НАш сайт systemetech.ru www-root /var/www/www-root/data/www/systemetech.ru 79.133.183.86 crypto\_top\_bd\_mysql PHP FastCGI (Nginx + PHP-FPM) 8.1.2 (native)

вся программа и бот во вложении. сейчас вроде все работает хорошо. Так вот мне нужно допилить их до идеала. Сделать их обучаемыми. Также мне нужно чтобы анатомически открывались сделки, тейк Профит тейк лосс это вроде все есть. Мне важно, чтобы у меня не копилось кучи логов, условно говоря если программа будет работать целый день. Логов может накопиться очень много. По этому, нужно чистить не нужные. Логировать и собирать анализ только самого важно, что реально потом поможет в открытии сделок смене стратегии и т.д. нужно все это до мелочей продумать, представь что ты самый лучший трейдер и ты постоянно анализируешь и делаешь выводы по сделкам. Так вот это все мне нужно прописать грамотно. Чтобы стратегии в дальнейшем корректировались и автоматически выбирались основываясь на опыте. Что еще важно , у нас сейчас в дашборд не выводится баланс. В том числе, я бы хотел видеть реальные графики на отдельной странице в дашборде, если это реально. К примеру, я выбираю валюты на которых торговать, и все они отображаются через trading view у меня на сайте. Или другими способами. В общем мне нужен полный анализ, лучшие стратегии, обучаемые стратегии, все что только возможно, должна быть самая продвинутая современная модель работы. также в дальнейшем мы будем подключать другие стратегии, анализаторы новостей, твитов и других вещей. Все это должно быть максимально расширяемо и удобно! Напиши сначала документацию полную, что и за чем будет идти, что нужно исправить в каком коде и т.д. далее напиши правки и если нужно, новые коды, но только чтобы везде зависимости были настроены!

Edit

Architected comprehensive crypto trading bot enhancement strategy.

24s

Пользователь предоставил большую кодовую базу криптотрейдинг бота и просит:

1. Допилить до идеала, сделать обучаемыми
2. Автоматическое открытие сделок (уже есть)
3. Оптимизация логирования - чистить ненужные логи, сохранять только важное для анализа
4. Продумать систему анализа как профессиональный трейдер
5. Автокорректировка и выбор стратегий на основе опыта
6. Исправить отображение баланса в дашборде
7. Добавить реальные графики (TradingView или другие)
8. Создать самую продвинутую современную модель работы
9. Подготовить к расширению - новости, твиты и т.д.

Нужно написать:

1. Полную документацию с планом улучшений
2. Правки кода
3. Новые модули если нужно

Проанализирую текущую архитектуру:

1. **Основные компоненты:**
   * main.py - точка входа
   * bot/manager.py - основной менеджер бота
   * strategies/ - стратегии торговли
   * analysis/ - модули анализа
   * web/ - веб-интерфейс и API
   * exchange/ - работа с биржей
   * core/ - ядро системы
2. **Текущие проблемы:**
   * Логирование не оптимизировано для долгой работы
   * Требуется существенная модернизация интеллектуальных компонентов
   * Визуализация и аналитика находятся на примитивном уровне

Необходимо кардинально переработать систему машинного обучения, внедрив продвинутые алгоритмы адаптивного выбора торговых стратегий. Критически важно разработать интеллектуальный механизм, который будет анализировать рыночные данные в реальном времени и автоматически корректировать торговую логику.

**📚 ПОЛНАЯ ДОКУМЕНТАЦИЯ ПО УЛУЧШЕНИЮ CRYPTO TRADING BOT**

**🎯 Цели проекта**

Превратить существующий бот в самообучающуюся систему профессионального уровня с:

* Машинным обучением для выбора стратегий
* Продвинутой аналитикой как у профессиональных трейдеров
* Оптимизированным логированием для долгой работы
* Визуализацией данных через графики
* Расширяемой архитектурой для будущих модулей

**📋 План улучшений**

**1. Система логирования и хранения данных 🗂️**

**Проблема:** Накопление большого количества логов при долгой работе **Решение:**

* Создать src/core/log\_manager.py - умная ротация логов
* Создать src/analysis/trade\_journal.py - журнал сделок для ML
* Обновить src/core/clean\_logging.py - добавить уровни важности

**Что сохранять:**

* ✅ Все сделки с полным контекстом (цена входа/выхода, индикаторы, причины)
* ✅ Изменения стратегий и их эффективность
* ✅ Критические события (ошибки, большие убытки/прибыли)
* ✅ Рыночные аномалии
* ❌ Рутинные проверки состояния
* ❌ Повторяющиеся сообщения

**2. Машинное обучение для стратегий 🤖**

**Текущее состояние:** Базовый auto\_strategy\_selector.py **Улучшения:**

* Создать src/ml/strategy\_optimizer.py - оптимизация параметров стратегий
* Создать src/ml/market\_predictor.py - предсказание движений рынка
* Создать src/ml/risk\_calculator.py - ML оценка рисков
* Обновить src/strategies/auto\_strategy\_selector.py - добавить нейросети

**Модели ML:**

* LSTM для предсказания цен
* Random Forest для выбора стратегий
* Reinforcement Learning для оптимизации параметров

**3. Профессиональная аналитика 📊**

**Новые модули:**

* src/analysis/market\_profile.py - профиль рынка и объемный анализ
* src/analysis/sentiment\_analyzer.py - анализ настроений (подготовка для новостей)
* src/analysis/correlation\_matrix.py - корреляции между парами
* src/analysis/performance\_attribution.py - атрибуция производительности

**Улучшение существующих:**

* advanced\_analytics.py - добавить больше метрик
* market\_analyzer.py - добавить паттерны свечей, уровни Фибоначчи

**4. Веб-интерфейс и визуализация 🎨**

**Исправления:**

* Баланс не отображается - исправить в api\_routes.py
* Добавить эндпоинт для графиков

**Новые функции:**

* Создать src/web/charts.py - интеграция с TradingView или lightweight-charts
* Создать отдельную страницу для графиков в дашборде
* Добавить реалтайм обновление через WebSocket

**5. Система обучения на опыте 🎓**

**Новые компоненты:**

* src/ml/experience\_replay.py - хранение и воспроизведение опыта
* src/ml/strategy\_evolution.py - эволюция стратегий
* База данных для хранения опыта

**6. Расширяемая архитектура 🔧**

**Подготовка для будущих модулей:**

* Создать src/plugins/ - директория для плагинов
* Создать src/plugins/base.py - базовый класс для плагинов
* Система событий для интеграции

**🛠️ Порядок реализации**

**Этап 1: Оптимизация логирования (1-2 дня)**

1. Создать log\_manager.py
2. Создать trade\_journal.py
3. Обновить clean\_logging.py
4. Интегрировать в bot/manager.py

**Этап 2: Исправление дашборда (1 день)**

1. Исправить отображение баланса
2. Добавить WebSocket обновления баланса
3. Создать страницу графиков

**Этап 3: Улучшение ML (3-4 дня)**

1. Создать ML модули
2. Обучить модели на исторических данных
3. Интегрировать в выбор стратегий

**Этап 4: Профессиональная аналитика (2-3 дня)**

1. Создать новые модули анализа
2. Улучшить существующие
3. Интегрировать в принятие решений

**Этап 5: Система обучения (2-3 дня)**

1. Создать систему опыта
2. Реализовать эволюцию стратегий
3. Настроить автообучение

**Этап 6: Расширяемость (1-2 дня)**

1. Создать систему плагинов

## Подготовить для новостей/твитов Конкретные правки кода

### 1. ****Создание системы управления логами****

<artifacts> <artifact identifier="log-manager" type="application/vnd.ant.code" language="python" title="src/core/log\_manager.py - Умное управление логами"> """ Менеджер логов с ротацией и приоритетами Сохраняет только важную информацию для обучения """ import os import json import logging import asyncio from datetime import datetime, timedelta from pathlib import Path from typing import Dict, List, Optional, Any from collections import deque import aiofiles import pandas as pd from sqlalchemy import create\_engine, text

from .config import config from .clean\_logging import get\_clean\_logger

logger = get\_clean\_logger(**name**)

class LogPriority: """Приоритеты логов""" CRITICAL = 100 # Всегда сохранять (критические ошибки, большие убытки) HIGH = 80 # Важные события (сделки, изменения стратегий) MEDIUM = 50 # Полезная информация (сигналы, анализ) LOW = 20 # Отладочная информация NOISE = 0 # Шум (повторяющиеся сообщения)

class LogManager: """ Умный менеджер логов

Особенности:

- Автоматическая ротация логов

- Сохранение только важной информации

- Сжатие старых логов

- Экспорт в БД для анализа

"""

def \_\_init\_\_(self):

self.log\_dir = Path("logs")

self.log\_dir.mkdir(exist\_ok=True)

# Директории по приоритетам

self.priority\_dirs = {

"critical": self.log\_dir / "critical",

"trades": self.log\_dir / "trades",

"analysis": self.log\_dir / "analysis",

"debug": self.log\_dir / "debug"

}

for dir\_path in self.priority\_dirs.values():

dir\_path.mkdir(exist\_ok=True)

# Буферы для разных типов логов

self.buffers = {

"critical": deque(maxlen=1000),

"trades": deque(maxlen=5000),

"analysis": deque(maxlen=2000),

"debug": deque(maxlen=500)

}

# Счетчики для предотвращения спама

self.message\_counts = {}

self.last\_cleanup = datetime.utcnow()

# Запускаем фоновые задачи

self.\_start\_background\_tasks()

def \_start\_background\_tasks(self):

"""Запуск фоновых задач"""

asyncio.create\_task(self.\_rotation\_worker())

asyncio.create\_task(self.\_cleanup\_worker())

asyncio.create\_task(self.\_export\_worker())

async def log(self, message: str, level: str, priority: int,

metadata: Optional[Dict[str, Any]] = None):

"""

Логирование с приоритетом

Args:

message: Сообщение

level: Уровень (INFO, WARNING, ERROR и т.д.)

priority: Приоритет (LogPriority)

metadata: Дополнительные данные

"""

# Проверяем на спам

if self.\_is\_spam(message):

return

# Создаем запись

log\_entry = {

"timestamp": datetime.utcnow().isoformat(),

"level": level,

"priority": priority,

"message": message,

"metadata": metadata or {}

}

# Определяем категорию

category = self.\_categorize\_log(message, priority, metadata)

# Добавляем в буфер

if category in self.buffers:

self.buffers[category].append(log\_entry)

# Для критических - сразу сохраняем

if priority >= LogPriority.CRITICAL:

await self.\_save\_critical\_log(log\_entry)

def \_categorize\_log(self, message: str, priority: int,

metadata: Optional[Dict]) -> str:

"""Категоризация лога"""

if priority >= LogPriority.CRITICAL:

return "critical"

# Проверяем метаданные

if metadata:

if "trade\_id" in metadata or "position" in metadata:

return "trades"

if "analysis" in metadata or "strategy" in metadata:

return "analysis"

# Проверяем ключевые слова

trade\_keywords = ["trade", "position", "order", "buy", "sell", "profit", "loss"]

if any(keyword in message.lower() for keyword in trade\_keywords):

return "trades"

analysis\_keywords = ["analysis", "signal", "strategy", "indicator"]

if any(keyword in message.lower() for keyword in analysis\_keywords):

return "analysis"

return "debug"

def \_is\_spam(self, message: str) -> bool:

"""Проверка на спам"""

# Очищаем сообщение от временных меток

clean\_msg = self.\_clean\_message(message)

# Проверяем счетчик

current\_time = datetime.utcnow()

if clean\_msg in self.message\_counts:

last\_time, count = self.message\_counts[clean\_msg]

# Если прошло меньше минуты и уже больше 5 сообщений

if (current\_time - last\_time).seconds < 60 and count > 5:

return True

# Обновляем счетчик

if (current\_time - last\_time).seconds < 60:

self.message\_counts[clean\_msg] = (last\_time, count + 1)

else:

self.message\_counts[clean\_msg] = (current\_time, 1)

else:

self.message\_counts[clean\_msg] = (current\_time, 1)

return False

def \_clean\_message(self, message: str) -> str:

"""Очистка сообщения от изменяющихся данных"""

import re

# Удаляем временные метки

message = re.sub(r'\d{4}-\d{2}-\d{2}[T\s]\d{2}:\d{2}:\d{2}', '', message)

message = re.sub(r'\d{2}:\d{2}:\d{2}', '', message)

# Удаляем числа (цены, проценты и т.д.)

message = re.sub(r'\d+\.?\d\*', 'NUM', message)

# Удаляем ID

message = re.sub(r'#\d+', '#ID', message)

return message.strip()

async def \_save\_critical\_log(self, log\_entry: Dict):

"""Немедленное сохранение критических логов"""

try:

filename = self.priority\_dirs["critical"] / f"critical\_{datetime.utcnow().strftime('%Y%m%d')}.jsonl"

async with aiofiles.open(filename, 'a') as f:

await f.write(json.dumps(log\_entry, ensure\_ascii=False) + '\n')

# Также отправляем уведомление

if "metadata" in log\_entry and "notify" in log\_entry["metadata"]:

await self.\_send\_notification(log\_entry)

except Exception as e:

logger.error(f"Ошибка сохранения критического лога: {e}")

async def \_rotation\_worker(self):

"""Фоновая задача ротации логов"""

while True:

try:

await asyncio.sleep(300) # Каждые 5 минут

for category, buffer in self.buffers.items():

if len(buffer) > 100: # Если накопилось достаточно

await self.\_rotate\_buffer(category, buffer)

except Exception as e:

logger.error(f"Ошибка в rotation worker: {e}")

async def \_rotate\_buffer(self, category: str, buffer: deque):

"""Ротация буфера в файл"""

if not buffer:

return

try:

filename = self.priority\_dirs[category] / f"{category}\_{datetime.utcnow().strftime('%Y%m%d\_%H')}.jsonl"

# Копируем буфер

entries = list(buffer)

buffer.clear()

# Сохраняем

async with aiofiles.open(filename, 'a') as f:

for entry in entries:

await f.write(json.dumps(entry, ensure\_ascii=False) + '\n')

except Exception as e:

logger.error(f"Ошибка ротации буфера {category}: {e}")

async def \_cleanup\_worker(self):

"""Очистка старых логов"""

while True:

try:

await asyncio.sleep(3600) # Каждый час

# Очищаем счетчики спама

current\_time = datetime.utcnow()

old\_messages = [

msg for msg, (time, \_) in self.message\_counts.items()

if (current\_time - time).seconds > 3600

]

for msg in old\_messages:

del self.message\_counts[msg]

# Удаляем старые debug логи (старше 1 дня)

await self.\_cleanup\_old\_logs("debug", days=1)

# Архивируем старые analysis логи (старше 7 дней)

await self.\_archive\_old\_logs("analysis", days=7)

# Trades и critical хранятся дольше

await self.\_archive\_old\_logs("trades", days=30)

await self.\_archive\_old\_logs("critical", days=90)

except Exception as e:

logger.error(f"Ошибка в cleanup worker: {e}")

async def \_cleanup\_old\_logs(self, category: str, days: int):

"""Удаление старых логов"""

cutoff\_date = datetime.utcnow() - timedelta(days=days)

for file\_path in self.priority\_dirs[category].glob("\*.jsonl"):

if file\_path.stat().st\_mtime < cutoff\_date.timestamp():

file\_path.unlink()

logger.debug(f"Удален старый лог: {file\_path}")

async def \_archive\_old\_logs(self, category: str, days: int):

"""Архивация старых логов"""

import gzip

cutoff\_date = datetime.utcnow() - timedelta(days=days)

for file\_path in self.priority\_dirs[category].glob("\*.jsonl"):

if file\_path.stat().st\_mtime < cutoff\_date.timestamp():

# Сжимаем файл

gz\_path = file\_path.with\_suffix('.jsonl.gz')

with open(file\_path, 'rb') as f\_in:

with gzip.open(gz\_path, 'wb') as f\_out:

f\_out.write(f\_in.read())

file\_path.unlink()

logger.debug(f"Заархивирован лог: {file\_path} -> {gz\_path}")

async def \_export\_worker(self):

"""Экспорт важных логов в БД"""

while True:

try:

await asyncio.sleep(600) # Каждые 10 минут

# Экспортируем trades для анализа

await self.\_export\_trades\_to\_db()

# Экспортируем analysis для ML

await self.\_export\_analysis\_to\_db()

except Exception as e:

logger.error(f"Ошибка в export worker: {e}")

async def \_export\_trades\_to\_db(self):

"""Экспорт логов сделок в БД"""

# Здесь будет логика экспорта в таблицу trade\_logs

pass

async def \_export\_analysis\_to\_db(self):

"""Экспорт аналитических логов в БД"""

# Здесь будет логика экспорта в таблицу analysis\_logs

pass

async def get\_trade\_logs(self, trade\_id: Optional[int] = None,

days: int = 7) -> List[Dict]:

"""Получение логов сделок для анализа"""

logs = []

cutoff\_date = datetime.utcnow() - timedelta(days=days)

for file\_path in sorted(self.priority\_dirs["trades"].glob("\*.jsonl")):

if file\_path.stat().st\_mtime < cutoff\_date.timestamp():

continue

async with aiofiles.open(file\_path, 'r') as f:

async for line in f:

try:

log\_entry = json.loads(line)

# Фильтруем по trade\_id если указан

if trade\_id and log\_entry.get("metadata", {}).get("trade\_id") != trade\_id:

continue

logs.append(log\_entry)

except:

continue

return logs

async def get\_analysis\_logs(self, strategy: Optional[str] = None,

hours: int = 24) -> pd.DataFrame:

"""Получение логов анализа для ML"""

logs = []

cutoff\_time = datetime.utcnow() - timedelta(hours=hours)

# Читаем из буфера

for entry in self.buffers["analysis"]:

if datetime.fromisoformat(entry["timestamp"]) > cutoff\_time:

if strategy and entry.get("metadata", {}).get("strategy") != strategy:

continue

logs.append(entry)

# Читаем из файлов

for file\_path in sorted(self.priority\_dirs["analysis"].glob("\*.jsonl")):

if file\_path.stat().st\_mtime < cutoff\_time.timestamp():

continue

async with aiofiles.open(file\_path, 'r') as f:

async for line in f:

try:

log\_entry = json.loads(line)

if datetime.fromisoformat(log\_entry["timestamp"]) > cutoff\_time:

if strategy and log\_entry.get("metadata", {}).get("strategy") != strategy:

continue

logs.append(log\_entry)

except:

continue

# Конвертируем в DataFrame для удобного анализа

if logs:

df = pd.DataFrame(logs)

df['timestamp'] = pd.to\_datetime(df['timestamp'])

df.set\_index('timestamp', inplace=True)

return df

return pd.DataFrame()

def get\_summary\_stats(self) -> Dict[str, Any]:

"""Получение статистики по логам"""

stats = {

"buffer\_sizes": {k: len(v) for k, v in self.buffers.items()},

"spam\_filters": len(self.message\_counts),

"disk\_usage": {}

}

# Подсчет размера на диске

for category, dir\_path in self.priority\_dirs.items():

total\_size = sum(f.stat().st\_size for f in dir\_path.glob("\*") if f.is\_file())

stats["disk\_usage"][category] = f"{total\_size / 1024 / 1024:.2f} MB"

return stats

# Глобальный экземпляр

log\_manager = LogManager()

# Вспомогательные функции для легкого использования

async def log\_trade(message: str, trade\_id: int, metadata: Dict[str, Any] = None): """Логирование события сделки""" if metadata is None: metadata = {} metadata["trade\_id"] = trade\_id

await log\_manager.log(

message=message,

level="INFO",

priority=LogPriority.HIGH,

metadata=metadata

)

async def log\_analysis(message: str, strategy: str, metadata: Dict[str, Any] = None): """Логирование аналитического события""" if metadata is None: metadata = {} metadata["strategy"] = strategy metadata["analysis"] = True

await log\_manager.log(

message=message,

level="INFO",

priority=LogPriority.MEDIUM,

metadata=metadata

)

async def log\_critical(message: str, metadata: Dict[str, Any] = None, notify: bool = True): """Логирование критического события""" if metadata is None: metadata = {} metadata["notify"] = notify

await log\_manager.log(

message=message,

level="ERROR",

priority=LogPriority.CRITICAL,

metadata=metadata

)

async def log\_debug(message: str, metadata: Dict[str, Any] = None): """Логирование отладочной информации""" await log\_manager.log( message=message, level="DEBUG", priority=LogPriority.LOW, metadata=metadata ) </artifact> </artifacts>

### 2. ****Создание журнала сделок для ML****

<artifacts> <artifact identifier="trade-journal" type="application/vnd.ant.code" language="python" title="src/analysis/trade\_journal.py - Журнал сделок для обучения"> """ Журнал сделок для машинного обучения Сохраняет полный контекст каждой сделки для последующего анализа """ import pandas as pd import numpy as np from datetime import datetime, timedelta from typing import Dict, List, Optional, Any, Tuple import json from pathlib import Path import aiofiles from sqlalchemy import text from sqlalchemy.orm import Session

from ..core.database import SessionLocal from ..core.models import Trade, Signal from ..core.clean\_logging import get\_clean\_logger

logger = get\_clean\_logger(**name**)

class TradeJournal: """ Профессиональный журнал трейдера

Сохраняет:

- Полный контекст рынка при входе/выходе

- Причины принятия решений

- Эмоциональное состояние (имитация)

- Результаты и уроки

"""

def \_\_init\_\_(self):

self.journal\_dir = Path("data/journal")

self.journal\_dir.mkdir(parents=True, exist\_ok=True)

# Текущий активный журнал

self.current\_journal = self.journal\_dir / f"journal\_{datetime.now().strftime('%Y%m')}.jsonl"

# Кэш для быстрого доступа

self.recent\_entries = []

async def record\_trade\_entry(self, trade: Trade, signal: Signal,

market\_context: Dict[str, Any]) -> int:

"""

Запись входа в сделку

Args:

trade: Объект сделки

signal: Сигнал на вход

market\_context: Полный контекст рынка

Returns:

journal\_id: ID записи в журнале

"""

entry = {

"journal\_id": self.\_generate\_id(),

"timestamp": datetime.utcnow().isoformat(),

"type": "ENTRY",

"trade\_id": trade.id,

# Основная информация

"symbol": trade.symbol,

"side": trade.side.value if hasattr(trade.side, 'value') else str(trade.side),

"entry\_price": float(trade.entry\_price),

"quantity": float(trade.quantity),

"stop\_loss": float(trade.stop\_loss) if trade.stop\_loss else None,

"take\_profit": float(trade.take\_profit) if trade.take\_profit else None,

# Сигнал и стратегия

"signal": {

"action": signal.action,

"confidence": float(signal.confidence),

"strategy": signal.strategy,

"reason": signal.reason

},

# Рыночный контекст

"market\_context": self.\_extract\_market\_context(market\_context),

# Технические индикаторы на момент входа

"indicators": self.\_extract\_indicators(market\_context),

# Анализ риска

"risk\_analysis": self.\_analyze\_risk(trade, market\_context),

# "Эмоциональное состояние" (имитация психологии трейдера)

"trader\_state": self.\_assess\_trader\_state(market\_context),

# Ожидания

"expectations": {

"target\_profit\_percent": 2.0, # Можно рассчитать динамически

"expected\_duration\_hours": 4, # На основе исторических данных

"confidence\_level": float(signal.confidence)

}

}

# Сохраняем

await self.\_save\_entry(entry)

# Добавляем в кэш

self.recent\_entries.append(entry)

if len(self.recent\_entries) > 100:

self.recent\_entries.pop(0)

logger.info(f"📝 Записан вход в сделку {trade.symbol} в журнал")

return entry["journal\_id"]

async def record\_trade\_exit(self, trade: Trade, exit\_reason: str,

market\_context: Dict[str, Any]) -> None:

"""

Запись выхода из сделки

Args:

trade: Объект сделки

exit\_reason: Причина выхода

market\_context: Контекст рынка при выходе

"""

# Находим запись входа

entry\_record = await self.\_find\_entry\_record(trade.id)

if not entry\_record:

logger.warning(f"Не найдена запись входа для сделки {trade.id}")

return

exit\_entry = {

"journal\_id": self.\_generate\_id(),

"parent\_id": entry\_record["journal\_id"],

"timestamp": datetime.utcnow().isoformat(),

"type": "EXIT",

"trade\_id": trade.id,

# Результаты

"exit\_price": float(trade.exit\_price) if trade.exit\_price else 0,

"profit": float(trade.profit) if trade.profit else 0,

"profit\_percent": float(trade.profit\_percent) if trade.profit\_percent else 0,

"duration": str(trade.closed\_at - trade.created\_at) if trade.closed\_at else None,

"exit\_reason": exit\_reason,

# Рыночный контекст при выходе

"market\_context": self.\_extract\_market\_context(market\_context),

# Анализ исполнения

"execution\_analysis": self.\_analyze\_execution(trade, entry\_record),

# Уроки

"lessons": self.\_extract\_lessons(trade, entry\_record, market\_context),

# Оценка качества сделки

"trade\_quality": self.\_assess\_trade\_quality(trade, entry\_record)

}

await self.\_save\_entry(exit\_entry)

# Обновляем статистику

await self.\_update\_statistics(trade, entry\_record, exit\_entry)

logger.info(f"📝 Записан выход из сделки {trade.symbol} в журнал")

async def record\_market\_event(self, event\_type: str, symbol: str,

description: str, impact: str = "medium") -> None:

"""

Запись важного рыночного события

Args:

event\_type: Тип события (news, anomaly, pattern и т.д.)

symbol: Затронутый символ

description: Описание события

impact: Влияние (low, medium, high)

"""

event\_entry = {

"journal\_id": self.\_generate\_id(),

"timestamp": datetime.utcnow().isoformat(),

"type": "MARKET\_EVENT",

"event\_type": event\_type,

"symbol": symbol,

"description": description,

"impact": impact

}

await self.\_save\_entry(event\_entry)

def \_extract\_market\_context(self, context: Dict[str, Any]) -> Dict[str, Any]:

"""Извлечение важного рыночного контекста"""

return {

"trend": context.get("trend", {}),

"volatility": context.get("volatility", {}),

"volume\_analysis": context.get("volume\_analysis", {}),

"support\_resistance": {

"support": context.get("support"),

"resistance": context.get("resistance")

},

"market\_phase": self.\_determine\_market\_phase(context),

"correlation\_with\_btc": self.\_calculate\_btc\_correlation(context)

}

def \_extract\_indicators(self, context: Dict[str, Any]) -> Dict[str, float]:

"""Извлечение значений индикаторов"""

df = context.get("df")

if df is None or df.empty:

return {}

try:

last\_row = df.iloc[-1]

indicators = {

"rsi": float(last\_row.get("rsi", 50)),

"macd": float(last\_row.get("macd", 0)),

"bb\_position": float(last\_row.get("bb\_percent", 0.5)),

"volume\_ratio": float(context.get("volume\_analysis", {}).get("ratio", 1.0)),

"atr": float(context.get("volatility", {}).get("atr", 0))

}

# Добавляем дополнительные индикаторы если есть

for col in ["stoch\_k", "adx", "ema\_9", "ema\_21", "ema\_50"]:

if col in last\_row:

indicators[col] = float(last\_row[col])

return indicators

except Exception as e:

logger.error(f"Ошибка извлечения индикаторов: {e}")

return {}

def \_analyze\_risk(self, trade: Trade, context: Dict[str, Any]) -> Dict[str, Any]:

"""Анализ рисков сделки"""

entry\_price = float(trade.entry\_price)

stop\_loss = float(trade.stop\_loss) if trade.stop\_loss else entry\_price \* 0.98

take\_profit = float(trade.take\_profit) if trade.take\_profit else entry\_price \* 1.02

risk = abs(entry\_price - stop\_loss)

reward = abs(take\_profit - entry\_price)

return {

"risk\_amount": risk \* float(trade.quantity),

"risk\_percent": (risk / entry\_price) \* 100,

"reward\_amount": reward \* float(trade.quantity),

"reward\_percent": (reward / entry\_price) \* 100,

"risk\_reward\_ratio": reward / risk if risk > 0 else 0,

"position\_size\_percent": self.\_calculate\_position\_size\_percent(trade),

"volatility\_adjusted\_risk": risk / context.get("volatility", {}).get("atr", risk)

}

def \_assess\_trader\_state(self, context: Dict[str, Any]) -> Dict[str, Any]:

"""Оценка 'эмоционального состояния' трейдера"""

# Имитируем психологию трейдера на основе рыночных условий

recent\_wins = self.\_get\_recent\_performance()

volatility = context.get("volatility", {})

confidence = 0.5 # Базовая уверенность

# Корректируем на основе недавних результатов

if recent\_wins["win\_rate"] > 0.6:

confidence += 0.2

elif recent\_wins["win\_rate"] < 0.4:

confidence -= 0.2

# Волатильность влияет на стресс

stress\_level = min(volatility.get("daily", 0) \* 10, 1.0)

# FOMO индикатор (страх упустить возможность)

price\_change = context.get("df", pd.DataFrame()).get("close", pd.Series()).pct\_change().iloc[-10:].sum()

fomo\_level = min(abs(price\_change) \* 5, 1.0) if price\_change > 0 else 0

return {

"confidence": round(confidence, 2),

"stress\_level": round(stress\_level, 2),

"fomo\_level": round(fomo\_level, 2),

"recent\_performance": recent\_wins,

"trading\_mood": self.\_determine\_mood(confidence, stress\_level)

}

def \_determine\_mood(self, confidence: float, stress: float) -> str:

"""Определение настроения трейдера"""

if confidence > 0.7 and stress < 0.3:

return "confident"

elif confidence > 0.5 and stress < 0.5:

return "balanced"

elif confidence < 0.3 or stress > 0.7:

return "cautious"

else:

return "neutral"

def \_analyze\_execution(self, trade: Trade, entry\_record: Dict) -> Dict[str, Any]:

"""Анализ качества исполнения сделки"""

expected\_profit = entry\_record["expectations"]["target\_profit\_percent"]

actual\_profit = float(trade.profit\_percent) if trade.profit\_percent else 0

expected\_duration = timedelta(hours=entry\_record["expectations"]["expected\_duration\_hours"])

actual\_duration = trade.closed\_at - trade.created\_at if trade.closed\_at else timedelta(0)

return {

"profit\_vs\_expected": actual\_profit - expected\_profit,

"duration\_vs\_expected": str(actual\_duration - expected\_duration),

"exit\_timing": self.\_assess\_exit\_timing(trade),

"slippage": self.\_calculate\_slippage(trade, entry\_record),

"execution\_score": self.\_calculate\_execution\_score(trade, entry\_record)

}

def \_extract\_lessons(self, trade: Trade, entry\_record: Dict,

exit\_context: Dict[str, Any]) -> List[str]:

"""Извлечение уроков из сделки"""

lessons = []

# Анализируем что пошло не так или хорошо

if trade.profit and trade.profit > 0:

lessons.append(f"✅ Стратегия {entry\_record['signal']['strategy']} сработала в условиях {entry\_record['market\_context']['market\_phase']}")

if float(trade.profit\_percent or 0) > 5:

lessons.append("💰 Отличный результат! Проанализировать условия для повторения")

else:

lessons.append(f"❌ Убыток в условиях {entry\_record['market\_context']['market\_phase']}")

# Анализируем причины

if exit\_context.get("trend", {}).get("direction") != entry\_record["market\_context"]["trend"].get("direction"):

lessons.append("📊 Изменение тренда - нужно быстрее реагировать")

if exit\_context.get("volatility", {}).get("is\_high"):

lessons.append("⚡ Высокая волатильность - уменьшить размер позиции")

# Анализ риск-менеджмента

risk\_data = entry\_record["risk\_analysis"]

if risk\_data["risk\_reward\_ratio"] < 1.5:

lessons.append("⚠️ Низкое соотношение риск/прибыль - пересмотреть точки входа")

return lessons

def \_assess\_trade\_quality(self, trade: Trade, entry\_record: Dict) -> Dict[str, Any]:

"""Оценка качества сделки по множеству факторов"""

scores = {

"entry\_timing": 0,

"exit\_timing": 0,

"risk\_management": 0,

"profit\_capture": 0,

"strategy\_execution": 0

}

# Оценка времени входа (на основе последующего движения)

# TODO: Реализовать более сложную логику

# Оценка риск-менеджмента

risk\_data = entry\_record["risk\_analysis"]

if 1.5 <= risk\_data["risk\_reward\_ratio"] <= 3:

scores["risk\_management"] = 80

elif risk\_data["risk\_reward\_ratio"] > 3:

scores["risk\_management"] = 100

else:

scores["risk\_management"] = 50

# Оценка захвата прибыли

if trade.profit and trade.profit > 0:

if float(trade.profit\_percent or 0) >= entry\_record["expectations"]["target\_profit\_percent"]:

scores["profit\_capture"] = 100

else:

scores["profit\_capture"] = 70

else:

scores["profit\_capture"] = 0

# Общая оценка

total\_score = sum(scores.values()) / len(scores)

return {

"scores": scores,

"total\_score": round(total\_score, 1),

"grade": self.\_score\_to\_grade(total\_score)

}

def \_score\_to\_grade(self, score: float) -> str:

"""Преобразование оценки в букву"""

if score >= 90:

return "A+"

elif score >= 80:

return "A"

elif score >= 70:

return "B"

elif score >= 60:

return "C"

else:

return "D"

async def \_save\_entry(self, entry: Dict[str, Any]) -> None:

"""Сохранение записи в журнал"""

try:

async with aiofiles.open(self.current\_journal, 'a') as f:

await f.write(json.dumps(entry, ensure\_ascii=False) + '\n')

except Exception as e:

logger.error(f"Ошибка сохранения в журнал: {e}")

async def \_find\_entry\_record(self, trade\_id: int) -> Optional[Dict]:

"""Поиск записи входа для сделки"""

# Сначала ищем в кэше

for entry in reversed(self.recent\_entries):

if entry.get("trade\_id") == trade\_id and entry.get("type") == "ENTRY":

return entry

# Если не нашли, ищем в файле

try:

async with aiofiles.open(self.current\_journal, 'r') as f:

async for line in f:

entry = json.loads(line)

if entry.get("trade\_id") == trade\_id and entry.get("type") == "ENTRY":

return entry

except Exception as e:

logger.error(f"Ошибка поиска записи: {e}")

return None

def \_generate\_id(self) -> str:

"""Генерация уникального ID"""

import uuid

return str(uuid.uuid4())

def \_determine\_market\_phase(self, context: Dict) -> str:

"""Определение фазы рынка"""

# Упрощенная логика, можно расширить

trend = context.get("trend", {}).get("direction", "SIDEWAYS")

volatility = context.get("volatility", {}).get("daily", 0)

if trend == "UPTREND" and volatility < 0.02:

return "steady\_uptrend"

elif trend == "UPTREND" and volatility > 0.04:

return "volatile\_uptrend"

elif trend == "DOWNTREND":

return "downtrend"

else:

return "ranging"

def \_calculate\_btc\_correlation(self, context: Dict) -> float:

"""Расчет корреляции с BTC"""

# TODO: Реализовать расчет корреляции

return 0.0

def \_get\_recent\_performance(self) -> Dict[str, float]:

"""Получение последних результатов"""

db = SessionLocal()

try:

# Последние 20 сделок

recent\_trades = db.query(Trade).filter(

Trade.status == "CLOSED"

).order\_by(Trade.closed\_at.desc()).limit(20).all()

if not recent\_trades:

return {"win\_rate": 0.5, "avg\_profit": 0, "trades\_count": 0}

wins = len([t for t in recent\_trades if t.profit and t.profit > 0])

total\_profit = sum(t.profit or 0 for t in recent\_trades)

return {

"win\_rate": wins / len(recent\_trades),

"avg\_profit": total\_profit / len(recent\_trades),

"trades\_count": len(recent\_trades)

}

finally:

db.close()

def \_calculate\_position\_size\_percent(self, trade: Trade) -> float:

"""Расчет размера позиции в процентах от капитала"""

# TODO: Получить баланс и рассчитать

return 5.0

def \_assess\_exit\_timing(self, trade: Trade) -> str:

"""Оценка времени выхода"""

if not trade.profit:

return "unknown"

if trade.profit > 0:

# TODO: Проанализировать было ли движение дальше

return "good"

else:

# Вышли по стопу?

if abs(float(trade.exit\_price or 0) - float(trade.stop\_loss or 0)) < 0.01:

return "stop\_loss"

return "early\_exit"

def \_calculate\_slippage(self, trade: Trade, entry\_record: Dict) -> float:

"""Расчет проскальзывания"""

expected\_entry = entry\_record["signal"].get("price", 0)

actual\_entry = float(trade.entry\_price)

return abs(actual\_entry - expected\_entry) / expected\_entry \* 100 if expected\_entry > 0 else 0

def \_calculate\_execution\_score(self, trade: Trade, entry\_record: Dict) -> float:

"""Общая оценка исполнения"""

# Упрощенная логика

score = 50

if trade.profit and trade.profit > 0:

score += 25

if self.\_calculate\_slippage(trade, entry\_record) < 0.1:

score += 25

return score

async def \_update\_statistics(self, trade: Trade, entry\_record: Dict,

exit\_record: Dict) -> None:

"""Обновление общей статистики"""

# TODO: Обновить статистику в БД

pass

async def get\_journal\_analysis(self, days: int = 30) -> pd.DataFrame:

"""Получение анализа журнала за период"""

entries = []

cutoff\_date = datetime.utcnow() - timedelta(days=days)

# Читаем журналы за период

for journal\_file in sorted(self.journal\_dir.glob("journal\_\*.jsonl")):

if journal\_file.stat().st\_mtime < cutoff\_date.timestamp():

continue

async with aiofiles.open(journal\_file, 'r') as f:

async for line in f:

try:

entry = json.loads(line)

entries.append(entry)

except:

continue

# Конвертируем в DataFrame для анализа

if entries:

df = pd.DataFrame(entries)

df['timestamp'] = pd.to\_datetime(df['timestamp'])

# Добавляем дополнительные метрики

self.\_enrich\_dataframe(df)

return df

return pd.DataFrame()

def \_enrich\_dataframe(self, df: pd.DataFrame) -> None:

"""Обогащение DataFrame дополнительными метриками"""

# Группируем входы и выходы

trades\_df = df[df['type'].isin(['ENTRY', 'EXIT'])]

# TODO: Добавить расчет дополнительных метрик

pass

# Глобальный экземпляр

trade\_journal = TradeJournal() </artifact> </artifacts>

### 3. ****Обновление clean\_logging.py****

<artifacts> <artifact identifier="clean-logging-update" type="application/vnd.ant.code" language="python" title="Обновления для src/core/clean\_logging.py"> # Добавьте эти импорты в начало файла from .log\_manager import log\_manager, LogPriority, log\_trade, log\_analysis, log\_critical, log\_debug

# Обновите класс TradingLogFilter

class TradingLogFilter(logging.Filter): """Фильтр для торговых логов с интеграцией LogManager"""

# ... существующий код ...

def filter(self, record):

# Существующая логика фильтрации

result = super().filter(record)

if result:

# Отправляем в LogManager для умной обработки

asyncio.create\_task(self.\_send\_to\_log\_manager(record))

return result

async def \_send\_to\_log\_manager(self, record):

"""Отправка лога в LogManager"""

try:

# Определяем приоритет

priority = self.\_determine\_priority(record)

# Извлекаем метаданные если есть

metadata = getattr(record, 'metadata', {})

# Отправляем в LogManager

await log\_manager.log(

message=record.getMessage(),

level=record.levelname,

priority=priority,

metadata=metadata

)

except Exception as e:

# Не падаем из-за ошибки логирования

pass

def \_determine\_priority(self, record) -> int:

"""Определение приоритета лога"""

message = record.getMessage().lower()

# Критические события

if record.levelno >= logging.ERROR:

return LogPriority.CRITICAL

# Сделки

if any(word in message for word in ['trade', 'position', 'order', 'profit', 'loss']):

return LogPriority.HIGH

# Анализ

if any(word in message for word in ['analysis', 'signal', 'strategy']):

return LogPriority.MEDIUM

# Отладка

if record.levelno == logging.DEBUG:

return LogPriority.LOW

return LogPriority.NOISE

# Добавьте новые функции для удобного логирования

def log\_with\_metadata(logger, level, message, \*\*metadata): """Логирование с метаданными для LogManager""" extra = {'metadata': metadata} logger.log(level, message, extra=extra)

# Обновите функцию get\_clean\_logger

def get\_clean\_logger(name: str) -> logging.Logger: """Получить настроенный логгер с поддержкой метаданных""" if not \_logging\_manager.initialized: \_logging\_manager.setup\_logging()

logger = \_logging\_manager.get\_logger(name)

# Добавляем удобные методы

logger.log\_trade = lambda msg, trade\_id, \*\*kw: log\_with\_metadata(logger, logging.INFO, msg, trade\_id=trade\_id, \*\*kw)

logger.log\_analysis = lambda msg, strategy, \*\*kw: log\_with\_metadata(logger, logging.INFO, msg, strategy=strategy, analysis=True, \*\*kw)

logger.log\_critical = lambda msg, \*\*kw: log\_with\_metadata(logger, logging.ERROR, msg, critical=True, \*\*kw)

return logger

</artifact> </artifacts>

### 4. ****Исправление отображения баланса****

<artifacts> <artifact identifier="balance-fix" type="application/vnd.ant.code" language="python" title="Исправления для отображения баланса в api\_routes.py"> # Найдите функцию get\_dashboard\_data в api\_routes.py и обновите её:

@router.get("/api/dashboard") async def get\_dashboard\_data(db: Session = Depends(get\_db)): """Получить все данные для дашборда""" try: # Статус бота if bot\_manager: bot\_status = bot\_manager.get\_status()

# Получаем баланс

balance\_data = {"USDT": {"total": 0, "free": 0, "used": 0}}

try:

if bot\_manager.exchange:

balance = await bot\_manager.exchange.fetch\_balance()

if 'USDT' in balance:

balance\_data['USDT'] = {

"total": float(balance['USDT'].get('total', 0)),

"free": float(balance['USDT'].get('free', 0)),

"used": float(balance['USDT'].get('used', 0))

}

# Также сохраняем в БД для истории

balance\_record = Balance(

currency='USDT',

total=balance\_data['USDT']['total'],

free=balance\_data['USDT']['free'],

used=balance\_data['USDT']['used'],

timestamp=datetime.utcnow()

)

db.add(balance\_record)

db.commit()

except Exception as e:

logger.warning(f"Не удалось получить баланс с биржи: {e}")

# Получаем последний известный баланс из БД

last\_balance = db.query(Balance).filter(

Balance.currency == 'USDT'

).order\_by(desc(Balance.timestamp)).first()

if last\_balance:

balance\_data['USDT'] = {

"total": float(last\_balance.total),

"free": float(last\_balance.free),

"used": float(last\_balance.used)

}

else:

bot\_status = {"status": "unavailable", "is\_running": False}

balance\_data = {"USDT": {"total": 0, "free": 0, "used": 0}}

# ... остальной код ...

return {

"bot\_status": bot\_status,

"balance": balance\_data,

# ... остальные данные ...

}

# Также обновите WebSocket broadcast в ws\_manager:

async def \_broadcast\_worker(self): """Воркер для отправки обновлений""" while True: try: if bot\_manager: # Получаем актуальные данные status = bot\_manager.get\_status()

# Получаем баланс

balance\_data = {"USDT": {"total": 0, "free": 0, "used": 0}}

try:

if bot\_manager.exchange:

balance = await bot\_manager.exchange.fetch\_balance()

if 'USDT' in balance:

balance\_data['USDT'] = {

"total": float(balance['USDT'].get('total', 0)),

"free": float(balance['USDT'].get('free', 0)),

"used": float(balance['USDT'].get('used', 0))

}

except Exception as e:

logger.warning(f"Не удалось получить баланс для WebSocket: {e}")

# Отправляем обновление

await self.broadcast({

"type": "status\_update",

"data": {

\*\*status,

"balance": balance\_data

},

"timestamp": datetime.utcnow()

})

await asyncio.sleep(5) # Обновление каждые 5 секунд

except Exception as e:

logger.error(f"Ошибка в broadcast worker: {e}")

await asyncio.sleep(10)

</artifact> </artifacts>

### 5. ****Обновление дашборда для отображения баланса****

<artifacts> <artifact identifier="dashboard-balance-update" type="application/vnd.ant.code" language="javascript" title="Обновления JavaScript в dashboard.py для отображения баланса"> // Найдите функцию handleWebSocketMessage в dashboard.py и добавьте обработку баланса:

function handleWebSocketMessage(data) { console.log('WebSocket message:', data);

switch(data.type) {

case 'initial\_status':

case 'status\_update':

case 'status\_response':

updateFromStatusData(data.data);

// Обновляем баланс

if (data.data.balance) {

updateBalance(data.data.balance);

}

break;

// ... остальные case ...

}

}

// Добавьте новую функцию для обновления баланса: function updateBalance(balanceData) { if (balanceData.USDT) { const totalBalance = balanceData.USDT.total || 0; const freeBalance = balanceData.USDT.free || 0; const usedBalance = balanceData.USDT.used || 0;

// Обновляем отображение баланса

const balanceElement = document.getElementById('balance-display');

if (balanceElement) {

balanceElement.innerHTML = `

<strong>Баланс USDT:</strong><br>

Всего: $${totalBalance.toFixed(2)}<br>

Свободно: $${freeBalance.toFixed(2)}<br>

Используется: $${usedBalance.toFixed(2)}

`;

}

// Также можно добавить в заголовок

const headerBalance = document.getElementById('header-balance');

if (headerBalance) {

headerBalance.textContent = `$${totalBalance.toFixed(2)} USDT`;

}

}

}

// Обновите функцию loadStats для включения баланса: async function loadStats() { try { // Загружаем данные дашборда const dashboardResponse = await fetch('/api/dashboard', { headers: { 'Authorization': Bearer ${authToken} } });

if (dashboardResponse.ok) {

const dashboardData = await dashboardResponse.json();

// Обновляем баланс

if (dashboardData.balance) {

updateBalance(dashboardData.balance);

}

// ... остальной код обновления статистики ...

}

} catch (error) {

console.error('Error loading stats:', error);

}

}

// Добавьте в HTML часть дашборда отображение баланса: // Найдите stats-grid и добавьте новую карточку:

<div class="stat-card"> <div class="stat-number" id="header-balance">$0.00 USDT</div> <div class="stat-label">Баланс</div> <div id="balance-display" style="font-size: 0.8em; margin-top: 10px; color: #7f8c8d;"> Загрузка... </div> </div> </artifact> </artifacts>

### 6. ****Создание модуля для графиков****

<artifacts> <artifact identifier="charts-module" type="application/vnd.ant.code" language="python" title="src/web/charts.py - Модуль для графиков"> """ Модуль для интеграции графиков TradingView """ from fastapi import APIRouter, Depends, HTTPException from fastapi.responses import HTMLResponse from typing import List, Optional import json

from ..core.models import User from .auth import get\_current\_user

router = APIRouter(prefix="/charts", tags=["charts"])

@router.get("/", response\_class=HTMLResponse) async def charts\_page(current\_user: User = Depends(get\_current\_user)): """Страница с графиками""" return get\_charts\_html()

@router.get("/config") async def get\_chart\_config(current\_user: User = Depends(get\_current\_user)): """Получить конфигурацию графиков""" return { "symbols": [ {"symbol": "BYBIT:BTCUSDT", "name": "Bitcoin/USDT"}, {"symbol": "BYBIT:ETHUSDT", "name": "Ethereum/USDT"}, {"symbol": "BYBIT:BNBUSDT", "name": "BNB/USDT"}, {"symbol": "BYBIT:SOLUSDT", "name": "Solana/USDT"} ], "defaultSymbol": "BYBIT:BTCUSDT", "interval": "5", "theme": "dark" }

def get\_charts\_html() -> str: """HTML страница с графиками""" return """

<!DOCTYPE html> <html lang="ru"> <head> <meta charset="UTF-8"> <meta name="viewport" content="width=device-width, initial-scale=1.0"> <title>📊 Графики - Crypto Trading Bot</title> <style> \* { margin: 0; padding: 0; box-sizing: border-box; }

body {

font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;

background: #1a1a1a;

color: #fff;

overflow: hidden;

}

.header {

background: rgba(0, 0, 0, 0.8);

padding: 10px 20px;

display: flex;

justify-content: space-between;

align-items: center;

border-bottom: 1px solid #333;

}

.header h1 {

font-size: 1.5em;

color: #fff;

}

.controls {

display: flex;

gap: 15px;

align-items: center;

}

.symbol-selector {

padding: 8px 15px;

background: #2a2a2a;

border: 1px solid #444;

color: #fff;

border-radius: 5px;

cursor: pointer;

}

.btn {

padding: 8px 20px;

background: #3498db;

color: white;

border: none;

border-radius: 5px;

cursor: pointer;

font-weight: 600;

transition: all 0.3s ease;

}

.btn:hover {

background: #2980b9;

transform: translateY(-1px);

}

.charts-container {

display: grid;

grid-template-columns: 1fr 1fr;

grid-template-rows: 1fr 1fr;

gap: 10px;

padding: 10px;

height: calc(100vh - 60px);

}

.chart-wrapper {

background: #1e1e1e;

border: 1px solid #333;

border-radius: 8px;

overflow: hidden;

position: relative;

}

.chart-header {

background: rgba(0, 0, 0, 0.5);

padding: 10px;

position: absolute;

top: 0;

left: 0;

right: 0;

z-index: 10;

display: flex;

justify-content: space-between;

align-items: center;

}

.chart-title {

font-weight: 600;

font-size: 1.1em;

}

.chart-controls {

display: flex;

gap: 10px;

}

.timeframe-btn {

padding: 4px 12px;

background: #333;

border: 1px solid #555;

color: #ccc;

border-radius: 3px;

cursor: pointer;

font-size: 0.9em;

}

.timeframe-btn.active {

background: #3498db;

color: white;

border-color: #3498db;

}

.chart-container {

width: 100%;

height: 100%;

}

.tradingview-widget-container {

height: 100%;

}

.fullscreen-btn {

position: absolute;

top: 10px;

right: 10px;

padding: 8px;

background: rgba(52, 152, 219, 0.8);

border: none;

border-radius: 5px;

color: white;

cursor: pointer;

z-index: 20;

}

.loading {

display: flex;

justify-content: center;

align-items: center;

height: 100%;

font-size: 1.2em;

color: #666;

}

@media (max-width: 1200px) {

.charts-container {

grid-template-columns: 1fr;

grid-template-rows: repeat(4, 400px);

}

}

</style>

</head> <body> <div class="header"> <h1>📊 Графики криптовалют</h1> <div class="controls"> <select class="symbol-selector" id="mainSymbol"> <option value="BYBIT:BTCUSDT">Bitcoin (BTC/USDT)</option> <option value="BYBIT:ETHUSDT">Ethereum (ETH/USDT)</option> <option value="BYBIT:BNBUSDT">BNB (BNB/USDT)</option> <option value="BYBIT:SOLUSDT">Solana (SOL/USDT)</option> </select> <button class="btn" onclick="refreshCharts()">🔄 Обновить</button> <button class="btn" onclick="window.location.href='/'">🏠 Дашборд</button> </div> </div>

<div class="charts-container">

<!-- Chart 1: Main chart with indicators -->

<div class="chart-wrapper">

<div class="chart-header">

<span class="chart-title">Bitcoin/USDT</span>

<div class="chart-controls">

<button class="timeframe-btn" data-interval="1">1m</button>

<button class="timeframe-btn active" data-interval="5">5m</button>

<button class="timeframe-btn" data-interval="15">15m</button>

<button class="timeframe-btn" data-interval="60">1h</button>

<button class="timeframe-btn" data-interval="240">4h</button>

<button class="timeframe-btn" data-interval="D">1D</button>

</div>

</div>

<div id="chart1" class="chart-container">

<div class="loading">Загрузка графика...</div>

</div>

</div>

<!-- Chart 2: ETH -->

<div class="chart-wrapper">

<div class="chart-header">

<span class="chart-title">Ethereum/USDT</span>

</div>

<div id="chart2" class="chart-container">

<div class="loading">Загрузка графика...</div>

</div>

</div>

<!-- Chart 3: BNB -->

<div class="chart-wrapper">

<div class="chart-header">

<span class="chart-title">BNB/USDT</span>

</div>

<div id="chart3" class="chart-container">

<div class="loading">Загрузка графика...</div>

</div>

</div>

<!-- Chart 4: SOL -->

<div class="chart-wrapper">

<div class="chart-header">

<span class="chart-title">Solana/USDT</span>

</div>

<div id="chart4" class="chart-container">

<div class="loading">Загрузка графика...</div>

</div>

</div>

</div>

<script>

// Конфигурация графиков

const chartConfigs = [

{ id: 'chart1', symbol: 'BYBIT:BTCUSDT', title: 'Bitcoin/USDT' },

{ id: 'chart2', symbol: 'BYBIT:ETHUSDT', title: 'Ethereum/USDT' },

{ id: 'chart3', symbol: 'BYBIT:BNBUSDT', title: 'BNB/USDT' },

{ id: 'chart4', symbol: 'BYBIT:SOLUSDT', title: 'Solana/USDT' }

];

let currentInterval = '5';

let charts = {};

// Инициализация графиков

function initCharts() {

// Используем lightweight-charts или TradingView widget

loadTradingViewWidgets();

}

function loadTradingViewWidgets() {

chartConfigs.forEach((config, index) => {

const container = document.getElementById(config.id);

container.innerHTML = '';

// Создаем iframe для TradingView widget

const widgetContainer = document.createElement('div');

widgetContainer.className = 'tradingview-widget-container';

widgetContainer.style.height = '100%';

const iframe = document.createElement('iframe');

iframe.scrolling = 'no';

iframe.allowTransparency = true;

iframe.frameBorder = '0';

iframe.style.width = '100%';

iframe.style.height = '100%';

// Параметры виджета

const widgetParams = {

symbol: config.symbol,

interval: currentInterval,

theme: 'dark',

style: '1',

locale: 'ru',

toolbar\_bg: '#1e1e1e',

enable\_publishing: false,

hide\_side\_toolbar: false,

allow\_symbol\_change: false,

container\_id: `tradingview\_${index}`,

studies: index === 0 ? ['RSI@tv-basicstudies', 'MACD@tv-basicstudies'] : []

};

// URL для встраивания

const params = new URLSearchParams(widgetParams);

iframe.src = `https://s.tradingview.com/widgetembed/?${params.toString()}`;

widgetContainer.appendChild(iframe);

container.appendChild(widgetContainer);

});

}

// Альтернатива - использование lightweight-charts

async function loadLightweightCharts() {

// Загружаем библиотеку

const script = document.createElement('script');

script.src = 'https://unpkg.com/lightweight-charts/dist/lightweight-charts.standalone.production.js';

script.onload = () => {

chartConfigs.forEach(config => {

createLightweightChart(config);

});

};

document.head.appendChild(script);

}

function createLightweightChart(config) {

const container = document.getElementById(config.id);

container.innerHTML = '';

const chart = LightweightCharts.createChart(container, {

width: container.clientWidth,

height: container.clientHeight,

layout: {

backgroundColor: '#1e1e1e',

textColor: '#d1d4dc',

},

grid: {

vertLines: {

color: '#2a2a2a',

},

horzLines: {

color: '#2a2a2a',

},

},

priceScale: {

borderColor: '#485c7b',

},

timeScale: {

borderColor: '#485c7b',

timeVisible: true,

secondsVisible: false,

},

});

const candlestickSeries = chart.addCandlestickSeries({

upColor: '#26a69a',

downColor: '#ef5350',

borderVisible: false,

wickUpColor: '#26a69a',

wickDownColor: '#ef5350',

});

// Здесь нужно загрузить данные с вашего API

loadChartData(config.symbol, currentInterval).then(data => {

candlestickSeries.setData(data);

});

charts[config.id] = chart;

// Обновление размера при изменении окна

window.addEventListener('resize', () => {

chart.applyOptions({

width: container.clientWidth,

height: container.clientHeight

});

});

}

async function loadChartData(symbol, interval) {

// Здесь должен быть запрос к вашему API для получения данных

// Пример:

try {

const response = await fetch(`/api/chart-data/${symbol}?interval=${interval}`, {

headers: {

'Authorization': `Bearer ${localStorage.getItem('authToken')}`

}

});

if (response.ok) {

const data = await response.json();

return data.map(candle => ({

time: candle.timestamp / 1000,

open: candle.open,

high: candle.high,

low: candle.low,

close: candle.close

}));

}

} catch (error) {

console.error('Error loading chart data:', error);

}

return [];

}

// Обработчики событий

document.querySelectorAll('.timeframe-btn').forEach(btn => {

btn.addEventListener('click', function() {

document.querySelectorAll('.timeframe-btn').forEach(b => b.classList.remove('active'));

this.classList.add('active');

currentInterval = this.dataset.interval;

refreshCharts();

});

});

document.getElementById('mainSymbol').addEventListener('change', function() {

const newSymbol = this.value;

chartConfigs[0].symbol = newSymbol;

chartConfigs[0].title = this.options[this.selectedIndex].text;

document.querySelector('.chart-title').textContent = chartConfigs[0].title;

refreshCharts();

});

function refreshCharts() {

initCharts();

}

// Инициализация при загрузке

document.addEventListener('DOMContentLoaded', () => {

// Проверяем авторизацию

if (!localStorage.getItem('authToken')) {

window.location.href = '/';

return;

}

initCharts();

// Автообновление каждые 30 секунд

setInterval(refreshCharts, 30000);

});

</script>

</body> </html> """ </artifact> </artifacts>

### 7. ****Создание ML модуля для оптимизации стратегий****

<artifacts> <artifact identifier="strategy-optimizer" type="application/vnd.ant.code" language="python" title="src/ml/strategy\_optimizer.py - ML оптимизатор стратегий"> """ Машинное обучение для оптимизации параметров стратегий """ import numpy as np import pandas as pd from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor from sklearn.model\_selection import train\_test\_split, cross\_val\_score from sklearn.preprocessing import StandardScaler from sklearn.metrics import mean\_squared\_error, r2\_score import optuna import joblib from typing import Dict, List, Tuple, Optional, Any from datetime import datetime, timedelta import asyncio from pathlib import Path

from ..core.database import SessionLocal from ..core.models import Trade, Signal from ..analysis.trade\_journal import trade\_journal from ..core.clean\_logging import get\_clean\_logger

logger = get\_clean\_logger(**name**)

class StrategyOptimizer: """ Оптимизатор параметров стратегий с использованием ML

Использует:

- RandomForest для предсказания прибыльности

- Optuna для оптимизации гиперпараметров

- Reinforcement Learning для адаптации

"""

def \_\_init\_\_(self):

self.models\_dir = Path("models/optimizers")

self.models\_dir.mkdir(parents=True, exist\_ok=True)

# Модели для каждой стратегии

self.models: Dict[str, Any] = {}

self.scalers: Dict[str, StandardScaler] = {}

self.optimal\_params: Dict[str, Dict] = {}

# История оптимизации

self.optimization\_history = []

# Загружаем существующие модели

self.\_load\_models()

async def optimize\_strategy(self, strategy\_name: str,

lookback\_days: int = 30) -> Dict[str, Any]:

"""

Оптимизация параметров конкретной стратегии

Args:

strategy\_name: Название стратегии

lookback\_days: Период для анализа

Returns:

Оптимальные параметры

"""

logger.info(f"🔧 Начинаем оптимизацию стратегии {strategy\_name}")

# Получаем исторические данные

historical\_data = await self.\_get\_historical\_data(strategy\_name, lookback\_days)

if len(historical\_data) < 50:

logger.warning(f"Недостаточно данных для оптимизации {strategy\_name}")

return self.\_get\_default\_params(strategy\_name)

# Подготавливаем данные для ML

X, y = self.\_prepare\_training\_data(historical\_data)

# Обучаем модель

model = self.\_train\_model(strategy\_name, X, y)

# Оптимизируем параметры с Optuna

optimal\_params = await self.\_optimize\_with\_optuna(strategy\_name, model, X, y)

# Сохраняем результаты

self.optimal\_params[strategy\_name] = optimal\_params

self.\_save\_optimization\_result(strategy\_name, optimal\_params)

logger.info(f"✅ Оптимизация завершена для {strategy\_name}: {optimal\_params}")

return optimal\_params

async def \_get\_historical\_data(self, strategy\_name: str,

days: int) -> pd.DataFrame:

"""Получение исторических данных по стратегии"""

db = SessionLocal()

try:

# Получаем сделки

end\_date = datetime.utcnow()

start\_date = end\_date - timedelta(days=days)

trades = db.query(Trade).filter(

Trade.strategy == strategy\_name,

Trade.created\_at >= start\_date,

Trade.status == "CLOSED"

).all()

# Получаем данные из журнала

journal\_data = await trade\_journal.get\_journal\_analysis(days)

# Объединяем данные

data = []

for trade in trades:

trade\_data = {

"trade\_id": trade.id,

"symbol": trade.symbol,

"profit": float(trade.profit) if trade.profit else 0,

"profit\_percent": float(trade.profit\_percent) if trade.profit\_percent else 0,

"duration": (trade.closed\_at - trade.created\_at).total\_seconds() if trade.closed\_at else 0,

"entry\_price": float(trade.entry\_price),

"exit\_price": float(trade.exit\_price) if trade.exit\_price else 0

}

# Добавляем данные из журнала если есть

journal\_entry = journal\_data[

journal\_data['trade\_id'] == trade.id

] if not journal\_data.empty else pd.DataFrame()

if not journal\_entry.empty:

entry\_record = journal\_entry[journal\_entry['type'] == 'ENTRY'].iloc[0] if len(journal\_entry[journal\_entry['type'] == 'ENTRY']) > 0 else None

if entry\_record is not None:

# Добавляем индикаторы

indicators = entry\_record.get('indicators', {})

for key, value in indicators.items():

trade\_data[f"indicator\_{key}"] = value

# Добавляем рыночный контекст

market\_context = entry\_record.get('market\_context', {})

trade\_data['market\_trend'] = market\_context.get('trend', {}).get('direction', 'UNKNOWN')

trade\_data['volatility'] = market\_context.get('volatility', {}).get('daily', 0)

trade\_data['volume\_ratio'] = market\_context.get('volume\_analysis', {}).get('ratio', 1)

data.append(trade\_data)

return pd.DataFrame(data)

finally:

db.close()

def \_prepare\_training\_data(self, df: pd.DataFrame) -> Tuple[np.ndarray, np.ndarray]:

"""Подготовка данных для обучения"""

# Признаки

feature\_columns = [col for col in df.columns if col.startswith('indicator\_') or

col in ['volatility', 'volume\_ratio', 'duration']]

# Добавляем категориальные признаки

if 'market\_trend' in df.columns:

trend\_dummies = pd.get\_dummies(df['market\_trend'], prefix='trend')

df = pd.concat([df, trend\_dummies], axis=1)

feature\_columns.extend(trend\_dummies.columns.tolist())

# Целевая переменная - прибыль в процентах

target = 'profit\_percent'

# Фильтруем только нужные колонки

available\_features = [col for col in feature\_columns if col in df.columns]

if not available\_features:

# Если нет признаков, используем базовые

logger.warning("Нет доступных признаков, используем базовые")

X = df[['duration']].fillna(0).values

y = df[target].fillna(0).values

else:

X = df[available\_features].fillna(0).values

y = df[target].fillna(0).values

return X, y

def \_train\_model(self, strategy\_name: str, X: np.ndarray,

y: np.ndarray) -> RandomForestRegressor:

"""Обучение модели для предсказания прибыльности"""

# Разделяем данные

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X, y, test\_size=0.2, random\_state=42

)

# Масштабирование

scaler = StandardScaler()

X\_train\_scaled = scaler.fit\_transform(X\_train)

X\_test\_scaled = scaler.transform(X\_test)

# Обучаем модель

model = GradientBoostingRegressor(

n\_estimators=100,

learning\_rate=0.1,

max\_depth=5,

random\_state=42

)

model.fit(X\_train\_scaled, y\_train)

# Оценка

train\_score = model.score(X\_train\_scaled, y\_train)

test\_score = model.score(X\_test\_scaled, y\_test)

logger.info(f"📊 Модель для {strategy\_name} обучена. "

f"Train R²: {train\_score:.3f}, Test R²: {test\_score:.3f}")

# Сохраняем

self.models[strategy\_name] = model

self.scalers[strategy\_name] = scaler

# Сохраняем на диск

model\_path = self.models\_dir / f"{strategy\_name}\_model.pkl"

scaler\_path = self.models\_dir / f"{strategy\_name}\_scaler.pkl"

joblib.dump(model, model\_path)

joblib.dump(scaler, scaler\_path)

return model

async def \_optimize\_with\_optuna(self, strategy\_name: str, model: Any,

X: np.ndarray, y: np.ndarray) -> Dict[str, Any]:

"""Оптимизация параметров с помощью Optuna"""

def objective(trial):

# Определяем параметры для оптимизации в зависимости от стратегии

params = self.\_get\_strategy\_params(strategy\_name, trial)

# Симулируем результаты с этими параметрами

# В реальности здесь должен быть бэктест

simulated\_profit = self.\_simulate\_strategy(strategy\_name, params, X, y, model)

return simulated\_profit

# Создаем исследование

study = optuna.create\_study(

direction='maximize',

sampler=optuna.samplers.TPESampler(seed=42)

)

# Оптимизируем

study.optimize(objective, n\_trials=50, n\_jobs=1)

# Лучшие параметры

best\_params = study.best\_params

best\_value = study.best\_value

logger.info(f"🎯 Лучшие параметры для {strategy\_name}: {best\_params}")

logger.info(f"💰 Ожидаемая прибыль: {best\_value:.2f}%")

return best\_params

def \_get\_strategy\_params(self, strategy\_name: str, trial: optuna.Trial) -> Dict:

"""Получение параметров для оптимизации"""

if strategy\_name == "momentum":

return {

"rsi\_period": trial.suggest\_int("rsi\_period", 10, 20),

"ema\_fast": trial.suggest\_int("ema\_fast", 5, 15),

"ema\_slow": trial.suggest\_int("ema\_slow", 15, 30),

"rsi\_oversold": trial.suggest\_int("rsi\_oversold", 20, 35),

"rsi\_overbought": trial.suggest\_int("rsi\_overbought", 65, 80),

"min\_momentum\_score": trial.suggest\_float("min\_momentum\_score", 0.5, 0.8)

}

elif strategy\_name == "scalping":

return {

"bb\_period": trial.suggest\_int("bb\_period", 15, 25),

"bb\_std": trial.suggest\_float("bb\_std", 1.5, 2.5),

"rsi\_period": trial.suggest\_int("rsi\_period", 5, 10),

"min\_profit\_percent": trial.suggest\_float("min\_profit\_percent", 0.2, 0.5),

"max\_loss\_percent": trial.suggest\_float("max\_loss\_percent", 0.3, 0.7)

}

elif strategy\_name == "multi\_indicator":

return {

"min\_confidence": trial.suggest\_float("min\_confidence", 0.5, 0.75),

"min\_indicators\_confirm": trial.suggest\_int("min\_indicators\_confirm", 2, 4),

"rsi\_oversold": trial.suggest\_int("rsi\_oversold", 25, 35),

"rsi\_overbought": trial.suggest\_int("rsi\_overbought", 65, 75),

"adx\_threshold": trial.suggest\_int("adx\_threshold", 20, 30)

}

else:

# Дефолтные параметры

return {

"stop\_loss\_percent": trial.suggest\_float("stop\_loss\_percent", 1.0, 3.0),

"take\_profit\_percent": trial.suggest\_float("take\_profit\_percent", 2.0, 6.0),

"position\_size\_percent": trial.suggest\_float("position\_size\_percent", 3.0, 10.0)

}

def \_simulate\_strategy(self, strategy\_name: str, params: Dict,

X: np.ndarray, y: np.ndarray, model: Any) -> float:

"""Симуляция результатов стратегии с заданными параметрами"""

# Это упрощенная симуляция

# В реальности здесь должен быть полный бэктест

# Используем модель для предсказания

scaler = self.scalers.get(strategy\_name)

if scaler:

X\_scaled =

тут написание оборвалось, нужно продолжить написание!