AutoML FraudShield – SRS + TDD + Deployment Manifest (v1)

0. Project Overview

Project Name: AutoML FraudShield
Goal: Build a production-grade, continuously learning fraud detection system that:

- Ingests transactional data (batch + streaming)
- Maintains an online/offline feature store
- Trains and evaluates models via a reproducible ML pipeline
- Registers, versions, and deploys models via a model registry
- Monitors performance & drift and triggers retraining when needed
- Exposes a low-latency scoring API for real-time predictions

Primary Audience:

- Hiring managers & teams evaluating you as an AI/ML Engineer, MLOps Engineer, or AI Architect.
- Secondary: Fintech / fraud / risk stakeholders who want to see a full ML lifecycle.

Target Platform (v1): Google Cloud Platform

- **Data & Storage:** BigQuery, Cloud Storage
- **Feature Store:** Vertex AI Feature Store
- **Pipeline Orchestration:** Vertex AI Pipelines (orchestrated via Python SDK)
- **Training & Experiments:** Vertex AI Custom Training + Vertex Experiments
- **Model Registry:** Vertex AI Model Registry

- **Serving:** FastAPI on Cloud Run (online scoring)
- **Monitoring & Drift:** BigQuery + scheduled pipeline + dashboards (Looker / custom)
- **IaC:** Terraform

---

1. Scope & Requirements (SRS)

1.1 In-Scope (v1)

1. **Data ingestion (batch)**
    - Load historical transactions from CSV/Parquet files into BigQuery.
    - Apply schema validation and basic quality checks.
2. **Feature engineering & feature store**
    - Define a small but realistic feature set for fraud detection:
        - Transaction-level features (amount, time, merchant, channel).
        - Customer-level aggregates (rolling sum/count, average ticket, recency).
        - Device/geo features (country, IP risk score – stubbed or derived).
    - Materialize features into **Vertex AI Feature Store**:
        - Offline store (BigQuery-backed).
        - Online store (low latency feature reads for scoring API).
3. **Model training & evaluation**
    - Implement a reproducible pipeline that:
        - Pulls labeled data from the offline feature store / BigQuery.
        - Splits into train/validation/test with time-aware split.

        - Trains at least 2–3 models (e.g., XGBoost, LightGBM, a simple DNN).
        - Logs metrics and artifacts (ROC AUC, PR AUC, confusion metrics, calibration).
        - Stores metrics in Vertex Experiments.

4. **Hyperparameter optimization (HPO)**
    - Run a small hyperparameter search (grid or Bayesian) for the primary model.
    - Automatically select the best model based on ROC AUC (primary) and PR AUC (secondary).
5. **Model registry and deployment**
    - Register the winning model in **Vertex Model Registry** with:
        - Model version
        - Training data reference
        - Metrics
    - Deploy the best model to:
        - A **Vertex Endpoint** for internal use (optional but nice), and
        - A **FastAPI scoring service on Cloud Run** that calls:
            - Vertex Endpoint **or**
            - A local model artifact (configurable for portability).
6. **Online prediction API**
    - FastAPI / Cloud Run service:
        - `POST /v1/score` that accepts a JSON payload of transaction attributes.
        - Calls feature store for additional features given entity IDs.
        - Calls model (Vertex Endpoint or local artifact) and returns:

        - Fraud probability (0–1)
        - Risk band (e.g., Low/Med/High)
        - Model version used
        - Optional explanation stub (e.g., SHAP in v2).

7. **Monitoring & drift**
    - Log all predictions + ground truth labels (when they arrive) to BigQuery.
    - Implement a **scheduled Vertex Pipeline** or Cloud Scheduler job that:
        - Computes performance over recent time windows.
        - Computes input distribution drift vs. training baseline.
        - Emits a “retrain recommended” signal when thresholds are breached.
8. **Automatic retraining (triggered)**
    - Expose a retraining pipeline entrypoint that:
        - Can be run manually (CLI / SDK call).
        - Can be triggered by a Cloud Scheduler job when:
            - Data volume threshold met (e.g., N new labeled days), or
            - Drift metrics exceed threshold.

1.2 Out-of-Scope (v1)

- Complex multi-tenant isolation and billing.
- Formal approval workflows / human-in-the-loop UI.
- Real production-grade label ingestion from external systems (we’ll mock labels).
- Fully robust security hardening & compliance (we’ll follow best practices but not formal audits).

1.3 Functional Requirements (FR)

FR-1 – The system must train at least one baseline model and one tuned model and store results.
FR-2 – The pipeline must log metrics including ROC AUC, PR AUC, and confusion matrix.
FR-3 – The system must register the best model to a model registry with metadata.
FR-4 – The scoring API must handle at least 50–100 RPS at p95 latency < 200 ms (with warm instances).
FR-5 – Predictions must include model version for traceability.
FR-6 – The drift job must compute at least one feature distribution metric (e.g., PSI, KL divergence, or simple mean/std diff) vs. the training window.
FR-7 – The retraining pipeline must be runnable via CLI / SDK without manual reconfiguration.

1.4 Non-Functional Requirements (NFR)

NFR-1 – Reproducibility:
All training and evaluation must be traceable and reproducible from code + configs.

NFR-2 – Observability:

- Key stages emit logs + metrics.
- Training runs are visible in Vertex Experiments or MLflow-like UI.

NFR-3 – Configurability:

- Environment configuration (dev/stage/prod) controlled via config files + Terraform variables.

NFR-4 – Cost awareness:

- Pipelines should be designed to run on modest resources (e.g., small custom training cluster).

---

2. High-Level Architecture (TDD – System View)

2.1 Components

1. **Data ingestion & prep**

    - Scripts to load CSV/Parquet into BigQuery.
    - Data validation using `great_expectations` or `pandera` (optional).
2. **Feature engineering service**
    - Python module to generate offline features.
    - Vertex AI Feature Store entity types:
        - `customers`
        - `cards`
        - `merchants`
    - Feature groups for aggregates (e.g., 7-day transaction count).
3. **Training pipeline (Vertex AI Pipelines)**
    - Steps:
        - Load training data (BigQuery → DataFrame).
        - Join with features from Feature Store (offline).
        - Train baseline and tuned models.
        - Evaluate and compare.
        - Register best model.
        - Optionally deploy to Vertex Endpoint.
4. **Model registry**
    - Vertex Model Registry entries with:
        - `model_id`, `version`, `metrics`, `train_data_ref`, `created_at`.
5. **Scoring API**

- FastAPI application:
    - `POST /v1/score`
- Reads online features from Feature Store given `customer_id` & `card_id`.
- Calls Vertex Endpoint or local model.

6. **Monitoring & retraining**
    - BigQuery table: `fraudshield.predictions_log`
    - Scheduled job:
        - Pulls last N days predictions + labels.
        - Computes performance + drift metrics.
        - Triggers training pipeline when conditions met.

2.2 Data Flow (Conceptual)

1. **Initial setup**
    - Load historical labeled transactions → BigQuery.
    - Materialize initial features into Feature Store.
2. **Model training**
    - Pipeline runs:
        - Query training dataset (transactions + features).
        - Train & evaluate.
        - Register + deploy best model.
3. **Online scoring**
    - Client → FastAPI `/score` with transaction payload.

   - Service:
     - Fetch features (online FS) for entities.
     - Call model.
     - Log prediction + payload to BigQuery.
4. **Monitoring & retrain**
   - Scheduled job computes performance & drift.
   - If thresholds exceeded → run training pipeline again.
   - New best model registered & deployed.

---

3. Detailed Design (TDD – Component-Level)

3.1 Data Model (Simplified)

BigQuery Table `fraudshield.transactions`

- `transaction_id` (STRING, PK)
- `customer_id` (STRING)
- `card_id` (STRING)
- `merchant_id` (STRING)
- `timestamp` (TIMESTAMP)
- `amount` (FLOAT64)
- `currency` (STRING)
- `channel` (STRING) // e.g., POS, eCommerce, ATM

- `is_fraud` (BOOL)

Feature Store Entity Types

1. `customers`
    - key: `customer_id`
    - features (examples):
        - `customer_txn_count_7d` (INT64)
        - `customer_txn_amount_sum_7d` (FLOAT64)
        - `customer_avg_ticket_30d` (FLOAT64)
        - `customer_geo_entropy_30d` (FLOAT64 – optional stub)
2. `cards`
    - key: `card_id`
    - features:
        - `card_txn_count_7d`
        - `card_txn_amount_sum_7d`
3. `merchants`
    - key: `merchant_id`
    - features:
        - `merchant_txn_count_30d`
        - `merchant_chargeback_rate_90d` (stubbed or simulated)

BigQuery Table `fraudshield.predictions_log`

- `prediction_id` (STRING)
- `transaction_id` (STRING)
- `customer_id` (STRING)
- `card_id` (STRING)
- `timestamp` (TIMESTAMP)
- `score` (FLOAT64)
- `risk_band` (STRING)
- `model_version` (STRING)
- `is_fraud` (BOOL, nullable)
- `logged_at` (TIMESTAMP)

3.2 Training Pipeline Steps (Pseudo-Implementation)

Step 1 – Load training data

- SQL query over `transactions` with filters (date range, not null label).
- Option to downsample majority class (non-fraud) or use class weights.

Step 2 – Join with features

- Call Feature Store offline API to join entity features.
- Produce a training DataFrame with:
    - Raw features (amount, channel, etc.)
    - Engineered aggregates (from FS)
    - Label (`is_fraud`)

Step 3 – Train baseline & tuned models

- Baseline: XGBoost with default-ish parameters.
- Tuned: HPO with Optuna or Vertex Hyperparameter Tuning.

Step 4 – Evaluation

- Time-aware split: earliest 70% for train, next 15% for valid, last 15% for test.
- Compute metrics:
  - ROC AUC, PR AUC
  - F1 at a chosen threshold
  - Confusion matrix
- Log to Vertex Experiments.

Step 5 – Model selection & registry

- Compare candidate models by ROC AUC (primary) and PR AUC (tie-break).
- Push winning model to Model Registry with:
  - `metrics.json`
  - `training_pipeline_id`
  - `data_ref` (e.g., BQ snapshot + FS version).

Step 6 – Deployment

- Option 1: Deploy to Vertex Endpoint (standard deployment).
- Option 2: Export model artifact (e.g., `model.pkl`) to GCS where Cloud Run can mount or download.

3.3 Scoring API Design

Endpoint: `POST /v1/score`

Request (example):


```
{
  "transaction_id": "tx_123",
  "customer_id": "cust_456",
  "card_id": "card_789",
  "merchant_id": "m_001",
  "timestamp": "2025-01-15T12:34:56Z",
  "amount": 125.40,
  "currency": "USD",
  "channel": "ECOM"
}
```


Steps:

1. Validate payload.
2. Read online features from Feature Store:
   - `customers` by `customer_id`
   - `cards` by `card_id`
   - `merchants` by `merchant_id`
3. Assemble feature vector for the model.
4. Call Vertex Endpoint or local model artifact.
5. Map score to risk band, e.g.:
   - `score < 0.2` → LOW
   - `0.2 ≤ score < 0.6` → MEDIUM
   - `score ≥ 0.6` → HIGH
6. Write log row to `predictions_log`.

Response (example):

```json
{
  "transaction_id": "tx_123",
  "score": 0.82,
  "risk_band": "HIGH",
  "model_version": "fraudshield-xgb-v5",
  "explanations": null
}
```

---

4. Deployment Manifest (DM – Infra & Environments)

4.1 Environments

- **dev**: single project, low quotas, cheap resources, debug logging.
- **prod**: separate GCP project, stricter IAM, higher quotas, monitoring.

4.2 GCP Resources (per project)

- **BigQuery**
  - Dataset: `fraudshield`
  - Tables: `transactions`, `predictions_log`, plus intermediate views.
- **Cloud Storage**
  - `gs://fraudshield-raw-{env}` (raw CSV/Parquet)
  - `gs://fraudshield-artifacts-{env}` (models, configs)
  - `gs://fraudshield-pipeline-logs-{env}`
- **Vertex AI**
  - Feature Store:

            - Entity types: `customers`, `cards`, `merchants`
    - Pipelines:
        - `fraudshield_training_pipeline`
        - `fraudshield_monitoring_pipeline`
    - Model Registry:
        - Artifact name prefix: `fraudshield-`
    - (Optional) Online Endpoint:
        - `fraudshield-online-endpoint`
- **Cloud Run**
    - Service: `fraudshield-api-{env}` (FastAPI scoring)
    - Min instances: 1 (for low cold start).
- **Cloud Scheduler**
    - Job: `fraudshield-monitoring-daily` → triggers monitoring pipeline.
    - Job: `fraudshield-retraining-weekly` (optional; may instead be triggered by monitoring).
- **Logging/Monitoring**
    - Cloud Logging & Monitoring dashboards built on:
        - API latency & error rates
        - Training runs
        - Drift metrics (visualized from BigQuery tables)

4.3 Terraform Layout

- `infra/terraform/envs/dev/main.tf` – Dev project + resources.
- `infra/terraform/envs/prod/main.tf` – Prod project + resources.
- Variables for project ID, region, bucket names, etc.

---

5. Repository / Directory Structure

Proposed repo layout:

```
fraudshield/
├── README.md
├── pyproject.toml              # or setup.cfg/requirements.txt
├── docs/
│   ├── FraudShield_SRS_TDD_DM_v1.md
│   └── architecture_diagrams/  # (mermaid or PNG)
├── infra/
│   └── terraform/
│       ├── modules/
│       │   ├── bigquery/
│       │   ├── feature_store/
│       │   ├── vertex_pipelines/
│       │   ├── cloud_run_api/
│       │   └── monitoring/
│       └── envs/
│           ├── dev/
│           │   ├── main.tf
│           │   └── variables.tf
│           └── prod/
│               ├── main.tf
│               └── variables.tf
├── data/
│   ├── sample_transactions.csv
│   └── schemas/
│       ├── transactions_schema.json
│       └── predictions_log_schema.json
├── features/
```

```
│   ├── feature_definitions.py     # high-level feature sets
│   ├── build_offline_features.py
│   ├── register_feature_store.py
│   └── tests/
│       └── test_feature_defs.py
├── pipelines/
│   ├── training/
│   │   ├── pipeline_definition.py # Vertex AI Pipelines DAG
│   │   ├── components/
│   │   │   ├── load_data_component.py
│   │   │   ├── build_dataset_component.py
│   │   │   ├── train_model_component.py
│   │   │   ├── evaluate_model_component.py
│   │   │   └── register_model_component.py
│   │   └── launch_training_pipeline.py
│   ├── monitoring/
│   │   ├── drift_monitoring_pipeline.py
│   │   └── launch_monitoring_pipeline.py
│   └── utils/
│       ├── metrics.py
│       └── bq_utils.py
├── api/
│   ├── app/
│   │   ├── main.py              # FastAPI app
│   │   ├── config.py
│   │   ├── schemas.py           # Pydantic models
│   │   ├── routers/
│   │   │   └── scoring.py
│   │   ├── services/
│   │   │   ├── feature_store_client.py
│   │   │   ├── model_client.py # Vertex Endpoint or local model
│   │   │   └── logging_client.py
│   │   └── tests/
│   │       ├── test_scoring_endpoint.py
│   │       └── test_feature_fetch.py
│   └── Dockerfile
├── models/
```

```
│   ├── train_model.py          # direct training script (for local
dev)
│   ├── model_utils.py
│   └── artifacts/              # local model artifacts for dev
└── notebooks/
    ├── 01_exploration.ipynb
    ├── 02_feature_prototyping.ipynb
    └── 03_model_sanity_check.ipynb
```

---

6. Roadmap / Future Enhancements

- **v2:** Add SHAP-based explanations to `/score` responses.
- **v2:** Add multi-tenant support via `tenant_id` dimension.
- **v2:** Add UI dashboard (Streamlit) for:
  - Monitoring metrics
  - Triggering retrains
  - Browsing model versions
- **v3:** Replace Vertex Feature Store with Feast for portability (optional).