Практический проект для курса AI-driven developer.

Описание задачи.

Цель проекта: Разработать прототип интеллектуального консультанта по подбору автомобиля, демонстрирующий практические навыки интеграции внешнего LLM-API с backend-системой, проектирования API и создания современного веб-интерфейса.
Краткое описание: Веб-приложение, в котором пользователь в диалоговом режиме формулирует свои требования к автомобилю (бюджет, тип, ключевые параметры). На основе диалога система, используя LLM для анализа запроса и вызова функций (tool calling), выполняет поиск по каталогу, сравнивает модели по заданным критериям и формирует персонализированную сравнительную таблицу (1-3 модели) для принятия решения.

Учебные фокусы:

- Архитектура backend для интеграции с LLM.
- Реализация pattern «LLM Orchestrator» с function calling.
- Проектирование REST API для взаимодействия между сервисами.
- Создание реактивного чат-интерфейса на современном стеке React.

Бизнес-задачи

Основная цель (MVP):
Создать работающий прототип, который наглядно демонстрирует полный цикл работы AI-консультанта: от диалога с пользователем до формирования структурированной рекомендации.

Ключевые учебные/технические задачи:

1. Реализовать ядро интеллектуального подбора: Обеспечить корректный парсинг требований из естественного языка

(LLM) и преобразование их в структурированный поисковый запрос к каталогу.
2. Обеспечить устойчивый диалог: Реализовать логику ведения контекстного диалога, где LLM может задавать уточняющие вопросы и вызывать backend-функции (поиск, сравнение).
3. Сформировать понятный результат: Разработать алгоритм сравнения и форматирования итоговых данных (сравнительная таблица) на основе найденных вариантов.
4. Подготовить масштабируемую архитектуру: Спроектировать и реализовать систему из слабо связанных сервисов (пользователи, каталог, чат, LLM-оркестратор), готовую к расширению функциональности.
5. Учет истории поисков и профиля пользователя.

Функциональные блоки продукта (MVP).

1. Модуль взаимодействия (Frontend - MVP Core)

- Чат-интерфейс:
    - Отправка/получение сообщений (пользователь / ассистент).
    - Индикатор процесса «печатает...».
    - Очистка истории диалога.
- Панель результатов:
    - Отображение итоговой сравнительной таблицы (1-3 модели) после завершения диалога.
    - Ключевые параметры: модель, цена, год, тип топлива, расход, примерная стоимость обслуживания/налога в год.

2. Сервисный слой (Backend)

- Сервис диалога (Chat Service):
    - Управление сессиями и историей сообщений.
    - Прием сообщений от фронтенда и маршрутизация в LLM-оркестратор.
    - Сохранение финальных результатов подбора.

- LLM-оркестратор (AI Gateway Service):
    - Ядро системы. Управление контекстом диалога.
    - Интеграция с внешним LLM-API (OpenAI/Anthropic/Ollama).
    - Парсинг intent пользователя и вызов соответствующих tools (функций).
    - Управление flow диалога (запрос недостающих параметров).
- Сервис поиска (Search Service):
    - Хранение структурированных данных об автомобилях (бренды, модели, характеристики).
    - Функция (tool) поиска: Прием структурированного запроса (фильтры) и возврат списка моделей.
    - Функция (tool) сравнения: Прием списка моделей и возврат сравнения по ключевым параметрам.
- Сервис пользователей (User Service):
    - Регистрация / аутентификация / JWT.
    - Хранение профилей пользователей.
- Сервис диалога (Chat Service):
    - Привязка истории диалогов и результатов к авторизованному пользователю.

3. Модуль управления учетной записью (Frontend - Full)

- Авторизация:
    - Страница входа.
    - Сохранение JWT-токена, защищенные маршруты.
- Личный кабинет (User Profile):
    - Просмотр и редактирование данных профиля (имя, email, аватар).
    - История поисков: Список датированных сессий диалога с AI-консультантом.
    - Сохраненные результаты: Возможность просмотреть, сохранить и удалить конкретные результаты подбора (таблицы сравнения).

4. Соответствующий бэкенд-функционал (Backend - Full)

- User Service:
  - CRUD для данных профиля.
  - Смена пароля.
- Chat Service:
  - Эндпоинт для получения списка сессий текущего пользователя (`GET /api/chat/sessions`).
  - Эндпоинт для получения детальной истории сообщений конкретной сессии.
  - Эндпоинты для сохранения/удаления результата подбора в «избранное».

Описание структуры БД.

Справочники

brands

- `id (PK, UUID)`
- `name (VARCHAR, NOT NULL, UNIQUE) -- "Toyota", "BMW"`
- `country_id (FK → countries.id) -- Страна происхождения`
- `created_at`

models

- `id (PK, UUID)`
- `brand_id (FK → brands.id, NOT NULL)`
- `name (VARCHAR, NOT NULL) -- "Camry", "X5"`
- `generation (VARCHAR, NULL) -- Номер поколения (опционально)`
- `production_start_year (INT) -- Год начала выпуска`
- `production_end_year (INT, NULL) -- Год окончания`
- `average_price_rub (DECIMAL, NULL) -- Средняя рыночная цена`
- `created_at`

body_types

- `id (PK, UUID)`
- `name (VARCHAR, UNIQUE) -- "седан", "внедорожник"`

- code (VARCHAR, UNIQUE) -- "sedan", "suv", "hatchback"

fuel_types

- id (PK, UUID)
- name (VARCHAR, UNIQUE) -- "бензин", "дизель", "гибрид"
- code (VARCHAR, UNIQUE) -- "petrol", "diesel", "hybrid"

transmissions

- id (PK, UUID)
- name (VARCHAR, UNIQUE) -- "автоматическая", "механическая"
- code (VARCHAR, UNIQUE) -- "automatic", "manual", "cvt"

drive_types

- id (PK, UUID)
- name (VARCHAR, UNIQUE) -- "передний", "задний", "полный"
- code (VARCHAR, UNIQUE) -- "fwd", "rwd", "awd

countries

- id (PK, UUID)
- name (VARCHAR)
- iso_code (CHAR(2))

cities

- id (PK, UUID)
- country_id (FK → countries.id)
- name (VARCHAR)

Характеристики модели

model_specifications

- id (PK, UUID)
- model_id (FK → models.id, NOT NULL)
- body_type_id (FK → body_types.id, NOT NULL)
- fuel_type_id (FK → fuel_types.id, NOT NULL)
- transmission_id (FK → transmissions.id, NOT NULL)

- drive_type_id (FK → drive_types.id, NOT NULL)
- engine_volume_l (DECIMAL(3,1)) -- Объем двигателя, л
- horsepower (INT) -- Лошадиные силы
- fuel_consumption_combined (DECIMAL(4,1)) -- Средний расход
- insurance_cost_per_year_rub (DECIMAL) -- Примерная страховка
- annual_tax_cost_rub (DECIMAL) -- Примерный налог в год
- maintenance_cost_per_year_rub (DECIMAL) -- Обслуживание в год
- is_active (BOOLEAN, DEFAULT TRUE) -- Активна для подбора
- created_at
- updated_at

search_results

- id (PK, UUID)
- chat_session_id (FK → chat_sessions.id, UNIQUE)
- user_id (FK → users.id, NOT NULL)
- search_query_summary (TEXT)
- result_data (JSONB, NOT NULL)
- is_saved (BOOLEAN, DEFAULT FALSE)
- created_at

Индексы для поиска

```
INDEX idx_model_spec_search (
  model_id,
  fuel_type_id,
  price_range_id,
  is_active
)
```

Пользователи и роли

users

- id (PK, UUID)
- email (VARCHAR, UNIQUE, NOT NULL)
- password_hash (VARCHAR, NOT NULL)
- name (VARCHAR)
- avatar_url (VARCHAR, NULL)
- role (ENUM: 'user', 'admin') -- *Роль в системе*
- status (ENUM: 'active', 'blocked', 'pending')
- created_at
- updated_at

user_profiles

- user_id (PK, FK → users.id, ON DELETE CASCADE)
- preferred_budget_min_rub (DECIMAL)
- preferred_budget_max_rub (DECIMAL)
- preferred_body_type_id (FK → body_types.id, NULL)
- preferred_fuel_type_id (FK → fuel_types.id, NULL)
- city_id (FK → cities.id, NULL)
- updated_at

Чаты

chat_sessions

- id (PK, UUID)
- user_id (FK → users.id, NULL) -- *NULL для неавторизованных*
- title (VARCHAR(255)) -- *"Подбор SUV до 3 млн"*
- context_summary (JSONB, NULL) -- *Суммаризация требований пользователя от AI*
- status (ENUM: 'active', 'completed', 'archived')
- created_at
- finished_at (TIMESTAMP, NULL)

chat_messages

- id (PK, UUID)

- chat_session_id (FK → chat_sessions.id, ON DELETE CASCADE)
- role (ENUM: 'user', 'assistant', 'system')
- content (TEXT) -- Текст сообщения
- metadata (JSONB, NULL) -- Доп. данные (например, вызванные tools)
- created_at

Вспомогательные таблицы для AI и сравнения

comparison_criteria

- id (PK, UUID)
- code (VARCHAR, UNIQUE) -- 'price', 'fuel_consumption', 'annual_cost'
- name (VARCHAR) -- "Цена", "Расход топлива", "Стоимость владения в год"
- description (TEXT)
- units (VARCHAR) -- "руб.", "л/100км", "руб./год"
- priority (INT) -- Порядок отображения
- is_active (BOOLEAN, DEFAULT TRUE)

external_data_cache

- id (PK, UUID)
- key_hash (VARCHAR, UNIQUE) -- Хэш параметров запроса
- data (JSONB) -- Ответ внешнего сервиса
- expires_at (TIMESTAMP) -- Время инвалидации
- created_at

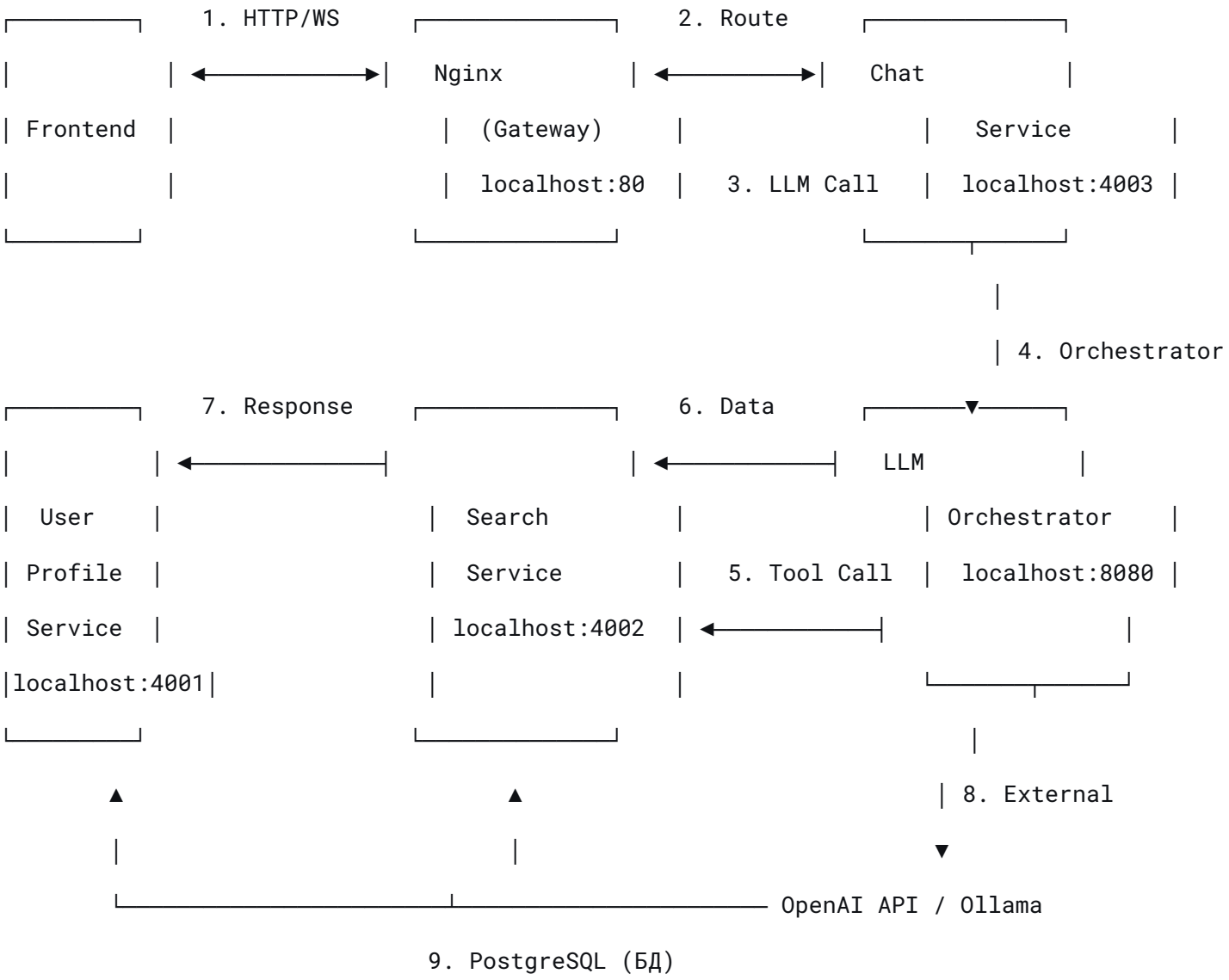
Технологии и логика взаимодействия.

1. Технологический стек (конкретизированный)

Слой	Технологии	Обоснование для учебного проекта
Фронтенд	React 19 + TypeScript, Vite, Ant Design, TanStack Query, Zustand, React Hook Form + Zod	Современный стек с полной типизацией. React 19 с `useOptimistic`, `useActionState`.
API Gateway	Nginx (как reverse proxy)	Достаточно для MVP. Простая настройка, работает в Docker.
Бэкенд (микросервисы)	Node.js 20+, Express 5, TypeScript, Prisma ORM, Zod, JWT	Единый стек для всех сервисов. Prisma для удобной работы с БД.
Базы данных	PostgreSQL 16 (основная), Redis 7 (кэш, сессии)	Стандартный выбор. Redis для кэша запросов и контекста диалогов.
LLM-интеграция	OpenAI API (GPT-4 Turbo) или Ollama (локально)	OpenAI — простота интеграции. Ollama — бесплатно, локально.
Инфраструктура	Docker + Docker Compose, GitHub Actions (базовый CI)	Контейнеризация для воспроизводимости.

Мониторинг/Логи	Winston (логи), встроенные health-checks	Минимально необходимое для отладки.

2. Архитектурная схема взаимодействия (последовательность)



3. Детальная логика для ключевого сценария "Подбор автомобиля"

Шаг 1: Инициализация диалога

`Frontend → POST /api/chat/sessions { user_id } → Chat Service`

Chat Service → Создаёт chat_session → Возвращает session_id

фронтенду

Шаг 2: Пользователь отправляет сообщение

Frontend → POST /api/chat/sessions/{id}/messages { "content": "Ищу седан до 2 млн" }

Nginx → Маршрутизация на Chat Service (localhost:4003)

Шаг 3: Обработка сообщения AI-оркестратором

Chat Service → POST /api/llm/process { session_id, message } → LLM Orchestrator

LLM Orchestrator:

1. Достаёт историю диалога из Redis по session_id
2. Отправляет историю + новое сообщение в OpenAI API с tools:
3. - search_cars(budget_max, body_type, ...)
4. - compare_models(model_ids[])
5. LLM возвращает tool call "search_cars"

Шаг 4: Выполнение поиска

LLM Orchestrator → POST /api/search { budget_max: 2000000, body_type: "sedan" } → Search Service

Search Service:

1. Проверяет кэш в Redis (search:cars:sedan:2000000)
2. Если нет → запрос к PostgreSQL (model_specifications)
3. Возвращает 5-10 подходящих моделей с характеристиками

Шаг 5: Формирование ответа AI

LLM Orchestrator получает результаты поиска → отправляет их в контексте обратно в LLM

LLM анализирует → может:

1. Задать уточняющий вопрос (сохраняется в историю)
2. Вызвать compare_models для топ-3 вариантов
3. Сформировать финальный ответ с таблицей сравнения
4. LLM использует справочник `comparison_criteria` для формирования финальной таблицы сравнения

Шаг 6: Сохранение результата

1. LLM Orchestrator → Сохраняет финальный результат в search_results (через Chat Service)
2. Chat Service → Отправляет ответ фронтенду (возможно, streaming)
3. LLM Orchestrator сохраняет финальный результат в таблицу `search_results` через Chat Service

Шаг 7: Отображение и сохранение

Frontend отображает сообщение AI с таблицей сравнения

Пользователь может нажать "Сохранить в избранное" → PATCH /api/search-results/{id}

4. Роль каждого компонента (уточнённая)

Компонент	Назначение	Технологии	Порт
Nginx Gateway	Единая точка входа, балансировка, CORS	Nginx	80
User Service	Аутентификация, профили, JWT	Node.js, Express, Prisma	4001
Search Service	Управление каталогом моделей, поиск	Node.js, Express, Prisma, Redis	4002
Chat Service	Сессии диалогов, сообщения, история	Node.js, Express, Prisma	4003
LLM Orchestrator	Интеграция с AI, управление диалогом, tool calling	Node.js, Express, OpenAI SDK	8080
PostgreSQL	Основное хранилище данных	PostgreSQL 16	5432
Redis	Кэш запросов, контекст диалогов, сессии	Redis 7	6379

5. Особенности реализации для учебного проекта

1. Упрощённая аутентификация: JWT-токены, хранящиеся в HttpOnly cookies.
2. Кэширование: Redis для:
    - `search:cars:{query_hash}` — результаты поиска (TTL 10 мин)
    - `chat:{session_id}:context` — текущий контекст диалога
    - `session:{user_id}` — активные сессии пользователя
3. Error Handling: Единый формат ответа:

```
{
    "success": boolean,
    "data": any,
    "error": { "code": string, "message": string }
}
```

Промпты

Фронтенд (React 19 + TypeScript)

Создай production-ready веб-приложение на React 19 с TypeScript для AI-консультанта по подбору автомобилей.

Технический стек:
- React 19 (useOptimistic, useActionState, Suspense)
- TypeScript 5.4+
- Vite 5+ с vite-tsconfig-paths
- Ant Design 5.x + Tailwind CSS
- React Router 7+
- Zustand для глобального состояния
- Axios + TanStack Query v5
- React Hook Form + Zod

Структура проекта (feature-based):

```
src/
├── features/
│   ├── auth/          # Логин, регистрация, protected routes
│   ├── chat/          # Чат-интерфейс, отправка сообщений
│   ├── profile/       # Личный кабинет, история, избранное
│   └── search/        # Отображение результатов, таблица сравнения
├── components/        # UI компоненты (Message, CarCard, ComparisonTable)
├── lib/               # API клиенты, утилиты
├── types/             # Типы TypeScript
└── stores/            # Zustand stores
```

Функциональные требования:

1. Авторизация:
   - Страницы /login и /register
   - Валидация форм через Zod
   - Сохранение JWT в HttpOnly cookie
   - ProtectedRoute для защиты /chat и /profile

2. Чат-интерфейс (/chat):
   - Header с информацией о пользователе и выходом
   - Список сообщений (пользователь справа, ассистент слева)
   - Поле ввода с поддержкой Enter для отправки
   - Индикатор "Ассистент печатает..."
   - Автоскролл к последнему сообщению
   - Кнопка очистки истории текущего чата

3. Отображение результатов:
   - Когда AI возвращает результат поиска, отображать сравнительную таблицу
   - Таблица должна показывать 1-3 модели с параметрами: цена, год, расход, стоимость владения
   - Кнопка "Сохранить в избранное" (только для авторизованных)

4. Личный кабинет (/profile):
   - Редактирование профиля (имя, email, аватар)
   - Вкладка "История поисков": список сессий чата с датами
   - Вкладка "Избранное": сохранённые результаты с возможностью удаления

API контракты (реализовать в lib/api.ts):
- POST /api/auth/login
- POST /api/auth/register
- POST /api/chat/sessions
- POST /api/chat/sessions/{id}/messages
- GET /api/chat/sessions (история)
- POST /api/search-results/{id}/save (сохранить в избранное)

Обязательные фичи React 19:
- useOptimistic для отправки сообщений
- useActionState для форм
- Suspense для ленивой загрузки компонентов
- Error boundaries для обработки ошибок

Качество кода:
- Полная типизация TypeScript
- ESLint + Prettier + Husky pre-commit
- Мобильная адаптивность (Ant Design Grid)
- ARIA-атрибуты для доступности
- Скелетоны при загрузке

Бэкенд (Микросервисы на Node.js)

Создай микросервисную backend-систему на Node.js/Express с тремя сервисами и Nginx Gateway.

Архитектура:
```
backend/
├── gateway/nginx.conf
├── services/
│   ├── users/      # User Service (аутентификация, профили)
│   ├── search/     # Search Service (поиск и сравнение автомобилей)
│   └── chat/       # Chat Service (сессии, сообщения, интеграция с LLM)
├── shared/         # Общие типы, утилиты, JWT middleware
└── docker-compose.yml
```

Технологии:
- Node.js 20+
- Express 5
- TypeScript

- Prisma ORM
- PostgreSQL 16
- Redis 7
- JWT аутентификация
- Zod валидация
- Winston логирование

1. User Service (порт 4001):
   - POST /api/auth/register
   - POST /api/auth/login
   - GET /api/users/profile
   - PUT /api/users/profile
   - Модели: User, UserProfile

2. Search Service (порт 4002):
   - POST /api/search/cars
     Body: { filters: { budget_max, body_type, fuel_type, ... } }
     Response: { models: [...], total: number }
   - POST /api/search/compare
     Body: { model_ids: string[] }
     Response: { comparison: [...] }
   - Использует таблицы: model_specifications, comparison_criteria
   - Кэширует результаты в Redis (TTL 10 минут)

3. Chat Service (порт 4003):
   - POST /api/chat/sessions
   - POST /api/chat/sessions/{id}/messages
   - GET /api/chat/sessions (история пользователя)
   - POST /api/chat/sessions/{id}/save-result
   - Модели: ChatSession, ChatMessage, SearchResult
   - Интегрируется с LLM Orchestrator (POST /api/llm/process)

4. Nginx Gateway (порт 80):
   - Конфиг с проксированием:
     location /api/auth/   → users:4001
     location /api/users/   → users:4001
     location /api/search/ → search:4002
     location /api/chat/   → chat:4003

Требования к коду:
- TypeScript strict mode
- Единый формат ответов: { success, data, error }

- Централизованная обработка ошибок
- Валидация входящих данных через Zod
- Миграции Prisma для каждой БД
- Health check эндпоинты

LLM Orchestrator (Node.js + OpenAI)

Создай LLM Orchestrator сервис на Node.js для интеграции с OpenAI API и backend-сервисами.

Архитектура:
llm-orchestrator/
├── src/
│ ├── tools/ # Определения функций для LLM
│ ├── adapters/ # Адаптеры для OpenAI/Ollama
│ ├── services/ # Клиенты для backend API
│ ├── utils/ # Парсеры, форматтеры
│ └── index.ts # Основной Express сервер
├── docker-compose.yml
└── .env.example

Функционал:

1. Интеграция с LLM:
   - Поддержка OpenAI GPT-4 Turbo (через официальный SDK)
   - Альтернативно: Ollama для локального запуска
   - System prompt с контекстом: "Ты - консультант по подбору автомобилей..."

2. Tools (function calling):
   - search_cars(filters: object): Поиск автомобилей через Search Service
   - compare_models(model_ids: string[]): Сравнение моделей
   - get_user_preferences(user_id: string): Получение предпочтений из профиля
   - save_search_result(session_id: string, result: object): Сохранение результата

3. Логика работы:
   - Принимает POST /api/llm/process { session_id, message }
   - Загружает историю диалога из Redis по session_id
   - Отправляет запрос в LLM с tools
   - Обрабатывает tool calls, делая реальные запросы в backend
   - Формирует финальный ответ с использованием comparison_criteria

- Сохраняет результат в search_results через Chat Service

4. Конфигурация:
- Порт: 8080
- Redis для кэша контекста
- Загрузка comparison_criteria из БД при старте
- Настройка rate limiting для OpenAI API

Требования:
- Асинхронная обработка tool calls
- Логирование всех вызовов LLM
- Поддержка streaming ответов (SSE)
- Graceful degradation при недоступности LLM

Базы данных (Prisma + PostgreSQL + Redis)

Создай конфигурацию баз данных для микросервисов с Prisma ORM.

Структура:
```
db/
├── prisma/
│   ├── schema.user.prisma    # Схема для User Service
│   ├── schema.search.prisma  # Схема для Search Service
│   ├── schema.chat.prisma    # Схема для Chat Service
│   └── migrations/
├── redis.conf
└── docker-compose.yml
```

1. PostgreSQL схемы:

User Service schema:
```
model User {
  id           String    @id @default(uuid())
  email        String    @unique
  passwordHash  String
  name         String?
  avatarUrl    String?
  role         UserRole  @default(USER)
  status       UserStatus @default(ACTIVE)
  profile      UserProfile?
```

```
  chatSessions  ChatSession[]
  searchResults SearchResult[]
}

Search Service schema:
model ModelSpecification {
  id                  String   @id @default(uuid())
  model               Model    @relation(fields: [modelId], references: [id])
  modelId             String
  bodyType            BodyType @relation(fields: [bodyTypeId], references: [id])
  bodyTypeId          String
  fuelType            FuelType @relation(fields: [fuelTypeId], references: [id])
  fuelTypeId          String
  engineVolumeL       Float?
  horsepower          Int?
  fuelConsumptionCombined   Float?
  insuranceCostPerYearRub   Float?
  annualTaxCostRub          Float?
  maintenanceCostPerYearRub Float?
  isActive            Boolean  @default(true)
}

Chat Service schema:
model SearchResult {
  id              String     @id @default(uuid())
  chatSession     ChatSession @relation(fields: [chatSessionId], references: [id])
  chatSessionId   String     @unique
  user            User       @relation(fields: [userId], references: [id])
  userId          String
  searchQuerySummary  String?
  resultData      Json
  isSaved         Boolean    @default(false)
  createdAt       DateTime   @default(now())
}
```

2. Redis конфигурация:
   - Кэш поиска: search:cars:{hash}
   - Контекст чата: chat:{session_id}:history
   - Сессии: session:{user_id}
   - TTL: 10 минут для кэша, 24 часа для сессий

3. Docker Compose:

- PostgreSQL с тремя базами данных (users, search, chat)
- Redis с persistence
- Настройки для Prisma миграций