for side-cache:

can combine both (
1. Write through → Data in cache is always up to date.
2. Lazy loading → add TTL to decrease stale data

Decoupling infrastructure asynchronously

problem:
- chains of servers tightly coupled, integrated
  - ↳ disruption in one = fatal
- cannot scale up/down layer separately
  - connecting layer needs care

Solution: + use intermediaries between layers → evolve / scale independently

X synchronous decoupling → internal facing load balancing?
→ failed step logging = complex
→ handling sudden traffic spikes
↳ over provisioning compute = $$$
→ out-of-order delivery
↳ timestamp = complicated

√ asynchronous decoupling
→ Messaging → reliable communication between components
↙ Queues
↘ Topics

- Amazon SQS (Simple Queue Service)

+ store messages in order (pipeline)
producer - consumer model

+ to prevent another consumer get same msg. the visibility timeout hides the msg.

↳ meanwhile, another consumer can pull following msg
⇒ allows scaling & load balancing

! consumer must explicitly delete msg from the queue
⇒ allows easy recovery from failed steps

→ what will happen when consumer that pooled the message died or the process failed
⇒ after a while, visibility timeout will expire and msg will be available to process again

→ what if subsequent retries fails?
⇒ msg is redirected to a "dead-letter queue" to analyse why it's stuck, or process it another way.

→ sudden traffic spikes?
⇒ buffering prevents consumer from being flooded

Queue types:

- FIFO queues
  - strict ordering
  - no-duplicates guarantee
  - soft limited to 300 msg/s
  → don't serve msg to > 1 consumer at a time
- Standard queues (default)
  - nearly unlimited scaling
  - best-effort ordering (msg might be delivered in an order different from which they were sent)
  - At-least-once delivery (dups possible)

Use cases:

- async & reliable communicat° between processing layers
  - e.g. network failures, batch process, backlogs traffic burst
- load balancing → multi consumer running concurrently
- scaling tripper → queue length = scaling criteria

→ avoid empty response for which we still have to pay ⇒ set long-polling (not default)

message size = max 256kB of text

→ up to 2GB use Amazon SQS extended client library for Java to store messages in S3 and send references to them.

- Amazon SNS (Simple Notification Service)

publish - subscribe model

→ once successfully delivered, there's no way to recall it

if no consumers available, then msg is lost

⇒ each sub consume msg once (consumed copies disapear)

⇒ sub receive all msg published to the topic
AND all sub to same topic receive same msg.

⇒ each sub can do something different in parrallel

Characteristics:

- supports diff delivery formats/transports for receiving notification
  - HTTP/ HTTPS
  - email, email-json
  - Standard SQS queues (not FIFO)
  - SMS / mobile push notificat
  - λ functions
- Highly scalable
  - soft limit = 10 million subscription per topic
    100k topic per account
- message size = SNS 256kB
  SMS 1600 Bytes

- Best effort ordering

- at-least-once delivery

- msg filtering
  = sub creates a filter policy
  ↳ only get notification that it interested in instead of every posts to topic

Use cases:
- parallel perform diff operations on same data
  e.g. deliver same data to prod and uat.
- push msg in diff formats
  mail & SMS
- time sensitive events
  ↳ notify event instantly

- Amazon MQ = fully managed Apache MQ service
  ↳ migrate apps to AWS
  ↳ integrate hybrid env

Networking

AWS VPC → virtual network

→ logical data center in AWS, have gateways, route tables, network access control lists, subnets and security groups

subnet → 1 AZ

security groups = stateful → virtual firewall, associated with instances when they are launched.

ACL = stateless

VPC peering = same / across AWS accounts

↳ you must have direct access

can't hop from 1 VPC to another via another VPC

VPC Transit = VPC in different region

NAT = install things from internet on my secured private instances.

⇒ ONE WAY ACCESS TO INTERNET

↳ Public subnet

↳ have an Elastic IP -> so to have a public IP

↳ must have 1 route from private subnet to NAT

≠ Bastian.
↳ administer instance using SSH

⇒ NAT Gateway (managed by AWS)

↳ Not associated with security groups

↳ automatically gets public IP

→ Block IP address using Network ACL, not security groups.

⇒ always have 2 public + 2 private subnet
⇒ different AZ.

↳ also applies to ELB

AWS Virtual Gateway → connect multiple locations

↑ to connect via internet

set up VPN

Border Gateway Protocal (BGP)

with ASN (Autonomous System Number)

and IP prefixes.

BGP peering = configured between customer gateway router & VGW using unique BGP ASN at each location.

⇒ VGW will receive prefixes & re-advertise to other peers.

⇒ Non-overlapping IP address pool should be configured at each regional office so that BGP ASN is unique

VPN tunnel → 2 different VPN endpoint, using ECMP = carry traffic on both VPN endpoints ⇒ ↗performance ↗ data transfer

↘ enable on Client device (on-premise)