Implementation Blueprint

_Generated on 2025-06-01

Architecture Guide

1. High-level Architectural Overview

The architecture for the data pipeline project revolves around processing activity records, updating financial positions, applying bookkeeping rules, and calculating market values. It involves a sequence of tasks orchestrated in Directed Acyclic Graphs (DAGs) to ensure efficient data processing. The system is designed to handle high volumes of data with low latency, ensuring data integrity and accuracy throughout the pipeline.

2. Key Components and Responsibilities

- **Data Pipeline Components**: DAG tasks, job definitions, source systems, raw data schemas, target data models, lineage definitions, and data quality rules.
- **Responsibilities**:
  - Extracting, transforming, and loading data from source systems.
  - Applying business rules to update financial positions and margin balances.
  - Calculating market values based on product definitions and portfolio data.
  - Ensuring data quality through defined rules and checks.

3. Design Patterns and Principles

- **Event-Driven Architecture**: Utilizing Kafka for real-time data streaming and processing.
- **Micro-batching**: Using Airflow for orchestrating complex data pipelines in batch mode.
- **Service-Oriented Architecture**: Breaking down functionalities into individual tasks and jobs for modularity and scalability.

4. Scalability, Reliability, Observability, Maintainability

- **Scalability**: Leveraging Spark for large-scale data processing and Kubernetes for container orchestration.
- **Reliability**: Implementing error handling, retries, and monitoring in DAG tasks for fault tolerance.
- **Observability**: Utilizing Prometheus and Grafana for monitoring and tracking system performance.
- **Maintainability**: Ensuring clear documentation, modular design, and version control for ease of maintenance.

5. Data or Integration Flow Strategies

- **Data Flow**: Sequential processing of activity records to update positions, apply bookkeeping rules, and calculate market values.
- **Integration**: Extracting data from source systems, transforming it based on defined rules, and loading it into target data models.

6. Summary of Artifacts

The artifacts include user stories for financial position management, entity definitions for key data structures, DAG tasks for data pipeline operations, job definitions for batch processing, source systems for data ingestion, raw data schemas for data structure, target data models for storage, lineage definitions for data flow, and data quality rules for maintaining data integrity. These artifacts collectively define the architecture and design of the data pipeline project.

Detailed Artifact Definitions

Processing Activity Records to Update Financial Positions

- **Represents**: This artifact represents the feature of processing a stream of activity records and updating financial position documents sequentially.
- **Influences Architecture and System Design**: This feature influences the architecture by requiring a system capable of processing a high volume of records per second with low latency. It also necessitates a sequential processing mechanism to update financial positions accurately.
- **Design or Data Flow Considerations**: The system design should include a mechanism to detect changes in position keys and trigger booking operations accordingly. It should also handle successful updates, message publishing, and logging efficiently.
- **Specific Examples**: An example implementation could involve using a message queue to handle incoming activity records, a database to store financial positions, and a logging mechanism to track updates.

Applying Bookkeeping Rules to Margin Positions

- **Represents**: This artifact represents the feature of applying bookkeeping rules to margin positions based on activity records.
- **Influences Architecture and System Design**: This feature influences the architecture by requiring validation of activity and position attributes using bookkeeping rules. It also impacts the design by updating margin balances and positions based on validated rules.
- **Design or Data Flow Considerations**: The system design should include components for retrieving bookkeeping rules, validating activities, and updating margin-related entities. Error logging with metadata is also essential for tracking issues.
- **Specific Examples**: An example implementation could involve a rule engine for applying bookkeeping rules, services for updating margin balances and positions, and a logging system to capture errors.

Calculating Market Value of Products

- **Represents**: This artifact represents the feature of calculating the market value of products based on defined rules and portfolio control data.
- **Influences Architecture and System Design**: This feature influences the architecture by requiring a pricing service to calculate market values accurately. It also impacts the design by storing calculated values for downstream systems to access.
- **Design or Data Flow Considerations**: The system design should include components for defining pricing rules, calculating market values based on product definitions, and storing results for accessibility. Rounding calculations to the nearest whole number is also a design consideration.
- **Specific Examples**: An example implementation could involve a pricing engine for rule-based calculations, a data store for storing calculated values, and APIs for downstream systems to retrieve

market values.

Entity Definitions

Entity: ActivityRecord

Description: _

- **firm** (`string`): 4-character firm code
- **ledger** (`string`): Ledger identifier
- **branch** (`string`): Branch code
- **base** (`string`): Account base number (used to determine account type)
- **subAcctKey** (`string`): Sub-account key
- **acctCategory** (`string`): Account category (e.g., RETL, OMNI)
- **rr** (`string`): Registered representative code
- **usCitizenship** (`string`): US citizenship indicator
- **nyResident** (`string`): New York residency indicator
- **paper** (`string`): List of agreement indicators
- **intracctEligibleType** (`string`): Eligible intra-account types
- **intracctMinAmt** (`string`): Minimum transfer amount
- **accountClosedDate** (`string`): Account closure date

Entity: Position

Description: _

- **firmPostrmt** (`string`): Firm identifier for the position
- **ledgerPostrmt** (`string`): Ledger identifier for the position
- **branchPostrmt** (`string`): Branch code for the position
- **basePostrmt** (`string`): Base account number for the position
- **acctTypePostrmt** (`string`): Account type
- **tdQtyShortSide** (`string`): TD quantity
- **houseReqt** (`string`): House margin requirement
- **equityReqt** (`string`): Equity margin requirement
- **loss** (`string`): Loss on position
- **shortExpDate** (`string`): Expiration date
- **pairoffAcctType** (`string`): Pair-off account type
- **pairoffCusip** (`string`): Pair-off CUSIP
- **pairoffTdQty** (`string`): Pair-off TD quantity

Entity: PositionMetadata

Description: _

- **bondRedeemSw** (`string`): Bond redemption indicator
- **acctTransferredSw** (`string`): Account transferred indicator
- **reverseSplitSw** (`string`): Reverse split indicator
- **acctActivityQty** (`string`): Aggregate account activity quantity
- **acctActivityAmt** (`string`): Aggregate account activity amount

Entity Definitions

Entity: ManagedPosition

Description: _

- **firmProdpcc** (`string`): Firm code for product control
- **ledgerProdpcc** (`string`): Ledger identifier
- **branchProdpcc** (`string`): Branch code
- **baseProdpcc** (`string`): Base account number
- **subAcctProdpcc** (`string`): Sub-account identifier
- **uls** (`string`): ULS identifier
- **cusip** (`string`): CUSIP identifier
- **productLine** (`string`): Product line code
- **price** (`string`): Current price
- **prevPrice** (`string`): Previous day's price
- **strikePrice** (`string`): Strike price
- **contractSize** (`string`): Contract size
- **expOrMaturityDate** (`string`): Expiration or maturity date
- **securityFoundSw** (`string`): Indicator if security was found
- **ulsFoundSw** (`string`): Indicator if ULS was found

Entity: BookkeepingRule

Description: _

- **maxxKeyCode** (`string`): Key identifier
- **shortDesc** (`string`): Short description
- **longDesc** (`string`): Long description
- **updTdBalSw** (`string`): Update trade date balance switch
- **updSdBalSw** (`string`): Update settlement date balance switch
- **adjSmaSw** (`string`): Adjust SMA switch
- **updTdQtySw** (`string`): Update TD quantity switch
- **updSdQtySw** (`string`): Update SD quantity switch

Entity: MarginBalance

Description: _

- **tdBal** (`string`): Trade date balance
- **sdBal** (`string`): Settlement date balance
- **sma** (`string`): Special Memorandum Account balance
- **mvTdLong** (`string`): Market value trade date long
- **mvTdShort** (`string`): Market value trade date short
- **cashAvail** (`string`): Cash available
- **loanValue** (`string`): Loan value
- **reqtHouse** (`string`): House requirement
- **reqtExch** (`string`): Exchange requirement
- **equity** (`string`): Equity value

Entity Definitions

Entity: MarginRuleConfig

Description: _

- **nyrsLevel** (`string`): Margin rule level
- **nyrsAction** (`string`): Margin rule action
- **nyrsMgnRule** (`string`): Margin rule code
- **nyrsMgnSeq** (`string`): Margin rule sequence

Entity: ProductDefinition

Description: _

- **prodline** (`string`): Product line
- **descShort** (`string`): Short description
- **factor** (`string`): Pricing factor
- **defaultPriceForMgn** (`string`): Default price for margin calculations
- **fractionsAllowedSw** (`string`): Indicator if fractional units are allowed
- **marginableFlag** (`string`): Marginable status indicator
- **contribsToEquitySw** (`string`): Indicates equity contribution applicability

Entity: PortfolioConcentration

Description: _

- **tdQtyConc** (`string`): Trade date quantity concentration
- **marketVal** (`string`): Market value
- **cusipConc** (`string`): CUSIP concentration
- **acctCategoryConc** (`string`): Account category concentration

stories Artifacts

- **Represents**: These artifacts represent user stories that describe specific functionalities or features that need to be implemented in the project. They outline the requirements, acceptance criteria, entities involved, business rules, and other relevant details for each story.
- **Influences Architecture and System Design**: These stories influence the architecture and system design by providing a clear understanding of the functionalities that need to be implemented. They help in identifying the entities involved, business rules, data flow, and interactions between different components of the system.
- **Design or Data Flow Considerations**: The design considerations include how to process activity records, update positions, apply bookkeeping rules, and manage margin balances. The data flow considerations involve sequential processing of records, updating relevant entities based on rules, and logging errors and updates.
- **Specific Examples**:
    - For the "Process Activity Records to Update Position" story, the system must process a stream of activity records, update position documents sequentially, and publish a 'position.updated' message after each successful update.

- In the "Apply Activity Records to Managed Position" story, the system iterates through incoming activity records, applies each to its associated position, and stops when a change in positionKey is detected.
- The "Apply Bookkeeping Rules to Margin Position" story involves applying activity records to different position types, validating attributes using bookkeeping rules, updating margin balances, and managing margin positions based on rules.

stories Artifacts

- **Represents**: This artifact represents a user story for calculating the market value of a product based on certain conditions and rules.
- **Influences Architecture and System Design**: This user story influences the architecture and system design by requiring the implementation of a data pipeline that can process and calculate the market value of a product based on the defined rules and conditions.
- **Design or Data Flow Considerations**:
  - Data from entities like ProductDefinition, ManagedPosition, and PortfolioConcentration needs to be collected and processed to calculate the market value.
  - Business rules need to be implemented in the data pipeline to determine the correct pricing rule based on flags and conditions.
  - The calculated market value needs to be rounded to the nearest whole number and stored with the productId and timestamp for downstream systems to access.
- **Specific Examples**:
  - The data pipeline will need to fetch attributes like price, quantity, factor, contractSize, etc., from the entities involved to perform the calculations.
  - Different pricing rules need to be applied based on flags like EXT_RULE_FLAG to compute the market value accurately.

dag_tasks Artifacts

- **Represents**: These artifacts represent individual tasks or operations within the data pipeline. Each task has a specific operation it performs, such as extracting data or loading data to a staging area.
- **Influences Architecture and System Design**: These artifacts influence the architecture and system design by defining the sequence of tasks in the data pipeline. The dependencies between tasks determine the order in which they need to be executed, impacting the overall flow of data through the system.
- **Design or Data Flow Considerations**: When designing the architecture, considerations need to be made for how data will flow between tasks. This includes ensuring that data is passed correctly between tasks, handling errors or retries, and optimizing the performance of the pipeline.
- **Specific Examples**:
  - The task "ExtractActivityRecords" represents an operation to extract activity records from a data source.
  - The task "LoadActivityRecordsToStaging" depends on "ExtractActivityRecords" and represents loading the extracted records to a staging area.
  - The task "ExtractPositionData" represents another operation to extract position data from a different data source.

dag_tasks Artifacts

- **Represents**: These artifacts represent individual tasks in a Directed Acyclic Graph (DAG) that define operations to be performed in a data pipeline.
- **Influences architecture and system design**:
  - These artifacts influence the architecture by defining the sequence of operations and dependencies between tasks in the data pipeline.
  - They guide the design of the system by determining the flow of data and processing steps.
- **Design or data flow considerations**:
  - Considerations need to be made for task dependencies to ensure that tasks are executed in the correct order.
  - Error handling and retries should be incorporated into the design to handle failures in individual tasks.
- **Specific examples if relevant**:
  - In the provided artifacts, tasks like 'ExtractPositionData' and 'ExtractManagedPositionData' serve as data extraction operations, while 'LoadPositionDataToStaging' and 'LoadManagedPositionDataToStaging' represent loading data into staging areas. The dependencies between these tasks ensure that data is processed in the correct order.

dag_tasks Artifacts

- **Represents**: Each artifact represents a specific task or operation in the data pipeline process. It includes details like task ID, project ID, operation name, dependencies, and creation timestamp.
- **Influences Architecture and System Design**: These artifacts play a crucial role in defining the sequence of operations in the data pipeline. They help in structuring the workflow and determining the dependencies between different tasks. This influences the overall architecture by guiding how data moves through the system.
- **Design or Data Flow Considerations**: Design considerations include ensuring that tasks are executed in the correct order based on dependencies, handling failures and retries, monitoring task progress, and optimizing task performance. Data flow considerations involve understanding how data is passed from one task to another, ensuring data integrity, and managing data transformations.
- **Specific Examples**:
  - The task 'ExtractBookkeepingRules' does not have any dependencies, indicating it is the initial task in the pipeline responsible for extracting bookkeeping rules data.
  - The task 'LoadBookkeepingRulesToStaging' depends on 'ExtractBookkeepingRules', showing that it can only be executed after the extraction task is completed, ensuring data consistency.
  - The task 'ExtractMarginBalanceData' is another independent task that extracts margin balance data, showing the parallel processing capability of the data pipeline.

dag_tasks Artifacts

- **Represents**: This artifact represents individual tasks in a Directed Acyclic Graph (DAG) for a data pipeline project. Each task corresponds to a specific operation that needs to be performed as part of the overall data processing workflow.
- **Influences architecture and system design**: The presence of these dag_tasks influences the architecture by defining the sequence of operations and dependencies between tasks. It guides the design of the data pipeline system by outlining the steps required to transform and move data from source to destination.

- **Design or data flow considerations**: Design considerations include ensuring that tasks are executed in the correct order based on dependencies, handling task failures and retries, monitoring task progress, and optimizing task execution for performance. Data flow considerations involve understanding how data moves through the pipeline from extraction to loading stages.
- **Specific examples if relevant**: In the provided artifact, we have tasks such as 'ExtractProductDefinitionData' and 'LoadProductDefinitionDataToStaging' which demonstrate the extraction and loading of product definition data respectively. The dependencies between tasks indicate the order in which these operations need to be executed to ensure data consistency.

dag_tasks Artifacts

- **Represents**: These artifacts represent individual tasks or operations within the data pipeline. Each task has a specific operation to perform, such as extracting, loading, or transforming data.
- **Influences Architecture and System Design**: The presence of these artifacts influences the overall architecture and system design by defining the sequence of operations in the data pipeline. It helps in organizing and orchestrating the flow of data from extraction to transformation to loading.
- **Design or Data Flow Considerations**: Design considerations include ensuring that tasks are executed in the correct order based on dependencies. Data flow considerations involve understanding the input and output requirements of each task to ensure smooth data processing.
- **Specific Examples**:
  - `ExtractPortfolioConcentrationData`: This task represents the operation of extracting portfolio concentration data. It serves as the starting point in the data pipeline.
  - `LoadPortfolioConcentrationDataToStaging`: This task loads the extracted data into a staging area. It has a dependency on the `ExtractPortfolioConcentrationData` task, indicating the order of execution.
  - `TransformActivityRecords`: This task transforms activity records. It has a dependency on `LoadActivityRecordsToStaging`, suggesting that the staging data needs to be transformed.

dag_tasks Artifacts

- **Represents**: These artifacts represent specific tasks or operations that need to be performed as part of the data pipeline. Each task has a unique task_id, operation to be performed, dependencies on other tasks, and the project it belongs to.
- **Influences architecture and system design**: These artifacts influence the architecture and system design by defining the sequence of operations in the data pipeline. They help in organizing and structuring the flow of data processing tasks.
- **Design or data flow considerations**:
  - Task dependencies: The dependencies specified for each task determine the order in which tasks need to be executed. This influences the overall data flow and ensures that tasks are executed in the correct sequence.
  - Scalability: The design should consider the scalability of the tasks to handle large volumes of data efficiently. This may involve parallel processing, distributed computing, or other techniques.
- **Specific examples**:
  - **TransformPositionData**:
    - Represents a task to transform raw position data.
    - Depends on the task 'LoadPositionDataToStaging'.

        - This task could involve data cleaning, normalization, or any other transformation required on the position data.
    - **TransformManagedPositionData**:
        - Represents a task to transform managed position data.
        - Depends on the task 'LoadManagedPositionDataToStaging'.
        - This task could involve specific transformations related to managed positions.
    - **TransformBookkeepingRules**:
        - Represents a task to transform bookkeeping rules data.
        - Depends on the task 'LoadBookkeepingRulesToStaging'.
        - This task could involve processing and applying bookkeeping rules to the data.

dag_tasks Artifacts

- **Represents**: These artifacts represent specific tasks or operations that need to be performed as part of the data pipeline. Each task has a unique task_id, operation to be performed, dependencies on other tasks, and the project it belongs to.
- **Influences architecture and system design**: These artifacts influence the architecture and system design by defining the sequence of operations in the data pipeline. They help in organizing and orchestrating the flow of data processing tasks.
- **Design or data flow considerations**: Design considerations include ensuring that tasks are executed in the correct order based on their dependencies. Data flow considerations involve passing data between tasks efficiently and handling any errors or exceptions that may occur during task execution.
- **Specific examples**:
    - Task 'TransformMarginBalanceData' depends on 'LoadMarginBalanceDataToStaging' task.
    - Task 'TransformProductDefinitionData' depends on 'LoadProductDefinitionDataToStaging' task.
    - Task 'TransformPortfolioConcentrationData' depends on 'LoadPortfolioConcentrationDataToStaging' task.

dag_tasks Artifacts

- **Represents**: These artifacts represent individual tasks within the data pipeline. Each task corresponds to a specific operation that needs to be performed on the data.
- **Influences architecture and system design**: The presence of these tasks influences the overall architecture of the system by defining the sequence of operations that need to be executed. It also impacts the design by determining the dependencies between tasks and how they are orchestrated.
- **Design or data flow considerations**: Design considerations include how to handle task dependencies, error handling mechanisms, task scheduling, and resource allocation. Data flow considerations involve ensuring that data is passed correctly between tasks and that the output of one task serves as the input to another.
- **Specific examples if relevant**: For example, the task 'ApplyPositionUpdates' depends on the output of 'TransformActivityRecords' and 'TransformPositionData'. This dependency needs to be managed effectively to ensure that the tasks are executed in the correct order and that the data is processed accurately.

dag_tasks Artifacts

- **Represents**: The dag_tasks artifacts represent specific tasks or operations that need to be executed as part of the data pipeline. These tasks typically have dependencies on other tasks.

- **Influences architecture and system design**: These artifacts influence the architecture and system design by defining the sequence of operations in the data pipeline. They help in organizing the workflow and ensuring that tasks are executed in the correct order.
- **Design or data flow considerations**: Design considerations include ensuring that dependencies are properly defined and managed to maintain the integrity of the data pipeline. Data flow considerations involve understanding the input and output requirements of each task and how they interact with other tasks in the pipeline.
- **Specific examples**:
  - The task "UpdateMarginBalances" depends on "ApplyBookkeepingRules" and "TransformMarginBalanceData".
  - The task "CalculateMarketValues" depends on "TransformProductDefinitionData", "TransformManagedPositionData", and "TransformPortfolioConcentrationData".
  - The task "ValidateDataQuality" depends on "ApplyPositionUpdates", "ApplyManagedPositionUpdates", "UpdateMarginBalances", and "CalculateMarketValues".

dag_tasks Artifacts

- **Represents**: The `dag_tasks` artifacts represent individual tasks or operations within a data pipeline. Each task has a specific operation to perform and may have dependencies on other tasks.
- **Influences Architecture and System Design**: These artifacts influence the architecture and system design by defining the sequence of operations in the data pipeline. They help in organizing and orchestrating the flow of data processing.
- **Design or Data Flow Considerations**: When designing the system, considerations need to be made for task dependencies, error handling, parallel processing, and task scheduling. The order in which tasks are executed and how they interact with each other is crucial for the overall pipeline performance.
- **Specific Examples**:
  - `MaskSensitiveData`: This task represents an operation to mask or anonymize sensitive data before further processing. It depends on the `ValidateDataQuality` task, indicating that data quality checks need to be passed before sensitive data masking can occur.
  - `ArchiveProcessedData`: This task represents archiving the processed data. It depends on the `MaskSensitiveData` task, meaning that sensitive data masking must be completed before archiving the processed data.

dags Artifacts

- **Position_Update_Processing**
  - Represents a DAG that handles the processing of position updates based on activity records.
  - Influences architecture by defining the sequence of tasks and data flow for processing position updates.
  - Design considerations include ensuring data integrity during extraction, transformation, and loading processes.
  - Example: Extracting activity records, transforming them, extracting position data, applying updates, and validating data quality.
- **Managed_Position_Processing**

    - Represents a DAG that handles the processing of managed position updates based on activity records.
    - Similar to the previous artifact but focused on managed positions, influencing architecture in a similar manner.
    - Design considerations align with the previous DAG but tailored to managed positions data.
    - Example: Extracting managed position data, transforming it, applying updates, and archiving processed data.

- **Margin_Balance_Updating**

    - Represents a DAG that handles the application of bookkeeping rules and updates to margin balances.
    - Influences architecture by incorporating bookkeeping rules and margin balance data processing.
    - Design considerations include managing complex rule application and ensuring accurate margin balance updates.
    - Example: Extracting bookkeeping rules, transforming them, updating margin balances, and archiving processed data.

dags Artifacts

- **Represents**: This artifact represents a Directed Acyclic Graph (DAG) in the context of a data pipeline project. A DAG is a collection of tasks with dependencies between them.
- **Influences architecture and system design**: The DAG artifact influences the architecture and system design by defining the sequence of tasks and their dependencies. It helps in orchestrating the flow of data and processing in the pipeline.
- **Design or data flow considerations**: When designing the data flow, considerations need to be made for the order of tasks, data dependencies between tasks, error handling, and monitoring of task execution.
- **Specific examples if relevant**: In the given example, the DAG "Market_Value_Calculation" includes tasks like data extraction, transformation, calculation, validation, and archiving. Each task represents a step in the data pipeline process, and the DAG orchestrates the flow of data through these tasks according to the defined schedule.

job_definitions Artifacts

- **Represents**: These artifacts represent the definition of different jobs within the data pipeline project. Each job has a unique ID, project ID, name, type (batch in this case), schedule, retry policy, resource requirements (CPU and memory), and creation timestamp.
- **Influences Architecture and System Design**: The job_definitions artifacts influence the architecture and system design by defining the specific tasks that need to be executed as part of the data pipeline. These artifacts help in structuring the workflow, resource allocation, and scheduling of jobs within the system.
- **Design or Data Flow Considerations**: When designing the system architecture, considerations need to be made for how these jobs will interact with each other, how data will flow between them, and how resources will be allocated to ensure efficient execution. It is important to design error handling mechanisms based on the retry policy specified for each job.
- **Specific Examples**:

- The `DailyPositionValuationBatch` job runs daily at 2:00 AM, requires 2 cores of CPU and 4GB of memory, and has a retry policy of 3 retries with exponential backoff.
- The `DailyManagedPositionUpdateBatch` job runs daily at 3:00 AM, has similar resource requirements and retry policy.
- The `DailyBookkeepingBatch` job runs daily at 4:00 AM, with similar specifications as the other jobs.

job_definitions Artifacts

- **What this artifact represents**
  - The job_definitions artifact represents a specific job within the data pipeline project. It includes details such as job ID, project ID, job name, type (e.g., batch or real-time), schedule, retry policy, resources required (e.g., CPU, memory), and creation timestamp.
- **How it influences architecture and system design**
  - The job_definitions artifact influences the architecture and system design by defining the individual tasks or jobs that need to be executed as part of the data pipeline. It helps in organizing and orchestrating the flow of data processing within the system.
- **Design or data flow considerations**
  - Design considerations include how each job interacts with other jobs in the pipeline, the order of execution, error handling mechanisms, resource allocation, and scalability. Data flow considerations involve how data is passed between different jobs, transformations applied at each stage, and ensuring data integrity throughout the pipeline.
- **Specific examples if relevant**
  - For example, the 'DailyMarketValueCalculationBatch' job mentioned in the artifact runs daily at 5:00 AM, calculates market values, and requires 2 cores and 4GB of memory. This job would be part of a larger data pipeline responsible for processing market data and generating reports.

Source Systems Artifacts

- **User Management System**
  - Represents a database containing user profile information.
  - Influences architecture by requiring data extraction, transformation, and loading processes to integrate user data from this system into the overall data pipeline.
  - Design considerations include ensuring data security and privacy measures are in place when handling sensitive user information.
  - Example: When a new user registers in the system, their profile information needs to be extracted and processed for further analysis or storage.
- **Product Catalog**
  - Represents an API that provides product details.
  - Influences architecture by requiring real-time or scheduled data retrieval from the API to update the product catalog in the data pipeline.

- Design considerations include handling API rate limits, error handling for failed API requests, and caching mechanisms to improve performance.
- Example: The data pipeline fetches product details from the API at regular intervals to keep the product catalog up-to-date for users.

raw_data_schemas Artifacts

- **What this artifact represents**
  - The raw_data_schemas artifacts represent the schema definitions of raw data sources used in the project. These artifacts define the structure of the data that will be ingested into the system.
- **How it influences architecture and system design**
  - The raw data schemas influence the architecture and system design by providing a blueprint for how the data will be processed, transformed, and stored within the system. The schema definitions help in designing the data pipelines and data processing logic.
- **Design or data flow considerations**
  - Design considerations include mapping the raw data schemas to the target data model, handling schema evolution, and ensuring data quality and integrity throughout the data pipeline. Data flow considerations involve how the data will be transformed, enriched, and loaded into the target system.
- **Specific examples if relevant**
  - For example, the schema definition for the 'User Management System' includes fields like 'user_id', 'username', 'email', and 'registration_date' with their respective data types. This schema will influence how user data is processed and stored within the system. Similarly, the schema definition for the 'Product Catalog' includes fields like 'product_id', 'product_name', 'product_description', and 'price', which will impact how product data is handled in the system.

target_data_models Artifacts

- **What this artifact represents**
  - This artifact represents a specific target data model called "User Profile" with a wide schema that includes attributes like user_id, username, email, registration_date, product_id, product_name, product_description, and price.
- **How it influences architecture and system design**
  - This artifact influences the architecture and system design by defining the structure and attributes of the "User Profile" data model. It helps in determining how data will be stored, accessed, and processed within the system.
- **Design or data flow considerations**
  - Design considerations include how the data will be ingested, transformed, and loaded into the target data model. Data flow considerations involve ensuring that the data pipeline can

efficiently handle the schema defined in the artifact.

- **Specific examples if relevant**
  - For example, the "User Profile" data model can be used to store information about users, their associated products, and pricing details. This data can be utilized for personalization, analytics, and reporting purposes within the system.

lineage_definitions Artifacts

- **What this artifact represents**
  - This artifact represents the lineage or data flow between different data sources, transformations, and destinations within the project.
- **How it influences architecture and system design**
  - It influences the architecture by providing insights into how data moves through the system, helping in designing efficient data pipelines and ensuring data integrity.
- **Design or data flow considerations**
  - Design considerations include ensuring proper data mapping, handling data transformations effectively, and maintaining data lineage for traceability and auditing purposes.
- **Specific examples if relevant**
  - In the given example, the artifact shows the lineage from the User Management System to the User Profile through a specific transformation of joining with the Product Catalog on user_id. This information can guide the design of the data pipeline to incorporate this transformation accurately.

data_quality_rules Artifacts

- **What this artifact represents**
  - The data_quality_rules artifacts represent specific rules or conditions that data must adhere to in order to maintain data quality within the system.
- **How it influences architecture and system design**
  - These artifacts influence the architecture and system design by defining the criteria for data quality checks that need to be implemented within the data pipeline. This ensures that the data flowing through the system meets certain quality standards.
- **Design or data flow considerations**
  - Design considerations include incorporating these data quality rules into the data pipeline architecture at appropriate stages to validate data. Data flow considerations involve ensuring that data passes through these quality checks seamlessly without impacting performance.
- **Specific examples if relevant**

    - For example, the "Check Null User ID" rule ensures that the user_id field is not null in the incoming data. This rule can be implemented as a validation step in the data pipeline to prevent any records with null user IDs from entering the system.

Tech Stack Implementation Guide

Implementation Guide for Software/Data Platform Project

Setup Instructions:

- Set up React for frontend
- Use Node.js for backend development
- Utilize PostgreSQL for database storage
- Implement Kafka for messaging
- Use Airflow for orchestration
- Utilize Spark for data processing
- Use HDFS for storage layer
- Implement Prometheus and Grafana for observability
- Utilize Docker and Kubernetes for containerization and orchestration

Apache Spark Usage:

- Use Spark for processing entities and DAG tasks
- Implement Spark jobs to read from Kafka and write to HDFS
- Utilize Spark transformations and actions for data processing
- Example: Spark job to process ActivityRecord entity and write to HDFS

Airflow DAG Design:

- Design Airflow DAGs around pipeline stages
- Define tasks for each stage using Airflow operators
- Use dependencies between tasks to orchestrate the pipeline flow
- Example: Airflow DAG for processing Position entity and storing in PostgreSQL

MongoDB Schema Design:

- Design MongoDB schema using extracted entities
- Define collections for each entity with appropriate fields
- Utilize MongoDB indexes for efficient querying
- Example: MongoDB schema for ManagedPosition entity

Kafka Messaging Patterns:

- Implement Kafka topics for supporting flows or DAGs
- Use Kafka producers and consumers for data streaming
- Implement partitioning strategies for scalability

- Example: Kafka topic for streaming PositionMetadata updates

HDFS Usage:

- Utilize HDFS for batch storage or intermediate checkpoints
- Store processed data in HDFS for further analysis
- Implement data partitioning in HDFS for efficient data retrieval
- Example: Store MarginBalance data in HDFS for historical analysis

Integration Best Practices:

- Ensure seamless integration between components using appropriate connectors
- Use Kafka Connect for integrating Kafka with other components
- Implement data pipelines with error handling and retries
- Monitor data flow between components for data consistency

Security and Scalability Recommendations:

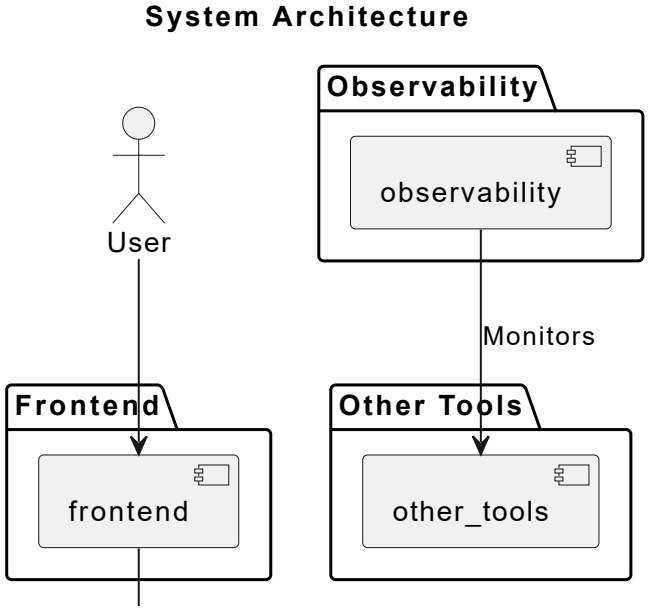
- Implement role-based access control for data security
- Use SSL encryption for data transmission between components
- Implement horizontal scaling for Spark jobs and Airflow tasks
- Monitor system performance and scale resources as needed

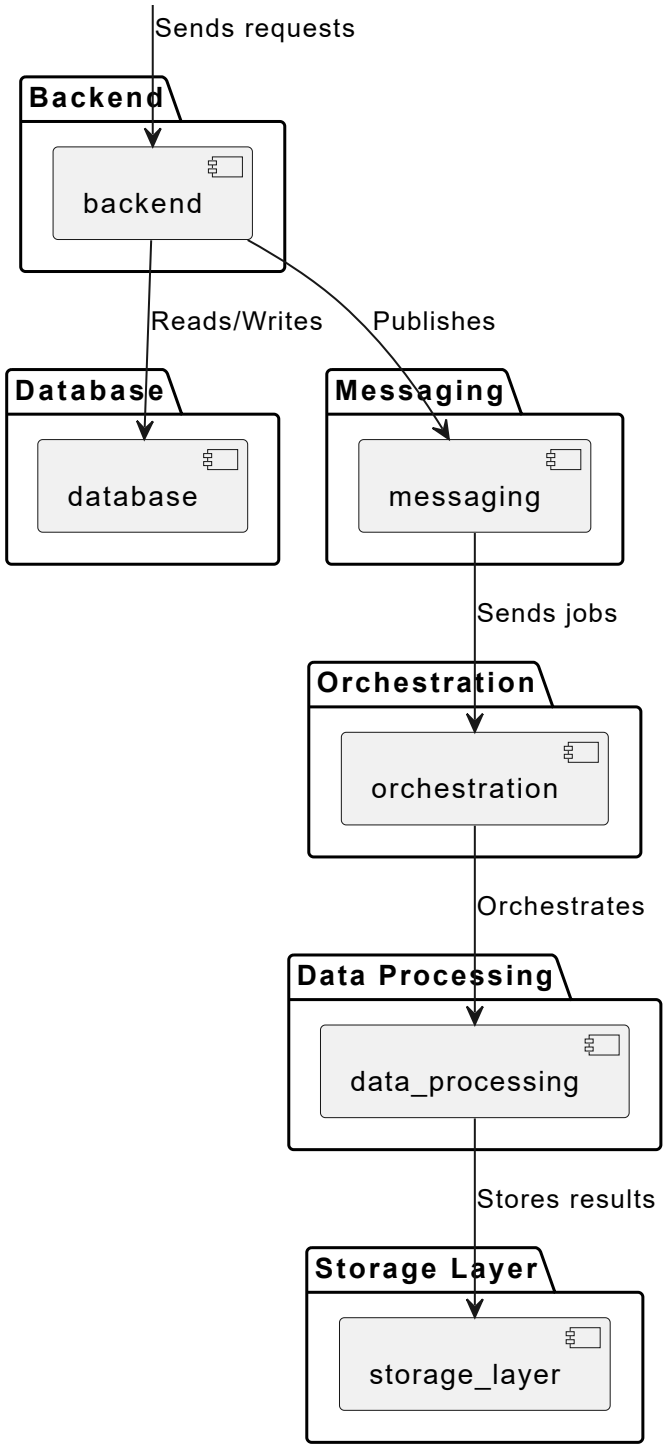
Common Pitfalls to Avoid:

- Avoid data skew in Spark jobs by implementing proper data partitioning
- Monitor Kafka lag to prevent data processing delays
- Ensure proper error handling in Airflow DAGs to handle failures gracefully
- Regularly monitor HDFS storage usage to prevent data loss or corruption

By following these implementation guidelines, you can effectively build and deploy a robust software/data platform project using the specified tech stack.

Final System Diagram



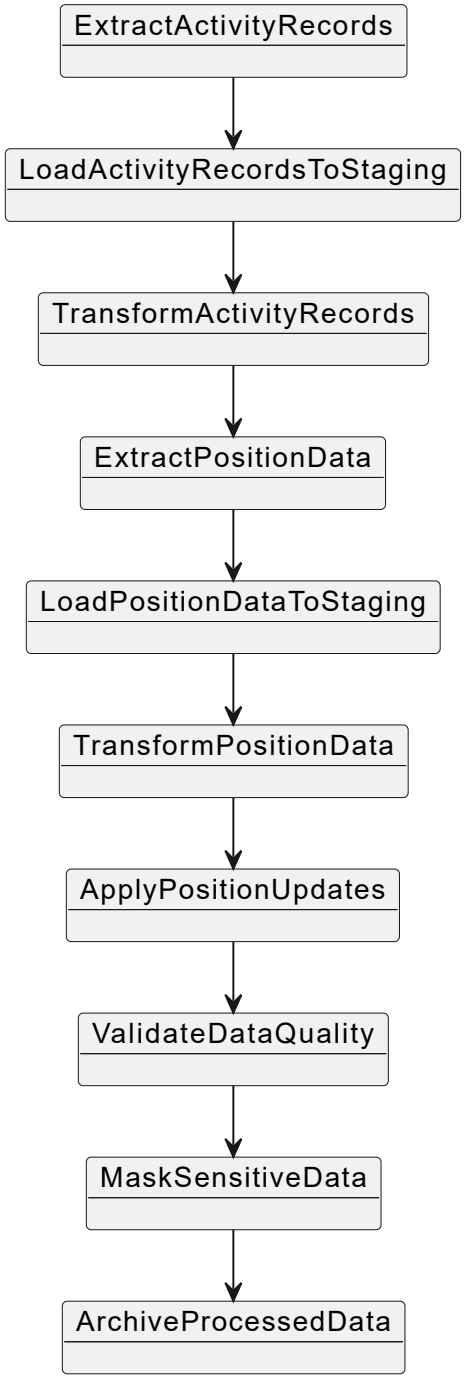


System-Level Diagrams

Dag Diagrams

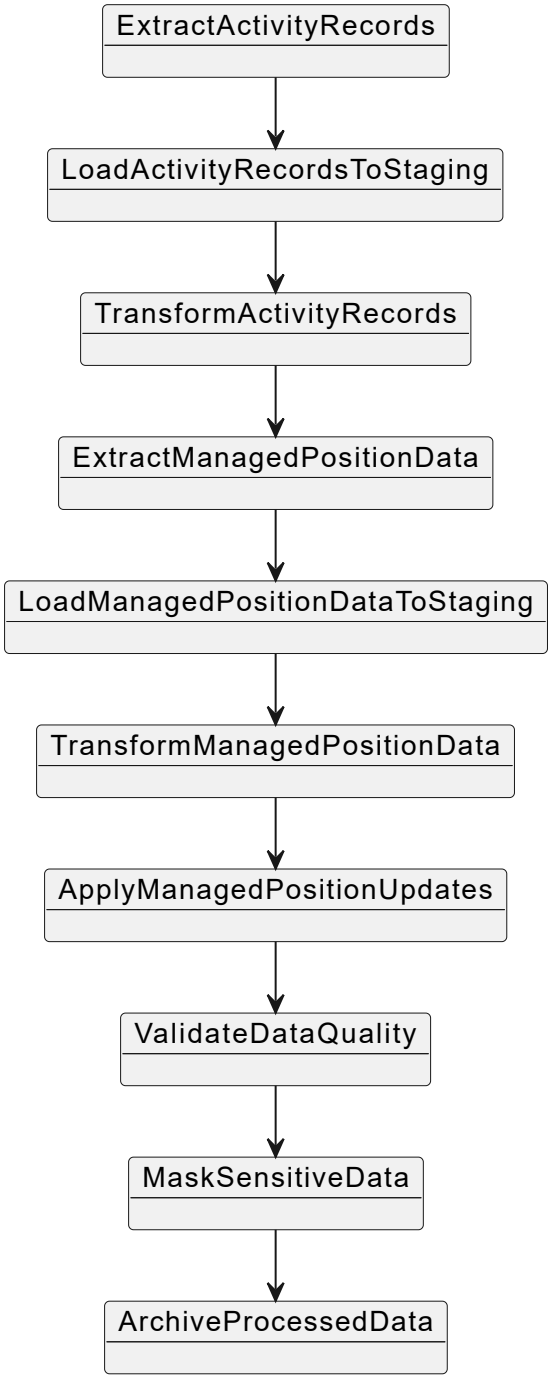
Dag Diagram 1

DAG: Position_Update_Processing



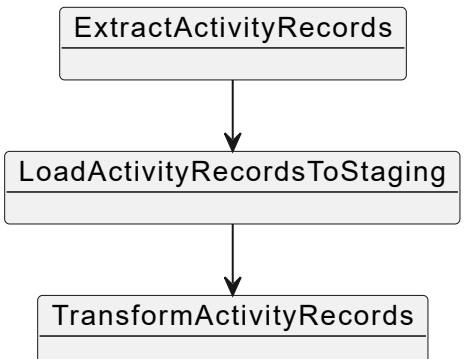
Dag Diagram 2

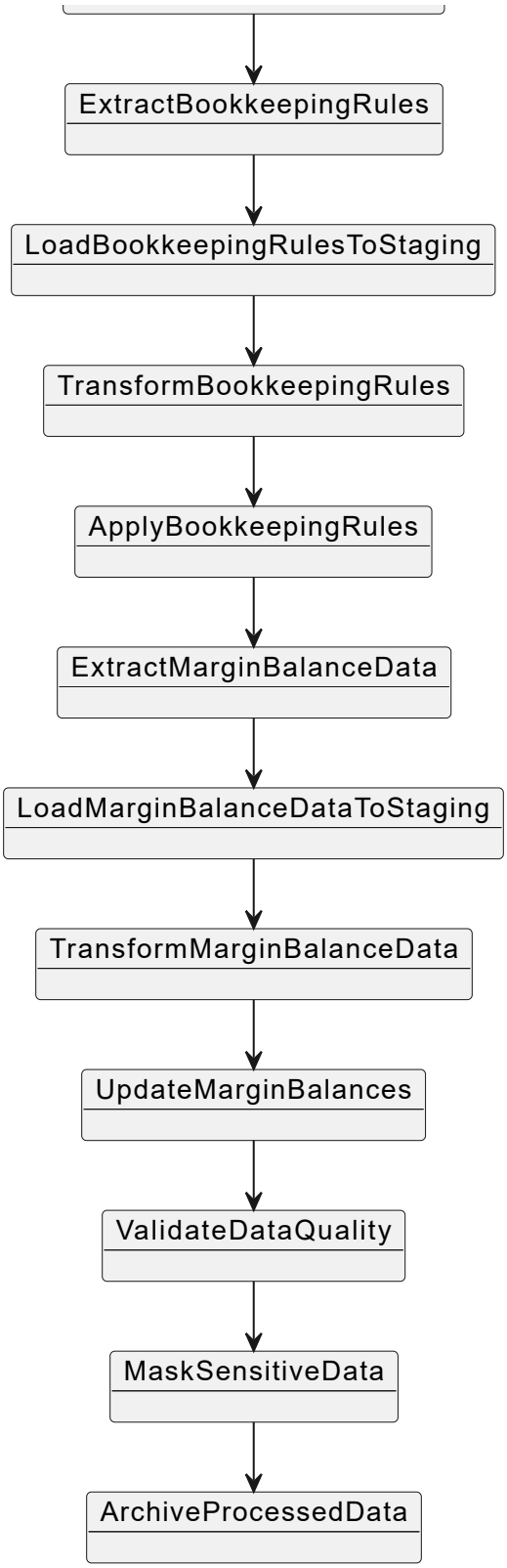
DAG: Managed_Position_Processing



Dag Diagram 3

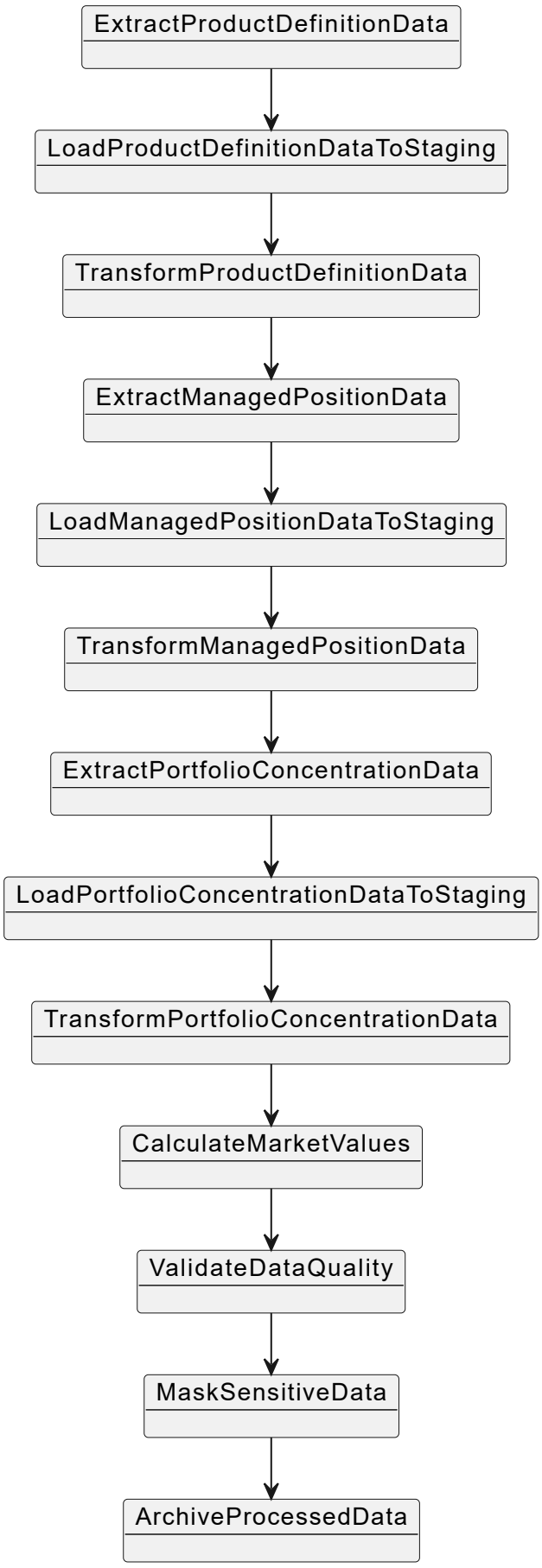
DAG: Margin_Balance_Updating





Dag Diagram 4

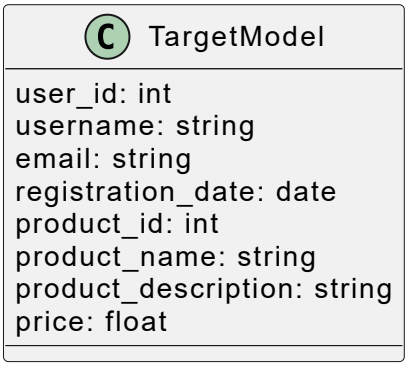
DAG: Market_Value_Calculation



Target_Data_Model Diagrams

Target_Data_Model Diagram 1

Target Model: TargetModel (Tabular)



Architectural Decision Records (ADRs)

ADR 1: ADR 1: Processing Activity Records Decision

Context: The system needs to process a stream of activity records and update financial position documents sequentially with low latency and high throughput.

Decision: Utilize a message queue system to handle incoming activity records, a database for storing financial positions, and a logging mechanism to track updates.

Alternatives: Considered using a stream processing system instead of a message queue for real-time processing.

Rationale: The chosen approach with a message queue allows for efficient handling of high volumes of records, sequential processing, and easy scalability.

ADR 2: ADR 2: Applying Bookkeeping Rules Decision

Context: The system needs to apply bookkeeping rules to margin positions based on activity records.

Decision: Implement a rule engine for applying bookkeeping rules, services for updating margin balances and positions, and a logging system for error tracking.

Alternatives: Considered manual rule application without a rule engine.

Rationale: The rule engine provides flexibility for rule changes, automation of rule application, and better error tracking for validation failures.

ADR 3: ADR 3: Calculating Market Value Decision

Context: The system needs to calculate the market value of products based on defined rules and portfolio control data.

Decision: Utilize a pricing engine for rule-based calculations, a data store for storing calculated values, and APIs for downstream systems to retrieve market values.

Alternatives: Considered manual calculations without a pricing engine.

Rationale: The pricing engine ensures accurate and consistent calculations, easy retrieval of market values, and scalability for handling large volumes of data.

ADR 4: ADR 1: Processing Activity Records

Context: The data pipeline project requires a mechanism to process a stream of activity records and update financial positions sequentially.

Decision: Utilize a message queue to handle incoming activity records, a database to store financial positions, and a logging mechanism to track updates.

Alternatives: Consider using a stream processing framework instead of a message queue.

Rationale: A message queue provides reliable message delivery, scalability, and decoupling of components, ensuring efficient processing of high volumes of records with low latency.

ADR 5: ADR 2: Applying Bookkeeping Rules

Context: The project needs to apply bookkeeping rules to margin positions based on activity records.

Decision: Implement a rule engine for applying bookkeeping rules, services for updating margin balances and positions, and a logging system to capture errors.

Alternatives: Consider manual rule application instead of a rule engine.

Rationale: A rule engine automates the application of complex bookkeeping rules, ensures consistency, and reduces manual errors, enhancing the accuracy and efficiency of updating margin positions.

ADR 6: ADR 3: Calculating Market Value

Context: The project requires calculating the market value of products based on defined rules and portfolio control data.

Decision: Utilize a pricing engine for rule-based calculations, a data store for storing calculated values, and APIs for downstream systems to retrieve market values.

Alternatives: Consider manual calculations instead of a pricing engine.

Rationale: A pricing engine automates the calculation process, ensures consistency in applying pricing rules, and provides a scalable solution for calculating market values accurately and efficiently.

ADR 7: Designing Data Pipeline Tasks

Context: The data pipeline project requires defining individual tasks for data extraction, transformation, and loading.

Decision: To create specific DAG tasks for each operation in the data pipeline.

Alternatives: 1. Using generic tasks for all operations. 2. Defining tasks at a higher level of abstraction.

Rationale: Creating specific DAG tasks allows for better orchestration, error handling, and monitoring of each operation, ensuring data integrity and efficient processing.

ADR 8: Dependency Management in DAG Tasks

Context: The data pipeline tasks have dependencies that need to be managed effectively.

Decision: To define dependencies between tasks in the DAG.

Alternatives: 1. Allowing tasks to run independently. 2. Using a centralized dependency management system.

Rationale: Defining dependencies between tasks ensures the correct order of execution, data flow, and prevents issues like race conditions or data inconsistencies.

ADR 9: Handling Data Quality Rules in the Pipeline

Context: Data quality rules need to be enforced within the data pipeline.

Decision: To incorporate data quality checks as part of the data processing tasks.

Alternatives: 1. Implementing data quality checks separately. 2. Skipping data quality checks for performance reasons.

Rationale: Including data quality checks within the pipeline tasks ensures that only high-quality data is processed further, maintaining the integrity of the system and downstream processes.

ADR 10: Design Decision for Extracting Margin Balance Data

Context: The data pipeline project requires extracting margin balance data for further processing.

Decision: Use the task 'ExtractMarginBalanceData' to extract margin balance data from the source system.

Alternatives: Consider using a different data extraction approach or tool for extracting margin balance data.

Rationale: The decision to use the 'ExtractMarginBalanceData' task aligns with the defined architecture and ensures consistency in data extraction processes.

ADR 11: Design Decision for Loading Product Definition Data

Context: The project involves loading product definition data into a staging area for processing.

Decision: Utilize the task 'LoadProductDefinitionDataToStaging' to load product definition data.

Alternatives: Explore alternative methods for loading product definition data to the staging area.

Rationale: By choosing the 'LoadProductDefinitionDataToStaging' task, the data flow remains consistent and follows the defined architecture guidelines.

ADR 12: Design Decision for Transforming Activity Records

Context: Transformation of activity records is a critical step in the data pipeline.

Decision: Implement the task 'TransformActivityRecords' for transforming activity records.

Alternatives: Consider different transformation approaches or tools for activity record processing.

Rationale: Selecting the 'TransformActivityRecords' task ensures a standardized transformation process and maintains data integrity throughout the pipeline.

ADR 13: Decision on DAG Task Dependencies

Context: The data pipeline project requires defining dependencies between tasks in Directed Acyclic Graphs (DAGs) to ensure proper execution order.

Decision: Decided to establish dependencies between tasks based on their logical order of execution and data flow requirements.

Alternatives: Considered using a dynamic dependency resolution system or a manual task execution approach without explicit dependencies.

Rationale: Establishing clear dependencies between tasks ensures that data flows correctly through the pipeline, prevents race conditions, and maintains the integrity of the data processing workflow.

ADR 14: Decision on Job Scheduling and Resource Allocation

Context: The project involves defining job schedules, retry policies, and resource requirements for efficient execution.

Decision: Decided to schedule jobs at specific times, implement retry policies for fault tolerance, and allocate appropriate CPU and memory resources.

Alternatives: Considered using event-driven triggers for job execution, implementing custom retry logic, or dynamic resource allocation.

Rationale: Scheduled job execution ensures timely processing, retry policies handle transient failures, and resource allocation optimizes performance and scalability of the data pipeline.

ADR 15: Decision on Processing Activity Records to Update Financial Positions

Context: The data pipeline project requires a mechanism to process activity records and update financial positions sequentially.

Decision: Utilize a message queue system to handle incoming activity records, a database for storing financial positions, and a logging mechanism to track updates.

Alternatives: Consider using a stream processing system instead of a message queue, storing positions in-memory for faster access, or using a different logging mechanism.

Rationale: The selected decision provides a reliable and scalable solution for processing activity records, updating positions accurately, and tracking updates efficiently.

ADR 16: Decision on Applying Bookkeeping Rules to Margin Positions

Context: The project requires applying bookkeeping rules to margin positions based on activity records.

Decision: Implement a rule engine for applying bookkeeping rules, services for updating margin balances and positions, and a logging system for error tracking.

Alternatives: Consider manual rule application, using a different rule engine, or implementing custom validation logic.

Rationale: The decision to use a rule engine provides a scalable and maintainable solution for applying complex bookkeeping rules, ensuring accurate updates to margin positions.

ADR 17: Decision on Calculating Market Value of Products

Context: The project involves calculating the market value of products based on defined rules and portfolio data.

Decision: Utilize a pricing engine for rule-based calculations, a data store for storing calculated values, and APIs for downstream systems to retrieve market values.

Alternatives: Consider manual calculations, using a different pricing engine, or storing values in a different data format.

Rationale: The decision to use a pricing engine and data store provides a scalable and efficient solution for calculating market values accurately and making them accessible to downstream systems.

ADR 18: Designing Data Pipeline DAGs

Context: Given the need to orchestrate data processing tasks in the data pipeline project using Directed Acyclic Graphs (DAGs).

Decision: To design specific DAGs for tasks like Market Value Calculation, Position Update Processing, Managed Position Processing, and Margin Balance Updating.

Alternatives: Considered using a single DAG for all tasks or creating separate DAGs for each task type.

Rationale: Creating separate DAGs allows for better organization, scalability, and maintenance of the data pipeline. Each DAG can focus on specific operations and dependencies, making it easier to manage and troubleshoot.

ADR 19: Defining Job Definitions

Context: With the requirement to schedule and execute batch jobs within the data pipeline project.

Decision: To define job definitions for Daily Position Valuation, Daily Managed Position Update, Daily Bookkeeping, and Daily Market Value Calculation batches.

Alternatives: Considered using a generic job definition for all batches or defining jobs dynamically based on project requirements.

Rationale: Defining specific job definitions for each batch allows for better control over scheduling, resource allocation, and retry policies. It ensures that each batch job is tailored to its specific processing needs.

ADR 20: Incorporating Source Systems

Context: Given the need to integrate data from external sources like the User Management System and Product Catalog into the data pipeline project.

Decision: To incorporate the User Management System and Product Catalog as source systems in the project artifacts.

Alternatives: Considered using only internal data sources or excluding one of the external sources.

Rationale: Incorporating external source systems enriches the data pipeline with valuable information, enabling more comprehensive data processing and analysis. It enhances the project's capabilities and data quality.

ADR 21: Designing Data Pipeline for User Profile Management

Context: The project involves integrating user profile data from the User Management System and Product Catalog into a unified User Profile data model.

Decision: To design a data pipeline that extracts, transforms, and loads data from source systems into the target User Profile data model.

Alternatives: Considered alternative data integration approaches, such as real-time streaming vs. batch processing, and different data transformation techniques.

Rationale: Batch processing was chosen for data integration to ensure consistency and reliability in handling large volumes of data. Transformations will be applied to align source data with the target data model, ensuring data quality and integrity.

ADR 22: Ensuring Data Quality in User Profile Data

Context: Data quality rules have been defined to maintain the integrity of the User Profile data.

Decision: To implement data quality checks, such as 'Check Null User ID' and 'Check Null Product ID', within the data pipeline.

Alternatives: Considered different data quality rules and severity levels for validation.

Rationale: The chosen data quality rules focus on critical data attributes like user_id and product_id to prevent null values and ensure data completeness. By enforcing these rules, data integrity is maintained throughout the pipeline.

ADR 23: Defining Data Lineage for User Profile Data

Context: The lineage between the User Management System, Product Catalog, and User Profile data needs to be established.

Decision: To document the data flow and transformations involved in moving data from source systems to the User Profile data model.

Alternatives: Considered different ways of documenting data lineage, such as visual diagrams or textual descriptions.

Rationale: By documenting the lineage and transformations, stakeholders can understand how data is processed and ensure transparency and traceability in data movements. This documentation aids in troubleshooting and auditing data flow processes.