iT EXPERTISE

РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

Cognitum AI Platform
Версия 0.4.4

ООО «ИТ-Экспертиза»

Дата сборки: 2025-12-23 Сборка: #522fc53

Оглавление

1 Термины и сокращения 4

1.1 Специфичные термины платформы 4

2 Введение 6

2.1 Назначение платформы 6

2.2 Целевая аудитория 6

2.3 Область применения 7

3 Архитектура платформы 9

3.1 Компоненты системы 9

3.2 Модель взаимодействия компонентов 10

3.3 Модель развёртывания 11

4 Веб-интерфейс управления 14

4.1 Основные разделы интерфейса 14

4.2 Навигация 14

4.3 Аутентификация и авторизация 14

4.4 Панель мониторинга (Dashboard) 16

4.5 Управление агентами 18

4.6 Управление LLM-провайдерами 20

4.7 Работа с очередями задач (Queues) 22

4.8 Управление хранилищами (Storages) 24

4.9 Просмотр логов 26

4.10 Тестирование моделей (Chat) 28

4.11 Администрирование 30

5 API платформы 33

5.1 Форматы API 33

5.2 Документация API 33

5.3 OpenAI-совместимый API 33

5.4 REST API для задач (Jobs) 35

5.5 Аутентификация в API 38

6 SDK для разработки агентов 40

6.1 Требования 40

6.2 Исходный код 40

6.3 Установка SDK 40

6.4 Конфигурация агента (AgentConfig) 41

6.5 Обработка задач (Job Handlers) 42

6.6 Чат-обработчики (Chat Handlers) 44

6.7 LLM-вызовы через платформу 46

6.8 Логирование и метрики 47

6.9 Работа с хранилищами 49

7 Система задач (Jobs) 51

7.1 Основные возможности 51

7.2 Концепция задач 51

7.3 Жизненный цикл задачи 52

7.4 Синхронное и асинхронное выполнение 54

7.5 Зависимости задач (Prerequisites) 55
7.6 Отложенное выполнение 57

8 Интеграция с внешними системами 61

8.1 Подключение LLM-провайдеров 61
8.2 Интеграция с LDAP/Active Directory 62
8.3 Подключение S3-хранилищ 64

9 Приложения 66

9.1 Справочник переменных окружения 66
9.2 Коды ошибок 67
9.3 Примеры конфигураций 68

iT EXPERTISE

1 Термины и сокращения

В настоящем документе используются следующие термины и сокращения:

Термин	Определение
AI-агент	Автономный программный модуль, выполняющий задачи с использованием технологий искусственного интеллекта
API	Application Programming Interface — программный интерфейс приложения
Control Plane	Централизованный компонент управления платформой, координирующий работу агентов
Docker	Платформа контейнеризации для развертывания приложений
Embedding	Векторное представление текста, используемое для семантического поиска
JetStream	Подсистема персистентных сообщений в NATS
Job	Задача — единица работы, выполняемая агентом
JWT	JSON Web Token — стандарт токенов для аутентификации
LDAP	Lightweight Directory Access Protocol — протокол доступа к каталогам
LLM	Large Language Model — большая языковая модель
Loki	Система агрегации логов от Grafana
MCP	Model Context Protocol — протокол для расширения возможностей LLM
NATS	Высокопроизводительный брокер сообщений
Neo4j	Графовая база данных
Ollama	Платформа для локального запуска LLM-моделей
PostgreSQL	Реляционная СУБД с открытым исходным кодом
Qdrant	Векторная база данных для хранения эмбеддингов
RAG	Retrieval-Augmented Generation — генерация с дополнением из внешних источников
REST	Representational State Transfer — архитектурный стиль API
S3	Simple Storage Service — объектное хранилище (стандарт Amazon)
SDK	Software Development Kit — набор инструментов разработчика
SMTP	Simple Mail Transfer Protocol — протокол отправки электронной почты
SSE	Server-Sent Events — технология потоковой передачи данных
Stream	Поток сообщений в JetStream с гарантией доставки
TTL	Time To Live — время жизни объекта
UUID	Universally Unique Identifier — универсальный уникальный идентификатор

1.1 Специфичные термины платформы

Термин	Определение
Cognitum	Название платформы для создания и управления AI-агентами
Платформа	Cognitum AI Platform — программный комплекс
Агент	AI-агент, зарегистрированный в платформе
Провайдер LLM	Источник языковых моделей (OpenAI, Azure, Ollama)
Хранилище	Внешнее хранилище данных, подключенное к платформе
Очередь	JetStream-стрим для обмена сообщениями
Задача (Job)	Асинхронная единица работы с жизненным циклом
Чат-модель	Модель агента, доступная через Chat API

Термин	Определение
Дистрибутив	Пакет для развертывания платформы

2 Введение

Настоящий документ является руководством пользователя программного продукта **Cognitum AI Platform** (далее — Платформа, Cognitum).

Документ содержит описание функциональных характеристик платформы, её архитектуры, пользовательского интерфейса, программных интерфейсов (API) и средств разработки агентов (SDK).

Руководство предназначено для следующих категорий пользователей:

- Администраторы платформы
- Разработчики AI-агентов
- Интеграторы корпоративных систем
- Технические специалисты IT-подразделений

Перед началом работы с платформой рекомендуется ознакомиться с разделами «Термины и сокращения» и «Архитектура платформы».

2.1 Назначение платформы

Cognitum AI Platform — корпоративная платформа для создания, развёртывания и управления AI-агентами на предприятии.

2.1.1 Основные функции платформы

1. **Централизованное управление AI-агентами**
   - Регистрация и мониторинг агентов
   - Просмотр логов и метрик в реальном времени
   - Управление жизненным циклом агентов
2. **Единая точка доступа к LLM-моделям**
   - Поддержка множества провайдеров (OpenAI, Azure, Ollama)
   - Централизованный учёт использования
   - Маршрутизация запросов к моделям
3. **Система асинхронных задач**
   - Очереди задач с гарантированной доставкой
   - Поддержка зависимостей между задачами
   - Отложенное выполнение и TTL
4. **OpenAI-совместимый API**
   - Интеграция с существующими инструментами
   - Использование моделей агентов из внешних приложений
5. **SDK для разработки агентов**
   - Python SDK с поддержкой async/await
   - Готовые абстракции для работы с LLM
   - Автоматическое логирование и метрики

2.1.2 Ключевые преимущества

- **On-premise развёртывание** — данные остаются внутри периметра предприятия
- **Поддержка локальных LLM** — работа с моделями через Ollama
- **Контейнеризация** — простое развёртывание и масштабирование
- **Единый интерфейс** — управление всеми агентами из одного места

2.2 Целевая аудитория

Платформа Cognitum предназначена для следующих категорий пользователей:

2.2.1 Администраторы IT-инфраструктуры

Задачи: - Развёртывание и настройка платформы - Управление пользователями и правами доступа - Мониторинг состояния системы - Резервное копирование и восстановление

Необходимые знания: - Администрирование Linux/Windows серверов - Работа с Docker и docker-compose - Базовые знания PostgreSQL - Понимание сетевых протоколов

2.2.2 Разработчики AI-агентов

Задачи: - Создание агентов с использованием SDK - Интеграция с LLM-моделями - Обработка задач из очередей - Отладка и тестирование агентов

Необходимые знания: - Программирование на Python 3.10+ - Асинхронное программирование (asyncio) - Работа с REST API - Базовое понимание работы LLM

2.2.3 Интеграторы корпоративных систем

Задачи: - Интеграция платформы с корпоративными системами - Настройка обмена данными через API - Подключение внешних хранилищ - Настройка LDAP-аутентификации

Необходимые знания: - Проектирование интеграционных решений - Работа с REST/OpenAI API - Настройка LDAP/Active Directory - Работа с S3-совместимыми хранилищами

2.2.4 Бизнес-пользователи

Задачи: - Использование чат-интерфейса для взаимодействия с агентами - Просмотр результатов выполнения задач - Тестирование моделей

Необходимые знания: - Базовые навыки работы с веб-интерфейсами - Понимание принципов работы с AI-ассистентами

2.3 Область применения

Платформа Cognitum может применяться в различных сценариях автоматизации бизнес-процессов с использованием искусственного интеллекта.

2.3.1 Типовые сценарии использования

2.3.1.1 Обработка документов

- Извлечение данных из неструктурированных документов
- Классификация и маршрутизация документов
- Генерация сводок и резюме
- Перевод документов

2.3.1.2 Корпоративные ассистенты

- Ответы на вопросы сотрудников по внутренним регламентам
- Поиск информации в корпоративной базе знаний
- Автоматизация типовых запросов в службу поддержки

2.3.1.3 Анализ данных

- Анализ текстовых отчётов и обратной связи
- Выявление паттернов в данных
- Генерация аналитических отчётов

2.3.1.4 Интеграция с бизнес-системами

- Обогащение данных из внешних источников
- Автоматическая обработка входящих запросов

- Интеграция AI-функций в существующие workflow

2.3.2 Ограничения применения

Платформа **не предназначена** для:

- Задач, требующих гарантированного времени отклика менее 100 мс
- Обработки потоковых данных в реальном времени
- Задач, критичных к безопасности без дополнительных мер защиты
- Замены специализированных систем машинного обучения

2.3.3 Требования к окружению

Для эффективной работы платформы рекомендуется:

- Стабильное сетевое соединение между компонентами
- Достаточный объём оперативной памяти для LLM-моделей (при локальном использовании)
- SSD-накопители для баз данных
- Резервирование критичных компонентов

3 Архитектура платформы

Cognitum построена на базе микросервисной архитектуры с использованием современных технологий обработки сообщений и контейнеризации.

Платформа состоит из следующих основных компонентов:

- **Веб-приложение** — фронтенд (Next.js) и бэкенд (FastAPI)
- **Брокер сообщений** — NATS с JetStream
- **Базы данных** — PostgreSQL, Qdrant, Neo4j
- **Система логирования** — Grafana Loki
- **AI-агенты** — контейнеризированные микросервисы

Все компоненты взаимодействуют через единую шину сообщений NATS, что обеспечивает слабую связанность и высокую отказоустойчивость системы.

3.1 Компоненты системы

3.1.1 Веб-приложение (cognitum-app)

Основной компонент платформы, объединяющий веб-интерфейс и серверную логику.

Фронтенд (Next.js 14): - Современный веб-интерфейс для управления платформой - Серверный рендеринг для быстрой загрузки - Адаптивный дизайн

Бэкенд (FastAPI): - REST API для управления платформой - OpenAI-совместимый API для работы с моделями - Control Plane для координации агентов - Job Scheduler для планирования задач

3.1.2 NATS JetStream

Высокопроизводительный брокер сообщений с поддержкой персистентности.

Основные стримы:

Стрим	Subjects	Назначение
AGENTS	`agent.>`	Логи, метрики, healthcheck агентов
JOBS	`job.>`	Очередь задач и результаты
CONTROL	`control.>`	Управляющие команды

Преимущества JetStream: - Гарантированная доставка сообщений (at-least-once) - Персистентное хранение сообщений - Возможность повторной обработки (replay) - Durable consumers для отказоустойчивости

3.1.3 PostgreSQL

Основная реляционная база данных платформы.

Хранимые данные: - Конфигурации агентов и LLM-провайдеров - Пользователи и права доступа - Задачи (jobs) и их результаты - API-токены - Аудитные записи

3.1.4 Qdrant

Векторная база данных для хранения эмбеддингов.

Применение: - Семантический поиск - RAG-системы (Retrieval-Augmented Generation) - Хранение векторных представлений документов

3.1.5 Neo4j

Графовая база данных для сложных связей.

Применение: - Хранение контекстов диалогов - Связи между сущностями - Knowledge graphs

3.1.6 Grafana Loki

Система агрегации и хранения логов.

Возможности: - Централизованный сбор логов со всех компонентов - Поиск по логам через веб-интерфейс - Интеграция с Grafana для визуализации - Автоматическая ротация и архивация

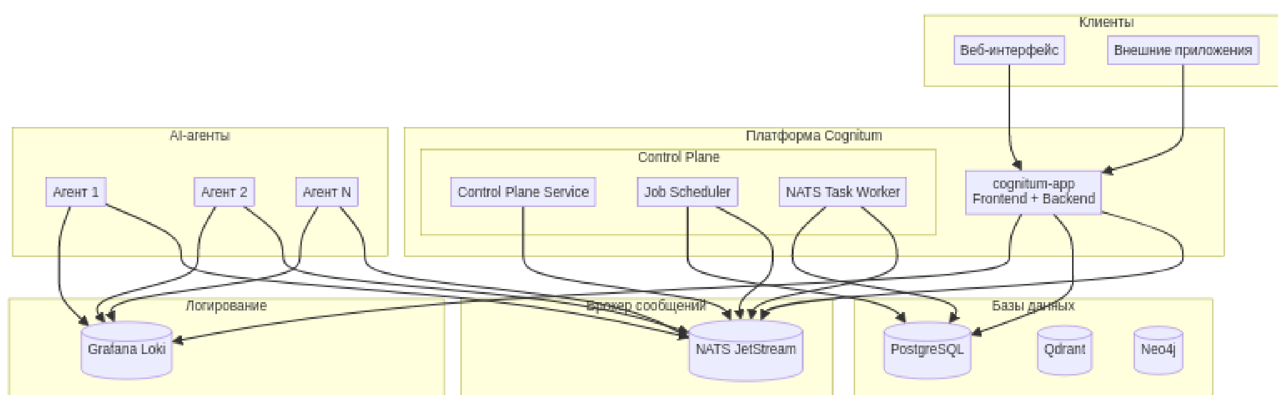
3.1.7 AI-агенты

Контейнеризированные микросервисы, выполняющие задачи.

Характеристики агента: - Изолированный Docker-контейнер - Подключение к платформе через NATS - Регистрация возможностей при запуске - Автоматический мониторинг (healthcheck)

3.2 Модель взаимодействия компонентов

3.2.1 Архитектурная схема



3.2.2 Потоки данных

3.2.2.1 Регистрация агента

1. Агент подключается к NATS при запуске
2. Отправляет запрос регистрации в Control Plane
3. Control Plane сохраняет информацию об агенте в PostgreSQL
4. Агент начинает отправлять heartbeat-сообщения
5. Control Plane обновляет статус агента

3.2.2.2 Выполнение задачи

1. Клиент создаёт задачу через API
2. Бэкенд сохраняет задачу в PostgreSQL (статус: `pending`)
3. Job Scheduler проверяет готовность задачи
4. При готовности публикует задачу в NATS (`job.<type>`)
5. Агент получает задачу из очереди
6. Агент выполняет задачу и отправляет результат
7. NATS Task Worker получает результат
8. Worker обновляет статус задачи в PostgreSQL

3.2.2.3 LLM-вызов через агента

1. Агент формирует запрос к LLM
2. Отправляет RPC-запрос в Control Plane через NATS
3. Control Plane маршрутизирует запрос к провайдеру
4. Получает ответ от провайдера
5. Возвращает результат агенту

3.2.3 Гарантии доставки

Платформа использует JetStream для обеспечения надёжной доставки сообщений:

Тип сообщения	Гарантия	Стрим
Задачи	At-least-once	JOBS (WORK_QUEUE)
Логи	Best-effort	AGENTS (LIMITS)
Управление	At-least-once	CONTROL (WORK_QUEUE)

3.2.4 Отказоустойчивость

- **NATS** — кластерная конфигурация (опционально)
- **PostgreSQL** — репликация и резервное копирование
- **Агенты** — автоматический перезапуск при сбоях
- **Задачи** — повторная обработка при ошибках

3.3 Модель развёртывания

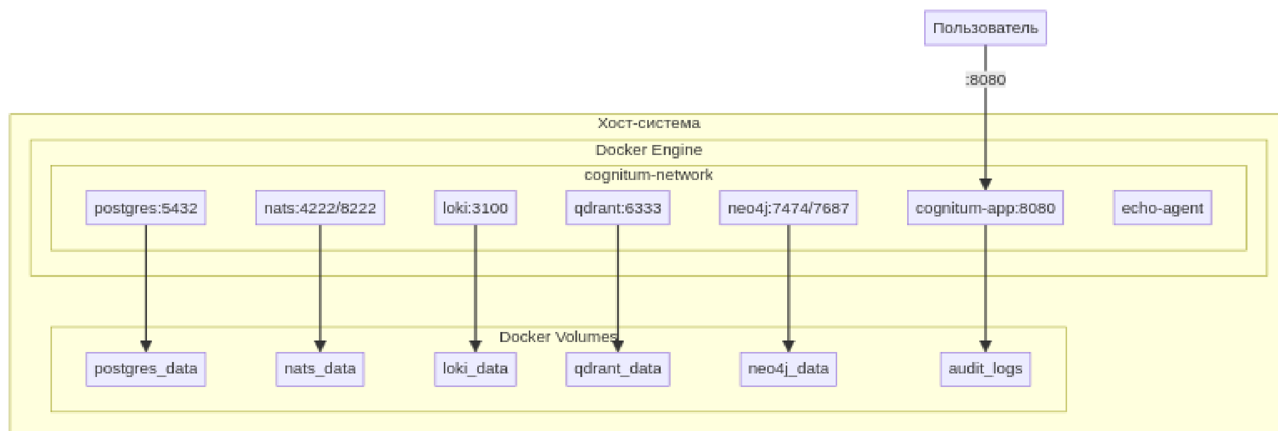
3.3.1 Контейнеризация

Все компоненты платформы поставляются в виде Docker-образов и развёртываются с помощью docker-compose.

Основные образы:

Образ	Описание
`cognitum-app`	Веб-приложение (фронтенд + бэкенд)
`cognitum-echo-agent`	Пример агента
`postgres:15`	База данных PostgreSQL
`nats:2.10-alpine`	Брокер сообщений
`grafana/loki`	Система логирования
`qdrant/qdrant`	Векторная БД
`neo4j`	Графовая БД

3.3.2 Схема развёртывания



3.3.3 Сетевая конфигурация

Все контейнеры объединены в единую Docker-сеть `cognitum-network`, что обеспечивает:

- Изоляцию от внешних сетей
- DNS-разрешение имён контейнеров
- Безопасное взаимодействие компонентов

Порты, доступные извне:

Порт	Сервис	Назначение
8080	cognitum-app	Веб-интерфейс и API
4222	NATS	Подключение агентов
8222	NATS	Мониторинг NATS
5432	PostgreSQL	Прямой доступ к БД (опционально)
3100	Loki	API логов
6333	Qdrant	API векторной БД
7474	Neo4j	Веб-консоль Neo4j

3.3.4 Варианты развёртывания

3.3.4.1 Локальная разработка

Минимальная конфигурация для разработки и тестирования:

- Все компоненты на одном хосте
- Один экземпляр каждого сервиса
- Локальные Docker volumes

3.3.4.2 Production (одиночный сервер)

Конфигурация для небольших нагрузок:

- Все компоненты на одном сервере
- Внешнее хранилище для volumes
- Настроенное резервное копирование

3.3.4.3 Production (распределённая)

Конфигурация для высоких нагрузок:

- Отдельные серверы для баз данных
- Кластер NATS
- Балансировка нагрузки
- Репликация PostgreSQL

3.3.5 Системные требования

Минимальные (разработка): - 4 CPU ядра - 8 GB RAM - 20 GB SSD

Рекомендуемые (production): - 8+ CPU ядер - 16+ GB RAM - 100+ GB SSD - Сетевое хранилище для данных

4 Веб-интерфейс управления

Платформа Cognitum предоставляет современный веб-интерфейс для управления всеми аспектами работы системы.

Веб-интерфейс доступен по адресу, указанному при развёртывании (по умолчанию: `http://localhost:8080`).

4.1 Основные разделы интерфейса

Раздел	Описание
Dashboard	Панель мониторинга с обзором состояния системы
Agents	Управление AI-агентами
LLM APIs	Настройка LLM-провайдеров
Queues	Мониторинг очередей сообщений
Storages	Управление хранилищами
Logs	Просмотр логов агентов
Chat	Тестирование моделей
Admin	Администрирование (пользователи, токены)

4.2 Навигация

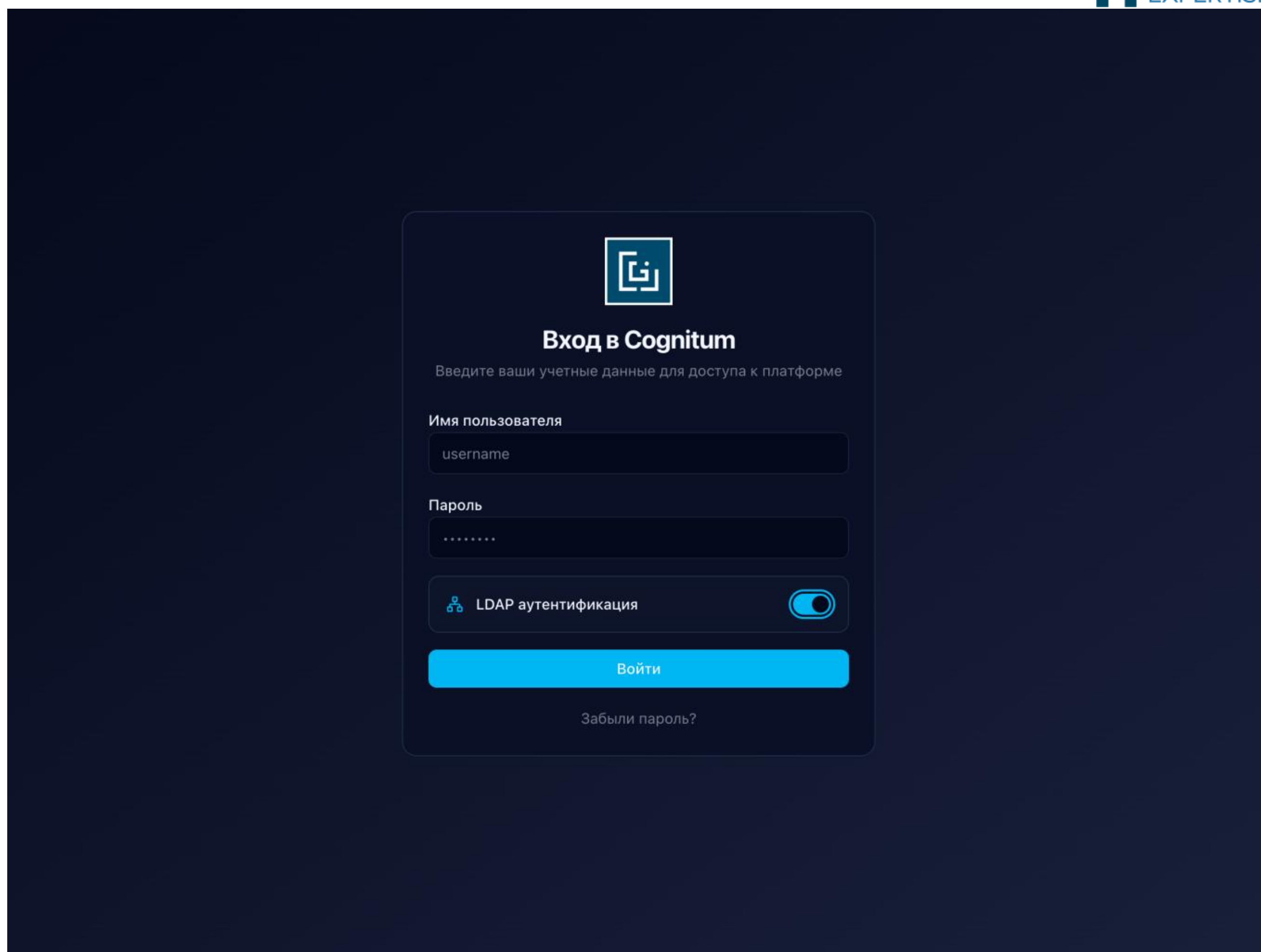
Основное меню расположено в левой части экрана и обеспечивает быстрый доступ ко всем разделам платформы.

В верхней части экрана отображается: - Текущий пользователь - Кнопка выхода из системы - Переключение темы (светлая/тёмная)

4.3 Аутентификация и авторизация

4.3.1 Вход в систему

При первом обращении к веб-интерфейсу пользователь перенаправляется на страницу входа.



Страница входа

Поля формы входа: - **Username** — имя пользователя или email - **Password** — пароль - **Remember me** — сохранить сессию

4.3.2 Учётные данные по умолчанию

При первом запуске платформы автоматически создаётся учётная запись администратора:

- **Логин:** `admin`
- **Пароль:** `admin`

Важно: Обязательно смените пароль администратора после первого входа!

4.3.3 Методы аутентификации

Платформа поддерживает несколько методов аутентификации:

4.3.3.1 Локальная аутентификация

Пользователи создаются в самой платформе. Пароли хранятся в зашифрованном виде в базе данных.

4.3.3.2 LDAP/Active Directory

При включённой LDAP-интеграции пользователи могут входить с корпоративными учётными данными.

Настройка LDAP: - `LDAP_ENABLED=true` - `LDAP_SERVER` — URL сервера LDAP - `LDAP_BASE_DN` — базовый DN для поиска - `LDAP_USER_DN_TEMPLATE` — шаблон DN пользователя

4.3.4 JWT-токены

После успешной аутентификации создаётся JWT-токен:

- Срок действия настраивается через `JWT_EXPIRATION_HOURS`
- По умолчанию: 24 часа
- При истечении токена требуется повторный вход

4.3.5 Восстановление пароля

Если настроен SMTP-сервер, доступна функция восстановления пароля:

1. Нажмите «Forgot password?» на странице входа
2. Введите email, связанный с учётной записью
3. Получите письмо со ссылкой для сброса пароля
4. Установите новый пароль

4.3.6 Роли пользователей

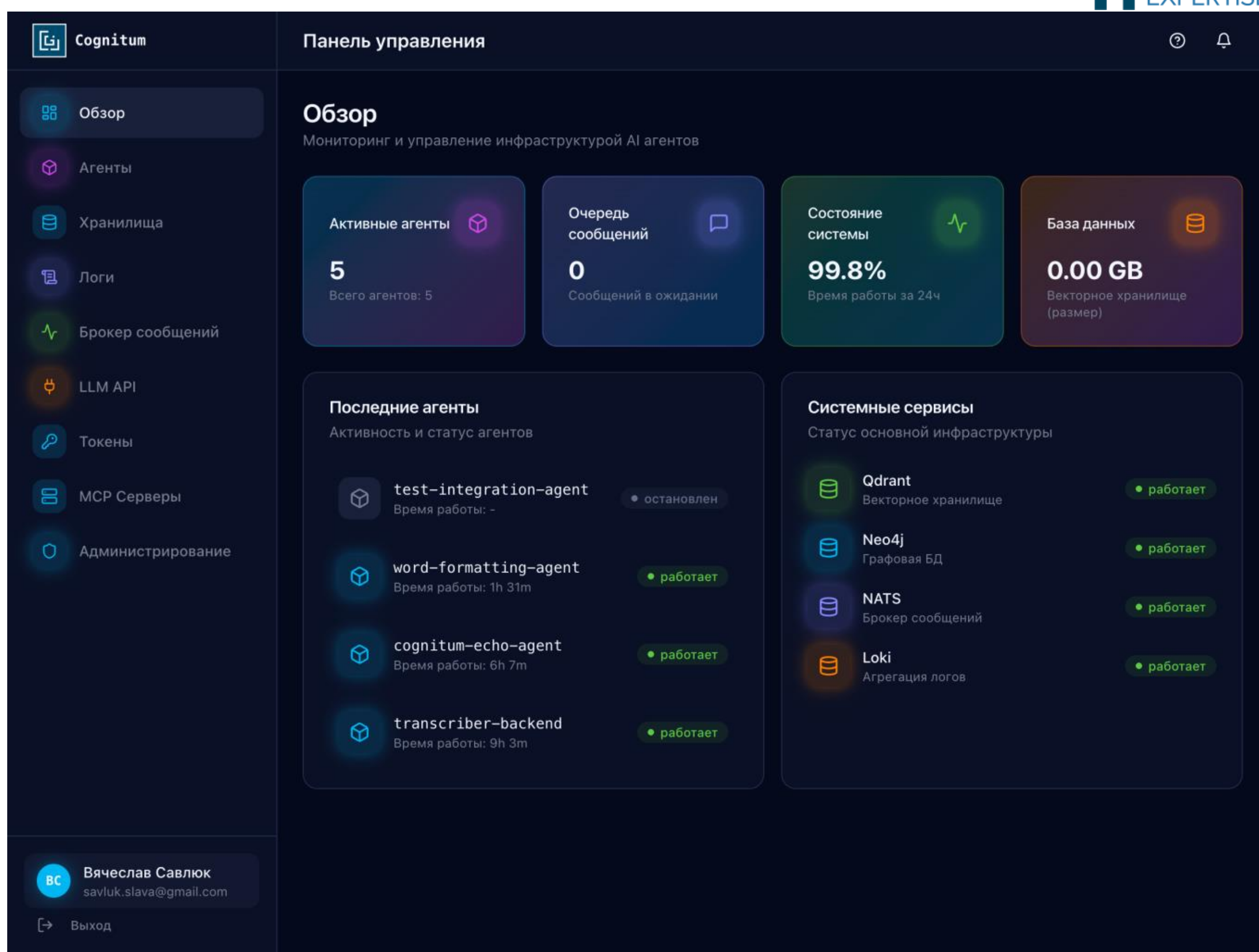
Роль	Права
admin	Полный доступ ко всем функциям
user	Просмотр агентов, логов, использование API

4.3.7 Безопасность сессий

- Сессии привязаны к IP-адресу (опционально)
- Автоматический выход при неактивности
- Возможность принудительного завершения всех сессий

4.4 Панель мониторинга (Dashboard)

Dashboard — главная страница платформы, отображающая сводную информацию о состоянии системы.



Панель мониторинга

4.4.1 Обзор состояния

На панели отображаются ключевые метрики:

4.4.1.1 Агенты

- **Всего агентов** — общее количество зарегистрированных агентов
- **Online** — агенты в активном состоянии
- **Offline** — недоступные агенты

4.4.1.2 Задачи

- **Pending** — задачи в очереди на выполнение
- **Running** — выполняющиеся задачи
- **Completed** — успешно завершённые (за период)
- **Failed** — завершённые с ошибкой (за период)

4.4.1.3 LLM-провайдеры

- **Активных провайдеров** — количество настроенных провайдеров
- **Доступных моделей** — общее число моделей

4.4.2 Виджеты

4.4.2.1 График активности задач

Временной график, показывающий: - Количество созданных задач - Количество завершённых задач - Количество ошибок

Доступные периоды: 1 час, 24 часа, 7 дней.

4.4.2.2 Список последних событий

Лента событий системы: - Регистрация/отключение агентов - Завершение задач - Ошибки системы

4.4.2.3 Состояние компонентов

Индикаторы доступности: - NATS — подключение к брокеру сообщений - PostgreSQL — состояние базы данных - Loki — система логирования

4.4.3 Обновление данных

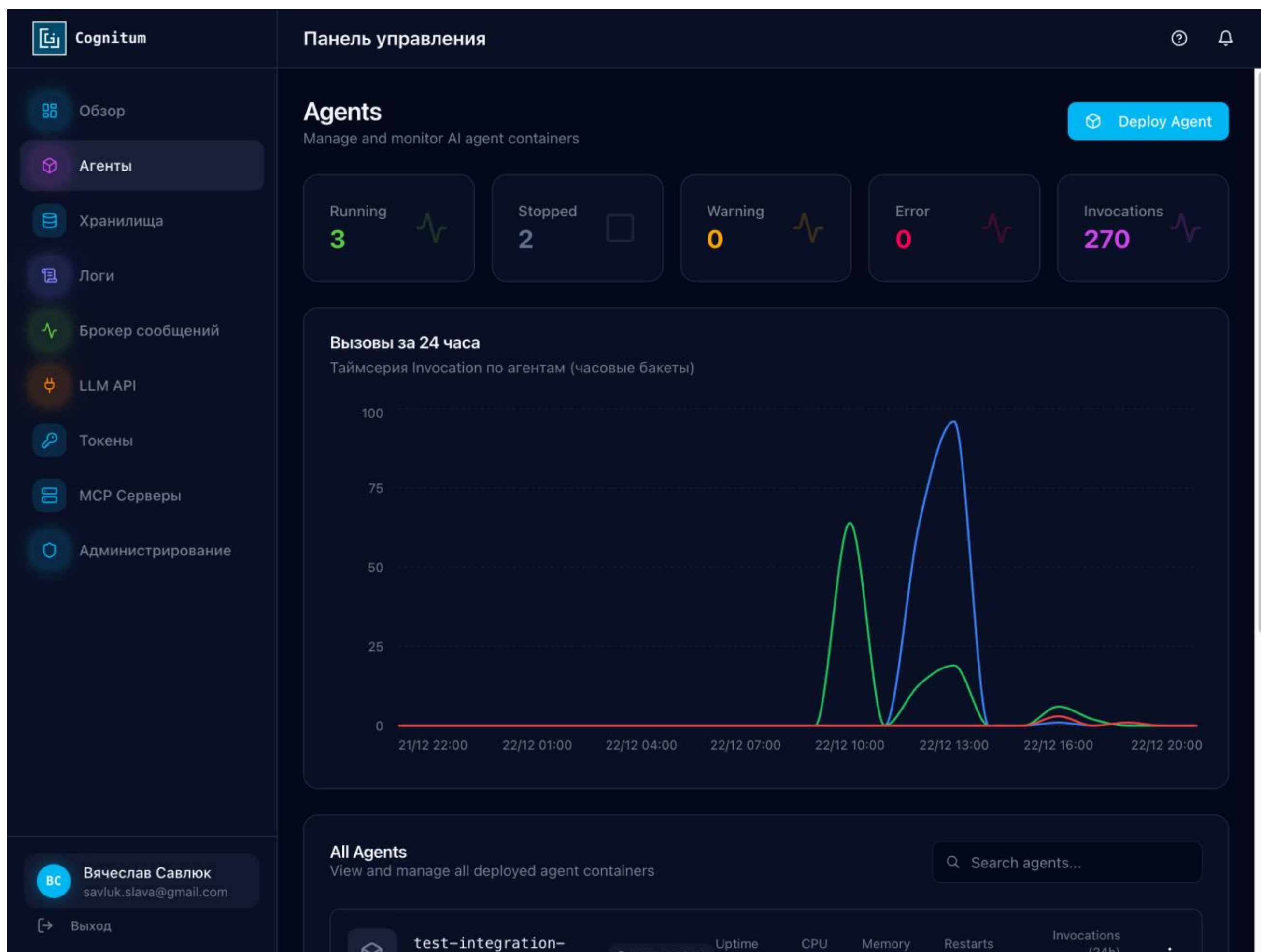
- Автоматическое обновление каждые 30 секунд
- Ручное обновление по кнопке «Refresh»

4.4.4 Быстрые действия

С Dashboard доступны быстрые переходы: - К списку агентов - К очередям задач - К настройкам LLM

4.5 Управление агентами

Раздел **Agents** предоставляет полную информацию о зарегистрированных AI-агентах.



Управление агентами

4.5.1 Список агентов

Таблица со списком всех агентов содержит:

Колонка	Описание
Name	Уникальное имя агента
Status	Текущий статус (online/offline)
Description	Описание агента
Job Types	Поддерживаемые типы задач
Last Seen	Время последнего heartbeat
Actions	Доступные действия

4.5.2 Статусы агентов

- **Online** (зелёный) — агент активен и отвечает на healthcheck
- **Offline** (красный) — агент недоступен более 30 секунд
- **Starting** (жёлтый) — агент в процессе запуска

4.5.3 Карточка агента

При клике на агента открывается детальная информация:

4.5.3.1 Основная информация

- Имя и описание
- Время запуска
- Версия SDK

4.5.3.2 Конфигурация

- Настройки агента (settings)
- Схема настроек (если определена)
- Форма редактирования настроек

4.5.3.3 Возможности

- **Job Types** — типы задач, которые обрабатывает агент
- **Chat Models** — чат-модели, предоставляемые агентом

4.5.3.4 Метрики

- Количество обработанных задач
- Количество LLM-вызовов
- Средняя длительность обработки

4.5.3.5 Логи

Встроенный просмотр логов агента с фильтрацией по уровню.

4.5.4 Настройка агента

Если агент определил `settings_schema`, администратор может настроить его параметры:

1. Откройте карточку агента
2. Перейдите на вкладку «Settings»
3. Заполните форму настроек
4. Нажмите «Save»

Агент получит обновлённые настройки при следующем запросе конфигурации.

4.5.5 Типы настроек

Тип	Описание	Пример
`string`	Текстовое поле	API-ключ
`number`	Числовое поле	Таймаут
`boolean`	Переключатель	Включить кэш
`select`	Выбор из списка	Уровень логирования
`storage`	Выбор хранилища	Векторная БД
`llm`	Выбор LLM-модели	Модель для генерации

4.5.6 Доступ агентов к ресурсам

Администратор может ограничить доступ агентов к: - LLM-провайдерам - Хранилищам - Другим агентам

Настройка выполняется через форму редактирования агента.

4.6 Управление LLM-провайдерами

Раздел **LLM APIs** позволяет настраивать подключения к различным провайдерам языковых моделей.

LLM API

4.6.1 Поддерживаемые провайдеры

Провайдер	Описание

Провайдер	Описание
OpenAI	Официальный API OpenAI (GPT-4, GPT-3.5)
Azure OpenAI	OpenAI через Microsoft Azure
Ollama	Локальные модели (Llama, Mistral, Qwen)
OpenAI-compatible	Любой API, совместимый с OpenAI

4.6.2 Список провайдеров

Таблица содержит:

Колонка	Описание
Name	Название провайдера
Type	Тип (OpenAI, Azure, Ollama)
Base URL	Базовый URL API
Models	Количество моделей
Status	Статус подключения

4.6.3 Добавление провайдера

1. Нажмите кнопку «Add Provider»
2. Выберите тип провайдера
3. Заполните параметры подключения
4. Нажмите «Save»

4.6.3.1 Параметры OpenAI

- **Name** — название для отображения
- **API Key** — ключ API OpenAI
- **Organization ID** (опционально) — ID организации
- **Base URL** (опционально) — для прокси

4.6.3.2 Параметры Azure OpenAI

- **Name** — название для отображения
- **API Key** — ключ Azure
- **Endpoint** — URL endpoint Azure
- **API Version** — версия API (например, `2024-02-15-preview`)
- **Deployment Name** — имя deployment

4.6.3.3 Параметры Ollama

- **Name** — название для отображения
- **Base URL** — URL сервера Ollama (например, `http://ollama:11434`)

4.6.4 Управление моделями

После добавления провайдера платформа автоматически получает список доступных моделей.

4.6.4.1 Информация о модели

- **Internal ID** — внутренний идентификатор для API
- **Display Name** — отображаемое название
- **Is Embedding** — модель для эмбеддингов
- **Is Multimodal** — поддержка изображений
- **Context Window** — размер контекста

4.6.4.2 Настройка модели

Для каждой модели можно настроить: - Отображаемое название - Доступность для агентов - Ограничения использования

4.6.5 Тестирование подключения

1. Откройте карточку провайдера
2. Нажмите «Test Connection»
3. Система проверит доступность API

4.6.6 Статусы провайдеров

- **Active** (зелёный) — провайдер доступен
- **Error** (красный) — ошибка подключения
- **Disabled** (серый) — провайдер отключён

4.6.7 Мониторинг использования

Для каждого провайдера доступна статистика: - Количество запросов - Использованные токены - Средняя задержка ответа

4.7 Работа с очередями задач (Queues)

Раздел **Queues** предоставляет мониторинг JetStream стримов и очередей задач.

Брокер сообщений

4.7.1 Системные стримы

Платформа использует три основных JetStream стрима:

4.7.1.1 AGENTS

Стрим для событий агентов.

Параметр	Значение
Subjects	agent.>
Retention	LIMITS
Max Age	7 дней
Max Messages	1,000,000

Типы сообщений: - agent.<name>.logs — логи агента - agent.<name>.metrics — метрики - agent.<name>.healthcheck — heartbeat - agent.<name>.events — события

4.7.1.2 JOBS

Стрим для задач.

Параметр	Значение
Subjects	job.>
Retention	WORK_QUEUE
Max Age	24 часа
Max Messages	100,000

Типы сообщений: - job.<type> — задачи для агентов - job.<id>.result — результаты выполнения

4.7.1.3 CONTROL

Стрим для управления.

Параметр	Значение
Subjects	control.>
Retention	WORK_QUEUE
Max Age	1 час
Max Messages	10,000

Типы сообщений: - control.llm.invoke — запросы к LLM - control.config.request — запросы конфигурации - control.agent.register — регистрация агентов

4.7.2 Мониторинг стримов

Для каждого стрима отображается:

- **Messages** — количество сообщений в стриме
- **Bytes** — размер данных
- **Consumers** — количество подписчиков
- **First/Last Seq** — номера сообщений

4.7.3 Consumers

Список подписчиков (consumers) стрима:

Колонка	Описание

Колонка	Описание
Name	Имя consumer (durable)
Pending	Сообщения в обработке
Ack Pending	Ожидают подтверждения
Redelivered	Повторно доставленные

4.7.4 Управление задачами

4.7.4.1 Просмотр задач

Таблица задач с фильтрами: - По статусу (pending, running, completed, failed) - По типу задачи - По агенту - По времени создания

4.7.4.2 Детали задачи

При выборе задачи отображается: - ID и тип задачи - Статус и временные метки - Тело задачи (body) - Результат выполнения - Ошибки (если есть)

4.7.4.3 Действия с задачами

- **Retry** — повторное выполнение неудачной задачи
- **Cancel** — отмена задачи в очереди
- **View Result** — просмотр результата

4.7.5 Очистка очередей

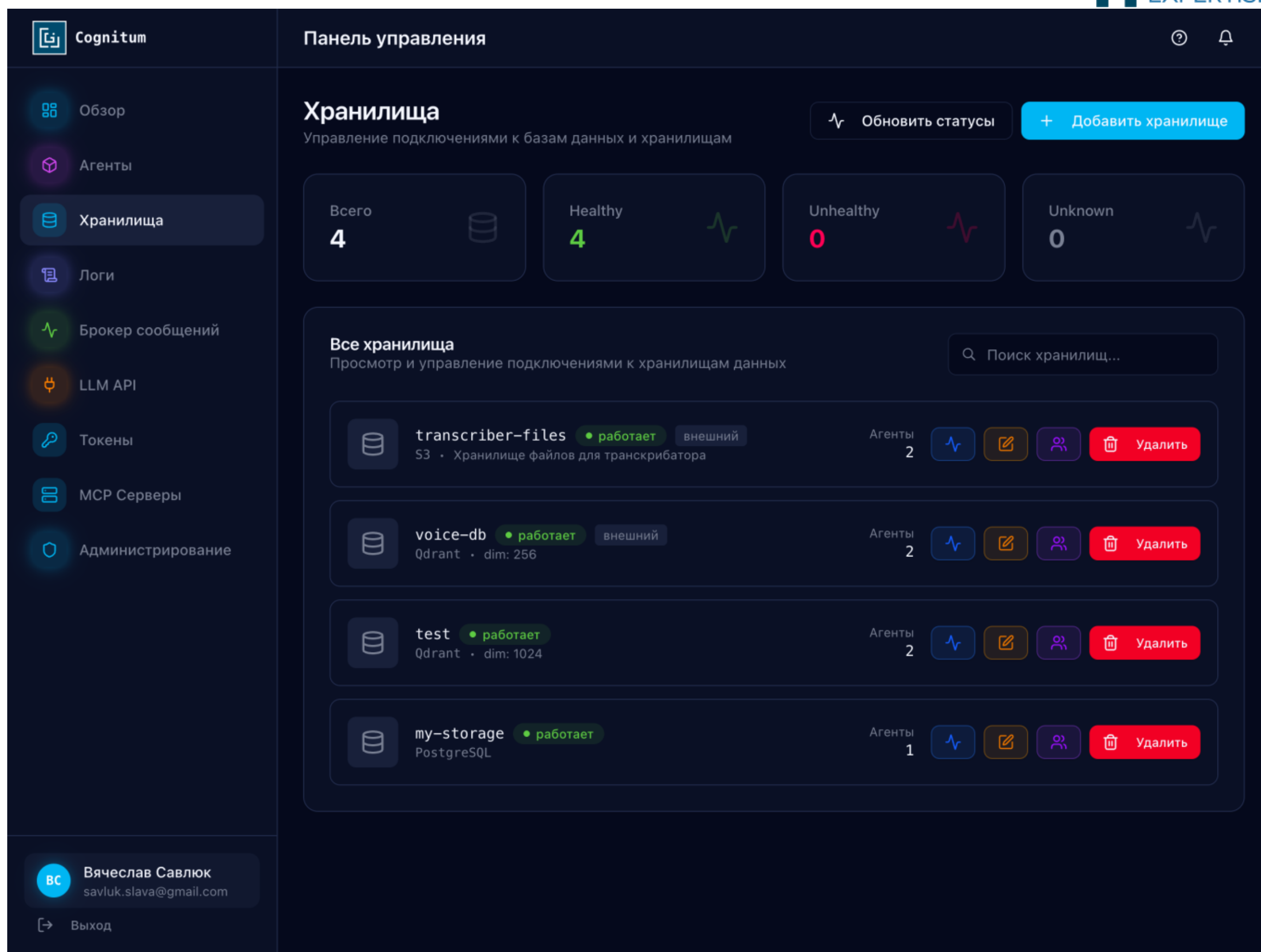
Администратор может очистить очереди:

1. Выберите стрим
2. Нажмите «Purge»
3. Подтвердите действие

> **Внимание:** Очистка удаляет все сообщения без возможности восстановления!

4.8 Управление хранилищами (Storages)

Раздел **Storages** позволяет настраивать внешние хранилища данных для использования агентами.



Хранилища

4.8.1 Поддерживаемые типы хранилищ

Тип	Описание	Применение
PostgreSQL	Реляционная БД	Структурированные данные
Qdrant	Векторная БД	Эмбеддинги, семантический поиск
Neo4j	Графовая БД	Связи, knowledge graphs
S3	Объектное хранилище	Файлы, документы
SMB	Сетевая папка	Корпоративные файлы

4.8.2 Список хранилищ

Таблица содержит:

Колонка	Описание
Name	Уникальное имя хранилища
Type	Тип хранилища
Status	Статус подключения
Agents	Агенты с доступом

4.8.3 Добавление хранилища

1. Нажмите «Add Storage»
2. Выберите тип хранилища
3. Заполните параметры подключения

4. Настройте доступ агентов
5. Нажмите «Save»

4.8.3.1 Параметры PostgreSQL

```
Host: hostname
Port: 5432
Database: dbname
Username: user
Password: ****
SSL Mode: prefer/require/disable
```

4.8.3.2 Параметры Qdrant

```
URL: http://qdrant:6333
API Key: (опционально)
```

4.8.3.3 Параметры Neo4j

```
URI: bolt://neo4j:7687
Username: neo4j
Password: ****
Database: neo4j
```

4.8.3.4 Параметры S3

```
Endpoint: https://s3.amazonaws.com
Access Key: ****
Secret Key: ****
Bucket: bucket-name
Region: us-east-1
```

4.8.3.5 Параметры SMB

```
Server: //server/share
Username: domain\\user
Password: ****
```

4.8.4 Тестирование подключения

1. Откройте карточку хранилища
2. Нажмите «Test Connection»
3. Результат отобразится в уведомлении

4.8.5 Доступ агентов

Для каждого хранилища настраивается список агентов с доступом:

1. Откройте редактирование хранилища
2. В секции «Agent Access» выберите агентов
3. Сохраните изменения

Агент может получить данные подключения через SDK:

```
storages = await agent.get_storages()
```

4.8.6 Безопасность

- Пароли хранятся в зашифрованном виде
- Данные подключения передаются агентам по защищённому каналу
- Доступ ограничен списком разрешённых агентов

4.9 Просмотр логов

Раздел **Logs** предоставляет централизованный доступ к логам всех агентов.

Логи

4.9.1 Источники логов

Логи собираются из двух источников:

1. **NATS JetStream** — логи, отправленные агентами через SDK
2. **Grafana Loki** — логи Docker-контейнеров

4.9.2 Интерфейс просмотра

4.9.2.1 Фильтры

Фильтр	Описание
Agent	Выбор агента
Level	Уровень логирования
Time Range	Временной диапазон
Search	Текстовый поиск

4.9.2.2 Уровни логирования

Уровень	Цвет	Описание
DEBUG	Серый	Отладочная информация
INFO	Синий	Информационные сообщения
WARNING	Жёлтый	Предупреждения
ERROR	Красный	Ошибки

4.9.2.3 Формат записи

Каждая запись лога содержит: - Временная метка - Уровень - Имя агента - ID задачи (если применимо) - Сообщение - Дополнительные поля (JSON)

4.9.3 Просмотр логов агента

1. Перейдите в раздел **Logs**
2. Выберите агента в фильтре
3. Установите временной диапазон
4. Примените фильтры

4.9.4 Поиск по логам

Поддерживается полнотекстовый поиск:

- По тексту сообщения
- По значениям дополнительных полей
- По ID задачи

4.9.5 Live-режим

Включите «Live tail» для просмотра логов в реальном времени:

1. Нажмите кнопку «Live»
2. Новые записи будут появляться автоматически
3. Для остановки нажмите «Pause»

4.9.6 Экспорт логов

Логи можно экспортировать:

1. Примените нужные фильтры
2. Нажмите «Export»
3. Выберите формат (JSON, CSV)
4. Скачайте файл

4.9.7 Контекст задачи

При клике на ID задачи в логе:

1. Открывается модальное окно с деталями задачи
2. Отображаются все логи, связанные с этой задачей
3. Доступен переход к результату задачи

4.9.8 Хранение логов

- Логи в NATS хранятся 7 дней
- Логи в Loki хранятся согласно настройкам retention
- Старые логи автоматически удаляются

4.10 Тестирование моделей (Chat)

Раздел **Chat** предоставляет интерфейс для тестирования LLM-моделей и чат-моделей агентов.

4.10.1 Интерфейс чата

4.10.1.1 Выбор модели

В верхней части экрана расположен селектор моделей:

- **LLM Models** — модели от провайдеров (OpenAI, Ollama)
- **Agent Models** — чат-модели, предоставляемые агентами

4.10.1.2 Область сообщений

Центральная область отображает историю диалога:

- Сообщения пользователя (справа)
- Ответы модели (слева)
- Системные сообщения (по центру)

4.10.1.3 Поле ввода

В нижней части:

- Текстовое поле для ввода сообщения
- Кнопка отправки
- Индикатор загрузки (при ожидании ответа)

4.10.2 Режимы работы

4.10.2.1 Обычный режим

Модель генерирует полный ответ, который отображается после завершения.

4.10.2.2 Streaming режим

Ответ отображается по мере генерации (посимвольно).

> Streaming поддерживается не всеми моделями.

4.10.3 Параметры генерации

Дополнительные параметры доступны в боковой панели:

Параметр	Описание	Диапазон
Temperature	Креативность	0.0 - 2.0
Max Tokens	Максимальная длина	1 - 128000
Top P	Nucleus sampling	0.0 - 1.0

4.10.4 Системный промпт

Можно задать системный промпт для контекста:

1. Нажмите «System Prompt»
2. Введите инструкции для модели
3. Промпт будет добавлен к каждому запросу

4.10.5 История диалогов

- История сохраняется в localStorage браузера
- Доступна кнопка «Clear Chat» для очистки
- Каждый диалог привязан к выбранной модели

4.10.6 Тестирование агентов

При выборе модели агента:

1. Запрос отправляется агенту через платформу

2. Агент обрабатывает запрос и возвращает ответ
3. Поддерживается сохранение контекста диалога

4.10.7 Мультимодальность

Если модель поддерживает изображения (`is_multimodal: true`):

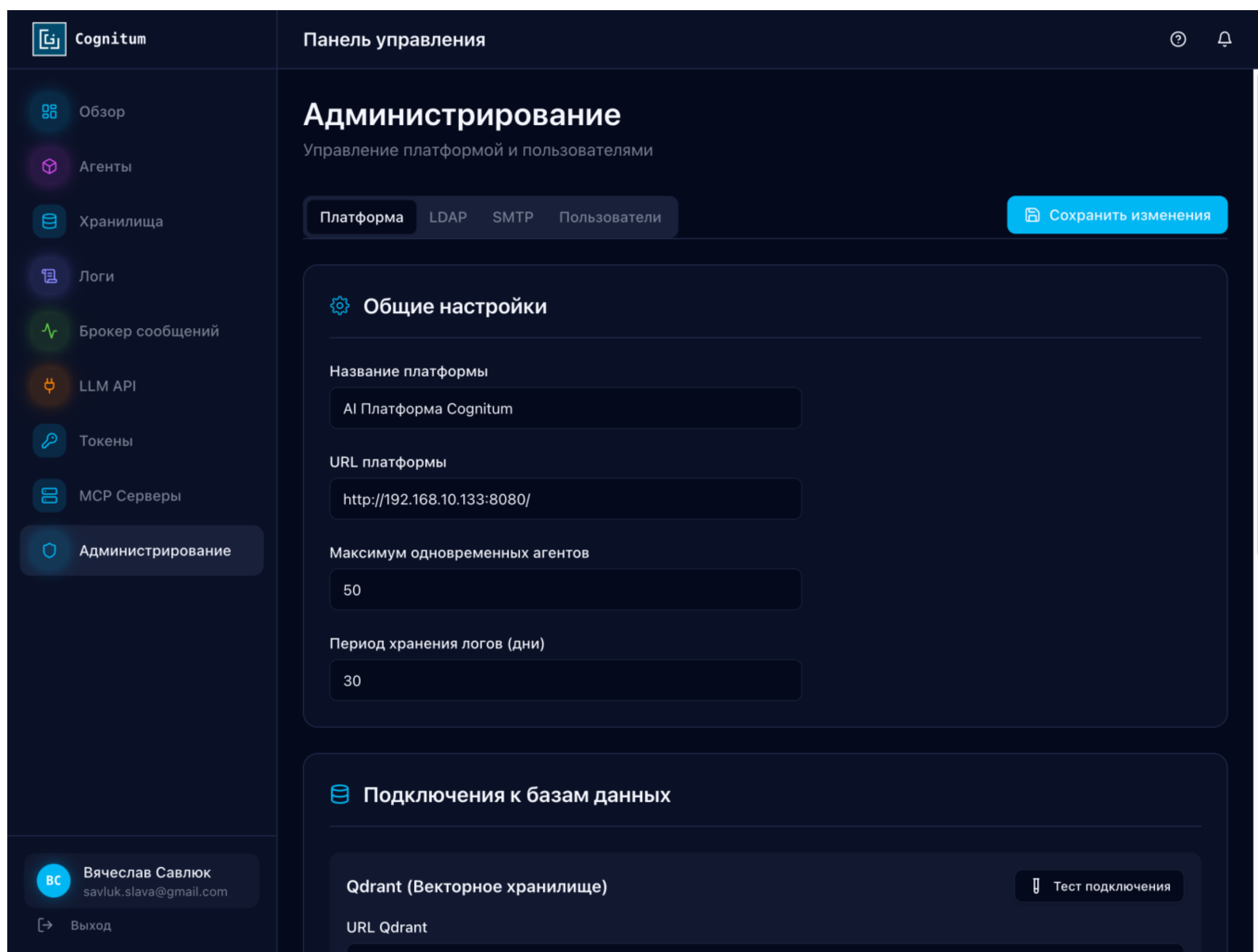
1. Нажмите кнопку «Attach Image»
2. Выберите файл изображения
3. Изображение будет отправлено вместе с текстом

4.10.8 Копирование ответов

- Кнопка копирования для каждого сообщения
- Поддержка Markdown-форматирования
- Подсветка кода в ответах

4.11 Администрирование

Раздел **Admin** доступен пользователям с ролью администратора.



Администрирование

4.11.1 Управление пользователями

4.11.1.1 Список пользователей

Таблица содержит:

Колонка	Описание
Username	Имя пользователя
Email	Электронная почта
Role	Роль (admin/user)
Status	Статус (active/disabled)
Last Login	Последний вход

4.11.1.2 Создание пользователя

1. Нажмите «Add User»
2. Заполните форму:
   - Username (обязательно)
   - Email (обязательно)
   - Password
   - Role
3. Нажмите «Create»

4.11.1.3 Редактирование пользователя

1. Выберите пользователя в списке
2. Внесите изменения
3. Нажмите «Save»

4.11.1.4 Сброс пароля

1. Откройте карточку пользователя
2. Нажмите «Reset Password»
3. Введите новый пароль
4. Пользователь получит уведомление (если настроен SMTP)

4.11.1.5 Деактивация пользователя

1. Откройте карточку пользователя
2. Переключите статус на «Disabled»
3. Пользователь не сможет войти в систему

4.11.2 Управление API-токенами

API-токены позволяют внешним приложениям обращаться к API платформы.

4.11.2.1 Список токенов

Колонка	Описание
Name	Название токена
Owner	Владелец
Created	Дата создания
Last Used	Последнее использование
Expires	Срок действия

4.11.2.2 Создание токена

1. Нажмите «Create Token»
2. Введите название
3. Выберите срок действия (или «Never expires»)
4. Нажмите «Create»
5. **Скопируйте токен** — он показывается только один раз!

4.11.2.3 Отзыв токена

1. Найдите токен в списке
2. Нажмите «Revoke»
3. Подтвердите действие

4.11.3 Настройки системы

4.11.3.1 Общие настройки

- **Platform Name** — название инсталляции
- **Default Language** — язык по умолчанию

4.11.3.2 Безопасность

- **Session Timeout** — время жизни сессии
- **Max Login Attempts** — максимум попыток входа
- **Password Policy** — требования к паролям

4.11.3.3 Аудит

- **Enable Audit Log** — включение аудита
- **Retention Days** — срок хранения записей

4.11.4 Просмотр аудит-лога

Аудит фиксирует:

- Вход/выход пользователей
- Изменения конфигурации
- Создание/удаление объектов
- API-запросы

4.11.4.1 Фильтры аудита

- По пользователю
- По типу действия
- По временному диапазону
- По объекту

4.11.4.2 Экспорт аудита

Аудит-лог можно экспортировать для внешнего анализа или архивации.

5 API платформы

Cognitum предоставляет REST API для интеграции с внешними системами.

API доступен по базовому URL: `http://<host>:8080/api/`

5.1 Форматы API

Платформа поддерживает два формата API:

1. **REST API** — для управления платформой и задачами
2. **OpenAI-совместимый API** — для работы с LLM-моделями

5.2 Документация API

Интерактивная документация доступна по адресам:

- Swagger UI: `http://<host>:8080/api/docs`
- ReDoc: `http://<host>:8080/api/redoc`
- OpenAPI JSON: `http://<host>:8080/api/openapi.json`

5.3 OpenAI-совместимый API

Платформа предоставляет API, совместимый с OpenAI, что позволяет использовать модели агентов из любых приложений, поддерживающих стандарт OpenAI.

5.3.1 Базовый URL

```
http://<host>:8080/api/v1/
```

5.3.2 Аутентификация

Используйте API-токен в заголовке `Authorization`:

```
Authorization: Bearer <api_token>
```

5.3.3 Endpoints

5.3.3.1 GET /v1/models

Список доступных моделей.

Ответ:

```
{
  "object": "list",
  "data": [
    {
      "id": "gpt-4o-mini",
      "object": "model",
      "created": 1700000000,
      "owned_by": "openai"
    },
    {
      "id": "agent-my-agent-chat",
      "object": "model",
      "created": 1700000000,
      "owned_by": "agent:my-agent"
    }
  ]
}
```

IT EXPERTISE

5.3.3.2 POST /v1/chat/completions

Генерация ответа модели.

Запрос:

```
{
  "model": "gpt-4o-mini",
  "messages": [
    {"role": "system", "content": "You are a helpful assistant."},
    {"role": "user", "content": "Hello!"}
  ],
  "temperature": 0.7,
  "max_tokens": 1000,
  "stream": false
}
```

Ответ:

```
{
  "id": "chatcmpl-abc123",
  "object": "chat.completion",
  "created": 1700000000,
  "model": "gpt-4o-mini",
  "choices": [
    {
      "index": 0,
      "message": {
        "role": "assistant",
        "content": "Hello! How can I help you today?"
      },
      "finish_reason": "stop"
    }
  ],
  "usage": {
    "prompt_tokens": 20,
    "completion_tokens": 10,
    "total_tokens": 30
  }
}
```

5.3.3.3 POST /v1/embeddings

Создание векторных представлений (эмбеддингов).

Запрос:

```
{
  "model": "text-embedding-ada-002",
  "input": "The quick brown fox"
}
```

Ответ:

```
{
  "object": "list",
  "data": [
    {
      "object": "embedding",
      "index": 0,
      "embedding": [0.0023, -0.0012, ...]
    }
  ],
  "model": "text-embedding-ada-002",
  "usage": {
    "prompt_tokens": 5,
    "total_tokens": 5
  }
}
```

5.3.4 Streaming

Для потоковой генерации используйте `"stream": true`:

```
{
  "model": "gpt-4o-mini",
  "messages": [{"role": "user", "content": "Hello!"}],
  "stream": true
}
```

Ответ приходит в формате Server-Sent Events (SSE):

```
data: {"id":"chatcmpl-abc","choices":[{"delta":{"content":"Hello"}}]}

data: {"id":"chatcmpl-abc","choices":[{"delta":{"content":"!"}}]}

data: [DONE]
```

5.3.5 Использование с OpenAI SDK

```
from openai import OpenAI

client = OpenAI(
    base_url="http://localhost:8080/api/v1",
    api_key="your_api_token"
)

response = client.chat.completions.create(
    model="gpt-4o-mini",
    messages=[{"role": "user", "content": "Hello!"}]
)

print(response.choices[0].message.content)
```

5.3.6 Модели агентов

Чат-модели агентов доступны с префиксом `agent-`:

```
agent-{agent_name}-{model_internal_id}
```

Например: `agent-my-agent-rag-chat`

5.4 REST API для задач (Jobs)

API для работы с системой задач.

5.4.1 Endpoints

5.4.1.1 POST /api/jobs

Создание новой задачи.

Запрос:

```
{
  "job_type": "process-document",
  "body": {
    "document_url": "https://example.com/doc.pdf",
    "options": {"extract_images": true}
  },
  "prerequisites": ["uuid-of-parent-job"],
  "not_before": "2024-01-01T12:00:00Z",
  "ttl_seconds": 3600
}
```

Параметры:

Поле	Тип	Описание

Поле	Тип	Описание
job_type	string	Тип задачи (должен быть зарегистрирован агентом)
body	object	Тело задачи (параметры)
prerequisites	array	ID задач-зависимостей
not_before	datetime	Время отложенного выполнения
ttl_seconds	integer	Время жизни задачи

Ответ (201 Created):

```
{
  "job_id": "550e8400-e29b-41d4-a716-446655440000",
  "status": "pending",
  "created_at": "2024-01-01T10:00:00Z"
}
```

5.4.1.2 POST /api/jobs/invoke

Синхронное выполнение задачи (ожидание результата).

Запрос:

```
{
  "job_type": "echo-job",
  "body": {"message": "Hello"},
  "timeout": 30
}
```

Ответ:

```
{
  "job_id": "550e8400-e29b-41d4-a716-446655440000",
  "status": "completed",
  "result": {"echo": "Hello"},
  "duration_ms": 150
}
```

5.4.1.3 GET /api/jobs/{job_id}

Получение информации о задаче.

Ответ:

```
{
  "id": "550e8400-e29b-41d4-a716-446655440000",
  "job_type": "process-document",
  "agent_name": "document-processor",
  "status": "completed",
  "body": {...},
  "result": {...},
  "created_at": "2024-01-01T10:00:00Z",
  "started_at": "2024-01-01T10:00:05Z",
  "completed_at": "2024-01-01T10:00:15Z"
}
```

5.4.1.4 GET /api/jobs/{job_id}/result

Получение результата задачи.

Ответ:

```
{
  "job_id": "550e8400-e29b-41d4-a716-446655440000",
  "status": "completed",
  "result": {
    "processed": true,
```

```
    "pages": 10
  }
}
```

5.4.1.5 GET /api/jobs

Список задач с фильтрацией.

Query-параметры:

Параметр	Описание
`status`	Фильтр по статусу
`job_type`	Фильтр по типу
`agent_name`	Фильтр по агенту
`limit`	Количество записей
`offset`	Смещение

Ответ:

```
{
  "items": [...],
  "total": 100,
  "limit": 20,
  "offset": 0
}
```

5.4.1.6 DELETE /api/jobs/{job_id}

Отмена задачи.

Ответ (200 OK):

```
{
  "job_id": "550e8400-e29b-41d4-a716-446655440000",
  "status": "cancelled"
}
```

5.4.2 Статусы задач

Статус	Описание
`pending`	Создана, ожидает обработки
`delayed`	Отложена (not_before или prerequisites)
`queued`	Отправлена в очередь агенту
`running`	Выполняется агентом
`completed`	Успешно завершена
`failed`	Завершена с ошибкой
`expired`	Истёк TTL
`timeout`	Превышен таймаут выполнения
`cancelled`	Отменена

5.4.3 Коды ошибок

Код	Описание
400	Некорректный запрос
401	Не авторизован
404	Задача не найдена
409	Конфликт (задача уже завершена)

Код	Описание
503	Нет доступных агентов для типа задачи

5.5 Аутентификация в API

5.5.1 Методы аутентификации

API поддерживает два метода аутентификации:

1. **JWT-токен** — для веб-интерфейса и краткосрочных сессий
2. **API-токен** — для интеграций и долгосрочного доступа

5.5.2 JWT-аутентификация

5.5.2.1 Получение токена

POST /api/auth/login

```
{
  "username": "user@example.com",
  "password": "password"
}
```

Ответ:

```
{
  "access_token": "eyJhbGciOiJIUzI1NiIs...",
  "token_type": "bearer",
  "expires_in": 86400
}
```

5.5.2.2 Использование токена

```
Authorization: Bearer eyJhbGciOiJIUzI1NiIs...
```

5.5.2.3 Обновление токена

POST /api/auth/refresh

```
Authorization: Bearer <current_token>
```

5.5.3 API-токены

API-токены создаются в веб-интерфейсе (Admin → API Tokens).

5.5.3.1 Формат токена

```
cog_xxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

5.5.3.2 Использование

```
Authorization: Bearer cog_xxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

или в заголовке `X-API-Key`:

```
X-API-Key: cog_xxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

5.5.4 Области действия (Scopes)

Scope	Описание
`read`	Чтение данных
`write`	Создание и изменение
`admin`	Административные функции
`llm`	Доступ к LLM API
`jobs`	Управление задачами

5.5.5 Примеры использования

5.5.5.1 cURL

```
# С JWT-токеном
curl -H "Authorization: Bearer eyJ..." \
     http://localhost:8080/api/agents

# С API-токеном
curl -H "X-API-Key: cog_xxx" \
     http://localhost:8080/api/agents
```

5.5.5.2 Python (requests)

```python
import requests

headers = {
    "Authorization": "Bearer cog_xxx"
}

response = requests.get(
    "http://localhost:8080/api/agents",
    headers=headers
)
```

5.5.5.3 Python (OpenAI SDK)

```python
from openai import OpenAI

client = OpenAI(
    base_url="http://localhost:8080/api/v1",
    api_key="cog_xxx"  # API-токен как api_key
)
```

5.5.6 Ошибки аутентификации

Код	Ошибка	Описание
401	`unauthorized`	Токен отсутствует или недействителен
401	`token_expired`	Срок действия токена истёк
403	`forbidden`	Недостаточно прав
403	`token_revoked`	Токен отозван

5.5.7 Безопасность

- Передавайте токены только по HTTPS
- Не храните токены в открытом виде
- Используйте отдельные токены для разных интеграций
- Регулярно ротируйте токены
- Отзывайте неиспользуемые токены

6 SDK для разработки агентов

Cognitum SDK — библиотека Python для создания AI-агентов, интегрированных с платформой.

SDK обеспечивает:

- Подключение к платформе через NATS
- Регистрацию агента и его возможностей
- Обработку задач из очередей
- Предоставление чат-моделей
- Централизованное логирование
- Вызовы LLM через Control Plane

6.1 Требования

- Python 3.10+
- NATS-сервер (входит в состав платформы)

6.2 Исходный код

SDK поставляется в составе дистрибутива в папке `sdk/`.

6.3 Установка SDK

6.3.1 Из дистрибутива

```
cd sdk
pip install -e .
```

6.3.2 Зависимости

SDK устанавливает следующие зависимости:

- `nats-py` — клиент NATS
- `pydantic` — валидация данных
- `httpx` — HTTP-клиент (для LLM)

6.3.3 Проверка установки

```
from cognitum_agent import Agent, AgentConfig

print("SDK installed successfully!")
```

6.3.4 Минимальный пример

```
import asyncio
from cognitum_agent import Agent, AgentConfig

async def main():
    agent = Agent(AgentConfig(
        name="my-agent",
        nats_url="nats://localhost:4222"
    ))

    async with agent:
        await agent.publish_log("info", "Agent started!")
        # Агент работает...

asyncio.run(main())
```

6.3.5 Переменные окружения

SDK использует следующие переменные:

Переменная	Описание	По умолчанию
NATS_URL	URL NATS-сервера	nats://nats:4222

6.3.6 Запуск в Docker

Пример Dockerfile для агента:

```
FROM python:3.11-slim

WORKDIR /app

# Копирование SDK
COPY sdk/ /sdk/
RUN pip install /sdk/

# Копирование агента
COPY agent/ /app/

CMD ["python", "main.py"]
```

6.4 Конфигурация агента (AgentConfig)

AgentConfig определяет параметры агента.

6.4.1 Основные параметры

```
from cognitum_agent import AgentConfig

config = AgentConfig(
    name="my-agent",                    # Уникальное имя
    description="Описание агента",      # Описание
    nats_url="nats://localhost:4222",   # URL NATS
)
```

6.4.2 Все параметры

Параметр	Тип	По умолчанию	Описание
name	str	**обязательный**	Уникальное имя агента
description	str	None	Описание для UI
nats_url	str	nats://nats:4222	URL NATS-сервера
job_types	list[str]	[]	Типы задач (deprecated)
job_schemas	list[dict]	[]	Схемы задач
settings_schema	list[dict]	[]	Схема настроек
chat_models	list[dict]	[]	Чат-модели
llm_timeout_s	float	30.0	Таймаут LLM
rpc_timeout_s	float	30.0	Таймаут RPC
healthcheck_interval_s	float	2.0	Интервал heartbeat

6.4.3 Схема настроек (settings_schema)

Определяет параметры, редактируемые через UI:

```
settings_schema=[
    {
        "name": "api_key",
        "type": "string",
        "description": "API ключ",
        "required": True
    },
    {
        "name": "temperature",
        "type": "number",
```

```
        "default": 0.7,
        "min": 0,
        "max": 2
    },
    {
        "name": "model",
        "type": "llm",
        "llm_type": "llm",
        "description": "Модель для генерации"
    },
    {
        "name": "storage",
        "type": "storage",
        "storage_type": "qdrant",
        "description": "Векторное хранилище"
    }
]
```

6.4.4 Типы полей настроек

Тип	Описание
`string`	Текстовое поле
`number`	Числовое поле
`boolean`	Переключатель
`select`	Выбор из списка
`storage`	Выбор хранилища
`llm`	Выбор LLM-модели

6.4.5 Схема задач (job_schemas)

```
job_schemas=[
    {
        "job_type": "process-document",
        "description": "Обработка документа",
        "parameters_schema": [
            {"name": "url", "type": "string", "required": True},
            {"name": "format", "type": "select", "options": ["pdf", "docx"]}
        ]
    }
]
```

6.4.6 Получение настроек в коде

```
# Получить значение настройки
api_key = agent.get_setting("api_key")
temperature = agent.get_setting("temperature", default=0.7)
```

6.5 Обработка задач (Job Handlers)

Job handlers — функции для обработки задач из очередей.

6.5.1 Регистрация обработчика

```
from cognitum_agent import Agent, AgentConfig

agent = Agent(AgentConfig(
    name="my-agent",
    job_schemas=[
        {"job_type": "process-data", "description": "Обработка данных"}
    ]
))

@agent.job_handler("process-data")
async def handle_process(job):
    # job.job_id — ID задачи
    # job.job_type — тип задачи
    # job.body — тело задачи (dict)
```

```
    data = job.body.get("data")
    result = process_data(data)

    return {"processed": True, "result": result}
```

6.5.2 Структура JobMessage

```
class JobMessage:
    job_id: str             # UUID задачи
    job_type: str           # Тип задачи
    body: dict              # Тело задачи
    prerequisites: list     # ID зависимостей
    expires_at: datetime    # Время истечения
    ttl_seconds: int        # TTL
```

6.5.3 Параллельная обработка

```
@agent.job_handler("heavy-task", max_parallel=5)
async def handle_heavy(job):
    # До 5 задач обрабатываются одновременно
    pass
```

6.5.4 Отложенное выполнение (Defer)

```
@agent.job_handler("retry-task")
async def handle_retry(job):
    try:
        result = await external_api_call()
        return {"success": True, "result": result}
    except ServiceUnavailable:
        # Отложить задачу на 60 секунд
        job.defer(60)
        return  # Возврат игнорируется при defer
```

6.5.5 Общий обработчик

```
@agent.on_job
async def handle_any_job(job):
    # Вызывается для всех типов задач
    print(f"Received job: {job.job_type}")
    return {"handled": True}
```

6.5.6 Логирование в контексте задачи

SDK автоматически добавляет `job_id` в логи:

```
@agent.job_handler("my-task")
async def handle_task(job):
    await agent.publish_log("info", "Processing started")
    # В логе будет job_id автоматически

    await agent.publish_log("info", "Done", extra={"items": 10})
    return {"success": True}
```

6.5.7 Создание подзадач

```
@agent.job_handler("parent-task")
async def handle_parent(job):
    # Создать подзадачу
    child_id = await agent.submit_job(
        "child-task",
        body={"parent_id": job.job_id}
    )

    # Дождаться результата
    result = await agent.wait_for_job_result(child_id, timeout_s=30)

    return {"child_result": result}
```

6.5.8 Проверка зависимостей

```
@agent.job_handler("dependent-task")
async def handle_dependent(job):
    if job.prerequisites:
```

```
        deps = await agent.check_job_dependencies(job.prerequisites)

        if not all(d.get("completed") for d in deps.values()):
            job.defer(5)  # Подождать зависимости
            return

    # Все зависимости завершены
    return {"success": True}
```

6.6 Чат-обработчики (Chat Handlers)

Chat handlers позволяют агенту предоставлять чат-модели для взаимодействия через API.

6.6.1 Регистрация чат-модели

```
agent = Agent(AgentConfig(
    name="my-agent",
    chat_models=[
        {
            "internal_id": "my-chat",
            "name": "My Chat Model",
            "description": "Чат с RAG",
            "supports_stream": True,
            "supports_reasoning": False,
            "is_multimodal": False
        }
    ]
))
```

6.6.2 Параметры чат-модели

Параметр	Тип	Описание
internal_id	str	Уникальный ID модели
name	str	Отображаемое название
description	str	Описание
supports_stream	bool	Поддержка streaming
supports_reasoning	bool	Поддержка reasoning
is_multimodal	bool	Поддержка изображений

6.6.3 Регистрация обработчика

```
@agent.chat_handler("my-chat")
async def handle_chat(request):
    messages = request.get("messages", [])
    stream = request.get("stream", False)

    # Получить последнее сообщение пользователя
    user_message = messages[-1]["content"]

    # Генерация ответа
    response = await generate_response(user_message)

    return {
        "content": response,
        "usage": {
            "prompt_tokens": 100,
            "completion_tokens": 50,
            "total_tokens": 150
        }
    }
```

6.6.4 Сохранение контекста

```
@agent.chat_handler("my-chat", store_context=True)
async def handle_chat_with_context(request):
    # Контекст автоматически сохраняется в платформе
    messages = request.get("messages", [])
```

```
    # messages содержит всю историю диалога
    pass
```

6.6.5 Streaming-ответы

```
@agent.chat_handler("my-chat")
async def handle_chat_stream(request):
    if request.get("stream"):
        async def generate():
            for chunk in ["Hello", " ", "World", "!"]:
                yield {"content": chunk}

        return generate()
    else:
        return {"content": "Hello World!"}
```

6.6.6 Использование LLM

```
@agent.chat_handler("my-chat")
async def handle_chat_llm(request):
    messages = request.get("messages", [])

    # Вызов LLM через платформу
    response = await agent.llm_chat(
        messages=messages,
        model="gpt-4o-mini",
        temperature=0.7
    )

    return {
        "content": response["choices"][0]["message"]["content"],
        "usage": response.get("usage", {})
    }
```

6.6.7 RAG-интеграция

```
@agent.chat_handler("rag-chat")
async def handle_rag(request):
    messages = request.get("messages", [])
    user_query = messages[-1]["content"]

    # Поиск релевантных документов
    docs = await search_documents(user_query)

    # Формирование контекста
    context = "\n".join([d["content"] for d in docs])

    # Генерация с контекстом
    system_prompt = f"Используй контекст:\n{context}"

    response = await agent.llm_chat(
        messages=[
            {"role": "system", "content": system_prompt},
            *messages
        ],
        model="gpt-4o-mini"
    )

    return {"content": response["choices"][0]["message"]["content"]}
```

6.6.8 Доступ через API

Чат-модель агента доступна по адресу:

```
POST /api/v1/chat/completions
{
  "model": "agent-my-agent-my-chat",
  "messages": [...]
}
```

6.7 LLM-вызовы через платформу

SDK предоставляет методы для работы с LLM через Control Plane.

6.7.1 Chat Completions

```
response = await agent.llm_chat(
    messages=[
        {"role": "system", "content": "You are a helpful assistant."},
        {"role": "user", "content": "Hello!"}
    ],
    model="gpt-4o-mini",
    temperature=0.7,
    max_tokens=1000
)

# Ответ
content = response["choices"][0]["message"]["content"]
usage = response["usage"]
```

6.7.2 Параметры llm_chat

Параметр	Тип	Описание
`messages`	list	История сообщений
`model`	str	ID модели
`temperature`	float	Креативность (0-2)
`max_tokens`	int	Максимум токенов
`top_p`	float	Nucleus sampling
`stop`	list	Стоп-последовательности

6.7.3 Embeddings

```
result = await agent.ask_embedding(
    text="Текст для векторизации",
    model="text-embedding-ada-002"
)

vector = result["data"][0]["embedding"]
dimension = len(vector)
```

6.7.4 Batch Embeddings

```
texts = ["Текст 1", "Текст 2", "Текст 3"]

result = await agent.ask_embedding(
    text=texts,
    model="text-embedding-ada-002"
)

vectors = [item["embedding"] for item in result["data"]]
```

6.7.5 Мультимодальные запросы

```
import base64

# Кодирование изображения
with open("image.jpg", "rb") as f:
    image_b64 = base64.b64encode(f.read()).decode()

response = await agent.llm_chat(
    messages=[
        {
            "role": "user",
            "content": [
                {"type": "text", "text": "Что на картинке?"},
                {
                    "type": "image_url",
                    "image_url": {
                        "url": f"data:image/jpeg;base64,{image_b64}"
```

```
                    }
                }
            ]
        }
    ],
    model="gpt-4-vision"
)
```

6.7.6 Информация о моделях

```
# Список всех моделей
models = agent.get_available_models()

for model in models:
    print(f"{model['id']}: {model['name']}")
    print(f"  Embedding: {model.get('is_embedding')}")
    print(f"  Multimodal: {model.get('is_multimodal')}")

# Информация о конкретной модели
model_info = agent.get_model_info("gpt-4o-mini")
```

6.7.7 Выбор модели из настроек

```
# В settings_schema агента:
# {"name": "llm_model", "type": "llm", "llm_type": "llm"}

@agent.job_handler("generate")
async def handle_generate(job):
    model_id = agent.get_setting("llm_model")

    response = await agent.llm_chat(
        messages=[{"role": "user", "content": job.body["prompt"]}],
        model=model_id
    )

    return {"response": response["choices"][0]["message"]["content"]}
```

6.7.8 Обработка ошибок

```
from cognitum_agent.exceptions import LLMError, LLMTimeoutError

try:
    response = await agent.llm_chat(messages=[...], model="gpt-4")
except LLMTimeoutError:
    await agent.publish_log("error", "LLM request timed out")
except LLMError as e:
    await agent.publish_log("error", f"LLM error: {e}")
```

6.8 Логирование и метрики

SDK предоставляет инструменты для централизованного логирования.

6.8.1 Публикация логов

```
await agent.publish_log("info", "Сообщение")
await agent.publish_log("debug", "Отладка", extra={"key": "value"})
await agent.publish_log("warning", "Предупреждение")
await agent.publish_log("error", "Ошибка", extra={"code": 500})
```

6.8.2 Уровни логирования

Уровень	Описание
`debug`	Отладочная информация
`info`	Информационные сообщения
`warning`	Предупреждения
`error`	Ошибки

6.8.3 Дополнительные поля

```
await agent.publish_log(
    "info",
```

```
    "Обработано документов",
    extra={
        "documents_count": 10,
        "duration_ms": 1500,
        "source": "s3://bucket/path"
    }
)
```

6.8.4 Автоматическое добавление job_id

В контексте job handler SDK автоматически добавляет `job_id`:

```
@agent.job_handler("my-task")
async def handle_task(job):
    # job_id добавляется автоматически
    await agent.publish_log("info", "Processing started")

    # Эквивалентно:
    # await agent.publish_log("info", "Processing started",
    #                         extra={"job_id": job.job_id})
```

6.8.5 Конфигурация логирования

```
AgentConfig(
    name="my-agent",
    auto_logging=True,          # Автоотправка логов
    log_level="INFO",           # Минимальный уровень
    log_to_stdout=True,         # Вывод в консоль
    log_exclude_modules=[       # Исключить шумные модули
        "httpx",
        "urllib3"
    ]
)
```

6.8.6 Интеграция с Python logging

```
import logging

logger = logging.getLogger(__name__)

@agent.job_handler("my-task")
async def handle_task(job):
    logger.info("Starting task")  # Автоматически отправляется в платформу

    try:
        result = process()
        logger.info("Task completed", extra={"result": result})
    except Exception as e:
        logger.error("Task failed", exc_info=True)
        raise
```

6.8.7 Метрики

SDK собирает метрики автоматически:

```
# Получить текущие метрики
metrics = agent.metrics.snapshot()

print(f"Jobs processed: {metrics['jobs_processed']}")
print(f"Jobs failed: {metrics['jobs_failed']}")
print(f"LLM requests: {metrics['llm_requests']}")
print(f"LLM tokens: {metrics['llm_tokens']}")
```

6.8.8 Структура метрик

Метрика	Описание
`jobs_processed`	Обработано задач
`jobs_failed`	Неудачных задач
`llm_requests`	Запросов к LLM

Метрика	Описание
`llm_tokens`	Использовано токенов
`uptime_seconds`	Время работы

6.8.9 Просмотр логов

Логи доступны в веб-интерфейсе платформы:

1. Раздел **Logs**
2. Выберите агента
3. Примените фильтры по уровню и времени

6.9 Работа с хранилищами

SDK позволяет получать данные подключения к хранилищам, настроенным в платформе.

6.9.1 Получение списка хранилищ

```
storages = await agent.get_storages()

for storage in storages:
    print(f"Name: {storage['name']}")
    print(f"Type: {storage['type']}")
    print(f"Connection: {storage['connection_data']}")
```

6.9.2 Структура данных хранилища

```
{
    "name": "my-qdrant",
    "type": "qdrant",
    "connection_data": {
        "url": "http://qdrant:6333",
        "api_key": "..."  # если настроен
    }
}
```

6.9.3 Использование в settings_schema

```
AgentConfig(
    name="rag-agent",
    settings_schema=[
        {
            "name": "vector_storage",
            "type": "storage",
            "storage_type": "qdrant",
            "description": "Хранилище для эмбеддингов",
            "required": True
        }
    ]
)
```

6.9.4 Получение данных выбранного хранилища

```
@agent.job_handler("index-document")
async def handle_index(job):
    # Получить имя хранилища из настроек
    storage_name = agent.get_setting("vector_storage")

    # Получить данные подключения
    storages = await agent.get_storages()
    storage = next(
        (s for s in storages if s["name"] == storage_name),
        None
    )

    if not storage:
        raise ValueError(f"Storage '{storage_name}' not found")

    # Использовать connection_data
```

```
    conn = storage["connection_data"]
    # ...
```

6.9.5 Примеры подключений

6.9.5.1 Qdrant

```
from qdrant_client import QdrantClient

storage = await get_storage("my-qdrant")
conn = storage["connection_data"]

client = QdrantClient(
    url=conn["url"],
    api_key=conn.get("api_key")
)
```

6.9.5.2 PostgreSQL

```
import asyncpg

storage = await get_storage("my-postgres")
conn = storage["connection_data"]

pool = await asyncpg.create_pool(
    host=conn["host"],
    port=conn["port"],
    database=conn["database"],
    user=conn["username"],
    password=conn["password"]
)
```

6.9.5.3 S3

```
import boto3

storage = await get_storage("my-s3")
conn = storage["connection_data"]

s3 = boto3.client(
    "s3",
    endpoint_url=conn["endpoint"],
    aws_access_key_id=conn["access_key"],
    aws_secret_access_key=conn["secret_key"],
    region_name=conn.get("region")
)

bucket = conn["bucket"]
```

6.9.6 Безопасность

- Данные подключения передаются по защищённому каналу
- Агент получает только хранилища, к которым ему разрешён доступ
- Пароли не логируются автоматически

6.9.7 Кэширование

SDK кэширует список хранилищ при запуске. Для обновления:

```
# Принудительное обновление
storages = await agent.get_storages(refresh=True)
```

7 Система задач (Jobs)

Система задач обеспечивает асинхронное выполнение операций с гарантией доставки и отслеживанием результатов.

7.1 Основные возможности

- Асинхронное и синхронное выполнение
- Зависимости между задачами (prerequisites)
- Отложенное выполнение (not_before)
- Время жизни задач (TTL)
- Автоматическая маршрутизация к агентам
- Персистентное хранение в PostgreSQL
- Доставка через NATS JetStream

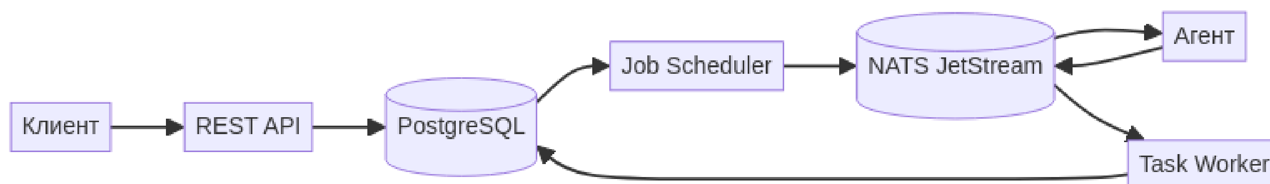
7.2 Концепция задач

7.2.1 Что такое задача (Job)?

Задача — единица работы, которая:

1. Создаётся через API или SDK
2. Сохраняется в базе данных
3. Доставляется агенту через очередь
4. Выполняется агентом
5. Результат сохраняется и возвращается

7.2.2 Компоненты системы задач



7.2.3 Типы задач (Job Types)

Каждый агент регистрирует типы задач, которые может обрабатывать:

```
agent = Agent(AgentConfig(
    name="document-processor",
    job_schemas=[
        {"job_type": "parse-pdf", "description": "Парсинг PDF"},
        {"job_type": "extract-text", "description": "Извлечение текста"}
    ]
))
```

7.2.4 Маршрутизация задач

При создании задачи платформа:

1. Проверяет, что `job_type` зарегистрирован
2. Определяет агента-исполнителя
3. Сохраняет задачу в PostgreSQL
4. Scheduler доставляет задачу агенту

7.2.5 Гарантии доставки

Система обеспечивает **at-least-once** доставку:

- Задача не потеряется при перезапуске компонентов
- При сбое агента задача будет переобработана
- Результат сохраняется в базе данных

7.2.6 Идемпотентность

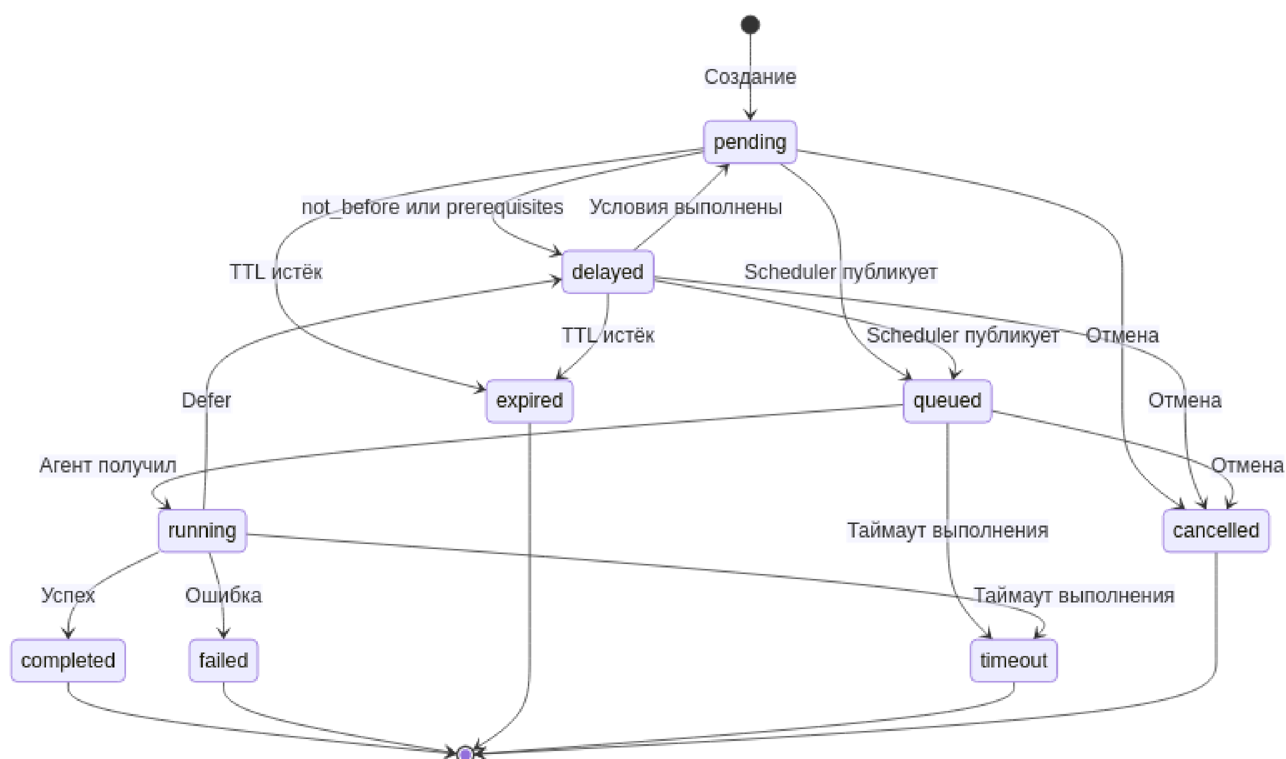
Рекомендуется делать обработчики задач идемпотентными, так как задача может быть доставлена повторно при:

- Сбое агента во время обработки
- Проблемах сети
- Перезапуске компонентов

```
@agent.job_handler("process-order")
async def handle_order(job):
    order_id = job.body["order_id"]

    # Проверить, не обработан ли уже заказ
    if await is_order_processed(order_id):
        return {"status": "already_processed"}

    # Обработать заказ
    await process_order(order_id)
    await mark_order_processed(order_id)

    return {"status": "success"}
```

7.3 Жизненный цикл задачи

7.3.1 Диаграмма состояний



7.3.2 Описание состояний

Статус	Описание

Статус	Описание
`pending`	Задача создана, ожидает обработки
`delayed`	Отложена (не наступило not_before или не завершены prerequisites)
`queued`	Опубликована в NATS, ожидает получения агентом
`running`	Выполняется агентом
`completed`	Успешно завершена
`failed`	Завершена с ошибкой
`expired`	Истёк TTL до начала выполнения
`timeout`	Превышен таймаут выполнения
`cancelled`	Отменена пользователем

7.3.3 Временные метки

Для каждой задачи фиксируются:

Поле	Описание
`created_at`	Время создания
`started_at`	Время начала выполнения
`completed_at`	Время завершения
`not_before`	Время отложенного запуска
`expires_at`	Время истечения TTL

7.3.4 Пример жизненного цикла

1. **10:00:00** — Задача создана (`pending`)
2. **10:00:01** — Scheduler проверяет готовность
3. **10:00:01** — Задача опубликована в NATS (`queued`)
4. **10:00:02** — Агент получил задачу (`running`)
5. **10:00:15** — Агент завершил обработку (`completed`)

7.3.5 Обработка ошибок

При ошибке в агенте:

1. Агент отправляет статус `failed` с текстом ошибки
2. Worker обновляет статус в базе данных
3. Результат содержит поле `error`

```
@agent.job_handler("risky-task")
async def handle_risky(job):
    try:
        result = await do_work()
        return {"success": True, "result": result}
    except Exception as e:
        # Ошибка автоматически записывается
        raise  # SDK отправит статус failed
```

7.3.6 Повторная обработка

Для повторной обработки неудачной задачи:

1. Через API: `POST /api/jobs/{id}/retry`
2. Через UI: кнопка «Retry» в карточке задачи

Создаётся новая задача с тем же телом.

7.4 Синхронное и асинхронное выполнение

7.4.1 Асинхронное выполнение

Стандартный режим — задача создаётся и сразу возвращается ID:

```
# API
POST /api/jobs
{
  "job_type": "process-document",
  "body": {"url": "..."}
}

# Ответ (201 Created)
{
  "job_id": "550e8400-...",
  "status": "pending"
}
```

Клиент должен: 1. Сохранить `job_id` 2. Периодически проверять статус 3. Получить результат после завершения

7.4.2 Синхронное выполнение (Invoke)

Для быстрых операций — ожидание результата в одном запросе:

```
# API
POST /api/jobs/invoke
{
  "job_type": "echo-job",
  "body": {"message": "Hello"},
  "timeout": 30
}

# Ответ (после выполнения)
{
  "job_id": "550e8400-...",
  "status": "completed",
  "result": {"echo": "Hello"},
  "duration_ms": 150
}
```

7.4.3 SDK: Синхронное выполнение

```python
# Асинхронно: создать и забыть
job_id = await agent.submit_job("process-data", {"data": "..."})

# Синхронно: дождаться результата
result = await agent.run_job_sync(
    "process-data",
    body={"data": "..."},
    timeout_s=30
)
print(result)  # Результат выполнения
```

7.4.4 Ожидание результата

```python
# Создать задачу
job_id = await agent.submit_job("long-task", {"data": "..."})

# Дождаться результата отдельно
result = await agent.wait_for_job_result(job_id, timeout_s=60)
```

7.4.5 Когда использовать

Режим	Применение
Асинхронный	Длительные операции, пакетная обработка
Синхронный	Быстрые операции (<30 сек), интерактивные запросы

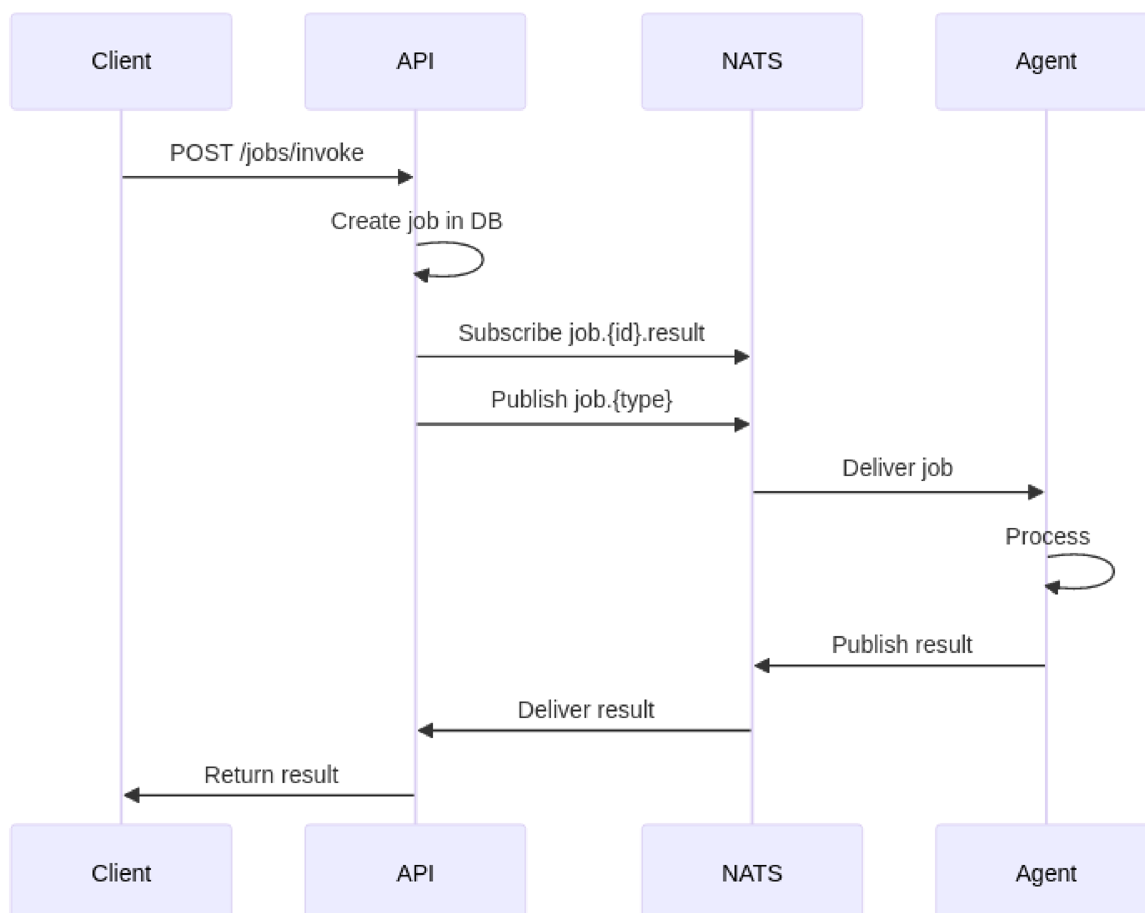
7.4.6 Таймауты

Для синхронного выполнения важно указать timeout:

```
try:
    result = await agent.run_job_sync(
        "slow-task",
        body={...},
        timeout_s=60  # Максимальное ожидание
    )
except TimeoutError:
    print("Task took too long")
```

7.4.7 Реализация синхронного режима

Платформа использует: 1. Публикация задачи напрямую в NATS (минуя Scheduler) 2. Core NATS подписка на `job.{id}.result` 3. Ожидание с таймаутом 4. Возврат результата



7.5 Зависимости задач (Prerequisites)

Система позволяет создавать задачи, которые должны дождаться завершения других задач.

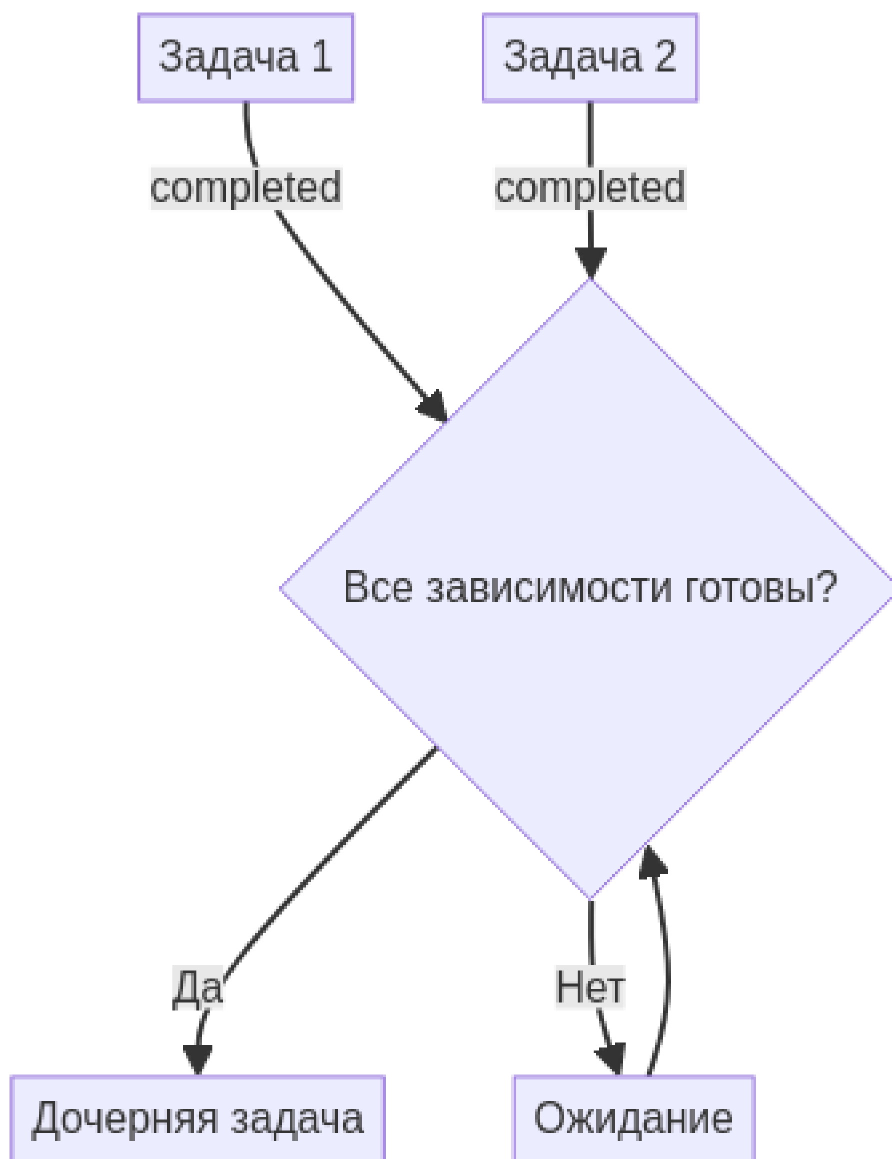
7.5.1 Создание задачи с зависимостями

```
# API
POST /api/jobs
{
  "job_type": "combine-results",
  "body": {"output": "final.json"},
  "prerequisites": [
    "550e8400-...",  # ID задачи 1
    "550e8401-..."   # ID задачи 2
```

```
  ]
}
```

7.5.2 Логика работы

1. Задача создаётся со статусом `delayed`
2. Scheduler периодически проверяет prerequisites
3. Когда все зависимости в статусе `completed`, задача переходит в `pending`
4. Задача публикуется агенту



7.5.3 SDK: Создание цепочки задач

```python
# Создать первую задачу
job1_id = await agent.submit_job("extract", {"source": "doc1.pdf"})

# Создать вторую задачу
job2_id = await agent.submit_job("extract", {"source": "doc2.pdf"})

# Создать задачу, зависящую от обеих
combine_id = await agent.submit_job(
    "combine",
    body={"sources": [job1_id, job2_id]},
    prerequisites=[job1_id, job2_id]
)
```

7.5.4 Проверка зависимостей в обработчике

```
@agent.job_handler("dependent-task")
async def handle_dependent(job):
    if job.prerequisites:
        # Проверить статус зависимостей
        deps = await agent.check_job_dependencies(job.prerequisites)

        incomplete = [
            jid for jid, info in deps.items()
            if not info.get("completed")
        ]

        if incomplete:
            await agent.publish_log(
                "warning",
                f"Prerequisites not ready: {incomplete}"
            )
            job.defer(5)  # Повторить через 5 сек
            return

    # Все зависимости готовы
    return await process_task(job)
```

7.5.5 Ограничения

- Prerequisites должны существовать в базе данных
- Максимум 100 зависимостей на задачу
- Циклические зависимости не допускаются
- При `failed` зависимости дочерняя задача остаётся `delayed`

7.5.6 Каскадная отмена

При отмене родительской задачи дочерние задачи **не отменяются** автоматически. Это поведение по умолчанию, так как:

- Дочерние задачи могут иметь другие зависимости
- Может потребоваться частичная обработка

Для каскадной отмены используйте API:

```
# Отменить задачу и все зависимые
DELETE /api/jobs/{id}?cascade=true
```

7.5.7 Пример: Параллельная обработка

```
# Создать параллельные задачи
tasks = []
for doc in documents:
    job_id = await agent.submit_job("process-doc", {"doc": doc})
    tasks.append(job_id)

# Создать финальную задачу
await agent.submit_job(
    "merge-results",
    body={"task_ids": tasks},
    prerequisites=tasks
)
```

7.6 Отложенное выполнение

Система позволяет планировать выполнение задач на определённое время.

7.6.1 Параметр not_before

```
# API
POST /api/jobs
{
  "job_type": "send-notification",
```

```
  "body": {"message": "Reminder!"},
  "not_before": "2024-01-15T09:00:00Z"
}
```

Задача будет выполнена не раньше указанного времени.

7.6.2 TTL (Time To Live)

Время жизни задачи от момента создания:

```
POST /api/jobs
{
  "job_type": "urgent-task",
  "body": {...},
  "ttl_seconds": 3600  # 1 час
}
```

Если задача не выполнена за TTL, она переходит в статус `expired`.

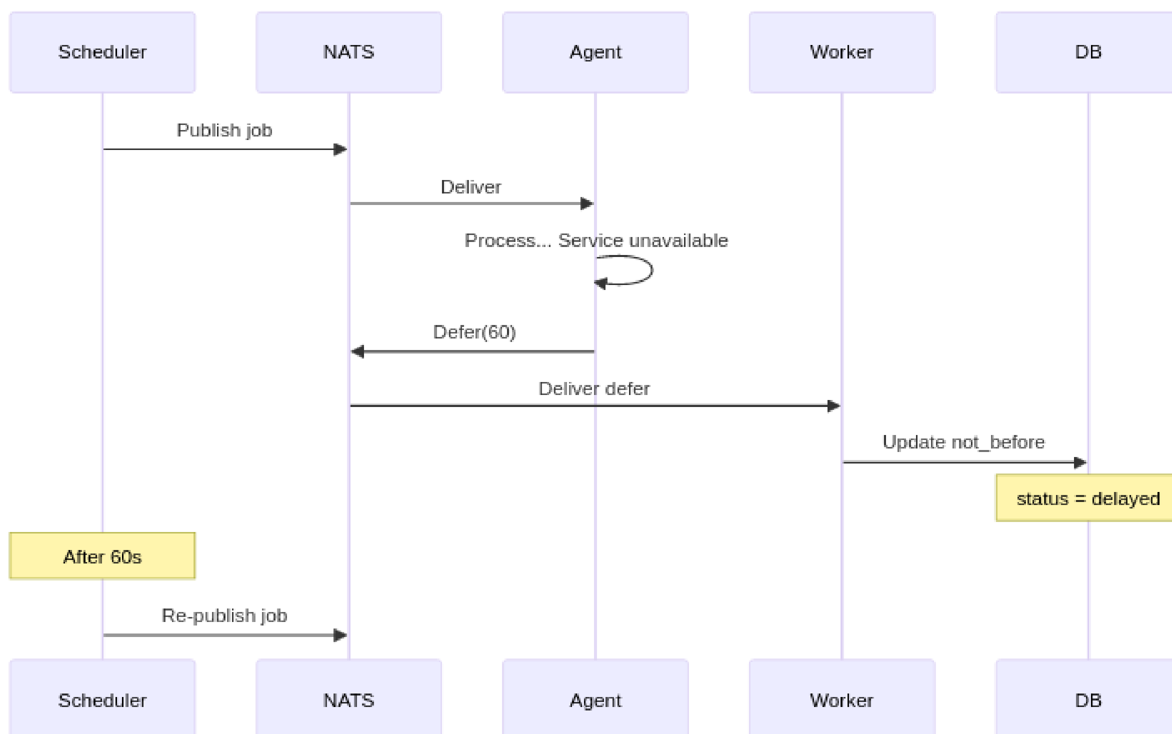
7.6.3 Defer в обработчике

Агент может отложить задачу во время обработки:

```
@agent.job_handler("retry-task")
async def handle_retry(job):
    try:
        result = await call_external_api()
        return {"success": True, "result": result}
    except ServiceUnavailable:
        # Сервис недоступен, повторить через 60 сек
        job.defer(60)
        return  # Результат игнорируется при defer
```

7.6.4 Логика Defer

1. Агент вызывает `job.defer(seconds)`
2. SDK отправляет специальный статус
3. Worker обновляет `not_before` в базе
4. Задача возвращается в статус `delayed`
5. Scheduler повторно опубликует через указанное время



7.6.5 SDK: Отложенные задачи

```python
from datetime import datetime, timedelta

# Задача через 1 час
await agent.submit_job(
    "scheduled-report",
    body={"type": "daily"},
    not_before=datetime.utcnow() + timedelta(hours=1)
)

# Задача с TTL
await agent.submit_job(
    "time-sensitive",
    body={...},
    ttl_seconds=300  # 5 минут на выполнение
)
```

7.6.6 Комбинирование параметров

```
POST /api/jobs
{
  "job_type": "scheduled-task",
  "body": {...},
  "not_before": "2024-01-15T09:00:00Z",
  "ttl_seconds": 7200,  # 2 часа после not_before
  "prerequisites": ["..."]
}
```

Задача: 1. Ждёт завершения prerequisites 2. Ждёт наступления not_before 3. Должна завершиться до expires_at (not_before + TTL)

7.6.7 Планирование задач

Платформа **не** предоставляет cron-подобного планировщика. Для регулярных задач используйте:

1. Внешний планировщик (cron, systemd timer)
2. Отдельный сервис с расписанием
3. Создание следующей задачи по завершении текущей:

```python
@agent.job_handler("recurring-task")
async def handle_recurring(job):
    # Выполнить работу
    result = await do_work()

    # Создать следующую задачу
    await agent.submit_job(
        "recurring-task",
        body=job.body,
        not_before=datetime.utcnow() + timedelta(hours=1)
    )

    return result
```

8 Интеграция с внешними системами

Платформа Cognitum поддерживает интеграцию с различными внешними системами и сервисами.

8.1 Подключение LLM-провайдеров

8.1.1 OpenAI

Подключение к официальному API OpenAI.

Настройка: 1. Перейдите в **LLM APIs → Add Provider** 2. Выберите тип **OpenAI** 3. Заполните параметры: - **Name**: Название (например, "OpenAI Production") - **API Key**: Ключ из platform.openai.com - **Organization ID**: (опционально) ID организации

Доступные модели: - GPT-4, GPT-4 Turbo, GPT-4o - GPT-3.5 Turbo - text-embedding-ada-002, text-embedding-3-small/large

8.1.2 Azure OpenAI

Подключение через Microsoft Azure.

Предварительные требования: - Подписка Azure с доступом к Azure OpenAI Service - Созданный ресурс Azure OpenAI - Развёрнутые модели (deployments)

Настройка: 1. Выберите тип **Azure OpenAI** 2. Заполните параметры: - **Name**: Название - **API Key**: Ключ из Azure Portal - **Endpoint**: URL ресурса (например, `https://myresource.openai.azure.com`) - **API Version**: Версия API (например, `2024-02-15-preview`)

Добавление моделей:

Для каждого deployment создайте модель: - **Deployment Name**: Имя deployment в Azure - **Model Name**: Название модели (gpt-4, gpt-35-turbo)

8.1.3 Ollama (локальные модели)

Запуск LLM локально без отправки данных во внешние сервисы.

Установка Ollama:

```
# Linux/macOS
curl -fsSL https://ollama.com/install.sh | sh

# Или Docker
docker run -d -v ollama:/root/.ollama -p 11434:11434 ollama/ollama
```

Загрузка моделей:

```
ollama pull llama2
ollama pull mistral
ollama pull qwen:14b
```

Настройка в платформе: 1. Выберите тип **Ollama** 2. Укажите **Base URL**: `http://ollama:11434` (или IP хоста)

Платформа автоматически обнаружит загруженные модели.

8.1.4 OpenAI-совместимые API

Любой сервис, реализующий OpenAI API.

Примеры: - vLLM - LocalAI - LM Studio - Text Generation WebUI

Настройка: 1. Выберите тип **OpenAI Compatible** 2. Заполните параметры: - **Base URL**: URL API (например, `http://localhost:8000/v1`) - **API Key**: Ключ (если требуется)

8.1.5 Рекомендации

Сценарий	Рекомендуемый провайдер
Production, высокое качество	OpenAI, Azure
On-premise, конфиденциальность	Ollama, vLLM
Разработка, тестирование	Ollama
Корпоративный compliance	Azure OpenAI

8.2 Интеграция с LDAP/Active Directory

Платформа поддерживает аутентификацию пользователей через корпоративный каталог.

8.2.1 Включение LDAP

Добавьте в `.env`:

```
LDAP_ENABLED=true
LDAP_SERVER=ldap://ldap.company.local:389
LDAP_BASE_DN=dc=company,dc=local
LDAP_USER_DN_TEMPLATE=uid={username},ou=users,dc=company,dc=local
```

8.2.2 Параметры конфигурации

Параметр	Описание
`LDAP_ENABLED`	Включить LDAP (`true`/`false`)
`LDAP_SERVER`	URL сервера (ldap:// или ldaps://)
`LDAP_BASE_DN`	Базовый DN для поиска
`LDAP_USER_DN_TEMPLATE`	Шаблон DN пользователя
`LDAP_SERVICE_ACCOUNT`	Сервисный аккаунт для поиска
`LDAP_SERVICE_ACCOUNT_PASSWORD`	Пароль сервисного аккаунта

8.2.3 Active Directory

Для интеграции с AD используйте формат UPN:

```
LDAP_SERVER=ldap://ad.company.com:389
LDAP_BASE_DN=dc=company,dc=com
LDAP_USER_DN_TEMPLATE={username}@company.com
LDAP_SERVICE_ACCOUNT=cognitum-service@company.com
LDAP_SERVICE_ACCOUNT_PASSWORD=****
```

8.2.4 OpenLDAP

Для OpenLDAP используйте DN:

```
LDAP_SERVER=ldap://ldap.company.local:389
LDAP_BASE_DN=dc=company,dc=local
LDAP_USER_DN_TEMPLATE=uid={username},ou=people,dc=company,dc=local
LDAP_SERVICE_ACCOUNT=cn=admin,dc=company,dc=local
LDAP_SERVICE_ACCOUNT_PASSWORD=****
```

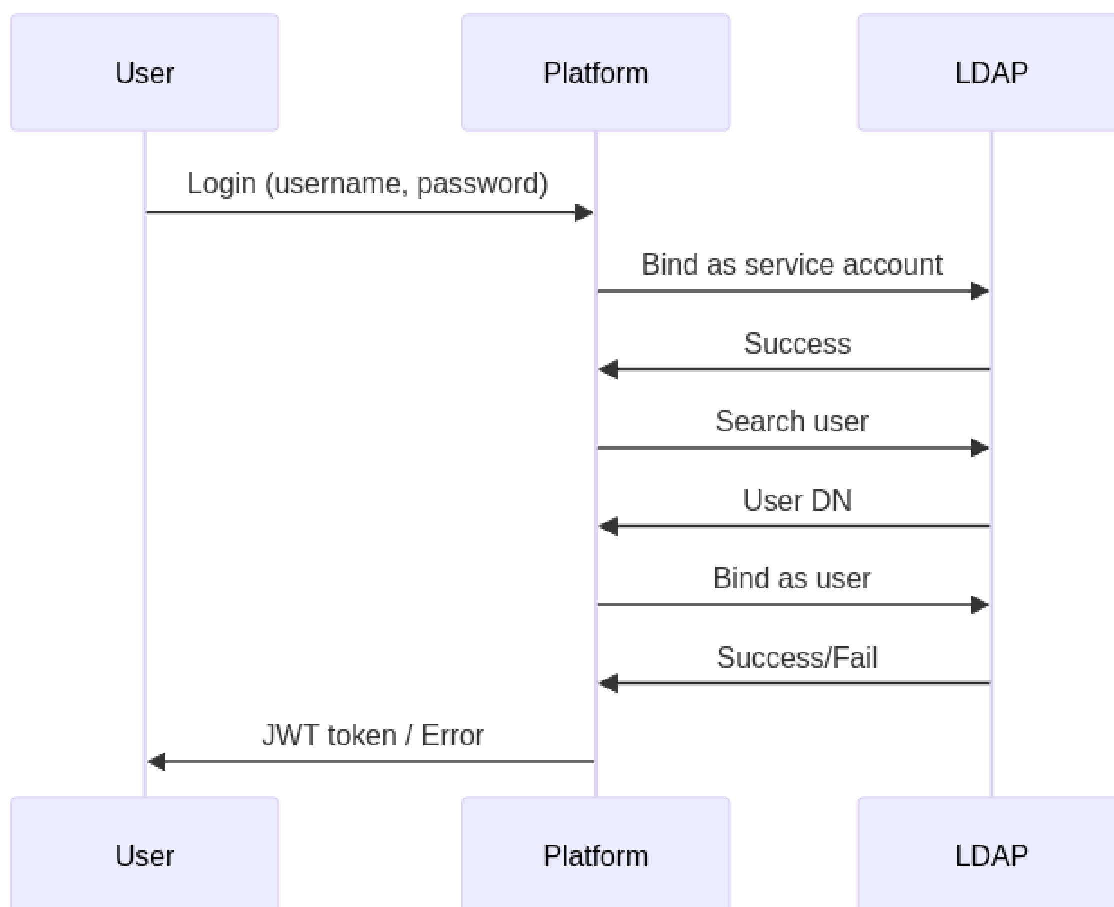
8.2.5 LDAPS (SSL/TLS)

Для защищённого подключения:

```
LDAP_SERVER=ldaps://ldap.company.local:636
```

Убедитесь, что сертификат сервера доверен.

8.2.6 Процесс аутентификации



8.2.7 Синхронизация пользователей

При первом входе через LDAP:

1. Платформа проверяет учётные данные в LDAP
2. Создаётся локальная запись пользователя
3. Атрибуты (email, имя) копируются из LDAP

При последующих входах:

1. Аутентификация через LDAP
2. Обновление локальных атрибутов

8.2.8 Локальные и LDAP пользователи

Платформа поддерживает оба типа пользователей одновременно:

- LDAP-пользователи аутентифицируются через каталог
- Локальные пользователи (включая admin) — через базу данных

8.2.9 Troubleshooting

Ошибка подключения:

```
# Проверить доступность сервера
ldapsearch -x -H ldap://ldap.company.local:389 -b "dc=company,dc=local"
```

Ошибка bind: - Проверьте формат `LDAP_USER_DN_TEMPLATE` - Убедитесь в правильности пароля сервисного аккаунта

Пользователь не найден: - Проверьте `LDAP_BASE_DN` - Убедитесь, что пользователь существует в указанном OU

8.3 Подключение S3-хранилищ

Платформа позволяет подключать S3-совместимые объектные хранилища.

8.3.1 Поддерживаемые хранилища

- Amazon S3
- MinIO
- Yandex Object Storage
- VK Cloud Storage
- Ceph (с S3 gateway)
- Любое S3-совместимое хранилище

8.3.2 Добавление S3 хранилища

1. Перейдите в **Storages** → **Add Storage**
2. Выберите тип **S3**
3. Заполните параметры:

Параметр	Описание	Пример
Name	Уникальное имя	`company-documents`
Endpoint	URL S3 API	`https://s3.amazonaws.com`
Access Key	Ключ доступа	`AKIA...`
Secret Key	Секретный ключ	`****`
Bucket	Имя bucket	`my-bucket`
Region	Регион	`us-east-1`

8.3.3 Примеры конфигурации

8.3.3.1 Amazon S3

```
Endpoint: https://s3.amazonaws.com
Region: us-east-1
Access Key: AKIAIOSFODNN7EXAMPLE
Secret Key: wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
Bucket: my-bucket
```

8.3.3.2 MinIO

```
Endpoint: http://minio:9000
Region: us-east-1
Access Key: minioadmin
Secret Key: minioadmin
Bucket: cognitum-files
```

8.3.3.3 Yandex Object Storage

```
Endpoint: https://storage.yandexcloud.net
Region: ru-central1
Access Key: <service_account_key_id>
Secret Key: <service_account_secret>
Bucket: my-bucket
```

8.3.4 Использование в агенте

```
import boto3

# Получить данные хранилища
storages = await agent.get_storages()
```

```python
s3_storage = next(s for s in storages if s["name"] == "company-documents")
conn = s3_storage["connection_data"]

# Создать клиент
s3 = boto3.client(
    "s3",
    endpoint_url=conn["endpoint"],
    aws_access_key_id=conn["access_key"],
    aws_secret_access_key=conn["secret_key"],
    region_name=conn.get("region", "us-east-1")
)

# Работа с файлами
bucket = conn["bucket"]

# Загрузить файл
s3.upload_file("local.txt", bucket, "remote/path/file.txt")

# Скачать файл
s3.download_file(bucket, "remote/path/file.txt", "local.txt")

# Получить список файлов
response = s3.list_objects_v2(Bucket=bucket, Prefix="remote/path/")
for obj in response.get("Contents", []):
    print(obj["Key"])
```

8.3.5 Права доступа

Убедитесь, что учётные данные имеют необходимые права:

- `s3:GetObject` — чтение файлов
- `s3:PutObject` — запись файлов
- `s3:ListBucket` — список файлов
- `s3:DeleteObject` — удаление (если требуется)

8.3.6 Безопасность

- Используйте отдельные учётные данные для платформы
- Ограничьте доступ только необходимыми bucket'ами
- Используйте HTTPS для endpoint
- Регулярно ротируйте ключи

8.3.7 Тестирование подключения

1. Откройте карточку хранилища
2. Нажмите **Test Connection**
3. Платформа проверит доступ к bucket

9 Приложения

Справочная информация и примеры конфигураций.

9.1 Справочник переменных окружения

Полный список переменных окружения платформы.

9.1.1 База данных

Переменная	По умолчанию	Описание
POSTGRES_USER	aiplatform	Пользователь PostgreSQL
POSTGRES_PASSWORD	aiplatform	Пароль PostgreSQL
POSTGRES_DB	aiplatform	Имя базы данных
POSTGRES_PORT	5432	Порт PostgreSQL

9.1.2 NATS

Переменная	По умолчанию	Описание
NATS_PORT	4222	Порт клиентских подключений
NATS_MONITORING_PORT	8222	Порт мониторинга

9.1.3 Приложение

Переменная	По умолчанию	Описание
APP_ENV	development	Окружение (development/production)
APP_SECRET_KEY	change_me_secret	Секретный ключ приложения
FRONTEND_PORT	8080	Порт веб-интерфейса

9.1.4 JWT

Переменная	По умолчанию	Описание
JWT_SECRET_KEY	APP_SECRET_KEY	Ключ подписи JWT
JWT_EXPIRATION_HOURS	24	Время жизни токена

9.1.5 LDAP

Переменная	По умолчанию	Описание
LDAP_ENABLED	false	Включить LDAP
LDAP_SERVER	—	URL сервера
LDAP_BASE_DN	—	Базовый DN
LDAP_USER_DN_TEMPLATE	—	Шаблон DN пользователя
LDAP_SERVICE_ACCOUNT	—	Сервисный аккаунт
LDAP_SERVICE_ACCOUNT_PASSWORD	—	Пароль сервисного аккаунта

9.1.6 SMTP

Переменная	По умолчанию	Описание
SMTP_HOST	—	SMTP сервер
SMTP_PORT	587	Порт SMTP
SMTP_USER	—	Пользователь SMTP
SMTP_PASSWORD	—	Пароль SMTP
SMTP_FROM	noreply@cognitum.local	Адрес отправителя
SMTP_TLS	true	Использовать TLS

9.1.7 Аудит

Переменная	По умолчанию	Описание
AUDIT_ENABLED	true	Включить аудит
AUDIT_FILE_PATH	/var/log/cognitum/audit.log	Путь к файлу
AUDIT_ROTATION_SIZE	100MB	Размер ротации
AUDIT_ROTATION_COUNT	10	Количество файлов
AUDIT_RETENTION_DAYS	365	Срок хранения

9.1.8 Дополнительные БД

Переменная	По умолчанию	Описание
QDRANT_PORT	6333	Порт Qdrant
NEO4J_HTTP_PORT	7474	HTTP порт Neo4j
NEO4J_BOLT_PORT	7687	Bolt порт Neo4j
NEO4J_PASSWORD	changeme	Пароль Neo4j
LOKI_PORT	3100	Порт Loki

9.1.9 Docker образы

Переменная	По умолчанию	Описание
APP_IMAGE	cognitum-app:latest	Образ приложения
AGENT_IMAGE	cognitum-echo-agent:latest	Образ агента

9.2 Коды ошибок

9.2.1 HTTP коды ответов API

Код	Описание
200	Успешный запрос
201	Ресурс создан
204	Успешно, без содержимого
400	Некорректный запрос
401	Не авторизован
403	Доступ запрещён
404	Ресурс не найден
409	Конфликт
422	Ошибка валидации
500	Внутренняя ошибка сервера
503	Сервис недоступен

9.2.2 Ошибки аутентификации

Код	Сообщение	Описание
AUTH001	invalid_credentials	Неверный логин или пароль
AUTH002	token_expired	Токен истёк
AUTH003	token_invalid	Недействительный токен
AUTH004	token_revoked	Токен отозван
AUTH005	account_disabled	Учётная запись отключена
AUTH006	ldap_error	Ошибка LDAP

9.2.3 Ошибки задач

Код	Сообщение	Описание
JOB001	job_type_not_found	Тип задачи не зарегистрирован
JOB002	no_agent_available	Нет доступных агентов
JOB003	job_not_found	Задача не найдена
JOB004	invalid_prerequisites	Недействительные зависимости
JOB005	job_already_completed	Задача уже завершена
JOB006	job_timeout	Таймаут выполнения
JOB007	job_expired	TTL истёк

9.2.4 Ошибки LLM

Код	Сообщение	Описание
LLM001	model_not_found	Модель не найдена
LLM002	provider_unavailable	Провайдер недоступен
LLM003	rate_limit_exceeded	Превышен лимит запросов
LLM004	context_too_long	Превышен размер контекста
LLM005	api_key_invalid	Недействительный API-ключ

9.2.5 Ошибки агентов

Код	Сообщение	Описание
AGT001	agent_not_found	Агент не найден
AGT002	agent_offline	Агент недоступен
AGT003	registration_failed	Ошибка регистрации
AGT004	config_invalid	Некорректная конфигурация

9.2.6 Ошибки хранилищ

Код	Сообщение	Описание
STR001	storage_not_found	Хранилище не найдено
STR002	connection_failed	Ошибка подключения
STR003	access_denied	Доступ запрещён

9.2.7 Формат ответа об ошибке

```
{
  "error": {
    "code": "JOB001",
    "message": "Job type 'unknown-task' is not registered",
    "details": {
      "job_type": "unknown-task",
      "available_types": ["echo-job", "process-document"]
    }
  }
}
```

9.3 Примеры конфигураций

9.3.1 Минимальная конфигурация (разработка)

```
# .env

# База данных
POSTGRES_USER=aiplatform
POSTGRES_PASSWORD=aiplatform
POSTGRES_DB=aiplatform

# Приложение
```

```
APP_ENV=development
APP_SECRET_KEY=dev_secret_key_change_in_production

# Порт
FRONTEND_PORT=8080
```

9.3.2 Production конфигурация

```
# .env

# База данных
POSTGRES_USER=cognitum_prod
POSTGRES_PASSWORD=xK9#mP2$vL7@nR4%qW5!
POSTGRES_DB=cognitum

# Neo4j
NEO4J_PASSWORD=zQ8!wE3$rT6@yU9#

# Приложение
APP_ENV=production
APP_SECRET_KEY=a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6q7r8s9t0u1v2w3x4y5z6

# JWT
JWT_SECRET_KEY=9z8y7x6w5v4u3t2s1r0q9p8o7n6m5l4k3j2i1h0g9f8e7d6c5b4a3
JWT_EXPIRATION_HOURS=8

# Аудит
AUDIT_ENABLED=true
AUDIT_RETENTION_DAYS=2555
```

9.3.3 Конфигурация с LDAP

```
# .env

# ... основные настройки ...

# LDAP
LDAP_ENABLED=true
LDAP_SERVER=ldaps://ldap.company.com:636
LDAP_BASE_DN=dc=company,dc=com
LDAP_USER_DN_TEMPLATE=uid={username},ou=users,dc=company,dc=com
LDAP_SERVICE_ACCOUNT=cn=cognitum-service,ou=services,dc=company,dc=com
LDAP_SERVICE_ACCOUNT_PASSWORD=sV7#nB9$mK2@lH5%
```

9.3.4 Конфигурация с SMTP

```
# .env

# ... основные настройки ...

# SMTP
SMTP_HOST=smtp.company.com
SMTP_PORT=587
SMTP_USER=cognitum@company.com
SMTP_PASSWORD=eM4!iL9$pW2@sS7#
SMTP_FROM=Cognitum Platform <noreply@company.com>
SMTP_TLS=true
```

9.3.5 Пример агента

```
# agent/main.py

import asyncio
from cognitum_agent import Agent, AgentConfig

agent = Agent(AgentConfig(
    name="document-processor",
    description="Обработка документов",

    job_schemas=[
        {
            "job_type": "parse-document",
            "description": "Парсинг документа",
```

```
                "parameters_schema": [
                    {"name": "url", "type": "string", "required": True},
                    {"name": "format", "type": "select", "options": ["pdf", "docx"]}
                ]
            }
        ],

        settings_schema=[
            {
                "name": "llm_model",
                "type": "llm",
                "llm_type": "llm",
                "description": "Модель для анализа"
            },
            {
                "name": "storage",
                "type": "storage",
                "storage_type": "s3",
                "description": "Хранилище документов"
            }
        ],

        chat_models=[
            {
                "internal_id": "doc-chat",
                "name": "Document Chat",
                "description": "Чат по документам",
                "supports_stream": True
            }
        ]
))

@agent.job_handler("parse-document")
async def handle_parse(job):
    url = job.body["url"]
    # ... обработка ...
    return {"success": True, "pages": 10}

@agent.chat_handler("doc-chat")
async def handle_chat(request):
    messages = request["messages"]
    # ... генерация ответа ...
    return {"content": "Ответ на основе документов"}

async def main():
    async with agent:
        await agent.publish_log("info", "Agent started")
        # Агент работает до остановки

asyncio.run(main())
```

9.3.6 Dockerfile агента

```
FROM python:3.11-slim

WORKDIR /app

# Установка SDK
COPY sdk/ /sdk/
RUN pip install /sdk/

# Установка зависимостей агента
COPY requirements.txt .
RUN pip install -r requirements.txt

# Копирование кода агента
COPY agent/ .

# Запуск
CMD ["python", "main.py"]
```

9.3.7 docker-compose.yml для агента

```yaml
services:
  my-agent:
    build: ./agent
    environment:
      - NATS_URL=nats://nats:4222
    networks:
      - cognitum-network
    restart: unless-stopped
    depends_on:
      - nats

networks:
  cognitum-network:
    external: true
```