HTTP error code, 3XX means redirection, 4XX means client side error (think about 404 when no connect, 403 means understood but denied), 5XX means server side error, 504 means gateway time out, 429 means too many request

**Developer theories:**

CI & CD (delivery or deployment) Maybe read whitepaper

* Development best practice enable frequent software changes to be applied whilst maintaining system and service stability
* Thousands of changes a day possible, Facebook, Netflix, AWS use this approach, AWS did 50 million deployment per year without outages.
* CI means Developer -> Git -> Automated build -> Automated test
* Delivery means the whole process of the delivery, deployment only means deployment phase. Continuous deployment means automated deployment based on successful build and test, continuous delivery may involve manual deployment.
* This way, releases can be small and incremental, instead of major every time. All decisions are built into the CI/CD process.
* In the context of AWS, we have CodeCommit (Git based) -> CodeBuild (can do basic test) -> Your choices of tests -> Code Deploy (To EC2, on premise, Lambda). All of this can be modelled, visualised and automated in workflow tool CodePipeline.

CodeCommit:

* Private Git repository, can use on local machine to interact with.
* Branch -> commit -> merge. Just like GitHub, you can branch, edit and commit in GUI
* You can create pull request, once conflict solved you can create. The owner of the master can then merge the code into master and has the option to delete source branch. Generally, you want to do that instead of keeping an outdated branch.
* Can configure notification through CloudWatch (monitor) and SNS (notify, Subscribe) event rule, on events like pull request, commits, etc.

CodeDeploy:

* Can deploy to EC2, on premise, Lambda. Reduces downtime, avoid risks, quick releases. Integrates with GitHub, Jenkins, pipelines, Atlassian’s suit.
* Like elastic Beanstalk deployment policy, we can have deploy type:
  + In-Place or rolling update, update one by one and organize traffic with an ELB, only for on premise or EC2, not for Lambda (no need anyways). To roll back, you have to do the same sequential replacement.
  + Blue/Green deployment deploy with new instances and failover. No Down time, fast switch back.
* Deployment group: the set of EC2 or lambda to which new version are being deployed
* Deployment configuration: rules and success/failure conditions for deployment
* AppSpec file: special configuration file, to tell CodeDeploy what to do, in .yml format (only for Lambda can use JSON). It specifies what file to where, what to do before and after, etc.
  + Structure slightly differ depend on your deployment target
  + Has Version (not enabled yet, only 0.0 by default), resources, hooks (BeforeAllowTraffic and AfterAllowTraffic, used to validate deployment before and after traffic is allowed to pass through)
  + For Lambda resource, must have: Name, Alias, Current version, Target version
  + For EC2 it has version (again not enabled), OS, files (need to be copied and destination), hooks (BeforeInstall, AfterInstall, ApplicationStart and ValidationService. They can be scripts and can do anything a script can do)
  + For EC2 and on premises deployment, AppSpec must be in the root directory
  + BeforeBlockTraffic, BlockTraffic and AfterBlockTraffic are three sections to specify ELB behaviour (what to run) during each stages of de-register/re-register instances during deployment. Don’t have to worry for Lambda.
  + Other hooks include: ApplicationStop, DownloadBundle, Install, etc.
* Run order cannot be changed (for in-place deployment):
  + BeforeBlockTraffic -> BlockTraffic -> AfterBlockTraffic
  + ApplicationStop -> DownloadBundle -> BeforeInstall -> Install -> AfterInstall -> ApplicationStart -> ValidateService
  + BeforeAllowTraffic -> AllowTraffic -> AfterAllowTraffic
* Revision: Everything need to be deployed, exes, config, resources etc.
* Application: Combination of deployment group, configuration and revision
* CodeDeploy need a service role, to be able to call APIs for handling EC2 launch and termination, autoscaling, read tags (identify servers), etc.
* To manually deploy to EC2, you must install CodeDeploy agent. Then create an application in CodeDeploy, upload zipped source through code deploy to S3, create a deployment group in the console, config:
  + deployment type: In-Place, Blue/Green
  + environment: EC2, Autoscaling EC2, On premise, what’s the tag to ID the server
  + Setting: One by one, half at once, all at once or custom (this is deployment configuration, not deploy type)
  + ELB setting
* Once deployment group is created, create a deployment, config:
  + Where’s the revision (S3 or GitHub), location link
  + Fail handling (when target has a same named file, retain/overwrite/fail)
  + Rollback handling based on deployment fail/CloudWatch alarm thresholds
* You can also avoid all these by using a sample deploy wizard, the wizard runs CloudFormation in the back.

AWS Pipeline:

* Fully managed CI/CD, execute user defined release process. Fast, reliable, consistent, without human error. However, there’s manual approval feature.
* Works with all AWS developer tools, can deploy to EC2, Lambda, work with EBS, CloudFormation, ECS, as well as third party tools like GitHub and Jenkins.
* Triggered (S3 based code need CloudWatch detecting S3 versioning to help trigger) when new source is available, stops when one step fails, before deployment
* Pipeline needs a role too
* Steps to create a pipeline:
  + Specify source S3, GitHub, CodeCommit ->
  + Specify build services Jenkins, CodeBuild, Solano ->
  + Deploy service CodeDeploy, ECS, EBS CloudFormation
* Once source is updated, it starts a “staging” phase, to deploy code. Once finished, you can go into each instance and see what deployment stage is wrong.
* When using S3 as source, a better practice is to delete the code zip and upload a new one, otherwise S3 has eventual consistency and new code may not be picked up. Also careful the name of the zip file should be the same, Cannot even change the version number.
* Delete method: Delete stack in CloudFormation

CodeBuild + Docker (ECS and ECR):

* Firstly, add access to roles/user to interact with dockers
* Docker containers is a standalone executable and has everything software need to run
* ECS allows you to run a cluster of docker images in the cloud
* CodeBuild can build docker image to be used as artifects. You can also run the docker image you created elsewhere in ECS.
* To create docker images on local machine, run the following command:
  + Docker build -t docker\_repo
  + Docker tag … (to tag docker image with a Alias)
  + Docker push docker\_repo\_destination
* ECS can be initiated based on different template, it creates VPC, subnet, Auto Scalling Group and OS image to run docker, basically all the layers needed to run a container, So it’s not serverless.
* Once created, you have access to both your container instances and the underlying infrastructures including EC2 instances.
* You then need to create a docker repo, and push your docker images into it.
* To active a container, create a task, configure docker repo/image Alias, and then port mapping from 80 to container port.
* Then user the task to create a service, to package the ability to run a “task” into a “service”.
* You can load-balance a number of containers, and config auto Scaling based on CloudWatch alarms
* All the steps above can be automated by CodeBuild, with user defined procedures in buildspec.yml, which contains pre-build, build, post-build commands, variables, artefacts, etc.
* In CodeBuild, configure:
  + Source, how to build (OS, runtime/docker, build command if not using default buildspec.yml, inline build override default file)
* When build procedures (called a build project) are all configure, you can start a building docker images. Once building starts, you can see logs including status (fail/pass) for each stage.
* Difference from virtualisation (run code on virtual machine): virtualisation still have guest OS and dependencies, that consumes a lot of resources. Docker image is much lighter weighted
* Make micro-services more feasible
* ECS task definition maintains several docker cluster at all time, working as a load balancer

**Application services:**

SQS (Simple queue service), first ever AWS service, worth reading FAQs

* Distributed queue system, temporary repository for messages that awaiting processing, great way to decoupling components, so when component crushed, or data flow congested, messages can be re-processed after visibility timeout.
* Always Poll based, push based use SNS.
* Can be set to autoscaling (auto scale up or down when the number of consumers based on how many messages are in the queue), can be consumed by server, lambda, EC2 etc. It is pool based so it doesn't push. it supports 256KB data of any text format, billed in 64KB blocks, including JSON, lifecycle 1 min to 14 days, default 4 days
* Support standard or FIFO, highly reliably and available, order is most likely (best effort) maintained but not guaranteed in standard, FIFO has guaranteed ordering, limited 300 transaction per seconds (standard is near unlimited).
* When pulled, message will become invisible until computation is done, but will only be deleted when processing is over, so in case of failure, it can be pulled again
* Visibility time out has max limit of 12 hours, and default is 30 seconds. when processing is longer than timeout, it could be processed by another server because the first server did not return with confirmation. Say if a task takes minutes, set the invisible time to 10 minutes is more reasonable than using the default 30 seconds. If process takes longer than 12 hours, maybe skip SQS altogether.
* short pulling: pull all the time and respond instantly if there's nothing. Long pulling pulls periodically and only return when there is something or time out. Long pulling can save cost, but when there’s always enough task, or you want instance response when there’s no job, short pooling make sense. Maximum Long pool timeout is 20 Second
* You can add your own mechanism to handle duplication (or any other application level tracking). FIFO queue guarantees message gets delivered once and once only
* Delay Seconds: when a message arrives, make it invisible for a few seconds (What’s the use case??)

SWF (Simple workflow service)

* Compare to SQS, SWF manages workflows that could be code, script or human actions, SQS message oriented
* 3 actors: starter, decider, worker
* SWF decider can monitor the workflow and assigns tasks to workers according to decider's decision. tasks are assigned once and once only (Main difference from SQS)
* Maximum workflow period can be 1 years measured in seconds, 31536000

SNS (Simple notification service), includes SES (simple email service)

* publish-subscribe paradigm, notifications are pushed to subscribers, if needed it can pool (pull) notifications use SQS. Cannot receive, just push.
* Can deliver to IOS, google, fire OS, using Baidu message service in China. Can also send SMS or email to SQS or any http(s) endpoint.
* SNS can trigger lambda, when a lambda subscribes to a SNS, then Lambda can react based on the SNS message input (more likely what’s passed in is parameter)
* Subscribers subscribes to "topics", one topic can support deliveries to multiple endpoint types. e.g. group IOS, android and regular users and regular users and send them different messages (smartphone gets push notifications and normal gets text)
* Messages are stored in multiple AZ, so highly reliable
* Pay as you go, prices vary according to delivering method
* web based point-and-click interface
* data format is JSON

SES (Simple Email Service)

* Help marketing/notification/transaction emails to their customers, doesn’t need to confirm subscription, a email address is all that’s needed.
* Used to receive emails, and incoming emails can be delivered to S3 or trigger lambda, can trigger SNS too

Elastic transcoder: convert media file format. Provide pre-sets for popular formats so don't need to guess, pay based on the minutes and resolution. Use case: source video -> lambda -> transcoder -> S3 for web consumption

API gateway: Routing API calls to services (lambda or EC2 server).

Can use API caching to speed up API response speed by caching response for a TTL instead of letting it run every time, scales automatically, however you want to throttle to prevent over flushing. To do this you can connect to cloud watch to log all request

When resources are not accessible because of same origin policy, active cross origin resource sharing (CORS)

Kinesis (video and data stream): data streaming service (in analytics session), you send your continuous data transfer in small sizes (KBs) to kinesis (IoT, stock price, purchase orders, social network, geospatial etc.). Producers send data to Kinesis for a retention of 24 hrs - 7days, data stored in shards, then consumers consume data from shards, and send results to other AWS services (EMR, S3, DynamoDB, redshift).

Maximum 5 transaction per second, 2M max for read and 1000 transaction per second, 1M max for write, so can handle way more intensive write than read. All sizes include partition keys. If this is not enough, provision more shards.

Kinesis firehose: send stream to firehose and shoot to S3/Elasticsearch Cluster (and maybe then redshift). You can optionally add lambda or some other pulling mechanism for analytics. It doesn't have shards so have no retention period (don’t have to manage yourself now), don't worry about consumers. For something more require more of a warehousing fashion.

Kinesis Analytics: Allow SQL data in kinesis/Kenises firehose (e.g. ETL written in SQL), and send processed data out to redshift, S3, RDB, Elasticsearch Cluster, etc.

CloudFormation:

* create “stack” from template, AWS provide a selection of templates, can take some time. Say using the template for kinesis, it generates all the resources needed (can be seen in the log), for a website to display the clicks on several websites in real-time. With an EC2 producing and consuming data, a dynamo DB storing data and Kinesis which can be monitored through a number of metrics in the console.
* CloudFormation can handle the deletion of the whole stack too.
* CloudFormation is under management tools, where elastic beanstalk sits under compute. One it’s more about operation excellence and the other is more specific for deployment.
* Templates supports Yaml and JSON, CloudFormation interprets them and do the job.
* Benefit: Consistency, fast, version control, reuse, free service, roll back/delete flexibility
* Process: create template -> upload (once uploaded it goes to S3 and automatically encrypted)-> CloudFormation does the job.
* Attributes in templates (AWS provides a lot of example code, In both formats):
  + Version (Only support 2010-09-09), Description, Metadata, Parameters (input, can define names for future reference, like keypairs), Conditions (decisions based on environment), Mapping (region specific value), Transform (customised code/ templet snippets/Lambda outside of main template stored in S3/ Specify SAM), Resources (resources to be deployed), Output (Something helpful like instance ID, next step command, etc.)
  + Resource is the only mandatory field
  + YMAL can have comments, Regular expressions, intrinsic functions
* For Launching EC2, careful the AMI ID is region specific (in resource).
* You can config to make the stack roll back If a part of the provisioning fails (in console or use a flag in CLI)
* CloudFormation will roll back when anything fail, and will not execute when there’s an syntax error.

Serverless Application Model (SAM):

* An extension to CloudFormation to easily define serverless resources like APIs, Lambdas, DynamoDB etc.
* Provides SAM CLI with the following command:
  + SAM package: package your template into a SAM compatible template
  + SAM deploy: deploy to a certain stack, with the SAM compatible template, and an extra IAM role for the deployment to allow the Lambda to run

To use SAM in CloudFormation:

* Have a S3 bucket ready for code
* Package code into SAM package, with Lambda.yml, including a AWS::Serverless-20XX-XX-XX specifies this is a SAM template as transformation. Get code uploaded to S3 and a template generated.
* Run deploy command to deploy code from S3 to Lambda (in the form of a CloudFormation stack, with a role Lambda needed to run)

Elastic beanstalk:

* Used to deploy Web-apps developed in different languages onto several popular platforms like Apache Tomcat, Nginx, Passenger and IIS.
* Developer can focus on writing code, just like lambda. Basically, a more specific, GUI based (not JSON) version of CloudFormation.
* You can customize the environment being created by using configuration file, like modify the one in maven. These files are written in JSON or YAML, ends with .config and placed in .ebextensions folder (in top level folder of you code bundle).
* Handles deployment, capacity provisioning, load balancing, auto-scaling and application health. OS patching and package updates can be managed also. Integrated with CloudWatch and X-Ray for monitoring and trouble-shooting.
* You still retain full control of all the resources (however you can let AWS manage it for you in “managed updates pannel”).
* When RDS is needed, you can launch one within your env, it’s good for Dev and Test. However, in production you may want your RDS has an independent lifecycle, instead of a lifecycle that’s tied to the env (e.g. RDS will be terminated when env is terminated).
* For production, it’s better to launch directly from RDS, to do this you need to add a security group for your env, and provide the connection string to the stand-alone RDS.
* Choose a platform, upload your code/use a template. Upload code can be a zip file, it’s going to S3 anyways. Once deployed, it generates a “env”, with a detailed configuration panel to modify EC2s, Load balancer, etc.
* Source control is done by uploading another file package in console. Deployment policy is configurable in configurations, and once changed, you can deploy new versions in the application versions page, and new version deployment will follow these policies:
* Deployment policies (important!!):
  + Also called EBS deployment policy, do not mix with elastic block storage
  + All at once policy: deploys the new version to all instances all at once, all the services will be down, and if fails it must roll back by re-deploy the previous version. Better for dev and test, or down time is not critical.
  + Rolling policy: deploys in batches, each batch will be taken out in turn for update. If fails it roll back in batches too, not ideal for performance critical applications where no instance can be down at any given time.
  + Rolling with additional batch: batch but with an additional batch to replace one batch at a time, no down time and no performance impact on the old features, but new features only comes online gradually.
  + Immutable: black/red group, deploy on a completely new group, once finished the whole system failover. If failed just keep using the old one.
* And it handles deletion of all the resources too

**Database:**

To move RDS between VPCs, use snapshots and recreate

Relational vs NoSQL: For relational, you must understand the data in advance, otherwise it is hard to change after, or you have to insert null value. NoSQL has collection (table), document rows, Key Value pairs (fields).

{“key1”:”value1”,”key2”: [ {“key3”:”value3”, “key4”:”value4”}]}

Data Warehousing: Used for BI, tools like Cognos, Jaspersoft, SQL Server Reporting Services, Oracle Hyperion and SAP NetWeaver. Used to pull large set of data for further consumption.

Ways to lower latency: add read replicas, use direct connect, put database closer to you customer or use CloudFront/ElastiCache (Memcached and Redis), switch to redshift (When doing OLAP)

Aurora is AWS's RDB, compatible but better performance than MySQL and better disaster recovery. Other available choices are oracle, PostgreSQL, SQL Server and MariaDB

DynamoDB:

* stores everything as JSON (HTML and XML are also acceptable), the combined key and name length must be smaller than 400k. Great for session/log processing
* Access is controlled in IAM, you can add condition to an IAM policy (in JSON) to restrict a user to only access their own data (Using dynamodb:LeadingKeys)
* Still have table and primary key, has items and attributes in a table, key-value based.
* There are two types of primary key:
  + Partition Key (hash key): unique attribute, e.g. user ID, it determines the physical location the data is stored, so no partition keys can be the same. Maximum 2048byte
  + Composite key: Use a partition key + sort key (range key), e.g. UserID + Timestamp for a bbs post
  + Careful, Partition key is good enough to find records, but you can still use sort key to filter. Sometime questions will offer keys not suitable as partition keys.
* Fully managed and serverless, you can view a lot in the console like Oracle SQL developer
* In the lab it used an EC2 with PHP environment, to run a php script triggered by a browser access.
* Query in CLI: aws dynamodb get-time –table-name [table\_name] –key ‘{“Id”: {“N”:“205”}}’. Carefully specifying datatype is important
* DynamoDB, single digit latency, always on SSD, 3 geographically distinct data centre (not AZ, but still different facility)
* DynamoDB uses optimistic concurrency control
* Consistency model: if need read within one second write, use eventual consistent read, if need less than one second, use strong consistent read
* For read/write capacity unit, you can pay-as-you-go or reserve for 1 year to 3 years. You can monitor in the metrics to compare between provisioned and consumed to change on the fly.
* Scaling DynamoDB does not have down time (called push button scaling), but scaling RDB might have
* You can only create one secondary index at a time
* Local secondary index:
  + Can only be created when you’re creating your table, cannot be changed later.
  + Has the same partition key as your original table, but different sort key.
  + Gives you a different view of your data, organised according to an alternative Sort key
  + Queries are based on ordered sort key
  + Works better when you need the perspective with another sort key.
* Global secondary index:
  + Much more flexible, can be configured later, can use another partition key
  + Speeds up with any query regarding the new partition key also.

Scan VS Query API calls: two ways to fetch data, available from console

* Query: Specifically find an item, can restrict returned results, always sorted by the sort key if there is one, by default in ascending order. Can reverse the order by ScanIndexForward parameter (even this is query not scan…) to false. Eventually consistent, unless explicitly set to be strongly consistent. ProjectionExpression parameter works too.
* Scan: examines every item in the table, by default returns everything, unless refined by ProjectionExpression parameter (basically use this for all filters). Only scan 1 partition (1MB) at a time. You can do parallel scan. You can also restrict scan in a specific table/domain

Query is more efficient than scan, scan costs a lot of throughput, especially as the table grows. To improve performance, you can set a small page size say only return 40 items. But in general you should avoid scan, and you should design you DB for query, get and BatchGetItem APIs.

Provisioned throughput (remember how to do calculations):

* Charged based on read capacity unit and write capacity unit, one read capacity unit can handle 1 strong consistent read per second or 2 eventual consistent read per second, up to 4kB. Eventual consistent read is the default.
* One write capacity unit can handle 1 write per second up to 1KB. So, it can be expensive for write, but cheap for read, because of the unit size.
* So 5 Read capacity unit can handle 20KB/s strongly consistent read or 40KB/s eventual consistent read. 5 write capacity unit can handle 5KB/s write consistent write
* To calculate: round up to the nearest whole number of units for each read/write, then time frequency. Pay attention eventual consistent divided by 2. Careful about if its read/write, and if it’s read, what’s the consistency.

DynamoDB Accelerator (DAX):

* Fully managed in-memory cache for DynamoDB only, provides 10X read improvement, but only for read. Great for bursty workload and read-heavy event.
* Write-through caching only, when data is written to the backend, it’s written to the cache. Allows you to point your API calls directly to DAX. DAX response on a cache hit, otherwise it queries the backend DB. If it is a cache miss, it does an eventually consistent get, load the cache and respond to client.
* This reduce the load on the DynamoDB tables and reduce Provisioned Read capacity.
* Not suitable for strongly consistent read. Obviously not for write intensive, apps do not require heavy read and quick response.

Online Transaction Processing (OLTP) vs Online Analytics Processing (OLAP): OLTP is more like processing online shopping transactions, frequent (high IO) but simple, RDB is mostly used for OLTP. OLAP deals more complicated queries, could be really computation intensive, DWH services has different design to meet this demand. (RedShift)

Data warehousing DB is normally a copy of production DB, so that production DB will not be over pressured

Creation of RDS instance: Select engine -> configuration (instance type, size, storage, name, username and password) -> Backup/encryption/additional settings -> launch

RDS instances never give public IP address, only DNS address. So when using RDS, you never deal with IP addresses.

ElastiCache: in-memory caching, Support MySQL, MariaDB, PostgreSQL, Oracle, and Microsoft SQL Server.

ElastiCache is designed to reduce latency and improve performance, cached information may include read intensive database result or computation-intensive calculation result. Typical use case: Social Networking, Q & A portals and other read intensive jobs. Computational intensive jobs like recommendation engines can benefit too.

ElastiCache vs read replica: cache if a small amount of resource is accessed frequently, read replica is full capacity of RDB is frequently required

ElastiCache types (two opensource engines): Memcached for objects work like auto scale group, and Redis for key-value pairs work like a RDS. Memcached has no multi AZ, Redis has multi AZ, It is basically a CloudFront for cache the most common queries (e.g. top sellers), thus allows better performance for web-apps

Use Memcached when you: store object, need something simple, planning running on large cache node, want to scale out. Memcached is widely adopted, can do Multi-thread, but does not do multi-AZ.

Use Redis when you: looking for storing lists, hash, sets. When you need ranking and sorting, when you need persistence, or need multi AZ, talks pubs/subs. Redis is opensource, support key-value pairs, support master/slave replication too.

2 Caching strategies:

Lazy Loading: Loads the cache when necessary, if not cached query DB and write cache so it will be a cache hit next time. Only the requested is cached, cache node failure will not be fatal, just bring in another empty cache. Disadvantages: cache penalty as cache miss cost time, and data in the database changes, cache will still serve the oud data unless there’s a TTL.

Write-Through: Write cache whenever data is written to DB, so data is never out of date. Disadvantages: write penalty, but users are more tolerating when writing than when reading. When lost it will not be written again, however you can implement with Lazy-loading. Resource waste is also a concern as most of the data may not be read.

RDB instance doesn't have a public IP address either like ELB, only a DNS name, AWS manages the mapping from the DNS to the internal IP address

If EC2 cannot connect to a DB instance, most likely it is due to port 3306 not being accessible to the security group that EC2 is in, you should open port 3306 to EC2's security group (add the security group as a valid source to access for the RDS instance)

Automated backup: enabled by default, stored in S3, size equal to your DB and it’s free, daily snapshot + transaction log, can be used to recover to any point in time (within a second) with the retention period (up to 35 days). In case of recover (manually triggered), AWS restores by apply logged transactions to the latest snapshot. During backup window (configurable), IO may be suspended, and latency might be elevated.

Snapshots: manually done, user initiated, will be stored even after RDS instance is terminated (automated backup snapshots will be deleted when terminated, but manual snapshot is a standalone file), normally comes with several seconds of down time.

whenever an RDS snapshot is restored from a snapshot, it will be a new instance with a new endpoint (DNS address)

Once a DB is encrypted, all data, snapshots, read replicas, and automated backups are encrypted. Encryption is available for all RDS engines (AWS KMS). Encrypting an existing RDB instance is not possible, you can only create a snapshot and encrypt during the copy process.

Multi AZ – For disaster recovery only. Available for all RDS engines and default for Aurora. Any changes made on an RDB in one AZ will be duplicated in another RDB in another AZ. In the case of losing active DB, it will fail over to the duplicated in another AZ. Backup is synchronized, non like read-replica.

In the case of multi AZ, when failing over to secondary RDB, the endpoint (DNS) does not change, AWS handles the mapping to the new private address. It is used for database maintenance too and no admin intervention required.

Read replica: redundancy for performance boost (scaling out). Default for Aurora, available for other RDB but need to be turned on. You can have up to 5 read replica, and you can have read replicas of read replicas (watch out for latency). They can be in different AZ or even region.

Read replica is not available for SQL Server or Oracle, only AWS and open source RDBs, must have automatic backup turned on in order to deploy read replica, each replica has different endpoint.

Read replicas can be used in conjunction with multi AZ so read replica can automatically fail over. A read replica can be promoted into a standalone database for other things, and it breaks replication.

Read replicas do replications Asynchronously

Upon the creation of read replica, you can specify region, AZ and whether to encrypt the database

Multi AZ can be set up when needed in the “modify” menu. Use case: when dev and testing you don not need multi AZ but you need it in production.

Upon deletion of RDS you’ll be given an chance to chance to create a final snapshot.

Redshift can have a single node of 160GB, or multi node configuration with a lead node handling client communication, and up to 128 compute nodes

Columnar data store - Redshift actually store Colum data sequentially on the media, which makes column query more IO efficient

Columnar data store make compression more efficient, since they're all same type of data. You don't have to worry about what scheme to use, AWS select the best for you.

Redshift uses Massively Parallel Processing (MPP) to boost fast performance, achieved by multi-node configuration

RedShift is charged by hour (only compute node, unit is "one instance hour"), and backups, and transfer within a VPC

Redshift is currently only available in 1 AZ, but can restore snapshots to another AZ for potential recovery needs

Aurora (Probably will not be tested):

* bespoke SQL engine for AWS, only works with AWS, cannot run locally, 5X faster than My SQL, 1/10 cost of commercial DB due to read replicas being automatically enabled
* Aurora scaling has down time, but only takes several minutes storage scales in 10 GB incremental, computer scales up to 32 CPUs and 244GB of memory
* Aurora maintains 2 copies in each AZ and in at least 3 AZ, so 6 copies minimum; 15 additional Aurora replica and 5 Additional MySQL read replica is possible but cannot be failed over to.
* Aurora is self-healing, by constantly scanning for error and fix it; Not supported by all regions at the moment.
* When Aurora is provisioned, you can assign priority to each instance, it's like rank in army, when a higher tier instance is down, it will fall over to the highest tier instance working

When the first Aurora is provisioned, it creates a whole cluster and the cluster name is used to generate the endpoint of the entire DB. Always use Aurora cluster endpoint, instead of instance endpoint, unless specific cases.

* Aurora automatic copies are only for recovery and performance (physical media only), the virtual machine running the instance is still a single point of failure, unless using "create an Aurora replica" option
* Aurora replicas can only be used for read but not write, write has to go through the primary instance. In the case of failover, endpoint DNS will remain the same and the new instance will be able to write
* In case of deleting a cluster, replicas must be deleted first before primary instance and the whole cluster can be deleted
* Aurora is designed to transparently handle to loss of 2 copies without affecting the write availability and loss of 3 copies without affecting the read availability

Kenises: fully managed service for bring in streaming data at massive scale to the cloud (IoT, social media, etc), and then become available for processing within a second (for redshift/EMR).

**Serverless:**

Brief history of cloud computing: abstract more and more.

* Data centers (don’t manage it on premise, but still have to talk to people to rack stuff) ->
* IAAS (EC2, still have to worry above infrastructure level) ->
* PAAS (Elastic beanstalk, provision all the stuff to run your code, but still there’s stuff) ->
* Container (ECS, still have to optimize containers) ->
* Serverless (Lambda, code just runs…)

RDS except serverless Aurora, are not serverless.

Architecture of A Could Guru:

* Angular frontend
* Auth0 for authentication
* S3 + Cloud front for supply video
* Firebase for real time streaming (Owned by Google, so it’s multi-cloud)
* Stripe for credit card payment
* AWS API -> Lambda handles everything else

Lambda:

* Amazon uses this for echo/Alexa, task it can do is called a “skill”, a skill can have a maximum of 3 words as name. Audio stream cannot exceed 90 seconds, response invokes Lambdas ARN. Maximum run is 900 seconds.
* Can copy code into IDE, use CloudFormation or upload zip file to upload code.
* Can be event-driven (trigger, better to remember all types: API gateway, AWS IoT, Alexa Skills Kit, Alexa Smart Home, CloudFront, CloudWatch Event, CloudWatch Logs, CodeCommit, Cognito Sync Trigger, DynamoDB, Kinesis, S3, SNS)
* Can respond to HTTP request with API gateway
* Supports C#, Python, Java, Node.js, go
* Charged based on request, 1M per month free, $0.2 per M thereafter and for computation time, rounded to 100ms, charged 0.00001667 per GB-second
* Continuous scaling (out, not up), unless there’s an outage
* Can get complicated, debug uses X-Ray
* Can do things globally, so for services that’s regional, use Lambda for global capacity. However, Lambda itself still exist in only one region
* Three ways to create Lambda: Author from scratch, Blueprints, Serverless Application Repository (A bunch of pre-built function for you to use)
* Minimum access needed: Simple micro service permission, this allows Lambda to run. If interaction with other resources is needed, you need to assign Lambda a more powerful role. Resources a Lambda has access to will show in the panel upon attaching the policy
* Lambda can take in environment variables (parameter files)
* Can be tested in cloud IDE by feeding input straight away
* Lambda timeout can be set for heavy tasks, maximum is 5 minutes. Time not enough may result in task being partially completed.
* You can set up testcases, like in leetcode.

API Gateway (under network and content delivery)

* Imagine a waiter in a restaurant, serves as an API between customer and chef
* publish, maintain, monitor and secure APIs, few clicks to create a door (waiter) to backend
* Can connect to DynamoDB, EC2, and of course, Lambda
* RESTful (Representational State Transfer, JSON)
* SOAP (Simple Object Access Protocol, from late 90s, xml)
* API Gateway can create a http endpoint for RESTful API
* Can also send different endpoint to different target
* You can create, clone exist or use example API, you can also import from a swagger v2.0 file
* Can Track and control usage with API key (no key, no response)
* Can use throttle keys to prevent DDoS attacks.
* Default throttle limit is 10000 per second and 5000 concurrent requests, if over the limit will get a 429 too many request errors, down to million second. But you can lift these limits in console for a charge.
* You can use API Gateway for SOAP pass through to package your legacy SOAP API
* Monitor with CloudWatch, maintain multiple versions of API
* Process: Define API -> Define Resources and Nested reduces (URL path) -> Configure -> HTTP method for each resource -> set security -> choose target -> set request and response transformation -> Deploy to different stage (dev, test, etc.) -> API gateway domain by default, but you can customize (hide AWS logo). Route 53 domain gets free SSL/TLS cert
* API can cache, has a time to live, can be stage specific, if cached will not look up backend. This can bypass the throttle limit also.
* API has to be deployed manually when configured
* CORS is important, check again!!!
* Error - “Origin policy cannot be read at the remote resource”, you need to enable CORS on API Gateway typically with AJAX/JavaScript, enforced by the client

Version Control with Lambda:

* Can publish multiple version for multiple stage, published version cannot be changed
* Latest version has $LATEST, the ARN with it is a qualified ARN, otherwise it is an unqualified ARN. Only the version with $LATEST can be changed.
* Alias name can be used to point to different versions, so when another version is needed, you can just change the pointer. ARN will end with :Alias\_name. This way you can have a dev, test and prod pointing to different versions.
* Version number increases in sequential order, number is assigned upon “publish version”.
* You can have an Alias to point to a split of different version, but only split two ways.

X-Ray (under developer tools)

* Way of visualizing serverless app, for debugging and optimization. Tells you how long each service takes on average and how many requests per seconds for example.
* Provides: interceptors to trace incoming http request, client handler to monitor how you call other AWS services, HTTP client to monitor how your app calls external http services.
* You have to code with X-Ray SDK, which send JSON to X-Ray Daemon. Daemon sends to X-Ray API, and API sends info to X-Ray console for visualization.
* Integrates with ELB, Lambda, API Gateway, EC2, EBS, mainly to see the traces.
* Supports all language supported by Lambda, plus ruby
* X-Ray need a role to be able to log a comprehensive picture

Step functions

* allow you to analyses and debug serverless applications in steps, like a finite state automata, uses amazon step language.
* Great way to visualize your application. When something go wrong you can know straight away. It’s part of application integration, not Lambda. It reflects application status in real time.

**Compute:**

Re-sizable compute capacity in the cloud

Supported virtualisation type: Paravirtual Machine (PVM) and Hardware virtual Machine (HVM)

Runs on hypervisor, which run on host hardware. ec2 class/type naming: i3.xlarge, “i” means class, 3 means generation, xlarge means size, generally follows 2X rule. Letters for instance types never change, but generation might change.

EC2s are launched from AMIs (Amazon machine images). Images are maintained by both AWS and community; marketable ones can be found in AWS marketplace. The current recommended version is Amazon Linux 2, which uses yum as its default package manager

EC2 has to have “auto assign public IP” ticked, in order to have a public IP.

Instance type use cases to be added (Not tested in associate except sysop):

Fight DR MC PX….

* M5: multipurpose, general purpose, Skylake CPU
* T2: General Purpose Burstable instances, cheapest, can be bursted a bit but for limited time. T2 unlimited can be bursted indefinitely
* R4: RAM optimised, 8:1 memory ratio
* X1/x1e: 2T memory, super memory optimised, 16:1/32:1 memory to CPU ratio
* I3: I/O intensive, 3.3 million IOPS, cheapest per IOPS
* EC2 Bare Metal: All support (disc management, networking, etc) are offloaded, so no hypervisor needed, can be used as dedicated host
* D2: lowest cost per unit storage, using magnetic storage, for DWH, log processing, etc. High throughput application
* H1: more CPU, less storage D2, for big data, MapReduce, application need processing a lot but don't store. cheaper than D2
* C5: compute intensive, custom Skylake, 2:1 memory CPU ratio, gaming, scientific, video encoding etc. Netflix uses it
* G3: graph intensive, parallel computing, huge number of cores, can do ML, AI, scientific research (weather, air dynamics)
* New elastic GPU: attachable GPU pay as you go, OpenGL compliant
* P3: Powerhouse, GPU based VM, Nvidia Tesla chip, for HPC (high performance computation)
* F1: FPGA (Field programmable gate array), ability to program the hardware, thus " circuit gate array that's programmable on field", bake algorithm onto hardware. images available in marketplace. FPGA functions are highly independent (helps parallel processing), highest performance compare to a micro controller, High frequency trading use them too. VHDL and VERILOG language. Custom data width (not just 32/64 bit) and operation is all possible (3bit computer)

Reserved instance: convertible instances is possible (change instance type on the fly) if you spend more. Reserved instance can be scheduled too (Only use at the end of week for example).

Spot instance: as soon as spot price goes over you bid it will be terminated, you won't be charged if it's terminated by AWS. Requested upon launch EC2 instance, price shown straightaway.

Dedicated host: to deal with stupid regulation that against multi-tenant virtualisation. Can be on-demand or reserved. Once set up both VPC and its instances cannot be changed back to default hosting.

On demand use cases: sudden need, testing/dev, short term project, etc.

EBS used to boot OS is the root drive, root volume cannot be encrypted unless use third party tool (bit locker for windows for example) from EC2 OS or turned into snapshot first and then copy it somewhere and encrypt it in the copy process. Additional volume however, can be easily encrypted You cannot detach root volume and attaches to another instance.

EBS classes (All IOPS are per volume):

* General purpose SSD(GP2), 3IOPS per GiB, minimum 100IOPS and 3000 IOPS on 1TB disk. can do up to 10000 IOS, but more likely 3000
* Provisioned SSD (IO1) can go up to 20000 IOPS
* Throughput optimised HDD (ST1, for large data, log, cannot be boot volume)
* Cold HDD (SC1, file server, for archive, cannot be boot volume)
* Magnetic (Can boot, cheapest, on the way out, not recommended)

bootup script can be applied, can be in the form of plain text or file

By default, the EBS attached to EC2 are detached and deleted on termination. When detached it is automatically un-mounted, however attaching it does not automatically mount. Only stored redundantly in single AZ.

Commands to config additional EBS volume:

* lsblk
* file -s /dev/xvdf (check if the volume has a file system, returns data when there’s no data, returns file system detail otherwise)
* mkfs -t ext4 /dev/xvdf (creating file system)
* mount /dev/xvdf /mountdir (mount EBS with file system to a mount dir)
* umount -d /dev/xvdf (un-mount, mountpoint would be cleared in lsblk)
* API call is called AttachVolume

By configuring a security group for an EC2 instance, SSH IP can be restricted down to only allow “my IP” address from the terminal, but you may lose access after several hours when ISP changed your IP. For SSH logging on from mac/Linux, use SSH ec2-user@publicip -i keypair.pem. If using putty, username can be stored in Putty from the beginning.

To terminate an instance with protection, go to settings disable the protection first

System check: infrastructure is alright

Instance check: if instance OS works

Security group is layer 4 virtual firewall, any new rules will be applied immediately, It’s first line of defence for EC2

By default, all inbound is blocked, all outbound is allowed. Any inbound rule automatically introduces outbound rule (being Stateful)

You can only white list inbound with security group, black list need NACL

Because you cannot black list with security group, you can assign more than one group to one EC2

EC2 instance and EBS must be in the same availability zone, otherwise the latency won't make it work. EBS volumes now can be modified on the fly (size and type), which wasn't possible before

To deploy in another region, copy the image to that region, you can encrypt in the copy process

EBS Snapshots/ EC2 Images (AMI):

* Snapshot are for volume, exist on S3, snapshots are incremental.
* To migrate EBS instance to another AZ, create a snapshot, store as image (AMI), and re-implement in the other AZ.
* Best practice is only taking snapshot when EC2 is stopped so everything is flushed, but it is possible to take one when EC2 is running
* If taking a snapshot of a RAID array, you must: freeze file system, unmount RAID and shut down EC2 then take snapshot.
* Snapshot vs image, image is for the whole EC2 including OS and license and directly launchable, snapshot is only for one volume. Images can be created from root volume snapshots. Image is directly launchable.
* Snapshot of encrypted volume is encrypted, restored volume from encrypted snapshot are encrypted. Once encrypted, you cannot undo it. You can choose default or your custom key
* Encryption of additional volumes has to be configured from the beginning
* Encrypted snapshots cannot be shared, because key is controlled by AWS account.
* There's a lot of special AMI for sale when launching an EC2, launching from your own image is also an option
* AMI is a great way to store your common steps for a solution, so when you launch an instance it works straight away. AMI based auto scaling is basically auto-heal, highly recommended
* EBS backed instance has the option the keep the root volume, instance store doesn't have this option
* Instance store might be cheap, considering IO cost and volume. But it really is not used a lot.

Load Balancer (Within EC2 panel):

* Types: Application (intelligent, layer7 for routing different requirement HTTP(S), can even make routing decisions, sending specific requests to a specified server), Network (layer7, for performance but more naive, super-fast, can handle millions of request per second but expensive, TCP), classic (also called ELB, layer 4 or layer 7, not as good as Application one, legacy, don’t use), can cross AZ but not regions, Multi AZ is selected upon creation
* Load Balancer error 504: balancer is still there, but connection to EC2(gateway) timeout. This is based on health-checking a specific page (say index.html).
* When using a classic ELB, User IPv4 address can still be passed in within the X-Forward-For header so you can look it up from your EC2 server and know who accessed your page
* ELB can launch instances from instance template
* The EC2s it balances are called a target group
* It is possible to load balancing subnets, but it must be no more than one subnet per AZ, and at least two AZ.

When trying to access S3 from a EC2, it's fine if they are in the same region, if not, use the region flag

AWS CLI:

* Can install separately as a stand alone tool, Amazon AMIs has it installed
* Commands like: aws [service] [command], for example:
  + aws s3 ls
  + aws s3 mb s3://bucket\_name
  + aws s3 cp example.txt s3://bucket\_name
* If the EC2 you’re using CLI from does not have a role, you must create a user with an accese key, then use the generate public and private access key with the command: aws configure
* When access key is made invalid from console, it becomes invalid instantly in command line.
* Commands can be looked up from official documentations

Remember the URL to check the metadata of a EC2: curl http:169.254.169.254/latest/meta-data/ (curl is a command line tool to fetch a URL)

Auto Scaling:

* use launch configurations to config standard instance, instances will be deployed evenly in different AZs. However, to change configurations of EC2 you must start over
* Spec is defined in launch configuration
* Auto scaling group typically receives traffics from load balancer
* Health check grace period: when to perform health check again after scaling is performed
* Auto scaling happens only in one region, multiple region must use Route53 to config

Placement group: multiple EC2 running close to each other for faster interaction, only certain type can be used, AWS recommend using identical instances. Separate: still close, but different hardware, multi AZ, clustered: one AZ; they all cannot merge, cannot move in instances, do not span VPC.

You can run container still works on EC2, just don't have to worry about OS.

**IAM (universal service, Security group and ACL don’t span VPCs):**

Can creating users and granting access, gives centralised control

Amazon Cognito Identity federation (Or identity pool, Granular Permissions):

* Grant temporary access to AWS resources when successfully authenticated by other web-based identity provider, this could be Facebook, google, or your custom user pool.
* When authenticated, user receives a token (authorization code grant) to trade for temporary access to AWS resources. Cognito handles the brokage. This happens on a on-demand basis so no credentials should be pre-set or stored.
* Key features include:
  + Sign in to your own app
  + Support guest user
  + Broker between your App and ID provider
  + Sync user data for multiple device
  + Recommended if your app uses AWS (answer this when asked what’s recommend)

Cognito User Pools:

* User sign in to User Pools via their own credential or Facebook, Amazon, Google, etc, Cognito trade the success for authentication for JSON Web tokens (JWTs, Implicit grant)
* Then Users use JWTs to trade for temporary AWS credentials
* With user pools you can set up your own identities, e.g. user registration
* Cognito uses Push Synchronization to push updates across multiple devices (utilizing SNS)

Identity federation vs user pool: Weather you need handle identification yourself, or just need to grant temporary access.

App-client (who’s going to use this user pool) setting in user pool setup:

* Call-back URL: the URL to go to upon successful sign-in, for testing you can use https://example.com
* Sign out URL: the URL to go to after sign-out
* Allowed Oauth Scope: phone, email, etc. (will be verified upon creation)

Password rule can be setup upon creation of the user pool. Cognito URL is customizable under “Domain name”, Cognito even provide a service to customization of logos for the login page (button, banners, etc.)

When user pool is successfully set up, Use the link [https://[Domain\_name]/login?response\_type=token&client\_id=[example\_client\_id]&redirect\_url=[redirect\_url](https://[Domain_name]/login?response_type=token&client_id=%5bexample_client_id%5d&redirect_url=%5bredirect_url)]

And it will jump to a login/sign up page, and from here everything will be handled by Cognito. Once a user signed up, you can see these users in “Users and groups” under Cognito. You can also group users and assign different access from here. You can also assign Cognito groups with IAM roles.

Under federation you have a choice to federate Facebook/Google/Amazon identity in a push-button manner. But it’s beyond scope of the exam.

Delete order: delete domain -> delete pool

Can do multi-factor authentication too, and it is recommended

Provide temporary access for users/devices and services. When AWS access has to depend on another set of credentials, create a federation proxy or identity provider and then use AWS security token service to create temporary tokens.

Manages password strength and their rotation policy (expiration, reuse, etc.)

integrates with other AWS services

Has User, Group (for users only that need identical access, users can inherit access from a group policy), Roles (for resources, including users), both group and roles use policies (JSON with access granted or denied)

When a new user is created, they have no access to anything by default, not even changing password, unless "require change password when login" is ticked or password management access is granted

Roles have types, for AWS services is different from say for AWS account/Web identity/SAML federation. When trying to attach/Replace roles, only the types that could be applied will show up. When a role is attached, it become effective instantly.

When clicked into a user details, all policies attached for any reason (group, role, inline) will be displayed.

Creating group is a recommended practice. A user can belong to more than one group.

When creating a resource in AWS, create a role to lock down low limit on what it can do

policies: A JSON document that defines permissions, then attach to user/group/roles. A user can have multiple policies, say one inherited from group and one attached directly for him (or inline). When policies contradict each other, denies overwrites allow, and AWS always allows only minimum access (align with security best practice). A policy comes into effect straight away.

* Managed policies: policies managed by AWS for common use cases, those you can search and apply in console. Saves you effort since they are typically complicated, cannot modify. If you need modify, create a custom one. AWS occasionally change it.
* Customised: recommended to start with a copy of managed policy
* inline: not re-usable, only for the specific entity, and will be lost when entity is deleted. Recommend for temporary use, entity specific use, etc.

It is recommended to only login to root when absolutely needed. Only one MFA is allowed for root, if MFA is lost you need verification process to get MFA removed.

Access key ID and secret access key for users can only be used in CMI or SDK, can be deactivated by root, for console access always user name and password. Private access key can only be acquired once upon creation. Access key pair must be regenerated if lost, cannot be recovered.

Create one key pair for each user is recommended (so when one is compromised you don’t have to replace all of them).

When key pairs are uploaded to public (say as part of your code to GitHub), assume it is compromised straightaway. It’s used just as username/password pair through CLI.

Account is used to uniquely identify an AWS account, it’s hard to remember so can be customised to anything, as long as it is universally unique.

Cross account access: Say allowing dev account access production account as read-only

* Create a developer in dev account, associate with a new group (say cross-developer)
* Get the account ID of dev account
* Log into the Production Account and create a policy which only has access to the resource that developer is allowed to see, which will then be attached to cross-account role in production account.
* Create cross account role, which is a special kind of role called “role for cross account access”, add the account ID of the developer account, attach the cross-account policy
* Give the new group in developer account an in-line policy, make it able to Assume Roles in IAM: production account: role/role for cross account access
* Logon to developer, select switch role from top right corner, give account, role name and switch
* You’re in production account with “role for cross account access”

**KMS:**

Under IAM, Keys has regions, not global.

You can create keys with a name and specify if you BYO or use KMS (called key material option). In the process you can specify what group can manage and/or can use the key. The user of the key can be an external account, but manager cannot.

KMS master key cannot be exported, when needed use Cloud HSM (used to be very expensive like $5000 per month, but now can pay by hour, dedicated device, KMS is multi-tenant).

API calls (important!!):

* Login to an EC2, configure user with user credentials of a KMS key user
* Aws kms encrypt –key-id [key-id] …….
* Aws kms decrypt –ciphertext-blob ……. (doesn’t require key because it’s bundled with user)
* Aws kms re-encrypt –destination-key-id [key-id] ……. (encrypt again)
* Aws kms enable-key-rotation

KMS envelop encryption: KMS master encrypts envelope key (or data key), envelope key encrypts data. Decrypt process just decrypt to get plaintext envelope key first and use it to decrypt data.

Deletion of key can only be done by schedule deletion, cannot delete instantly anyways.

**Storage:**

S3(global namespace, key-value based, but has a region when created):

* File size can be from 0 to 5TB, the is no limit for total storage, consider bucket as a folder except it has a universal namespace. One account can have 100 buckets at most.
* Read after write consistency happens almost instantly, eventual consistency can take seconds to sync through availability zone (only for overwrite, not new puts, S3 default setup, not user setup, not available for One Zone obviously)
* Name can be lower case letter, periods, numbers and dashs
* Objects (functionalities related these features can be done to object level):
  + Key (name)
  + Value (data)
  + Version ID (for lifecycle also)
  + Metadata (pre-defined tags, e.g. content type, expiration, etc.)
  + Sub resources: Custom tag, bucket policies (bucket level), ACL, cross origin resource sharing, transfer acceleration
* Standard built for 9\*4 availability, guarantee 9\*3. 9\*3 availability for IA, guarantee 99%, One Zone IA only have a 99.5% availability, guarantee 99. They all have 9\*11 durability. One zone IA is about 20% cheaper than standard S3
* S3 standard and IA can survive 2 AZ lost (stored in 3AZ), one zone - IA cannot, if the zone is lost data is lost, so better use for re-producible file (still stored redundantly but only one zone). IA charges retrieval fee by GB.
* Storage class can be selected upon object uploading
* Can create access log and can output to another bucket. For auditing/analysis.
* When using the “Open” button in S3, it uses your AWS console credentials, so you have access to private objects. If you use URL, it launches a separate page and can only access public file.
* Policy can be created with policy generator. Choose one that’s as close as possible and modify. “Principle” in policy generator means the initiator of the action.
* Private objected can be accessed via a pre-signed URL, even it is accessed directly from the bucket
* When trying to upload a file that is not allowed, it gives an upload error “forbidden”.
* Cross origin resource sharing: For one S3 object to refer to another S3 object, say in static hosting, you can use index page in one bucket to call an image in another bucket (just better organising files). Use static hosting endpoint to get the file address, use “allow origin” in the referrer’s account and refer to the file. Always use the website URL (the one ended with amazonaws.com), not the regular bucket URL (the one ended with /bucketname).
* RRS: if comes up it is used for things that can be easily re-created. It sacrifices durability to 99.99%. Used for thumbnails, transcoded video, sacrifices durability. It's the most expensive and nobody use it now.
* Pricing: Storage, request (data out), storage management (tagging), cross region transfer(replication), and transfer acceleration (CloudFront + backbone) cost money
* S3 transfer acceleration: Users using edge locations and AWS backbones to upload files to Bucket, URL: S3-accelerat.amazonaws.com/
* S3 performance: S3 can stand 3500 puts request and 5500 gets request per second since July 2018, so you don’t have to optimise before the demand reaches this level. In the old days when constantly facing 100 PUTS, DELETS, LIST or 300 GETS per seconds, you should optimise by:
  + For GET intensive work, use CloudFront
  + For non-GET intensive work, because S3 use prefixes of keys to decide what physical partition data will be stored, so using random prefix (maybe a hash) on the S3 object key will distribute I/O workload. It doesn’t have to be a prefix, just adding randomness would do.
* Standard URL: bucketname.s3-web-reigion.amazonaws.com s3-web-reigion.amazonaws.com/bucketname
* Glacier: expeditated 3-5 mins, standard 3-5 hours, bulk 24 hours, IA and standard are million seconds. IA doesn't mean slow, just more expensive if access too frequently
* Bucket name must be unique, and it's associated with a DNS address
* You can give other AWS account S3 access, and do cross-account logging, commonly used within organizations
* Remember if successful from CLI or API you'll get a HTTP 200
* Client (encrypt then upload) or Server-side encryption may be used, down to object level. In transition, it uses SSH/TSL using https. (TLS is replacing SSH)
* Server-side encryption: SSE-S3, SSE-KMS (key management service), SSE-C(customer), and SSE plus Client-side encryption are the four encryption types.
* You can enforce all files to be encrypted, by setting up a policy to check if a file has the encrypted header and denies all PUTS request without the header upon uploading. And this enforcing works even default encrypt is not selected. To enable this kind of enforcement, use “add condition” option in the policy generator when creating the allow rule.
* When SSE is enforced, you must select the encryption option when uploading, otherwise you’ll get a forbidden error.
  + SSE-S3 uses AES256, with a S3 managed key, and key is encrypted with an Amazon managed master key which is frequently rotated. Will have this header included upon uploading: x-amz-server-side-encryption: AES256
  + SSE-KMS, more advanced, has the option to create envelope key to encrypt your main key, and can trace log of all encrypting and decrypting related activities, like who used key and why. Will have this header included upon uploading: x-amz-server-side-encryption: ams: kms
  + SSE-C, AWS manage encryption and decryption, and you manage your own key
  + Client-side encryption: you encrypt the file and then upload
* By default, the bucket and all objects are private
* S3 objects do not inherit bucket tags
* When a public file gets updated, it stops being public by default. Remember this because when hosting a static website on S3, all linked files must be public.
* S3 can be used to build simple FTP application
* For S3 you don't have to handle load balancing or other infrastructures, it scales.
* For static domain hosting, domain name must be the same as bucket name if the website is going public, traffic cost is negligible. Static website URL has “website” in it, not standard bucket URL
* S3 supports websites redirects
* Buckets are ordered in alphabetical order, or lexicographically.
* S3 error message are in xml by default
* Server/Object Logging, versioning and server-side encryption (SSE-S3 and SSE-KMS) are set up upon creation. You can also give access to other AWS accounts upon creation
* S3 versioning:
  + Keeps every single version of the tracked file, for big files it can be expensive.
  + S3 versioning can only be suspended but cannot be disabled once enabled.
  + S3 versioning work in the form of overwriting same name objects.
  + S3 versioning integrates with lifecycle rules
  + S3 can config MFA delete. S3 will put a delete marker if a file is deleted, and when delete marker is also deleted, the file will be restored.
* S3 cross-region replication:
  + Cannot be applied in the same region, and obviously no need.
  + Replication can be in another account, say there's a backup account for the entire organization.
  + S3 cross-region replication can be sub-folders or entire bucket. Requires versioning enabled on both buckets. Can be glacier if it's just a backup, or it can be any S3 storage class, commonly S3-IA.
  + Upon the creation of a replication bucket, existing files won't be copied, only new one will be replicated automatically. So existing files must be copied through. Easiest way to copy the existing files to replication bucket is to use recursive copy in CLI.
  + When object is deleted, replica bucket doesn’t delete it to protect from malicious deletions
  + The deletion of delete marker won't be synced anyways
* S3 Lifecycle management:
  + Doesn't require versioning but can manage current and previous versions separately.
  + S3 Lifecycle management can be used when say data is only relevant for a certain period.
  + You can setup different tiers of storage for each stage of lifecycle, S3 -> S3 IA -> Glacier -> delete(expired). Lifecycle rules can be set differently for current and previous versions.
  + When an object is transitioned into glacier, it remains a reference in S3 to trigger restoration. However, if an object is directly stored into glacier, you must download and re-upload

EFS:

* Elastic version of block storage, redundantly stored in multiple AZ. Uses network File System version 4 protocol, can support thousands of concurrent NFS connections.
* EFS and EC2s using it must be in the same security group
* Mounting instruction and command are provided in the console, can be mounted on any location, just run the provided command in terminal (maybe replace where you want to mount, can be any folder)
* EFS can be used to centralised mountable storage managing your application content, just like it's stored in EBS, so no need to copy like S3
* But EFS is still a file server, with permission management. It's better than EBS because can be mounted to multiple EC2s at once. So, can have 2 servers serving out the same file, and don’t need multiple copies of data.
* When used in conjunction with CloudFormation, just mount an EFS to any EC2 that’s provisioned by auto-scaling group, and it became a server without running any batch.
* EC2s connect to EFS through mount targets in each AZ, therefore it is recommended you have a mount target in each AZ.
* Can give EFS file/directory level permissions. Can encrypt files and have multiple performance level (General, Max I/O) and throughput mode (Bursting-Provisioned) available

CDN and CloudFront

* CDN is not quicker for the first user, and edge location only cache data for a certain period (TTL, time to live). TTL should not be too long for frequently changing objects, and decommission objects cost money (done through invalidation panel in the console). TTL is set by the object being cached
* CloudFront work with non-amazon, even on-premise services too, just set your source instead of using an AWS resource as "origin"
* A distribution is the name of a collection of edge locations (like a distributor), and it is the first thing you create in CloudFront. A distribution is treated as a resource, with unique ARN. One distribution can have multiple origin. Note: distribution must have access to origin to make it work.
* Because a distribution works across multiple regions, it takes about 15 mins to provision (push the configuration to all locations). It works across all edge locations by default, unless you Geo-restrict it
* Geo-restriction can be black or white list
* You can specify connection type: HTTP and HTTPs, redirect or HTTPs only
* Whether or not allowing uploading is done by allowing or denying HTTP PUT
* There’re two different types of distribution, RTMP (real time messaging protocol, for adobe flash/media streaming) and web distribution (anything that can be distributed with HTTP/HTTPS, websites, both static and dynamic content)
* Once CloudFront is setup, you can block direct origin access, and use distribution only. This means all request must go through edge location.
* Upon creation it will generate a domain for accessing the distribution
* Cloud can filter user by only accepting sign URL or signed cookies. (Use case: secure CloudFront by only allow paid customer to access) However, if geo-restriction is activated, even signed URL won't work, because the door’s not there even you have the key.
* WAF works with CloudFront nicely, to protect from SQL injection and cross site scripting, not network layer but rather application layer.
* You can configure a “path pattern” to restrict access to certain folders
* You can configure customised error page whenever origin returns 4XX or 5XX error code
* You can bring your own C-NAME
* Delete order: disable distribution (takes another 15mins) -> delete distribution
* CloudFront has global load balancing facilities and works with all kinds of content.
* CloudFront allows Geo-distributions, black or white list.
* Even when origin is disabled, CloudFront can still serve cached content
* Remember S3 transfer-acceleration utilise CloudFront edge locations too

Storage gateway: must download onto VM in your datacentre, consists of a local storage/cache and a buffer that handles multi-part upload. works as an attached file system for S3

Types: File (NFS), Volume (Stored/Cached, iSCSI), Tape(backupVTL)

File gateway backups local datacentre to S3

EBS are not encrypted by default, but can be encrypted.

Stored volume backups EBS snapshots to S3, a complete copy is kept locally, maximum 16 TB. Cached volume only keeps recent state locally and rest in S3 as EBS snapshot, Maximum 32TB. Tape backups physical tapes to virtual tapes.

Snowball: Once Import/Export, reinvented for better management. Bypassing internet for large data set, snowball/snowball edge/snowball mobile

Snowball can import to or export from S3, Glacier must be restored into S3 first to use snowball.

Standard Snowball: 50/80 TB size, 256bit encryption, erased after data received and verified. Snow edge: 100TB, has compute capacity (use like a black box on plane). snowmobile: 100PB transition in truck for data centre migration

Snowball needs a local software client (CLI) to access, and credentials can be obtained from the console. You need manifest file and private key to access snowball.

Connect through ethernet and It works as the S3 buckets, except the physical device is local. You can compare performance before and after in different regions with showcase link, the further the faster

Multipart upload is faster and is a must for single file bigger than 5GB (limit for PUT). It is recommended for anything larger than 100M as it can:

* Improved throughput - You can upload parts in parallel to improve throughput.
* Quick recovery from any network issues - Smaller part size minimizes the impact of restarting a failed upload due to a network error.
* Pause and resume object uploads - You can upload object parts over time. Once you initiate a multipart upload there is no expiry; you must explicitly complete or abort the multipart upload.
* Begin an upload before you know the final object size - You can upload an object as you are creating it.
* Objects are not reassembled until the CompleteMultipartUpload operation is called

Check:

S3 request header

HTTP response https://docs.aws.amazon.com/AmazonS3/latest/API/ErrorResponses.html

S3 naming rule

DynamoDB batch write/read

Application load balancer can handle more than one SSL certificates