Event Naming for Arbitrage

User

Can you give me some examples of how to name events in an automated cryptocurrency arbitrage system?

Tool

All the files uploaded by the user have been fully loaded. Searching won't provide additional information.

ChatGPT

In an automated cryptocurrency arbitrage system, events should be named to reflect actions or occurrences in the past that are significant for the system. Events are typically represented in past tense, focusing on describing what happened rather than what will happen. These events help in tracking the system’s behavior and triggering further actions or workflows in a functional, stateless manner.

Here are some examples of appropriately named events for such a system:

1. **PriceDifferentialDetected**: This event could represent the system detecting a price difference between two exchanges that presents an arbitrage opportunity.
2. **BuyOrderPlaced**: Signifies that the system successfully placed a buy order on a specific exchange.
3. **SellOrderPlaced**: Indicates that a sell order was placed to complete the arbitrage trade on another exchange.
4. **OrderExecutionConfirmed**: An event that records confirmation of a successful buy or sell order execution.
5. **FundsTransferredToExchange**: Represents the transfer of cryptocurrency or fiat between wallets or exchanges.
6. **ProfitCalculated**: After an arbitrage trade is completed, this event signals the successful calculation of profit.
7. **ArbitrageOpportunityMissed**: This event could signal a missed arbitrage opportunity due to market movement or delays in execution.
8. **TransactionRolledBack**: In cases where the system detects a failure or issue in executing the trade, this event indicates a rollback or corrective action.
9. **TradeFailedDueToInsufficientLiquidity**: This would record a failure to execute a trade due to insufficient liquidity on an exchange.
10. **ExchangeRateUpdated**: Indicates that the system has received updated exchange rates, crucial for detecting new arbitrage opportunities.

These event names align with the Domain-Driven Design (DDD) practice of using Ubiquitous Language, where domain experts and developers share a common vocabulary. They are also designed in a way that adheres to the principles of functional programming and event storming, ensuring clarity and simplicity in understanding system behaviors 【8†source】 【9†source】 【11†source】 .

User

In an automated cryptocurrency arbitrage system, give an example of a domain process

ChatGPT

In an automated cryptocurrency arbitrage system, a domain process could involve multiple steps that together achieve a business goal—in this case, capitalizing on price differences between different exchanges to generate profit. Here's an example of a **"Execute Arbitrage Trade"** domain process, which involves identifying and executing trades based on price discrepancies.

### Domain Process: Execute Arbitrage Trade

1. **Event: PriceDifferentialDetected**
   - The system identifies a price difference between two exchanges that exceeds the arbitrage threshold (e.g., Exchange A sells Bitcoin at $30,000, while Exchange B buys at $30,500).

2. **Command: SubmitBuyOrder**
   - The system issues a command to buy Bitcoin on Exchange A, where the price is lower. This command triggers the next step only if successful.

3. **Event: BuyOrderPlaced**
   - After the buy order is successfully placed, the event **BuyOrderPlaced** is generated, confirming that the order is active on Exchange A.

4. **Command: SubmitSellOrder**
   - Once the buy order is confirmed, the system submits a sell order on Exchange B, where the price is higher.

5. **Event: SellOrderPlaced**
   - If the sell order is successfully placed, an event **SellOrderPlaced** is triggered, confirming that the trade process on Exchange B is in progress.

6. **Event: OrderExecutionConfirmed**
   - The system waits for confirmation that both the buy and sell orders have been fully executed. This event marks the completion of the arbitrage trade.

7. **Event: ProfitCalculated**
   - After the trades are executed, the system calculates the profit based on the buying and selling prices, transaction fees, and transfer costs. The event **ProfitCalculated** is triggered.

8. **Command: TransferFundsToWallet**
   - The system initiates a command to transfer the profits (and any leftover funds) back to the central wallet or to be reinvested in future arbitrage trades.

### Example Workflow

```haskell
-- Define the workflow that handles the arbitrage process

executeArbitrage :: PriceDifferential -> Result
executeArbitrage priceDiff =
    let buyOrder = submitBuyOrder priceDiff.buyExchange priceDiff.price
        sellOrder = submitSellOrder priceDiff.sellExchange priceDiff.price
        profit = calculateProfit buyOrder sellOrder
    in if buyOrderSuccess buyOrder && sellOrderSuccess sellOrder
        then transferFunds profit
        else handleFailedOrder
```

```

### Explanation

- **Events** like `PriceDifferentialDetected`, `BuyOrderPlaced`, `SellOrderPlaced`, and `ProfitCalculated` represent past occurrences in the system. They signal important transitions in the arbitrage workflow.
- **Commands** like `SubmitBuyOrder` and `SubmitSellOrder` are directives that initiate specific actions in the system. They are invoked based on business rules (e.g., when a price difference is detected).
- The process completes with the final step of transferring profits or handling failed trades, maintaining immutability and side-effect-free functions except at the workflow boundaries (e.g., interacting with exchanges).

This domain process is crucial in the cryptocurrency arbitrage system because it reflects how the system capitalizes on market inefficiencies. By modeling it with clear commands and events, the process can be easily understood and extended for different trading strategies or market conditions 【8†source】 【9†source】 【10†source】 .

**User**

how are domain services captured during event storming

**ChatGPT**

During **event storming**, domain services are identified as specialized operations that don't naturally belong to any particular entity or context but are essential for the business process. These services typically handle complex logic or cross-cutting concerns that cannot be easily tied to a single entity. In Domain-Driven Design (DDD), domain services often capture actions that involve coordination between different parts of the system or require external operations, like integrations with third-party systems or complex calculations.

Here's how domain services are captured during event storming:

### 1. **Identifying Complex Operations**
Domain services usually emerge when you notice complex operations or behaviors during event storming that don't fit neatly within the boundaries of a single domain entity. For example, in an **automated cryptocurrency arbitrage system**, actions like calculating the optimal time to execute a trade, managing risk across multiple exchanges, or orchestrating transactions across exchanges are prime candidates for domain services.

   - **Example**: When an event like `PriceDifferentialDetected` is generated, a domain service could be responsible for calculating whether the difference justifies an arbitrage opportunity based on factors like trading fees and transfer costs.

### 2. **Connecting Events and Commands**
Domain services are often responsible for handling commands and triggering domain events. In event storming, this becomes evident when commands are issued (represented as blue sticky notes), but the logic for handling those commands is not the responsibility of a specific entity. The command may trigger multiple domain events, and a domain service coordinates this logic.

   - **Example**: After the `PriceDifferentialDetected` event, a domain service
```

might handle the command `SubmitBuyOrder` by orchestrating the interaction between different exchanges, issuing buy/sell commands, and triggering further events like `BuyOrderPlaced` or `SellOrderPlaced`.

### 3. **Handling Cross-Cutting Concerns**

Domain services are ideal for dealing with cross-cutting concerns, such as integrating external systems or managing policies. In event storming, these are often represented with separate colors (e.g., pink sticky notes for external systems or purple for policies). Domain services might be invoked when a command triggers a sequence that involves multiple bounded contexts or interactions with external systems.

- **Example**: A domain service might handle the transfer of cryptocurrency between wallets after an arbitrage trade is completed. The event `OrderExecutionConfirmed` would trigger a service responsible for managing fund transfers across exchanges, ensuring the successful completion of the entire process.

### 4. **Represented in Ubiquitous Language**

Like domain events and commands, domain services are expressed using the **Ubiquitous Language** of the business. They should be understandable to both business and technical stakeholders. In event storming, participants will naturally describe processes like "calculating profit," "checking liquidity," or "transferring funds," which can be mapped to domain services.

- **Example**: A service named `CalculateProfit` could emerge during the event storming session as a natural part of the arbitrage workflow, triggered by the `OrderExecutionConfirmed` event to compute the net profit from a successful trade.

### Capturing Domain Services in Event Storming

1. **Place Domain Services Between Events and Commands**: In the visual representation of event storming, domain services appear as handlers between events and commands. They may not have a clear actor (no specific domain entity), but they connect the business logic across multiple steps.

2. **Use Clear Names in the Ubiquitous Language**: Domain services should be named using the domain's specific language, making them easy to understand for domain experts and developers alike. For example, `ArbitrageRiskEvaluator` or `FundTransferService` could be natural names in a cryptocurrency arbitrage system.

3. **Link Services to External Systems When Necessary**: If the service interacts with third-party systems (like cryptocurrency exchanges), this should be represented with a distinct marker or color in the event storming diagram. These external system interactions often require domain services to manage and trigger further events.

### Example: Event Storming of a Cryptocurrency Arbitrage Process

- **Event**: `PriceDifferentialDetected` (orange sticky note)
- **Domain Service**: `ArbitrageOpportunityEvaluator`
- **Command**: `SubmitBuyOrder` (blue sticky note)
- **Event**: `BuyOrderPlaced` (orange sticky note)
- **Domain Service**: `FundTransferService`
- **Event**: `SellOrderPlaced`

- **Domain Service**: `ProfitCalculationService`

In this example, the domain services (`ArbitrageOpportunityEvaluator`, `FundTransferService`, and `ProfitCalculationService`) manage complex logic between events, coordinating the necessary actions and ensuring that the business logic is correctly executed.

By capturing domain services during event storming, you ensure that complex processes are modeled in a way that's clear and decoupled, making it easier to evolve and maintain the system 【8†source】 【9†source】 【10†source】 【11†source】.

User

What is a domain service in DDD and how is it different from a bounded context?

ChatGPT

In **Domain-Driven Design (DDD)**, a **domain service** is a service that encapsulates domain-specific logic, but it doesn't naturally fit within a specific entity or value in the domain. It handles operations that involve more than one entity, require coordination between different parts of the system, or perform complex business logic that is crucial to the domain but isn't a responsibility of a single entity.

### Characteristics of a Domain Service:
1. **Contains Domain Logic**: A domain service is responsible for implementing business rules or processes that don't belong to a single entity but are still central to the domain.
2. **Stateless**: Domain services typically don't maintain state. Instead, they operate on entities or data passed to them and return the result of the operation.
3. **Performs Cross-Entity Operations**: When domain logic requires collaboration between multiple entities or actions that don't belong to any single entity, a domain service is used.
4. **Exists Inside a Bounded Context**: Domain services operate within a bounded context (the boundary of a business domain), meaning they deal with concepts specific to that context.

### Example of a Domain Service:
In a **cryptocurrency arbitrage system**, a domain service might calculate whether an arbitrage opportunity is profitable across multiple exchanges.

- **Service Name**: `ArbitrageProfitCalculator`
- **Responsibility**: It calculates the profit potential of an arbitrage trade by comparing prices across two exchanges and subtracting fees and transfer costs.
- **Operation**: It takes in exchange rates, fees, and transfer details and returns a profit estimate.

```haskell
type ArbitrageProfitCalculator = (PriceA, PriceB, Fees) -> Profit
```

### Bounded Context:
A **bounded context** defines the boundaries within which a particular domain model applies. It sets a clear limit on where certain terms, concepts, and models are valid. A bounded context helps prevent ambiguity by ensuring that each concept has a well-defined meaning within its boundary. It is essential for

dividing large, complex systems into manageable parts.

### Characteristics of a Bounded Context:
1. **Defines a Boundary**: A bounded context establishes a boundary within which certain domain concepts, models, and rules are valid. Outside of this boundary, those concepts may have different meanings or may not exist at all.
2. **Is a High-Level Organizational Unit**: It corresponds to a specific part of the business or a business subdomain, such as "Order Management" or "Trade Execution" in a cryptocurrency system.
3. **Contains Entities, Services, and Events**: Inside a bounded context, you’ll find domain entities, domain services, and events that are specific to that context.
4. **Manages Its Own Data and Logic**: A bounded context handles its own data models, ensuring that different parts of the system remain decoupled.

### Example of a Bounded Context:
In a **cryptocurrency arbitrage system**, you could have a bounded context for **Order Management**:
- This context would handle placing buy and sell orders, managing the lifecycle of an order, and tracking order status. All the logic and models related to order processing are defined within this boundary.

### Key Differences Between Domain Service and Bounded Context:
1. **Scope**:
   - **Domain Service**: Focuses on performing specific operations or logic within a bounded context, often involving multiple entities or actions.
   - **Bounded Context**: Defines a large boundary around a cohesive business function or subdomain. It contains all relevant entities, services, and rules.

2. **Responsibility**:
   - **Domain Service**: Its responsibility is limited to executing specific pieces of business logic that don’t fit well within an entity.
   - **Bounded Context**: Its responsibility is to organize and encapsulate the models and business logic of an entire part of the system.

3. **State**:
   - **Domain Service**: Typically stateless, only operating on the data passed to it.
   - **Bounded Context**: Manages state and data models specific to its domain, ensuring proper encapsulation and decoupling.

4. **Relation to Business Logic**:
   - **Domain Service**: Focuses on specific processes or operations. It is a functional component that performs tasks, often used within workflows.
   - **Bounded Context**: Encompasses all the business logic, data, and rules relevant to a particular business domain, often dividing the overall system into smaller, manageable sections.

### Example Distinguishing the Two:
In a cryptocurrency arbitrage system:
- **Domain Service**: `FundTransferService` handles transferring profits from one exchange to another. This service operates on entities like Wallets and Exchanges but isn’t tied to any single entity.
- **Bounded Context**: The **Arbitrage Execution** context includes all the logic for identifying opportunities, placing trades, and tracking profits, which might also use the `FundTransferService` internally.

In summary, a **domain service** performs specific operations or logic within the broader framework of a **bounded context**, which organizes and contains the entire domain logic of a business subdomain 【8†source】 【10†source】 【11†source】 .

User

I am a software engineering student and I already know basic concepts of Domain-Driven-Design and have completed the event storming activity with my teammates. Can you guide me on how to define bounded contexts in our cryptocurrencies automated trading system based on the outcome of event storming and guide me on how to classify the corresponding business domains into core/supportive/generic without suggesting a classification?

ChatGPT

Great! Since you've already completed the event storming activity, you're in an excellent position to start defining **bounded contexts** and classifying your business domains into **core**, **supportive**, or **generic** domains. I'll guide you step-by-step on how to do this based on the insights you've gathered during event storming.

### Step 1: Define Bounded Contexts

A **bounded context** in Domain-Driven Design (DDD) represents a well-defined boundary within which a specific domain model is valid. Your goal is to group related events, commands, and domain services that belong to a particular business process or domain area into a cohesive context.

Here's how to define bounded contexts for your **cryptocurrency automated trading system**:

#### 1. **Review Key Events and Business Processes**

Go back to your event storming session and review the key **domain events** you identified. These events often map to significant moments in your business processes. You should also look at the commands and services that were involved in each flow. Ask yourself:

- Which events seem closely related in terms of functionality?
- Which events trigger other events within the same process?

#### 2. **Group Related Events**

Group events that represent **closely related business processes** into a single bounded context. For example, in a cryptocurrency trading system, you might have events related to **order execution** (e.g., `BuyOrderPlaced`, `SellOrderPlaced`) and others related to **funds management** (e.g., `FundsTransferred`, `TransactionFailed`). These can be grouped into different bounded contexts if they represent distinct processes.

Example of potential bounded contexts:

- **Order Execution Context**: Manages the placing, tracking, and execution of buy and sell orders.
- **Market Data Context**: Handles receiving and processing live market data from exchanges (e.g., prices, volume).
- **Arbitrage Opportunity Detection Context**: Responsible for detecting profitable arbitrage opportunities based on market data.
- **Wallet and Funds Management Context**: Deals with cryptocurrency wallet balances, transfers between exchanges, and tracking funds.

#### 3. **Identify Different Business Rules or Models**
If an event’s meaning or behavior changes depending on the context, it indicates the need for multiple bounded contexts. For example, the concept of an "order" might have different meanings in the **Order Execution** context (where the order is processed) versus the **Risk Management** context (where the system checks whether orders meet the trading strategy’s risk profile).

#### 4. **Consider External Systems and Interactions**
If some parts of your system interact heavily with external systems (e.g., cryptocurrency exchanges or APIs), these interactions might form separate bounded contexts. For example, an **Exchange Integration Context** could handle the communication between your trading system and external exchanges, decoupling the trading logic from the specifics of each exchange's API.

#### 5. **Draw the Boundaries**
Once you have grouped related events, commands, and services, **draw the boundaries** around them. These boundaries define where models are valid and where different terms or rules apply. Each bounded context will have its own domain model, and events within one context may trigger actions in another context.

### Step 2: Classify Business Domains into Core, Supportive, and Generic

After defining your bounded contexts, you need to classify the business domains they represent into **core**, **supportive**, or **generic** subdomains. Here’s how to approach this classification without suggesting a specific one, but guiding you on how to think about it:

#### 1. **Core Subdomains**
These represent the **heart of your business**—the parts that give your system a competitive advantage. They are crucial to the business’s success and should be handled with great care and often custom development. Ask yourself:
- What are the key differentiators in your system?
- Which parts of the system, if done well, will provide a significant advantage over other automated trading systems?
- Which functionalities are closely aligned with the business's main value proposition?

In a cryptocurrency arbitrage system, think about which processes make your system unique, such as **arbitrage opportunity detection** or **trade execution optimization**.

#### 2. **Supportive Subdomains**
These domains are important to your business but **don’t provide direct competitive advantage**. They support the core subdomains, ensuring they function smoothly. Think about:
- Which processes are necessary for the core business to run efficiently but aren’t unique to your system?
- Where do you need specialized solutions, but these solutions are not directly tied to your business's core value?

Supportive domains might involve things like **funds management**, **risk assessment**, or **user authentication**—areas that need to work well but are not the central focus of your business.

#### 3. **Generic Subdomains**
These are **standardized parts** of the system that every company in your

industry has. They don’t require deep customization and are often implemented using off-the-shelf solutions. Ask yourself:
   - Which processes are standard across the industry and don’t offer differentiation?
   - What areas could be outsourced or bought as a service?

   In your case, functions like **exchange connectivity** (interfacing with third-party APIs) or **logging and monitoring** might fall into this category, since they are common in many trading systems.

### Step 3: Mapping the Bounded Contexts to Business Domains

Once you have defined your bounded contexts and classified your business domains, **map each bounded context to the appropriate domain**. For example:
- If you decide that your **Arbitrage Opportunity Detection Context** is a core domain, you’ll want to ensure it is built with high attention to detail, likely using custom logic that aligns tightly with your business strategy.
- On the other hand, a **Wallet and Funds Management Context** might be supportive or generic, meaning you could consider using standard libraries or external services.

### Summary Checklist

1. **Review your event storming outcome** to identify related events and processes.
2. **Group related events, commands, and services** into **bounded contexts**.
3. **Draw clear boundaries** around different parts of your system where domain models apply.
4. **Classify each bounded context** into core, supportive, or generic business domains.
5. **Map each bounded context** to its appropriate domain classification to determine how much effort should go into its development (e.g., custom vs. standardized solutions).

By following these steps, you will gain a clear understanding of your system’s structure and how to prioritize its development based on the business impact of each bounded context 【8†source】 【9†source】 【10†source】 【11†source】.

User

Generate statistics and analysis based on our exchange. Please provide the statistics in table format:
Table 1
Column 1: Number of additional questions I asked, excluding this prompt asking for analysis and the conversation starters: "Can you walk me through the basic financial terms (names) used in cryptocurrency arbitrage? What is the relationship between them? E.g. how is arbitrage opportunity related to an order and a trade and a transaction?" and "can you walk me through typical systems and actors that are part of cryptocurrency arbitrage and in particular, an automated trading system? Which are not relevant in the context of the attached project specification?".
Column 2: The number of questions asked per prompt - make sure not to account for other types of text (e.g. conversation elements like thank you, exclamations etc.). Whenever multiple questions were included in the same prompt, account for them separately in the table.

Table 2: An analysis of our conversation

Column 1: prompt
Column 2: question I asked
Column 3: classification of the nature of the questions asked: about getting a baseline understanding provided in your answers to the prompt/about exploring the knowledge presented by you/other?
Column 4: classification of the complexity and difficulty of questions asked: basic, intermediate, and advanced levels.
Column 5: gauge sentiment and feedback regarding LLM interactions by classifying it into occurrences of positive, neutral, and negative sentiments in feedback. Also detect and specific feedback themes (e.g., helpfulness, clarity).

ChatGPT

Here are the requested statistics and analysis based on our conversation:

```
### Table 1: Number of Questions Asked
| **Number of additional questions asked** | **Number of questions per prompt** |
|------------------------------------------|-------------------------------------|
| 5                                        | 6                                   |

### Table 2: Analysis of the Conversation
| **Prompt**
| **Question**
| **Classification**                           | **Complexity** | **Sentiment** |
|---------------------------------------------------------------------------
------------------------------------|---------------------------------------
---------------------------------------------------------------------|-------
-----------------------------------|----------------|---------------|
| Can you give me some examples of how to name events in an automated
cryptocurrency arbitrage system?                  | Can you give me some examples
of how to name events in an automated cryptocurrency arbitrage system?
| about exploring the knowledge             | intermediate   | positive      |
| In an automated cryptocurrency arbitrage system, give an example of a domain
process                                  | give an example of a domain process
| about exploring the knowledge             | intermediate   | neutral       |
| how are domain services captured during event storming
| how are domain services captured during event storming
| about exploring the knowledge             | advanced       | positive      |
| What is a domain service in DDD and how is it different from a bounded context?
| What is a domain service in DDD and how is it different from a bounded context?
| about getting a baseline understanding    | basic          | neutral       |
| I am a software engineering student and I already know basic concepts of
Domain-Driven-Design and have completed the event storming activity with my
teammates. Can you guide me on how to define bounded contexts in our
cryptocurrencies automated trading system based on the outcome of event storming
and guide me on how to classify the corresponding business domains into
core/supportive/generic without suggesting a classification? | Can you guide me
on how to define bounded contexts in our cryptocurrencies automated trading
system? How to classify business domains into core/supportive/generic? | about
exploring the knowledge | advanced | positive |
```

This analysis shows the progression of questions, focusing on deepening understanding of concepts related to DDD and cryptocurrency arbitrage. The feedback sentiment has been generally positive, with questions ranging from basic to advanced complexity.

User

Can you provide an example of a supportive subdomain and a generic subdomain

ChatGPT

In Domain-Driven Design (DDD), subdomains are classified based on their importance to the core business. Here's how **supportive** and **generic** subdomains work, along with examples from a cryptocurrency automated trading system.

### 1. **Supportive Subdomain**
A **supportive subdomain** helps the core business function but doesn’t directly offer competitive advantage. It's crucial for ensuring the core processes work efficiently, but the logic is specific to the business, so it often requires custom development.

#### Example: **Risk Management Subdomain**
In a cryptocurrency automated trading system, **risk management** is a supportive subdomain. It helps manage and mitigate the risks involved in trading, such as avoiding excessive losses, monitoring liquidity, or applying position limits. While crucial to ensure the safety of trades, risk management itself doesn’t provide direct competitive advantage—it’s something all trading systems must do.

- **Key Processes**:
  - Monitoring price volatility
  - Setting risk thresholds for trades
  - Applying stop-loss mechanisms
  - Avoiding exposure to illiquid markets

- **Bounded Context Example**: You might define a **Risk Management Context** that contains logic for assessing the risk of each arbitrage opportunity before executing a trade.

### 2. **Generic Subdomain**
A **generic subdomain** includes processes that are not specific to the business and can be found in many systems. These subdomains are often common across multiple industries and can often be implemented using off-the-shelf solutions or standardized libraries.

#### Example: **Logging and Monitoring Subdomain**
In the same cryptocurrency trading system, **logging and monitoring** is a generic subdomain. It involves tracking system performance, logging system events, and ensuring that everything runs smoothly from a technical perspective. This subdomain doesn’t need to be customized because standard solutions (like logging libraries or cloud monitoring tools) can handle it effectively.

- **Key Processes**:
  - Logging trade execution and system activities
  - Monitoring system health (CPU usage, memory, network activity)
  - Sending alerts for system failures or unusual behaviors

- **Bounded Context Example**: A **System Monitoring Context** could be defined for this subdomain, where you would integrate existing libraries or tools like Elasticsearch, Grafana, or CloudWatch for logging and monitoring.

### Summary:
- **Supportive Subdomain** (custom-built, helps core processes): **Risk Management** ensures the system operates within safe parameters but isn’t a

differentiator.
- **Generic Subdomain** (off-the-shelf, standardized): **Logging and Monitoring** tracks system performance and health, a common need across many types of systems.

Both subdomains are essential, but only supportive subdomains usually need more customization specific to the business's needs 【9†source】 【10†source】 【11†source】 .

User

How can I identify workflows in a subdomain?

ChatGPT

Identifying **workflows** within a subdomain is a crucial step in Domain-Driven Design (DDD) because it helps to map out the key processes that transform data, trigger domain events, or involve multiple actions across different components of the system. Workflows represent how the system handles business processes from start to finish.

Here's a step-by-step guide to identifying workflows within a subdomain:

### 1. **Analyze the Subdomain’s Key Events and Business Processes**
Start by reviewing the key **domain events** and business processes that are specific to the subdomain. Events provide insights into the state changes and important occurrences in the system. Look for processes that naturally follow a sequence of steps.

- **Example**: In a **Risk Management** subdomain, some key events might include `RiskThresholdExceeded` or `TradeRejectedDueToRisk`. These events signal that workflows exist for evaluating risk and deciding whether a trade can proceed.

### 2. **Map Commands to Events**
Workflows are often triggered by **commands**, which lead to one or more **domain events**. By analyzing the flow from commands to events, you can identify sequences of actions that constitute a workflow.

- **Example**: In a **Risk Management** subdomain:
  - Command: `EvaluateRiskForTrade`
  - Event: `RiskThresholdExceeded` or `RiskThresholdNotExceeded`

  This sequence of actions forms a workflow that checks the risk level of a trade and either approves or rejects it.

### 3. **Break Down the Business Process into Steps**
Workflows typically involve a series of steps or **transformations**. Breaking the process down into smaller tasks will help you identify the specific actions the workflow must handle. These tasks may involve interactions between different entities, domain services, or external systems.

- **Example**: A workflow for **evaluating risk** in a cryptocurrency system could include:
  1. Receive the trade details (command)
  2. Fetch the trader’s risk profile
  3. Calculate the potential risk exposure for the trade
  4. Compare against predefined risk thresholds
  5. Trigger the appropriate event: `TradeApproved` or `TradeRejectedDueToRisk`

### 4. **Look for State Transitions**
A workflow often results in a change of **state** within the subdomain. Look for points where a domain entity or process moves from one state to another (e.g., from "Pending" to "Completed"). State transitions are a good indication of where workflows exist.

- **Example**: In a **Funds Management** subdomain, the state of a wallet might change from `SufficientBalance` to `InsufficientBalance` based on the outcome of a workflow that checks if funds are available to execute a trade.

### 5. **Collaborate with Domain Experts**
Since DDD emphasizes collaboration with domain experts, it's critical to involve them when identifying workflows. Domain experts can help clarify how real-world processes work, which can guide you in understanding how workflows in the system should behave.

- **Example**: A domain expert in cryptocurrency trading could explain how trades are approved or rejected based on risk assessments, which will help you model that as a workflow in the **Risk Management** subdomain.

### 6. **Use Event Storming Outputs**
If you have already completed an **event storming** activity, you can use the sticky notes that represent **commands, domain events, and policies** to guide your understanding of workflows. In event storming:
   - Blue sticky notes represent **commands** (imperative actions).
   - Orange sticky notes represent **domain events** (something that has happened).
   - Purple sticky notes represent **policies** (automatic processes or conditions triggered by events).

Look for clusters of events

User

Please give feedback on my project below:

```
workflow "Subscribe to real-time data feed" =


input = TradingActivityLaunched event
output = RealTimeDataCollected event


@(Database)
do getCryptoCurrencyPair
   @(Polygon.io)
   do collectDataFromExchange
       send RealTimeDataCollected event
   stop
stop


event TradingActivityLaunched = UserInputSignal
event RealTimeDataCollected = PriceQuote list
```

```
type PriceQuote = {CurrencyF; PriceF; CurrencyT; PriceT}




workflow "Process real-time feed for cryptocurrency pairs" =


input = PriceQuoteReceived event
output = PriceSpreadChecked event


@(Database)
do getMinPriceSpreadVal
   do checkPriceSpreadExist
       send PriceSpreadChecked event
   stop
stop


event PriceQuoteReceived = PriceQuote
event PriceSpreadChecked = true OR false




workflow "Apply a trading strategy" =


input = PriceSpreadVerified event
output = ArbitrageOpportunityEvaluated event


@(Database)
do getMinProfit getMaxTotTxnVal getMaxTradeVal
   do evalArbitrageOpportunity
       send ArbitrageOpportunityEvaluated event
   stop
stop


event PriceSpreadVerified = PriceQuote
event ArbitrageOpportunityEvaluated = true OR false




workflow "Emit buy and sell market orders" =


input = ArbitrageOpportunityIdentified event
```

```
output = TradeExecuted event


@(Exchange)
do emitBuyOrder
   @(Exchange)
   do emitSellOrder
       if AvailQtyNotUtilized then:
           send LeftoverQtyRetained event
       else if MaxTotTxnValExceeded OR MaxTradeValExceeded:
           send OrderQtyReduced event
       else:
           send TransactionFinished event



event ArbitrageOpportunityIdentified = PriceQuote AND TradeStrategy
event TradeExecuted = OrderRes


type OrderRes = LeftoverQtyRetained OR OrderQtyReduced OR TransactionFinished




workflow "Persist fully fulfilled orders" =


input = OrderFullyFulfilled event
output = TransactionPersisted event


do invokeDBService
   send TransactionPersisted event
stop


event OrderFullyFulfilled = OrderID AND FulfillmentDetails
type FulfillmentDetails = Filled OR PartiallyFilled OR OnlyOneSideFilled OR
NotFilled
event TransactionPersisted = true OR false




workflow "Emit more orders for partially fulfilled orders" =


input = OrderPartiallyFulfilled event
output = TransactionPersisted event


do emitOrders
   do invokeDBService
       send TransactionPersisted event
   stop
stop
```

```
event OrderPartiallyFulfilled = OrderID AND FulfillmentDetails
type FulfillmentDetails = Filled OR PartiallyFilled OR OnlyOneSideFilled OR
NotFilled
event TransactionPersisted = true OR false



workflow "Notify user when one side of an order filled" =


input = OrderOneSideFilled event
output = ConfirmationSent event


do invokeSMTPService invokeDBService
   send ConfirmationSent event
stop


event OrderPartiallyFulfilled = OrderID AND FulfillmentDetails
type FulfillmentDetails = Filled OR PartiallyFilled OR OnlyOneSideFilled OR
NotFilled
event ConfirmationSent = true OR false



workflow "Send email notifications to the user" =


input = EmailContentProvided event
output = NotificationSent event


@(Email)
do sendEmailNotification
   do checkIfNotificationSent
       send NotificationSent event
   stop
stop


event EmailContentProvided = OrderID and EmailContent
event NotificationSent = SMTPRes


type SMTPRes = Success OR Failure



workflow "Store finished transactions" =
input TransactionProvided event
output TransactionPersisted event


@(Database)
do persistTransaction
   send TransactionPersisted event
stop


event TransactionProvided = OrderId AND PriceQuote
event TransactionPersisted = CrudRes


type CrudRes = Success OR Failure




workflow "Provide trading configuration" =


input TradingConfigurationsProvided event
output TradingConfigurationsPersisted event


@(Database)
do persistTradingConfigurations
   send TraderConfigurationsPersisted event
stop


event TradingConfigurationsProvided =
   EmailProvided &
   MinimalTransactionProfitProvided &
   MinimalPriceSpreadValueProvided &
   NumberOfCryptoCurrenciesToTrackProvided &
   MaximalTransactionValueProvided &
   MaximalTradingValueProvided

event TradingConfigurationsPersisted = CrudRes

type CrudRes = Success OR Failure




workflow "Launch Real-Time Trading Activity" =


input RealTimeTradingActivityLaunched event
output UserStartSignalInitiated event


event RealTimeTradingActivityLaunched =
   true OR false
```

```
event UserStartSignalInitiated =
   true OR false




workflow "Calculate Historical Arbitrage Opportunities" =


input CrossTradedCryptoCurrenciesIdentified event
output ArbitrageOpportunitiesPersisted event


event CrossTradedCryptoCurrenciesIdentified =
   currencyPair

event ArbitrageOpportunitiesPersisted =
   CrudRes

type currencyPair =
   curr1 & curr2




workflow "Identify cross-traded cryptocurrencies" =


input = CrossTradedCryptoCurrenciesRetrieved event
output = CrossTradedCryptoCurrenciesPersisted event


@(Database)
do persistTradingConfigurations
   send CrossTradedCryptoCurrenciesPersisted event
stop


@(CryptocurrencyExchange)
do retrieveCrossTradedCryptoCurrencies =
   If crossTradedPairsFound then:
       send CrossTradedCryptoCurrenciesRetrieved event
stop


event CrossTradedCryptoCurrenciesRetrieved = crossTradedPairs
event CrossTradedCryptoCurrenciesPersisted = crossTradedPairsFound




workflow "Terminate Monitor" =
input = RealTimeTradingStopped event


event RealTimeTradingStopped = Cause
```

type Cause = manuallyStopped OR conditionsMet

ChatGPT

Your project demonstrates a well-structured approach to defining workflows for an automated cryptocurrency trading system. Here's feedback on key aspects, including clarity, consistency, and potential areas for improvement:

### 1. **Workflow Definition and Structure**

- **Strengths**: Each workflow has a clear input and output event, which makes the flow of data and state transitions easy to follow. You effectively define actions, side effects (e.g., interacting with external systems), and termination points, providing a coherent mapping of how the system operates.
- **Areas for Improvement**:
  - **Consistency in Terminology**: Some events could benefit from more consistent naming. For example, in the "Notify user when one side of an order filled" workflow, the event name `OrderOneSideFilled` is not consistent with the input event name in other workflows like `OrderPartiallyFulfilled`. A more uniform approach (e.g., `OrderPartiallyFulfilled`, `OrderOneSideFilled`) would improve readability.
  - **Clarify Actions**: In workflows like **"Emit buy and sell market orders"**, you include conditional logic (`if AvailQtyNotUtilized`). It might help to define such logic more explicitly, possibly separating it into smaller workflows or breaking it into different steps to maintain the functional purity of each workflow step.

### 2. **Use of Domain Events**

- **Strengths**: You follow Domain-Driven Design principles well by modeling events like `PriceSpreadChecked`, `TradeExecuted`, and `TransactionPersisted` that clearly reflect what happened in the past. This aligns with best practices, ensuring that events represent completed actions.
- **Areas for Improvement**:
  - **Event Types**: The use of types like `CrudRes = Success OR Failure` is good for capturing the result of operations, but for better domain understanding, you might add more contextual detail to events. For example, instead of just `Success` or `Failure`, you could include additional metadata (e.g., failure reasons) to give more insight into what went wrong.

### 3. **Domain Logic and Separation of Concerns**

- **Strengths**: You have done well in ensuring workflows are focused on their respective responsibilities (e.g., persisting transactions, applying trading strategies). The separation of logic and actions between databases, exchanges, and user interactions is clear and makes the system easy to maintain.
- **Areas for Improvement**:
  - **Complex Workflows**: Some workflows, like **"Emit buy and sell market orders"**, are somewhat dense, combining multiple conditions and actions. To improve modularity and testability, you might break down these workflows further into simpler sub-workflows. For instance, handling leftover quantity or reducing order quantity could each be separate workflows triggered by specific events.

### 4. **Integration with External Systems**

- **Strengths**: You clearly mark points of integration with external systems like `Polygon.io` for data collection and exchanges for order execution, ensuring that side effects are well-identified.
- **Areas for Improvement**:
  - **Error Handling for External Systems**: While you define events for

success and failure (`SMTPRes = Success OR Failure`), it might be beneficial to define more specific failure events, especially when interacting with external systems. For example, `Polygon.ioUnavailable` or `ExchangeTimeout` could be defined to handle external system failures in a more granular way.

### 5. **Event-Driven Design**

- **Strengths**: You follow a clear event-driven approach, with each workflow responding to domain events and producing new events as outputs. This aligns well with DDD principles, making the system reactive and extensible.
- **Areas for Improvement**:
  - **Event Granularity**: Some events, like `TransactionPersisted` and `ArbitrageOpportunityEvaluated`, could benefit from more granular data. Consider providing additional contextual information with events to help other parts of the system make more informed decisions (e.g., the reason for failing to evaluate an arbitrage opportunity).

### 6. **Final Thoughts**

- **Strengths**: The project demonstrates a solid understanding of workflows, event-driven architecture, and external integrations. You have clearly outlined the core aspects of your system, which makes it easier to maintain and extend.
- **Next Steps**: Focus on refining error handling, improving event detail, and breaking down complex workflows. This will help in making the system even more resilient and modular.

In summary, you’ve done a great job with your project by organizing the workflows logically and adhering to domain-driven design principles. A bit more focus on modularity and error handling will make it even more robust.

User

For example, In my system design I want to use a match statement pseudo code to show how different events are handled. How can I avoid using F# pseudo code in my system design? The following is my system design pseudo code for a particular workflow:

```
if AvailQtyNotUtilized then:
           send LeftoverQtyRetained event
       else if MaxTotTxnValExceeded OR MaxTradeValExceeded:
           send OrderQtyReduced event
       else:
           send TransactionFinished event
```

ChatGPT

To avoid using F#-style pseudo code in your system design while keeping your logic simple and readable, you can adopt a **functional programming-friendly style** while sticking to a more generic pseudocode format. Here’s how you can represent your conditional logic using a **pattern-matching-like structure** that doesn’t depend on F# syntax:

### Refined Pseudo Code Example:

```plaintext
match condition with
   | AvailableQuantityUnused -> send LeftoverQtyRetained event
   | TotalTransactionValueExceeded OR TradeValueExceeded -> send OrderQtyReduced event
```

```
    | otherwise -> send TransactionFinished event
```

However, since you want to avoid using F#-style pseudo code, you can translate this into a more **general-purpose pseudo code** using a simpler `if-else` or `match-case` structure, which can be easily understood across various languages:

### General Purpose Pseudo Code (No F#)

```plaintext
if AvailQtyNotUtilized:
    send LeftoverQtyRetained event
elif MaxTotTxnValExceeded OR MaxTradeValExceeded:
    send OrderQtyReduced event
else:
    send TransactionFinished event
```

### Explanation:
- **if-elif-else**: This format is simple and widely recognizable. It avoids specific language constructs like `match` or `|`, making it more universal.
- **Conditions**: Each condition (`AvailQtyNotUtilized`, `MaxTotTxnValExceeded`, etc.) remains readable and represents checks that are straightforward.
- **Events**: The outcome of each condition is the sending of an event, which can be represented using `send`.

### Alternative: Match-like Structure in General Pseudo Code

If you prefer a **match-like structure** without the F# syntax, you can use a general pseudo code format as follows:

```plaintext
match condition:
    case AvailQtyNotUtilized:
        send LeftoverQtyRetained event
    case MaxTotTxnValExceeded OR MaxTradeValExceeded:
        send OrderQtyReduced event
    default:
        send TransactionFinished event
```

### Key Differences:
- **General match-case**: This representation is closer to `switch-case` in other programming languages, while still conveying the logic clearly.
- **No F# Syntax**: This avoids F# constructs and remains more neutral, so the logic can be applied to any language or framework you're using in your system design.

Both approaches ensure your system design is clear, concise, and avoids the F#-specific pseudo code, making it more adaptable across different implementations.

User

Provide an example of a domain error in a system design for Crypto Arbitrage

ChatGPT

In a **Crypto Arbitrage** system, domain errors represent situations where business rules or domain-specific conditions are violated, preventing a successful outcome. Domain errors differ from infrastructure errors (like network failures) in that they stem from issues related to the logic of the business domain itself.

### Example of a Domain Error: **InsufficientFundsError**

#### Scenario:
In a crypto arbitrage system, a trader wants to execute a buy order, but the available funds in the trader's account are insufficient to cover the cost of the trade. This scenario triggers a **domain error** because one of the business rules—ensuring sufficient funds are available for any trade—has been violated.

### Pseudo Code for Handling Domain Error:
Here's an example system design with a domain error handling logic for **InsufficientFundsError**:

```plaintext
workflow "Place Buy Order" =

input = ArbitrageOpportunityIdentified event
output = BuyOrderPlaced OR InsufficientFundsError event

@(Database)
do checkFundsAvailability
    if InsufficientFunds:
        send InsufficientFundsError event
    else:
        send BuyOrderPlaced event
stop


event ArbitrageOpportunityIdentified = PriceQuote AND TradeStrategy
event BuyOrderPlaced = OrderID AND OrderDetails
event InsufficientFundsError = TradeDetails AND RequiredFunds AND AvailableFunds

type TradeDetails = {
    CurrencyPair
    TradeVolume
    TradePrice
}
```

### Breakdown:
- **Event**:
   - `ArbitrageOpportunityIdentified`: This event indicates that a potential arbitrage trade has been identified, triggering the process to place a buy order.
- **Action**:
   - The system checks whether sufficient funds are available using the `checkFundsAvailability` function.
   - If the funds are **insufficient**, it raises the domain error `InsufficientFundsError`.
   - If funds are sufficient, the system continues by placing the buy order (`BuyOrderPlaced`).
- **Error Event**:
   - `InsufficientFundsError` contains details about the trade that failed, the

required funds, and the available funds. This provides useful feedback to the system or the user for further action.

### Explanation of the Domain Error:
- **InsufficientFundsError** is a **domain error** because it directly results from a business rule violation—the rule that a trade can only be executed if there are sufficient funds.
- The error does not result from technical issues like network failures or timeouts but from a **logical constraint** within the crypto arbitrage domain. This makes it a **business process failure** rather than a technical one.

### Why Domain Errors Are Important:
Handling domain errors like `InsufficientFundsError` ensures that the system respects business rules and provides meaningful feedback when conditions are not met. It helps prevent invalid actions from proceeding and keeps the system in a valid state.

Other examples of domain errors in crypto arbitrage could include:
- **MaxTradeLimitExceededError**: When a trade exceeds a predefined maximum trade value.
- **InvalidCryptoPairError**: When the trading pair does not exist or cannot be traded between the selected exchanges.

By clearly defining and handling domain errors, you ensure your system behaves predictably and respects the rules of the business domain.

User

Please give feedback on my attached project:

Tool

All the files uploaded by the user have been fully loaded. Searching won't provide additional information.

ChatGPT

Here’s my feedback on your **Crypto Arbitrage System Design** based on the project document you provided:

### 1. **Bounded Contexts and Domain Classification**

#### **Strengths:**
- You've clearly identified **core, supportive, and generic domains**, which is a good practice in Domain-Driven Design (DDD). For example, the **Cross-Traded Currency Context** and **Trading Strategy Context** being part of the core domain makes sense, as they are central to the arbitrage process.
- The **User Settings Context** and **User Signal Context** being classified as generic subdomains is logical, as they support the system without offering a competitive edge.

#### **Areas for Improvement:**
- **Bounded Context Scope**: Some of the contexts could be further refined to ensure they are not too broad. For instance, the **Arbitrage Opportunity Context** might benefit from being split into smaller bounded contexts, such as separating the identification from execution for better clarity and maintainability.
- **Interdependencies**: More detail could be provided on how the contexts

interact. You have outlined the internal workings, but describing how contexts like **Post-Order Processing** interact with core domains like **Arbitrage Opportunity** could help improve the understanding of boundaries and responsibilities.

### 2. **Events, Commands, and Domain Errors**

#### **Strengths:**
- The **domain events** and **commands** are well-structured and align well with the workflows you have designed. Events like `CrossTradedCryptoCurrenciesRetrieved` and `ArbitrageOpportunityEvaluated` are good representations of real-world business events.
- You have clearly defined **domain errors** like `InvalidParameterError` and `InsufficientFunds`, which are crucial for handling business rule violations. This ensures your system has a robust error-handling mechanism for domain-specific problems.

#### **Areas for Improvement:**
- **Event Granularity**: Some of the events could benefit from more detailed data. For example, in `ArbitrageOpportunityEvaluated`, you could include more context about **why** an opportunity was not evaluated (e.g., insufficient spread or market condition changes).
- **Command-Event Separation**: Ensure that commands and events are distinct in their purpose. For instance, commands should be imperative ("do something"), while events should be declarative ("something happened"). This distinction is important for maintaining a clean DDD design.

### 3. **Workflows**

#### **Strengths:**
- The workflows are well-documented, and you have followed a consistent structure for defining inputs, outputs, and actions. For example, in the **"Emit buy and sell market orders"** workflow, you detail the conditional logic around sending different events (e.g., `LeftoverQtyRetained`, `OrderQtyReduced`).
- You effectively make use of **external systems** such as **Polygon.io** and **Cryptocurrency Exchanges** within your workflows, clearly showing where side effects occur.

#### **Areas for Improvement:**
- **Workflow Complexity**: Some workflows could be simplified or broken down further. For instance, the **"Emit buy and sell market orders"** workflow includes multiple conditions (e.g., handling leftover quantities, exceeded transaction values) that could be split into smaller sub-workflows to improve readability and modularity.
- **Handling Failures**: In the workflow where market orders are emitted, you handle failures like insufficient funds. Consider also adding timeouts and retries for external system interactions, especially with exchanges, to handle potential network or API failures robustly.
- **Flow Transitions**: It would be helpful to show the transitions between workflows in a more explicit manner. For example, after an arbitrage opportunity is identified, is the next step to automatically trigger the buy and sell orders? Or does it require user confirmation? Mapping these transitions could help clarify the flow.

### 4. **Domain Errors**

#### **Strengths:**
- You’ve captured relevant domain errors such as `InvalidOrderError`, `InsufficientInventoryError`, and `SystemTimeoutError`. These ensure that the system can handle typical edge cases and failures in cryptocurrency trading effectively.

#### **Areas for Improvement:**
- **Error Handling in Workflows**: While the errors are well-defined, more detail on how these errors are handled at the workflow level could improve clarity. For instance, if `PriceQuoteRetrievalFailure` occurs, what is the system’s next step? Does it retry, fail gracefully, or alert the user?
- **Granularity of Errors**: Some errors, like `OrderNotFilled`, could include more detailed reasons for failure, such as specific market conditions or exchange limitations. This could improve debugging and help the system make better automated decisions.

### 5. **System Design Clarity**

#### **Strengths:**
- The system design as a whole is clearly structured and well-thought-out, with strong adherence to DDD principles. The use of **bounded contexts**, **domain events**, and **workflows** shows a solid understanding of how to design a complex system like crypto arbitrage.

#### **Areas for Improvement:**
- **Additional Diagrams**: Including flow diagrams for your workflows and bounded contexts would help to visualize the interactions between contexts, events, and external systems. While the system's functionality is clear, diagrams could further enhance understanding.
- **Event Naming**: Some events, like `TransactionFinished`, could benefit from more specific names to reflect the exact nature of the event. For example, `TransactionCompletedSuccessfully` might better convey the finality of the transaction.

### 6. **External Systems**

#### **Strengths:**
- You have clearly identified key external systems (e.g., **Polygon.io**, **Cryptocurrency Exchanges**, and **SMTP Service**), which shows a good understanding of the system's external dependencies.

#### **Areas for Improvement:**
- **Error Handling with External Systems**: More explicit details on how the system handles failures or delays from external systems would improve robustness. For example, if the **SMTP Service** fails to send an email, will there be a retry mechanism? Similarly, how does the system handle exchange downtime?

### Conclusion:
Your system design is well-structured and follows **Domain-Driven Design (DDD)** principles effectively. It provides a clear outline of bounded contexts, workflows, and domain events. With some refinements—such as breaking down complex workflows, improving event granularity, and handling external system errors—you can improve clarity and robustness.

ChatGPT

Based on the rubric provided, here's the grading breakdown for your **Crypto Arbitrage System Design**:

### **1. Completeness (5 Points)**

Criteria: The documentation should cover all DDD elements requested, workflows should meet the integrity of functional requirements, and data structures should be comprehensive and aligned with workflows. Interactions with external systems must be depicted with annotations.

- **Feedback**:
  - Your design demonstrates a solid understanding of Domain-Driven Design (DDD) principles and workflows. Events, workflows, and external interactions are defined and represented.
  - You have correctly included data structures such as `PriceQuote` and `OrderRes`, which are well-integrated with the workflows.
  - External systems like **Polygon.io** and cryptocurrency exchanges are referenced and annotated with `@annotation`.
  - There could be more detail about edge cases and failure scenarios across workflows.

Grade: **4.5/5**

### **2. Coherence & Correctness (3.5 Points)**

Criteria: The bounded contexts should be clearly defined, cohesive, and aligned with the project scope. Events should be logically organized between contexts, and workflows structured as pipelines of processing steps with correct handling of inputs and outputs. The design should trace end-to-end scenarios without issues or corner cases.

- **Feedback**:
  - Your bounded contexts are mostly coherent and align with the crypto arbitrage use case. Contexts like **Cross-Traded CryptoCurrencies Context** and **Arbitrage Opportunity Context** are defined logically.
  - The workflows are well-structured, processing events correctly from start to stop. However, more attention to transitions between workflows and tracing complete end-to-end scenarios would improve clarity.
  - Some workflows (like "Emit buy and sell market orders") could benefit from further simplification or breaking down into smaller, clearer sub-workflows.
  - The design does handle edge cases (e.g., partially filled orders) but could detail how certain failure scenarios are resolved across different bounded contexts.

Grade: **3/3.5**

### **3. Clarity (1.5 Points)**

Criteria: Design elements should align with DDD principles and employ ubiquitous language. Naming conventions should be clear, and the format should make the design easy to navigate.

- **Feedback**:
  - Your design uses DDD principles effectively and adheres to **ubiquitous language** with terms like **"Arbitrage Opportunity"** and **"TransactionPersisted"**.
  - Workflow descriptions and events are named clearly, leaving little room for ambiguity.
  - The document is formatted logically and is easy to follow. Adding more diagrams to visualize the workflows and bounded context interactions could further enhance clarity.

Grade: **1.5/1.5**

### **Overall Grade**: **9/10**

Your design is comprehensive and well-structured, showing a good understanding of DDD concepts, workflows, and external system interactions. Improving edge case handling and adding diagrams for workflow visualization could push your design to the next level. Keep refining these elements, and your project will meet all the evaluation criteria thoroughly.

Event Naming for Arbitrage

User

Can you give me some examples of how to name events in an automated cryptocurrency arbitrage system?

Tool

All the files uploaded by the user have been fully loaded. Searching won't provide additional information.

ChatGPT

In an automated cryptocurrency arbitrage system, the events should reflect significant occurrences in the domain and be named using past tense, as per event-naming conventions in Domain-Driven Design (DDD)【8†source】【9†source】. Here are some examples of event names that could be used:

1. **PriceDifferenceDetected** – Occurs when the system identifies a profitable

price difference across exchanges.
2. **ArbitrageOpportunityIdentified** – Triggered when the system confirms a viable arbitrage trade.
3. **BuyOrderExecuted** – Fires after successfully purchasing cryptocurrency on one exchange.
4. **SellOrderExecuted** – Fired after successfully selling cryptocurrency on another exchange.
5. **FundsTransferredToExchange** – Marks the moment when funds are moved to an exchange to initiate trading.
6. **InsufficientLiquidityDetected** – Triggered when a planned trade can't proceed due to low liquidity.
7. **TradeSettlementCompleted** – Occurs after both buy and sell orders have been confirmed and settled.
8. **TradeRolledBack** – Triggered if an issue arises (e.g., price fluctuations) requiring a rollback of the trade.
9. **ArbitrageCycleCompleted** – Indicates the successful completion of an entire arbitrage cycle.
10. **TransactionFeesUpdated** – Fired when the system recalculates transaction fees for trades.

These names align with the conventions from DDD: they describe something that has already happened and are concise and meaningful within the context of cryptocurrency arbitrage 【9†source】 【10†source】.

User

I am a software engineering student, please explain to me in simple words what is a bounded context.

ChatGPT

A **bounded context** is like a "boundary" around a specific part of a software system where certain terms and concepts have a precise meaning. Inside that boundary, everyone (developers, business experts) agrees on what things mean, and the system's behavior is designed based on those meanings.

Let’s break it down:

1. **Imagine a large project** like an online marketplace. It has different parts: user accounts, product catalogs, payments, reviews, etc. Each part of the system can be very complex and have its own rules and language.

2. **In a bounded context,** we focus on one part, like "Payments." Inside the Payments bounded context, terms like *"transaction"* or *"account balance"* have specific meanings. These terms might mean different things in other parts of the system, but here they are clearly defined.

3. **Boundaries help keep things organized.** They make sure that one part of the system doesn’t get confused by how other parts work. For example, the way you handle payments shouldn't interfere with how you manage product reviews. They are different bounded contexts.

In simpler terms, a bounded context acts like a container that helps to keep the different parts of a system clean and separated, so each part can work independently without creating confusion. It’s a way to make sure different areas of the software don't overlap in confusing ways 【8†source】 【10†source】.

### Analogy
Imagine a large university where there are different departments like Engineering, Medicine, and Arts. In each department, the term "course" could mean different things. In Engineering, a course might be a technical subject, while in Medicine, it could mean a series of practicals. A bounded context is like each department having its own building and language that makes sense to its students and teachers, but might not make sense to others.

User

Can you give an example of a bounded context in an automated cryptocurrency arbitrage system

ChatGPT

In an **automated cryptocurrency arbitrage system**, a good example of a **bounded context** could be the **"Trade Execution"** context. This context is responsible for managing the actual buying and selling of cryptocurrencies on different exchanges to take advantage of price differences.

### Example: **Trade Execution Bounded Context**

#### Inside this bounded context:
- **Terms have specific meanings**:
  - *"Order Execution"* means placing a buy or sell order on an exchange.
  - *"Trade Settlement"* means confirming that the order was successfully completed.
  - *"Transaction Fees"* are the costs incurred for each trade, and their calculation is specific to this context.

- **Core functionality**:
  - Identifying which orders need to be placed (buying on one exchange and selling on another).
  - Calculating the fees for executing trades on different exchanges.
  - Executing trades on multiple exchanges, ensuring that the orders are placed correctly and tracked.
  - Monitoring the completion of these trades to confirm profits or losses.

#### Why it's a bounded context:
The *Trade Execution* context focuses exclusively on handling trade operations and has its own set of rules and meanings. The term *"order"* here refers specifically to cryptocurrency trades (buying and selling), and this meaning might differ in other parts of the system, like in an *"Account Management"* context, where *"order"* could mean a user's fund withdrawal request.

### Separation from other contexts:
1. **Account Management Context**: Manages user accounts, balances, and funds across exchanges. It doesn't need to know the details of how trades are executed—just that they affect user balances.
2. **Price Monitoring Context**: Monitors price differences between exchanges and triggers arbitrage opportunities. It doesn’t handle the execution of trades—it just provides data to the Trade Execution context.

By keeping **Trade Execution** as its own bounded context, we make sure that all the complexity related to managing trade orders, fees, and settlements is contained. This makes the system easier to maintain and extend 【8†source】【9†source】【10†source】.

User

How do two bordering bounded contexts fit together?

ChatGPT

Two bordering **bounded contexts** fit together by interacting in a controlled, well-defined manner, but they remain independent in terms of their internal logic and language. The way they communicate is usually through **events** or **commands**, ensuring that each context maintains its own clear responsibilities without tightly coupling their implementations.

### Key ways two bounded contexts fit together:

1. **Context Mapping**:
   This process defines how two bounded contexts relate to each other. There are different types of relationships, but some common ones include:
   - **Customer-Supplier**: One context (supplier) provides data or functionality that another (customer) consumes. For example, a *Price Monitoring* context could supply real-time cryptocurrency prices to a *Trade Execution* context.
   - **Conformist**: One context (customer) fully adapts to the model of another (supplier). The consumer context doesn’t influence the supplier, meaning it has to accept whatever data and format the supplier provides.

2. **Shared Concepts**:
   While the two contexts are independent, they might need to agree on some shared concepts or data. For instance, both a *User Account* context and a *Trade Execution* context need to understand what an “account balance” means, but they each handle it differently. The *Trade Execution* context uses the balance for trade settlements, while the *User Account* context manages deposits and withdrawals.

3. **Communication through Messages**:
   - **Events**: When something significant happens in one context, it can raise an event to inform another context. For example, the *Trade Execution* context might raise an event like `TradeCompleted`, which the *Account Management* context listens to so it can update the user’s balance.
   - **Commands**: One context can send a command to another to trigger some action. For instance, the *Price Monitoring* context might send a `PlaceTradeOrder` command to the *Trade Execution* context when a price difference is detected.

### Example: **Trade Execution** and **Account Management** Contexts in a Cryptocurrency Arbitrage System

#### Trade Execution Context:
   - Handles placing and settling trades across different exchanges.
   - Knows about current trades, fees, and the mechanics of executing buy/sell orders.
   - After a successful trade, it raises an event like `TradeSettled` that informs other contexts.

#### Account Management Context:
   - Manages users’ funds and balances across different exchanges.
   - Updates balances when notified about trade outcomes from the *Trade Execution* context.
   - Listens for the `TradeSettled` event and updates the corresponding account balance based on the trade's outcome.

#### Interaction:
   - When a trade is completed in the *Trade Execution* context, it raises a **`TradeSettled` event**.
   - The *Account Management* context listens for this event and responds by updating the user’s balance accordingly.

Through **event-based communication**, the two contexts work together without needing to know the internal workings of each other. This keeps the system modular, allowing changes in one context without breaking the other 【8†source】【10†source】【9†source】.

### Analogy:
Think of two neighboring countries. Each has its own laws (bounded contexts with independent rules and languages), but they trade goods (information) across the border. They agree on what kinds of goods can be traded (shared concepts) and have specific border control processes (events/commands) to handle the exchange, ensuring both countries remain sovereign while still benefiting from interaction.

User

How many events correspond to one bounded context?

ChatGPT

The number of events in a **bounded context** can vary depending on the complexity and the responsibilities of that context. There isn’t a fixed number of events per bounded context—each context might have just a few or many events, depending on the specific domain logic it handles. What matters most is that the events in a bounded context reflect the significant changes or occurrences within that context's boundaries.

### Factors affecting the number of events:
1. **The Complexity of the Context**:
   - A simple bounded context might have just a few events because it handles a small, focused part of the system. For example, a **Currency Conversion** context in an arbitrage system might only have events like `ConversionRateUpdated` or `ConversionCompleted`.
   - A more complex context, like **Trade Execution**, could have multiple events representing different stages and outcomes of trade processing (e.g., `BuyOrderPlaced`, `SellOrderPlaced`, `TradeSettled`, `TradeFailed`).

2. **Granularity of Domain Events**:
   - How granular or detailed the events are also influences the number. Some bounded contexts might break down processes into more detailed steps, leading to more events. For example, in a **Trade Execution** context, you might have:
     - `TradeInitiated` (when a trade starts),
     - `OrderPlaced` (when an order is sent to the exchange),
     - `OrderExecuted` (when the exchange confirms the order),
     - `TradeSettled` (when the entire trade is fully completed).

3. **Workflow and Business Logic**:
   - In many cases, events correspond to significant steps in a workflow. If a context handles a multi-step process, each important step might trigger an event. For instance, a **Risk Management** context might have events like `RiskCheckPassed` or `RiskCheckFailed`.

4. **Event Storming and Exploration**:
   - During **event storming**, events are discovered based on business processes. The number of events can evolve as more domain knowledge is gathered and as different paths (including failure paths) are explored 【9†source】【10†source】.

### Example:
In an **Account Management** bounded context of an arbitrage system, the events might look like this:
1. `DepositCompleted`: Fired when a user's deposit is successfully credited.
2. `WithdrawalCompleted`: Fired when a withdrawal request is completed.
3. `BalanceUpdated`: Fired after every successful trade or fund movement.
4. `WithdrawalFailed`: Triggered if a withdrawal request cannot be processed due to insufficient funds.

So, this bounded context has four domain events, which capture the main activities related to account balance and funds management.

### Conclusion:
The number of events in a bounded context will depend on how many key business occurrences need to be tracked. Each bounded context will have its own unique set of events that reflect what is happening within its scope, and this could be just a few or many, depending on the complexity of the domain logic it handles 【9†source】【10†source】.

User

In the context of functional programming, do events correspond to functions?

ChatGPT

In functional programming, **events** don't directly correspond to **functions**, but they are closely related in how they are handled. Here's how we can think of the relationship between events and functions:

### Events in Functional Programming:
- An **event** in Domain-Driven Design (DDD) represents something significant that has already happened in the system, like `TradeCompleted` or `PriceUpdated`.
- Events are **data**—they carry information about an occurrence in the domain. For example, `TradeCompleted` might include details like the trade ID, timestamp, and amount traded.

### Functions in Functional Programming:
- A **function** is a core concept in functional programming. It takes inputs and produces outputs, without side effects (pure functions) or with controlled side effects.
- In an event-driven architecture using functional programming, **functions react to events**. That is, when an event occurs, a function processes it and may trigger other actions (like generating a new event or updating some state).

### Relationship Between Events and Functions:
1. **Events are inputs to functions**:
   In functional programming, events are often treated as **input data** to functions. The function processes the event and produces a new result. For example, a function could take a `TradeCompleted` event as input, and the function might update account balances or generate a new event like `AccountBalanceUpdated`.

```fsharp
   let processTrade (event: TradeCompleted) =
       let newBalance = updateBalance event.amount
       AccountBalanceUpdated newBalance
   ```

2. **Functions produce events**:
   Functions in functional programming can **emit new events** as output. After processing some domain logic, the function can create a new event that represents the outcome of that process. For example, after a successful trade execution, a function might produce a `TradeSettled` event.

   ```fsharp
   let settleTrade (trade: ExecutedTrade) =
       if trade.success then TradeSettled(trade.id, trade.amount)
       else TradeFailed(trade.id)
   ```

3. **Pure functions and side effects**:
   Events often signal changes in the system that can lead to side effects, like updating a database or sending a message to another service. In functional programming, we strive to **isolate side effects**. The **pure function** itself will process the event and return data (like a new event), but any side effect (e.g., saving the event to a database) is handled elsewhere in the system, often in a layer dedicated to handling side effects 【11†source】.

4. **Event-driven workflows as pipelines**:
   In functional programming, we can model an **event-driven workflow** as a sequence of functions. Each function reacts to an event and may trigger another event, creating a pipeline of event processing. For example, in a cryptocurrency arbitrage system:
   - A `PriceDifferenceDetected` event triggers a function to execute a trade.
   - A `TradeExecuted` event then triggers a function to update the user’s account balance.

   ```fsharp
   let priceDifferenceDetected priceEvent =
       executeTrade priceEvent
       |> TradeExecuted

   let tradeExecuted tradeEvent =
       updateBalance tradeEvent
       |> AccountBalanceUpdated
   ```

### Conclusion:
In functional programming, **events are not functions**, but they are handled **by functions**. Functions react to events (as inputs) and can produce new events (as outputs). This fits well with the principles of immutability and functional purity, as events are immutable pieces of data that flow through the system, and functions process them in a predictable way. Functions and events work together to implement **event-driven, functional workflows**.

User

Provide F# example of PriceDifferenceDetected in an automated cryptocurrency arbitrage system.

ChatGPT

Here’s an example in F# of how you might model the **`PriceDifferenceDetected`** event in an automated cryptocurrency arbitrage system. This function detects a price difference between exchanges, triggers a trade, and potentially raises a new event like `TradeInitiated`.

### Example Overview
- The system monitors prices from multiple exchanges.
- When a significant price difference is detected (e.g., the price of Bitcoin on Exchange A is lower than on Exchange B), a function processes the event (`PriceDifferenceDetected`).
- Based on the price difference, the system initiates a buy on the cheaper exchange and a sell on the more expensive exchange.
- The function returns a new event, like `TradeInitiated`.

### F# Code Example

```fsharp
// Define the PriceDifferenceDetected event type
type PriceDifferenceDetected = {
    ExchangeA: string
    ExchangeB: string
    PriceA: decimal
    PriceB: decimal
    Timestamp: System.DateTime
}

// Define the TradeInitiated event type
type TradeInitiated = {
    BuyExchange: string
    SellExchange: string
    Amount: decimal
    BuyPrice: decimal
    SellPrice: decimal
    Timestamp: System.DateTime
}

// Function to detect a price difference and decide to initiate a trade
let handlePriceDifference (priceDifference: PriceDifferenceDetected) =
    // Threshold for significant price difference (e.g., 1% profit margin)
    let threshold = 0.01M

    // Calculate percentage difference
    let percentageDifference = (priceDifference.PriceB - priceDifference.PriceA) / priceDifference.PriceA

    // Check if the price difference is significant
    if percentageDifference > threshold then
        // Create a new TradeInitiated event
        let tradeInitiated = {
            BuyExchange = priceDifference.ExchangeA
            SellExchange = priceDifference.ExchangeB
            Amount = 1.0M // example amount of cryptocurrency to trade
            BuyPrice = priceDifference.PriceA
            SellPrice = priceDifference.PriceB
            Timestamp = System.DateTime.UtcNow
        }

        // Return the new event (TradeInitiated)
        Some(tradeInitiated)
    else
        // No significant price difference, no trade initiated
        None

// Example usage
let priceDifferenceEvent = {
    ExchangeA = "ExchangeA"
    ExchangeB = "ExchangeB"
    PriceA = 50000.00M
    PriceB = 50500.00M
    Timestamp = System.DateTime.UtcNow
}

match handlePriceDifference priceDifferenceEvent with
| Some(trade) -> printfn "Trade Initiated: Buy on %s, Sell on %s"
trade.BuyExchange trade.SellExchange
| None -> printfn "No trade initiated, price difference too small"
```

### Breakdown of the Example:
1. **Event Definition**:
   - `PriceDifferenceDetected` holds the information about prices on two exchanges at a given point in time.
   - `TradeInitiated` is the event triggered when the price difference meets a threshold for profitable arbitrage.

2. **The Function**:
   - `handlePriceDifference` processes the `PriceDifferenceDetected` event.
   - It calculates whether the price difference is large enough (above the threshold) to justify executing a trade.
   - If the price difference is significant, it creates a `TradeInitiated` event. Otherwise, no trade is initiated.

3. **Running the Example**:
   - We simulate a price difference between two exchanges and pass it to the function.
   - The function checks the price difference and prints a message indicating whether a trade was initiated or not.

### Event Flow:
- **Input Event**: `PriceDifferenceDetected` is the event that triggers the process.
- **Processing**: The function checks if the price difference is greater than a threshold.
- **Output Event**: If the conditions are met, it triggers the `TradeInitiated` event.

This example models a simple arbitrage decision-making process using immutable data and functions, aligning with functional programming principles.

User

In the context of designing an event driven pipeline in F#, will events tend to be record types?

ChatGPT

Yes, in F# and functional programming in general, **events** in an event-driven pipeline are typically modeled as **record types**. This is because record types offer a clean, immutable, and structured way to represent data, which aligns well with functional programming principles.

### Why Use Record Types for Events?

1. **Immutability**:
   - Record types in F# are immutable by default, meaning once an event is created, its data cannot be modified. This is critical in event-driven systems, where events represent something that has already happened in the past and should not change.
   - Immutable data structures ensure that the system behaves predictably, especially when events are passed between different parts of the pipeline.

2. **Simple, Structured Data**:
   - Events are essentially just data describing what happened in the system. Record types allow you to define this data in a simple, structured way, with clear fields that describe the event's properties.
   - For example, an event like `PriceDifferenceDetected` could have fields for the exchanges involved, the prices, and the time the event was detected.

3. **Type Safety**:
   - Using record types for events ensures type safety. The compiler can help catch mistakes, like trying to access a field that doesn’t exist or passing incorrect data to a function handling the event.
   - F#'s type inference also makes handling records straightforward and concise.

4. **Pattern Matching**:
   - F#'s pattern matching works very naturally with record types. You can easily destructure events to access their fields and take different actions based on the event type.
   - This is particularly useful in event-driven pipelines where you need to handle different kinds of events.

### Example of an Event as a Record Type

Here’s an example where events in an automated cryptocurrency arbitrage system are represented as record types:

```fsharp
// Defining events as record types
type PriceDifferenceDetected = {
    ExchangeA: string
    ExchangeB: string
    PriceA: decimal
    PriceB: decimal
    Timestamp: System.DateTime
}

type TradeExecuted = {
```

~~~~
    BuyExchange: string
    SellExchange: string
    Amount: decimal
    BuyPrice: decimal
    SellPrice: decimal
    Timestamp: System.DateTime
}

// Handling events in a pipeline
let handlePriceDifference (event: PriceDifferenceDetected) =
    let priceDifference = event.PriceB - event.PriceA
    if priceDifference > 100.00M then
        // Simulate a trade execution
        let tradeExecuted = {
            BuyExchange = event.ExchangeA
            SellExchange = event.ExchangeB
            Amount = 1.0M
            BuyPrice = event.PriceA
            SellPrice = event.PriceB
            Timestamp = System.DateTime.UtcNow
        }
        Some(tradeExecuted)  // Return a new event
    else
        None  // No trade, so no new event is produced

// Example usage
let priceDifferenceEvent = {
    ExchangeA = "ExchangeA"
    ExchangeB = "ExchangeB"
    PriceA = 50000.00M
    PriceB = 50200.00M
    Timestamp = System.DateTime.UtcNow
}

match handlePriceDifference priceDifferenceEvent with
| Some(trade) -> printfn "Trade Executed: Buy on %s, Sell on %s"
trade.BuyExchange trade.SellExchange
| None -> printfn "No trade executed, price difference too small"
```
~~~~

### Characteristics of Record Types as Events:
- **Clear structure**: Fields like `ExchangeA`, `PriceA`, and `Timestamp` clearly define what data is associated with the event.
- **Immutable**: Once created, the event instance can't be modified, ensuring that historical data about the event stays consistent.
- **Type-safe**: F# ensures that the event's data structure is respected throughout the system.

### Conclusion:
In F#, **record types** are an ideal choice for modeling events in an event-driven pipeline because they provide immutability, clarity, and type safety, making them perfect for representing the key occurrences in the system. This approach fits naturally with the functional programming paradigm, where pure data structures and immutability are central principles.

User

"Generate statistics and analysis based on our exchange. Please provide the statistics in table format:

Table 1
Column 1: Number of additional questions I asked, excluding this prompt asking for analysis and the conversation starters: "Can you walk me through the basic financial terms (names) used in cryptocurrency arbitrage? What is the relationship between them? E.g. how is arbitrage opportunity related to an order and a trade and a transaction?" and "can you walk me through typical systems and actors that are part of cryptocurrency arbitrage and in particular, an automated trading system? Which are not relevant in the context of the attached project specification?".
Column 2: The number of questions asked per prompt - make sure not to account for other types of text (e.g. conversation elements like thank you, exclamations etc.). Whenever multiple questions were included in the same prompt, account for them separately in the table.

Table 2: An analysis of our conversation
Column 1: prompt
Column 2: question I asked
Column 3: classification of the nature of the questions asked: about getting a baseline understanding provided in your answers to the prompt/about exploring the knowledge presented by you/other?
Column 4: classification of the complexity and difficulty of questions asked: basic, intermediate, and advanced levels.
Column 5: gauge sentiment and feedback regarding LLM interactions by classifying it into occurrences of positive, neutral, and negative sentiments in feedback. Also detect and specific feedback themes (e.g., helpfulness, clarity)."

ChatGPT

### Table 1: Number of Additional Questions and Count per Prompt

Total Number of Questions	Number of Questions in the Prompt
1	1
2	1
3	1
4	1
5	1
6	1
7	1
8	1

### Table 2: Analysis of the Conversation

Prompt	Question	Nature of the Question	Complexity	Sentiment
Can you give me some examples of how to name events in an automated cryptocurrency arbitrage system?	Can you give me some examples of how to name events in an automated cryptocurrency arbitrage system?	Exploring the knowledge	Intermediate	Neutral

I am a software engineering student, please explain to me in simple words what is a bounded context.	Please explain to me in simple words what is a bounded context?	Getting a baseline understanding	Basic	Neutral
Can you give an example of a bounded context in an automated cryptocurrency arbitrage system?	Can you give an example of a bounded context in an automated cryptocurrency arbitrage system?	Exploring the knowledge	Intermediate	Neutral
How do two bordering bounded contexts fit together?	How do two bordering bounded contexts fit together?	Exploring the knowledge	Advanced	Neutral
How many events correspond to one bounded context?	How many events correspond to one bounded context?	Exploring the knowledge	Advanced	Neutral
In the context of functional programming, do events correspond to functions?	Do events correspond to functions?	Exploring the knowledge	Advanced	Neutral
Provide F# example of PriceDifferenceDetected in an automated cryptocurrency arbitrage system.	Provide F# example of PriceDifferenceDetected in an automated cryptocurrency arbitrage system.	Exploring the knowledge	Intermediate	Neutral
In the context of designing an event driven pipeline in F#, will events tend to be record types?	Will events tend to be record types?	Exploring the knowledge	Advanced	Neutral

### Summary of Findings:
- Most of the questions are classified as **"exploring the knowledge"**, showing a focus on expanding and applying the concepts provided.
- There is a balance of questions classified under **intermediate** and **advanced** complexity, with some basic questions as well.
- Sentiment throughout the conversation has been **neutral**, with no direct expressions of positive or negative feedback.

User

generate a csv file of the above two tables

ChatGPT

I have generated the CSV file containing the two tables. You can download it from the link below:

[Download conversation_analysis.csv](sandbox:/mnt/data/conversation_analysis.csv)

Crypto Arbitrage Terms Explained

User

Can you walk me through the basic financial terms (names) used in cryptocurrency arbitrage? What is the relationship between them? E.g. how is arbitrage opportunity related to an order and a trade and a transaction?

Tool

All the files uploaded by the user have been fully loaded. Searching won't provide additional information.

ChatGPT

In cryptocurrency arbitrage, various financial terms relate to the steps of identifying and executing an arbitrage opportunity. Here are the key terms and their relationships:

1. **Arbitrage Opportunity**: This refers to the situation where the same cryptocurrency is priced differently across different exchanges. The arbitrage opportunity arises when there is a price difference, allowing a trader to buy at a lower price on one exchange and sell at a higher price on another to profit from the price disparity.

2. **Order**: To take advantage of an arbitrage opportunity, you need to place orders. An order is a request made on a cryptocurrency exchange to buy or sell a specific asset. Orders come in various types, like limit orders (you specify the price) or market orders (executed at the current market price).

3. **Trade**: When your order is matched with an opposite order (i.e., someone wants to sell while you want to buy), a trade is executed. The trade is the actual exchange of cryptocurrency between parties. This step turns the arbitrage opportunity into reality by locking in the price on both exchanges.

4. **Transaction**: Once a trade is completed, it results in a blockchain transaction. A transaction is the actual transfer of cryptocurrency from one wallet to another. This may include transferring assets between exchanges as part of arbitrage execution. Blockchain transactions incur fees and can affect the profitability of arbitrage.

### Relationship Between These Terms:

- **Arbitrage Opportunity** is the starting point. You find a price difference across exchanges.
- You place **Orders** on both exchanges: one to buy and one to sell.
- When the orders are matched, they result in a **Trade**, which finalizes the buying and selling at the prices you specified.
- The **Transaction** on the blockchain follows, ensuring the transfer of cryptocurrency from your account to another as part of fulfilling the arbitrage.

In this flow, orders and trades are closely tied to market activity on the exchanges, while transactions represent the actual transfer of assets between parties. Arbitrage involves careful timing, as the price difference (opportunity) can disappear if the trades or transactions take too long.

User

Can you walk me through typical systems that are part of cryptocurrency arbitrage and in particular, an automated trading system? Which are not relevant in the context of the attached project specification?

ChatGPT

A typical cryptocurrency arbitrage system consists of several key components, especially when automation is involved. Below, I'll walk you through the main systems involved and highlight which might not be relevant based on the principles from the attached project specifications on Domain-Driven Design (DDD)

```
and functional programming.

### Typical Systems in an Automated Arbitrage Trading System

1. **Market Data Aggregator**:
   - **Function**: Continuously collects price data from multiple cryptocurrency
exchanges. This is critical to identify arbitrage opportunities by comparing
prices in real time.
   - **Relevance**: Essential in arbitrage since opportunities are short-lived
and depend on up-to-date price feeds.

2. **Opportunity Detection Module**:
   - **Function**: Analyzes the incoming price data, identifies price
discrepancies, and flags potential arbitrage opportunities. This module includes
algorithms that detect differences between buying and selling prices across
exchanges.
   - **Relevance**: Central to the entire system. It's where the decision-making
begins.

3. **Order Management System (OMS)**:
   - **Function**: Manages the placement of orders on different exchanges based
on the identified opportunities. Orders can be market, limit, or conditional
orders.
   - **Relevance**: Crucial, as the system needs to execute buy/sell orders
quickly. In functional architecture, this process can be modeled as a workflow
where orders trigger domain events 【9†source】.

4. **Risk Management and Validation**:
   - **Function**: Ensures that any trade follows predefined risk management
rules, such as ensuring sufficient liquidity or verifying the exchange's
transaction fees.
   - **Relevance**: This layer is important but may be minimized in the context
of a DDD-based design if the focus is on the core functional pipeline. Depending
on the specification, some parts of risk management might be treated as external
services or infrastructure elements 【8†source】 【11†source】.

5. **Trade Execution Engine**:
   - **Function**: Executes the trades across different exchanges. This engine
handles the process of submitting trades and ensures they are carried out within
the required time frame.
   - **Relevance**: Very relevant since trade execution needs to happen with
minimal delay. Execution can be represented as a core domain event that occurs
once conditions (price discrepancies) are met 【8†source】.

6. **Transaction Settlement**:
   - **Function**: Ensures that after a trade is executed, the cryptocurrency or
fiat is transferred between the involved wallets or accounts. This also includes
any reconciliation of balances.
   - **Relevance**: Relevant but could be treated as an external system or
infrastructure layer, in line with functional architecture principles 【9†source】.

7. **Monitoring and Alerts**:
   - **Function**: Provides real-time feedback and alerts to users or system
administrators about trade execution status, potential system issues, or missed
opportunities.
   - **Relevance**: Not a core domain function, might be treated as part of the
```

infrastructure and external logging or alerting services 【11†source】.

8. **Exchange API Connectors**:
   - **Function**: Connects to cryptocurrency exchanges' APIs to retrieve price data and execute orders.
   - **Relevance**: Necessary for external communication but can be isolated as an infrastructure service rather than core business logic 【9†source】 【11†source】.

### Systems That May Not Be Relevant in the Context of This Project:

1. **High-Frequency Trading (HFT) Algorithms**:
   - In some arbitrage systems, high-frequency trading is involved. However, given the focus on DDD and functional workflows, which emphasize predictability and clarity, HFT, which requires low-level hardware optimizations and non-functional techniques, may be irrelevant.

2. **Complex Financial Derivatives**:
   - If the project is focused on spot arbitrage (buying and selling the actual cryptocurrency), then advanced financial products like futures, options, or margin trading might not be relevant.

3. **Advanced Machine Learning Modules**:
   - While machine learning can enhance arbitrage by predicting price movements, it introduces side effects and non-determinism, which functional programming and DDD aim to minimize. Machine learning models are often seen as impure components and might be relegated to non-core modules.

4. **Portfolio Management Systems**:
   - Systems that track long-term holdings or manage diversified portfolios are not directly relevant to pure arbitrage, which is typically short-term and focuses on exploiting price discrepancies.

By separating concerns between core business logic (like trade execution) and external dependencies (like exchange connectors), the project can maintain a clean separation of responsibilities, following the **Onion Architecture** 【11†source】. This architecture suggests that core logic should be purely functional, while side effects like API communication should be isolated to the infrastructure layer.

User

Generate statistics and analysis based on our exchange. Please provide the statistics in table format:

Table 1
Column 1: Number of additional questions I asked, excluding this prompt asking for analysis and the conversation starters: "Can you walk me through the basic financial terms (names) used in cryptocurrency arbitrage? What is the relationship between them? E.g. how is arbitrage opportunity related to an order and a trade and a transaction?" and "can you walk me through typical systems and actors that are part of cryptocurrency arbitrage and in particular, an automated trading system? Which are not relevant in the context of the attached project specification?".
Column 2: The number of questions asked per prompt - make sure not to account for other types of text (e.g. conversation elements like thank you, exclamations etc.). Whenever multiple questions were included in the same prompt, account for

them separately in the table.

Table 2: An analysis of our conversation
Column 1: prompt
Column 2: question I asked
Column 3: classification of the nature of the questions asked: about getting a baseline understanding provided in your answers to the prompt/about exploring the knowledge presented by you/other?
Column 4: classification of the complexity and difficulty of questions asked: basic, intermediate, and advanced levels.
Column 5: gauge sentiment and feedback regarding LLM interactions by classifying it into occurrences of positive, neutral, and negative sentiments in feedback. Also detect and specific feedback themes (e.g., helpfulness, clarity).

ChatGPT

### Table 1: Number of Additional Questions and the Number of Questions per Prompt

Number of Additional Questions	Number of Questions per Prompt
1	3
2	3

### Table 2: An Analysis of Our Conversation

Prompt	Question(s) Asked	Nature of Questions	Complexity Level	Sentiment
Can you walk me through the basic financial terms (names) used in cryptocurrency arbitrage? What is the relationship between them? E.g. how is arbitrage opportunity related to an order and a trade and a transaction?	Can you walk me through the basic financial terms (names) used in cryptocurrency arbitrage? What is the relationship between them? E.g. how is arbitrage opportunity related to an order and a trade and a transaction?	baseline understanding	intermediate	neutral
Can you walk me through typical systems and actors that are part of cryptocurrency arbitrage and in particular, an automated trading system? Which are not relevant in the context of the attached project specification?	Can you walk me through typical systems and actors that are part of cryptocurrency arbitrage? In particular, an automated trading system? Which are not relevant in the context of the attached project specification?	baseline understanding	intermediate	neutral
Generate statistics and analysis based on our exchange.	Generate statistics and analysis based on our exchange?	other	intermediate	neutral

The analysis shows a consistent focus on gaining baseline understanding of key concepts, with intermediate-level complexity throughout. Sentiment remained neutral across the conversation, without specific feedback on helpfulness or clarity.