

```
profit_factor=self._calculate_profit_factor(trades)  
)
```

День 15: Risk Management 2.0

```
python
```

"""

Задача: Создать продвинутую систему управления рисками

Файл: processing_analysis/advanced_risk_manager.py

"""

class AdvancedRiskManager:

def __init__(self, config: RiskConfig):

 self.max_portfolio_risk = config.max_portfolio_risk

 self.max_position_risk = config.max_position_risk

 self.correlation_threshold = config.correlation_threshold

async def evaluate_trade(

 self,

 proposed_trade: Trade,

 portfolio: Portfolio,

 market_conditions: MarketConditions

) -> RiskDecision:

Параллельная проверка всех риск-факторов

 risk_checks = **await** asyncio.gather(

 self._check_position_sizing(proposed_trade, portfolio),

 self._check_correlation_risk(proposed_trade, portfolio),

 self._check_market_regime(market_conditions),

 self._check_liquidity_risk(proposed_trade),

 self._check_concentration_risk(proposed_trade, portfolio)

)

Kelly Criterion для оптимального размера позиции

 kelly_size = self._calculate_kelly_criterion(

 win_probability=proposed_trade.confidence,

 win_loss_ratio=proposed_trade.expected_rr_ratio

)

Корректировка на основе режима рынка

```
if market_conditions.volatility_regime == "high":  
    kelly_size *= 0.5 # Уменьшаем позицию в волатильном рынке  
  
return RiskDecision(  
    approved=all(risk_checks),  
    position_size=min(kelly_size, self.max_position_risk),  
    stop_loss=self._calculate_dynamic_stop(proposed_trade, market_conditions),  
    take_profit=self._calculate_dynamic_target(proposed_trade, market_conditions),  
    risk_warnings=self._generate_risk_warnings(risk_checks)  
)
```

Фаза 4: Production Infrastructure (Неделя 4)

День 16-17: Monitoring и Alerting

python

"""

Задача: Создать comprehensive monitoring

Файлы:

- infrastructure/monitoring.py
- monitoring_feedback/alert_manager.py
- docker/prometheus.yml
- docker/grafana-dashboards/

"""

Prometheus мемуки

from prometheus_client import Counter, Histogram, Gauge, Summary

class TradingMetrics:

Performance мемуки

```
trades_total = Counter(  
    'trades_total',  
    'Total number of trades',  
    ['symbol', 'side', 'strategy']  
)
```

```
trade_latency = Histogram(  
    'trade_execution_seconds',  
    'Trade execution latency',  
    buckets=[0.1, 0.5, 1.0, 2.0, 5.0]  
)
```

```
portfolio_value = Gauge(  
    'portfolio_value_usd',  
    'Current portfolio value in USD'  
)
```

Health мемуки

```
api_health = Gauge(  
    'api_health',  
    'API health status',  
    ['status']  
)
```

```

        'api_health_status',
        'API health status',
        ['service']
    )

    # Business Mempuku
    daily_pnl = Gauge(
        'daily_pnl_usd',
        'Daily P&L in USD'
    )

    sharpe_ratio = Gauge(
        'strategy_sharpe_ratio',
        'Rolling Sharpe ratio',
        ['strategy', 'timeframe']
    )

    # Alert Manager
    class AlertManager:
        def __init__(self):
            self.slack_client = SlackClient()
            self.telegram_bot = TelegramBot()

        async def check_alerts(self):
            alerts = [
                self._check_drawdown_alert(),
                self._check_api_failure_alert(),
                self._check_position_size_alert(),
                self._check_unusual_market_activity()
            ]

            for alert in await asyncio.gather(*alerts):
                if alert.triggered:

```

```
        await self._send_alert(alert)

    async def _send_alert(self, alert: Alert):
        # Приоритетная отправка
        if alert.severity == "critical":
            await asyncio.gather(
                self.slack_client.send_urgent(alert),
                self.telegram_bot.send_message(alert),
                self._trigger_pagerduty(alert)
            )
```

День 18-19: Тестирование

python

"""

Задача: Создать comprehensive test suite

Файлы:

- tests/unit/test_*.py
- tests/integration/test_*.py
- tests/e2e/test_*.py
- .github/workflows/ci.yml

"""

Пример unit теста с mocking

```
import pytest
```

```
from unittest.mock import AsyncMock, patch
```

```
@pytest.mark.asyncio
```

```
async def test_data_collector_with_cache():
```

```
    # Arrange
```

```
    mock_cache = AsyncMock()
```

```
    mock_cache.get.return_value = {"price": 150.0}
```

```
    collector = AsyncDataCollector(cache=mock_cache)
```

```
    # Act
```

```
    result = await collector.fetch_market_data("AAPL")
```

```
    # Assert
```

```
    assert result["price"] == 150.0
```

```
    mock_cache.get.assert_called_once_with("market_data:AAPL")
```

Integration mecm

```
@pytest.mark.integration
```

```
async def test_full_signal_pipeline():
```

```
    # Используем test containers для Redis/RabbitMQ
```

```
    async with AsyncTestContainers() as containers:
```

```
redis = await containers.start_redis()
rabbit = await containers.start_rabbitmq()

# Настраиваем тестовое окружение
app = create_app(
    redis_url=redis.url,
    rabbit_url=rabbit.url,
    test_mode=True
)

# Запускаем полный pipeline
signals = await app.generate_signals(["AAPL", "GOOGL"])

assert len(signals) == 2
assert all(s.confidence > 0 for s in signals)
```

GitHub Actions CI

name: CI Pipeline

on: [push, pull_request]

jobs:

test:

runs-on: ubuntu-latest

services:

redis:

image: redis:7

postgres:

image: postgres:15

steps:

- uses: actions/checkout@v3

- name: Set up Python

uses: actions/setup-python@v4

with:

python-version: '3.11'

- name: Install dependencies

run: |

pip install -r requirements.txt

pip install -r requirements-dev.txt

- name: Run tests

run: |

pytest tests/ --cov=market_agent --cov-report=xml

- name: Upload coverage

uses: codecov/codecov-action@v3

День 20-21: Deployment и Security

python

"""

Задача: Подготовить production deployment

Файлы:

- docker/Dockerfile.production
- docker/k8s/*.yaml
- security/vault_config.py
- .env.example

"""

Multi-stage Dockerfile для оптимизации

FROM python:3.11-slim as builder

WORKDIR /build

COPY requirements.txt .

RUN pip wheel --no-cache-dir --no-deps --wheel-dir /wheels -r requirements.txt

FROM python:3.11-slim

WORKDIR /app

Security: non-root user

RUN useradd -m -u 1000 trader && \

mkdir -p /app/logs && \

chown -R trader:trader /app

Install dependencies

COPY --from=builder /wheels /wheels

RUN pip install --no-cache /wheels/*

Copy application

COPY --chown=trader:trader . .

USER trader

CMD ["python", "-m", "market_agent.main"]

Kubernetes deployment c security best practices

apiVersion: apps/v1

kind: Deployment

metadata:

name: market-agent

spec:

replicas: 3

template:

spec:

securityContext:

runAsNonRoot: true

runAsUser: 1000

fsGroup: 1000

containers:

- name: market-agent

image: market-agent:latest

securityContext:

allowPrivilegeEscalation: false

readOnlyRootFilesystem: true

capabilities:

drop:

- ALL

env:

- name: VAULT_TOKEN

valueFrom:

secretKeyRef:

name: vault-token

key: token

volumeMounts:

- name: tmp

mountPath: /tmp

- name: logs

mountPath: /app/logs

volumes:

- name: tmp
emptyDir: {}
- name: logs
emptyDir: {}

Фаза 5: Оптимизация и масштабирование (Неделя 5)

День 22-23: Performance Optimization

python

"""

Задача: Оптимизировать критические пути

Файлы:

- core/performance_optimizer.py

- scripts/profile_bottlenecks.py

"""

Профилирование и оптимизация

import cProfile

import pstats

from memory_profiler import profile

class PerformanceOptimizer:

def __init__(self):

self.profiler = cProfile.Profile()

@profile *# Для отслеживания памяти*

async def optimize_data_pipeline(self):

Использование numpy для векторизации

prices = np.array(market_data['prices'])

Vectorized вычисления вместо циклов

returns = np.diff(prices) / prices[:-1]

volatility = np.std(returns) * np.sqrt(252)

Использование numba для JIT компиляции

@numba.jit(nopython=True)

def calculate_indicators(prices, volumes):

Критические вычисления с JIT

pass

Cython для ультра-критичных частей

from .cython_modules import fast_correlation

```
correlation_matrix = fast_correlation(returns)

# Оптимизация запросов к БД
class OptimizedDataStore:
    def __init__(self):
        self.connection_pool = asyncpg.create_pool(
            min_size=10,
            max_size=20,
            command_timeout=60
        )

    async def bulk_insert_trades(self, trades: List[Trade]):
        # Используем COPY для массовой вставки
        async with self.connection_pool.acquire() as conn:
            await conn.copy_records_to_table(
                'trades',
                records=[(t.symbol, t.price, t.quantity) for t in trades],
                columns=['symbol', 'price', 'quantity']
            )
```

День 24-25: Auto-scaling и Load Balancing

python

"""

Задача: Реализовать auto-scaling

Файлы:

- docker/k8s/hpa.yaml
- core/load_balancer.py

"""

Horizontal Pod Autoscaler

apiVersion: autoscaling/v2

kind: HorizontalPodAutoscaler

metadata:

name: market-agent-hpa

spec:

scaleTargetRef:

apiVersion: apps/v1

kind: Deployment

name: market-agent

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

- type: Pods

```

pods:
  metric:
    name: pending_tasks
  target:
    type: AverageValue
    averageValue: "100"

# Load Balancer для распределения задач
class TaskLoadBalancer:
    def __init__(self, workers: List[Worker]):
        self.workers = workers
        self.task_queue = asyncio.Queue()

    async def distribute_tasks(self):
        while True:
            task = await self.task_queue.get()

            # Выбираем worker с наименьшей загрузкой
            worker = min(
                self.workers,
                key=lambda w: w.current_load
            )

            await worker.assign_task(task)

```

Инструкции по использованию для Gemini

1. **Начни с Фазы 1** - это критический фундамент
2. **Следуй принципу "Test First"** - пиши тесты перед кодом
3. **Используй type hints везде** - для лучшей поддержки IDE
4. **Документируй каждый модуль** - с примерами использования

5. **Делай code review** после каждой фазы

Критические метрики успеха

- **Latency:** < 100ms для принятия решения
- **Throughput:** > 1000 символов параллельно
- **Uptime:** > 99.9%
- **Test Coverage:** > 80%
- **API costs:** < \$500/месяц для MVP

Чек-лист готовности к production

- ☐ Все критические пути покрыты тестами
- ☐ Monitoring и alerting настроены
- ☐ Backup и disaster recovery протестированы
- ☐ Security audit пройден
- ☐ Documentation написана
- ☐ Performance benchmarks соответствуют требованиям
- ☐ Compliance требования выполнены

Важно: Этот план рассчитан на поэтапную реализацию. Каждая фаза должна быть полностью завершена и протестирована перед переходом к следующей. Используй Git Flow для управления версиями и feature branches для каждой новой функциональности. # Мастер-промт для Gemini 2.0 Flash: Разработка Market Agent v2.0

Контекст проекта

Ты - Senior Python Developer, работающий над алгоритмическим торговым ботом "Market Agent". Проект уже имеет базовую структуру и философию, но требует модернизации для production-ready состояния.

Текущее состояние проекта:

- Модульная архитектура с разделением на слои
- Синхронный код без асинхронности
- Базовая интеграция с ufinance и технические индикаторы
- Простое логирование в файл
- MVP стадия без тестов

Целевое состояние:

- Асинхронная event-driven архитектура
- Многоуровневое кэширование с Redis
- Интеграция с профессиональными API (Polygon.io, Alpaca)
- LLM интеграция для анализа
- Comprehensive тестирование и мониторинг

Фаза 1: Подготовка инфраструктуры (Неделя 1)

День 1-2: Миграция на асинхронность

```
python
```

"""

Задача: Переписать data_collector.py на asyncio

Файл: input_collection/async_data_collector.py

"""

Требования:

1. Использовать aiohttp вместо requests

2. Реализовать батчинг запросов (до 100 символов параллельно)

3. Добавить retry логику с exponential backoff

4. Интегрировать aiocache для in-memory кэширования

Пример структуры:

```
import asyncio
```

```
import aiohttp
```

```
from typing import List, Dict, Optional
```

```
from tenacity import retry, stop_after_attempt, wait_exponential
```

```
import aiocache
```

```
class AsyncDataCollector:
```

```
    def __init__(self, cache_ttl: int = 300):
```

```
        self.cache = aiocache.Cache()
```

```
        self.session: Optional[aiohttp.ClientSession] = None
```

```
        self.semaphore = asyncio.Semaphore(100) # Лимит параллельных запросов
```

```
    async def __aenter__(self):
```

```
        self.session = aiohttp.ClientSession()
```

```
        return self
```

```
    async def __aexit__(self, exc_type, exc_val, exc_tb):
```

```
        await self.session.close()
```

```
    @retry(
```

```
        stop=stop_after_attempt(3),
```

```

        wait=wait_exponential(multiplier=1, min=4, max=10)
    )
    async def fetch_market_data(self, symbol: str) -> Dict:
        # Проверяем кэш
        cache_key = f"market_data:{symbol}"
        cached = await self.cache.get(cache_key)
        if cached:
            return cached

        # Fetch с ограничением параллельности
        async with self.semaphore:
            # Реализация запроса к API
            pass

    async def collect_batch(self, symbols: List[str]) -> Dict[str, Dict]:
        tasks = [self.fetch_market_data(symbol) for symbol in symbols]
        results = await asyncio.gather(*tasks, return_exceptions=True)

        # Обработка результатов и ошибок
        return {
            symbol: result
            for symbol, result in zip(symbols, results)
            if not isinstance(result, Exception)
        }

```

День 3-4: Настройка Redis и очередей

python

"""

Задача: Создать инфраструктуру для кэширования и очередей

Файлы:

- core/cache_manager.py
- core/queue_manager.py
- docker-compose.yml для локальной разработки

"""

Requirements:

1. Redis для кэширования с TTL

2. Redis Pub/Sub для real-time событий

3. RabbitMQ для надёжной доставки критических сообщений

4. Абстракция для переключения между backend'ами

Создать docker-compose.yml:

version: '3.8'

services:

redis:

image: redis:7-alpine

ports:

- "6379:6379"

volumes:

- redis_data:/data

command: redis-server --appendonly yes

rabbitmq:

image: rabbitmq:3-management

ports:

- "5672:5672"

- "15672:15672"

environment:

RABBITMQ_DEFAULT_USER: trading_bot
RABBITMQ_DEFAULT_PASS: secure_password

День 5: Event Bus и Circuit Breakers

python

"""

Задача: Реализовать event-driven архитектуру

Файлы:

- core/event_bus.py
- infrastructure/circuit_breaker.py (улучшить существующий)

"""

Event Bus с поддержкой приоритетов:

```
from asyncio import PriorityQueue
from dataclasses import dataclass, field
from typing import Any
import time

@dataclass(order=True)
class Event:
    priority: int
    timestamp: float = field(default_factory=time.time)
    type: str = field(compare=False)
    data: Any = field(compare=False)

class EventBus:
    def __init__(self):
        self.queue = PriorityQueue()
        self.handlers = defaultdict(list)

    async def emit(self, event_type: str, data: Any, priority: int = 5):
        event = Event(priority=priority, type=event_type, data=data)
        await self.queue.put(event)

    async def process_events(self):
        while True:
            event = await self.queue.get()
```

```
for handler in self.handlers[event.type]:  
    asyncio.create_task(handler(event.data))
```

Фаза 2: Интеграция внешних API (Неделя 2)

День 6-7: Polygon.io интеграция

```
python
```


"""

Задача: Создать клиент для Polygon.io с WebSocket поддержкой

Файл: input_collection/market_data/polygon_client.py

"""

Требования:

1. REST API для исторических данных

2. WebSocket для real-time потока

3. Автоматическое переподключение

4. Нормализация данных в единый формат

class PolygonClient:

def __init__(self, api_key: str, event_bus: EventBus):

self.api_key = api_key

self.event_bus = event_bus

self.ws_client = None

async def stream_quotes(self, symbols: List[str]):

WebSocket подключение с auto-reconnect

async for quote in self._ws_stream(symbols):

await self.event_bus.emit(

"market.quote",

{"symbol": quote.symbol, "price": quote.price},

priority=1 *# Высокий приоритет для рыночных данных*

)

День 8-9: Social signals интеграция

python

"""

Задача: Реализовать Scout/Hunter паттерн для социальных данных

Файлы:

- input_collection/social_data/bullaware_client.py

- scripts/social_data_scout.py

"""

Scout процесс (запускается раз в час):

`async def scout_social_data():`

`client = BullAwareClient(rate_limit=10) # 10 req/min`

Получаем список топ-трейдеров

`top_traders = await client.get_top_traders(limit=100)`

Медленно обходим их портфели

`for trader in top_traders:`

`portfolio = await client.get_portfolio(trader.id)`

Сохраняем в Redis с TTL 2 часа

```
    await redis.setex(
        f"portfolio:{trader.id}",
        7200,
        json.dumps(portfolio)
    )
```

Уважаем rate limits

`await asyncio.sleep(6) # 10 req/min = 1 req/6 sec`

Hunter процесс (основное приложение):

`async def get_social_sentiment(symbol: str) -> float:`

Молниеносное чтение из кэша

`pipe = redis.pipeline()`

`for i in range(100): # Top 100 traders`

```
pipe.get(f"portfolio:{i}")
```

```
portfolios = await pipe.execute()
```

```
# Быстрый подсчёт sentiment
```

```
return calculate_social_sentiment(symbol, portfolios)
```

День 10: LLM интеграция

```
python
```

"""

Задача: Создать умный LLM оркестратор с кэшированием

Файл: processing_analysis/llm_orchestrator.py

"""

Требования:

1. Поддержка Claude 3.5 и GPT-4

2. Semantic кэширование похожих запросов

3. Fallback между моделями

4. Структурированный вывод

class LLMOrchestrator:

def __init__(self, strategy_type: str):

self.strategy = strategy_type

self.claude_client = anthropic.AsyncAnthropic()

self.openai_client = openai.AsyncOpenAI()

self.semantic_cache = SemanticCache() *# Используем embeddings*

async def analyze_market_context(

self,

symbol: str,

market_data: dict

) -> StructuredAnalysis:

Проверяем semantic cache

cached = **await** self.semantic_cache.get_similar(

f"analyze {symbol} {self.strategy}",

threshold=0.95

)

if cached:

return cached

Строим промпт с учётом стратегии

prompt = self._build_analysis_prompt(symbol, market_data)

```
try:
    # Пробуем Claude (лучше для анализа)
    result = await self._query_claude(prompt)
except Exception as e:
    # Fallback на GPT-4
    result = await self._query_gpt4(prompt)

# Кэшируем результат
await self.semantic_cache.store(prompt, result)

return result
```

Фаза 3: Продвинутая обработка и анализ (Неделя 3)

День 11-12: Ensemble Scoring Engine

python

"""

Задача: Создать адаптивную систему взвешивания сигналов

Файл: processing_analysis/ensemble_scorer.py

"""

class EnsembleScorer:

def __init__(self, strategy_config: dict):

 self.weights = strategy_config['signal_weights']

 self.confidence_threshold = strategy_config['confidence_threshold']

async def calculate_composite_score(

 self,

 symbol: str,

 signals: Dict[str, Signal]

) -> TradingDecision:

Динамическая корректировка весов на основе performance

 adjusted_weights = **await** self._adjust_weights_by_performance(

 symbol,

 signals

)

Расчёт ensemble score

 scores = {}

 confidences = {}

for signal_type, signal **in** signals.items():

 weight = adjusted_weights.get(signal_type, 0)

 scores[signal_type] = signal.value * weight

 confidences[signal_type] = signal.confidence

Финальное решение с учётом confidence

 composite_score = **sum**(scores.values())

 composite_confidence = **np.average**(

```
list(confidences.values()),  
weights=list(adjusted_weights.values())  
)  
  
return TradingDecision(  
    symbol=symbol,  
    action=self._score_to_action(composite_score),  
    confidence=composite_confidence,  
    reasoning=self._generate_reasoning(signals, scores)  
)
```

День 13-14: Backtesting Engine

python

"""

Задача: Создать высокопроизводительный backtesting engine

Файл: monitoring_feedback/backtest_engine.py

"""

Использовать векторизованные операции для скорости

class VectorizedBacktester:

def __init__(self, strategy: TradingStrategy):

 self.strategy = strategy

async def run_backtest(

 self,

 symbols: List[str],

 start_date: datetime,

 end_date: datetime

) -> BacktestResults:

Параллельная загрузка данных

 market_data = **await** self._load_market_data_parallel(

 symbols,

 start_date,

 end_date

)

Векторизованный расчёт сигналов

 signals = self._vectorized_signal_generation(market_data)

Симуляция исполнения с учётом slippage и комиссий

 trades = self._simulate_execution(

 signals,

 slippage_model=self.realistic_slippage,

 commission_model=self.ibkr_commission

)

Расчёт метрик

```
return BacktestResults(  
    total_return=self._calculate_return(trades),  
    sharpe_ratio=self._calculate_sharpe(trades),  
    max_drawdown=self._calculate_max_drawdown(trades),  
    win_rate=self._calculate_win_rate(trades),  
    profit_factor=self._calculate_profit_factor(trades)  
)
```