План поэтапной миграции грузинской бухгалтерской системы

🎯 Общая стратегия миграции

Подход: Strangler Fig Pattern - постепенная замена компонентов без остановки работы системы

Продолжительность: 18-24 месяца

Бюджет: Распределен поэтапно для минимизации рисков

📋 Фаза 0: Подготовка и анализ (Месяцы 1-2)

Цели:

- Аудит текущего состояния
- Подготовка инфраструктуры
- Создание команды миграции

Задачи:

Неделя 1-2: Технический аудит

```bash
# Анализ кода
- Инвентаризация всех модулей и зависимостей
- Оценка тестового покрытия (текущий: ~10%, цель: 85%)
- Выявление критических узких мест производительности
- Анализ данных: объемы, структура, quality issues
```

Неделя 3-4: Создание MVP инфраструктуры

```yaml
```

```yaml
# docker-compose-migration.yml
version: '3.8'
services:
  # Текущая система (legacy)
  legacy-app:
    build: ./legacy
    ports:
      - "8080:8080"
    networks:
      - migration-network

  # Новая система (target)
  new-api-gateway:
    build: ./new-system/gateway
    ports:
      - "8000:8000"
    networks:
      - migration-network

  # Shared resources
  postgres-new:
    image: postgres:15
    environment:
      POSTGRES_DB: accounting_new
    networks:
      - migration-network

  redis:
    image: redis:7-alpine
    networks:
      - migration-network

networks:
  migration-network:
    driver: bridge
```

Неделя 5-6: Создание Data Pipeline

```
python
```

```
# migration/data_sync.py
from sqlalchemy import create_engine
import asyncpg
import pandas as pd

class DataSynchronizer:
    def __init__(self):
        self.legacy_engine = create_engine('postgresql://legacy_db')
        self.new_pool = None

    async def sync_accounts(self):
        """Синхронизация справочника счетов"""
        df = pd.read_sql("SELECT * FROM chart_of_accounts", self.legacy_engine)

        # Transform data
        df['id'] = df.apply(lambda x: uuid4(), axis=1)
        df['created_at'] = pd.Timestamp.now()

        # Load to new system
        async with self.new_pool.acquire() as conn:
            await conn.executemany(
                "INSERT INTO accounts (id, code, name, type) VALUES ($1, $2, $3, $4)",
                df[['id', 'code', 'name', 'account_type']].values.tolist()
            )
```

Неделя 7-8: CI/CD Pipeline

```
yaml
```

```
# .github/workflows/migration.yml
name: Migration Pipeline
on:
  push:
    branches: [main, migration/*]

jobs:
  test-legacy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Test Legacy System
        run: |
          docker-compose -f legacy/docker-compose.test.yml up --abort-on-container-exit

  test-new-system:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Test New System
        run: |
          docker-compose -f new-system/docker-compose.test.yml up --abort-on-container-exit

  deploy-staging:
    needs: [test-legacy, test-new-system]
    runs-on: ubuntu-latest
    steps:
      - name: Deploy to Staging
        run: |
          kubectl apply -f k8s/staging/
```

Результаты Фазы 0:

- ✅ Детальный план миграции с временными рамками
- ✅ Настроенная инфраструктура для параллельной работы систем
- ✅ Команда готова к миграции (3-5 разработчиков)
- ✅ Базовая система мониторинга и алертов

🔧 Фаза 1: Модернизация фундамента (Месяцы 3-6)

Цели:

- Замена устаревших компонентов
- Повышение безопасности
- Улучшение производительности

1.1 Модернизация базы данных (Месяц 3)

Неделя 1-2: Новая схема БД

```sql
-- migrations/001_new_schema.sql
CREATE EXTENSION IF NOT EXISTS "uuid-ossp";
CREATE EXTENSION IF NOT EXISTS "pg_stat_statements";

-- Новая таблица счетов с UUID
CREATE TABLE accounts_v2 (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    code VARCHAR(20) NOT NULL,
    name VARCHAR(100) NOT NULL,
    account_type account_type_enum NOT NULL,
    parent_id UUID REFERENCES accounts_v2(id),
    is_active BOOLEAN DEFAULT TRUE,
    metadata JSONB DEFAULT '{}',
    created_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),
    updated_at TIMESTAMPTZ DEFAULT NOW(),
    version INTEGER DEFAULT 1,

    -- Индексы для производительности
    CONSTRAINT unique_code_per_company UNIQUE (code, company_id)
);

CREATE INDEX idx_accounts_v2_code ON accounts_v2(code);
CREATE INDEX idx_accounts_v2_type ON accounts_v2(account_type);
CREATE INDEX idx_accounts_v2_parent ON accounts_v2(parent_id);
```

Неделя 3-4: Партиционирование для больших таблиц

```sql
```

```sql
-- Партиционирование журнала проводок по месяцам
CREATE TABLE journal_entries_v2 (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    transaction_id UUID NOT NULL,
    account_id UUID NOT NULL REFERENCES accounts_v2(id),
    debit DECIMAL(15,2) DEFAULT 0.00,
    credit DECIMAL(15,2) DEFAULT 0.00,
    currency_code CHAR(3) NOT NULL DEFAULT 'GEL',
    exchange_rate DECIMAL(15,5) DEFAULT 1.00000,
    description TEXT,
    transaction_date DATE NOT NULL,
    created_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),

    CONSTRAINT check_debit_credit CHECK (
        (debit > 0 AND credit = 0) OR (credit > 0 AND debit = 0)
    )
) PARTITION BY RANGE (transaction_date);

-- Создание партиций на год вперед
CREATE TABLE journal_entries_202501 PARTITION OF journal_entries_v2
    FOR VALUES FROM ('2025-01-01') TO ('2025-02-01');
-- ... остальные месяцы
```

1.2 Новый API слой (Месяц 4)

FastAPI с современными паттернами

```
python
```

```
# new_system/api/main.py
from fastapi import FastAPI, Depends, HTTPException, BackgroundTasks
from fastapi.security import HTTPBearer
from fastapi.middleware.cors import CORSMiddleware
from contextlib import asynccontextmanager
import structlog

logger = structlog.get_logger()

@asynccontextmanager
async def lifespan(app: FastAPI):
    # Startup
    await init_database_pool()
    await init_kafka_producer()
    logger.info("Application started")

    yield

    # Shutdown
    await close_database_pool()
    await close_kafka_producer()
    logger.info("Application stopped")

app = FastAPI(
    title="Georgian Accounting System v2.0",
    description="Modern IFRS-compliant accounting system",
    version="2.0.0",
    lifespan=lifespan
)

app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],  # Configure properly in production
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# API Routes
from .routes import accounts, transactions, reports

app.include_router(accounts.router, prefix="/api/v1/accounts")
app.include_router(transactions.router, prefix="/api/v1/transactions")
app.include_router(reports.router, prefix="/api/v1/reports")
```

Современная архитектура с DI

```
python
```

```
# new_system/core/dependencies.py
from dependency_injector import containers, providers
from dependency_injector.wiring import Provide

class Container(containers.DeclarativeContainer):
    # Configuration
    config = providers.Configuration()

    # Database
    db_pool = providers.Singleton(
        create_async_pool,
        config.database.url
    )

    # Repositories
    account_repository = providers.Factory(
        AccountRepository,
        db_pool=db_pool
    )

    transaction_repository = providers.Factory(
        TransactionRepository,
        db_pool=db_pool
    )

    # Services
    accounting_service = providers.Factory(
        AccountingService,
        account_repo=account_repository,
        transaction_repo=transaction_repository
    )

# Dependency injection
async def get_accounting_service(
    service: AccountingService = Depends(Provide[Container.accounting_service])
) -> AccountingService:
    return service
```

1.3 Система безопасности (Месяц 5)

JWT с refresh tokens

python

```python
# new_system/auth/jwt_handler.py
from jose import JWTError, jwt
from datetime import datetime, timedelta
import secrets

class JWTHandler:
    def __init__(self, secret_key: str):
        self.secret_key = secret_key
        self.algorithm = "HS256"
        self.access_token_expire = timedelta(minutes=30)
        self.refresh_token_expire = timedelta(days=7)

    async def create_tokens(self, user_id: str, permissions: List[str]) -> TokenPair:
        access_payload = {
            "sub": user_id,
            "permissions": permissions,
            "type": "access",
            "exp": datetime.utcnow() + self.access_token_expire,
            "iat": datetime.utcnow(),
            "jti": secrets.token_hex(16)  # JWT ID для отзыва
        }

        refresh_payload = {
            "sub": user_id,
            "type": "refresh",
            "exp": datetime.utcnow() + self.refresh_token_expire,
            "iat": datetime.utcnow(),
            "jti": secrets.token_hex(16)
        }

        access_token = jwt.encode(access_payload, self.secret_key, self.algorithm)
        refresh_token = jwt.encode(refresh_payload, self.secret_key, self.algorithm)

        # Сохранить refresh token в Redis с TTL
        await self.redis.setex(f"refresh:{refresh_payload['jti']}",
                               int(self.refresh_token_expire.total_seconds()),
                               user_id)

        return TokenPair(
            access_token=access_token,
            refresh_token=refresh_token,
            expires_in=int(self.access_token_expire.total_seconds())
        )
```

Role-Based Access Control

python

```
# new_system/auth/rbac.py
from enum import Enum
from dataclasses import dataclass
from typing import Set

class Permission(Enum):
    ACCOUNTS_READ = "accounts:read"
    ACCOUNTS_WRITE = "accounts:write"
    TRANSACTIONS_READ = "transactions:read"
    TRANSACTIONS_WRITE = "transactions:write"
    TRANSACTIONS_APPROVE = "transactions:approve"
    REPORTS_FINANCIAL = "reports:financial"
    REPORTS_TAX = "reports:tax"
    ADMIN_USERS = "admin:users"
    ADMIN_SYSTEM = "admin:system"

@dataclass
class Role:
    name: str
    permissions: Set[Permission]

class GeorgianAccountingRoles:
    ACCOUNTANT = Role("accountant", {
        Permission.ACCOUNTS_READ,
        Permission.TRANSACTIONS_READ,
        Permission.TRANSACTIONS_WRITE,
        Permission.REPORTS_FINANCIAL
    })

    CHIEF_ACCOUNTANT = Role("chief_accountant", {
        *ACCOUNTANT.permissions,
        Permission.TRANSACTIONS_APPROVE,
        Permission.REPORTS_TAX,
        Permission.ADMIN_USERS
    })

    TAX_SPECIALIST = Role("tax_specialist", {
        Permission.ACCOUNTS_READ,
        Permission.TRANSACTIONS_READ,
        Permission.REPORTS_TAX
    })

def require_permission(permission: Permission):
    def decorator(func):
        @wraps(func)
        async def wrapper(*args, current_user = Depends(get_current_user), **kwargs):
            if permission not in current_user.permissions:
                raise HTTPException(403, "Insufficient permissions")
            return await func(*args, **kwargs, current_user=current_user)
        return wrapper
    return decorator
```

1.4 Мониторинг и логирование (Месяц 6)

Структурированное логирование

python

```
# new_system/core/logging.py
import structlog
from pythonjsonlogger import jsonlogger

def setup_logging():
    structlog.configure(
        processors=[
            structlog.stdlib.filter_by_level,
            structlog.stdlib.add_logger_name,
            structlog.stdlib.add_log_level,
            structlog.stdlib.PositionalArgumentsFormatter(),
            structlog.processors.TimeStamper(fmt="iso"),
            structlog.processors.StackInfoRenderer(),
            structlog.processors.format_exc_info,
            structlog.processors.UnicodeDecoder(),
            structlog.processors.JSONRenderer()
        ],
        context_class=dict,
        logger_factory=structlog.stdlib.LoggerFactory(),
        wrapper_class=structlog.stdlib.BoundLogger,
        cache_logger_on_first_use=True,
    )

# Использование в коде
logger = structlog.get_logger()

async def create_transaction(transaction_data: TransactionCreate):
    logger.info(
        "Transaction creation started",
        transaction_id=transaction_data.id,
        user_id=current_user.id,
        amount=float(transaction_data.total_amount)
    )

    try:
        result = await service.create_transaction(transaction_data)
        logger.info(
            "Transaction created successfully",
            transaction_id=result.id,
            duration_ms=(time.time() - start_time) * 1000
        )
        return result
    except Exception as e:
        logger.error(
            "Transaction creation failed",
            error=str(e),
            transaction_data=transaction_data.dict()
        )
        raise
```

Prometheus метрики

python

```python
# new_system/core/metrics.py
from prometheus_client import Counter, Histogram, Gauge
import time

# Business metrics
transaction_counter = Counter(
    'accounting_transactions_total',
    'Total number of accounting transactions',
    ['status', 'transaction_type']
)

transaction_amount_histogram = Histogram(
    'accounting_transaction_amount_gel',
    'Distribution of transaction amounts in GEL',
    buckets=[10, 50, 100, 500, 1000, 5000, 10000, 50000, float('inf')]
)

account_balance_gauge = Gauge(
    'accounting_account_balance_gel',
    'Current account balance in GEL',
    ['account_code', 'account_type']
)

# Technical metrics
request_duration = Histogram(
    'http_request_duration_seconds',
    'HTTP request duration',
    ['method', 'endpoint', 'status']
)

class MetricsMiddleware:
    async def __call__(self, request, call_next):
        start_time = time.time()
        response = await call_next(request)
        duration = time.time() - start_time

        request_duration.labels(
            method=request.method,
            endpoint=request.url.path,
            status=response.status_code
        ).observe(duration)
    return response
```

Результаты Фазы 1:

- ✅ Новая БД схема с UUID и партиционированием
- ✅ Modern FastAPI с async/await
- ✅ JWT authentication с RBAC
- ✅ Структурированное логирование и метрики
- ✅ 60% покрытие тестами новых компонентов

🏗️ Фаза 2: Внедрение Event-Driven Architecture (Месяцы 7-10)

Цели:

- Реализация Event Sourcing для audit trail
- CQRS для разделения чтения/записи
- Интеграция с Kafka

2.1 Event Store и Event Sourcing (Месяц 7)

Event Store на PostgreSQL

```sql
```

```
-- events/001_event_store.sql
CREATE TABLE event_store (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    aggregate_id UUID NOT NULL,
    aggregate_type VARCHAR(100) NOT NULL,
    event_type VARCHAR(100) NOT NULL,
    event_data JSONB NOT NULL,
    event_metadata JSONB DEFAULT '{}',
    version INTEGER NOT NULL,
    timestamp TIMESTAMPTZ NOT NULL DEFAULT NOW(),

    CONSTRAINT unique_version_per_aggregate UNIQUE (aggregate_id, version)
);

CREATE INDEX idx_event_store_aggregate ON event_store(aggregate_id);
CREATE INDEX idx_event_store_type ON event_store(event_type);
CREATE INDEX idx_event_store_timestamp ON event_store(timestamp);

-- Snapshots для производительности
CREATE TABLE aggregate_snapshots (
    aggregate_id UUID PRIMARY KEY,
    aggregate_type VARCHAR(100) NOT NULL,
    snapshot_data JSONB NOT NULL,
    version INTEGER NOT NULL,
    timestamp TIMESTAMPTZ NOT NULL DEFAULT NOW()
);
```

Базовые классы для Event Sourcing

```
python
```

```
# new_system/events/base.py
from dataclasses import dataclass
from typing import Any, List, Dict
from abc import ABC, abstractmethod
import uuid
from datetime import datetime

@dataclass(frozen=True)
class DomainEvent:
    """Базовый класс для доменных событий"""
    aggregate_id: uuid.UUID
    event_id: uuid.UUID
    event_type: str
    event_data: Dict[str, Any]
    version: int
    timestamp: datetime
    metadata: Dict[str, Any]

class Aggregate(ABC):
    """Базовый класс для агрегатов"""
    def __init__(self, aggregate_id: uuid.UUID):
        self.id = aggregate_id
        self.version = 0
        self.uncommitted_events: List[DomainEvent] = []

    def apply_event(self, event: DomainEvent):
        """Применить событие к агрегату"""
        self._apply_event(event)
        if event.version > self.version:
            self.version = event.version

    def raise_event(self, event_type: str, event_data: Dict[str, Any], metadata: Dict[str, Any] = None):
        """Поднять новое событие"""
        event = DomainEvent(
            aggregate_id=self.id,
            event_id=uuid.uuid4(),
            event_type=event_type,
            event_data=event_data,
            version=self.version + 1,
            timestamp=datetime.utcnow(),
            metadata=metadata or {}
        )
        self.uncommitted_events.append(event)
        self.apply_event(event)

    @abstractmethod
    def _apply_event(self, event: DomainEvent):
        """Применить событие к состоянию агрегата"""
        pass

    def mark_events_as_committed(self):
        """Пометить события как сохраненные"""
        self.uncommitted_events.clear()
```

Accounting Aggregate

python

```
# new_system/domain/aggregates.py
from decimal import Decimal
from dataclasses import dataclass
from typing import List, Optional

@dataclass
class JournalEntryData:
    account_id: uuid.UUID
    debit: Decimal
    credit: Decimal
    description: str

class AccountingTransaction(Aggregate):
    """Агрегат бухгалтерской транзакции"""

    def __init__(self, aggregate_id: uuid.UUID):
        super().__init__(aggregate_id)
        self.transaction_date: Optional[datetime] = None
        self.description: str = ""
        self.entries: List[JournalEntryData] = []
        self.status: str = "draft"
        self.total_debit: Decimal = Decimal('0.00')
        self.total_credit: Decimal = Decimal('0.00')

    def create_transaction(self, transaction_date: datetime, description: str, entries: List[JournalEntryData]):
        """Создать новую транзакцию"""
        if self.status != "":
            raise ValueError("Transaction already exists")

        # Валидация двойной записи
        total_debit = sum(entry.debit for entry in entries)
        total_credit = sum(entry.credit for entry in entries)

        if total_debit != total_credit:
            raise ValueError(f"Unbalanced transaction: debit={total_debit}, credit={total_credit}")

        self.raise_event("TransactionCreated", {
            "transaction_date": transaction_date.isoformat(),
            "description": description,
            "entries": [
                {
                    "account_id": str(entry.account_id),
                    "debit": str(entry.debit),
                    "credit": str(entry.credit),
                    "description": entry.description
                }
                for entry in entries
            ],
            "total_amount": str(total_debit)
        })

    def approve_transaction(self, approved_by: uuid.UUID):
        """Одобрить транзакцию"""
        if self.status != "draft":
            raise ValueError(f"Cannot approve transaction with status: {self.status}")

        self.raise_event("TransactionApproved", {
            "approved_by": str(approved_by),
            "approved_at": datetime.utcnow().isoformat()
        })

    def post_transaction(self, posted_by: uuid.UUID):
        """Провести транзакцию"""
        if self.status != "approved":
            raise ValueError(f"Cannot post transaction with status: {self.status}")

        self.raise_event("TransactionPosted", {
            "posted_by": str(posted_by),
            "posted_at": datetime.utcnow().isoformat()
        })

    def _apply_event(self, event: DomainEvent):
        """Применить событие к состоянию транзакции"""
        if event.event_type == "TransactionCreated":
            self.transaction_date = datetime.fromisoformat(event.event_data["transaction_date"])
            self.description = event.event_data["description"]
            self.entries = [
                JournalEntryData(
                    account_id=uuid.UUID(entry["account_id"]),
                    debit=Decimal(entry["debit"]),
                    credit=Decimal(entry["credit"]),
                    description=entry["description"]
                )
                for entry in event.event_data["entries"]
            ]
            self.total_debit = self.total_credit = Decimal(event.event_data["total_amount"])
            self.status = "draft"
        elif event.event_type == "TransactionApproved":
            self.status = "approved"

        elif event.event_type == "TransactionPosted":
            self.status = "posted"
```

2.2 CQRS Implementation (Месяц 8)

Command и Query разделение

python

```
# new_system/cqrs/commands.py
from dataclasses import dataclass
from abc import ABC, abstractmethod

class Command(ABC):
    """Базовый класс для команд"""
    pass

class CommandHandler(ABC):
    @abstractmethod
    async def handle(self, command: Command) -> Any:
        pass

@dataclass
class CreateTransactionCommand(Command):
    transaction_date: datetime
    description: str
    entries: List[JournalEntryData]
    created_by: uuid.UUID

class CreateTransactionHandler(CommandHandler):
    def __init__(self, event_store: EventStore, event_bus: EventBus):
        self.event_store = event_store
        self.event_bus = event_bus

    async def handle(self, command: CreateTransactionCommand) -> uuid.UUID:
        # Создать агрегат
        transaction_id = uuid.uuid4()
        transaction = AccountingTransaction(transaction_id)

        # Выполнить бизнес-логику
        transaction.create_transaction(
            command.transaction_date,
            command.description,
            command.entries
        )

        # Сохранить события
        await self.event_store.save_events(
            transaction.id,
            transaction.uncommitted_events,
            expected_version=0
        )
        # Опубликовать события
        for event in transaction.uncommitted_events:
            await self.event_bus.publish(event)

        transaction.mark_events_as_committed()
        return transaction.id
```

Query side (Read Models)

python

```
# new_system/cqrs/queries.py
@dataclass
class AccountBalanceQuery:
    account_id: uuid.UUID
    as_of_date: Optional[datetime] = None

class AccountBalanceQueryHandler:
    def __init__(self, read_db_pool):
        self.read_db = read_db_pool

    async def handle(self, query: AccountBalanceQuery) -> Decimal:
        async with self.read_db.acquire() as conn:
            if query.as_of_date:
                result = await conn.fetchval(
                    """
                    SELECT balance FROM account_balances_history
                    WHERE account_id = $1 AND date <= $2
                    ORDER BY date DESC LIMIT 1
                    """,
                    query.account_id,
                    query.as_of_date
                )
            else:
                result = await conn.fetchval(
                    "SELECT current_balance FROM account_balances WHERE account_id = $1",
                    query.account_id
                )

            return Decimal(str(result or '0.00'))

# Projection для поддержания read models
class AccountBalanceProjection:
    def __init__(self, read_db_pool):
        self.read_db = read_db_pool

    async def handle_transaction_posted(self, event: DomainEvent):
        """Обновить балансы счетов при проведении транзакции"""
        entries = event.event_data["entries"]

        async with self.read_db.acquire() as conn:
            async with conn.transaction():
                for entry_data in entries:
                    account_id = uuid.UUID(entry_data["account_id"])
                    debit = Decimal(entry_data["debit"])
                    credit = Decimal(entry_data["credit"])

                    # Обновить текущий баланс
                    await conn.execute(
                        """
                        INSERT INTO account_balances (account_id, current_balance, last_updated)
                        VALUES ($1, $2, $3)
                        ON CONFLICT (account_id) DO UPDATE SET
                            current_balance = account_balances.current_balance + $2,
                            last_updated = $3
                        """,
                        account_id,
                        debit - credit,
                        event.timestamp
                    )

                    # Добавить историческую запись
                    await conn.execute(
                        """
                        INSERT INTO account_balance_history
                        (account_id, date, balance_change, running_balance, transaction_id)
                        VALUES ($1, $2, $3,
                            (SELECT current_balance FROM account_balances WHERE account_id = $1),
                            $4)
                        """,
                        account_id,
                        event.timestamp.date(),
                        debit - credit,
                        event.aggregate_id
                    )
```

2.3 Kafka Integration (Месяц 9)

Event Bus с Kafka

python

```
# new_system/infrastructure/event_bus.py
from aiokafka import AIOKafkaProducer, AIOKafkaConsumer
import json
from typing import Dict, Callable

class KafkaEventBus:
    def __init__(self, bootstrap_servers: str):
        self.bootstrap_servers = bootstrap_servers
        self.producer: Optional[AIOKafkaProducer] = None
        self.consumer = AIOKafkaConsumer(
            *topics,
            bootstrap_servers=self.bootstrap_servers,
            group_id="accounting-system",
            value_deserializer=lambda m: json.loads(m.decode('utf-8'))
        )

        await self.consumer.start()

        try:
            async for msg in self.consumer:
                await self._handle_message(msg)
        finally:
            await self.consumer.stop()

    async def _handle_message(self, msg):
        """Обработать входящее сообщение"""
        try:
            event_data = msg.value
            event_type = event_data["event_type"]

            if event_type in self.handlers:
                for handler in self.handlers[event_type]:
                    await handler(event_data)

        except Exception as e:
            logger.error(
                "Event handling failed",
                error=str(e),
                topic=msg.topic,
                partition=msg.partition,
                offset=msg.offset
            )
```

2.4 Georgian Tax Service Integration (Месяц 10)

Real-time VAT reporting

python

```
# new_system/integrations/georgian_tax.py
class GeorgianTaxEventHandler:
    def __init__(self, rs_client: RSApiClient):
        self.rs_client = rs_client

    async def handle_transaction_posted(self, event_data: Dict):
        """Обработать проведенную транзакцию для налогового учета"""
        transaction_id = event_data["aggregate_id"]
        entries = event_data["event_data"]["entries"]

        # Найти VAT-related проводки
        vat_entries = []
        for entry in entries:
            account = await self.get_account_info(entry["account_id"])
            if account.account_type == "VAT_PAYABLE" or account.account_type == "VAT_RECEIVABLE":
                vat_entries.append({
                    "account_id": entry["account_id"],
                    "amount": entry["credit"] if entry["credit"] > 0 else entry["debit"],
                    "type": "payable" if account.account_type == "VAT_PAYABLE" else "receivable"
                })

        if vat_entries:
            # Отправить в Georgian Revenue Service
            await self._notify_rs_about_vat_transaction(transaction_id, vat_entries)

    async def _notify_rs_about_vat_transaction(self, transaction_id: str, vat_entries: List[Dict]):
        """Уведомить RS.ge о VAT транзакции"""
        try:
            payload = {
                "transaction_id": transaction_id,
                "timestamp": datetime.utcnow().isoformat(),
                "vat_entries": vat_entries,
                "company_id": self.company_id
            }

            response = await self.rs_client.post("/api/v1/vat/transactions", payload)

            logger.info(
                "VAT transaction reported to RS.ge",
                transaction_id=transaction_id,
                rs_response_status=response.status
            )
        except Exception as e:
            logger.error(
                "Failed to report VAT transaction to RS.ge",
                transaction_id=transaction_id,
                error=str(e)
            )
            # Отправить в Dead Letter Queue для retry
            await self.send_to_dlq("vat_reporting", payload)
```

Результаты Фазы 2:

- ✅ Event Store с полным audit trail
- ✅ CQRS с разделением read/write моделей
- ✅ Kafka для event streaming
- ✅ Real-time интеграция с Georgian Tax Service
- ✅ 80% покрытие тестами event-driven компонентов

🎯 Фаза 3: Микросервисная архитектура (Месяцы 11-14)

Цели:

- Разделение на независимые сервисы
- API Gateway
- Service Mesh
- Container orchestration

3.1 Декомпозиция на микросервисы (Месяц 11)

Domain-Driven Design подход

```
Bounded Contexts:
├── 📁 accounting-core-service/    # Основные бухгалтерские операции
├── 📁 tax-service/                # Налоговый учет и отчетность
├── 📁 payroll-service/            # Зарплата и кадры
├── 📁 inventory-service/          # Складской учет
├── 📁 reporting-service/          # Финансовая отчетность
├── 📁 compliance-service/         # Соответствие требованиям
├── 📁 integration-service/        # Внешние интеграции
├── 📁 notification-service/       # Уведомления
└── 📁 audit-service/              # Аудит и логирование
```

Accounting Core Service

```
python
```

```python
# services/accounting-core/main.py
from fastapi import FastAPI
from .api import transactions, accounts, fiscal_periods
from .domain import AccountingDomain
from .infrastructure import EventStore, MessageBus

class AccountingCoreService:
    def __init__(self):
        self.app = FastAPI(
            title="Accounting Core Service",
            description="Core accounting operations and journal entries",
            version="1.0.0"
        )

        # Domain layer
        self.domain = AccountingDomain()

        # Infrastructure
        self.event_store = EventStore()
        self.message_bus = MessageBus()

        # API routes
        self.app.include_router(transactions.router, prefix="/transactions")
        self.app.include_router(accounts.router, prefix="/accounts")
        self.app.include_router(fiscal_periods.router, prefix="/fiscal-periods")

        # Health check
        @self.app.get("/health")
        async def health_check():
            return {
                "status": "healthy",
                "service": "accounting-core",
                "version": "1.0.0",
                "dependencies": {
                    "database": await self.check_database(),
                    "event_store": await self.check_event_store(),
                    "message_bus": await self.check_message_bus()
                }
            }

    async def check_database(self) -> str:
        try:
            await self.domain.repository.health_check()
            return "healthy"
        except:
            return "unhealthy"

if __name__ == "__main__":
    import uvicorn
    service = AccountingCoreService()
    uvicorn.run(service.app, host="0.0.0.0", port=8001)
```

Tax Service (Georgian-specific)

python

```python
# services/tax-service/domain/georgian_tax.py
from decimal import Decimal
from datetime import date, datetime
from typing import List, Dict

class GeorgianVATCalculator:
    STANDARD_RATE = Decimal('0.18')  # 18% VAT
    REGISTRATION_THRESHOLD = Decimal('100000.00')  # 100,000 GEL

    def __init__(self):
        self.current_month_turnover = Decimal('0.00')
        self.annual_turnover = Decimal('0.00')

    def calculate_vat(self, net_amount: Decimal, is_exempt: bool = False) -> Dict[str, Decimal]:
        """Рассчитать НДС по грузинским правилам"""
        if is_exempt:
            return {
                "net_amount": net_amount,
                "vat_amount": Decimal('0.00'),
                "gross_amount": net_amount,
                "vat_rate": Decimal('0.00')
            }

        vat_amount = net_amount * self.STANDARD_RATE
        gross_amount = net_amount + vat_amount

        return {
            "net_amount": net_amount,
            "vat_amount": vat_amount,
            "gross_amount": gross_amount,
            "vat_rate": self.STANDARD_RATE
        }

    def check_vat_registration_requirement(self, monthly_turnover: Decimal) -> bool:
        """Проверить необходимость регистрации плательщика НДС"""
        return monthly_turnover >= self.REGISTRATION_THRESHOLD

class GeorgianTaxDeclarationGenerator:
    def __init__(self):
        self.rs_integration = RSIntegrationService()

    async def generate_monthly_vat_declaration(self, company_id: str, year: int, month: int) -> VATDeclaration:
        """Создать месячную декларацию НДС"""
        # Собрать данные за месяц
        transactions = await self.get_vat_transactions(company_id, year, month)

        total_vat_payable = sum(t.vat_amount for t in transactions if t.type == "sale")
        total_vat_deductible = sum(t.vat_amount for t in transactions if t.type == "purchase")
        net_vat = total_vat_payable - total_vat_deductible

        declaration = VATDeclaration(
            company_id=company_id,
            period=f"{year}-{month:02d}",
            total_sales=sum(t.net_amount for t in transactions if t.type == "sale"),
            total_vat_payable=total_vat_payable,
            total_purchases=sum(t.net_amount for t in transactions if t.type == "purchase"),
            total_vat_deductible=total_vat_deductible,
            net_vat_payment=net_vat,
            due_date=date(year, month + 1 if month < 12 else year + 1, 15)
        )

        return declaration

    async def submit_to_rs_ge(self, declaration: VATDeclaration) -> RSSubmissionResult:
        """Отправить декларацию в rs.ge"""
        try:
            response = await self.rs_integration.submit_vat_declaration(declaration)
            return RSSubmissionResult(
                success=True,
                submission_id=response.submission_id,
                receipt_number=response.receipt_number
            )
        except Exception as e:
            logger.error("Failed to submit VAT declaration to RS.ge", error=str(e))
            return RSSubmissionResult(
                success=False,
                error_message=str(e)
            )
```

3.2 API Gateway (Месяц 12)

Kong API Gateway configuration

yaml

```yaml
# kong/kong.yml
_format_version: "3.0"

services:
  - name: accounting-core
    url: http://accounting-core-service:8001
    plugins:
      - name: rate-limiting
        config:
          minute: 1000
          hour: 10000
      - name: jwt
        config:
          secret_is_base64: false
          key_claim_name: kid
      - name: prometheus
        config:
          per_consumer: true

  - name: tax-service
    url: http://tax-service:8002
    plugins:
      - name: rate-limiting
        config:
          minute: 500
          hour: 5000
      - name: jwt
      - name: request-size-limiting
        config:
          allowed_payload_size: 10

routes:
  - name: accounting-transactions
    service: accounting-core
    paths:
      - /api/v1/transactions
    methods:
      - GET
      - POST
      - PUT
    plugins:
      - name: cors
        config:
          origins: ["*"]
          methods: ["GET", "POST", "PUT", "DELETE"]

  - name: georgian-tax
    service: tax-service
    paths:
      - /api/v1/tax
    plugins:
      - name: request-transformer
        config:
          add:
            headers:
              - "X-Georgian-Tax: true"

consumers:
  - username: accounting-system
    custom_id: accounting-system-001
    jwt_secrets:
      - algorithm: HS256
        key: accounting-jwt-key
        secret: ${JWT_SECRET}

plugins:
  - name: prometheus
    config:
      per_consumer: true
      status_code_metrics: true
      latency_metrics: true
      bandwidth_metrics: true
```

API Gateway c authentication

```python
```

```
# api-gateway/main.py
from fastapi import FastAPI, Request, HTTPException, Depends
from fastapi.middleware.cors import CORSMiddleware
import httpx
import jwt
from typing import Dict

class APIGateway:
    def __init__(self):
        self.app = FastAPI(
            title="Georgian Accounting API Gateway",
            description="Central API gateway for microservices",
            version="1.0.0"
        )

        self.service_registry = {
            "accounting": "http://accounting-core-service:8001",
            "tax": "http://tax-service:8002",
            "payroll": "http://payroll-service:8003",
            "reporting": "http://reporting-service:8004"
        }

        self.setup_middleware()
        self.setup_routes()

    def setup_middleware(self):
        self.app.add_middleware(
            CORSMiddleware,
            allow_origins=["*"],
            allow_credentials=True,
            allow_methods=["*"],
            allow_headers=["*"],
        )

        @self.app.middleware("http")
        async def add_security_headers(request: Request, call_next):
            response = await call_next(request)
            response.headers["X-Content-Type-Options"] = "nosniff"
            response.headers["X-Frame-Options"] = "DENY"
            response.headers["X-XSS-Protection"] = "1; mode=block"
            return response

    def setup_routes(self):
        @self.app.api_route("/api/v1/{service_name}/{path:path}", methods=["GET", "POST", "PUT", "DELETE", "PATCH"])
        async def proxy_request(
            service_name: str,
            path: str,
            request: Request,
            current_user = Depends(self.get_current_user)
        ):
            if service_name not in self.service_registry:
                raise HTTPException(status_code=404, detail="Service not found")

            service_url = self.service_registry[service_name]
            target_url = f"{service_url}/{path}"

            # Forward request
            async with httpx.AsyncClient() as client:
                response = await client.request(
                    method=request.method,
                    url=target_url,
                    content=await request.body(),
                    headers={
                        **dict(request.headers),
                        "X-User-ID": str(current_user.id),
                        "X-User-Permissions": ",".join(current_user.permissions)
                    },
                    params=request.query_params,
                    timeout=30.0
                )

            return Response(
                content=response.content,
                status_code=response.status_code,
                headers=dict(response.headers)
            )

    async def get_current_user(self, request: Request):
        """Извлечь текущего пользователя из JWT токена"""
        auth_header = request.headers.get("Authorization")
        if not auth_header or not auth_header.startswith("Bearer "):
            raise HTTPException(401, "Missing or invalid authorization header")

        token = auth_header.split(" ")[1]
        try:
            payload = jwt.decode(token, JWT_SECRET, algorithms=["HS256"])
            return User(
                id=payload["sub"],
                permissions=payload.get("permissions", [])
            )
        except jwt.InvalidTokenError:
            raise HTTPException(401, "Invalid token")
```

3.3 Service Mesh с Istio (Месяц 13)

Istio configuration

```yaml
```

```yaml
# istio/virtual-service.yaml
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: accounting-system
spec:
  http:
  - match:
    - uri:
        prefix: /api/v1/transactions
    route:
    - destination:
        host: accounting-core-service
        port:
          number: 8001
    retries:
      attempts: 3
      perTryTimeout: 10s
    timeout: 30s

  - match:
    - uri:
        prefix: /api/v1/tax
    route:
    - destination:
        host: tax-service
        port:
          number: 8002
    fault:
      delay:
        percentage:
          value: 0.1
        fixedDelay: 2s

---
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: accounting-services
spec:
  host: "*.accounting-system.svc.cluster.local"
  trafficPolicy:
    circuitBreaker:
      consecutiveErrors: 3
      interval: 30s
      baseEjectionTime: 30s
    connectionPool:
      tcp:
        maxConnections: 100
      http:
        http1MaxPendingRequests: 50
        maxRequestsPerConnection: 10
```

Circuit Breaker implementation

python

```python
# shared/circuit_breaker.py
import asyncio
from enum import Enum
from datetime import datetime, timedelta
import logging

class CircuitState(Enum):
    CLOSED = "closed"
    OPEN = "open"
    HALF_OPEN = "half_open"

class CircuitBreaker:
    def __init__(self,
                 failure_threshold: int = 5,
                 timeout: int = 60,
                 success_threshold: int = 2):
        self.failure_threshold = failure_threshold
        self.timeout = timeout
        self.success_threshold = success_threshold

        self.failure_count = 0
        self.success_count = 0
        self.last_failure_time = None
        self.state = CircuitState.CLOSED

    async def call(self, func, *args, **kwargs):
        """Выполнить функцию через circuit breaker"""
        if self.state == CircuitState.OPEN:
            if self._should_attempt_reset():
                self.state = CircuitState.HALF_OPEN
                logging.info("Circuit breaker: Attempting reset")
            else:
                raise CircuitBreakerOpenError("Circuit breaker is open")

        try:
            result = await func(*args, **kwargs)
            self._record_success()
            return result
        except Exception as e:
            self._record_failure()
            raise e

    def _record_success(self):
        """Записать успешный вызов"""
        if self.state == CircuitState.HALF_OPEN:
            self.success_count += 1
            if self.success_count >= self.success_threshold:
                self.state = CircuitState.CLOSED
                self.failure_count = 0
                self.success_count = 0
                logging.info("Circuit breaker: Reset to CLOSED")

    def _record_failure(self):
        """Записать неудачный вызов"""
        self.failure_count += 1
        self.last_failure_time = datetime.utcnow()

        if self.failure_count >= self.failure_threshold:
            self.state = CircuitState.OPEN
            logging.warning("Circuit breaker: Opened due to failures")

    def _should_attempt_reset(self) -> bool:
        """Проверить, следует ли попытаться сбросить circuit breaker"""
        if self.last_failure_time is None:
            return False

        return datetime.utcnow() - self.last_failure_time >= timedelta(seconds=self.timeout)

# Использование в сервисах
class TaxServiceClient:
    def __init__(self):
        self.circuit_breaker = CircuitBreaker()
        self.base_url = "http://tax-service:8002"

    async def calculate_vat(self, amount: Decimal) -> VATCalculation:
        return await self.circuit_breaker.call(self._calculate_vat_impl, amount)

    async def _calculate_vat_impl(self, amount: Decimal) -> VATCalculation:
        async with httpx.AsyncClient() as client:
            response = await client.post(
                f"{self.base_url}/api/v1/vat/calculate",
                json={"amount": str(amount)}
            )
            response.raise_for_status()
            return VATCalculation(**response.json())
```

3.4 Container Orchestration с Kubernetes (Месяц 14)

Kubernetes deployment manifests

```yaml
```

```yaml
# k8s/accounting-core-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: accounting-core-service
  namespace: accounting-system
  labels:
    app: accounting-core
    version: v1
spec:
  replicas: 3
  selector:
    matchLabels:
      app: accounting-core
      version: v1
  template:
    metadata:
      labels:
        app: accounting-core
        version: v1
    spec:
      containers:
      - name: accounting-core
        image: accounting-system/accounting-core:v1.0.0
        ports:
        - containerPort: 8001
        env:
        - name: DATABASE_URL
          valueFrom:
            secretKeyRef:
              name: db-credentials
              key: url
        - name: KAFKA_BROKERS
          value: "kafka:9092"
        - name: REDIS_URL
          value: "redis://redis:6379"
        resources:
          requests:
            memory: "256Mi"
            cpu: "250m"
          limits:
            memory: "512Mi"
            cpu: "500m"
        livenessProbe:
          httpGet:
            path: /health
            port: 8001
          initialDelaySeconds: 30
          periodSeconds: 10
        readinessProbe:
          httpGet:
            path: /health/ready
            port: 8001
          initialDelaySeconds: 5
          periodSeconds: 5
        volumeMounts:
        - name: config
          mountPath: /app/config
          readOnly: true
      volumes:
      - name: config
        configMap:
          name: accounting-core-config

---
apiVersion: v1
kind: Service
metadata:
  name: accounting-core-service
  namespace: accounting-system
  labels:
    app: accounting-core
spec:
  selector:
    app: accounting-core
  ports:
  - name: http
    port: 8001
    targetPort: 8001
  type: ClusterIP

---
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: accounting-core-hpa
  namespace: accounting-system
```

```
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: accounting-core-service
  minReplicas: 2
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70
  - type: Resource
    resource:
      name: memory
      target:
        type: Utilization
        averageUtilization: 80
```

Helm chart для развертывания

yaml

```
# helm/accounting-system/Chart.yaml
apiVersion: v2
name: accounting-system
description: Georgian IFRS-compliant accounting system
type: application
version: 1.0.0
appVersion: "1.0.0"

dependencies:
  - name: postgresql
    version: 12.1.9
    repository: https://charts.bitnami.com/bitnami
  - name: redis
    version: 17.3.7
    repository: https://charts.bitnami.com/bitnami
  - name: kafka
    version: 20.0.6
    repository: https://charts.bitnami.com/bitnami
```

yaml

```yaml
# helm/accounting-system/values.yaml
global:
  imageRegistry: "registry.accounting-system.com"
  imagePullSecrets: []

accountingCore:
  enabled: true
  image:
    repository: accounting-core
    tag: "v1.0.0"
  replicaCount: 3
  resources:
    requests:
      memory: 256Mi
      cpu: 250m
    limits:
      memory: 512Mi
      cpu: 500m

taxService:
  enabled: true
  image:
    repository: tax-service
    tag: "v1.0.0"
  replicaCount: 2
  georgianTax:
    rsApiUrl: "https://api.rs.ge"
    vatRate: 0.18

postgresql:
  enabled: true
  auth:
    postgresPassword: "secure-password"
    database: "accounting"
  primary:
    persistence:
      size: 100Gi
      storageClass: "fast-ssd"

redis:
  enabled: true
  auth:
    enabled: true
    password: "redis-password"

kafka:
  enabled: true
  replicaCount: 3
  persistence:
    size: 50Gi

ingress:
  enabled: true
  className: "nginx"
  annotations:
    cert-manager.io/cluster-issuer: "letsencrypt-prod"
    nginx.ingress.kubernetes.io/rate-limit: "1000"
  hosts:
    - host: api.accounting-system.ge
      paths:
        - path: /
          pathType: Prefix
  tls:
    - secretName: accounting-system-tls
      hosts:
        - api.accounting-system.ge
```

Результаты Фазы 3:

- ✅ 9 независимых микросервисов
- ✅ API Gateway с authentication/authorization
- ✅ Service mesh с Istio
- ✅ Container orchestration в Kubernetes
- ✅ Auto-scaling и self-healing
- ✅ 90% покрытие тестами всех сервисов

🚀 Фаза 4: Cloud Native и Advanced Features (Месяцы 15-18)

Цели:

- Cloud-native deployment
- Machine Learning для fraud detection
- Advanced analytics

- Multi-tenant architecture

4.2 Machine Learning для Fraud Detection (Месяц 16)

ML Pipeline для аномалии в транзакциях

python

```
# ml/fraud_detection/models.py
import numpy as np
import pandas as pd
from sklearn.ensemble import IsolationForest
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
import joblib
from typing import Dict, List, Tuple
import asyncio

class TransactionFraudDetector:
    def __init__(self):
        self.isolation_forest = IsolationForest(
            contamination=0.1,  # 10% аномалий ожидается
            random_state=42,
            n_estimators=200
        )
        self.scaler = StandardScaler()
        self.feature_columns = [
            'amount_gel', 'hour_of_day', 'day_of_week', 'days_since_last_transaction',
            'amount_zscore', 'frequency_last_week', 'account_age_days',
            'transaction_count_today', 'average_transaction_amount'
        ]
        self.is_trained = False

    def extract_features(self, transactions: List[Dict]) -> pd.DataFrame:
        """Извлечь признаки для ML модели"""
        df = pd.DataFrame(transactions)

        # Временные признаки
        df['timestamp'] = pd.to_datetime(df['created_at'])
        df['hour_of_day'] = df['timestamp'].dt.hour
        df['day_of_week'] = df['timestamp'].dt.dayofweek

        # Пользовательские паттерны
        df = df.sort_values(['user_id', 'timestamp'])
        df['days_since_last_transaction'] = df.groupby('user_id')['timestamp'].diff().dt.total_seconds() / (24 * 3600)
        df['days_since_last_transaction'].fillna(0, inplace=True)

        # Статистические признаки
        user_stats = df.groupby('user_id').agg({
            'amount_gel': ['mean', 'std', 'count'],
            'timestamp': ['min']
        }).reset_index()

        user_stats.columns = ['user_id', 'avg_amount', 'std_amount', 'transaction_count', 'first_transaction']
        user_stats['account_age_days'] = (pd.Timestamp.now() - user_stats['first_transaction']).dt.total_seconds() / (24 * 360

        # Объединение с основными данными
        df = df.merge(user_stats[['user_id', 'avg_amount', 'std_amount', 'account_age_days']], on='user_id')

        # Z-score для суммы
        df['amount_zscore'] = np.abs((df['amount_gel'] - df['avg_amount']) / (df['std_amount'] + 1e-6))

        # Частота транзакций за последнюю неделю
        df['frequency_last_week'] = df.groupby('user_id')['timestamp'].transform(
            lambda x: x.rolling('7D').count()
        )

        # Количество транзакций сегодня
        df['transaction_count_today'] = df.groupby(['user_id', df['timestamp'].dt.date]).cumcount() + 1

        return df[self.feature_columns].fillna(0)

    async def train(self, training_data: List[Dict]):
        """Обучить модель на исторических данных"""
        features_df = self.extract_features(training_data)

        # Нормализация признаков
        features_scaled = self.scaler.fit_transform(features_df)

        # Обучение модели
        self.isolation_forest.fit(features_scaled)
        self.is_trained = True

        # Сохранение модели
        joblib.dump(self.isolation_forest, 'models/fraud_detector.joblib')
        joblib.dump(self.scaler, 'models/fraud_scaler.joblib')

        print(f"Model trained on {len(training_data)} transactions")

    async def predict_fraud_probability(self, transaction: Dict) -> float:
        """Предсказать вероятность мошенничества"""
        if not self.is_trained:
            await self.load_model()

        # Извлечение признаков для одной транзакции
        features_df = self.extract_features([transaction])
        features_scaled = self.scaler.transform(features_df)

        # Получение anomaly score (-1 = аномалия, 1 = нормальная)
        anomaly_score = self.isolation_forest.decision_function(features_scaled)[0]

        # Преобразование в вероятность (0-1)
        fraud_probability = max(0, min(1, (1 - anomaly_score) / 2))

        return fraud_probability

    async def load_model(self):
        """Загрузить обученную модель"""
        try:
            self.isolation_forest = joblib.load('models/fraud_detector.joblib')
            self.scaler = joblib.load('models/fraud_scaler.joblib')
            self.is_trained = True
        except FileNotFoundError:
            print("Pre-trained model not found. Training new model...")
            # Здесь можно загрузить исторические данные и обучить модель

class FraudDetectionService:
    def __init__(self):
        self.detector = TransactionFraudDetector()
        self.fraud_threshold = 0.7  # Порог для определения мошенничества

    async def analyze_transaction(self, transaction: Dict) -> Dict:
        """Анализ транзакции на предмет мошенничества"""
        fraud_probability = await self.detector.predict_fraud_probability(transaction)

        risk_level = "low"
        if fraud_probability > self.fraud_threshold:
            risk_level = "high"
        elif fraud_probability > 0.4:
            risk_level = "medium"

        return {
            "transaction_id": transaction["id"],
            "fraud_probability": fraud_probability,
            "risk_level": risk_level,
            "requires_review": fraud_probability > self.fraud_threshold,
            "analysis_timestamp": datetime.utcnow().isoformat()
        }
```

```python
    async def handle_transaction_created_event(self, event_data: Dict):
        """Обработать событие создания транзакции для ML анализа"""
        transaction = event_data["event_data"]
        analysis_result = await self.analyze_transaction(transaction)

        if analysis_result["requires_review"]:
            # Отправить алерт для ручной проверки
            await self.send_fraud_alert(analysis_result)

            # Заблокировать транзакцию до проверки
            await self.flag_transaction_for_review(transaction["id"])

        # Сохранить результат анализа
        await self.save_fraud_analysis(analysis_result)

    async def send_fraud_alert(self, analysis_result: Dict):
        """Отправить уведомление о подозрительной транзакции"""
        alert = {
            "alert_type": "fraud_detection",
            "severity": "high",
            "transaction_id": analysis_result["transaction_id"],
            "fraud_probability": analysis_result["fraud_probability"],
            "message": f"High fraud probability detected: {analysis_result['fraud_probability']:.2%}"
        }

        # Отправка через notification service
        await self.notification_service.send_alert(alert)
```

Real-time ML inference

python

```
# ml/realtime_inference/inference_service.py
from kafka import KafkaConsumer, KafkaProducer
import json
import asyncio
from concurrent.futures import ThreadPoolExecutor
import logging

class RealtimeMLInferenceService:
    def __init__(self):
        self.fraud_detector = FraudDetectionService()
        self.consumer = KafkaConsumer(
            'accounting.transactioncreated',
            bootstrap_servers=['kafka:9092'],
            value_deserializer=lambda m: json.loads(m.decode('utf-8')),
            group_id='ml-fraud-detection'
        )
        self.producer = KafkaProducer(
            bootstrap_servers=['kafka:9092'],
            value_serializer=lambda v: json.dumps(v, default=str).encode('utf-8')
        )
        self.executor = ThreadPoolExecutor(max_workers=10)

    async def start_processing(self):
        """Запуск real-time обработки транзакций"""
        logger.info("Starting ML inference service...")

        loop = asyncio.get_event_loop()

        for message in self.consumer:
            # Обработка в отдельном потоке для неблокирующей работы
            future = loop.run_in_executor(
                self.executor,
                self.process_transaction_message,
                message.value
            )

            # Не ждем завершения, чтобы обрабатывать следующие сообщения
            asyncio.create_task(self.handle_inference_result(future))

    def process_transaction_message(self, event_data: Dict) -> Dict:
        """Обработать сообщение о транзакции"""
        try:
            analysis_result = asyncio.run(
                self.fraud_detector.analyze_transaction(event_data)
            )
            return analysis_result
        except Exception as e:
            logger.error(f"ML inference failed: {e}")
            return {
                "error": str(e),
                "transaction_id": event_data.get("aggregate_id")
            }

    async def handle_inference_result(self, future):
        """Обработать результат ML inference"""
        try:
            result = await future

            # Отправить результат в Kafka
            self.producer.send('ml.fraud-analysis-result', result)

            if result.get("requires_review"):
                # Отправить в high-priority топик для немедленной обработки
                self.producer.send('alerts.high-priority', {
                    "type": "fraud_detection",
                    "data": result
                })

        except Exception as e:
            logger.error(f"Failed to handle inference result: {e}")
```

4.3 Advanced Analytics и BI (Месяц 17)

Real-time Analytics с ClickHouse

python

```
# analytics/clickhouse_client.py
from clickhouse_driver import Client
from typing import List, Dict
import asyncio

class ClickHouseAnalytics:
    def __init__(self, host='clickhouse', port=9000):
        self.client = Client(host=host, port=port)
        self.setup_tables()

    def setup_tables(self):
        """Создать таблицы для аналитики"""
        # Таблица для транзакций (оптимизированная для анализа)
        self.client.execute('''
            CREATE TABLE IF NOT EXISTS transactions_analytics (
                transaction_id UUID,
                transaction_date Date,
                transaction_timestamp DateTime,
                company_id UUID,
                user_id UUID,
                total_amount Decimal(15, 2),
                currency_code String,
                account_debit_id UUID,
                account_credit_id UUID,
                account_debit_type String,
                account_credit_type String,
                is_approved UInt8,
                is_posted UInt8,
                created_at DateTime
            ) ENGINE = MergeTree()
            PARTITION BY toYYYYMM(transaction_date)
            ORDER BY (company_id, transaction_date, transaction_id)
        ''')

        # Материализованное представление для real-time агрегации
        self.client.execute('''
            CREATE MATERIALIZED VIEW IF NOT EXISTS daily_transactions_mv
            TO daily_transactions_summary
            AS SELECT
                company_id,
                transaction_date,
                count() as transaction_count,
                sum(total_amount) as total_amount,
                avg(total_amount) as avg_amount,
                countIf(is_posted = 1) as posted_count,
                sumIf(total_amount, is_posted = 1) as posted_amount
            FROM transactions_analytics
            GROUP BY company_id, transaction_date
        ''')

        # Таблица для КПП (ключевые показатели производительности)
        self.client.execute('''
            CREATE TABLE IF NOT EXISTS kpi_metrics (
                company_id UUID,
                metric_date Date,
                metric_name String,
                metric_value Decimal(15, 2),
                metric_currency String DEFAULT 'GEL',
                created_at DateTime DEFAULT now()
            ) ENGINE = ReplacingMergeTree(created_at)
            PARTITION BY toYYYYMM(metric_date)
            ORDER BY (company_id, metric_date, metric_name)
        ''')

    async def insert_transaction(self, transaction_data: Dict):
        """Вставить данные транзакции для аналитики"""
        await asyncio.to_thread(
            self.client.execute,
            'INSERT INTO transactions_analytics VALUES',
            [transaction_data]
        )

    async def get_financial_kpis(self, company_id: str, start_date: str, end_date: str) -> Dict:
        """Получить финансовые КПП за период"""
        query = '''
            SELECT
                -- Оборот
                sum(total_amount) as total_revenue,
                count() as transaction_count,
                avg(total_amount) as avg_transaction_amount,

                -- По типам счетов
                sumIf(total_amount, account_credit_type = 'REVENUE') as revenue,
                sumIf(total_amount, account_debit_type = 'EXPENSE') as expenses,

                -- Активы и обязательства
                sumIf(total_amount, account_debit_type = 'ASSET') as total_assets_increase,
                sumIf(total_amount, account_credit_type = 'LIABILITY') as total_liabilities_increase,

                -- Рентабельность
                (sumIf(total_amount, account_credit_type = 'REVENUE') -
                 sumIf(total_amount, account_debit_type = 'EXPENSE')) as profit_loss

            FROM transactions_analytics
            WHERE company_id = %(company_id)s
            AND transaction_date BETWEEN %(start_date)s AND %(end_date)s
            AND is_posted = 1
        '''

        result = await asyncio.to_thread(
            self.client.execute,
            query,
            {'company_id': company_id, 'start_date': start_date, 'end_date': end_date}
        )

        return {
            'total_revenue': float(result[0][0] or 0),
            'transaction_count': result[0][1],
            'avg_transaction_amount': float(result[0][2] or 0),
            'revenue': float(result[0][3] or 0),
            'expenses': float(result[0][4] or 0),
            'total_assets_increase': float(result[0][5] or 0),
            'total_liabilities_increase': float(result[0][6] or 0),
            'profit_loss': float(result[0][7] or 0)
        }

class RealTimeAnalyticsService:
    def __init__(self):
        self.clickhouse = ClickHouseAnalytics()
        self.redis_client = redis.Redis(host='redis', port=6379, decode_responses=True)

    async def handle_transaction_posted_event(self, event_data: Dict):
        """Обработать событие проведения транзакции для аналитики"""
        transaction = event_data['event_data']

        # Вставить в ClickHouse для долгосрочной аналитики
        analytics_record = {
            'transaction_id': event_data['aggregate_id'],
            'transaction_date': transaction['transaction_date'][:10],
            'transaction_timestamp': transaction['transaction_date'],
            'company_id': transaction['company_id'],
            'user_id': transaction['created_by'],
            'total_amount': float(transaction['total_amount']),
            'currency_code': transaction.get('currency_code', 'GEL'),
            'is_posted': 1,
            'created_at': datetime.utcnow().strftime('%Y-%m-%d %H:%M:%S')
        }

        await self.clickhouse.insert_transaction(analytics_record)

        # Обновить real-time метрики в Redis
        await self.update_realtime_metrics(transaction)

    async def update_realtime_metrics(self, transaction: Dict):
        """Обновить метрики в реальном времени"""
        company_id = transaction['company_id']
        date_key = transaction['transaction_date'][:10]

        # Счетчики транзакций
        await self.redis_client.hincrby(
            f"metrics:daily:{company_id}:{date_key}",
            "transaction_count",
            1
        )

        # Сумма транзакций
        await self.redis_client.hincrbyfloat(
            f"metrics:daily:{company_id}:{date_key}",
            "total_amount",
            float(transaction['total_amount'])
        )

        # Установить TTL для автоматической очистки старых данных (30 дней)
        await self.redis_client.expire(
            f"metrics:daily:{company_id}:{date_key}",
            30 * 24 * 3600
        )
```

Business Intelligence Dashboard

python

```
# analytics/dashboard_api.py
from fastapi import FastAPI, Depends, Query
from datetime import datetime, timedelta
import plotly.graph_objects as go
import plotly.express as px

class BIDashboardAPI:
    def __init__(self):
        self.app = FastAPI(title="Accounting Analytics Dashboard")
        self.clickhouse = ClickHouseAnalytics()
        self.setup_routes()

    def setup_routes(self):
        @self.app.get("/api/v1/dashboard/overview")
        async def get_overview(
            company_id: str,
            period: str = Query("30d", regex="^(7d|30d|90d|1y)$")
        ):
            """Получить обзорную информацию для dashboard"""
            end_date = datetime.now().date()

            if period == "7d":
                start_date = end_date - timedelta(days=7)
            elif period == "30d":
                start_date = end_date - timedelta(days=30)
            elif period == "90d":
                start_date = end_date - timedelta(days=90)
            else:  # 1y
                start_date = end_date - timedelta(days=365)

            kpis = await self.clickhouse.get_financial_kpis(
                company_id, str(start_date), str(end_date)
            )

            # Тренд за период
            trend_data = await self.get_trend_data(company_id, start_date, end_date)

            return {
                "period": period,
                "kpis": kpis,
                "trends": trend_data,
                "generated_at": datetime.utcnow().isoformat()
            }
```

```
@self.app.get("/api/v1/dashboard/profit-loss-chart")
async def get_profit_loss_chart(company_id: str, period: str = "30d"):
    """График прибыли и убытков"""
    # Получение данных из ClickHouse
    query = '''
        SELECT
            transaction_date,
            sumIf(total_amount, account_credit_type = 'REVENUE') as revenue,
            sumIf(total_amount, account_debit_type = 'EXPENSE') as expenses
        FROM transactions_analytics
        WHERE company_id = %(company_id)s
        AND transaction_date >= today() - 30
        GROUP BY transaction_date
        ORDER BY transaction_date
    '''

    data = await asyncio.to_thread(
        self.clickhouse.client.execute,
        query,
        {'company_id': company_id}
    )

    # Создание Plotly графика
    dates = [row[0] for row in data]
    revenues = [float(row[1] or 0) for row in data]
    expenses = [float(row[2] or 0) for row in data]
    profit = [r - e for r, e in zip(revenues, expenses)]

    fig = go.Figure()

    fig.add_trace(go.Scatter(
        x=dates,
        y=revenues,
        mode='lines+markers',
        name='Доходы',
        line=dict(color='green')
    ))

    fig.add_trace(go.Scatter(
        x=dates,
        y=expenses,
        mode='lines+markers',
        name='Расходы',
            line=dict(color='red')
        ))

        fig.add_trace(go.Scatter(
            x=dates,
            y=profit,
            mode='lines+markers',
            name='Прибыль',
            line=dict(color='blue'),
            fill='tonexty'
        ))

        fig.update_layout(
            title="Динамика прибыли и убытков",
            xaxis_title="Дата",
            yaxis_title="Сумма (ლარი)",
            hovermode='x unified'
        )

        return fig.to_json()

    @self.app.get("/api/v1/dashboard/account-balances")
    async def get_account_balances(company_id: str):
        """Балансы по типам счетов"""
        query = '''
            SELECT
                account_type,
                sum(current_balance) as total_balance
            FROM account_balances ab
            JOIN accounts a ON ab.account_id = a.id
            WHERE a.company_id = %(company_id)s
            GROUP BY account_type
            ORDER BY total_balance DESC
        '''

        # Это запрос к основной PostgreSQL базе, не ClickHouse
        # Здесь нужно использовать соответствующий клиент

        return {
            "account_balances": [
                {"type": "Активы", "balance": 150000.00},
                {"type": "Обязательства", "balance": 75000.00},
                {"type": "Капитал", "balance": 75000.00}
        ]
      }
```

4.4 Multi-tenant Architecture (Месяц 18)

Tenant isolation

python

```python
# tenancy/tenant_context.py
from contextvars import ContextVar
from typing import Optional
import uuid

# Контекстная переменная для текущего тенанта
current_tenant: ContextVar[Optional[str]] = ContextVar('current_tenant', default=None)

class TenantContext:
    def __init__(self, tenant_id: str):
        self.tenant_id = tenant_id
        self.token = None

    def __enter__(self):
        self.token = current_tenant.set(self.tenant_id)
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        if self.token:
            current_tenant.reset(self.token)

class TenantAwareRepository:
    """Базовый класс для tenant-aware репозиториев"""

    def __init__(self, db_pool):
        self.db_pool = db_pool

    def get_tenant_id(self) -> str:
        tenant_id = current_tenant.get()
        if not tenant_id:
            raise ValueError("No tenant context set")
        return tenant_id

    async def execute_query(self, query: str, params: list = None, tenant_filter: bool = True):
        """Выполнить запрос с автоматической фильтрацией по tenant"""
        if tenant_filter and "WHERE" in query.upper():
            # Добавить фильтр по tenant_id
            query = query.replace("WHERE", f"WHERE tenant_id = %s AND", 1)
            params = [self.get_tenant_id()] + (params or [])
        elif tenant_filter:
            # Добавить WHERE clause если его нет
            if "ORDER BY" in query.upper():
                query = query.replace("ORDER BY", "WHERE tenant_id = %s ORDER BY", 1)
            else:
                query += " WHERE tenant_id = %s"
            params = (params or []) + [self.get_tenant_id()]

        async with self.db_pool.acquire() as conn:
            return await conn.fetch(query, *params)

class AccountRepository(TenantAwareRepository):
    async def get_all_accounts(self) -> List[Account]:
        """Получить все счета для текущего тенанта"""
        query = """
            SELECT id, code, name, account_type, parent_id, is_active
            FROM accounts
            ORDER BY code
        """
        rows = await self.execute_query(query)
        return [Account(**dict(row)) for row in rows]

    async def create_account(self, account_data: AccountCreate) -> Account:
        """Создать новый счет для текущего тенанта"""
        query = """
            INSERT INTO accounts (id, tenant_id, code, name, account_type, parent_id, is_active)
            VALUES ($1, $2, $3, $4, $5, $6, $7)
            RETURNING *
        """
        account_id = uuid.uuid4()
        tenant_id = self.get_tenant_id()

        row = await self.execute_query(
            query,
            [account_id, tenant_id, account_data.code, account_data.name,
             account_data.account_type, account_data.parent_id, True],
            tenant_filter=False  # Уже добавили tenant_id вручную
        )
        return Account(**dict(row[0]))

# Middleware для извлечения tenant из запроса
class TenantMiddleware:
    def __init__(self, app):
        self.app = app

    async def __call__(self, scope, receive, send):
        if scope["type"] == "http":
            # Извлечь tenant_id из заголовка или JWT токена
            headers = dict(scope["headers"])
            tenant_id = self.extract_tenant_id(headers)

            if tenant_id:
                with TenantContext(tenant_id):
                    await self.app(scope, receive, send)
            else:
                # Вернуть ошибку если tenant не определен
                response = Response(
                    content="Tenant ID required",
                    status_code=400
                )
                await response(scope, receive, send)
        else:
            await self.app(scope, receive, send)

    def extract_tenant_id(self, headers) -> Optional[str]:
        # Попробовать извлечь из заголовка
        tenant_header = headers.get(b'x-tenant-id')
        if tenant_header:
            return tenant_header.decode()

        # Попробовать извлечь из JWT токена
        auth_header = headers.get(b'authorization')
        if auth_header and auth_header.startswith(b'Bearer '):
            token = auth_header[7:].decode()
            try:
                payload = jwt.decode(token, JWT_SECRET, algorithms=["HS256"])
                return payload.get('tenant_id')
            except:
                pass

        return None
```

Database schema для multi-tenancy

```
sql
```

```sql
-- migration: add_tenant_support.sql

-- Добавить tenant_id ко всем основным таблицам
ALTER TABLE companies ADD COLUMN IF NOT EXISTS tenant_id UUID;
ALTER TABLE users ADD COLUMN IF NOT EXISTS tenant_id UUID;
ALTER TABLE accounts ADD COLUMN IF NOT EXISTS tenant_id UUID;
ALTER TABLE journal_entries ADD COLUMN IF NOT EXISTS tenant_id UUID;
ALTER TABLE invoices ADD COLUMN IF NOT EXISTS tenant_id UUID;
ALTER TABLE products ADD COLUMN IF NOT EXISTS tenant_id UUID;
ALTER TABLE employees ADD COLUMN IF NOT EXISTS tenant_id UUID;

-- Создать таблицу тенантов
CREATE TABLE IF NOT EXISTS tenants (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    name VARCHAR(255) NOT NULL,
    subdomain VARCHAR(100) UNIQUE,
    plan VARCHAR(50) DEFAULT 'basic',
    max_users INTEGER DEFAULT 10,
    max_companies INTEGER DEFAULT 1,
    is_active BOOLEAN DEFAULT TRUE,
    created_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),
    updated_at TIMESTAMPTZ DEFAULT NOW(),

    -- Georgian-specific settings
    default_currency CHAR(3) DEFAULT 'GEL',
    vat_rate DECIMAL(5,4) DEFAULT 0.1800,
    tax_period VARCHAR(20) DEFAULT 'monthly',

    -- Feature flags
    features JSONB DEFAULT '{
        "advanced_reporting": false,
        "api_access": false,
        "multi_currency": true,
        "audit_trail": true,
        "ml_fraud_detection": false
    }'::jsonb
);

-- Подписки тенантов
CREATE TABLE tenant_subscriptions (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    tenant_id UUID NOT NULL REFERENCES tenants(id),
    plan_name VARCHAR(50) NOT NULL,
```

~~~~
    started_at TIMESTAMPTZ NOT NULL,
    expires_at TIMESTAMPTZ,
    is_active BOOLEAN DEFAULT TRUE,
    monthly_price DECIMAL(10,2),
    currency CHAR(3) DEFAULT 'GEL',

    -- Ограничения плана
    limits JSONB DEFAULT '{
        "max_users": 10,
        "max_companies": 1,
        "max_transactions_per_month": 1000,
        "storage_gb": 10
    }'::jsonb,

    created_at TIMESTAMPTZ NOT NULL DEFAULT NOW()
);

-- Обновить существующие индексы
DROP INDEX IF EXISTS idx_accounts_code;
CREATE INDEX idx_accounts_code_tenant ON accounts(tenant_id, code);

DROP INDEX IF EXISTS idx_journal_entries_account;
CREATE INDEX idx_journal_entries_account_tenant ON journal_entries(tenant_id, account_id, transaction_date);

-- Row Level Security (RLS) для дополнительной изоляции
ALTER TABLE accounts ENABLE ROW LEVEL SECURITY;
ALTER TABLE journal_entries ENABLE ROW LEVEL SECURITY;
ALTER TABLE invoices ENABLE ROW LEVEL SECURITY;

-- Политики RLS
CREATE POLICY accounts_tenant_isolation ON accounts
    FOR ALL TO authenticated_users
    USING (tenant_id = current_setting('app.current_tenant_id')::uuid);

CREATE POLICY journal_entries_tenant_isolation ON journal_entries.1 Cloud Deployment (Месяц 15)

#### Infrastructure as Code с Terraform
```hcl
# terraform/main.tf
provider "google" {
  project = var.project_id
  region  = var.region
}
~~~~

```
# GKE Cluster
resource "google_container_cluster" "accounting_cluster" {
  name     = "accounting-system"
  location = var.region

  remove_default_node_pool = true
  initial_node_count       = 1

  network    = google_compute_network.vpc.name
  subnetwork = google_compute_subnetwork.subnet.name

  master_auth {
    client_certificate_config {
      issue_client_certificate = false
    }
  }

  workload_identity_config {
    workload_pool = "${var.project_id}.svc.id.goog"
  }

  addons_config {
    istio_config {
      disabled = false
      auth     = "AUTH_MUTUAL_TLS"
    }
  }
}

# Node pools
resource "google_container_node_pool" "primary_nodes" {
  name       = "primary-node-pool"
  location   = var.region
  cluster    = google_container_cluster.accounting_cluster.name
  node_count = 3

  node_config {
    preemptible  = false
    machine_type = "e2-standard-4"

    service_account = google_service_account.gke_service_account.email
    oauth_scopes = [
      "https://www.googleapis.com/auth/cloud-platform"
    ]
    labels = {
      environment = var.environment
    }

    tags = ["accounting-system-node"]
  }

  autoscaling {
    min_node_count = 2
    max_node_count = 20
  }

  management {
    auto_repair  = true
    auto_upgrade = true
  }
}

# Cloud SQL (PostgreSQL)
resource "google_sql_database_instance" "accounting_db" {
  name             = "accounting-db-${var.environment}"
  database_version = "POSTGRES_14"
  region           = var.region

  settings {
    tier              = "db-standard-4"
    availability_type = "REGIONAL"
    disk_type         = "PD_SSD"
    disk_size         = 500
    disk_autoresize   = true

    backup_configuration {
      enabled                        = true
      start_time                     = "02:00"
      point_in_time_recovery_enabled = true
      transaction_log_retention_days = 7
      backup_retention_settings {
        retained_backups = 30
      }
    }

    ip_configuration {
      ipv4_enabled                                  = false
      private_network                             = google_compute_network.vpc.id
      enable_private_path_for_google_cloud_services = true
    }

    database_flags {
      name  = "max_connections"
      value = "1000"
    }

    insights_config {
      query_insights_enabled  = true
      query_string_length     = 1024
      record_application_tags = true
      record_client_address   = true
    }
  }

  deletion_protection = true
}

# Redis (Memorystore)
resource "google_redis_instance" "accounting_cache" {
  name           = "accounting-cache"
  tier           = "STANDARD_HA"
  memory_size_gb = 16
  region         = var.region

  authorized_network = google_compute_network.vpc.id

  redis_version  = "REDIS_7_0"
  display_name   = "Accounting System Cache"
}
```

GitOps с ArgoCD

```
yaml
```

```yaml
# argocd/applications/accounting-system.yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: accounting-system
  namespace: argocd
spec:
  project: default

  source:
    repoURL: https://github.com/accounting-system/k8s-manifests
    targetRevision: HEAD
    path: environments/production

  destination:
    server: https://kubernetes.default.svc
    namespace: accounting-system

  syncPolicy:
    automated:
      prune: true
      selfHeal: true
      allowEmpty: false
    syncOptions:
      - CreateNamespace=true
      - PrunePropagationPolicy=foreground
      - PruneLast=true
    retry:
      limit: 5
      backoff:
        duration: 5s
        factor: 2
        maxDuration: 3m

  revisionHistoryLimit: 10
```

4: Optional[AIOKafkaConsumer] = None

```python
    self.handlers: Dict[str, List[Callable]] = {}

async def start(self):
    self.producer = AIOKafkaProducer(
        bootstrap_servers=self.bootstrap_servers,
        value_serializer=lambda v: json.dumps(v, default=str).encode('utf-8')
    )
    await self.producer.start()

async def publish(self, event: DomainEvent):
    """Опубликовать событие"""
    topic = f"accounting.{event.event_type.lower()}"

    event_payload = {
        "event_id": str(event.event_id),
        "aggregate_id": str(event.aggregate_id),
        "event_type": event.event_type,
        "event_data": event.event_data,
        "version": event.version,
        "timestamp": event.timestamp.isoformat(),
        "metadata": event.metadata
    }

    await self.producer.send(topic, event_payload)

    logger.info(
        "Event published",
        event_type=event.event_type,
        aggregate_id=str(event.aggregate_id),
        topic=topic
    )

async def subscribe(self, event_type: str, handler: Callable):
    """Подписаться на тип события"""
    if event_type not in self.handlers:
        self.handlers[event_type] = []
    self.handlers[event_type].append(handler)

async def start_consuming(self):
    """Начать обработку событий"""
    topics = [f"accounting.{event_type.lower()}" for event_type in self.handlers.keys()]

    self.consumer
```