1️⃣ Problem definition (1–2 minutes)

- Separate backend services per Salesforce object (e.g., Lead, Opportunity, Account), and each service integrated with Salesforce differently
- Integration was tight and synchronous, so failures cascaded and debugging was painful
- Burned engineering time without improving quality or user value
- Maintenance cost was high because there was no common platform or shared integration pattern across services
- Training implemented differently per object, which led to inconsistent user experience and uneven model quality

Goal

> Build a single, stable integration surface for Salesforce and move heavy compute to an async, tenant-isolated ML platform with reliable training + prediction and clear model lifecycle.

2️⃣ Functional requirements (3–4 minutes)

- Tenant onboarding (create tenant namespace + metadata)
- Consent management
  - allowTraining (use data for training)
  - allowStorePredictions (persist outcomes)
- Bulk export from Salesforce (per tenant) into staging (S3/lakehouse)
- Training orchestration per tenant (start, retry, resume, cancel)
- Store artifacts + metrics, register model versions, promote active model
- On-demand prediction for a Salesforce record (lead/opportunity/account etc.)
- Support both sync and async prediction modes
- Write results back into Salesforce objects/fields and/or Salesforce-hosted store
- Observability and admin APIs for status, run history, and failures

---

3️⃣ Non-functional requirements (2–3 minutes)

Highlight only what drives architecture:

- High availability for prediction path (99.9% target)
- Eventual consistency acceptable for training and async prediction callbacks
- Multi-tenant isolation (data + model + access)
- Burst handling (spiky predictions, bulk exports)
- Idempotency and resumability (exports, training runs, prediction requests)
- Compliance: consent gating must be enforced server-side
- Encryption in transit + at rest, least privilege IAM

This justifies **single integration layer + event-driven queues + orchestrated ML pipeline**.

4️⃣ APIs (2 minutes)

Keep it short and interview-ready:

Tenant + consent

- `POST /v1/tenants`
- `POST /v1/tenants/{tenantId}/consent`
- `GET  /v1/tenants/{tenantId}/status`

Export + training

- `POST /v1/tenants/{tenantId}/exports`
- `GET  /v1/tenants/{tenantId}/exports/{exportJobId}`
- `POST /v1/tenants/{tenantId}/training-runs`
- `GET  /v1/tenants/{tenantId}/training-runs/{trainingRunId}`

Prediction

- Sync: `POST /v1/tenants/{tenantId}/predict`
- Async: `POST /v1/tenants/{tenantId}/predict:async` + `GET /v1/predictions/{requestId}`

Callbacks to Salesforce (internal integration)

- `POST /v1/salesforce/callbacks/prediction-complete`
- `POST /v1/salesforce/callbacks/model-ready`

Mention **direct multi-service SF integration only to reject it**. Shows judgment.

5️⃣ Data model + partitioning (5 minutes)

Explain the minimum entities that make the system real:

- **Tenant**: `tenant_id`, `sf_org_id`, `status`, timestamps
- **Consent**: `tenant_id`, `allow_training`, `allow_store_predictions`, `updated_by`, `version`
- **ExportJob**: `export_job_id`, `tenant_id`, `sf_bulk_job_id`, `dataset_uri`, `schema_hash`, `status`
- **TrainingRun**: `training_run_id`, `tenant_id`, `export_job_id`, `status`, `sagemaker_job_arn`, `metrics_uri`, `model_artifact_uri`
- **ModelRegistry**: `tenant_id`, `model_version`, `status (staged/active/deprecated)`, `artifact_uri`
- **PredictionRequest**: `request_id` (idempotency key), `tenant_id`, `entity_type`, `entity_id`, `status`, `model_version_used`, `result_uri`

Partitioning / isolation (this is the key point):

- **Primary partition key: `tenant_id`**
- Storage layout: `s3://…/{tenant_id}/{export_job_id}/...` and `.../{model_version}/...`
- DB indexes: `(tenant_id, status)`, `(tenant_id, created_at)`, `(tenant_id, entity_id)`

This naturally leads into HLD.

6️⃣ High-level design (5–7 minutes)

You already have the right architecture. Walk through **one clean training path** and **one clean prediction path**.

Training path (write-heavy, async)

- Salesforce (Einstein/app) → API Gateway
- Tenant Service validates consent + config
- Export Worker uses SF Bulk API → writes dataset to S3/lakehouse
- Orchestrator creates TrainingRun → sends message to `SQS.training`
- Workers/Lambda start SageMaker training job
- Artifacts + metrics → S3
- ModelRegistry updated → new active model (optional gating)
- Callback worker updates Salesforce (model-ready + metadata)

Prediction path (read/compute, sync + async)

- Salesforce → API Gateway → Inference Service
- Resolve tenant active model version
- **Sync**: call warm model endpoint or cached runtime → return prediction
- **Async**: enqueue `SQS.inference` → worker runs prediction → callback writes back to Salesforce

Don't enumerate every box. Focus on flow and why boundaries exist.

7️⃣ Deep dive (10–15 minutes)

Slow down here. Pick **1–2 problems** that show engineering depth.

Deep dive A: Consent-gated training correctness under retries (the "hard" one)

Cover only:

- **Server-side consent gate**
  - Orchestrator checks `Consent.allow_training == true` at training-run creation

    time
  - Option: re-check right before starting SageMaker job (protect against mid-flight revoke)
- **Idempotency**
  - `trainingRunId` is the idempotency key
  - Workers must be able to receive duplicates and do nothing if already `Succeeded/Running`
- **At-least-once delivery**
  - SQS may deliver duplicate messages
  - Worker logic: read TrainingRun state → only start job if state == `Queued`
- **Resumability**
  - ExportJob and TrainingRun statuses are persisted
  - If export fails, retry export without duplicating datasets (use `exportJobId` path)
- **DLQ + replay**
  - Poison messages go to DLQ with reason
  - Operator can replay after fixing config/schema mismatch

This is where interviewers engage because it's real failure-mode thinking.

Deep dive B: Prediction spike control and callback reliability

Cover only:

- **Sync vs async switching**
  - Use API Gateway throttles per tenant
  - If queue depth high or endpoint saturated → force async
- **Model loading strategy**
  - Warm endpoints for top tenants (latency)
  - Lambda runtime load for long tail (cost), accept cold-start
  - Hybrid: keep N hottest models warm
- **Request idempotency**
  - `requestId` required from Salesforce
  - if same requestId repeats → return stored result or current status
- **Callback delivery**
  - Salesforce write-back can fail (rate limits, transient errors)
  - Use retry with backoff + "pending callback" status

  - Persist last error + next retry time for ops visibility

8️⃣ Tradeoffs and closing (2 minutes)

End strong:

- Async pipeline adds backend complexity (orchestrator, queues, retries)
- Eventual consistency for training and async prediction callbacks
- Some latency/cost tradeoffs depending on warm endpoints vs Lambda model loads
- But you get:
  - Cleaner Salesforce integration (single surface)
  - Much better reliability under spikes
  - Tenant isolation + consent enforcement
  - Real model lifecycle and auditability
  - Easier evolution over time (new models/features without breaking SF)