Мастер-промпт 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:

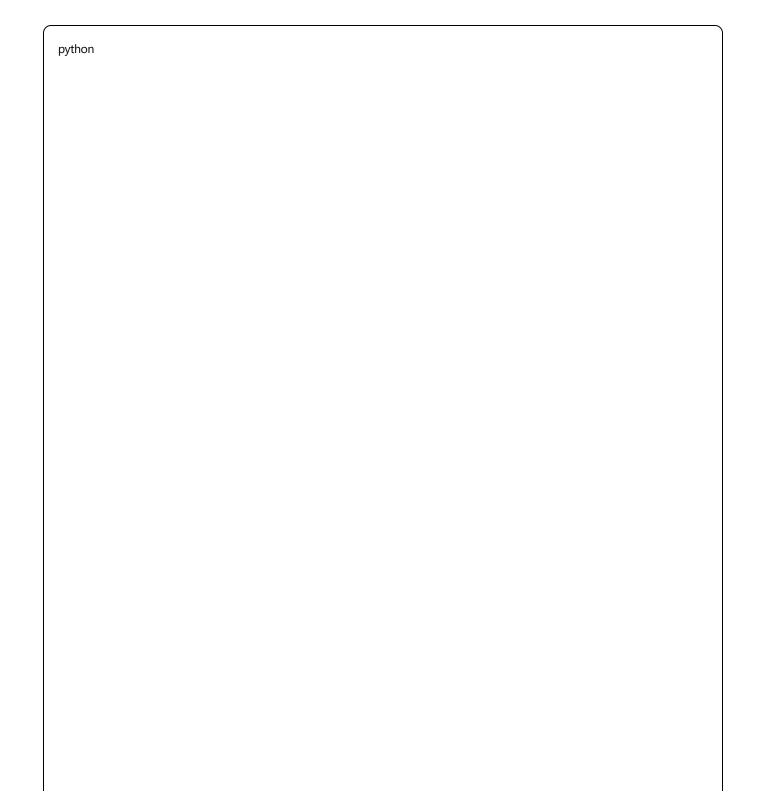
- Модульная архитектура с разделением на слои
- Синхронный код без асинхронности
- Базовая интеграция с yfinance и технические индикаторы
- Простое логирование в файл
- MVP стадия без тестов

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



```
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-тетогу 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 АРІ
    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 АРІ (Неделя 2)

День 6-7: Polygon.io интеграция

| python | | | |
|--------|--|--|--|
| 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 c 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



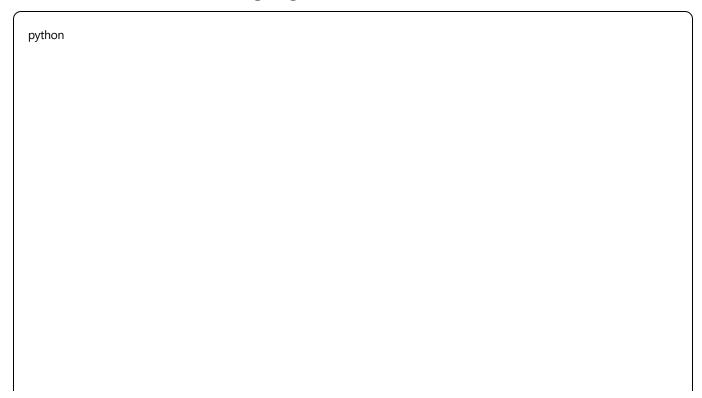
```
Задача: Создать умный LLM оркестратор с кэшированием
Файл: processing_analysis/llm_orchestrator.py
# Требования:
# 1. По∂∂ержка Claude 3.5 и GPT-4
# 2. Semantic кэширование похожих запросов
# 3. Fallback между моделями
# 4. Структурированный вывод
class LLMOrchestrator:
  def __init__(self, strategy_type: str):
    self.strategy = strategy_type
    self.claude_client = anthropic.AsyncAnthropic()
    self.openai_client = openai.AsyncOpenAl()
    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

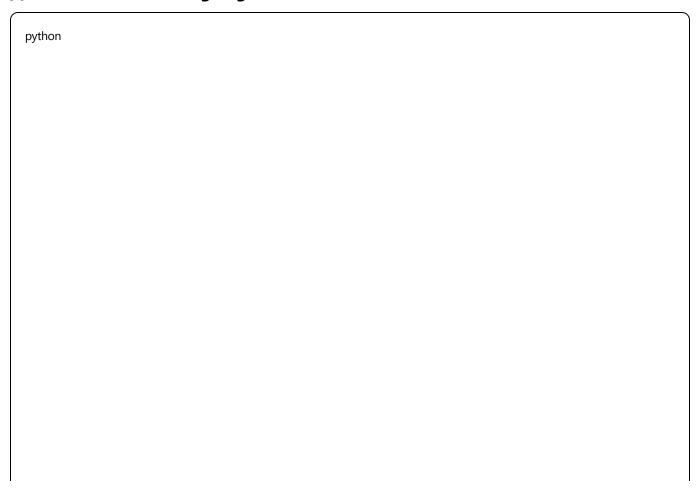


```
Задача: Создать адаптивную систему взвешивания сигналов
Файл: processing_analysis/ensemble_scorer.py
class EnsembleScorer:
  def __init__(self, strategy_config: dict):
    self.weights = strategy_config['signal_weights']
    self.confidence_threshold = strategy_config['confidence_threshold']
  async 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
     # Pacyëm 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



```
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