I. High-Level Architectural Flow

The system is built on an **Event-Driven Architecture (EDA)**. This means the frontend (Dashboard) and backend (Processing) are decoupled. The "glue" holding them together is the Kafka message bus (Confluent Cloud).

II. Detailed Technical Flow

Phase 1: Identity & Entitlements (Steps 1–3)

Goal: Establish *who* the user is and *what* they are allowed to do without hardcoding logic in the application code.

- **1. Authentication (Clerk / Auth0):**
  - User signs up on the frontend.
  - **Result:** User gets a JWT (JSON Web Token) containing sub: "user_123". This token is sent with every subsequent API request.[1]

- **2. Entitlement Service (The Gatekeeper):**
  - **Database:** A table in **Neon (Postgres)** called user_subscriptions.
  - **Workflow:**
    1. User buys a plan via Stripe on the frontend.
    2. Stripe sends a webhook (checkout.session.completed) to your Go backend.
    3. Backend updates Neon:
       ```sql
       INSERT INTO user_subscriptions (user_id, plan, features, expiry)
       VALUES ('user_123', 'ADVANCED', '{"realtime": true, "alerting": true}', '2026-01-01');
       ```

  - **Frontend Check:** When the dashboard loads, it calls GET /v1/user/me. The backend joins the Clerk ID with the Neon subscription table. If plan == 'FREE', the "API Keys" tab is grayed out.

Phase 2: The Unified Ingestion Layer (Steps 4 & 8)

Goal: Treat Batch (CSV) and Streaming (API) as the *same* data source for the detection engine.

- **Path A: Batch Ingestion (Async CSV):**
  1. **Upload:** User POSTs a file to the backend.
  2. **Storage:** Backend streams the file to **Cloudflare R2** (S3 compatible) to avoid memory crashes.
  3. **Hydration:** A background Goroutine reads the file from R2 line-by-line.
  4. **Publish:** It validates each row and pushes it to the **Confluent Kafka** topic transactions.
  5. **User Feedback:** The API returns 202 Accepted immediately. A WebSocket or polling

mechanism updates the UI progress bar.

- **Path B: Real-Time Ingestion (API - Advanced Plan):**
    1. **Ingress:** Go-based service receives POST /transaction.
    2. **Entitlement Check:** Checks **Upstash Redis** cache: GET subscription:user_123. If invalid, return 403 Forbidden.
    3. **Publish:** Pushes directly to the *same* transactions topic used by the CSV worker.

Technical Win: The downstream ML engine doesn't know if a transaction came from a CSV or a live API call. It processes everything uniformly.

Phase 3: The Detection Engine (Step 5)

Goal: Apply hybrid detection logic (Rules + ML) in parallel.

- **The Message Bus (Kafka Topic: transactions):** All data lands here first.
- **Step 5.1 & 5.2: Enrichment & Rules (Stream Processor):**
    - **Technology:** A Python worker (using confluent-kafka and pandas for windowing) or a lightweight Flink job.
    - **Stateful Processing:** It maintains a rolling window of the last 10 minutes of transactions for this user.
    - **Feature Lookup:** It queries **Upstash Redis** for historical baselines (e.g., user_avg_spend).
    - **Rule Execution:** It runs deterministic checks (Velocity, Geo-fencing).
    - **Output:** Adds metadata to the event: {"rules_triggered": ["VELOCITY_HIGH"]}.
- **Step 5.3: ML Inference (The Model):**
    - **Service:** A Python consumer reads the enriched stream.
    - **Model:** Uses **ONNX Runtime** to run a pre-trained Isolation Forest model. This is lightweight and runs inside the container.
    - **Output:** Generates a probability score (e.g., 0.92).
    - **Non-Blocking:** This happens asynchronously; it does not block the ingestion API.
- **Step 5.4 & 5.5: The Decision & Explainability (Aggregation):**
    - **Logic:** A final "Aggregator" function combines the Rule Output and ML Score.
    - **Explainability Generator:** It maps error codes to human strings.
        - *Input:* rule: VELOCITY_5_MIN, ml_feature_contribution: [amount, time]
        - *Output:* "High velocity detected: 5th transaction in 5 minutes. Amount ($5000) is significantly above user average ($150)."

Phase 4: Storage & Presentation (Steps 6–7)

Goal: Efficiently store data for the dashboard.

- **Database (Neon Postgres):**
    - We use a standard Postgres table, partitioned by date if possible.
    - Schema: transactions table stores the raw data, anomalies table stores the detected issues with the JSON explanation.
- **Dashboard API:**

    ◦ The frontend calls GET /v1/anomalies.
    ◦ The backend queries Neon: SELECT * FROM anomalies WHERE user_id = 'user_123' ORDER BY timestamp DESC.
    ◦ Since Neon is serverless, it scales down when you aren't developing, costing $0.

Phase 5: Action & Evolution (Steps 9–10)

Goal: Proactive alerts and self-improving models.

- **Alerting Engine (The Notification Service):**
    - **Consumer:** Listens to the anomalies Kafka topic.
    - **Filter:** Checks user config (if severity == HIGH and plan == ADVANCED).
    - **Deduping:** Uses Redis keys (e.g., SETNX alert:user_123:last_10_min) to prevent spamming the user.
    - **Delivery:** Calls a free email provider API (like SendGrid free tier) or logs to console for dev.
- **Model Retraining (The Feedback Loop):**
    - **Trigger:** A weekly script.
    - **Data Source:** Pulls last week's data from Neon.
    - **Process:** Retrains the Isolation Forest model locally or in a cloud function.
    - **Deploy:** Saves the new .onnx model file to R2 storage. The running services pull the new model on restart.

III. System Schema (Core Data Structure)

This is the JSON object that flows through your Kafka pipeline.

JSON

```json
{
  "meta": {
    "trace_id": "abc-123-xyz",
    "timestamp": "2025-12-20T12:00:00Z",
    "source": "REALTIME_API",  // or "BATCH_CSV"
    "user_id": "user_99"
  },
  "data": {
    "tx_id": "tx_555",
    "amount": 4500.50,
    "currency": "USD",
    "location": "NY"
```

```
 },
 "enrichment": {
   "user_avg_spend": 120.00,
   "distance_from_last_tx": 5000  // km
 },
 "analysis": {
   "rule_flags": ["GEO_IMPOSSIBLE", "AMOUNT_SPIKE"],
   "ml_score": 0.98,
   "ml_prediction": "ANOMALY"
 },
 "verdict": {
   "final_severity": "CRITICAL",
   "explanation": "Transaction amount is 37x higher than average. Location (NY) is physically impossible relative to previous transaction (London, 10 mins ago)."
 }
}
```

IV. Summary of the Free Cloud Stack

- **Identity:** Clerk (Free tier)
- **Ingestion/Queue:** Confluent Cloud (Kafka) (Free credit)
- **Cache/State:** Upstash Redis (Free tier)
- **Storage/DB:** Neon Postgres (Free tier) & Cloudflare R2 (Free tier)
- **Compute:** Your Laptop (Development) -> Render/Railway (Deployment - Free/Cheap tiers)