

Hotel Matching System - MLOps Design Answers

1. How you would build data pipelines that ingest new / updated hotel records from different source systems.

We can structure the components as follows:

A. Data Collection Jobs

- Separate scheduled jobs for each data source (GDS, OTA, Direct)
- Each job pulls data, cleans basic fields, and saves raw data into AWS S3
- Runs on a flexible schedule (daily/hourly) depending on the source's update frequency

B. Processing Pipeline (Airflow)

- Reads the raw data from S3
- Performs validation of required fields, formats, and schema
- Cleans and standardizes fields such as country codes, phone numbers, and date formats
- Outputs cleaned and structured data into a separate S3 location

C. Feature Storage (Feast)

- Loads cleaned data from S3
- Builds ML-ready features used by the matching system
- Stores features in a fast-access feature store for online and batch use

D. Deployment

- All pipeline jobs run as Docker containers managed by Kubernetes
- Auto-scales based on data volume
- GitLab CI/CD enables one-click deployment from code to production

E. Monitoring

- Grafana dashboards show real-time metrics such as records per source, pipeline health, and processing latency
- Prometheus collects custom metrics, and AlertManager sends Slack alerts for failures or missing data
- ELK Stack (Elasticsearch, Logstash, Kibana) stores and indexes logs for fast debugging

Tools: Airflow (orchestration), Python (data processing), AWS S3 (storage), Feast (feature store), Docker + Kubernetes (deployment and scaling), Prometheus + Grafana + ELK (monitoring and logging)

2. ***How these pipelines are versioned and reproducible (data versioning, feature store, etc.).***

We ensure all pipelines are fully versioned and reproducible across code, data, and infrastructure.

A. Code Versioning

- All pipeline code stored in Git (GitLab/GitHub)
- Every change tagged with a version (v1.0, v1.1, etc.)
- Docker containers built directly from specific Git commit hashes

B. Data Versioning

- DVC (Data Version Control) tracks datasets stored in S3
- Each pipeline run records input data version, output data version, and code version used
- Enables complete rollback to previous data states

C. Feature Store (Feast)

- Separates feature definitions (code) from feature values (data)
- Tracks lineage to identify which pipeline version generated which features
- Ensures training/serving consistency for ML models

D. Pipeline Versioning

- Airflow DAGs stored and versioned in Git
- Each DAG run logs start/end time, parameters, and success/failure status
- MLflow tracks experiments, pipeline artifacts, and model metadata

E. Environment Reproducibility

- Docker images encapsulate the environment for each pipeline stage
- Conda or Pipenv lock files ensure identical dependency versions
- Terraform versions all infrastructure components (K8s, S3, IAM, etc.)

F. Reproducible Runs

To recreate any past pipeline run:

```
git checkout <commit_hash>
```

```
dvc checkout
```

```
docker run <image_tag>
```

Result

Every hotel matching output can be traced back to the exact code, dataset, and environment that produced it, ensuring full reproducibility and auditability.

3.

How you would support the data science team in training models such as Pairwise classifiers (same hotel vs not)

We enable the data science team through a structured ML platform:

A. Labelled Training Data Generation

- Weak supervision: auto-label hotel pairs using rules (e.g., same postal code + similar name = match)
- Human-in-the-loop UI: data scientists review and correct uncertain pairs
- Training set versioning: each labeled dataset versioned in DVC

B. Feature Engineering Support

- Feature Store (Feast): pre-computed similarity features:
 - Name similarity (Jaccard, Levenshtein, embeddings)
 - Address similarity
 - Phone/postal match flags
 - Geographic distance
- Feature catalog: documented and searchable features

C. Experiment Management

- MLflow Tracking:
 - Log experiments, parameters, and metrics
 - Compare model performance
 - Store trained models
- JupyterHub: pre-configured notebooks with data access

D. Training Pipeline

- i. `fetch_labeled_pairs()` # From feature store
- ii. `train_test_split()` # Time-based split
- iii. `train_pairwise_model()` # LightGBM/XGBoost
- iv. `evaluate()` # Precision/recall @ threshold
- v. `log_to_mlflow()` # Version model

E. Evaluation & Validation

- Holdout validation set from a different time period
- Cross-validation strategies for small datasets
- Business metrics: false match rate, false non-match rate
- Shadow testing: compare new model vs baseline on live data

F. Tools & Access

- S3 buckets for training data
 - Feature store API for feature retrieval
 - MLflow UI for experiment comparison
 - Compute resources: GPU/CPU clusters via Kubernetes
-

3. *How you would register model versions in a model registry with stages such as:*

- *Development*
- *Staging*
- *Production*

We use MLflow Model Registry with three stages:

A. Development

- New models from training
- Tested on sample data
- Access restricted to the data team

B. Staging

- Models that passed initial tests
- Tested on real data but not yet used for production decisions
- Promotion requires team lead approval

C. Production

- Models actively making decisions
- Closely monitored for performance and issues
- Can roll back to previous versions if problems occur

D. Approval Workflow

- Data scientist creates a Pull Request to promote a model
- Team lead reviews metrics and validation results in the PR
- If approved, GitHub Actions / GitLab CI runs automated tests
- Upon successful tests, the model moves to the next stage
- All changes are logged for auditing purposes

Tools: MLflow + GitHub/GitLab CI/CD + Slack notifications

4. How you would deploy the hotel matching service, for example:

- *Batch job that periodically **clusters hotels into canonical IDs**.*
- *Online service that, given a new hotel record, returns:*
 - *An existing **canonical_hotel_id** if it matches an existing cluster.*
 - *Or creates a **new canonical hotel** if no match is found.*

A. Batch Job (Nightly)

- Runs every night
- Processes all hotel data
- Groups duplicates into canonical IDs
- Updates the main hotel database

B. Online API (Real-time)

- FastAPI service running continuously
- Receives new hotel records in real time
- Checks against existing canonical groups
- Returns a matching hotel ID or creates a new one
- Used by booking systems when new hotels are added

C. How Both Work Together

- The online API uses the previous night's cluster results for fast matching
- The nightly batch job corrects mistakes and updates clusters for the next day

6. What the API or interface would look like, e.g.:

- *POST /match_hotel* with a new hotel record that returns *canonical_hotel_id* + *match score*.

A. Endpoint

- POST /match_hotel

B. Request

- Send hotel details (name, address, city, etc.) as JSON

C. Response

- Returns:
 - canonical_hotel_id (existing or new)
 - match_score (0–1)
 - status (“MATCHED” or “NEW”)

D. Example

```
{"canonical_hotel_id": "HOTEL_001", "match_score": 0.92, "status": "MATCHED"}
```

E. Features

- Fast (under 100ms)
- Versioned (/v1/)
- Authenticated using API keys
- Logged for monitoring

7. How you would handle latency and scalability as the number of hotel records grows.

A. Reduce Latency

- Caching: Redis cache for recent matches and for cluster representatives
- Efficient search: vector similarity search using FAISS, and city-based indexing to limit the search space
- Optimized code: precompute features instead of computing in real time, and use compiled similarity functions

B. Increase Scalability

- Microservices: separate matching, clustering, and API services to scale independently
- Database: PostgreSQL with read replicas and partitioning by region or country
- Batch processing: nightly clustering using Spark with incremental updates instead of full reprocessing

C. Monitoring & Auto-scaling

- Kubernetes auto-scales based on request rate
- Alerts trigger when latency exceeds 100ms
- Load testing performed before high-traffic events

D. Result

The system can scale from 10 to 10 million hotels without redesign.