A. Questions from Assignment 1 [25 Points]

1. What are three advantages to using Swagger?

- API contract is explicit and standardized (OpenAPI) as a single source of truth.
- Interactive, tryable docs (Swagger UI) speed exploration and reduce ambiguity.
- Client SDKs and server stubs can be generated to accelerate development.
- Mock servers from the spec unblock frontend/mobile and enable parallel work.
- Automated request/response validation and contract tests integrate into CI/CD.
- Versioning, linting, and reviews improve governance and API consistency.
- Faster onboarding with clear examples and consistent, discoverable documentation.

2. When you call the API from your frontend application hosted at http://www.yourdomain.com, you get the following error in the developer console: "No 'Access-Control-Allow-Origin' header is present on the requested resource. Origin 'http://www.yourdomain.com' is therefore not allowed access."

> What is the reason for this error? How do you resolve it? (describe every step)

- The browser is enforcing CORS: your API response lacks `Access-Control-Allow-Origin` permitting `http://www.yourdomain.com`, so the browser blocks it for security.
- Enable CORS on the API: if using API Gateway, turn on CORS for the resource/methods (this adds `Access-Control-Allow-Origin`, `-Methods`, `-Headers`).
- Add an `OPTIONS` method: configure preflight handling with allowed methods (e.g., GET, POST), allowed headers (e.g., Content-Type, Authorization), and allowed origin(s).
- Return headers from your backend: if using Lambda proxy integration, include CORS headers in every response (success and error), e.g., `Access-Control-Allow-Origin: https://www.yourdomain.com` and optionally `Vary: Origin`.
- If sending credentials (cookies/Authorization), set `Access-Control-Allow-Credentials: true` and use an explicit origin (no `*`), and ensure the frontend uses `fetch` with `credentials` as needed.
- Deploy the API changes to the correct stage and test preflight and actual request with curl/Postman and the browser DevTools Network tab.
- For custom domains/CDN, mirror CORS at the edge only if origin can't set headers; otherwise prefer origin to be source of truth and keep allowed origins

tight (avoid `*`).

3. You create a new Lambda function that returns the text "Hello world". You test it and it completes the execution in a few milliseconds. You then decide to use it to store a message in a DynamoDB table. You add code to do just that, on top of the original code. Upon testing the new version of the function, it times out after 3 seconds, over and over again.

> What might be the issue and how can you resolve it?

- Most likely networking: the function is placed in a VPC without internet/NAT or without a DynamoDB VPC gateway endpoint, so calls to DynamoDB hang until timeout.
- Fix networking: either remove the Lambda from the VPC (if it doesn't need VPC resources) or add a DynamoDB Gateway VPC Endpoint to the route table used by the subnets.
- If using NAT for public AWS APIs, ensure private subnets have default routes to a NAT Gateway and security groups/NACLs allow egress to AWS endpoints.
- Verify IAM permissions: attach a policy with `dynamodb:PutItem` (and needed actions) on the target table; missing permission usually causes immediate AccessDenied, not timeouts, but confirm anyway.
- Initialize the AWS SDK client outside the handler to reduce per-invocation overhead; ensure you `await` the `PutItem`/`PutCommand` promise to finish.
- Check the function timeout setting; increase it temporarily to diagnose, then right-size after networking is fixed. Add logs around the DynamoDB call to pinpoint where it blocks.
- Confirm region alignment: the Lambda's region and DynamoDB table region should match to avoid cross-region latency and endpoint resolution issues.

4. You have an SQS queue with 100,000 messages in it. How would you go about reading the messages on AWS? Please be descriptive in terms of the infrastructure and the permissions needed.

- Choose a consumption model: either event-driven Lambda triggers on the queue or a fleet of EC2/ECS workers polling with long polling (`WaitTimeSeconds`).
- Configure batching and concurrency: set `BatchSize` (up to 10 for Lambda), tune `maximum concurrency`/worker count to drain backlog quickly without overloading downstreams.
- Set an appropriate `VisibilityTimeout` > (processing time + retries) so messages aren't prematurely re-delivered; use `ChangeMessageVisibility` if

processing varies.
- Ensure idempotency: deduplicate via message IDs, idempotency keys, or conditional writes so retries don't produce duplicate side effects.
- Add a DLQ with redrive policy: after N failed receives/process attempts, move to DLQ for inspection and remediation.
- Grant least-privilege IAM: consumers need `sqs:ReceiveMessage`, `sqs:DeleteMessage`, `sqs:ChangeMessageVisibility` on the queue; producers need `sqs:SendMessage`.
- Monitor and scale: use CloudWatch metrics (ApproximateNumberOfMessages, AgeOfOldestMessage) and autoscale workers; consider FIFO if order/exact-once is required.

5. For the following questions, imagine you have the following resources on AWS: an API Gateway deployment, a Lambda function connected to one of the API Gateway methods that responds to each request, and a Lex bot called "Concierge". You create a new intent called HotelBookingIntent in Lex. You add a few utterances and build it. When you start testing it in the Lex test console, everything works as expected. You then write code to call your Lex bot from a Lambda function, but all the incoming messages are not understood by the Lex bot.

> What is the most likely reason for Lex failing to understood messages sent through the Lambda function, yet everything working correctly through the Lex test console?

- Likely a version/alias or locale mismatch: the console uses `$LATEST`, while your code calls a published version/alias lacking the new intent or uses a different `localeId`.
- For Lex V2, confirm you pass the correct `botId`, `botAliasId`, and `localeId` that contain the trained intent; for V1, publish and update the alias used by your code.
- Ensure your Lambda sends plain user text to `RecognizeText` (V2) or `PostText` (V1); wrong payload shape/slots/session state can cause fallback.
- Verify input mode: if sending audio, use the correct API (`RecognizeUtterance`) and content-type; otherwise Lex may not interpret it.
- Check IAM permissions: the Lambda role must allow `lex:*Recognize*` for the specific bot/alias; AccessDenied may mask as failures.
- Rebuild and republish after adding utterances; then update your alias to point to the new bot version and redeploy your Lambda/API Gateway stage.

- Inspect Lex logs/analytics for missed utterances and session state to confirm the bot version/locale and recognize patterns.

6. You have a payment API that calls a third party API service to make payments. The API receives millions of requests per hour. At some point, the payment service, starts failing intermittently due to the high volume of transactions.

> How would you re-architect the system to ensure that all payments are processed without an impact to the customer experience?

- Decouple with a durable queue/stream (SQS or Kinesis): accept the payment request synchronously, enqueue a command, and return immediately with a tracking ID.
- Add scalable workers (Lambda or ECS) to drain the queue and call the third-party with retries and exponential backoff; scale concurrency dynamically.
- Implement idempotency keys to avoid double-charging on retries; persist command status in a datastore and provide a status query API/webhook.
- Use a circuit breaker and rate limiter: when the provider degrades, shed load, fail fast, and gradually recover; respect provider quotas.
- Introduce DLQ and dead-letter workflows to capture permanently failing transactions for manual review or alternate routing.
- Provide user experience fallbacks: optimistic UI, status pages, and asynchronous notifications so the customer isn't blocked on provider latency.
- Add observability: structured logs, metrics (success rate, latency, queue depth, age), and alerts to tune concurrency and spot provider issues early.

7. What steps would you take in API Gateway to secure your API? Your goal is to both authenticate and authorize the users. (describe every step)

- Choose an auth mechanism: Cognito/JWT authorizer for end-user auth; IAM auth for service-to-service; optionally API keys with usage plans for metering.
- Configure a JWT authorizer: supply issuer, JWKS URI, and audiences; attach it to routes requiring user auth.
- Define fine-grained authorization: use route/claim-based authorization (scopes/claims) or a Lambda authorizer to compute an IAM policy per request.
- Validate inputs: enable request validation against models, limit payload sizes, and require HTTPS; use WAF for IP allow/deny, bot, and OWASP protections.
- Apply throttling and quotas: usage plans and API keys per client; set burst/rate limits to protect backends from abuse.

- Enforce least privilege downstream: integration roles with minimal permissions to call Lambda/other services; avoid passing caller credentials.
- Monitor and rotate: enable access logs and metrics, set alerts, rotate secrets/keys, and review authorizer configs and resource policies regularly.

8. What is the most likely issue with the Lambda function's timeout?

- The configured timeout is too low for the actual work (network calls, cold start, I/O) causing the runtime to terminate before completion.
- The function is waiting on external resources (VPC without NAT/endpoint, slow third-party API), so it doesn’t return in time.
- In Node.js, unresolved async keeps the event loop open (not `await`ing promises, leaving connections open, or `context.callbackWaitsForEmptyEventLoop` issues).
- Database/HTTP clients are created per-invocation causing high latency; move client initialization outside the handler and reuse connections.
- Missing retries/backoff causes long single attempts instead of bounded quick retries; tune timeouts at the SDK/HTTP layer.
- Function is over-constrained: too little memory reduces CPU/network throughput; increasing memory often reduces latency.
- Logging/exception handling masks failures; add granular logs and metrics to locate the long pole and right-size the timeout.

9. What is the most likely reason for the missing RestaurantRecommendationIntent? How would you solve it?

- In our flow S3 → API Gateway → Lambda (LF0) → Lex → Lambda (LF1) → SQS, the console uses `$LATEST` but LF0 usually calls a published alias; alias/version drift hides newly added intents.
- Rebuild the bot, publish a new version, then update the alias that LF0 references to point to that version (with the correct `localeId`).
- Verify LF0’s configuration/env for `botId`, `botAliasId`, `localeId`, region, and make sure LF0 hits the same alias used in end-to-end tests.
- Ensure LF0 sends plain text to `RecognizeText`/`PostText` with proper session state; malformed payloads cause fallback and look like “missing intent.”
- Confirm IAM on LF0 allows Lex runtime for the specific bot/alias; inspect CloudWatch logs and Lex analytics to validate traffic hits the right alias.
- Test through API Gateway (not just the Lex console) to ensure the path your users take resolves the intent consistently.

- Lock deployments so API/LF0 always reference the intended Lex alias version to prevent future drift.

10. When would you use Elastic vs DynamoDB? Give examples to illustrate when one is preferable over the other.

- Use Elasticsearch/OpenSearch for full-text search, relevance ranking, fuzzy matching, aggregations, and near real-time log/analytics queries.
- Use DynamoDB for high-scale key-value and document storage with single-digit-ms reads/writes, predictable performance, and serverless operations.
- If you need complex search (prefix, wildcard, scoring) across text, Elastic is better; if you need primary-key lookups and simple filters, DynamoDB excels.
- Elastic supports faceted aggregations and ad-hoc queries; DynamoDB requires pre-modeled access patterns and secondary indexes.
- DynamoDB offers auto scaling, On-Demand capacity, streams, TTL, and strong/consistent read options; Elastic requires cluster sizing and maintenance.
- Example Elastic: search restaurants by name/cuisine with typo tolerance and facets by price/rating; log analytics dashboards.
- Example DynamoDB: store user profiles, sessions, carts, and order records keyed by userId/orderId with GSI for common queries.

11. You create a new intent called HotelBookingIntent in Lex. You add a few utterances and build it. When you start testing it in the Lex test console, everything works as expected. You then write code to call your Lex bot from the Lambda function, but when you make calls to your API, the Lambda function times out.

> What is the most likely reason for the Lambda function timing out when connected with the Lex bot, while everything is working correctly through the Lex test console?

- In our architecture, API Gateway invokes LF0 which calls Lex. LF0 timeouts are commonly from VPC egress issues or wrong Lex endpoint parameters.
- If LF0 runs in private subnets, add a NAT Gateway (or remove VPC) so it can reach Lex runtime; fix route tables and security groups for outbound internet.
- Double-check `botId`, `botAliasId`, `localeId`, region and service endpoint in LF0; mismatches can hang on connect/DNS until timeout.
- Use the correct operation (`RecognizeText`/`PostText`) and await it; log timestamps before/after the SDK call to pinpoint where it stalls.

- Temporarily increase LF0 timeout and memory to aid diagnostics; then right-size after networking is corrected.
- IAM issues usually fail fast, but confirm LF0's role allows Lex runtime APIs for your bot/alias.
- Validate by calling Lex from a local script with the same params to distinguish code vs environment problems.

12. In assignment 1, you created an API Gateway with a POST method. You also created a Lambda function that can handle the POST request. What are some reasons for using API Gateway instead of directly invoking the Lambda function?

- Authentication/Authorization at the edge: integrate Cognito/JWT/IAM authorizers and resource policies before invoking backend.
- Request shaping and validation: models, mapping templates/transformations, throttling, and size limits to protect Lambda.
- Routing and versioning: multiple routes/stages, custom domains, stage variables, and canary deployments.
- Caching and performance: API Gateway caching reduces Lambda invocations for idempotent GETs.
- Observability: access logs, metrics, tracing, and WAF integration for security controls.
- Traffic management: quotas, usage plans, and API keys for clients; rate limiting to prevent abuse.
- Protocol front-door: REST/HTTP/WebSocket support with consistent endpoints and integration to many backends.

13. In assignment 1, what is a session in Lex? How can it be useful?

- In Dining Concierge, the Lex session holds cuisine, location, date/time, and email gathered across turns.
- LF0/LF1 read/write session attributes to carry context like a `requestId` that LF1 uses when pushing a complete payload to SQS.
- This avoids re-asking questions and ensures only validated, complete requests are enqueued to Q1 for LF2.
- Session context enables corrections ("actually Italian tomorrow") without restarting the conversation.
- We can persist a compact session key in DynamoDB for idempotency and correlating SQS messages to SES notifications.
- Session expiry prevents stale data from leaking into new requests.

- Result: smoother UX and a clean handoff into the async pipeline (SQS → LF2 → OpenSearch/DynamoDB → SES).

14. In assignment 1, you notice that some of the documents are not being indexed in your elasticsearch index, and there are no error messages. How would you troubleshoot this scenario and ensure that all documents are indexed correctly? List at least 2 points of failures.

- In our system, LF2 queries OpenSearch for candidate IDs; missing docs there yields empty recommendations even if DynamoDB has rows.
- First validate the Yelp→OpenSearch ingestion logs and bulk responses; ensure LF2 is querying the intended index/alias.
- Check `_cluster/health` and `_cat/indices` for shard/unassigned/read-only states; address disk watermarks or ILM blocks.
- Confirm mappings and fields used by LF2 (`cuisine`, `city`, `geolocation`) are present and analyzers align with query clauses.
- If LF2 uses `.keyword` fields but mapping lacks them, adjust mapping or query the analyzed field.
- Add DLQ/retry for ingestion and structured logging in LF2 around OpenSearch queries to surface partial failures.

15. In assignment 1, When developing a RESTful API with AWS API Gateway, you encounter an issue where certain HTTP requests, particularly those with non-standard headers, are failing unexpectedly. You suspect that preflight requests might be involved. Explain the concept of preflight requests, why they are important for web security, and how you would troubleshoot and resolve this issue in your API Gateway setup.

- Our S3 frontend calls API Gateway with Authorization headers; browsers send an `OPTIONS` preflight first.
- Ensure every route defines an `OPTIONS` method and returns `Access-Control-Allow-Origin`, `Access-Control-Allow-Methods`, and `Access-Control-Allow-Headers` listing `Authorization, Content-Type, X-Amz-Date`.
- If using Lambda proxy, have LF0/LF1 include the same CORS headers on both success and error paths.
- Re-deploy the API stage and test from the S3 site origin; add `Vary: Origin` so caches don’t return mismatched headers.
- For credentials/cookies, never use `*`; list the exact S3 website origin.

- Use the Network tab to inspect both the preflight and the actual request; align headers accordingly.

16. In assignment 1, you created a DynamoDB table for storing restaurant data. You used restaurant/business ID as the primary key. What could possibly go wrong if you use insertedTimestamp as the primary key? How about using insertedTimestamp as the secondary key?

- In this architecture, LF2 fetches details by `restaurantId` after OpenSearch returns IDs; if the table PK were `insertedTimestamp`, LF2 would need a GSI or a scan, adding latency and cost.
- Timestamp as PK also creates hot partitions during bulk Yelp imports and complicates idempotency when multiple records share close timestamps.
- Keep table PK=`restaurantId`. If versioning/history is required, add SK=`insertedTimestamp` and query latest with `ScanIndexForward=false`.
- Alternatively, use a GSI with PK=`restaurantId`, SK=`insertedTimestamp` for historical reads while preserving fast primary-key lookups on the table.
- Align TTL/archival with timestamps but avoid using them as the primary access path unless truly necessary.

17. What is the purpose of a Dockerfile? a. To define the environment variables for a Docker container b. To configure the network settings for a Docker container c. To specify the runtime dependencies for a Docker container d. To define the Docker image and its dependencies

- Intuition: A Dockerfile is the blueprint from which images are built, describing base image, files, commands, and dependencies.
- Answer: d. To define the Docker image and its dependencies
- Why d is correct: A Dockerfile declares how to build the image (FROM, RUN, COPY, CMD), including dependencies and configuration.
- Why a is wrong: You can set env vars in a Dockerfile, but that’s only a part, not its purpose.
- Why b is wrong: Networking is configured at container runtime (docker run, compose, Kubernetes), not primarily in Dockerfile.
- Why c is incomplete: Dependencies are included but so are filesystem layers, entrypoints, and more; Dockerfile’s scope is the full image.
- Practical note: Repeatable builds, portability, and minimal, layered images are core benefits of Dockerfiles.

18. Which Docker component is responsible for managing images? a. Docker Engine b. Docker Registry c. Docker Compose d. Docker Hub

- Intuition: Images are stored and distributed by registries; the engine runs containers; compose orchestrates multi-container apps.
- Answer: b. Docker Registry
- Why b is correct: A registry stores and serves images via push/pull APIs (e.g., Docker Hub, ECR act as registries).
- Why a is wrong: Docker Engine builds/runs images locally but doesn’t manage remote distribution/storage.
- Why c is wrong: Compose defines multi-container apps; it doesn’t manage images.
- Why d is nuanced: Docker Hub is a hosted registry service; the generic component type is “registry,” hence b is the correct abstraction.

19. What is the main benefit of AWS Lambda? a. It eliminates the need for server administration b. It offers unlimited compute power and storage c. It supports only a limited number of programming languages d. It requires manual scaling to handle increased traffic

- Intuition: Serverless abstracts servers and scaling; you focus on code and events.
- Answer: a. It eliminates the need for server administration
- Why a is correct: Lambda is fully managed—no server provisioning/patching; scaling is automatic to demand.
- Why b is wrong: Quotas and limits exist; not unlimited.
- Why c is wrong: It supports many languages; the statement is negatively framed and false.
- Why d is wrong: Scaling is automatic; you set concurrency limits rather than manual scaling.

20. How does AWS Lambda handle state management? a. It provides a built-in state management system b. It does not handle state management, and requires the use of external storage systems c. It uses an in-memory cache for state management d. It relies on the programmer to handle state management within the function code

- Intuition: Lambda functions are ephemeral and scale horizontally; persistent state must live elsewhere.
- Answer: b. It does not handle state management, and requires the use of external storage systems

- Why b is correct: Persistent state goes to DynamoDB, S3, RDS, etc. Lambdas may reuse transient memory but it’s not guaranteed.
- Why a is wrong: There is no built-in persistent state store in Lambda.
- Why c is wrong: In-memory caches are per-container and ephemeral; not reliable across invocations or scales.
- Why d is incomplete: Developers orchestrate state usage, but persistence still requires external systems; b best captures the truth.

21. A__________ breakdown fields values of a document into a stream, and inverted indexes are created and updated using these values, and these stream of values are stored in the document. a. Analyzer b. Shard c. Filter d. Tokenizer

- Intuition: In search engines, tokens are produced from text to build inverted indexes; the component generating tokens is the tokenizer.
- Answer: d. Tokenizer
- Why d is correct: A tokenizer splits text into tokens (terms), which feed inverted index creation.
- Why a is nuanced: An analyzer is a pipeline that includes a tokenizer and filters; not just the breakdown step.
- Why b is wrong: A shard is a partition of data, unrelated to token creation.
- Why c is wrong: Filters modify tokens (e.g., lowercase, stopwords) after tokenization; they don’t perform the initial breakdown.

22. Which of the following is a collection of fields in a specific manner defined in JSON format? a. Node b. Shard c. Index d. Document

- Intuition: In Elasticsearch, a document is a JSON object with fields; an index stores many documents.
- Answer: d. Document
- Why d is correct: A document is the JSON record consisting of fields and values.
- Why a is wrong: A node is a server in the cluster.
- Why b is wrong: A shard is a physical partition of an index.
- Why c is wrong: An index is a logical namespace that holds many documents.

23. Which of the following runs on each node and ensures containers are running in a pod? a. Pod b. Etcd c. Kubelet d. Scheduler

- Intuition: The agent on each worker that talks to the control plane and manages pods is the kubelet.
- Answer: c. Kubelet

- Why c is correct: Kubelet watches the API server and ensures containers in pods are running on its node.
- Why a is wrong: A pod is a workload object, not an agent.
- Why b is wrong: etcd is the key-value store for cluster state, not per-node agent.
- Why d is wrong: The scheduler assigns pods to nodes; it doesn't run on every node to manage containers.

24. Which field in a Service YAML file specifies which pods the service should route traffic to? a. selector b. ports c. type d. targetPort

- Intuition: Services select pods via labels; the field that defines label matching is `selector`.
- Answer: a. selector
- Why a is correct: The `selector` matches pod labels to form the service's endpoints.
- Why b is wrong: `ports` define exposed ports, not target pods.
- Why c is wrong: `type` sets exposure (ClusterIP/NodePort/LoadBalancer), not selection.
- Why d is wrong: `targetPort` maps service port to container port, not selection.

25. What protocol governs the permissions granted by a web server to web browsers when accessing resources from different domains? Explain how this works.

- CORS (Cross-Origin Resource Sharing) controls whether a browser may access a cross-origin resource based on server-provided headers.
- The browser enforces CORS: on simple requests it checks `Access-Control-Allow-Origin` on the response; on non-simple it first sends an `OPTIONS` preflight.
- The server responds with allowed origins, methods, and headers (`Access-Control-Allow-*`), optionally credentials via `Access-Control-Allow-Credentials`.
- If the response doesn't authorize the origin/method/headers, the browser blocks access even if the server returned data.
- Credentials require explicit origins (no `*`) and matching `withCredentials`/`credentials` on the client.
- CORS is a browser security policy; servers and non-browser clients (curl) can access cross-origin endpoints without CORS.
- Proper configuration lives at the origin server (API), with CDNs/proxies forwarding or setting consistent headers.

26. Write a Kubernetes Service manifest to expose a Deployment named "my-app" on port 8080 within the cluster.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-app
spec:
  type: ClusterIP
  selector:
    app: my-app
  ports:
    - port: 8080
      targetPort: 8080
```

27. Explain the concept of serverless architecture and its benefits. How does it relate to the implementation of the Dining Concierge chatbot?

- Serverless offloads server provisioning and management to the provider; you deploy functions and managed services that scale automatically.
- You pay per-use, not for idle capacity, reducing cost for bursty workloads.
- Managed scaling and high availability remove much of the ops burden and enable faster iteration.
- Event-driven composition (API Gateway, Lambda, DynamoDB, SQS, Lex) fits conversational flows and background tasks.
- For Dining Concierge, API Gateway handles HTTP, Lambda runs intent logic, Lex manages NLP, DynamoDB stores data, and SQS/SES handle async tasks.
- This architecture decouples components, enabling independent scaling for chat intents, recommendations, and notifications.
- Observability with CloudWatch and X-Ray helps trace user interactions end-to-end without server management.

28. Design a data model for the DynamoDB table "yelp-restaurants" to efficiently store and query restaurant information. Discuss the choice of partition keys, sort keys, and secondary indexes to support various query patterns, including location-based searches and cuisine filtering.

- Align to our flow where LF2 queries OpenSearch for candidate `restaurantId`s, then DynamoDB for details.
- Base table: PK=`restaurantId` (single current record), attrs: `name`, `address`, `city`, `cuisine[]`, `rating`, `price`, `geohash`, `insertedTimestamp`.

- If we also browse directly from DynamoDB: GSI1 PK=`cuisine`, SK=`city#ratingDesc#restaurantId` to rank by rating within a city.
- For proximity: materialize `geohashPrefix` (5–7 chars) and add GSI2 PK=`geohashPrefix`, SK=`ratingDesc#restaurantId`; query adjacent prefixes for radius.
- For history/versioning: either table SK=`insertedTimestamp` (PK=`restaurantId`) or a GSI3 PK=`restaurantId`, SK=`insertedTimestamp` with `Limit=1, ScanIndexForward=false`.
- Keep index projections lean (`restaurantId`, `name`, `city`, `cuisine`, `rating`) to control cost; fetch full item by ID afterward.
- Maintain these fields at ingestion time and use Streams to audit/repair denormalized attributes if they drift.

29. In a microservices architecture, when choosing between AWS EC2 and AWS Lambda for handling tasks within a service, what distinguishes their use cases?

- Lambda: event-driven, bursty traffic, short-lived stateless tasks, managed scaling, pay-per-invoke, tight AWS integration.
- EC2: long-running processes, custom runtimes, persistent connections, special networking, or when you need OS-level control.
- Lambda suits background jobs, API handlers, and stream consumers; EC2 suits stateful services, heavy compute with custom deps, and low-latency sockets.
- Cold starts and time limits constrain Lambda; EC2 can maintain warm processes and control thread pools.
- Compliance or specialized hardware (GPUs, EFA) often requires EC2.
- Cost model: Lambda is economical at low-to-medium steady load; EC2 is better at high steady utilization.
- Operational model: Lambda reduces ops toil; EC2 gives full flexibility at the cost of management.

30. What would happen if your Docker container for MongoDB does not specify a persistent volume?

- Data is stored in the container's writable layer; when the container stops or is recreated, the data is lost.
- Upgrades and redeploys will start with an empty database unless a volume/bind mount persists `/data/db`.
- Backups and snapshots are difficult without a mounted volume; disaster recovery risk increases.

- Performance may degrade due to storage driver overhead compared to proper volumes.
- Multiple replicas/containers cannot easily share consistent data without volumes.
- Operational tasks (compaction, fsync) depend on underlying storage guarantees which ephemeral layers don't provide.
- Best practice: define a named volume or PVC (in Kubernetes) mapped to `/data/db` to persist state.

31. In Amazon Lex, which component is responsible for handling user input before Lex responds? a. Bot b. Intent c. Lambda code hook
    d. Slot type

- Intuition: Lex can invoke custom logic to validate/fulfill before replying; that hook processes the input.
- Answer: c. Lambda code hook
- Why c is correct: The code hook runs developer code to validate slots, fetch data, or fulfill intents prior to Lex's final response.
- Why a is wrong: A bot is a container for intents; it doesn't process inputs programmatically.
- Why b is wrong: An intent defines goal/slots/utterances, not arbitrary execution logic.
- Why d is wrong: Slot types define allowed values; they don't execute code on input.

32. Your serverless chatbot API experiences high response times after a period of inactivity. What is the most likely cause? a. The API Gateway caching mechanism is misconfigured. b. The Lambda function is experiencing a cold start delay.
    c. The Lex bot is consuming too many Lex slots per request. d. The Lambda function is running in a VPC without internet access.

- Intuition: Latency spikes after idle suggest container/runtime startup—classic cold start behavior.
- Answer: b. The Lambda function is experiencing a cold start delay.
- Why b is correct: After inactivity, Lambda initializes a new execution environment, adding startup latency.
- Why a is wrong: Cache misconfig would affect steady-state too, not specifically post-idle.
- Why c is wrong: Slot usage doesn't cause idle-correlated spikes.
- Why d is wrong: No-internet causes errors/timeouts generally, not just after idle.

33. Your chatbot API deployed on API Gateway + AWS Lambda is experiencing a sudden surge in traffic, leading to immediate failed requests. What is the most likely cause? a. The Lambda function's memory allocation is too low b. Lambda has reached its concurrent execution limit
c. The DynamoDB table is not optimized for query performance d. The API Gateway is not configured with a usage plan

- Intuition: Surges that fail immediately typically indicate throttling at concurrency limits.
- Answer: b. Lambda has reached its concurrent execution limit
- Why b is correct: Hitting regional/reserved concurrency caps results in throttled invocations and failures.
- Why a is wrong: Low memory slows execution but doesn't directly cause immediate surge failures.
- Why c is wrong: DynamoDB slowness impacts latency, not invocation admission.
- Why d is wrong: Usage plans throttle per API key, not core Lambda concurrency.

34. While testing your chatbot, you notice that Lex always returns the fallback intent, even when the user input clearly matches an intent. What is the most probable reason? a. The Lex bot is not properly trained with enough utterances b. The Lex bot has not been published and deployed to a specific version c. The Lambda code hook is failing and Lex is defaulting to fallback d. Any of the above can be a cause

- Intuition: Fallback can be triggered by low recognition confidence, unpublished changes, or hook failures.
- Answer: d. Any of the above can be a cause
- Why d is correct: Insufficient utterances lower confidence; unpublished versions exclude changes; failing hooks can push fallback.
- Why a is plausible: Poor training/data yields fallback.
- Why b is plausible: Console uses $LATEST; runtime alias may lag.
- Why c is plausible: Exceptions or timeouts in hooks cause fallback behavior.

35. Your replication controller in Kubernetes is set to maintain 5 replicas of a pod, but you notice that only 3 are running. What could be the reason? a. The remaining nodes in the cluster do not have enough resources
b. Kubernetes does not support replication for multi-container pods c. The pod deployment is using an unsupported Docker image d. The replication controller is waiting for approval to create new pods

- Intuition: If the scheduler can’t place pods, it’s usually due to CPU/memory constraints on nodes.
- Answer: a. The remaining nodes in the cluster do not have enough resources
- Why a is correct: Resource shortages leave pods Pending; only some replicas can be scheduled.
- Why b is wrong: Multi-container pods are fully supported and replicated as units.
- Why c is less likely: Image issues cause ImagePullBackOff/CrashLoop, not exactly 3/5 replicas placed.
- Why d is wrong: There’s no approval gate in replication.

36. Your Flask application on AWS EKS uses MongoDB for data storage. After a pod restart, all previously stored data is lost. What is the most likely issue? a. The MongoDB deployment is using an emptyDir volume instead of a Persistent Volume Claim (PVC). b. The Persistent Volume Claim (PVC) is not mounted under /data/db, MongoDB's default data directory. c. The Kubernetes cluster does not support stateful applications. d. The MongoDB container does not expose the correct port.

- Intuition: Data loss after restart indicates ephemeral storage; emptyDir is erased with pod lifecycle.
- Answer: a. The MongoDB deployment is using an emptyDir volume instead of a Persistent Volume Claim (PVC).
- Why a is correct: emptyDir resets on pod restart; PVC persists across reschedules.
- Why b is plausible but derivative: Wrong mount path leads Mongo to write to container FS (also ephemeral), but root cause is lack of proper persistence.
- Why c is wrong: Kubernetes supports StatefulSets and PVCs for stateful apps.
- Why d is unrelated: Ports affect connectivity, not persistence.

37. If your Minikube cluster fails to expose the To-Do app, which of the following is most likely the issue? a. The service is created as ClusterIP instead of NodePort. b. The Kubernetes API server is down. c. The pod is in Running state but has no logs. d. The To-Do app does not have enough replicas.

- Intuition: Minikube exposes services via NodePort/LoadBalancer; ClusterIP isn’t externally reachable by default.
- Answer: a. The service is created as ClusterIP instead of NodePort.
- Why a is correct: ClusterIP limits access to within the cluster; NodePort is needed for external access in Minikube.

- Why b is unlikely: If API server were down, more symptoms would be present.
- Why c is irrelevant: Logs don't control exposure.
- Why d is unrelated: Replica count doesn't change service type reachability.

38. If your liveness probe is failing continuously, what action will Kubernetes take? a. Kubernetes will scale down the pod b. Kubernetes will restart the failing pod
c. Kubernetes will log the failure but take no action d. Kubernetes will redirect traffic to a different service

- Intuition: Liveness is for self-healing; failure implies the container must be restarted.
- Answer: b. Kubernetes will restart the failing pod
- Why b is correct: Liveness failure triggers container restart to recover from deadlock/crash.
- Why a is wrong: Scaling decisions are separate from probes.
- Why c is wrong: Liveness is actionable, not passive.
- Why d is wrong: Traffic routing is governed by readiness, not liveness.

39. If your Flask application is failing after being deployed on AWS EKS with an error indicating 'ImagePullBackOff,' what should you check first? a. Ensure Docker is installed on your local machine b. Verify that the correct image is pushed to Docker Hub
c. Increase the number of replicas in the deployment d. Restart the Minikube cluster

- Intuition: ImagePullBackOff means nodes can't pull the image—check registry, tag, and credentials.
- Answer: b. Verify that the correct image is pushed to Docker Hub
- Why b is correct: Wrong/missing image or tag in the registry causes pull failures; also verify pull secrets if private.
- Why a is irrelevant: EKS nodes pull images; your local Docker install isn't used.
- Why c is wrong: Replicas don't fix pull failures.
- Why d is unrelated: Minikube is a local tool; EKS issues won't be fixed by it.

40. What is the difference between a Pod and a Deployment in Kubernetes? A) A Pod is a deployment strategy, while Deployment is a single container. B) A Deployment is a group of Pods, while a Pod is the smallest deployable unit. C) A Pod is used for scaling, while Deployment is a runtime environment. D) A Pod represents a service, while Deployment represents a container image.

- Intuition: Pods are the basic workload units; Deployments manage ReplicaSets and rollout/rollback for pods.
- Answer: B) A Deployment is a group of Pods, while a Pod is the smallest deployable unit.
- Why B is correct: Deployments declaratively manage pods via ReplicaSets; a pod encapsulates one or more containers sharing network/storage.
- Why A is wrong: A pod is not a strategy; a Deployment isn’t a single container.
- Why C is wrong: Deployments manage scaling; pods are the units that get scaled.
- Why D is wrong: Services are separate objects; Deployments don’t represent images.

41. What is the role of the Kubernetes Master in a cluster? A) Manages the deployment of applications B) Runs user applications in containers C) Controls and coordinates the overall cluster D) Provides storage volumes for containers

- Intuition: The master (control plane) manages cluster state, scheduling, and reconciliation of desired vs current state.
- Answer: C) Controls and coordinates the overall cluster
- Why C is correct: Control plane components (API server, scheduler, controller manager, etcd) govern scheduling and state.
- Why A is incomplete: It’s broader than deployments; it manages all controllers and orchestration.
- Why B is wrong: Workers run application pods; masters typically do not run workloads.
- Why D is wrong: Storage is provided via CSI and worker nodes; not a master responsibility.

42. How do you scale the worker node capacity in an Amazon EKS cluster? A) Update the EKS Control Plane configuration B) Add or remove EC2 instances from the Auto Scaling Group C) Use the eksctl command to resize the cluster D) There is no way to scale worker nodes in EKS.

- Intuition: Worker capacity is provided by EC2 node groups backed by Auto Scaling Groups (ASGs).
- Answer: B) Add or remove EC2 instances from the Auto Scaling Group
- Why B is correct: Scaling the ASG or Managed Node Group changes worker count.
- Why A is wrong: Control plane changes don’t add workers.
- Why C is tooling: eksctl drives ASG scaling; mechanism remains ASG sizing.

- Why D is false: EKS supports scaling via ASGs/Managed Node Groups.

43. What are DataFrames in Apache Spark, and how do they differ from RDDs? A) DataFrames are a collection of key-value pairs, while RDDs are tables. B) DataFrames are a higher-level abstraction built on top of RDDs, providing a structured representation of data. C) DataFrames and RDDs are interchangeable terms in Spark. D) DataFrames are only suitable for batch processing, while RDDs are for real-time processing.

- Intuition: DataFrames add schema and optimizer (Catalyst/Tungsten) on top of RDDs enabling SQL-like operations.
- Answer: B) DataFrames are a higher-level abstraction built on top of RDDs, providing a structured representation of data.
- Why B is correct: DataFrames have columns and types, allowing optimized query plans; RDDs are low-level distributed collections.
- Why A is wrong: It incorrectly characterizes both abstractions.
- Why C is wrong: They're related but not interchangeable.
- Why D is wrong: Both support streaming via Structured Streaming.

44. In AWS CloudFormation, what is the purpose of the "DependsOn" attribute in a resource definition? a. Specifies the resource's dependencies on other AWS services. b. Declares the order of execution for the CloudFormation stack. c. Identifies the resources that are dependent on the current resource. d. Enables parallel execution of resources for faster stack creation.

- Intuition: Use `DependsOn` to enforce creation/update order between resources in a template.
- Answer: b. Declares the order of execution for the CloudFormation stack.
- Why b is correct: It ensures one resource is created only after another completes.
- Why a is wrong: It's not a broad AWS dependency description; it's about intra-stack resource ordering.
- Why c is reversed: It specifies what this resource depends on, not the reverse.
- Why d is wrong: It may serialize operations rather than parallelize them.

45. What is the purpose of a stack policy in AWS CloudFormation? a. Enforces IAM policies on CloudFormation stacks. b. Defines the parameters for the CloudFormation stack. c. Specifies the permissions required for resources within a stack. d. Controls updates to stack resources by allowing or denying specific actions.

- Intuition: Stack policies protect critical resources from accidental modification during stack updates.
- Answer: d. Controls updates to stack resources by allowing or denying specific actions.
- Why d is correct: It restricts which resources can be updated and which actions are permitted during updates.
- Why a is wrong: IAM controls who can act; stack policies control what updates can affect.
- Why b is wrong: Parameters are separate; they don’t enforce update protection.
- Why c is wrong: Permissions are governed by IAM, not stack policy.

46. What is the primary distinction between a Deployment and a Service in Kubernetes? a. Deployments manage container scaling, while Services manage pod communication. b. Deployments handle container orchestration, while Services manage networking and load balancing. c. Deployments ensure high availability, while Services provide persistent storage for pods. d. Deployments control resource allocation, while Services handle container image deployment.

- Intuition: Deployments manage pod lifecycle and rollouts; Services provide stable networking and load balancing to pods.
- Answer: b. Deployments handle container orchestration, while Services manage networking and load balancing.
- Why b is correct: Deployments manage ReplicaSets/updates; Services route traffic via stable ClusterIP/DNS to matching pods.
- Why a is narrow: Services also do discovery/load-balancing; b captures both sides clearly.
- Why c is wrong: Services don’t provide storage; PersistentVolumes/Claims do.
- Why d is wrong: Deployments reference images, but Services don’t deal with images at all.

47. What is the primary purpose of Docker containers? a. Virtualizing hardware resources b. Running multiple operating systems on a host c. Isolating and packaging applications with their dependencies d. Managing networking configurations

- Intuition: Containers package applications and dependencies with OS-level isolation, not hardware virtualization.
- Answer: c. Isolating and packaging applications with their dependencies

- Why c is correct: Containers bundle code, runtime, and libraries into portable units that run consistently.
- Why a is wrong: That describes VMs, not containers.
- Why b is wrong: Containers share the host kernel; they don’t run multiple OS kernels.
- Why d is wrong: Networking is a feature, not the purpose.

48. How does Kubernetes maintain the desired state of a deployment? A. By using the Kubelet to periodically check and adjust the state of each pod. B. Through the control loop in the Kubernetes controller manager. C. By manually adjusting the state based on user input. D. Using an external state management system.

- Intuition: Kubernetes controllers continuously reconcile the desired state with the observed state.
- Answer: B. Through the control loop in the Kubernetes controller manager.
- Why B is correct: Deployment/ReplicaSet controllers watch resources and adjust pods to match spec.
- Why A is wrong: Kubelet enforces state on a node; it doesn’t manage deployment-level desired state.
- Why C is wrong: Reconciliation is automatic rather than manual.
- Why D is wrong: No external state system is required for reconciliation.

49. What is the role of etcd in a Kubernetes cluster? A. It acts as the primary database for Kubernetes, storing all cluster data. B. It is used for load balancing traffic between pods. C. It monitors the health of nodes in the cluster. D. It schedules pods to run on various nodes.

- Intuition: etcd is the strongly consistent key-value store backing the API server’s persisted cluster state.
- Answer: A. It acts as the primary database for Kubernetes, storing all cluster data.
- Why A is correct: All cluster objects are persisted in etcd via the API server.
- Why B is wrong: Services/proxies handle load balancing, not etcd.
- Why C is wrong: Controllers and node agents handle health; etcd stores state.
- Why D is wrong: The scheduler assigns pods; etcd only stores desired/observed state.

50. In Kubernetes, what is the difference between a ReplicaSet and a Deployment? A. A ReplicaSet provides rollback and update functionality, while a Deployment does not. B. A Deployment is for stateful applications, while a ReplicaSet is for stateless

applications. C. A Deployment manages ReplicaSets and provides declarative updates to pods. D. There is no difference; they are functionally identical.

- Intuition: Deployments orchestrate rollouts/rollbacks and manage ReplicaSets; ReplicaSets maintain a fixed number of pod replicas.
- Answer: C. A Deployment manages ReplicaSets and provides declarative updates to pods.
- Why C is correct: Deployment handles rolling updates and rollbacks; ReplicaSet ensures desired replica count.
- Why A is wrong: It's inverted; Deployments provide update/rollback features.
- Why B is wrong: Stateful apps use StatefulSets; both RS and Deployment typically handle stateless pods.
- Why D is wrong: They have distinct responsibilities.

51. What mechanism does Kubernetes use for service discovery? A. It uses an internal DNS server for service discovery. B. Services are discovered via environmental variables. C. It relies on an external service discovery mechanism. D. Both A and B.

- Intuition: Kubernetes injects environment variables and also runs cluster DNS (CoreDNS) for name resolution.
- Answer: D. Both A and B.
- Why D is correct: Env vars are populated for services; DNS resolves service names across the cluster.
- Why A alone is incomplete: Env var discovery exists too.
- Why B alone is incomplete: DNS is the primary scalable mechanism.
- Why C is wrong: External discovery isn't required in-cluster.

52. How does Kubernetes achieve fault tolerance at the application layer? A. By restarting pods that exit or get evicted. B. Through manual intervention by the cluster administrator. C. By using multiple master nodes in the cluster. D. By replicating the application across multiple cloud providers.

- Intuition: Controllers recreate or reschedule pods to maintain desired state after failures.
- Answer: A. By restarting pods that exit or get evicted.
- Why A is correct: Liveness/readiness with controllers cause unhealthy pods to be restarted or rescheduled automatically.
- Why B is wrong: Recovery is automated.
- Why C is control-plane HA, not application-layer tolerance.

- Why D is not a built-in Kubernetes feature.

53. What is the purpose of Kubernetes namespaces? A. To provide isolation at the node level. B. To enable different teams or projects to share a cluster without interference. C. To manage the different versions of an application. D. To isolate network traffic between pods.

- Intuition: Namespaces logically partition resources and names for multi-tenancy and policy scoping.
- Answer: B. To enable different teams or projects to share a cluster without interference.
- Why B is correct: Namespaces scope names, quotas, and RBAC to separate teams/applications.
- Why A is wrong: Isolation is logical, not node-level.
- Why C is wrong: Versions are managed by controllers and labels.
- Why D is wrong: NetworkPolicies handle network isolation, not namespaces alone.

54. In a Kubernetes cluster, how is the master node different from worker nodes? A. The master node runs application workloads, while worker nodes manage the cluster state. B. The master node contains the etcd database, while worker nodes run the Kubernetes API server. C. The master node schedules workloads and manages the cluster, while worker nodes run the scheduled workloads. D. There is no difference; all nodes perform the same functions.

- Intuition: Control plane vs data plane—masters manage, workers execute workloads.
- Answer: C. The master node schedules workloads and manages the cluster, while worker nodes run the scheduled workloads.
- Why C is correct: Masters host control plane components; workers run pods.
- Why A is reversed.
- Why B is inaccurate: API server and etcd are on masters; workers don’t run the API server.
- Why D is false: Roles are distinct by design.