Data Ingestion Pipeline

Requirement Summary:

Design a data ingestion pipeline to process batches of user and transaction data into a third-party CRM platform. Data will be provided in batches of files on an SFTP server. The pipeline must handle high data volumes, ensure high availability, manage errors effectively, and provide comprehensive reporting.

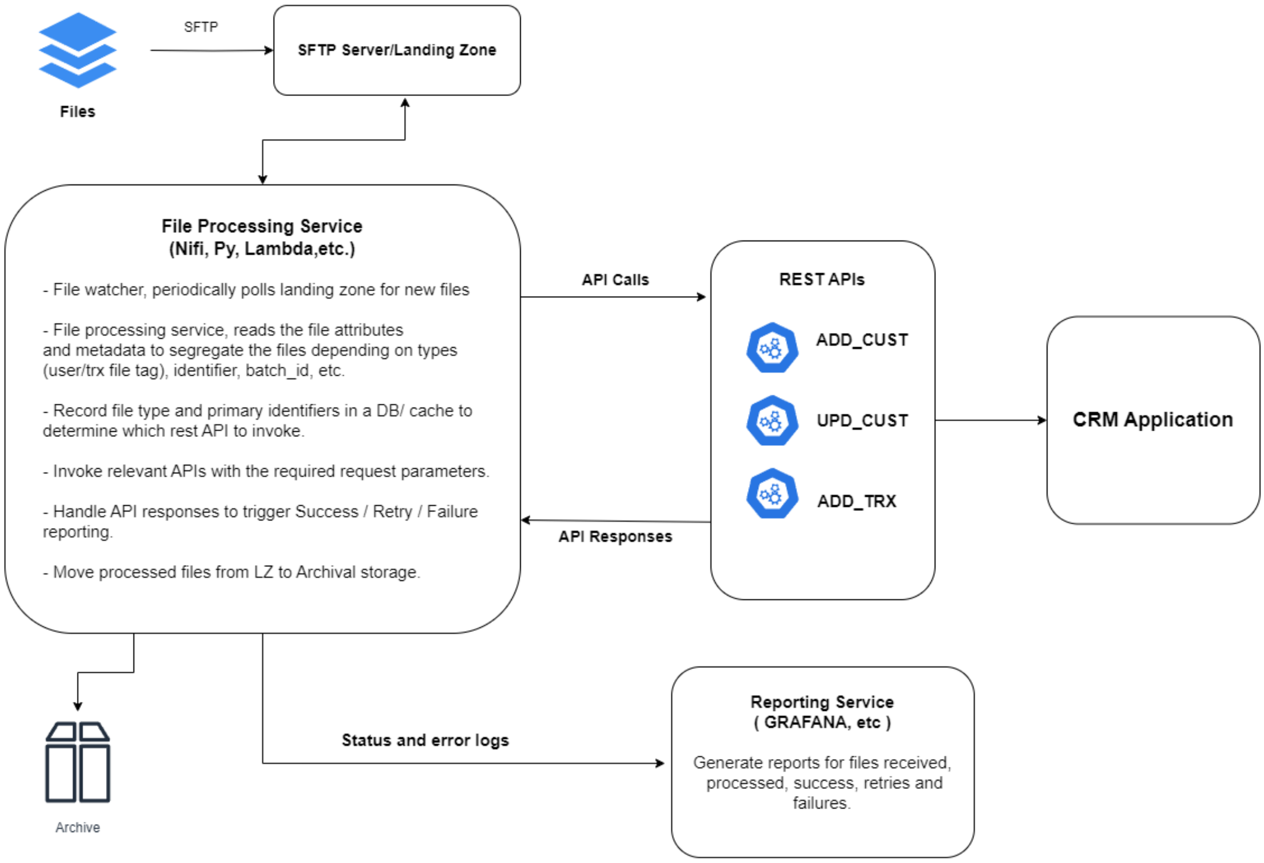
Functional requirements:

1. **Data Ingestion Layer:**
    - Landing Zone:
        - SFTP server (OpenSSH SFTP, AWS Transfer for SFTP, Azure SFTP etc)
    - Configuration Options:
        - File size
        - File format
        - Frequency of data uploads
    - Expected Data Volumes:
        - 100 million transactions
        - 20 million users
2. **File Processing Service:**
    - File Watcher: Periodically poll the Landing Zone/ SFTP Server for new files.
    - File processing service (like Apache Nifi, Python, Lambda, etc)
        - Manage files received on the LZ & segregate files based on the attributes like filename, metadata tags, etc.
        - Manage Unique identifiers on file attribute to avoid duplicate processing.
        - Setup batches to be processed based on size and type suitable for the Rest APIs.
        - Invoke relevant workflows and APIs for the specific file batches.
        - Ensure batches respect the API rate limits and response latencies.
        - Handle API response and trigger relevant error handling workflows like Retry, Failure actions and Success actions.
        - Send API responses, status codes and logs to Reporting Service for success/failure reports, dashboards and recon.
        - Move processed files from source location to archival storage with appropriate versioning and lifecycle rules (Amazon S3, Azure BLOB, etc.)
3. **API Response Handling:** (samples, not limited to. Refer relevant API dev documentation for detailed list of responses and actions)
    - **ADD_CUST API:**
        - **Response 200 –** Successful.
        - **Response 8006 –** Customer already exists Retry file with UPD_CUST API.
        - **Response 3217 –** Invalid File/source data. Mark as Failed. (No retry)
    - **UPD_CUST API: V**alidate response body for error
        - **Response 200** – Successful
        - **Response Conflicting profile error** - Failure
    - **ADD_TRX API:**
        - **Response 200 –** Successful
        - **Response 624 –** Incomplete body parameters. Mark as Failure. (No retry)
4. **Reporting Service:**
    - Logging: (ELK Stack, AWS CloudWatch, Azure Monitor, etc.)
        - Collect detailed logs for debugging and auditing.
        - Implement structured logging to facilitate easier analysis.
    - Reporting Dashboard: (Grafana, Kibana, etc.)
        - Provide real-time dashboards for successful ingestions and error reports.
        - Include metrics such as ingestion rate, processing latency, and error rates.

Non-Functional Requirements:

- **Scalability:**
  - The system must handle high data volumes (100 million transactions and 20 million users).
  - Implement auto-scaling for both the processing and integration layers based on load.
  - Use robust and scalable storage solutions (e.g., Amazon S3, Azure Blob Storage).
- **High Availability & Performance:**
  - The system must ensure high availability with minimal downtime.
  - Use a highly available SFTP service to ensure reliability.
  - Use an API Gateway to manage traffic and provide a unified entry point.
  - Employ a load balancer to distribute requests across multiple instances of the API service
- **Reliability**:
  - The system must handle errors gracefully and ensure data consistency.
  - Implement robust retry mechanisms and error handling
- **Security & Compliance**:
  - Ensure secure data transfer between components. (Encrypt data at-rest and transit)
  - Implement access control and encryption for sensitive data.
  - Ensure the system complies with relevant data protection regulations (e.g., GDPR).

High Level Solution Design


Files
SFTP
SFTP Server/Landing Zone
File Processing Service
(Nifi, Py, Lambda,etc.)
- File watcher, periodically polls landing zone for new files
- File processing service, reads the file attributes and metadata to segregate the files depending on types (user/trx file tag), identifier, batch_id, etc.
- Record file type and primary identifiers in a DB/ cache to determine which rest API to invoke.
- Invoke relevant APIs with the required request parameters.
- Handle API responses to trigger Success / Retry / Failure reporting.
- Move processed files from LZ to Archival storage.
API Calls
API Responses
REST APIs
ADD_CUST
UPD_CUST
ADD_TRX
CRM Application
Archive
Status and error logs
Reporting Service
(GRAFANA, etc)
Generate reports for files received, processed, success, retries and failures.

File Processing Service:

Summary:

The service will monitor an SFTP server for incoming files, process the files based on their attributes and metadata, invoke relevant REST APIs, handle responses, and manage file storage and reporting.

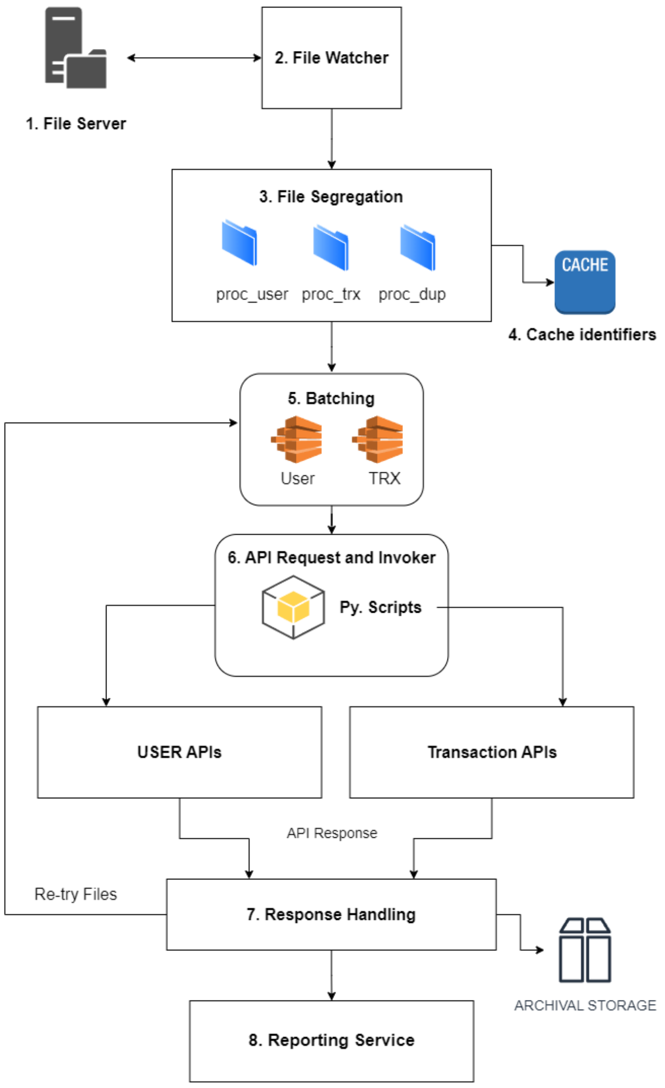
Functional requirements:

- **File Watcher**
  - Periodically poll an SFTP server location for incoming files.
- **File Segregation**
  - Segregate files into two separate folders (User Files and Transaction Files) based on file attributes and metadata.
- **Batch Processing**
  - Batch files for consumption by REST APIs, with a maximum of 100 files per API call.
- **API Invocation**
  - Invoke relevant APIs based on file attributes and metadata.
  - Ensure API call limits are respected (1000 calls per minute).
- **Response Handling**
  - Mark files as successful or failed based on API responses.
  - Allow certain response codes to trigger a retry via a different API.
- **Duplicate Processing Prevention**
  - Store file identifier tags in a database or cache to prevent duplicate processing.
- **File Storage Management**
  - Move processed files to archival storage.
  - Move failed files that need investigation to S3 storage.
- **Reporting**
  - Send data to a reporting service to enable comprehensive reports on successful and failed files.

Suggested Technologies:

- **SFTP Server:** OpenSSH SFTP, AWS Transfer for SFTP, Azure SFTP
- **File Storage:** EFS, FSx, etc.
- **DB/Cache Storage:** Redis, Elasticache, DynamoDB, etc.
- **Archival storage:** Amazon S3, Azure Blob, etc.
- **File Watcher:** Python, Paramiko (for SFTP)
- **File Processor:** Python, Apache Nifi, AWS Lambda, Azure Functions
- **Batching:** Python, Pandas (for data manipulation).
- **API Gateway:** AWS API Gateway, Azure API Management, etc.
- **Message Queue**: Apache Kafka, RabbitMQ, AWS SQS
- **Logging**: Python, JSON Parsing, AWS CloudWatch, Azure Monitor
- **Reporting Dashboard**: Kafka (for message queue), Grafana (for visualization).

High Level Design – File Processor



1. All files land at a specific inbound folder location.
2. File watcher polls the location periodically for new files

3. Files segregated and moved to specific proc_% sub folders, prepped for batching.
4. Record file unique identifiers into a db/cache to manage duplicates.

5. Pick-up files in batch sets and queue for API calls.
6. Use File attributes and py scripts to create API request body

Process the API calls

7. Response handler
- Read API responses and mark files as successful or failed.
- Move processed files to S3/archival storage.
- Translate failed responses and sets up 're-try' files back in Batch sets.
- Sends data relevant for reporting.

8. Reporting service
- Receives logs, status and other metrics to create reports.
- Ingest data via JSON parsing and Kafka / Amazon SQS
- Grafana for visualization and report generation.

Design walkthrough:

- File Watcher:
  - Functionality: Periodically polls the SFTP server location for incoming files.
  - Technology: Python, Paramiko (for SFTP)
  - Polling Interval: Configurable, e.g., every 5 minutes.
  - Configuration: File type, file size, frequency of data.
- File Segregation:
  - Functionality: Segregate files into User Files and Transaction Files based on file attributes and metadata.
  - File Attributes: Filename, file type, metadata fields.
  - Implementation: Python script to read file attributes and move files to appropriate directories.
- Duplicate Processing Prevention:

  - Functionality: Store file identifier tags in a database or cache to prevent duplicate processing.
  - Technology: Redis (for caching), PostgreSQL (for database).
  - Implementation: Check identifier tags before processing files.
-
- Batch Processing:
  - Functionality: Batch files for consumption by REST APIs.
  - Batch Size: Maximum 100 files per API call.
  - Technology: Python, Pandas (for data manipulation).
- API Invocation:
  - Functionality: Invoke relevant APIs based on file attributes and metadata.
  - API Limits: 1000 calls per minute.
  - Technology: Python, Requests library.
  - Implementation:
    - Read batched files.
    - Make API calls.
    - Handle rate limiting (throttling).
- Response Handling
  - Functionality: Handle responses from API calls and mark files as successful or failed.
  - Response Codes: Define codes that trigger success, failure, or retry via a different API.Implementation: Python, JSON parsing.
- File Storage Management:
  - Functionality: Move processed files to archival storage and failed files to S3 storage.
  - Technology: AWS SDK (Boto3 for S3).
  - Implementation: Python script to move files based on processing results.
- Reporting:
  - Functionality: Send data to a reporting service to enable comprehensive reports on successful and failed files.
  - Technology: Kafka (for message queue), Grafana (for visualization).
  - Implementation: Publish events to Kafka, visualize using Grafana.
