PeeGeeQ Financial Services Event Catalogue

A comprehensive guide for standardised event-driven architecture in financial services using PeeGeeQ's bitemporal event store capabilities.

Table of Contents

Introduction

Purpose

This document proposes to establish a standardised approach to event-driven architecture for financial services organizations using PeeGeeQ's bitemporal event store. It provides:

- **Systematic event naming conventions** that work across all financial domains
- **Comprehensive event dictionary** covering trading, custody, treasury, funds, and securities services
- **Technical implementation patterns** for PeeGeeQ integration
- **Migration strategies** for adopting standardized events

Key Principles

1. **Business-Centric**: Events represent real business activities and outcomes
2. **Cross-Domain Consistency**: Same patterns work across trading, custody, funds, and regulatory domains
3. **Future-Proof**: Naming and structure scales to new business requirements
4. **Audit-Ready**: Complete traceability for regulatory compliance
5. **Developer-Friendly**: Clear, predictable patterns reduce complexity

Financial Services Domains Covered

- **Trading**: Trade capture, confirmation, and lifecycle management
- **Custody**: Settlement instructions, confirmations, and position management
- **Treasury**: Cash movements, liquidity management, and funding
- **Fund Administration**: NAV calculation, subscriptions, redemptions, and transfers
- **Securities Services**: DVP/FOP settlement, securities lending, and safekeeping
- **Regulatory**: Compliance monitoring, reporting, and threshold management

- **Operations**: Exception management, reconciliation, and manual repairs

Event Naming Strategy

The Challenge

Creating event names that are:

1. **Meaningful** - clearly describe what happened
2. **Unique** - no conflicts across domains
3. **Consistent** - follow predictable patterns
4. **Scalable** - work for new domains/processes

Event Naming Pattern: `{entity}.{action}.{state}`

Event names follow a three-part pattern where:

- **Entity**: The business object being acted upon
- **Action**: The business action being performed
- **State**: The resulting state or outcome

Why This Works

1. Meaningful: Each name tells a complete story

- `trade.capture.completed` - A trade was captured and it completed successfully
- `instruction.settlement.matched` - A settlement instruction was matched with counterparty
- `position.reconciliation.failed` - A position reconciliation process failed

2. Unique: Three-part names eliminate conflicts

- `trade.confirmation.received` (Trading domain)
- `nav.validation.received` (Funds domain)
- `report.regulatory.received` (Regulatory domain)

3. Consistent: Same pattern across all domains

- All events follow `{entity}.{action}.{state}` structure
- Predictable naming makes integration easier

4. Scalable: Easy to add new events

- New entities: `collateral` , `proxy.voting` , `corporate.action`
- New actions: `substitution` , `recall` , `escalation`
- New states: `breached` , `acknowledged` , `disputed`

System Context: Why Not Include System Names?

A common question is whether event names should include the source system, like `trading-system.trade.capture.completed` . I recommend **against** this approach for several reasons:

Arguments Against System Prefixes

1. **Business Focus**: Events should represent business facts, **not** technical implementation details
2. **System Independence**: `trade.capture.completed` is a business fact regardless of which system captured it
3. **Coupling**: Adding system names couples the event schema to a technical architecture
4. **Evolution**: Systems get replaced, merged, or split - business events remain constant

CloudEvents Already Handles System Context

The CloudEvents specification provides the `source` field specifically for system identification:

```json
{
  "specversion": "1.0",
  "type": "com.fincorp.trading.equities.capture.TradeCaptured.v1",
  "source": "murex-trading-system",
  "id": "01234567-89ab-cdef-0123-456789abcdef",
  "time": "2024-01-15T10:30:00Z",
  "subject": "trade-12345",
  "data": {
    "eventName": "trade.capture.completed",
    "tradeId": "12345",
    "instrumentId": "AAPL",
    "quantity": 1000
  }
}
```

This approach separates **business semantics** (event name) from **technical context** (source system), providing the best of both worlds: clear business meaning with full system traceability.

Construction Rules

Entities (Business Objects)

Domain	Entities
Core Trading	trade, order, execution, allocation
Settlement	instruction, settlement, confirmation, matching
Positions	position, movement, transfer, safekeeping
Cash	cash, payment, funding, liquidity
Funds	nav, subscription, redemption, transfer, dividend
Securities Services	lending, collateral, recall, dvp, fop
Operations	exception, break, repair, reconciliation
Regulatory	compliance, report, threshold, violation
Corporate Actions	corporate.action, entitlement, election, proxy.voting
Reference Data	counterparty, security, account, rate

Actions (Business Processes)

Category	Actions
Lifecycle	capture, creation, initiation, generation
Processing	processing, calculation, validation, verification
Workflow	confirmation, approval, authorization, assignment
Movement	settlement, transfer, movement, delivery
Monitoring	detection, investigation, monitoring, checking
Resolution	resolution, repair, correction, escalation
Communication	notification, reporting, submission, announcement

States (Outcomes)

Category	States
Initiation	initiated, started, requested, created, generated
In-Progress	processing, pending, validating, investigating
Success	completed, finished, settled, matched, approved, confirmed
Failure	failed, rejected, disputed, unmatched, insufficient
Exceptional	breached, violated, escalated, expired, suspended
Resolution	resolved, corrected, repaired, acknowledged

Benefits

1. **Developer Friendly**: Clear, predictable naming patterns
2. **Business Friendly**: Names describe actual business processes
3. **Future-Proof**: Easy to extend with new entities, actions, and states
4. **Cross-Domain Consistency**: Works consistently across all financial services domains
5. **Audit Ready**: Event names clearly describe what happened in business terms
6. **Technical Integration**: Easy routing (`*.settlement.*`), filtering, and monitoring

Financial Services Event Dictionary

Event Dictionary Structure

Each event in the dictionary follows this standard format:

```
Event Type: com.fincorp.{domain}.{instrument}.{process}.{EventName}.v1
Event Name: {entity}.{action}.{state}
Routing Key: {domain}.{instrument}.{process}.{region}.{priority}
Description: Brief description of what the event represents
Payload: Core data elements
```

```
Triggers: What causes this event to be published
Consumers: Who typically subscribes to this event
```

Trading Domain Events

Trade Capture Events

```
Event Type: com.fincorp.trading.{instrument}.capture.TradeCaptured.v1
Event Name: trade.capture.completed
Routing Key: trading.{instrument}.capture.{region}.{priority}
Description: A new trade has been captured in the trading system
Payload: tradeId, instrumentId, quantity, price, counterpartyId, traderId, tradeDate
Triggers: Trade execution, manual trade entry, trade import
Consumers: Risk systems, settlement systems, regulatory reporting
```

Trade Confirmation Events

```
Event Type: com.fincorp.trading.{instrument}.confirmation.TradeConfirmed.v1
Event Name: trade.confirmation.received
Routing Key: trading.{instrument}.confirmation.{region}.normal
Description: Trade has been confirmed with counterparty
Payload: tradeId, confirmationId, confirmationStatus, confirmationDate
Triggers: Counterparty confirmation received, auto-confirmation timeout
Consumers: Settlement systems, operations teams, client reporting
```

Custody Domain Events

Settlement Instruction Events

```
Event Type: com.fincorp.custody.{instrument}.settlement.instruction.SettlementInstructed.v1
Event Name: instruction.settlement.created
Routing Key: custody.{instrument}.settlement.instruction.{region}.{priority}
Description: Settlement instruction has been created
Payload: instructionId, tradeId, securityId, quantity, settlementDate, counterparty
Triggers: Trade settlement due, manual instruction creation
Consumers: Custodians, settlement systems, exception management
```

Settlement Confirmation Events

```
Event Type: com.fincorp.custody.{instrument}.settlement.confirmation.SettlementConfirmed.v1
Event Name: instruction.settlement.completed
Routing Key: custody.{instrument}.settlement.confirmation.{region}.normal
Description: Settlement has been confirmed as completed
Payload: instructionId, settlementId, actualSettlementDate, settledQuantity
Triggers: Custodian confirmation, settlement system update
Consumers: Position systems, cash management, client reporting
```

Treasury Domain Events

Cash Movement Events

```
Event Type: com.fincorp.treasury.cash.movement.CashMoved.v1
Event Name: cash.movement.completed
Routing Key: treasury.cash.movement.{region}.{priority}
Description: Cash has moved between accounts
Payload: movementId, fromAccount, toAccount, amount, currency, movementType
Triggers: Settlement, fee payment, dividend payment, manual transfer
Consumers: Cash management, accounting, liquidity management
```

Liquidity Check Events

```
Event Type: com.fincorp.treasury.cash.liquidity.check.LiquidityChecked.v1
Event Name: cash.sufficiency.checked
Routing Key: treasury.cash.liquidity.check.{region}.high
Description: Liquidity sufficiency has been checked
Payload: checkId, accountId, requiredAmount, availableAmount, checkResult
Triggers: Pre-settlement check, large transaction validation
Consumers: Settlement systems, risk management, treasury operations
```

Fund Administration Domain Events

NAV Calculation Events

```
Event Type: com.fincorp.funds.{fund-type}.nav.calculation.NavCalculated.v1
Event Name: nav.calculation.completed
Routing Key: funds.{fund-type}.nav.calculation.{region}.high
Description: Net Asset Value has been calculated for a fund
Payload: fundId, shareClassId, navPerShare, valuationDate, totalNetAssets
Triggers: Daily NAV calculation, month-end valuation, ad-hoc calculation
Consumers: Transfer agent, pricing systems, client reporting, regulatory reporting
```

Subscription Processing Events

```
Event Type: com.fincorp.funds.{fund-type}.subscription.processing.SubscriptionProcessed.v1
Event Name: subscription.processing.completed
Routing Key: funds.{fund-type}.subscription.processing.{region}.normal
Description: Fund subscription has been processed
Payload: subscriptionId, fundId, investorId, subscriptionAmount, sharesAllocated
Triggers: Subscription order received, cash received, NAV available
Consumers: Transfer agent, custody systems, client reporting
```

Securities Services Domain Events

DVP Settlement Events

```
Event Type: com.fincorp.securities.{instrument}.dvp.settlement.DvpSettled.v1
Event Name: dvp.settlement.completed
Routing Key: securities.{instrument}.dvp.settlement.{region}.{priority}
Description: Delivery vs Payment settlement has been completed
Payload: settlementId, securityId, quantity, settlementAmount, deliveryAccount, paymentAccount
Triggers: Settlement instruction matching, clearing system confirmation
Consumers: Position systems, cash management, client reporting
```

Securities Lending Events

```
Event Type: com.fincorp.securities.{instrument}.securities.lending.SecuritiesLent.v1
Event Name: lending.agreement.executed
Routing Key: securities.{instrument}.securities.lending.{region}.normal
Description: Securities have been lent to a borrower
Payload: loanId, securityId, quantity, borrowerId, lendingRate, collateralValue
Triggers: Lending agreement execution, collateral posting
Consumers: Risk management, income tracking, regulatory reporting
```

Operational Events (Cross-Domain)

Exception Management Events

```
Event Type: com.fincorp.{domain}.{instrument}.exception.management.ExceptionManaged.v1
Event Name: exception.detection.automated
Routing Key: {domain}.{instrument}.exception.management.{region}.critical
Description: Operational exception has been identified and managed
Payload: exceptionId, sourceTransactionId, exceptionType, severity, assignedTo
Triggers: System error, validation failure, manual identification
Consumers: Operations teams, management dashboards, audit systems
```

Manual Repair Events

```
Event Type: com.fincorp.{domain}.{instrument}.manual.repair.ManualRepairExecuted.v1
Event Name: repair.manual.executed
Routing Key: {domain}.{instrument}.manual.repair.{region}.high
Description: Manual repair has been executed to fix an issue
Payload: repairId, originalTransactionId, repairType, executedBy, approvedBy
Triggers: Exception resolution, data correction, process override
Consumers: Audit systems, compliance teams, risk management
```

Event Structure & Standards

CloudEvents Envelope

All events use CloudEvents specification as the standard envelope:

```
{
  "specversion": "1.0",
  "type": "com.fincorp.trading.equities.capture.TradeCaptured.v1",
  "source": "trading-system",
  "id": "01234567-89ab-cdef-0123-456789abcdef",
  "time": "2024-01-15T10:30:00Z",
  "datacontenttype": "application/avro",
  "dataschema": "https://schemas.fincorp.com/trading/TradeCaptured/v1",
  "subject": "trade-12345",
  "traceparent": "00-4bf92f3577b34da6a3ce929d0e0e4736-00f067aa0ba902b7-01",
  "correlationid": "correlation-12345",
  "causationid": "causation-67890",
  "data": {
    // Avro-serialized payload
  }
}
```

Required Headers

- **traceparent**: W3C Trace Context for end-to-end tracing
- **correlationid**: Links related events in a business process
- **causationid**: Identifies the event that caused this event
- **validtime**: Business effective time (bi-temporal)
- **partitionkey**: Explicit partitioning for ordering

Event Type Naming Convention

```
com.{organization}.{domain}.{instrument-category}.{process}.{EventName}.v{version}
```

Examples:

- `com.fincorp.trading.equities.capture.TradeCaptured.v1`
- `com.fincorp.custody.bonds.settlement.instruction.SettlementInstructed.v1`
- `com.fincorp.funds.equity.funds.nav.calculation.NavCalculated.v1`

Payload Schema Standards

Avro Schema Structure

```
{
  "type": "record",
  "name": "TradeCapturedPayload",
  "namespace": "com.fincorp.trading.events",
  "fields": [
    {"name": "tradeId", "type": "string"},
    {"name": "instrumentId", "type": "string"},
    {"name": "quantity", "type": {"type": "bytes", "logicalType": "decimal", "precision": 18, "scale": 8}},
    {"name": "price", "type": {"type": "bytes", "logicalType": "decimal", "precision": 18, "scale": 8}},
    {"name": "counterpartyId", "type": "string"},
    {"name": "traderId", "type": "string"},
    {"name": "tradeDate", "type": {"type": "long", "logicalType": "timestamp-millis"}},
    {"name": "validTime", "type": {"type": "long", "logicalType": "timestamp-millis"}},
    {"name": "jurisdiction", "type": "string"},
    {"name": "notionalAmount", "type": {"type": "bytes", "logicalType": "decimal", "precision": 18, "scale": 8}}
  ]
}
```

Routing Key Strategy

Format: `{domain}.{instrument}.{process}.{region}.{priority}`

Examples:

- `trading.equities.capture.us.high`
- `custody.bonds.settlement.eu.normal`
- `funds.equity.funds.nav.calculation.global.critical`

Priority Levels:

- **critical**: Regulatory deadlines, system failures
- **high**: Large transactions, time-sensitive operations
- **normal**: Standard business operations
- **low**: Reporting, analytics, non-urgent processes

PeeGeeQ Integration Patterns

Bi-temporal Event Store Configuration

```
@Configuration
public class FinancialEventStoreConfig {

    @Bean
    public EventStore<TradeCapturedPayload> tradingEventStore() {
        return PgBiTemporalEventStore.<TradeCapturedPayload>builder()
            .withDataSource(tradingDataSource)
            .withTableName("trading_events")
            .withRoutingKeyExtractor(event ->
                generateTradingRoutingKey(event.getPayload()))
            .withSubscriptionFilters(Map.of(
                "high-value-trades", "payload.notionalAmount > 1000000",
                "failed-trades", "payload.status = 'FAILED'",
                "regulatory-reportable", "payload.notionalAmount > 500000"
            ))
            .build();
    }

    @Bean
    public EventStore<SettlementInstructedPayload> custodyEventStore() {
        return PgBiTemporalEventStore.<SettlementInstructedPayload>builder()
            .withDataSource(custodyDataSource)
            .withTableName("custody_events")
            .withRoutingKeyExtractor(event ->
                generateCustodyRoutingKey(event.getPayload()))
            .withSubscriptionFilters(Map.of(
                "failed-settlements", "payload.status = 'FAILED'",
                "high-value-settlements", "payload.notionalAmount > 5000000",
                "cross-border", "payload.jurisdiction != 'domestic'"
            ))
            .build();
    }
}
```

Event Publishing Patterns

```
@Service
public class TradingEventPublisher {

    private final EventStore<TradeCapturedPayload> eventStore;

    public void publishTradeCapture(Trade trade) {
        TradeCapturedPayload payload = TradeCapturedPayload.builder()
            .tradeId(trade.getId())
            .instrumentId(trade.getInstrumentId())
            .quantity(trade.getQuantity())
            .price(trade.getPrice())
            .counterpartyId(trade.getCounterpartyId())
            .traderId(trade.getTraderId())
            .tradeDate(trade.getTradeDate())
            .validTime(trade.getTradeDate()) // Business time
            .jurisdiction(trade.getJurisdiction())
            .notionalAmount(trade.getNotionalAmount())
            .build();

        BiTemporalEvent<TradeCapturedPayload> event = BiTemporalEvent.<TradeCapturedPayload>builder()
            .eventId(UuidV7.generate()) // Time-ordered UUID
            .eventType("com.fincorp.trading.equities.capture.TradeCaptured.v1")
            .source("trading-system")
            .subject("trade-" + trade.getId())
            .correlationId(trade.getCorrelationId())
            .causationId(trade.getCausationId())
            .validTime(trade.getTradeDate())
            .payload(payload)
            .build();

        eventStore.append(event);
    }
}
```

Event Subscription Patterns

```
@Component
public class FinancialEventHandlers {

    // Subscribe to all trade capture events
    @EventHandler
    public void handleTradeCaptured(
            @EventPattern("trading.*.capture.*.") BiTemporalEvent<TradeCapturedPayload> event) {

        // Update risk positions
        riskService.updatePosition(event);

        // Generate settlement instructions
        settlementService.createSettlementInstruction(event);

        // Check regulatory thresholds
        regulatoryService.checkThresholds(event);
    }

    // Subscribe to high-value transactions across all domains
    @EventHandler
    public void handleHighValueTransactions(
            @EventPattern("*.*.*.*.high") BiTemporalEvent<BaseFinancialEventPayload> event) {

        // Enhanced monitoring for large transactions
        monitoringService.trackHighValueTransaction(event);

        // Compliance review
        complianceService.reviewTransaction(event);
    }

    // Subscribe to all failed events for exception management
    @EventHandler
    public void handleFailedEvents(
            @EventPattern("*.*.*.*.") BiTemporalEvent<BaseFinancialEventPayload> event) {

        if ("FAILED".equals(event.getPayload().getStatus())) {
            exceptionService.createException(event);
        }
    }
}
```

Bi-temporal Query Patterns

```
@Service
public class FinancialEventQueryService {

    private final EventStore<BaseFinancialEventPayload> eventStore;

    // Query events as they were known at a specific point in time
    public List<BiTemporalEvent<BaseFinancialEventPayload>> getEventsAsOfSystemTime(
            String aggregateId, Instant systemTime) {

        return eventStore.findByAggregateId(aggregateId)
            .asOfSystemTime(systemTime)
            .toList();
    }

    // Query events that were valid during a specific business time period
    public List<BiTemporalEvent<BaseFinancialEventPayload>> getEventsValidDuring(
            String aggregateId, Instant validFrom, Instant validTo) {

        return eventStore.findByAggregateId(aggregateId)
            .validTimeBetween(validFrom, validTo)
            .toList();
    }

    // Query for audit trail - show all corrections and their history
    public List<BiTemporalEvent<BaseFinancialEventPayload>> getAuditTrail(
            String aggregateId) {

        return eventStore.findByAggregateId(aggregateId)
            .includeCorrections()
            .orderBySystemTime()
            .toList();
    }
}
```

Implementation Examples

Complete Trade Lifecycle Example

```
// 1. Trade Capture
@EventHandler
public void handleTradeExecution(TradeExecutionEvent execution) {
    // Publish trade captured event
    publishEvent("trade.capture.completed",
        TradeCapturedPayload.from(execution));
}

// 2. Trade Confirmation
@EventHandler
public void handleTradeCaptured(
```

```java
        @EventPattern("trade.capture.completed") BiTemporalEvent<TradeCapturedPayload> event) {

    // Send confirmation to counterparty
    confirmationService.sendConfirmation(event.getPayload());

    // Publish confirmation requested event
    publishEvent("trade.confirmation.requested",
        TradeConfirmationPayload.from(event.getPayload()));
}

// 3. Settlement Instruction Generation
@EventHandler
public void handleTradeConfirmed(
        @EventPattern("trade.confirmation.received") BiTemporalEvent<TradeConfirmationPayload> event) {

    // Generate settlement instruction
    SettlementInstruction instruction = settlementService.createInstruction(event.getPayload());

    // Publish settlement instruction created event
    publishEvent("instruction.settlement.created",
        SettlementInstructedPayload.from(instruction));
}

// 4. Position Update
@EventHandler
public void handleSettlementCompleted(
        @EventPattern("instruction.settlement.completed") BiTemporalEvent<SettlementInstructedPayload> event) {

    // Update positions
    positionService.updatePosition(event.getPayload());

    // Publish position update event
    publishEvent("position.update.applied",
        PositionUpdatedPayload.from(event.getPayload()));
}
```

Exception Handling Example

```java
@EventHandler
public void handleSettlementFailure(
        @EventPattern("instruction.settlement.failed") BiTemporalEvent<SettlementInstructedPayload> event) {

    // Create exception record
    Exception exception = exceptionService.createException(
        event.getPayload().getInstructionId(),
        "SETTLEMENT_FAILURE",
        event.getPayload().getFailureReason()
    );

    // Publish exception detected event
    publishEvent("exception.detection.automated",
        ExceptionManagedPayload.from(exception));

    // Assign to operations team
    operationsService.assignException(exception.getId());

    // Publish exception assignment event
    publishEvent("exception.assignment.completed",
        ExceptionManagedPayload.from(exception));
}
```

Migration Strategy

Phase 1: Foundation (2-3 weeks)

1. **Event Store Setup**: Configure PeeGeeQ bi-temporal event stores for each domain
2. **Schema Registry**: Set up Avro schema registry with compatibility rules
3. **Base Event Types**: Implement core event types (trade capture, settlement instruction)
4. **Routing Infrastructure**: Set up routing key generation and subscription patterns

Phase 2: Core Business Events (3-4 weeks)

1. **Trading Events**: Implement complete trade lifecycle events
2. **Settlement Events**: Add settlement instruction and confirmation events
3. **Position Events**: Implement position update and reconciliation events
4. **Cross-Domain Integration**: Connect trading → settlement → position workflows

Phase 3: Fund Administration & Securities Services (4-5 weeks)

1. **Fund Administration**: NAV calculation, subscription/redemption processing
2. **Securities Services**: DVP/FOP settlement, securities lending, safekeeping
3. **Operational Events**: Exception management, manual repairs, reconciliation

Phase 4: Advanced Features (2-3 weeks)

1. **Regulatory Events**: Compliance monitoring, regulatory reporting
2. **Corporate Actions**: Dividend processing, proxy voting, entitlements
3. **Advanced Queries**: Bi-temporal analytics, audit trails, forensic analysis

Phase 5: Production Optimization (1-2 weeks)

1. **Performance Tuning**: Optimize event store performance and indexing
2. **Monitoring**: Set up comprehensive monitoring and alerting
3. **Documentation**: Complete API documentation and runbooks

Migration Best Practices

1. **Incremental Adoption**: Start with one domain, expand gradually
2. **Dual Publishing**: Run old and new systems in parallel during transition
3. **Schema Evolution**: Use backward-compatible schema changes
4. **Testing Strategy**: Comprehensive integration testing with bi-temporal scenarios
5. **Rollback Plan**: Maintain ability to rollback to previous system

This restructured document provides a clear, logical progression from basic concepts through detailed implementation, making it much easier to understand and follow for both business and technical stakeholders.