



You have to answer the following questions:

1. What is a webapp? Explain its main components
2. What is a stateful web app
3. What is cloud? List 10 benefits and disadvantages of cloud .
4. Explain the service models: “Cloud as” □ SaaS, PaaS, IaaS in detail in tabular format
5. Explain the cloud delivery models: “Public , Proivate, Hybrid”
6. Swagger template for writing apis
7. Explain what is API? Explain all types: GET, POST, PUT, DELETE.
8. Explain the following swagger file

<>

swagger: '2.0'

info:

title: AI Customer Service API

description: 'AI Customer Service application, built during the Cloud and Big Data course at NYU.'

version: 1.0.0

schemes:

- https

basePath: /v1

produces:

- application/json

paths:

/chatbot:

post:

summary: The endpoint for the Natural Language Understanding API.

description: |

This API takes in one or more messages from the client and returns

one or more messages as a response. The API leverages the NLP

backend functionality, paired with state and profile information

```
arxt-aware reply.
tag Skip to content
  - NLU
  operationId: sendMessage
  produces:
    - application/json
  parameters:
    - name: body
      in: body
      required: true
      schema:
        $ref: '#/definitions/BotRequest'
  responses:
    '200':
      description: A Chatbot response
      schema:
        $ref: '#/definitions/BotResponse'
    '403':
      description: Unauthorized
      schema:
        $ref: '#/definitions/Error'
    '500':
      description: Unexpected error
      schema:
        $ref: '#/definitions/Error'
  definitions:
    BotRequest:
      type: object
      properties:
        messages:
          type: array
          items:
            $ref: '#/definitions/Message'
    BotResponse:
      type: object
      properties:
        messages:
          type: array
          items:
            $ref: '#/definitions/Message'
    Message:
      type: object
      properties:
```

```

type Skip to content
ty ↴
unstructured:
$ref: '#/definitions/UnstructuredMessage'

UnstructuredMessage:
type: object
properties:
id:
type: string
text:
type: string
timestamp:
type: string
format: datetime

Error:
type: object
properties:
code:
type: integer
format: int32
message:
type: string

```

</>

Thought for 51s >

1) What is a web app? Main components

1. Definition & request-response flow

1. A web application is software delivered over the internet through a browser or mobile client; the client sends HTTP(S) requests and the backend returns HTML/JSON (or similar) responses. Lecture_1&2_Cloud Basics
2. Typical client↔server layout: browser/mobile (frontend) → Web/API layer → application logic → data stores. Lecture_1&2_Cloud Basics

2. Frontend (client)

1. UI rendered in the browser or app (React/Angular/etc.).
2. Talks to backend via REST/JSON APIs. Lecture_1&2_Cloud Basics

3. Backend (services)

1. Web server & app code that implement business logic and expose Web APIs.

2. Often organized as microservices. Skip to content resilience and independent scaling.

Lecture_3_Microservices

4. Data layer & state

1. Databases/queues/caches (SQL/NoSQL, S3, SQS, Redis) for persistence and async work.
2. Design must account for “state vs stateless” behavior, especially when scaling elastically. Lecture_4.1_Elasticity

5. Infrastructure

1. Runs on compute (VMs/containers) with networking, load balancers, and IAM/security.
2. Cloud makes these programmable and elastic (on-demand, pay-as-you-go).

Lecture_1&2_Cloud Basics

2) What is a stateful web app?

1. Concept

1. “Stateful” means the server keeps per-user/session data across requests (e.g., carts, chat context).
2. Contrast: stateless APIs don’t retain request context server-side between calls.

Lecture_4.1_Elasticity

2. Where state lives

1. Memory/session store (Redis), DB rows, or files/objects (S3).
2. In microservices/serverless, push state to external stores so you can scale instances freely. Lecture_3_Microservices

3. Scaling implications

1. Sticky sessions tie users to instances; externalized state enables horizontal scaling & autoscale. Lecture_4.1_Elasticity
-

3) What is cloud? 10 benefits & 10 disadvantages

1. Definition (quick)

1. Cloud lets you programmatically request compute/storage/networking as services —on-demand, elastic, pay-as-you-go. Lecture_1&2_Cloud Basics

2. Top 10 benefits

1. On-demand provisioning. Lecture_1&2_Cloud Basics
2. Elastic scale up/down. Lecture_4.1_Elasticity

3. Pay-as-you-go economics ↗ s. over-provisioning.

[Skip to content](#)

[Lecture_1&2_Cloud_Basics](#)

4. Massive parallel capacity (thousands of machines briefly).

[Lecture_1&2_Cloud_Basics](#)

5. No hardware to install/repair.

[Lecture_1&2_Cloud_Basics](#)

6. Managed services accelerate development (e.g., S3/ELB/Lambda).

[Lecture_3_Microservices](#)

7. Global reach and availability.

8. Programmatic APIs/CLI for automation.

[Lecture_1&2_Cloud_Basics](#)

9. Faster experimentation/innovation.

[Lecture_3_Microservices](#)

10. Built-in observability & security primitives (IAM, logs).

3. Top 10 disadvantages

1. Ongoing OPEX; always-on workloads can be costlier than owned hardware.

[Lecture_1&2_Cloud_Basics](#)

2. Vendor lock-in and migration friction.

3. Latency to cloud regions and egress fees.

4. Limited low-level control vs. bare metal.

5. Compliance/data residency constraints → private/hybrid needs.

[Sample_Quiz_Spring2023](#)

6. Shared-tenancy & blast-radius of provider outages.

7. Operational limits/quotas (e.g., concurrent Lambdas).

[Sample_Quiz_Spring2025](#)

8. Cold starts on serverless paths.

[Sample_Quiz_Spring2025](#)

9. Complexity of IAM and multi-service architectures.

[Lecture_3_Microservices](#)

10. Observability cost (logs/metrics/traces).

4) Service models — SaaS vs PaaS vs IaaS (tabular)

Aspect	SaaS	PaaS	IaaS
What you get	Ready-to-use app	Managed runtime & platform	Virtualized compute/network/storage
You manage	Only your data & seat config	Your code & app configs	OS, middleware, your apps & data
Provider manages	App, runtime, OS, infra	Runtime, OS, infra	Physical infra, hypervisor, APIs
Dev velocity	Highest (no code)	High (just code)	Medium (full control)

Aspect	SaaS	Skip to content	IaaS
Customization	Low	Medium–High	Highest
When to use	Standardized needs (email, CRM)	Build/ship code quickly	Maximum control/flexibility
Examples	Gmail, Salesforce	App Engine, Elastic Beanstalk	EC2, GCE, Azure VMs
From lecture	“Use provider’s app; minimal control”	“Deploy your apps on provider tools; control app config”	“Provision CPUs, storage, networks; control OS/apps”
			Lecture_1&2_Cloud Basics

5) Cloud delivery models — Public, Private, Hybrid

1. Public cloud

1. Third-party provider runs multi-tenant infra you rent by the drink (AWS/GCP/Azure). Best for elasticity and speed.

Lecture_1&2_Cloud Basics

2. Private cloud

1. Cloud-like platform dedicated to one organization (on-prem or hosted) for tighter control/compliance and performance tuning.

Sample_Quiz_Spring2023

3. Hybrid cloud

1. Mix of private + public: keep sensitive or steady workloads private; burst/experiment in public to balance cost, scale, and compliance.

Sample_Quiz_Spring2023

6) Swagger (OpenAPI 2.0) template for writing APIs

yaml

 Copy code

```

swagger: "2.0"
info:
  title: Your API Name
  description: One-line purpose + any auth notes
  version: 1.0.0
schemes: [https]
host: api.example.com
  
```

```
basePath: /v1
consumes: [application/json]
produces: [application/json]
securityDefinitions:
  ApiKeyAuth:
    type: apiKey
    in: header
    name: Authorization
paths:
  /resource:
    get:
      summary: List resources
      tags: [Resource]
      parameters:
        - in: query
          name: page
          type: integer
      responses:
        "200":
          description: OK
          schema:
            $ref: "#/definitions/ResourceList"
    post:
      summary: Create resource
      tags: [Resource]
      parameters:
        - in: body
          name: body
          required: true
          schema: { $ref: "#/definitions/ResourceCreate" }
      responses:
        "201": { description: Created, schema: { $ref:
          "#/definitions/Resource" } }
definitions:
  Resource:
    type: object
    required: [id, name]
    properties: { id: {type: string}, name: {type: string} }
  ResourceCreate:
    type: object
    required: [name]
    properties: { name: {type: string} }
  ResourceList:
```

Skip to content

```

type: object
  properties: { items: { type: Skip to content
    "#/definitions/Resource" } } }
      { $ref:

```

(Why Swagger? It drives API design, docs, SDKs, and imports cleanly into API Gateway.)

Lecture_3_Microservices

7) What is an API? GET/POST/PUT/DELETE

1. API (Application Programming Interface)

1. A contract that lets software components talk in a standard way. On the web, REST APIs expose resources (URLs) with operations and schemas; great API design de-risks and accelerates the whole stack.

Lecture_3_Microservices

2. GET

1. Read-only retrieval; must be idempotent and safe; no body required.

3. POST

1. Create or process; not idempotent by default; body carries input.

4. PUT

1. Full replace of a resource at a known URL; idempotent.

5. DELETE

1. Remove a resource; idempotent (multiple DELETEs have same effect).

8) Explain the provided Swagger file (AI Customer Service API)

1. Overview

1. It's an OpenAPI 2.0 ("swagger: '2.0'") spec defining a single HTTPS JSON endpoint `POST /chatbot` that accepts a list of input messages and returns a list of messages as the chatbot's reply. The `basePath` is commented out (so paths are rooted at `/`).

Assignment1_Task_Sheet Lecture_3_Microservices

2. Operation details

1. **Path:** `/chatbot` with **POST** method.
2. **Purpose:** "NLU API" that uses state/profile to produce a context-aware reply.
3. **Request body:** `BotRequest → { "messages": [Message, ...] }`.
4. **Responses:**
 - `200 → BotResponse` with `messages` array (same `Message` schema).

- 403 / 500 → Error or Skip to content t32 and message:string .
- 5. **Tags/operationId:** NLU / semantic message used by SDK generators).

3. Model schemas (under definitions)

1. **Message** has type:string and unstructured: UnstructuredMessage .
2. **UnstructuredMessage** has id, text, timestamp (timestamp format is listed as datetime ; in OpenAPI 2.0 the canonical spelling is date-time —worth fixing).
3. **BotRequest/BotResponse** simply wrap arrays of Message .

4. End-to-end example

- Request

json

Copy code

```
POST /chatbot
Content-Type: application/json
{
  "messages": [
    {
      "type": "user",
      "unstructured": {
        "id": "u1",
        "text": "Suggest Italian tonight for 2 in Manhattan at 7pm",
        "timestamp": "2025-10-24T22:01:00Z"
      }
    }
  ]
}
```

- Response (200)

json

Copy code

```
{
  "messages": [
    {
      "type": "bot",
      "unstructured": {
        "id": "b1",
        "text": "Got it—Italian at 7pm in Manhattan for 2. I'll email
suggestions shortly.",
        "timestamp": "2025-10-24T22:01:01Z"
      }
    }
  ]
}
```

]

[Skip to content](#)

5. How you'd deploy/use it

1. Import the spec into Amazon API Gateway, back it with a Lambda (sendMessage) that calls your Lex bot, and return the Lex response in BotResponse . (This workflow is exactly how the course assignment wires Swagger → API Gateway → Lambda → Lex.)

[Assignment1_Task_Sheet](#) [Lecture_3_Microservices](#)

If you want, I can also convert that Swagger 2.0 spec to **OpenAPI 3.0**, add auth (API key/JWT), and generate a tiny **SDK usage snippet** for your frontend.



You have to answer the following questions:

1. What are Monolithic systems? Advantages and disadvantages?
2. What are Microservices? Advantages and disadvantages?
3. What is AWS: Lambda, API Gateway, Cognito, IAM, S3, , SNS, SQS
4. What is SDK Generation in API Gateway?
5. Explain the following recap:
<>
 - i. API Gateway
 - a. great API management tool
 - b. seamless integration with Swagger
 - c. generates SDKs for your API
 - ii. Lambda
 - a. serverless compute service
 - b. integrates with API Gateway
 - iii. Cognito
 - a. useful for login workflows
 - b. outputs temporary IAM credentials with custom permission</>

[Skip to content](#)

Thought for 18s >

1) Monolithic systems — definition, advantages, disadvantages

What: One deployable unit (single codebase/process or tightly bound modules) that contains UI, business logic, and data access.

Advantages

1. Simple to start: one repo, one build, one deploy.
2. In-process calls are fast; fewer network hops.
3. Easy end-to-end testing and debugging locally.
4. Strong transactional consistency (single DB).
5. Straightforward cross-cutting concerns (logging, auth) in one place.

Disadvantages

1. Scale is “all-or-nothing” (can’t scale one hot part alone).
 2. Slower releases—small change redeploys the whole app.
 3. Tight coupling → fragile code ownership and merge pain.
 4. Tech lock-in—the whole app shares frameworks/runtime.
 5. Poor fault isolation—one bug can take down everything.
 6. Grows into a “big ball of mud” over time.
-

2) Microservices — definition, advantages, disadvantages

What: A suite of small, independently deployable services around business capabilities, communicating over the network (HTTP, gRPC, messaging).

Advantages

1. Independent deploy/scale per service (cost & performance wins).
2. Team autonomy and faster parallel development.
3. Fault isolation—one service failure needn’t crash the whole app.
4. Polyglot freedom—choose best tech per service.
5. Clear bounded contexts → cleaner long-term codebases.

Disadvantages

1. Operational complexity (service Skip to content timeouts).
 2. Distributed data—consistency, transactions, and schemas get harder.
 3. Observability overhead (tracing, centralized logs/metrics).
 4. Network latency & costs vs in-process calls.
 5. Dev-env friction (spinning up many deps, contract testing).
 6. Cross-cutting concerns must be standardized (auth, rate-limits).
-

3) AWS services (quick, practical)

Lambda (serverless compute)

- Run functions on demand; auto-scales; pay per invocation/runtime.
- Great for APIs, event processing, cron; watch cold starts, timeouts.

API Gateway (managed API front door)

- Define REST/HTTP/WebSocket APIs, auth, throttling, usage plans.
- Integrates with Lambda, ALB, or any HTTP backend; request/response mapping.

Cognito (identity & auth)

- **User Pools:** sign-up/sign-in, JWT tokens (OIDC/OAuth2).
- **Identity Pools:** exchange identity for temporary AWS creds (STS) with IAM roles.

IAM (access control)

- Users, roles, policies. Grant least privilege to AWS resources/services.

S3 (object storage)

- Buckets for files; versioning, lifecycle, static website hosting, event triggers.

SNS (pub/sub notifications)

- Topics to fan-out messages to email, SMS, HTTP, SQS, Lambda.

SQS (message queue)

- Durable queues (standard or FIFO) to decouple producers/consumers; retries & DLQs.
-

4) What is SDK Generation in API Gateway?

- For **REST APIs**, API Gateway can **generate client SDKs** (historically Java, JavaScript, iOS/Android) from your API definition (often OpenAPI/Swagger).

- The SDK wraps your endpoints (Skip to content when configured), serialization, ~~base URL~~, and error models.
 - You download it from the console (or via CLI), drop it into your client, and call `client.methodName(params)` instead of hand-crafting HTTP.
 - Notes: it's mainly for REST APIs; many teams also use **OpenAPI Generator / Swagger Codegen** for broader language support and customization.
-

5) Recap explained

i. API Gateway

- **a. great API management tool:** Gives you a managed “front door” with routing, throttling, caching, request/response mapping, rate limits, usage plans, and auth (Cognito/JWT, Lambda authorizers).
- **b. seamless integration with Swagger:** Import an OpenAPI (Swagger) spec to create routes, models, and docs; keep API as code.
- **c. generates SDKs for your API:** For REST APIs, download ready-to-use client SDKs that include method stubs and SigV4 signing support.

ii. Lambda

- **a. serverless compute service:** Run code without managing servers; auto-scale, pay per use.
- **b. integrates with API Gateway:** API Gateway → Lambda is a common pattern for REST/HTTP APIs; mapping templates adapt HTTP payloads to Lambda’s event and back.

iii. Cognito

- **a. useful for login workflows:** User Pools provide hosted UI, sign-up/sign-in, MFA, password policies, social logins, and issue JWTs (ID/Access/Refresh).
- **b. outputs temporary IAM credentials with custom permission:** Identity Pools exchange an identity (e.g., User Pool token) for **temporary AWS credentials** via STS and attach **IAM roles** with fine-grained permissions (e.g., scoped S3 access). This enables secure, direct client→AWS access where appropriate.



You have to answer the following questions:

[Skip to content](#)

1. How Elasticity is supported in Amazon Services?
2. When would users need elasticity? What do they need for?
3. Features of elastic load balancing.
4. What is "Autoscale"?
5. How a IaaS cloud provider is able to provide machine or computing resources on demand.

Thought for 18s >

1) How is elasticity supported in Amazon (AWS) services?

1. **Autoscaling everywhere:**
 - **EC2 Auto Scaling Groups (ASG)** scale VM counts up/down by policies (CPU, requests/sec, SQS depth, custom metrics).
 - **Application Auto Scaling** adds the same to non-EC2 services (e.g., **DynamoDB R/W capacity, ECS tasks, Kinesis shards, SageMaker endpoints**).
2. **Elastic front doors: Elastic Load Balancing (ALB/NLB/GWLB)** spreads traffic across healthy targets and works with ASGs to add/remove capacity seamlessly.
3. **Serverless elasticity:** **Lambda** (concurrency scaling), **Fargate** (serverless containers), **Aurora Serverless v2**, **DynamoDB On-Demand**, **S3** (virtually unlimited storage).
4. **Observability-driven:** **CloudWatch** metrics/alarms feed scaling policies; **target tracking** (keep a metric at a setpoint) and **predictive scaling** smooth spikes.
5. **Global/regional pooling:** Multi-AZ and multi-region designs let AWS pull from shared resource pools to satisfy bursts without you pre-provisioning.

2) When do users need elasticity? What for?

1. **Spiky traffic:** product launches, flash sales, marketing campaigns, breaking news.
2. **Diurnal/seasonal patterns:** daytime peaks, weekend surges, holiday retail.
3. **Event-driven bursts:** queue backlogs, stream spikes, cron/batch windows.
4. **Cost control:** scale to zero/off-peak to avoid paying for idle capacity.
5. **Resilience:** replace failed capacity automatically; absorb DDoS/traffic shifts.
6. **Experimentation:** spin up large temporary fleets for tests/data processing.

[Skip to content](#)

3) Features of Elastic Load Balancing (ELB)

1. **Types:** ALB (HTTP/HTTPS, L7: host/path routing, WebSockets, gRPC), NLB (TCP/UDP, L4: ultra-high throughput, static IP/elastic IP), GWLB (L3/L4 for security appliances).
2. **Health checks & failover:** per-target health probes; only routes to healthy instances/containers.
3. **Smart routing** (ALB): host/path-based rules, header/query routing, weighted/canary, fixed or dynamic responses.
4. **Security & TLS:** TLS termination, SNI, integration with ACM certs, AWS WAF (ALB), security groups.
5. **Session features:** sticky sessions (cookies), connection draining (deregistration delay), slow-start (ALB).
6. **Scale & HA:** cross-zone load balancing, multi-AZ, auto capacity scaling; access logs, CloudWatch metrics.

4) What is “Autoscale”?

Definition: Automatic horizontal capacity adjustment to keep a target metric in range or follow a schedule—without manual intervention.

- **Core pieces:** min/max/desired capacity; scaling policies (**target tracking, step, scheduled/predictive**); CloudWatch alarms; **cooldowns**; health checks.
- **Signals:** CPU%, requests/target, queue length, latency, custom metrics (e.g., “items per second”).
- **Applied to:** EC2 (ASG), ECS/EKS tasks, DynamoDB capacity, Kinesis shards, SageMaker endpoints, Lambda (via concurrency settings).

5) How can an IaaS provider deliver compute “on demand”?

1. **Virtualization & isolation:** Hypervisors (and in AWS, the Nitro/KVM stack) slice physical hosts into secure VMs; lightweight snapshots speed boot.
2. **Resource pooling:** Large multi-tenant pools across AZs; schedulers place VMs/volumes where capacity exists.

3. **Image templates:** Prebuilt machine images ('s) + **EBS** snapshots let providers clone machines quickly.
4. **Software-defined everything:** APIs for network (VPC, ENI, security groups), storage (EBS/S3), and routing provision resources instantly.
5. **Automated control plane:** Orchestrators handle placement, health remediation, and lifecycle at scale; users call APIs/CLI to request/release capacity.
6. **Metering & quotas:** Fine-grained billing (per-second/minute) and guardrails ensure fair use and cost transparency.



You have to answer the following questions:

1. What is virtualization? Advantages, Disadvantages?
2. Explain types of virtualization in tabular format – Full, Para, Hardware assisted, OS-level
3. What is memory virtualization how is it implemented?
4. What is device and I/O virtualization?
5. What is a virtual appliance and how is it different from a virtual machine
6. Explain the following fields in Amazon lex: intent, utterances, slots, prompts, fulfillment
7. What is an Image?

Thought for 1m 43s >

Here you go—clear, exam-ready notes with quick citations back to your course PDFs.

1) What is virtualization? Advantages & disadvantages

Definition (what & why): Virtualization inserts a hypervisor layer that lets multiple OS instances (VMs) safely share one physical machine by trapping privileged instructions and mediating access to CPU, memory, network and devices. This enables on-demand creation of “virtual” machines and dynamic partitioning of resources.

Lecture_4.2_Virtualization

Lecture_4.1_Elasticity

Advantages (high-value):[Skip to content](#)

- **Consolidation & elasticity:** run many OSes on one host; spin VMs up/down on demand. [Lecture_4.1_Elasticity](#)

- **Isolation & safety:** hypervisor at ring-0; guests run non-privileged.

[Lecture_4.2_Virtualization](#)

- **Rapid provisioning:** boot prebuilt stacks (appliances/images) almost instantly.

[Lecture_4.2_Virtualization](#)

- **Live migration:** move running VMs between hosts for zero-downtime maintenance and load balancing. [Lecture_4.2_Virtualization](#)

Disadvantages (trade-offs):

- **Overheads/complexity:** extra layer (binary translation or device emulation) vs. bare metal; para-virt needs guest OS changes; hardware-assist requires CPU features.

[Lecture_4.2_Virtualization](#)

- **I/O indirection:** routing virtual devices to shared hardware can add latency/ops work.

[Lecture_4.2_Virtualization](#)

2) Types of virtualization (Full, Para, HW-assisted, OS-level)

Type	How it works	Guest OS changes?	Notes / Examples
Full virtualization	Hypervisor intercepts privileged ops (often via binary translation) and emulates hardware.	No	Classic VMware approach; maximal compatibility. Lecture_4.2_Virtualization
Paravirtualization	Guest OS is modified to call hypervisor for sensitive ops.	Yes	Xen's early model; lower overhead than full virtualization. Lecture_4.2_Virtualization Lecture_4.2_Virtualization
Hardware-assisted	CPU provides virtualization extensions (e.g., VT-x/AMD-V) to trap/handle privileged ops efficiently.	No	Reduces translation overhead vs. full virt. Lecture_4.2_Virtualization

Type	How it works	Skip to content ↗ S changes?	Notes / Examples
OS-level (containers)	Single kernel isolates processes into “containers” (namespaces/cgroups).	Not an OS guest; processes share host kernel	Lightweight isolation; different from VMs. <small>Lecture_4.2_Virtualization</small>

3) What is memory virtualization? How is it implemented?

The hypervisor virtualizes the OS's own virtual memory by adding **another MMU layer**—mapping each guest's (virtual→guest-physical) to real machine frames (guest-physical→host-physical). Practically, this means MMU/translation structures (e.g., shadow page tables / EPT/NPT with HW-assist) so multiple “guest MMUs” co-exist over the same RAM safely. Lecture_4.2_Virtualization

4) What is device & I/O virtualization?

The VMM routes each VM's I/O to **virtual devices** that emulate NICs/disks/etc., multiplexing them onto the shared physical hardware (e.g., virtual NICs, virtual switches). This ensures isolation while enabling sharing. Lecture_4.2_Virtualization Lecture_4.2_Virtualization

5) What is a virtual appliance? How is it different from a virtual machine?

- **Virtual appliance:** a **VM image file** that bundles an OS plus pre-installed application stack; “install once, replicate many,” so providers can boot instances quickly from a catalog. Lecture_4.2_Virtualization
- **Virtual machine (VM):** the **running instance** created by the hypervisor from such an image/appliance, with runtime config (IP rules, etc.). Lecture_4.2_Virtualization

6) Amazon Lex fields: intent, utterances, slots, prompts, fulfillment

- **Intent:** the goal/action your bot ‘[Skip to content](#) DiningSuggestionsIntent , GreetingIntent) that you define [...](#) Lex console. [Assignment1_Task_Sheet](#)
 - **Utterances:** example phrases that map to an intent; if you don’t provide enough, Lex can fall back to the fallback intent. [Sample_Quiz_Spring2025](#)
 - **Slots:** named parameters Lex must collect to fulfill an intent (e.g., location, cuisine, time, party size, email). [Assignment1_Task_Sheet](#)
 - **Prompts:** the bot’s questions to elicit required slot values during the conversation (e.g., “What cuisine would you like?”) and to confirm; you see this flow in the assignment where the bot converses to collect fields. [Assignment1_Task_Sheet](#)
 - **Fulfillment:** the action taken once an intent is ready—commonly a **Lambda code hook** that runs **before Lex responds**, letting you validate/manipulate parameters and craft the reply (and/or trigger downstream actions). [Assignment1_Task_Sheet](#) [Assignment1_Task_Sheet](#)
-

7) What is an “Image”?

Two common meanings you’ll encounter:

- **VM / appliance image:** a **virtual machine image file** containing an OS + preinstalled software stack used to create VMs quickly. [Lecture_4.2_Virtualization](#)
- **Container image (Docker):** a **read-only template** built from a Dockerfile (layered filesystem) that is used to create runnable containers; immutable and shareable.

[Lecture_5_Docker](#) [Lecture_5_Docker](#)

If you want, I can turn these into a 1-page crib sheet or add quick diagrams of hypervisor vs. containers.



You have to answer the following questions:

1. What is Docker Container Lifecycle?
2. What is a Docker Daemon?
3. Give Dockerfile template code and explain its fields.
4. What is a Docker Image?
5. What are the 2 modes of running- foreground and detached EXPLAIN IN TECHNICAL DETAIL but in simple words.

6. What's 'mes?
7. Skip to content works?
8. What is a Docker Compose?
9. What is Docker namespace and control groups?

Thought for 50s >

1) Docker container lifecycle

1. **States & flow (typical):** create → start (running) → pause/unpause (optional) → stop/kill (exited) → restart (back to running) → remove. Docker Engine (daemon) is the background service that performs these lifecycle operations. Lecture_5_Docker
 2. **Common commands that reflect lifecycle:** docker run (create+start), docker ps / ps -a (list running/all), docker stop / start / restart , docker rm , docker logs , docker exec . Lecture_5_Docker
-

2) What is the Docker daemon?

The **Docker Engine** is a client–server system with (a) the **daemon** (server process), (b) a REST API, and (c) the CLI. The daemon runs on the host and creates/manages **images, containers, networks, and volumes**, including their lifecycle. Lecture_5_Docker

3) Dockerfile – template + key fields

What it is: A text file of build instructions that produces an image in a consistent, reproducible way. Lecture_5_Docker

Minimal, practical template (Python web app):

dockerfile

 Copy code

```
# 1) Base image
FROM python:3.9-slim

# 2) Working directory inside the image
WORKDIR /app

# 3) Copy dependency manifest first for better layer caching
COPY requirements.txt .
```

[Skip to content](#)

```

# 4) Install system/app depender.
RUN pip install --no-cache-dir -r requirements.txt

# 5) Copy the rest of your source
COPY . .

# 6) (Optional) environment variables
# ENV PYTHONDONTWRITEBYTECODE=1

# 7) Expose container port (documentation hint)
EXPOSE 5000

# 8) Default process (PID 1) when the container starts
CMD ["python", "app.py"]
# (Alternative) ENTRYPOINT ["python","app.py"]

```

The highlighted fields (FROM/WORKDIR/COPY/RUN/EXPOSE/CMD) match the lecture example.

[Lecture_5_Docker](#)

4) What is a Docker image?

A **read-only, layered template** used to create containers. Built from a Dockerfile; layers are combined via a **union filesystem**; images are immutable and easily shared.

[Lecture_5_Docker](#)

5) Foreground vs detached mode (simple but technical)

- **Foreground (default):** docker run ... attaches your terminal's **STDIN/STDOUT/STDERR** to the container's main process (PID 1). You see logs live, and signals (e.g., Ctrl-C) are delivered to the process. Add **-it** to make it interactive (allocate a TTY and keep STDIN open).
- **Detached (-d) mode:** The container's process runs **in the background** with your terminal detached. You manage it via `docker logs`, `docker exec`, `docker stop/start`, etc. The process still becomes PID 1 **inside** the container's isolated namespace. (You can observe PIDs from host vs. inside a container as shown in the lecture demos.)

[Lecture_5_Docker](#)

[Lecture_5_Docker](#)

6) What are Docker vol

Skip to content

Managed directories on the host, outside a container's writable layer, used to **persist** and **share** data across container restarts and between multiple containers. Created and lifecycle-managed by Docker.

Lecture_5_Docker

7) What are Docker networks?

Virtual, Docker-managed networks that give containers isolated connectivity (IP addressing, DNS, routing) so they can talk to each other and/or the outside world. The Engine explicitly manages **networks** as first-class objects, and each container runs in an isolated environment with its **own processes, networks, and mounts**.

Lecture_5_Docker

Lecture_5_Docker

8) What is Docker Compose?

A YAML-driven tool to **define and run multi-container applications** (services, volumes, networks) together. Useful commands: `docker-compose up`, `down`, `ps`, `logs`.

Lecture_5_Docker

9) Docker namespaces & control groups

- **Namespaces:** kernel feature that **isolates** resources (PID, NET, MNT, UTS, IPC, USER). Processes inside a container “think” they’re PID 1, have their own interfaces/hostname, etc.
- **cgroups (control groups):** **limit and allocate** CPU, memory, I/O for a set of processes—ensuring fair distribution and preventing noisy neighbors.

Lecture_5_Docker

Lecture_5_Docker

Lecture_5_Docker

If you want, I can turn this into a one-page cheat sheet with the lifecycle commands on the left and the Dockerfile “build pipeline” on the right.



You have to answer the following questions in DETAIL -->
If explaining any of them requires explaining additional
things OR requires a scenario to explain - GO AHEAD

Explain “What is”, “What it does” and “How does it work” for following:

1. ReplicaSet and replicaController
2. Service
3. Ingress Controller
4. Masking of IP □ Ingress Domain Name(Service(actual IP of Pod))
5. Volumes □ local and remote
6. PersistentVolume, PersistentVolumeClaim □ How do PVs and PVCs work together.
7. Secret, ConfigMap

Thought for 33s >

Got it—here’s a clean, “What is / What it does / How it works” deep-dive for each Kubernetes concept, with tiny YAMLs and a simple scenario to make things stick.

1) ReplicaSet vs ReplicationController

What it is

1. **ReplicationController (RC)**: the *older* controller that keeps a specified number of Pods running. Equality-based selectors only. Largely superseded.
2. **ReplicaSet (RS)**: the *newer* RC with **set-based** label selectors (e.g., `in`, `notin`) and extra features. Typically managed by a **Deployment** (you seldom create RS directly).

What it does

- Ensures the desired number of *ready* Pods exist at all times (self-healing).
- Deletes extra Pods if there are too many; creates new ones if there are too few.
- When used via **Deployment**, supports rolling updates/rollbacks safely.

How it works

- Controller loop watches Pods matching its **label selector**.

- Compares **desired replicas** vs **current replicas**. Skip to content unless create/delete operations until they match.
- If a Pod dies (node crash, OOMKill), RS notices under-replication and creates a replacement.

Minimal ReplicaSet (usually created by a Deployment instead):

yaml

 Copy code

```

apiVersion: apps/v1
kind: ReplicaSet
metadata: { name: api-rs }
spec:
  replicas: 3
  selector:
    matchLabels: { app: api }
  template:
    metadata: { labels: { app: api } }
    spec:
      containers:
        - name: api
          image: ghcr.io/acme/api:1.0

```

2) Service

What it is

- A stable **virtual IP + DNS name** inside the cluster that fronts a dynamic set of Pods.

What it does

- **Load-balances** traffic across *ready* Pod endpoints.
- Decouples clients from Pod churn (scale, rollouts, failures).

How it works

- You define a **selector**; the Service's Endpoints/EndpointSlices are auto-populated with matching *ready* Pods.
- **kube-proxy** on each node programs **iptables/IPVS** rules to **NAT** connections destined to the Service's ClusterIP into one of the Pod IPs.
- Types: **ClusterIP** (default), **NodePort**, **LoadBalancer**, **ExternalName**, **headless** (**clusterIP: None**).

Example (ClusterIP):[Skip to content](#)

yaml

 [Copy code](#)

```

apiVersion: v1
kind: Service
metadata: { name: api-svc }
spec:
  selector: { app: api }
  ports:
    - port: 80
      targetPort: 8080

```

3) Ingress Controller

What it is

- **Ingress** is a *resource* that declares L7 (HTTP/HTTPS) routing rules (host/path → Service).
- An **Ingress Controller** is the *implementation* (e.g., NGINX, HAProxy, Traefik, AWS ALB) that reads Ingress resources and configures a reverse proxy / cloud load balancer accordingly.

What it does

- Centralizes **TLS termination, virtual hosting** (many domains), and **path-based routing**.
- Can add policies (rate limiting, auth plugins), and integrates with cert managers for HTTPS.

How it works

- The controller watches the API server for Ingress objects of its **ingressClass**.
- Rebuilds its proxy config (or cloud LB rules) to route Host + Path → **Service:Port**.
- Traffic flow (public): **Client** → **DNS** → **(Cloud) Load Balancer** → **Ingress Controller** → **Service** → **Pod**.

Example (NGINX Ingress):

yaml

 [Copy code](#)

```

apiVersion: networking.k8s.io/v1
kind: Ingress

```

```

metadata: Skip to content
  name: api-ing
  annotations: { kubernetes.io/ingress.class: nginx }

spec:
  tls:
    - hosts: [ api.example.com ]
      secretName: api-tls
  rules:
    - host: api.example.com
      http:
        paths:
          - path: /v1
            pathType: Prefix
        backend:
          service:
            name: api-svc
            port: { number: 80 }

```

4) “Masking of IP”: Ingress domain → Service (Pod IP hidden)

What it is

- A pattern where users hit `https://api.example.com` and never see Pod IPs. The domain resolves to a LB/Ingress; Ingress forwards to a **Service**; the Service NATs to **ephemeral Pod IPs**.

What it does

- **Masks** Pod churn: pods scale, roll, or die—clients still call the same domain.
- Enables TLS termination, WAF, and routing policies at the edge without exposing internals.

How it works (end-to-end)

1. **DNS** maps `api.example.com` → **LB public IP**.
2. LB forwards to **Ingress Controller** (pod/daemonset).
3. Ingress uses host/path rules to call **Service** `api-svc:80`.
4. **kube-proxy** picks a **Pod IP** from endpoints; **NAT** rewrites destination to the selected Pod.

5. Response NATs back transparently · learn the Pod IP; only the domain
Skip to content · (and maybe LB IP).
-

5) Volumes — local vs remote

What it is

- A **volume** gives a container a filesystem that outlives the container process (or shares data across containers in a Pod).

What it does

- Stores app data, caches, or config; supports persistence and data sharing.

How it works (two broad families)

- **Local (node-attached):**
 - `emptyDir` : ephemeral storage on the node; wiped when Pod is deleted.
 - `hostPath` : maps a host directory into the Pod (use cautiously).
 - **Local Persistent Volume**: binds a specific disk on a node; scheduled via node affinity.
- **Remote (network/block):**
 - Backed by **CSI** drivers (EBS/GCE PD/Azure Disk, NFS, Ceph, iSCSI, SMB, etc.).
 - Survives Pod rescheduling; can be re-attached to new nodes (subject to access mode).

Pod with an `emptyDir`:

yaml

 Copy code

```
volumes:  
  - name: scratch  
    emptyDir: {}  
containers:  
  - name: worker  
    image: alpine  
    volumeMounts:  
      - { name: scratch, mountPath: /tmp/work }
```

6) PersistentVolume (PV) & PersistentVolumeClaim (PVC)

What it is

- **PV**: a cluster-scoped storage resource (the *actual* provisioned storage).
- **PVC**: a namespaced *request* for storage (size, access mode, storage class).

What it does

- **Binds** app demands (PVC) to backing storage (PV) with the right size/class/zone/access mode.
- Supports **dynamic provisioning** via **StorageClass** so PVs are created on-the-fly.

How it works (binding & use)

1. You (or a controller) create a **PVC** (e.g., `20Gi, ReadWriteOnce, class gp3`).
2. The control plane finds/creates a **PV** that satisfies it.
3. PVC ↔ PV enter **Bound** state (1:1).
4. Pods mount the PVC; the kubelet attaches-mounts the underlying PV via its CSI driver.
5. On delete of the PVC, the PV follows its **reclaimPolicy**: `Delete` (destroy backing disk) or `Retain` (keep it).

StorageClass + PVC (dynamic provisioning):

yaml

 Copy code

```

apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata: { name: fast }
provisioner: ebs.csi.aws.com
parameters: { type: gp3, encrypted: "true" }
reclaimPolicy: Delete
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata: { name: data-pvc }
spec:
  accessModes: [ ReadWriteOnce ]
  storageClassName: fast
  resources: { requests: { storage: 20Gi } }

```

Pod consuming the PVC:

[Skip to content](#)

```

volumes:
  - name: data
    persistentVolumeClaim: { claimName: data-pvc }

containers:
  - name: db
    image: postgres:16
    volumeMounts:
      - { name: data, mountPath: /var/lib/postgresql/data }

```

Access modes refresher:

- “**RWO** (ReadWriteOnce): single node writer (most block volumes, e.g., EBS).”
 - “**RWX** (ReadWriteMany): multiple nodes can mount read/write (NFS/FSx/CSI that supports it).”
 - “**ROX** (ReadOnlyMany): many nodes read-only.”
-

7) Secret & ConfigMap

What it is

- **ConfigMap**: key/value **non-sensitive** configuration (flags, URLs, templates).
- **Secret**: key/value **sensitive** data (passwords, API keys, TLS certs). Stored base64-encoded in etcd; enable encryption-at-rest in production.

What it does

- Centralizes configuration, lets you **inject into Pods** via **environment variables** or **mounted files**.
- Allows clean separation of config from images (12-factor “config as env”).

How it works

- You create a ConfigMap/Secret.
- In your Pod spec, reference it either:
 - as **env vars** (`envFrom` / `env`), or
 - as a **volume** that projects keys as files.
- **Live updates**: when mounted as files, updates are reflected in-place (usually within a minute); env vars require Pod restart to pick up changes.
- Secrets can be of special types (`kubernetes.io/tls`, `kubernetes.io/dockerconfigjson`) and are namespaced.

ConfigMap + Secret + Pod example

[Skip to content](#)
[yaml](#)
Copy code

```

apiVersion: v1
kind: ConfigMap
metadata: { name: api-config }
data:
  LOG_LEVEL: "info"
  UPSTREAM_URL: "https://payments.example.com"
---
apiVersion: v1
kind: Secret
metadata: { name: api-secret }
type: Opaque
stringData:
  DB_PASSWORD: "s3cr3t!"
---
apiVersion: v1
kind: Pod
metadata: { name: api }
spec:
  containers:
    - name: api
      image: ghcr.io/acme/api:1.0
      envFrom:
        - configMapRef: { name: api-config }
        - secretRef: { name: api-secret }
      volumeMounts:
        - { name: tls, mountPath: /etc/tls, readOnly: true }
  volumes:
    - name: tls
      secret: { secretName: api-tls } # e.g., TLS cert+key

```

Quick scenario tying it all together

- You deploy a **Deployment** (manages a **ReplicaSet**) for `api` with `replicas: 5`.
- A **Service** (`api-svc`) exposes those Pods reliably inside the cluster.
- An **Ingress Controller** terminates TLS for `api.example.com` and routes `/v1` to `api-svc`.
- The app reads configuration from a **ConfigMap** and DB credentials from a **Secret**.

- The DB Pod mounts storage via a **PVC** bound to a **PV** dynamically provisioned from your cloud **StorageClass**.
- Clients only ever hit `api.example.com`; they never see Pod IPs—even as you roll versions or autoscale.

If you want, I can turn this into a one-page diagram (flow from DNS → Ingress → Service → Pod, plus PV/PVC wiring) and a ready-to-apply set of manifests for a small demo stack.



Ask anything

+  Extended thinking ▾



ChatGPT can make mistakes. Check important info.