**{**

**"snapshot": {**

**"snapshot\_id": "string", // Unique identifier for the snapshot (e.g., "snapshot\_20241224\_0800")**

**"timestamp": "ISO8601\_timestamp", // When the snapshot was captured (e.g., "2024-12-24T08:00:00Z")**

**"model": {**

**"model\_id": "string", // Unique ID for the process model (e.g., "model\_001")**

**"model\_name": "string", // Name of the model (e.g., "Order Fulfillment Process")**

**"model\_type": "string", // Type of the process model (e.g., "Petri Net", "BPMN")**

**"model\_definition": "string", // Serialized representation (e.g., JSON or XML)**

**"model\_metrics": {**

**"fitness": "float", // Measures how well the model fits observed behavior (0 to 1)**

**"soundness": "float", // Indicates logical consistency & absence of deadlocks in the model**

**"precision": "float", // Indicates how restrictive or overfitting the model is**

**"generalization": "float" // Indicates how well the model generalizes to new, unseen behavior**

**}**

**},**

**"variants": [**

**{**

**"variant\_id": "string", // Unique identifier for this variant (e.g., "variant\_001")**

**"variant\_name": "string", // Human-readable name/label (e.g., "Standard Path")**

**"activity\_sequence": [**

**"string" // Ordered list of activities (e.g., ["Start", "Approval", "End"])**

**],**

**"variant\_metrics": {**

**"frequency": "integer", // Number of times this variant occurred in the snapshot**

**"average\_duration": "float", // Avg. duration of this variant in seconds (snapshot-based)**

**"median\_duration": "float", // Median duration in seconds**

**"min\_duration": "float", // Minimum observed duration in seconds**

**"max\_duration": "float", // Maximum observed duration in seconds**

**"confirmation\_level": "float", // Confidence or verification level for the variant (0–1, or custom)**

**// Optional global-like metrics consolidated here**

**"total\_frequency": "integer", // Total frequency of this variant across all relevant snapshots**

**"global\_average\_duration": "float", // Global average duration across snapshots**

**"anomalies\_count": "integer" // Count of all anomalies for this variant across snapshots**

**},**

**"journeys": [**

**{**

**"journey\_id": "string", // Unique ID for the journey (e.g., "journey\_001")**

**"start\_time": "ISO8601\_timestamp",// Journey start time (e.g., "2024-12-24T07:00:00Z")**

**"end\_time": "ISO8601\_timestamp", // Journey end time (e.g., "2024-12-24T07:30:00Z")**

**"status": "string", // Current status of the journey ("Completed", "Failed", etc.)**

**"event\_sequence": [**

**"string" // Ordered list of event IDs (e.g., ["E1", "E2", "E3"])**

**],**

**"journey\_metrics": {**

**"duration": "float", // Total duration of the journey in seconds**

**"bottlenecks": [**

**"string" // Activities or steps causing delays**

**],**

**"fitness\_score": "float", // How well this journey fits the model (0–1)**

**"anomalies": [**

**{**

**"anomaly\_id": "string", // Unique identifier for the anomaly**

**"type": "string", // Type of anomaly (e.g., "Missing Activity", "Deviation")**

**"details": "string" // Description of what caused the anomaly**

**}**

**]**

**}**

**}**

**]**

**}**

**],**

**"snapshot\_metrics": {**

**"total\_variants": "integer", // Count of distinct variants in this snapshot**

**"total\_journeys": "integer", // Count of all journeys in this snapshot**

**"average\_duration": "float", // Average duration across all journeys in the snapshot**

**"median\_duration": "float", // Median duration across all journeys in the snapshot**

**"anomalies\_count": "integer", // Total anomalies detected in this snapshot**

**"average\_fitness\_score": "float" // Average fitness score across all journeys**

**}**

**}**

**}**

This **Snapshot JSON** structure is designed to capture a **point-in-time** view of your process data, including:

* **Model Details:** A summary of the process model and its metrics.
* **Variants and Journeys:** How different process variants are performing, with per-journey details.
* **Metrics:** Aggregated and time-specific insights for quick analysis.

{

"snapshot": {

"snapshot\_id": "string",

"timestamp": "ISO8601\_timestamp",

"model": {

"model\_id": "string",

"model\_name": "string",

"model\_type": "string",

"model\_definition": "string",

"model\_metrics": {

"fitness": "float",

"soundness": "float",

"precision": "float",

"generalization": "float"

}

},

"variants": [

{

"variant\_id": "string",

"variant\_name": "string",

"activity\_sequence": [

"string"

],

"variant\_metrics": {

"frequency": "integer",

"average\_duration": "float",

"median\_duration": "float",

"min\_duration": "float",

"max\_duration": "float",

"confirmation\_level": "float",

"total\_frequency": "integer",

"global\_average\_duration": "float",

"anomalies\_count": "integer"

},

"journeys": [

{

"journey\_id": "string",

"start\_time": "ISO8601\_timestamp",

"end\_time": "ISO8601\_timestamp",

"status": "string",

"event\_sequence": [

"string"

],

"journey\_metrics": {

"duration": "float",

"bottlenecks": [

"string"

],

"fitness\_score": "float",

"anomalies": [

{

"anomaly\_id": "string",

"type": "string",

"details": "string"

}

]

}

}

]

}

],

"snapshot\_metrics": {

"total\_variants": "integer",

"total\_journeys": "integer",

"average\_duration": "float",

"median\_duration": "float",

"anomalies\_count": "integer",

"average\_fitness\_score": "float"

}

}

}

**1. snapshot**

* **snapshot\_id (string):** Unique identifier for this snapshot (e.g., "snapshot\_20250101\_0800").
* **timestamp (ISO8601\_timestamp):** Exact time the snapshot was recorded (e.g., "2025-01-01T08:00:00Z").

**1.1. model**

* **model\_id (string):** Unique identifier for the process model (e.g., "model\_001").
* **model\_name (string):** Descriptive name for the model (e.g., "Order Fulfillment Process").
* **model\_type (string):** Type of the model (e.g., "BPMN", "Petri Net").
* **model\_definition (string):** A serialized representation of the model (e.g., JSON or XML).
* **model\_metrics (object):** Numerical indicators of model quality.
  + **fitness (float):** How well the model accounts for observed events (0–1).
  + **soundness (float):** Whether the model is logically consistent (0–1).
  + **precision (float):** Whether the model is too broad or restrictive (0–1).
  + **generalization (float):** Whether the model can handle new behaviors (0–1).

**1.2. variants (array)**

Each element represents a **distinct path** through the process during this snapshot.

* **variant\_id (string):** Unique identifier for the variant (e.g., "variant\_001").
* **variant\_name (string):** Human-readable label (e.g., "Standard Approval Path").
* **activity\_sequence (array of strings):** Ordered list of activities (e.g., ["Start", "Review", "End"]).

**1.2.1. variant\_metrics (object)**

* **frequency (integer):** Occurrences of this variant in the current snapshot.
* **average\_duration (float):** Average completion time (seconds) for this variant in the snapshot.
* **median\_duration (float):** Median completion time (seconds).
* **min\_duration (float):** Shortest observed completion time (seconds).
* **max\_duration (float):** Longest observed completion time (seconds).
* **confirmation\_level (float):** Numerical confidence indicator (0–1) for this variant.
* **total\_frequency (integer):** Overall occurrences across relevant snapshots.
* **global\_average\_duration (float):** Overall average completion time across snapshots.
* **anomalies\_count (integer):** Count of anomalies tied to this variant across snapshots.

**1.2.2. journeys (array)**

Each element details a single occurrence (case) that followed this variant.

* **journey\_id (string):** Unique ID for the journey (e.g., "journey\_123").
* **start\_time (ISO8601\_timestamp):** When this journey started.
* **end\_time (ISO8601\_timestamp):** When this journey ended.
* **status (string):** The journey’s state (e.g., "Completed", "Failed", "In Progress").
* **event\_sequence (array of strings):** Events in the order they occurred.

**1.2.2.1. journey\_metrics (object)**

* **duration (float):** Total time (seconds) the journey took.
* **bottlenecks (array of strings):** Activities or steps that caused delays.
* **fitness\_score (float):** Score showing how closely this journey followed the model (0–1).
* **anomalies (array):** List of unusual occurrences in this journey.
  + **anomaly\_id (string):** Unique ID for the anomaly.
  + **type (string):** Category or label (e.g., "Missing Activity", "Deviation").
  + **details (string):** Description of the anomaly.

**1.3. snapshot\_metrics (object)**

Aggregated statistics for the entire snapshot.

* **total\_variants (integer):** Number of variants detected in this snapshot.
* **total\_journeys (integer):** Number of journeys observed in this snapshot.
* **average\_duration (float):** Overall average time (seconds) for journeys in this snapshot.
* **median\_duration (float):** Overall median time (seconds) for journeys.
* **anomalies\_count (integer):** Total anomalies found in all journeys.
* **average\_fitness\_score (float):** Mean fitness score for all journeys in this snapshot.

Below is an extended solution architecture that integrates **custom variant management** into the Process Intelligence Platform. It covers how you can store user-defined variants in a dedicated database or reference store, incorporate them into the discovery/conformance workflows, and expose everything via a set of **APIs** (microservices or modular services) for both batch and real-time operation. This architecture aims to be highly scalable, flexible, and user-friendly.

**1. Key Objectives & Challenges**

1. **Scalability**: The bank (or credit card company) deals with millions of logs daily, so the solution must handle large volumes of events in both batch and real-time.
2. **Custom Variant Management**: Users must be able to define additional process variants that are not evident from the raw logs—ensuring “complete” or “future-proof” process models.
3. **Snapshot & Versioning**: Maintain versions of discovered models, reference data, and user-defined variants for tracking historical and current states.
4. **Real-Time Monitoring & Conformance**: Ability to detect deviations quickly and trigger alerts if a process or event exceeds a certain threshold.
5. **API-Driven Extensibility**: Provide a consistent set of APIs so that internal or external front-ends (or other systems) can consume, manage, and display process mining results, variants, and conformance checks.

**2. High-Level Architecture Components**

Here’s an overview of the major building blocks:

1. **Ingestion & Streaming Layer**
   * **Sources**: Applications, transaction systems, mainframes, CRM, etc.
   * **Transport**: Kafka / Kinesis / Event Hubs for real-time; Scheduled batch ingestion for data-lake style input (S3, ADLS, etc.).
2. **Data Processing & Normalization**
   * **Stream Processing**: Apache Flink / Spark Structured Streaming to parse and enrich events on the fly.
   * **Batch Processing**: Apache Spark or similar for large-scale transformations, cleaning, and normalization (case ID, event names, timestamps, etc.).
3. **Storage Layer**
   * **Data Lake** (e.g., S3, ADLS, HDFS) for raw & historical logs.
   * **Data Warehouse** (e.g., Snowflake, BigQuery, Redshift, Synapse) for aggregated analytics queries.
   * **Reference Store / Variant DB**: A dedicated (NoSQL or RDBMS) data store that houses:
     + Discovered process models (in BPMN or Petri Net form).
     + **User-defined custom variants** and their metadata.
     + Versioned references to activity definitions and naming conventions.
   * **Graph DB (Optional)** for complex relationship queries (Neo4j, JanusGraph) if advanced graph-based process queries are needed.
4. **Process Mining & Conformance Services**
   * **Discovery Service** (Inductive Miner, Fuzzy Miner, etc.) that reads from the data store and builds a base model.
   * **Variant Management Service** that merges user-supplied variants with discovered models.
   * **Conformance Checking Service** (alignment-based or token-replay) to measure fitness/precision and identify outlier traces.
5. **Scoring & Alerting Engine**
   * Accepts live streams of events, applies scoring rules or ML models (e.g., anomaly detection).
   * If a threshold is exceeded, triggers an event to the **Notification Service** (Slack, email, etc.).
6. **Batch Snapshot & Metrics Engine**
   * Scheduled job that consolidates the latest event data, runs conformance checks, and captures a snapshot of relevant metrics, top variants, performance metrics, etc.
7. **Real-Time Monitoring & Dashboard**
   * Continuously updates with new events.
   * Integrates partial conformance or streaming conformance checks to detect immediate deviations.
   * Provides a summary of active traces, pending tasks, or potential SLA violations.
8. **APIs & Front-End**
   * A set of microservices (or modular services) exposing REST/GraphQL APIs to allow business users (or external applications) to:
     + Upload custom variants.
     + Trigger a new discovery run.
     + Check conformance or run real-time monitoring queries.
     + Configure or retrieve snapshots, alerts, and metrics.

**3. Detailed Workflow with Custom Variants**

1. **Data Ingestion**
   * Multiple sources push event logs to the streaming/batch layer.
   * Events are normalized, cleaned, and enriched with reference data (e.g., organization IDs, product codes).
2. **Process Discovery**
   * **Discovery Service** generates a baseline process model from the logs, identifying the most common sequences of activities, and building an initial BPMN/Petri Net.
3. **User-Defined Custom Variants**
   * **User Interaction**: A business user (e.g., Process SME) notices that the discovered model is missing a known branch or special-case path.
   * **Variant Definition**: They log in to a front-end app and define (or upload) a custom variant—essentially specifying a sequence of activities not found in the raw logs.
   * **Storing Variants**: This custom variant is stored in the **Variant DB** with a reference ID and optional metadata (e.g., “High-risk exception path,” “Future process improvement path,” etc.).
   * **Variant Merge/Re-Discovery**: The Discovery Service (or a specialized **Variant Management Service**) merges the custom variant(s) with the baseline discovered model, producing a “comprehensive” version of the process model that includes both discovered paths and user-supplied paths.
4. **Conformance Checking**
   * The **Conformance Checking Service** uses the final integrated model to evaluate new or existing event logs.
   * Aligns each trace against the model to determine where (and how) it deviates.
   * Records conformance metrics, outlier traces, and potential process exceptions in the Data Warehouse or a specialized store.
5. **Scoring & Alerting**
   * The **Scoring Engine** can attach severity or risk scores to events that follow (or deviate from) either the discovered or user-defined paths.
   * Real-time triggers (via Kafka topics, for example) generate alerts if a critical threshold is breached.
6. **Batch Snapshot & Trends**
   * On a scheduled basis (e.g., every 3 hours or daily), a batch job runs a full conformance check on the last N hours of logs, merges relevant user-defined variants, and generates a snapshot of:
     + **Model coverage**: How many traces matched the discovered or custom variant paths?
     + **Violations**: How many were incomplete or had skipped steps?
     + **Performance metrics**: Throughput time, wait time, bottlenecks, etc.
   * Stores these snapshots for historical trending and auditing.
7. **Real-Time Dashboard**
   * A streaming pipeline updates the dashboard with current throughput, key KPIs, and any anomalies in near real-time.
   * Users can filter by specific variants to see if any custom or future path is being executed in the live system.

**4. Reference Solution Architecture Diagram**

A simplified conceptual diagram might look like this (text-based):

+---------------------+ +----------------------+

| Source Systems | | User / SME |

| (Core Banking, ... )| | (Front-end UI) |

+---------+-----------+ +----------+-----------+

| |

(Streaming/Bulk) |

v | Add/Define

+---------------------+ +-------------v-----------------+

| Ingestion Layer | | Variant Management Service |

| (Kafka / S3 / ...) | | (API to store/retrieve user- |

+---------+-----------+ | defined variants, metadata) |

| +-------------+-----------------+

v |

+---------------------+ Merge & Re-Discovery |

| Data Processing | <----------------------------+

| (Spark / Flink) |

+---------+-----------+

v

+---------------------+ +--------------------------------+

| Storage Layer | <--------------> | Process Discovery Service |

| - Data Lake (Raw) | (reads/writes) | Conformance Checking Service |

| - DWH (Aggregates) | ---------------- | Scoring & Alerting Service |

| - Variant DB | | (APIs for process mining logic)|

+---------------------+ +---------------+-----------------+

|

| Alerts/Results

v

+--------------------------------+

| Notification / Monitoring |

| (e.g., Slack, Email, SMS) |

+--------------------------------+

**Key points**:

* **Variant DB** (part of Storage Layer) holds user-defined process paths and metadata.
* **Process Discovery & Conformance Services** can incorporate user-defined paths into the final, “canonical” model.
* **APIs** used by front-end and other systems to manage variants, trigger discovery, run conformance checks, and retrieve results.

**5. API Strategy**

**5.1 Potential Microservices & Endpoints**

1. **Variant Management Service**
   * POST /variants — Create or upload a new user-defined variant (JSON or BPMN/Petri-net format).
   * GET /variants/{id} — Retrieve details of a custom variant.
   * PUT /variants/{id} — Update an existing variant.
   * GET /variants — List all variants; filter by process, date, etc.
2. **Process Discovery Service**
   * POST /discovery/run — Trigger a discovery job on selected event data.
   * GET /discovery/results/{jobId} — Fetch discovered model(s) and statistics.
   * POST /discovery/merge — Merge discovered model with a set of custom variants (returns new version ID).
3. **Conformance Checking Service**
   * POST /conformance/run — Run a conformance check on a selected model (discovered + custom variants) against a dataset/time range.
   * GET /conformance/results/{jobId} — Retrieve conformance metrics, alignment logs, outlier traces, etc.
4. **Scoring & Alerting Service**
   * POST /scoring/apply — Apply scoring rules or an ML model to a batch of events.
   * GET /alerts — Get active/past alerts.
   * POST /alerts/subscribe — Register for certain triggers (e.g., if event score > X, send Slack message).
5. **Snapshot Service**
   * POST /snapshot/create — Force creation of a snapshot for a process model, capturing key metrics.
   * GET /snapshot/{id} — Retrieve a historical snapshot (metrics, conformance results, variant coverage, etc.).
6. **Real-Time Monitoring Service**
   * GET /monitor/live — Provides a streaming or polling API to see ongoing trace statuses, partial conformance, or anomalies.

**5.2 Deployment & Access Control**

* Deploy these services in a containerized environment (Kubernetes, Docker Swarm, ECS, etc.) behind an **API Gateway** (e.g., Kong, Apigee, Amazon API Gateway, etc.).
* Implement **OAuth 2.0 / OpenID Connect** or a similar mechanism for secure, token-based access to each API.
* **Role-Based Access Control (RBAC)** to restrict which users can define new variants vs. who can only read results.

**6. Scalability & Performance Considerations**

1. **Distributed Processing**: Ensure ingestion and data transformations leverage horizontally scalable platforms like Spark or Flink.
2. **High-Performance Storage**: Use partitioned tables in your Data Warehouse to handle billions of rows.
3. **Variant DB Structure**: Store custom variants in a structured format (JSON, BPMN XML, etc.) plus associated metadata. Index them for fast queries.
4. **Caching**: Cache partial process models or sub-graphs to speed up real-time conformance checks.
5. **Load Balancing**: For microservices, scale out replicas behind a load balancer to handle spikes in API calls.

**7. Putting It All Together**

1. **Data arrives** (real-time or batch).
2. **Discovery** jobs run periodically or on-demand, generating baseline models.
3. **Users define additional variants** that are stored in a **Variant DB** with versioning.
4. **Merging** custom variants + discovered model ensures coverage for “future processes” or special cases.
5. **Conformance checks** are run in both real-time (for immediate anomalies) and batch (for comprehensive analysis).
6. **Alerts** are triggered when deviations or critical event scores pass thresholds.
7. **Snapshots** are saved regularly for historical trend analysis.
8. **APIs** provide seamless integration with front-end UIs and other systems in the enterprise, enabling user-friendly exploration, variant management, analytics dashboards, and more.

**8. Summary**

This extended architecture allows for:

* **Full-lifecycle process intelligence**: From ingestion and discovery to conformance, alerting, and reporting.
* **User-Defined Variants**: A dedicated store and microservice for capturing and merging user-specified process paths with discovered models.
* **Scalable & Modular**: Each service (discovery, conformance, alerting, variant management) can be scaled independently.
* **API-Driven**: A clearly defined set of APIs and services, deployable in a modern container/orchestrated environment, ensuring extensibility and maintainability.

By following this strategy, you’ll build a robust **Process Intelligence Platform** that not only handles massive event volumes but also accommodates business users’ needs to refine the discovered models with custom variants—leading to more accurate, actionable, and future-proof process insights.

Below is a high-level solution architecture proposal for a large-scale Process Intelligence Platform. The goal is to support process discovery, conformance checking, variant management, performance analysis, real-time monitoring, and event scoring/alerting. This architecture is based on best practices in process mining, combined with modern data pipelines and advanced analytics/monitoring frameworks.

**1. Core Objectives**

1. **Large-scale ingest of event logs** (millions of logs, multiple systems).
2. **Process discovery** from heterogeneous logs.
3. **Conformance checking** and variant analysis (detect outliers, measure fitness).
4. **Performance analysis** (throughput times, bottlenecks, etc.).
5. **Real-time monitoring** and alerting (via event scoring and thresholds).
6. **User-driven model refinement** (allow business users to merge, filter, or define their own variants).
7. **Scalable batch processing** (daily or every few hours) for snapshot-based metrics.

**2. Conceptual Flow**

Below is a conceptual flow of data and functionalities:

1. **Event Ingestion**
   * Pull event logs from multiple applications (e.g., credit card processes, customer onboarding, loan origination, etc.).
   * Can be near real-time (streaming) or batch files depending on source systems.
2. **Data Pre-Processing & Normalization**
   * Transform raw logs into a standardized schema for process mining.
   * Map each log line to a case (Trace ID) and activity (Event Name), plus timestamps and attributes.
3. **(Optional) Data Storage (Lake + Warehouse)**
   * Long-term storage of raw event logs in a Data Lake for historical analysis.
   * Aggregated or structured data in a Data Warehouse for reporting and quick analytics.
4. **Process Discovery**
   * Mine process models from the event logs (e.g., Petri nets, BPMN, or other notations).
   * Identify variants (commonly used vs. rare) and allow filtering/merging.
   * Potentially apply advanced algorithms (e.g., Inductive Miner, Fuzzy Miner) to handle noise.
5. **Conformance Checking**
   * Compare discovered models against event logs to measure fitness, precision, and detect outliers.
   * Flag traces or events that deviate from the “expected” path.
6. **Scoring & Alerting**
   * Apply a scoring mechanism for events (e.g., a business rule engine or ML-based approach) to identify critical events or suspicious behavior.
   * If score crosses a threshold, trigger notifications to relevant stakeholders.
7. **Batch Snapshots & Analytics**
   * Periodically (daily or every N hours), run conformance checks, performance analytics, and produce a “snapshot” of current state (variants, throughput, outliers, etc.).
   * Store snapshots for trend analysis over time.
8. **Real-Time Monitoring**
   * Continuously feed the streaming events into a real-time monitoring engine to detect anomalies, out-of-bounds performance, or critical events.
   * Provide a live dashboard with alerts.
9. **User Interface & Exploratory Tools**
   * Allows business users to:
     1. Explore discovered process models and variants.
     2. Filter noise/irrelevant activities.
     3. Merge or define new variants.
     4. Drill down on conformance violations, suspicious traces, or performance bottlenecks.
   * End-users can define threshold-based alerts or advanced rules for scoring.

**3. High-Level Architecture Components**

Below is a layered view of the architecture.

**3.1 Data Ingestion Layer**

* **Sources**:
  + Core Banking Systems (e.g., credit card transaction systems, lending systems, customer onboarding).
  + Any application logs (e.g., mainframe logs, microservices logs, CRM logs).
* **Transport**:
  + **Streaming**: Apache Kafka / AWS Kinesis / Azure Event Hubs for real-time ingest.
  + **Batch**: File-based ingest (CSV, JSON, etc.) into a distributed file system (e.g., HDFS or cloud storage like S3).

**3.2 Data Processing / ETL Layer**

* **Stream Processing**:
  + **Apache Flink** or **Apache Spark Structured Streaming** to process incoming logs in near real-time, parse/transform event attributes, and compute intermediate metrics (e.g., real-time KPIs).
* **Batch Processing**:
  + **Apache Spark** or similar big-data frameworks to handle large historical datasets, run deduplication, data cleaning, attribute normalization.
* **Enrichment**:
  + Join with reference data (customer or product master data) if needed.
  + Attach metadata (e.g., organizational roles, region, product type).

**3.3 Data Storage Layer**

* **Data Lake**:
  + Store raw event logs for historical analysis. (e.g., HDFS, Amazon S3, Azure Data Lake Storage)
* **Data Warehouse**:
  + For aggregated data, schema-based queries, and reporting (e.g., Snowflake, Amazon Redshift, Azure Synapse, BigQuery, or traditional on-prem).
* **Process Mining Database**:
  + Could be a specialized store or columnar database to hold case/event tables.
  + Some teams also use a **Graph Database** (like Neo4j) if the discovered models or relationships are best represented as a graph.

**3.4 Process Discovery & Analytics Engine**

* **Process Mining Algorithms** (e.g., ProM libraries, Apromore, commercial tools, or custom):
  + **Discovery**: Inductive Miner, Fuzzy Miner, etc.
  + **Conformance Checking**: Token-based replay or alignment-based conformance.
  + **Variant Analysis**: Identifying dominant vs. rare paths.
  + **Performance Analytics**: Activity durations, waiting times, throughput times, bottlenecks.
* **Scoring & Alerting Module**:
  + Assign scores to events in real-time or in batch.
  + Could be rule-based (e.g., certain activity or combination triggers high severity) or ML-driven (e.g., anomaly detection).
  + Threshold-based notifications or advanced dynamic thresholds.

**3.5 Real-Time Monitoring & Alerting**

* **Stream Processing** (again) or specialized real-time frameworks:
  + Continuously ingest new events, map them to ongoing cases/traces.
  + Perform “online conformance” or partial replay to detect if a sequence is deviating.
  + Maintain live metrics (e.g., time since last step, steps out of order, etc.).
* **Notification Mechanisms**:
  + Publish alerts to Slack, email, or other channels.
  + Integrate with existing monitoring/alerting solutions (e.g., Splunk, Datadog, Grafana).

**3.6 User Interface / Portal**

* **Process Explorer**:
  + Visualization of the discovered process map or BPMN diagram, color-coded by frequency or performance.
  + Show conformance or performance statistics, along with variant navigation.
* **Variant Management UI**:
  + Allow business users to “merge” or “exclude” certain variants and re-run discovery.
  + Provide a versioning system for different model snapshots.
* **Analytics & Reporting Dashboards**:
  + Aggregate metrics (on-time completion rates, SLA violations, conformance scores).
  + Historical trends (e.g., daily snapshots of conformance).
  + KPI-based dashboards (e.g., cycle time distribution, work-in-progress, throughput).
* **Alert Configuration**:
  + Business users define thresholds or patterns that trigger an alert (e.g., “if any event in a trace has Score > 80, send notification to risk team”).

**4. Key Architectural Concerns**

1. **Scalability & Performance**
   * Event volumes can be huge (millions/billions). Use distributed processing for ingestion (Kafka), transformation (Spark/Flink), and storage (Data Lake / Data Warehouse).
   * For real-time monitoring, ensure you have a robust streaming cluster that can handle peak loads.
2. **Data Quality & Standardization**
   * One of the biggest challenges in process mining is having consistent identifiers for cases, activities, and timestamps. Standardize these definitions across various source systems.
   * Use reference data (e.g., product codes, organizational hierarchies) for richer context.
3. **Flexibility in Process Discovery**
   * Different processes might require different discovery algorithms or parameter tuning. Provide a configurable approach.
   * Keep an option to incorporate user feedback (filtering out noise, merging variants).
4. **Conformance & Variant Explosion**
   * If the process is highly variable, you might have “variant explosion.” Offer filtering rules or thresholds that help keep the discovered process comprehensible.
   * Possibly store only the top X% (by frequency) variants or prune out negligible frequencies.
5. **Security & Compliance**
   * Financial data is highly sensitive. Ensure encryption at rest and in transit, role-based access control, and compliance with regulations (PCI-DSS, GDPR, etc.).
   * Mask or tokenize PII where needed.
6. **Cost Management**
   * With large volumes of data, plan carefully for your infrastructure (e.g., cloud-based pay-as-you-go vs. on-prem clusters).
   * Consider how often you truly need real-time vs. near real-time or daily batch.
7. **Extensibility**
   * Ability to plug in advanced analytics or ML models later (e.g., anomaly detection, predictive process monitoring).
   * Provide APIs or microservices for consumption by other tools (e.g., risk scoring engines, or next-best-offer in credit card workflows).

**5. Putting It All Together: Sample End-to-End**

1. **Data Ingestion**
   * Event logs from credit card application come via Kafka in real-time. Another set of logs from a legacy system is loaded daily as a batch CSV.
   * Both land in your Data Lake, with structured transformations stored in your Data Warehouse for fast queries.
2. **Process Discovery**
   * In an iterative or scheduled manner, the system picks up new data from the Warehouse.
   * Runs process discovery to generate or update the “As-Is” BPMN/Petri-net model.
   * Summarizes top variants and populates them in the UI.
3. **Variant Management & Conformance Checking**
   * Business analysts go into the UI, see that some variants correspond to known exceptions. They choose to “merge” or “remove” them.
   * A final “canonical” model is established for conformance checking.
   * The system runs alignment-based conformance, generates “fitness” metrics, and flags outliers.
4. **Scoring & Alerting**
   * The system uses a rule-based engine that assigns high severity scores to certain critical events (e.g., large transaction with unusual pattern).
   * If real-time stream detects an event with a high severity score, it immediately triggers an alert to the operational team.
5. **Batch Snapshots**
   * Every 3 hours (or daily), a batch job runs a full conformance check on all new event logs within that window.
   * A snapshot of metrics (throughput, conformance, bottlenecks) is stored, so you can see trends over time.
6. **Real-Time Monitoring**
   * A stream analytics layer references the discovered models to do partial “online conformance” checks.
   * If a trace is skipping critical steps or taking too long in a particular stage, an alert is raised in near real-time.
7. **Continuous Improvement**
   * As new processes are onboarded, more logs are ingested.
   * The same pipeline (or slightly modified configuration) is used to discover new models, check conformance, and feed insights back to business owners.

**6. Summary**

By combining a robust data ingestion layer (stream + batch), a scalable storage solution (Data Lake + Warehouse), process discovery and conformance checking engines, a flexible UI for variant management, and real-time monitoring with alerting/scoring, you can build a comprehensive **Process Intelligence Platform**.

Key to success will be:

* **Strong data governance** and consistent event schemas.
* **Efficient conformance checking** to handle large volumes.
* **User-centric model refinement** to avoid “analysis paralysis” from variant overload.
* **Real-time analytics** balanced with the right cost and complexity.

This architecture ensures the bank’s processes (like credit card applications, transactions, or customer service flows) can be discovered, monitored, and improved systematically, ultimately delivering better operational visibility and compliance for critical financial workflows.

Below is a **high-level strategy** for implementing a **Process Mining API** (in Python + Flask, or similar frameworks) that focuses on **how** to structure your data, **what** APIs are needed, and **where** to store information (including caching). This strategy aims to help you design a scalable, maintainable platform without delving into specific code snippets.

**1. Key Concepts & Goals**

1. **Variants**
   * A variant is a sequence of activities (events) representing one “path” or “scenario” within a process.
   * You’ll have discovered variants (from event logs) and custom variants (defined by users for exceptions or future paths).
2. **Process Models**
   * A process model can be discovered by combining multiple variants or logs.
   * Each model is typically associated with metadata (creation date, algorithm used, domain, etc.).
3. **Snapshots**
   * A snapshot is a point-in-time report/analysis of conformance or performance metrics (e.g., how well the incoming logs match the discovered model).
4. **Caching**
   * Large sets of logs or models can be expensive to re-load or re-calculate.
   * In-memory caching (e.g., Redis) helps avoid repeated queries or computations.
5. **Databases**
   * **Primary DB** (Relational or NoSQL) for storing process metadata, variants, models, snapshots, etc.
   * **Raw Event Logs** might reside in a data lake or search engine (like Elasticsearch).
   * **Caches** (Redis or similar) to store ephemeral or partial results (e.g., partial discovered models).

**2. Data Architecture & Storage Strategy**

**2.1 Variant Storage**

* **Table / Collection: variants**
  + **Columns / Fields** (example):
    - variant\_id (primary key, e.g., UUID)
    - name (string, e.g., "Loan Approval Exception Path")
    - activities (array of strings or a separate link table if in RDBMS)
    - type (e.g., discovered or custom)
    - created\_by (user ID)
    - created\_at (timestamp)
    - metadata (JSON) for additional context
  + **Usage**:
    - Allows you to quickly look up variants by ID, name, or type.
    - Custom variants can be added by users and flagged differently than discovered ones.

**2.2 Process Model Storage**

* **Table / Collection: process\_models**
  + **Columns / Fields** (example):
    - model\_id (primary key)
    - name (e.g., "Credit Card App Model - v2")
    - variant\_ids (array/list referencing the IDs that went into this model)
    - model\_artifact (could be a string/JSON representation of the BPMN or Petri Net)
    - created\_at, updated\_at
    - algorithm\_used (e.g., Inductive Miner, Fuzzy Miner)
    - metadata (JSON) for domain-specific data
  + **Usage**:
    - Each discovered or merged model can be retrieved or updated (e.g., when a new custom variant is integrated).
    - The model\_artifact might be a large object—some teams store it in a separate file store (S3 or local disk) and keep just a reference path in the DB.

**2.3 Snapshot Storage**

* **Table / Collection: snapshots**
  + **Columns / Fields** (example):
    - snapshot\_id (primary key)
    - model\_id (foreign key to process\_models)
    - time\_range (start/end timestamps)
    - conformance\_metrics (JSON: e.g., fitness, precision, # of outliers)
    - performance\_metrics (JSON: e.g., average throughput time, bottlenecks)
    - created\_at
    - metadata (JSON)
  + **Usage**:
    - A snapshot is generated when a user or a scheduled job runs conformance/performance analysis on a selected model and data subset.
    - Snapshots allow historical comparison (e.g., day-over-day, week-over-week).

**2.4 Caching Layer**

* **Redis** or **In-Memory Cache**
  + **Keys**:
    - variant:{variant\_id}: Partial or full data of a variant.
    - model:{model\_id}: In-memory or partial representation of the discovered process model.
    - Temporary data used during discovery or conformance checking.
  + **Eviction**:
    - Time-based (e.g., TTL of a few hours or days), or
    - Manual eviction on model updates (e.g., if a model changes, remove old cache keys).

**3. Required APIs & Flows (High-Level)**

Below is a summary of the main endpoints you would likely expose. These can be grouped into **blueprints** (or routes) if using Flask.

**3.1 Variant Management**

1. **GET /variants?filters=...**
   * Returns a filtered list of variants (by date, type, name substring, etc.).
   * You may retrieve from DB or index, optionally caching results for quick repeated queries.
2. **POST /variants**
   * Creates a new **custom** variant.
   * Request payload might include a name, an ordered list of activities, and optional metadata.
   * Store in the DB (variants table/collection) and optionally set a cache entry.
3. **GET /variants/{variant\_id}**
   * Returns the details of a specific variant.
   * Check cache first; if not found, fetch from DB, then cache for future.

**3.2 Process Discovery & Model Management**

1. **POST /discovery/run**
   * Inputs: A list of variant\_ids, plus optional parameters (noise threshold, etc.).
   * Or it might take a query that references raw logs in your event store.
   * **Process**:
     + Fetch needed variant data from DB (and/or logs from Elasticsearch/S3).
     + Apply the discovery algorithm to build a model.
     + Save the resulting model in process\_models and store the model\_artifact.
   * Return: model\_id.
2. **POST /discovery/re-discover**
   * Inputs: model\_id + new variant\_id to incorporate.
   * Logic: Load existing model, fetch the new variant from DB, merge or re-run a partial discovery.
   * Save updated/merged model as either a new model\_id or an updated record.
   * Return: updated\_model\_id.
3. **GET /models/{model\_id}**
   * Fetch an existing model from DB (or cache) to inspect.
   * Could return metadata or the entire BPMN/Petri Net artifact.

**3.3 Snapshot & Conformance**

1. **POST /snapshots**
   * Inputs: model\_id, time\_range, plus filters for which logs to check.
   * Logic:
     + Retrieve the process model.
     + Query the relevant event logs (in Elasticsearch or a data lake) within time\_range.
     + Run conformance checking and performance analysis.
     + Generate metrics, store them as a new record in snapshots.
   * Return: snapshot\_id.
2. **GET /snapshots/{snapshot\_id}**
   * Return the stored snapshot metrics (conformance, performance, etc.).

**3.4 Conformance Checking / On-Demand Analysis (Optional)**

* **POST /conformance/check**
  + Could be an on-demand endpoint for partial conformance checks (e.g., a single trace or real-time scenario).
  + You might not always store the results as a snapshot—this is more ephemeral.

**4. Caching & Performance Strategy**

1. **When to Cache**
   * Caching is ideal for repeated queries: e.g., the same variants are requested frequently, or the same model is loaded multiple times for conformance checks.
2. **What to Cache**
   * **Variant definitions**: Especially if they are large or numerous.
   * **Discovered models**: The “compiled” BPMN/Petri Net object that the conformance checker uses.
   * **Partial conformance results** (if you do incremental checks).
3. **Cache Expiry / Invalidation**
   * Set a TTL (Time-To-Live), or
   * Invalidate on updates: If a variant changes or a new variant merges into a model, delete relevant cache entries so you don’t serve stale data.
4. **Implementation Approaches**
   * **Redis**: Reliable for production.
   * **In-memory**: Simple, but less scalable across multiple application servers.

**5. Putting It All Together: Process Flow**

Here’s how these components might be orchestrated in a real scenario:

1. **User wants to add a custom variant**
   * They call **POST /variants** with the activities list and metadata.
   * The system saves it in the DB, returns a variant\_id, and writes a minimal entry in the cache.
2. **User or system triggers process discovery**
   * They call **POST /discovery/run** with relevant variant\_ids.
   * The system pulls variants from DB/cache, fetches additional logs if needed, runs the discovery algorithm, and saves a new record in process\_models.
   * A unique model\_id is returned.
3. **User merges a custom variant into an existing model**
   * **POST /discovery/re-discover** with the model\_id and the new variant\_id.
   * The system loads the existing model from DB (or cache), merges or re-runs discovery logic, and saves a new or updated model record, returning an updated\_model\_id.
4. **Conformance check & snapshot**
   * **POST /snapshots** with model\_id, time range, and possibly filtering parameters.
   * The system retrieves the model, queries logs in that date range, runs conformance checks, and calculates metrics.
   * Results are saved in snapshots. The user gets a snapshot\_id back to retrieve or reference later.
5. **Subsequent queries**
   * **GET /variants/{variant\_id}**: The system checks the cache for the variant. If it’s not there, it fetches from DB.
   * **GET /models/{model\_id}**: Similar approach.
   * **GET /snapshots/{snapshot\_id}**: Returns the stored metrics without re-running the analysis.

**6. Database & Lookup Strategy**

1. **Primary Relational Database** (PostgreSQL, MySQL, or similar):
   * Ideal if you want structured relationships (model–variants, snapshot–model references).
   * You can define foreign keys to maintain referential integrity (e.g., snapshots.model\_id references process\_models.model\_id).
2. **NoSQL Store** (MongoDB, Couchbase, etc.):
   * Could be used if your data is mostly document-oriented (variants and models stored as JSON objects) or if you need flexible schemas.
3. **Lookup Tables**
   * If you want a separate “lookup” for variant names or processes, you could maintain a simple table:
     + process\_lookup with process\_id, process\_name, etc.
     + Each variant references a process\_id to indicate the high-level process it belongs to.
4. **File Store vs. DB**
   * Storing large BPMN/Petri Net files or complex model artifacts directly in the DB might be cumbersome.
   * Consider saving them in a file store (e.g., AWS S3, on-prem storage) and storing only a reference in process\_models.

**7. Operational Considerations**

1. **Scaling the Application**
   * If using Flask, deploy behind a WSGI server (Gunicorn, uWSGI) with multiple workers.
   * Use a load balancer to handle high volumes of requests.
2. **Background / Async Processing**
   * Long-running discovery or conformance checks could be offloaded to a task queue (Celery, RQ).
   * The API endpoint could return a job ID immediately, and a separate endpoint (like GET /jobs/{job\_id}) gives the status of the job.
3. **Security & Access Control**
   * Typically, implement JWT or OAuth for your APIs.
   * Restrict who can create custom variants vs. who can only view or run snapshots.
4. **Logging & Auditing**
   * Log every creation or update of variants, models, and snapshots for traceability—crucial in regulated environments (like banking).
5. **Monitoring & Alerts**
   * Monitor the API usage, cache hits/misses, and DB performance with tools like Prometheus, Grafana, or Datadog.
   * This ensures you can quickly identify bottlenecks or scale appropriately.

**8. Summary**

This **strategic approach** for your Process Mining platform focuses on how to organize and store variants, process models, and snapshots, as well as the minimal but essential set of APIs to manage them. The **key ideas** include:

1. **Clear separation** of concepts:
   * **Variants**: sequences of activities (could be discovered or user-defined).
   * **Models**: discovered or merged from variants (stored as references and artifacts).
   * **Snapshots**: periodic conformance/performance outputs tied to a model.
2. **Flexible & scalable** data storage:
   * A **relational DB** or **NoSQL** for structured relationships and quick lookups.
   * A **file store** or object storage for large process artifacts.
   * A **cache** (Redis) to handle repeated lookups of heavy data (models, variants).
3. **API layering**:
   * **Variant** endpoints to fetch/create custom variants.
   * **Discovery/Model** endpoints to run or update the discovered process.
   * **Snapshot** endpoints to store conformance analysis over specific periods or logs.
4. **Caching strategy** to reduce redundant computations and accelerate frequent queries.

With this foundation, you can implement the **actual** business logic (discovery, conformance checking, performance metrics) in modular services, then expose them through well-structured endpoints. Over time, you can extend this strategy to incorporate real-time event streams, predictive process monitoring, or advanced analytics without major re-architecture.