

# Архитектура MVP бота

 chatgpt.com/c/694f6378-16c0-8329-824f-486e12a7e368

есть база данных с оборудованием -- public.equipment определение -- Drop table --  
DROP TABLE public.equipment; CREATE TABLE public.equipment ( id serial4 NOT  
NULL, url varchar(500) NOT NULL, "name" varchar(300) NOT NULL, category  
varchar(100) NULL, subcategory varchar(100) NULL, brand varchar(100) NULL, region  
varchar(100) NULL, description text NULL, main\_parameters jsonb DEFAULT '{}':jsonb  
NULL, additional\_parameters jsonb DEFAULT '{}':jsonb NULL, price varchar(100)  
NULL, photo\_links jsonb DEFAULT '[]':jsonb NULL, created\_at timestamp DEFAULT  
CURRENT\_TIMESTAMP NULL, updated\_at timestamp DEFAULT  
CURRENT\_TIMESTAMP NULL, embedding public.vector NULL, is\_active bool  
DEFAULT true NOT NULL, search\_vector tsvector NULL, CONSTRAINT  
equipment\_pkey PRIMARY KEY (id), CONSTRAINT equipment\_url\_key UNIQUE (url));  
CREATE INDEX equipment\_brand\_idx ON public.equipment USING btree (brand)  
WHERE (is\_active = true); CREATE INDEX equipment\_category\_idx ON  
public.equipment USING btree (category) WHERE (is\_active = true); CREATE INDEX  
equipment\_embedding\_idx ON public.equipment USING ivfflat (embedding  
vector\_cosine\_ops) WITH (lists='29'); CREATE INDEX equipment\_is\_active\_idx ON  
public.equipment USING btree (is\_active); CREATE INDEX equipment\_region\_idx ON  
public.equipment USING btree (region) WHERE (is\_active = true); CREATE INDEX  
equipment\_search\_vector\_idx ON public.equipment USING gin (search\_vector)  
WHERE (is\_active = true); CREATE INDEX equipment\_subcategory\_idx ON  
public.equipment USING btree (subcategory) WHERE ((is\_active = true) AND  
(subcategory IS NOT NULL)); -- Table Triggers create trigger  
equipment\_search\_vector\_trigger before insert or update on public.equipment for each  
row execute function equipment\_search\_vector\_update(); надо создать бота который  
бы помогала пользователям выбирать оборудование у каждого оборудования есть  
разделы с описанием и параметрами - JSON наименование параметра: значение  
бот должен понимать вопросы и находить необходимые данные в каталоге  
предложи архитектуру MVP как консольного приложения

Отталкиваясь от твоей схемы БД (PostgreSQL + **tsvector** + **embedding vector**) и того, что  
тебе нужен **MVP именно как консольное приложение**, предлагаю архитектуру  
максимально простую, но **правильную с заделом на рост (Telegram / Web / API)**.

Ниже — **реальная MVP-архитектура**, которую можно собрать за несколько дней.

## Цель MVP

Консольный бот, который:

1. Принимает вопрос пользователя в свободной форме

2. Понимает **что именно ищут**:

- тип оборудования
- параметры
- регион / бренд / цена (если есть)

3. Находит релевантное оборудование:

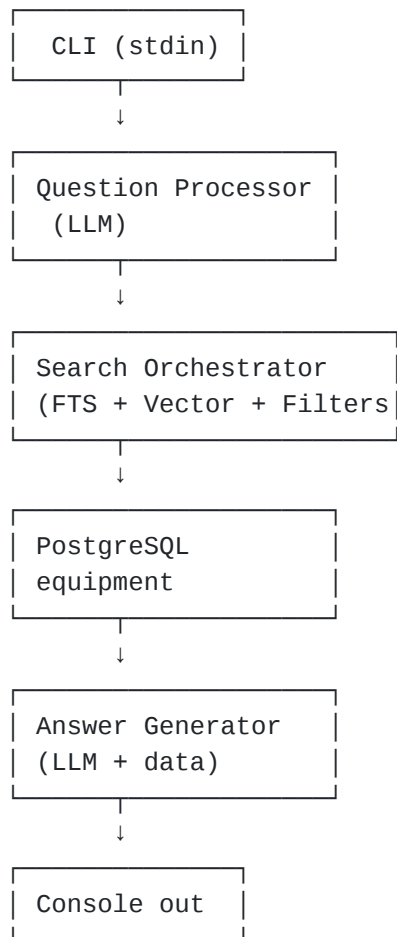
- через **full-text search**
- через **vector search**

4. Возвращает:

- список подходящих вариантов
- краткое описание + ключевые параметры

5. Умеет отвечать на уточняющие вопросы

SCSS



## Компоненты MVP (по слоям)

---

### 1 CLI слой (консоль)

---

#### Задача:

- принимать вопрос
- печатать ответ

bash

> Какой экскаватор подойдёт для работы в карьере до 20 тонн?

### Минимум:

- Node.js / Python
- readline / input()

## 2 Question Processor (понимание запроса)

---

### Задача

---

Преобразовать **человеческий вопрос** → **структурированный запрос**

### Вход:

---

arduino

"Нужен фронтальный погрузчик CAT для работы в Сибири"

## Выход (JSON):

---

json

```
{
  "intent": "equipment_search",
  "category": "погрузчик",
  "subcategory": "фронтальный",
  "brand": "CAT",
  "region": "Сибирь",
  "required_parameters": {
    "работа_в_холоде": true
  },
  "keywords": [
    "фронтальный погрузчик",
    "низкие температуры"
  ]
}
```

## Реализация MVP

---

- LLM (OpenAI / local LLM)
- жёсткий **system prompt**
- Выход строго в JSON

 **Важно:**

LLM не ищет, он только *понимает*.

### Search Orchestrator (поиск)

---

#### Источники поиска

---

##### 1. Full-text search

---

sql

```
SELECT *
FROM equipment
WHERE is_active = true
      AND search_vector @@ plainto_tsquery(:query)
LIMIT 20;
```

##### 2. Vector search

---

sql

```
SELECT *,
       1 - (embedding <=> :query_embedding) AS score
FROM equipment
WHERE is_active = true
ORDER BY embedding <=> :query_embedding
LIMIT 20;
```

### 3. Фильтры

---

sql

```
AND brand = :brand
AND category = :category
AND region = :region
```

### Логика MVP

---

1. Попробовать **FTS + фильтры**
2. Если мало результатов → **Vector search**
3. Объединить результаты
4. Оставить TOP-N (например 5)

### 4 Работа с JSON-параметрами

---

#### Пример параметров:

---

json

```
main_parameters = {
  "мощность_двигателя": "150 л.с.",
  "масса": "18 тонн",
  "глубина_копания": "6.5 м"
}
```

### Фильтрация (MVP — простая)

---

- текстовый поиск по `main_parameters::text`

- позже можно:
  - нормализовать параметры
  - выделить числовые диапазоны

sql

```
AND main_parameters::text ILIKE '%18 тонн%'
```

## 5 Answer Generator (формирование ответа)

---

### Вход:

---

- исходный вопрос
- 3–5 найденных записей оборудования

### Выход:

---

Человекочитаемый ответ:

python-repl

Подходящие варианты:

**1** CAT 320  
Масса: 20 тонн  
Глубина копания: 6.7 м  
Подходит для карьерных работ

**2** Hitachi ZX200  
...



## Реализация:

---

LLM + строгий prompt:

«Отвечай только на основе переданных данных. Не выдумывай.»

## 6 Минимальная структура проекта

---

pgsql

```
equipment-bot/  
├─ src/  
│  ├─ cli.ts  
│  ├─ llm/  
│  │  ├─ parseQuestion.ts  
│  │  └─ generateAnswer.ts  
│  ├─ search/  
│  │  ├─ ftsSearch.ts  
│  │  ├─ vectorSearch.ts  
│  │  └─ orchestrator.ts  
│  ├─ db/  
│  │  └─ pg.ts  
│  └─ types/  
│      └─ query.ts  
├─ prompts/  
│  ├─ question_parser.txt  
│  └─ answer_generator.txt  
└─ README.md
```

## Почему это хороший MVP

---

- ✓ Использует твою текущую БД **без изменений**
- ✓ Уже RAG (не чат ради чата)
- ✓ Можно:
  - обернуть в Telegram-бот
  - сделать REST API
  - добавить RabbitMQ / async индексирование
  - подключить ClickHouse / Supabase / LangChain

## Что можно добавить вторым этапом

---

- диалог с уточняющими вопросами
- сохранение контекста запроса
- scoring параметров
- сравнение оборудования
- рекомендации «лучше / дешевле / мощнее»

Если хочешь, следующим шагом могу:

- 🛠️ написать **system prompt** для парсинга вопроса
- 🖋️ показать **пример кода orchestrator**
- 📊 предложить **диаграмму (D2 / Mermaid)**
- 🤖 адаптировать под Telegram / Web

Скажи, куда двигаемся дальше.

что лучше выбрать python или Nodejs TypeScript

Коротко: **для твоего кейса лучше Node.js + TypeScript.**

Ниже — не абстрактно, а **прямо под твою задачу и твой опыт.**

## Контекст (почему это важно)

---

Ты уже:

- активно работаешь с **NestJS / TypeScript**
- используешь **PostgreSQL, JSONB, vector, tsvector**
- строишь **RAG / ботов / async-архитектуры**
- думаешь про **RabbitMQ, API, Telegram, Web**

Это сильно влияет на выбор.

## Node.js + TypeScript — РЕКОМЕНДУЮ

---

### Плюсы именно для твоего проекта

---

### Ты уже в экосистеме

---

- NestJS

- pg / prisma / typeorm
- WebSocket / Telegram
- RabbitMQ
- monorepo

### → Минимальный когнитивный оверхед

## 2 Отлично подходит для MVP → прод

---

Твой бот:

- консольный сейчас
- API / Telegram позже
- Web-интерфейс потом

Node.js легко масштабируется без переписывания кода.

## 3 LLM / RAG — уже зрелая экосистема

---

- OpenAI SDK
- LangChain.js
- pgvector
- embedding pipelines
- streaming

Python здесь не выигрывает для **production-RAG**.

## 4 TypeScript = контроль структуры

---

Для таких вещей как:

ts

```
ParsedQuery  
SearchIntent  
EquipmentResult
```

TypeScript даёт:

- строгие контракты
- меньше багов в LLM-интеграции
- проще рефакторинг

📌 В RAG-проектах **это критично**.

## 5 Консоль ≠ Python-only

---

Node CLI сейчас:

- readline / inquirer
- commander
- красивые интерактивные CLI

Нет никаких минусов.

## 🟡 Когда Python был бы лучше

---

Python имеет смысл, если бы ты:

- делал **ML / обучение моделей**
- активно использовал **pandas / numpy**
- писал **экспериментальный прототип на 1 вечер**

Но у тебя:

- не ML
- не ресёрч

- а продуктовый бот

## vs Честное сравнение под твой кейс

---

Критерий	Node.js + TS	Python
RAG / LLM прод	✓	⚠
Структура проекта	✓	⚠
PostgreSQL JSONB	✓	✓
CLI	✓	✓
Масштабирование	✓	⚠
Telegram / Web	✓	⚠
Async / очереди	✓	⚠
Твой опыт	✓🔥	✗

## 🧠 Моя рекомендация

---

### 👉 Node.js + TypeScript

👉 Структура сразу как сервис

👉 Консоль — просто первый интерфейс

### 🔧 Практический стек для MVP

---

text

```
Node.js 20+
TypeScript
pg
openai (or compatible)
readline / inquirer
dotenv
```

## 🧩 Если хочешь — следующий шаг

---

Могут:

- предложить **boilerplate** проекта
- написать **CLI entrypoint**
- сделать **типизацию ParsedQuery**
- сравнить **LangChain.js vs чистый SDK**
- показать как потом **без боли добавить Telegram**

Скажи, что именно делаем дальше 👍

openai (or compatible) - будем использовать облачные сервисы?

Короткий ответ: **не обязательно**.

Правильнее — **заложить архитектуру “cloud-agnostic”**, а в MVP *начать с облака*, чтобы быстрее проверить гипотезу.

Ниже — чётко и по делу.

## Принцип, который я предлагаю

---

| LLM = абстракция, а не конкретный провайдер

Ты **не выбираешь OpenAI** навсегда, ты выбираешь **интерфейс**.

## Варианты, которые у тебя есть

---

### Вариант 1 — Облачные LLM (рекомендую для MVP)

---

Примеры:

- OpenAI
- Azure OpenAI
- Mistral Cloud
- Anthropic
- Groq
- Together.ai

### Почему это разумно для старта:

---

- ✓ не надо поднимать GPU
- ✓ стабильное качество понимания запросов
- ✓ минимум инфраструктуры
- ✓ быстрее дойти до работающего бота

 **Важно:** данные оборудования — *не персональные*, риск минимальный.

## **Вариант 2 — Локальные модели (позже / опционально)**

---

Примеры:

- Ollama (Llama, Qwen, Mixtral)
- LM Studio
- vLLM
- Text Generation Inference

Подходит если:

- появятся требования по приватности
- нужен контроль стоимости
- большой трафик

## **Как правильно спроектировать (ключевой момент)**

---

### **Плохо (жёсткая привязка)**

---

```
ts
```

```
import OpenAI from "openai";  
  
const client = new OpenAI({ apiKey });
```

## ✓ Хорошо (абстракция)

---

ts

```
interface LLM {  
  parseQuestion(input: string): Promise<ParsedQuery>;  
  generateAnswer(input: AnswerInput): Promise<string>;  
}
```

### Реализации:

---

text

```
llm/  
├─ openai.llm.ts  
├─ ollama.llm.ts  
├─ mistral.llm.ts  
└─ index.ts
```

### Переключение:

ts

```
const llm = createLLM(process.env.LLM_PROVIDER);
```



## Где именно нужен LLM в твоём MVP

---

Задача	LLM нужен?
Парсинг запроса	✓
Embedding запроса	✓
SQL / фильтры	✗
Поиск в БД	✗
Формирование ответа	✓

 LLM не лезет в БД напрямую.

## Гибридный режим (лучший вариант)

---

text

```
graph TD
    User[Пользователь] --> LLM1[LLM (понимание)]
    LLM1 --> PG[PostgreSQL (поиск)]
    PG --> LLM2[LLM (ответ)]
```

Это и есть **правильный RAG**, а не “чат с базой”.

## Про данные и безопасность

---

Для MVP:

- данные оборудования ≠ персональные
- можно спокойно использовать облако

Для продакшена:

можно:

- анонимизировать
- не передавать raw JSON
- передавать только нужные поля
- перейти на локальную модель



## Рекомендация по стратегии

---

### Этапы:

---

- 1 MVP → **облачный LLM**
- 2 Проверка качества
- 3 Абстракция LLM (сразу!)
- 4 При росте → локальный inference



## Конкретный выбор для старта

---

Если кратко:

- **OpenAI / Azure OpenAI** — самый стабильный
- **Mistral Cloud** — дешевле, хороший EN/RU
- **Ollama** — для локальных экспериментов

Но код пишем так, будто завтра поменяем провайдера.



## Важный момент

---

Не используй:

- LangChain “магией” в MVP
- auto-SQL generation
- agent frameworks



Контроль логики должен быть у тебя.



## Что могу сделать дальше

---

- показать **интерфейс LLM + пример реализации**
- написать **prompt** для question parser
- предложить **env-конфиг** под несколько провайдеров

- помочь выбрать **модель под RU-запросы**

Скажи, что делаем следующим шагом.

мы можем для MVP использовать бесплатные сервисы

Да — для MVP вы вполне можете использовать бесплатные сервисы, если сделать это **осознанно и правильно**. Ниже — **реалистичная схема**, без розовых очков и с учётом твоей задачи.



## Реально бесплатные варианты для MVP

---



### 1. Локальные LLM (лучший “бесплатный” вариант)

---

#### ◆ Ollama (рекомендую №1)

---

- Бесплатно
- Работает локально
- Поддерживает:
  - LLaMA 3.x
  - Qwen 2.5
  - Mistral
- Отлично подходит для:
  - парсинга запросов
  - генерации ответов
  - embedding (некоторые модели)



Минус: требуется нормальная машина (16–32 ГБ RAM).

#### ◆ LM Studio

---

- Бесплатно
- GUI + local API
- Удобно для экспериментов

## 2. Бесплатные cloud-лимиты (ограниченно, но можно)

---

### ♦ Groq

---

- Бесплатный доступ
- Очень быстрый inference
- Подходит для **парсинга вопросов**
- Лимиты есть, но для MVP достаточно

### ♦ Mistral (trial)

---

- Обычно даёт стартовые кредиты
- Хорошо работает с RU/EN

⚠ Это **временное бесплатно**, но ок для проверки гипотезы.

## 3. Embeddings — самое узкое место

---

### Бесплатные варианты:

---

#### 1 Локальные embeddings

- `bge-small`
- `e5-small`
- `text-embedding-3-small` (если есть trial)

#### 2 Ollama embeddings

```
bash
```

```
ollama pull nomic-embed-text
```

📌 Качество ниже, чем у OpenAI, но **для MVP достаточно**.



## Рекомендуемая бесплатная MVP-конфигурация

---



### 100% бесплатно, локально

---

text

LLM: Ollama + Qwen2.5 / LLaMA 3.1  
Embeddings: nomic-embed-text  
DB: PostgreSQL + pgvector  
CLI: Node.js + TypeScript



### Гибрид (ещё бесплатно)

---

text

LLM parsing: Groq (free)  
Embeddings: локально  
Answer generation: локально



## Что важно понимать (честно)

---



### Ограничения бесплатных решений

---

- качество понимания сложных RU-запросов хуже
- иногда “галлюцинации”
- медленнее облака



**Но для MVP выбора оборудования — этого достаточно.**

## Про данные

---

Твоя БД:

- оборудование
- описания
- параметры

Это не персональные данные, значит:

- можно спокойно использовать cloud trial
- можно гибридить

## Архитектурный принцип (ключевой)

---

ts

```
interface LLMProvider {  
  parseQuestion(): Promise<ParsedQuery>;  
  embed(): Promise<number[]>;  
  generateAnswer(): Promise<string>;  
}
```

→ Бесплатно сейчас

→ Платно позже

→ Код не меняется

## Что я бы сделал на твоём месте

---

- 1 Взял **Ollama**
- 2 Поднял **Qwen2.5 7B**
- 3 Embeddings — **nomic-embed-text**
- 4 Сделал CLI MVP
- 5 Проверил реальные вопросы пользователей
- 6 Только потом думал о платных API

## → Следующий логичный шаг

---

Если хочешь, могу:

- предложить конкретную модель под RU
- показать как подключить Ollama к Node.js
- написать prompt для стабильного JSON
- сделать diagram MVP-архитектуры

Скажи, что делаем дальше 👍

Взял Ollama 2 Поднял Qwen2.5 7B 3 Embeddings — nomic-embed-text 4 Сделал CLI MVP 5 Проверил реальные вопросы пользователей 6 Только потом думал о платных API какой нужен компьютер?

Отвечаю максимально приземлённо — **какой компьютер реально нужен, без маркетинга.**

Ты выбрал:

- Ollama
- Qwen2.5 7B
- nomic-embed-text
- CLI MVP

Это хороший и разумный стек 👍

Теперь по железу.

## 🖥 Минимальные и рекомендуемые требования

---

### 🟢 Минимум (чтобы завелось)

---

| ⚠ Работать будет, но медленно

- CPU: 4 ядра (Intel i5 / Ryzen 5)
- RAM: 16 GB
- Диск: SSD, ~15 GB свободного места
- OS: Linux / macOS

Что ожидать:

- ответы 5–15 секунд
- embedding — терпимо
- параллельных запросов — ❌

## 🟡 Комфортный MVP (рекомендую)

---

| ✅ Оптимально для разработки и тестов

- **CPU:** 6–8 ядер
- **RAM:** 32 GB
- **GPU:** необязательно
- **Диск:** SSD NVMe

Что будет:

- ответы 1–4 секунды
- можно параллелить embedding + генерацию
- нормальный dev-цикл

📌 32 GB RAM — самый важный параметр

## 🟢 Если есть GPU (не обязательно)

---

### NVIDIA GPU

---

- **VRAM:** 8 GB (минимум)
- **Лучше:** 12–16 GB

С GPU:

- генерация почти мгновенная
- меньше нагрузка на CPU
- стабильнее при длинных ответах

Но ! GPU не обязателен для MVP.



## Сколько реально ест памяти

---

### Qwen2.5 7B (примерно)

---

Квантование	RAM
Q4_K_M	~5–6 GB
Q5	~7–8 GB
Q8	~10–12 GB

### nomic-embed-text

---

~1.5–2 GB RAM

### Итого

---

- **Минимум:** ~8–10 GB
- **Комфорт:** 14–18 GB
- Node.js + PostgreSQL + OS

➡ 16 GB — впритык

➡ 32 GB — идеально

## Реальные сценарии

---

### Локальная разработка

---

MacBook M1 / M2 с 16 GB

Работает, но лучше 24–32 GB

### Домашний ПК / сервер

---

Ryzen 7 + 32 GB RAM

Идеальный вариант

### VPS (если вдруг)

---

✗ Не рекомендую:

- дорого
- CPU inference медленный

- RAM 32 GB = 🖨️

## 🔧 Настройки Ollama (важно)

---

Пример:

```
bash
```

```
ollama run qwen2.5:7b-instruct-q4_K_M
```

Для MVP не бери Q8 — лишняя память.

## ⚠️ Честные ограничения

---

- несколько одновременных пользователей → ❌
- длинные диалоги → медленно
- сложные reasoning-запросы → иногда тупит

Но:

| Для выбора оборудования — более чем достаточно

## 🧠 Моя рекомендация

---

Если кратко:

---

- RAM: 32 GB
- CPU: 6–8 ядер
- GPU: опционально
- SSD обязателен

## ➡ Следующий шаг

---

Могут:

- подобрать конкретную модель под твоё железо

- дать **оптимальные параметры Ollama**
- показать **как ограничить контекст, чтобы не жрало RAM**
- помочь собрать **Docker-версию**

Скажи, на чём ты собираешься запускать (ПК / ноут / сервер).

А какой нужен будет VPS или сервер для орагнизация такого проекта в продакшн как сервиса Поиск по каталогу

Отвечу **как для реального продакшена**, а не “теоретически”.

Будем считать, что это **сервис поиска по каталогу оборудования (RAG)**, не чат-игрушка.

## Что именно нужно в проде

---

В продакшене у тебя есть **4 разных нагрузки**, и их **НЕ НУЖНО** решать одной машиной.

- 1 API / бот / CLI
- 2 PostgreSQL (поиск + JSONB + vector)
- 3 LLM (понимание запроса + ответ)
- 4 Embeddings / индексация (редко, но тяжело)

👉 **Главное правило:**  
**LLM ≠ БД ≠ API**



## Рекомендуемая прод-архитектура (минимум)

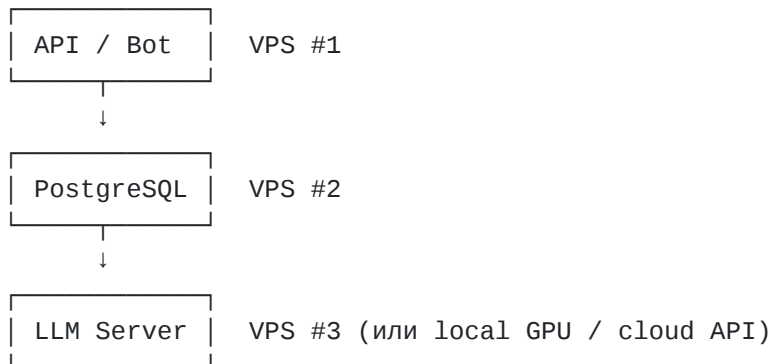
---



### MVP-прод (1–10 запросов / сек)

---

bash



## Конкретные характеристики серверов

---



### VPS #1 — API / Search Orchestrator

---

#### Нагрузка:

- HTTP
- LLM calls
- SQL запросы
- лёгкий CPU

#### Рекомендуемо:

---

- **CPU:** 2–4 vCPU
- **RAM:** 4–8 GB
- **Disk:** 20–40 GB SSD



Подойдёт любой VPS:

- Hetzner
- DigitalOcean
- Selectel
- OVH

## VPS #2 — PostgreSQL + pgvector

---

Это самый важный сервер после LLM

### Минимум:


---

- **CPU:** 4 vCPU
- **RAM:** 16 GB
- **Disk:** NVMe SSD
- **IOPS:** высокий

### Почему:

---

- vector search жрёт RAM
- JSONB + GIN + IVFFLAT
- кеш сильно влияет на скорость

 Если база > 1–2 млн записей → 32 GB RAM.

## VPS #3 — LLM (если локально)

---

### Обычный VPS без GPU

---

Не рекомендую:

- медленно
- дорого
- нестабильно

## Варианты LLM в проде

---

### Вариант А — Cloud LLM (рекомендую для начала)

---

- OpenAI / Mistral / Groq

- Не нужен LLM сервер
- Цена: от \$20–100 / мес

👉 Лучшее соотношение цена / качество

## Вариант В — Собственный GPU-сервер

---

### Минимум:

---

- **GPU:** 16 GB VRAM (RTX 4060 / A4000)
- **RAM:** 64 GB
- **CPU:** 8 ядер

### Когда имеет смысл:

---

- | 1000 запросов в день
- нужен полный контроль
- фиксированная нагрузка

📌 Это уже **не VPS**, а dedicated server.

## 🟡 Embeddings / индексация

---

👉 НЕ в real-time

### Правильно:

---

- отдельный worker
- очередь (RabbitMQ / BullMQ)
- запуск по cron / событию

### Сервер:

---

- можно использовать тот же, что API
- или временно масштабировать

## 💰 Пример реального бюджета (MVP)

---

Компонент	Цена / мес
API VPS	€10–20

---

Компонент	Цена / мес
PostgreSQL VPS	€25–40
Cloud LLM	€20–80
<b>Итого</b>	<b>€60–140</b>

## Частые ошибки

---

- ✗ Один сервер “на всё”
- ✗ PostgreSQL + LLM вместе
- ✗ CPU-only inference в проде
- ✗ embeddings в запросе пользователя

## Моя рекомендация (чётко)

---

### Для старта:

---

- API + поиск → VPS
- PostgreSQL → отдельный VPS
- LLM → **облако**
- Embeddings → async worker

### Потом:

---

- LLM → GPU сервер
- PostgreSQL → реплика / read-only
- Cache → Redis

## Короткий чеклист

---

- ✓ БД отдельно
- ✓ LLM отдельно
- ✓ async индексация
- ✓ без агентов
- ✓ без магии

## Следующий шаг

---

Могут:

- нарисовать **prod-архитектуру (D2 / Mermaid)**

- рассчитать **нагрузку** под **X** пользователей
- подобрать **конкретные тарифы** Hetzner / DO
- показать **docker-compose** для MVP-прода

Скажи, что хочешь следующим.

Есть ли смысл добавить в это проект MCP сервер для каталога

Коротко: **для MVP — нет, для развития экосистемы и B2B — да, смысл есть.**  
Но не как “ещё один сервер”, а как **стратегический слой доступа к каталогу.**

Разберём без хайпа.



## Что такое MCP в твоём контексте (по-честному)

---

Для тебя **MCP-сервер каталога** — это:

стандартизованный способ дать ИИ (и другим агентам) **контролируемый доступ** к каталогу оборудования  
**без прямого доступа к БД и API**

Не “умнее поиск”, а **другой уровень интеграции.**



## Текущая архитектура (без MCP)

---

Сейчас у тебя:

scss

```
User
↓
LLM (parse)
↓
Search Orchestrator
↓
PostgreSQL
↓
LLM (answer)
```

Это **идеально для пользовательского поиска.**



## Где MCP реально добавляет ценность

---

### MCP имеет смысл, если ты хочешь:

---

#### 1 Дать каталог внешним AI

---




Примеры:

- корпоративные AI-ассистенты
- Copilot-подобные системы
- агентные системы клиентов
- будущие marketplace-интеграции

 MCP = “**AI API**”, а не REST API.

#### 2 Разделить ответственность

---

-  Каталог живёт сам по себе
-  AI не знает SQL
-  MCP описывает *что можно*, а не *как искать*

#### 3 Подготовиться к агентным сценариям

---

Например:

- “Подбери технику + сравни + объясни почему”
- “Найди аналоги”
- “Подготовь коммерческое предложение”

 MCP — естественная точка входа для агентов.

## Когда MCP НЕ нужен

---

Для MVP:

- один бот
- один поиск
- один каталог
- контролируемый код

➡ MCP добавит **сложность без пользы**

**Если:**

---

- нет внешних интеграций
- нет multiple AI consumers
- нет агентных сценариев

👉 MCP будет **оверинжинирингом**

🟡 **Правильная стратегия (рекомендую)**

---

## Этап 1 — MVP (СЕЙЧАС)

---

- ❌ MCP не добавляем
- ✓ Оставляем **внутренний Search API**

## Этап 2 — Подготовка (очень важно)

---

- ✓ Проектируем каталог как **отдельный bounded context**

ts

```
CatalogService
├─ search()
├─ getById()
├─ compare()
└─ explainChoice()
```

👉 Уже сейчас **НЕ** привязываемся к UI / CLI.

## Этап 3 — MCP (когда появится смысл)

---

- ✓ Добавляем MCP-сервер поверх CatalogService

arduino

```
MCP Server
↓
CatalogService
↓
PostgreSQL
```

👉 Код поиска не переписывается

## 🧱 Как должен выглядеть MCP для каталога

---

### MCP tools (пример)

---

json

```
{
  "tools": [
    {
      "name": "search_equipment",
      "description": "Search equipment catalog with filters",
      "input_schema": {
        "category": "string",
        "brand": "string",
        "parameters": "object"
      }
    },
    {
      "name": "get_equipment",
      "input_schema": { "id": "number" }
    }
  ]
}
```

📌 MCP НЕ должен:

- принимать сырой SQL
- возвращать всю БД
- быть “умным”

## Ключевая мысль

---

| MCP — это не ускоритель MVP, а мультипликатор экосистемы

## Краткая таблица решения

---

Сценарий	MCP нужен
CLI MVP	✗
Telegram-бот	✗
Один AI	✗
Несколько AI	✓
B2B интеграции	✓
Агентные сценарии	✓
Marketplace	✓

## Моя рекомендация лично тебе

---

- 👉 Не добавляй MCP сейчас
- 👉 Спроектируй код так, чтобы MCP добавить без боли
- 👉 Вернись к MCP, когда появится второй consumer

## Могу помочь дальше

---

Если хочешь, могу:

- показать как спроектировать CatalogService “MCP-ready”
- набросать MCP schema именно под equipment
- сравнить MCP vs REST vs GraphQL для твоего кейса
- предложить эволюционный roadmap

Скажи, куда идём дальше.

показать как спроектировать CatalogService “MCP-ready”

Отличный вопрос. Ниже — **практическая схема**, как спроектировать **CatalogService** так, чтобы:

- сейчас он работал для **CLI / API / Telegram**
- **без переписывания кода** поверх него появился MCP-сервер
- поиск, бизнес-логика и БД были чётко разделены

Буду показывать на **Node.js + TypeScript**, как мы и обсуждали.

## Ключевая идея

---

**CatalogService** — это доменный слой

Он ничего не знает про:

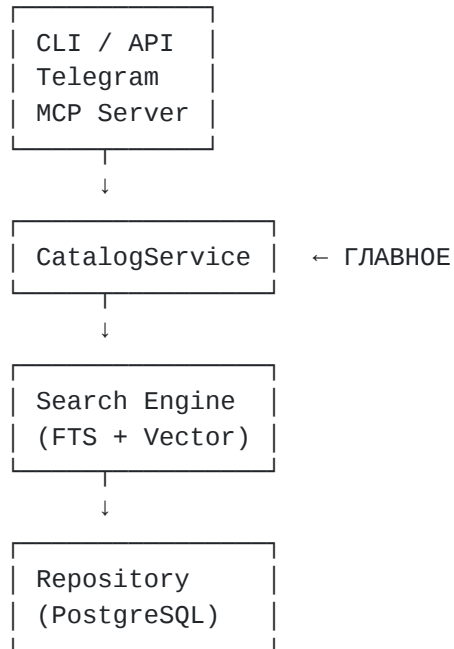
- CLI
- HTTP
- MCP
- LLM
- PostgreSQL конкретно

Он знает только:

- **что такое оборудование**
- **как его искать**
- **какие операции разрешены**



pgsql



## Рекомендуемая структура проекта

---

text

```
src/
├─ catalog/
│   ├─ catalog.service.ts
│   ├─ catalog.types.ts
│   ├─ catalog.errors.ts
│   ├─ catalog.policy.ts
│   └─ index.ts
├─ search/
│   ├─ search.engine.ts
│   ├─ fts.strategy.ts
│   ├─ vector.strategy.ts
│   └─ index.ts
├─ repository/
│   ├─ equipment.repository.ts
│   └─ index.ts
├─ msp/           ← появится позже
├─ api/
└─ cli/
```

## 1. Контракты (самое важное)

---

**catalog.types.ts**

---

ts

```
export type EquipmentSummary = {
  id: number;
  name: string;
  category?: string;
  brand?: string;
  price?: string;
  mainParameters: Record<string, string>;
};

export type EquipmentDetails = EquipmentSummary & {
  description?: string;
  additionalParameters?: Record<string, string>;
  photos?: string[];
};

export type SearchQuery = {
  text?: string;
  category?: string;
  subcategory?: string;
  brand?: string;
  region?: string;
  parameters?: Record<string, string | number>;
  limit?: number;
};

export type SearchResult = {
  items: EquipmentSummary[];
  total: number;
};
```

### Важно:

- MCP потом будет использовать **эти же типы**
- ты контролируешь **что можно вернуть наружу**



## 2. CatalogService (ядро)

---

### catalog.service.ts

---

ts

```
export class CatalogService {
  constructor(
    private readonly searchEngine: SearchEngine,
    private readonly repository: EquipmentRepository
  ) {}

  async search(query: SearchQuery): Promise<SearchResult> {
    return this.searchEngine.search(query);
  }

  async getById(id: number): Promise<EquipmentDetails | null> {
    return this.repository.findById(id);
  }

  async compare(ids: number[]): Promise<EquipmentSummary[]> {
    return this.repository.findMany(ids);
  }
}
```



- **никакого SQL**
- **никакого MCP**
- **никакого LLM**

### 3. Search Engine (стратегии поиска)

---

**search.engine.ts**

---

ts

```
export class SearchEngine {
  constructor(
    private readonly fts: FTSSearchStrategy,
    private readonly vector: VectorSearchStrategy
  ) {}

  async search(query: SearchQuery): Promise<SearchResult> {
    if (query.text) {
      const ftsResult = await this.fts.search(query);
      if (ftsResult.items.length >= 3) return ftsResult;
    }
    return this.vector.search(query);
  }
}
```

 **Логика понятная и управляемая**

## 4. Repository (PostgreSQL изолирован)

---

**equipment.repository.ts**

---

ts

```
export class EquipmentRepository {
  async findById(id: number) {
    // SQL тут
  }

  async findMany(ids: number[]) {
    // SQL тут
  }

  async ftsSearch(query: SearchQuery) {
    // tsvector
  }

  async vectorSearch(query: SearchQuery) {
    // pgvector
  }
}
```

 **МСП никогда** не увидит SQL

## 5. Как MCP встанет поверх (в будущем)

---

### MCP Server (позже)

---

CSS

```
MCP tool: search_equipment
      ↓
CatalogService.search()
      ↓
SearchEngine
      ↓
Repository
```

### MCP tool schema (пример)

---

json

```
{
  "name": "search_equipment",
  "input_schema": {
    "text": "string",
    "category": "string",
    "brand": "string"
  }
}
```

 MCP просто **мапит вход** → **CatalogService**

### **Безопасность и контроль**

---

Ты **сразу** выигрываешь:

- MCP не может:
  - делать JOIN
  - читать embedding
  - менять данные
- MCP может только:
  - `search`
  - `getById`
  - `compare`

## Почему это “MCP-ready”

---

- ✓ чистые контракты
- ✓ один источник бизнес-логики
- ✓ MCP — thin layer
- ✓ можно добавить:

- REST
- GraphQL
- MCP
- gRPC

без переписывания ядра

## Что я рекомендую сделать прямо сейчас

---

- 1 Реализовать `CatalogService`
- 2 Использовать его в CLI
- 3 **Не писать MCP**
- 4 Когда появится второй consumer → добавить MCP за 1 день

## Могу продолжить

---

Хочешь:

- пример **MCP server implementation**
- конкретные **SQL** для **repository**
- стратегию **rate limiting** для MCP

- diagram (Mermaid / D2)

Скажи, что дальше.

давай на основании этой архитектуры сделаем проект MVP для демонстрации принципа поиска в каталоге при обычных человеческих запросах. Возможно ли это без сторонних платных сервисов

Да, это полностью возможно без сторонних платных сервисов.

И более того — это правильный путь для демонстрационного MVP.

Ниже — конкретный, выполнимый план, без облаков, без подписок, на том, что ты уже выбрал.

## Что мы делаем

---

**MVP-проект**, который демонстрирует:

- поиск по каталогу оборудования
- понимание *человеческих* запросов
- RAG-подход (LLM не ходит в БД)
- MCP-ready архитектуру
- полностью **локальный стек**




## Технологический стек (100% бесплатно)

---

### Основа

---

- Node.js + TypeScript
- PostgreSQL + pgvector
- Ollama
  - `qwen2.5:7b-instruct-q4_K_M`
  - `nomic-embed-text`
- CLI (stdin/stdout)

 OpenAI  
 облака  
 платные API



## Архитектура MVP (реалистично)

---

SCSS

User (CLI)



QuestionParser (LLM)



CatalogService



SearchEngine



FTS (tsvector)



Vector Search (pgvector)



PostgreSQL



AnswerGenerator (LLM)



Console output

text

```
equipment-search-mvp/
├─ src/
│  ├─ cli/
│  │   └─ index.ts
│  │
│  ├─ llm/
│  │   ├─ ollama.client.ts
│  │   ├─ question.parser.ts
│  │   └─ answer.generator.ts
│  │
│  ├─ catalog/
│  │   ├─ catalog.service.ts
│  │   ├─ catalog.types.ts
│  │   └─ index.ts
│  │
│  ├─ search/
│  │   ├─ search.engine.ts
│  │   ├─ fts.strategy.ts
│  │   ├─ vector.strategy.ts
│  │   └─ index.ts
│  │
│  ├─ repository/
│  │   └─ equipment.repository.ts
│  │
│  └─ db/
│      └─ pg.ts
│
├─ prompts/
│   ├─ question_parser.txt
│   └─ answer_generator.txt
│
├─ docker-compose.yml
├─ package.json
└─ README.md
```

## Как LLM используется (строго)

---

### 1 QuestionParser (понимание запроса)

---

Вход:



Нужен гусеничный экскаватор для карьера, до 25 тонн

## Выход:

json

```
{
  "text": "гусеничный экскаватор карьер",
  "category": "экскаватор",
  "subcategory": "гусеничный",
  "parameters": {
    "масса": "до 25 тонн"
  }
}
```

 Только структура. Никакой логики поиска.

## 2 SearchEngine (чистая логика)

---

- FTS → если есть текст
- Vector → если мало результатов
- Фильтры → category / brand / region

## 3 AnswerGenerator (формирование ответа)

---

Получает:

- исходный вопрос
- найденные записи оборудования

Возвращает:

- краткий, человеческий ответ
- строго на основе данных

## Работа с БД (минимум)

---

### Что уже есть у тебя:

---

- `tsvector`
- `embedding`
- JSONB параметры

### Что нужно добавить:

---

- периодическую генерацию `embedding`
- CLI-команду `reindex`

## Демонстрационные сценарии

---

text

- > Нужен экскаватор для карьера до 25 тонн
- > Есть ли техника CAT для холодных регионов
- > Подбери погрузчик дешевле 5 млн
- > Какие варианты есть для аренды в Сибири

 Этого достаточно, чтобы **показать ценность**.

## Ограничения (честно)

---

- не идеально понимает сложные формулировки
- не делает reasoning
- иногда просит уточнение

Но:

| для демонстрации принципа — идеально

## Почему это хорошая демонстрация

---

- ✓ LLM используется правильно
- ✓ Нет vendor lock-in
- ✓ Архитектура масштабируемая
- ✓ MCP-ready
- ✓ Понятна бизнесу

## Что предлагаю дальше (по шагам)

---

### Шаг 1

---

Сделать **CLI + QuestionParser**

### Шаг 2

---

Подключить **CatalogService + SearchEngine**

### Шаг 3

---

Добавить **AnswerGenerator**

### Шаг 4

---

Прогнать реальные вопросы

## Готов продолжать

---

Я могу:

- начать с **конкретного кода (шаг 1)**
- написать **prompts**
- собрать **docker-compose**
- помочь с **SQL + индексацией**
- сделать **README** для демо

 Скажи, с **какого шага начинаем**.

у меня есть cursor где я буду вести разработку давай сделаем ТЗ которое отправлю как промпт ему что посоветуешь выбрать из агентов GPT 5 или другое?

Отлично, это правильный момент **зафиксировать ТЗ и отдать его Cursor** как **“ведущему исполнителю”**.

Отвечу в двух частях:

- 1 какого агента выбрать в Cursor
- 2 готовое ТЗ-промт, которое можно сразу вставлять

## 1 Какого агента выбрать в Cursor

---

### 🏆 Рекомендация: GPT-5 (reasoning / default)

---

#### Почему именно он для этого проекта

---

Твоя задача — не просто писать код, а:

- держать архитектуру в голове
- не ломать границы слоёв
- аккуратно работать с типами
- не “галлюцинировать” бизнес-логику

GPT-5 лучше всего:

- следует **длинному ТЗ**
- удерживает **архитектурные ограничения**
- аккуратно пишет TypeScript
- не превращает всё в “магический агент”

👉 Идеален для backend / архитектурного MVP

### 🟡 Когда можно переключаться на другой агент

---

Агент	Когда использовать
GPT-5	архитектура, сервисы, доменные слои
GPT-5 fast	рефакторинг, мелкие файлы
GPT-4.1	если нужен более “осторожный” код
Claude	иногда для prompt-инжиниринга
o3 / reasoning	если Cursor тупит на логике

📌 Но стартуй именно с GPT-5.

## 2 Готовое ТЗ (PROMPT ДЛЯ CURSOR)

---

Ниже — готовый промпт, его можно целиком скопировать в Cursor.

# PROMPT: Техническое задание для MVP

---

markdown

Ты – senior backend developer.

Твоя задача – реализовать MVP проекта поиска по каталогу оборудования.

 ВАЖНО:

- НЕ использовать платные внешние сервисы
- НЕ использовать OpenAI API
- Использовать только локальные сервисы
- Строго следовать архитектуре ниже
- Не упрощать архитектуру “ради скорости”

-----  
 Цель проекта  
-----





Создать демонстрационный MVP консольного приложения, которое:


- принимает обычные человеческие запросы (RU)
- понимает намерение пользователя
- ищет оборудование в PostgreSQL каталоге
- возвращает релевантные результаты
- демонстрирует принцип RAG (LLM не ходит в БД)

Проект должен быть MCP-ready (архитектурно), но MCP НЕ реализовывать.

-----  
 Технологический стек  
-----

- Node.js 20+
- TypeScript
- PostgreSQL + pgvector
- Ollama (локально)
  - LLM: qwen2.5:7b-instruct-q4\_K\_M
  - Embeddings: nomic-embed-text
- CLI (stdin/stdout)

-  OpenAI
-  облачные API
-  LangChain
-  agent frameworks

-----  
 Структура проекта (обязательная)  
-----

src/

```
├─ cli/
│   └─ index.ts
├─ llm/
│   ├── ollama.client.ts
│   ├── question.parser.ts
│   └─ answer.generator.ts
├─ catalog/
│   ├── catalog.service.ts
│   ├── catalog.types.ts
│   └─ index.ts
├─ search/
│   ├── search.engine.ts
│   ├── fts.strategy.ts
│   ├── vector.strategy.ts
│   └─ index.ts
├─ repository/
│   └─ equipment.repository.ts
└─ db/
    └─ pg.ts
```

## 🧠 Архитектурные принципы (ОБЯЗАТЕЛЬНО)

1. CatalogService – доменный слой
  - НЕ знает про CLI
  - НЕ знает про MCP
  - НЕ знает про LLM
  - НЕ содержит SQL
2. LLM используется ТОЛЬКО для:
  - парсинга пользовательского запроса → structured query
  - генерации ответа по найденным данным
3. Поиск:
  - FTS (tsvector) – основной
  - Vector search (pgvector) – fallback
  - Фильтры: category / brand / region
4. MCP НЕ реализовывать, но:
  - все типы и методы должны быть MCP-ready
  - CatalogService должен быть пригоден для MCP слоя

## 🌱 Основные сущности

SearchQuery:

- text?: string
- category?: string
- subcategory?: string

- brand?: string
- region?: string
- parameters?: Record<string, string | number>
- limit?: number

EquipmentSummary:

- id
- name
- category
- brand
- price
- mainParameters

### Логика поиска

1. Пользователь вводит текст
2. QuestionParser (LLM) возвращает SearchQuery (JSON)
3. SearchEngine:
  - пробует FTS
  - если результатов < 3 → vector search
4. Результаты передаются в AnswerGenerator
5. Ответ выводится в консоль

### CLI

Пример:

> Нужен гусеничный экскаватор для карьера до 25 тонн

Ответ:

- краткий список оборудования
- основные параметры
- без выдуманных данных

### Ограничения

- LLM НЕ имеет прямого доступа к БД
- НЕ использовать ORM
- НЕ генерировать SQL через LLM
- Код должен быть читаемым и модульным
- Минимум магии

### Результат

- Рабочий CLI MVP
- Чёткая архитектура
- Возможность легко добавить:
  - MCP сервер
  - HTTP API



- Telegram-бот

Начинай с:

- 1) создания структуры проекта
- 2) реализации `CatalogService` и типов
- 3) `CLI` + `QuestionParser`

## Как работать с Cursor дальше (совет)

---

### Лучший режим:

---

1. Вставил ТЗ целиком

2. Попросил:

| “Сначала создай только структуру проекта и пустые файлы”

3. Потом:

| “Теперь реализуй `CatalogService` и типы”

4. Потом:

| “Теперь `CLI` и `QuestionParser`”

👉 Не проси “сделай всё сразу”

### ➡ Могу помочь дальше

---

Если хочешь:

- разбить ТЗ на **итерации для Cursor**
- проверить первый коммит Cursor
- адаптировать ТЗ под **demo README**
- подготовить **follow-up prompt для MCP**

Скажи, как будешь запускать Cursor — **по файлам или целиком**.

стоит ли использовать GPT-5.2 в cursor

Для твоего кейса **GPT-5.2 в Cursor использовать можно, но не обязательно**. Давай разберём детально:

## 1 Отличие GPT-5 от GPT-5.2

Параметр	GPT-5	GPT-5.2
Память контекста	длинная	длиннее, немного стабильнее
Reasoning	сильный	чуть лучше, меньше ошибок на сложной логике
Code generation	очень хороший	чуть более аккуратный
Стоимость / ресурсы	стандарт	может чуть выше в облаке

**Вывод:** GPT-5.2 — это скорее **тюнинг стабильности и контекста**, но не “революция” в функционале.

## 2 Когда GPT-5.2 имеет смысл

- если планируешь **очень длинные цепочки задач** (например: parsing → multiple searches → answer formatting → multiple users)
- если нужно **чётко удерживать архитектурные границы** (MCP-ready, типы TypeScript)
- если хочешь минимизировать “галлюцинации” на сложных JSON-структурах

## 3 Когда хватит GPT-5

- MVP для демонстрации принципа поиска по каталогу
- Основные сценарии CLI / поиск + LLM
- Нет длинных цепочек reasoning / несколько агентов
- Экономия на ресурсах

## 4 Моя рекомендация для тебя

- Для MVP: хватит GPT-5
- Для production-ready версии / больших цепочек / агентного режима: GPT-5.2

Практически: начинаешь с GPT-5, когда проект “доживёт” до полноценной интеграции нескольких агентов или сложного reasoning — апгрейд до GPT-5.2.

Если хочешь, я могу сделать **рекомендованный setup Cursor для твоего MVP**, где покажу:

- какой агент выбрать
- как разделить parsing / search / answer

- как минимизировать потребление ресурсов

Хочешь, чтобы я это сделал?

какие лучше использовать

Судя по скриншоту, у тебя включено много вариантов GPT и Codex. Для **MVP поиска по каталогу с человеческими запросами** я бы рекомендовал использовать **минимально необходимое**, чтобы не перегружать Cursor и ресурсы.

## Рекомендации

### Основной агент для логики и reasoning

**GPT-5** — для всего, где нужна обработка естественного языка, генерация JSON для поиска, формирование ответа

✓ Это твой главный агент для QuestionParser и AnswerGenerator

### Для кода / TypeScript / архитектура

**GPT-5.1 Codex Max** — если хочешь, чтобы Cursor писал чистый TS-код, сервисы, типы, методы

✓ Использовать его для CatalogService, SearchEngine, Repository

### Не обязательно включать

- GPT-5.2 — можно позже для долгих цепочек reasoning, сейчас излишне
- Codex Mini / Low / High / Fast — лишние варианты, они только нагружают интерфейс
- Sonnet / Gemini / Grok Code / Opus — для твоего MVP не нужны

## Итоговая настройка для MVP

Агент	Использовать?	Для чего
GPT-5	✓	Основной reasoning, JSON parsing, ответы пользователю
GPT-5.1 Codex Max	✓	Генерация TypeScript кода, сервисов, CLI
GPT-5.2	✗ (опционально позже)	Для сложных цепочек и больших контекстов
Остальные	✗	Не нужны для MVP

Если хочешь, я могу составить **конкретный setup в Cursor**, где указаны роли агентов для каждого слоя архитектуры MVP, чтобы сразу запускать проект без лишних переключений.

Хочешь, чтобы я это сделал?

да

Отлично! Ниже предлагаю **готовый setup для Cursor** под твой MVP проекта поиска по каталогу. Он учитывает архитектуру, роли агентов и минимизирует лишние переключения.

## Рекомендованный setup Cursor для MVP

---

### 1 Основной агент: GPT-5

---

**Роль:**

- QuestionParser: преобразует человеческий запрос в JSON (**SearchQuery**)
- AnswerGenerator: формирует читаемый ответ на основе данных
- Общий reasoning для поиска, подсказки по бизнес-логике

**Использование:**

- Вставлять текст запроса → JSON для CatalogService
- Генерация кратких текстовых ответов
- Поддержка русского языка

### 2 Агент кода: GPT-5.1 Codex Max

---

**Роль:**

- Реализация **CatalogService, SearchEngine, Repository**
- Генерация TypeScript-кода, типов, интерфейсов
- Создание CLI, подключения к PostgreSQL, pgvector

**Использование:**

- Создание структуры проекта
- Реализация функций поиска
- Формирование JSON, TypeScript типы
- Модульные файлы, читаемый код

### 3 Не использовать на MVP:

- GPT-5.2 (пока слишком тяжёлый, нужен только для сложных цепочек reasoning)
- Sonnet 4.5 / Gemini 3 Flash / Grok Code / Opus 4.5
- Codex Mini / Low / High / Fast

Эти варианты только увеличат количество агентов, усложнят workflow, но не принесут пользы для MVP.



### 4 Роли агентов по слоям архитектуры

Слой	Агент	Роль
CLI	GPT-5	Приём вопроса, пересылка в QuestionParser
QuestionParser	GPT-5	Парсинг естественного языка в <b>SearchQuery</b>
CatalogService	GPT-5.1 Codex Max	TypeScript код сервисов и методов поиска
SearchEngine	GPT-5.1 Codex Max	Код FTS / Vector стратегии
Repository	GPT-5.1 Codex Max	SQL + pgvector, изолированная реализация
AnswerGenerator	GPT-5	Формирование текста ответа на основе найденных данных

### 5 Workflow Cursor (MVP)

1. User вводит запрос в CLI
2. GPT-5 (QuestionParser) → **SearchQuery**
3. CatalogService (Codex Max) → выполняет поиск через SearchEngine / Repository
4. GPT-5 (AnswerGenerator) → возвращает понятный результат
5. CLI выводит ответ пользователю

### 6 Настройки агентов в Cursor

- Включить:
  - GPT-5 
  - GPT-5.1 Codex Max 
- Отключить всё остальное

Если хочешь, я могу сразу сделать **готовый промпт для Cursor**, который ты вставляешь, и он настроит все роли агентов + порядок работы для MVP.

Хочешь, чтобы я это сделал?

посмотри в начале диалога MCP сервер - как он может улучшить проект

Отличный вопрос 👍

Ты очень вовремя к нему вернулся — **MCP здесь реально может дать архитектурное преимущество**, даже в MVP.

Я отвечу **строго в контексте начала диалога**:

каталог оборудования → поиск по БД → Ollama / локальные модели → бесплатный MVP  
→ CLI → дальше сервис.



## Что такое MCP в твоём проекте (практически)

---

**MCP (Model Context Protocol)** — это **стандартизированный сервер**, который:

- описывает **возможности каталога** (tools)
- даёт LLM **строгий интерфейс доступа к данным**
- отделяет *мышление модели* от *бизнес-логики*

👉 MCP ≠ “ещё один слой”

👉 MCP = **контракт между LLM и каталогом**

## ? Зачем MCP именно здесь

---

### Твоя текущая задача

---

“Бот должен понимать человеческие вопросы и находить данные в каталоге (PostgreSQL + JSONB + embeddings)”

### Без MCP

---

LLM:

- сам “придумывает”, как искать
- начинает галлюцинировать поля
- сложно масштабировать (UI, Telegram, Web, API)
- сложно заменить модель

## С MCP

---

LLM:

- видит только разрешённые операции
- не знает SQL
- не знает схему БД
- не может сделать “лишнее”



## Как MCP улучшает проект концептуально

---

### 1 Чёткая граница ответственности

---

nginx

LLM — думает  
MCP — знает  
CatalogService — ищет  
PostgreSQL — хранит

LLM не ищет, он вызывает tool.

### 2 MCP идеально ложится на каталог

---

Твой каталог — это **read-heavy, structured, deterministic system**  
Это идеальный кейс для MCP.

Пример tool'а:

ts

```
search_equipment(query: SearchQuery): Equipment[]
```

LLM не формирует SQL,  
LLM не знает pgvector,  
LLM только заполняет SearchQuery.

### 3 MCP = защита от галлюцинаций

---

LLM не может:

- выдумать параметр (`max_load_kg`, если его нет)
- выдумать категорию
- “додумать” цену

Если MCP не вернул данных → LLM честно говорит:

| «В каталоге нет такого оборудования»

### 4 Бесплатные модели + MCP = 🔥

---

Ты уже выбрал:

- Ollama
- Qwen 2.5 7B
- nomic-embed-text

**!** Это не самые умные модели,  
НО с MCP они становятся управляемыми.

Без MCP:

| “7B плохо рассуждает”

С MCP:

| “7B отлично заполняет JSON и выбирает tool”



## 5 МСР делает MVP «инвесторским»

---

Даже если MVP — CLI:

- архитектура сразу:
  - AI-agnostic
  - model-agnostic
  - channel-agnostic (CLI → Web → TG)
- легко объяснить:
  - | «LLM — только интерфейс, бизнес-логика отдельно»

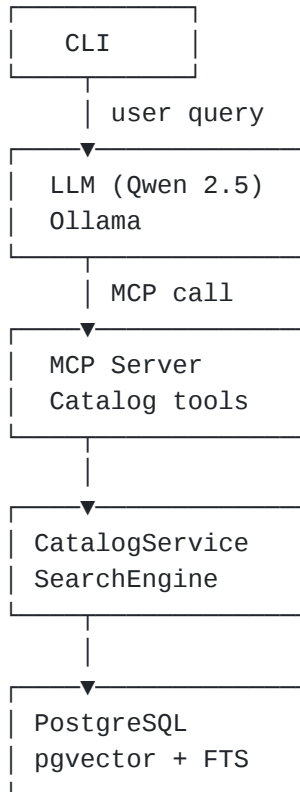
Это огромный плюс.



## Минимальная MCP-архитектура для твоего MVP

---

pgsql



## Какие tools нужны на MVP

---

Минимум 3 tool'a:

## 1 search\_equipment

---

json

```
{
  "query": "экскаватор для аренды в Москве",
  "filters": {
    "category": "excavator",
    "region": "Москва"
  }
}
```

## 2 get\_equipment\_by\_id

---

json

```
{
  "id": 123
}
```

## 3 list\_available\_filters

---

json

```
{}
```

👉 Это позволяет LLM:

- уточнять параметры
- задавать вопросы пользователю
- не гадать

## ? Стоит ли добавлять MCP в MVP?

---

Коротко:

---

**ДА**, если ты делаешь проект “как сервис”, а не демку.

**Если без MCP:**

---

- быстрее старт (–1 день)
- хуже масштабирование
- сложнее контроль LLM

**С MCP:**

---

- +1 день разработки
- +10x архитектурной чистоты
- готовность к росту

## 🔥 Мой честный совет

---

👉 **Делай MVP сразу MCP-ready**, даже если:

- MCP сервер сначала просто thin-wrapper над CatalogService
- LLM локальный
- CLI

Ты **ничего не теряешь**, но выигрываешь стратегически.

Если хочешь, следующим шагом я могу:

1. 📐 Нарисовать **MCP contract (tools + schemas)**
2. 👤 Дать скелет MCP сервера (Node.js / TS)
3. 🧠 Показать **prompt** для Qwen, который идеально работает с MCP
4. 🖋️ Сделать **пример реального запроса пользователя → tool call → ответ**

С чего продолжим?

объясни как LLM связаны с генерацией полей `embedding` и `search_vector` в базе данных

Отличный вопрос — это **ключевая точка всей архитектуры**, и если её понять, всё остальное встанет на свои места.





Объясню **спокойно, по слоям**, без магии и маркетинга.

## Главная идея (сразу)


---

**LLM НЕ “обновляет” `embedding` и `search_vector`.**

LLM:

-  не пишет в БД
-  не знает про `pgvector`
-  не считает `embedding`
-  не трогает `tsvector`

LLM только работает с текстом и JSON.

 Все поля `embedding` и `search_vector` — это **инфраструктурная ответственность backend**, а не модели.

## ◆ Разберём поля по отдельности

---

### `embedding vector`

---

```
sql
```

```
embedding public.vector
```

## Что это?

---

Числовое представление текста (например, 768 или 1024 float-значений).

## Зачем?

---

Для **семантического поиска**:

| “экскаватор для стройки” ≈ “гусеничная техника для земляных работ”



## Кто генерирует embedding?

---

**✗ НЕ LLM**



**Embedding-модель**, например:

- `nomic-embed-text` (через Ollama)
- `text-embedding-3-small` (если платно)
- bge / e5 (альтернативы)



## Когда генерируется embedding?

---

При **записи или обновлении оборудования**, а не при поиске.

Пример pipeline:

text

описание + параметры → embedding model → vector → БД

Пример:

ts

```
const textForEmbedding = `
Экскаватор CAT 320
Категория: экскаватор
Параметры:
- Масса: 21 т
- Мощность: 110 кВт
- Тип: гусеничный
`;

const embedding = embed(textForEmbedding);
```

👉 Это делается **один раз**, а не на каждый запрос.

## 📌 Роль LLM здесь

---

**Косвенная:**

- может помочь *сформировать текст* для embedding (нормализация)
- может помочь *выбрать стратегию поиска*

Но не считает векторы.

## 2 search\_vector tsvector

---

sql

search\_vector tsvector

## Что это?



---

PostgreSQL Full-Text Search (FTS):

- токенизация
- лемматизация
- индексы GIN

## Кто генерирует **search\_vector**?

---

-  НЕ LLM
-  НЕ embedding-модель

 **PostgreSQL trigger**

Ты уже это сделал правильно:

sql

```
create trigger equipment_search_vector_trigger
before insert or update
on public.equipment
for each row
execute function equipment_search_vector_update();
```

Внутри функции обычно:

sql

```
to_tsvector('russian',
  coalesce(name, '') || ' ' ||
  coalesce(description, '') || ' ' ||
  coalesce(category, '')
)
```

## Когда обновляется **search\_vector**?

---

- INSERT



- UPDATE

👉 **автоматически**, без участия AI.

## 📌 Роль LLM здесь

---

Тоже **косвенная**:

- помогает понять, **что искать**
- но не участвует в индексации

## 🧩 Тогда где вообще участвует LLM?

---

Вот ключевой момент 👉

## 🎯 Реальная роль LLM в поиске

---

LLM **НЕ** ищет

LLM **НЕ** индексирует

LLM **НЕ** пишет SQL

LLM делает **3 вещи**:

### 1 Понимание человеческого запроса

---

Пример:

| “Нужен экскаватор для аренды, не больше 25 тонн, Москва”

LLM превращает это в:

json

```
{
  "intent": "search",
  "filters": {
    "category": "excavator",
    "region": "Москва",
    "main_parameters.max_weight_t": {
      "lte": 25
    }
  },
  "semantic_query": "экскаватор для аренды до 25 тонн"
}
```

## 2 Выбор стратегии поиска

---

LLM решает (логически):

- есть фильтры → FTS + SQL
- вопрос “что лучше для...” → semantic search
- конкретная модель → точный поиск

**!** Но реализует стратегию **backend**, не LLM.

## 3 Формирование ответа

---

LLM берёт:

- найденные строки
- реальные параметры
- реальные значения

И пересказывает их человеку.

## Полный цикл (связь всего вместе)

---

sql

INSERT equipment

- └ backend формирует текст
- └ embedding model → embedding
- └ PostgreSQL trigger → search\_vector
- └ запись в БД

SEARCH

- └ user: "какой экскаватор взять"
- └ LLM → SearchQuery JSON
- └ backend:
  - └ FTS (search\_vector)
  - └ vector search (embedding)
  - └ SQL filters
- └ LLM → человеко-читаемый ответ

## Очень важная мысль (частая ошибка)

---

| "LLM = умный поиск"

 Нет.

LLM = интерфейс между человеком и системой поиска.

Поиск:

- PostgreSQL
- индексы
- pgvector
- фильтры

LLM:

- переводчик
- диспетчер

- рассказчик

## Почему твоя схема — правильная

---

Ты уже:

- используешь `pgvector`
- используешь `tsvector`
- отделяешь хранение от логики
- думаешь про MCP

👉 Это **архитектура уровня production**, не игрушка.

## Хочешь дальше?

---

Следующие логичные шаги:

- 1 Как формировать **правильный текст для embedding** (очень важно)
- 2 Как комбинировать **FTS + vector search** (hybrid search)
- 3 Как научить LLM **не лезть в БД**, а вызывать MCP
- 4 Как обновлять embedding при изменении JSON-параметров
- 5 Как сделать re-ranking результатов без платных API

Скажи, куда идём дальше — разберём глубоко 💪

Embedding-модель как ее выбираем и подключаем

Отлично, это **последний базовый кирпич**, без которого весь поиск “не взлетает”.  
Разберём **как инженер**, не как маркетолог.

## Что такое embedding-модель (практически)

---

Embedding-модель — это **функция**:

css

Text → Vector[float]

Где:

- похожие тексты → близкие векторы
- смысл важнее слов

👉 Она **НЕ LLM**

👉 Она дешёвая, быстрая, детерминированная

## 🎯 Критерии выбора embedding-модели

---

Для твоего проекта (каталог оборудования):

### 1 Язык

---

- 🇷🇺 Русский — **обязательно**
- 🇬🇧 Английский — желательно

❌ многие модели “формально поддерживают” русский, но плохо

### 2 Тип данных

---

Тексты:

- короткие названия
- технические описания
- списки параметров
- шумные данные (JSON → текст)

👉 модель должна хорошо работать с **mixed technical text**

### 3 Размер вектора

---

- 768 / 1024 — оптимально
- | 1536 — избыточно
- <384 — часто хуже качество

### 4 Инфраструктура

---

Ты хочешь:

- бесплатную
- локальную
- без API-ключей

- без лимитов

## Рекомендованные модели (2025)

---

### **nomic-embed-text** ★★★★★

---

Почему идеально для MVP:

- работает через **Ollama**
- хорошая поддержка русского
- размер вектора: **768**
- быстрая
- стабильная

 Твой текущий выбор — правильный

## Альтернативы (если понадобится)

---

### ◆ **bge-m3**

---

- multilingual
- 1024
- тяжелее
- лучше для длинных текстов

### ◆ **intfloat/e5-large-v2**

---

- очень хорош для поиска
- но сложнее в развёртывании

## Как подключается embedding-модель (реально)

---

### Вариант А — через Ollama (рекомендую)

---

#### 1 Установка модели

---

```
bash
```

```
ollama pull nomic-embed-text
```

#### 2 Пример запроса

---

```
bash
```

```
curl http://localhost:11434/api/embeddings \
-d '{
  "model": "nomic-embed-text",
  "prompt": "Экскаватор для земляных работ до 25 тонн"
}'
```

Ответ:

json

```
{  
  "embedding": [0.0123, -0.044, ...]  
}
```

### 3 Использование в коде (Node.js / TS)

---

ts

```
async function embed(text: string): Promise<number[]> {  
  const res = await fetch('http://localhost:11434/api/embeddings', {  
    method: 'POST',  
    headers: { 'Content-Type': 'application/json' },  
    body: JSON.stringify({  
      model: 'nomic-embed-text',  
      prompt: text,  
    }),  
  });  
};  
  
const json = await res.json();  
return json.embedding;  
}
```



## Вариант В — Python (если backend на Python)

---

python

```
import requests

def embed(text):
    r = requests.post(
        "http://localhost:11434/api/embeddings",
        json={
            "model": "nomic-embed-text",
            "prompt": text
        }
    )
    return r.json()["embedding"]
```



### Какой текст отдавать в embedding (ВАЖНО)

---

**!** Ключ к качеству поиска — **не модель, а текст**.



**Плохо**

---

text

CAT 320

text

Экскаватор CAT 320.  
Категория: экскаватор.  
Тип: гусеничный.  
Масса: 21 тонна.  
Мощность двигателя: 110 кВт.  
Назначение: земляные и строительные работы.

## Правило

---

| Embedding должен “читать” то же, что и человек

## Когда пересчитывать embedding

---

Пересчёт нужен, если меняется:

- name
- description
- category / subcategory
- параметры (JSON)

## Лучший вариант

---

- **backend job**
- или DB trigger → очередь → worker

✗ Не в runtime запроса пользователя

## Как embedding используется в поиске

---

sql

```
SELECT *
FROM equipment
WHERE is_active = true
ORDER BY embedding <-> :query_embedding
LIMIT 10;
```

Часто комбинируют с FTS:

sql

```
ORDER BY
  ts_rank(search_vector, plainto_tsquery(:q)) * 0.4
+
  (1 - (embedding <-> :q_vec)) * 0.6
DESC
```

## Где здесь LLM?

---

LLM:

- генерирует **query text** для embedding
- решает, **нужно ли вообще semantic search**
- перефразирует запрос

Но **НЕ** считает векторы.

SCSS

```
[User text]
  ↓
LLM → semantic_query (string)
  ↓
Embedding model → vector
  ↓
Postgres (pgvector)
  ↓
results
  ↓
LLM → human answer
```

## Итог (коротко)

---

- Embedding-модель выбираем по **языку + инфре**
- **nomic-embed-text** — идеален для MVP
- LLM и embedding — **разные сущности**
- Качество = текст для embedding
- MCP отлично управляет этим процессом

Если хочешь, следующим шагом могу:

- 1** Дать **готовую функцию генерации embedding-текста из твоей таблицы equipment**
- 2** Показать **hybrid search SQL** под твою схему
- 3** Описать **batch reindex pipeline**
- 4** Сделать **MCP tool для embedding + search**

Что берём дальше?

у меня Embedding сейчас строится через локальный LLM и это медленно но есть подключение к API LLM\_CHAT\_PROVIDER=groq GROQ\_API\_KEY=  
LLM\_MODEL=llama-3.3-70b-versatile  
#GROQ\_BASE\_URL=https://api.groq.com/openai/v1  
GROQ\_BASE\_URL=https://api.groq.com LLM\_DEBUG=true

LLM\_EMBEDDINGS\_PROVIDER=ollama EMBED\_MODEL=nomic-embed-text я могу использовать в качестве EMBEDDINGS PROVIDER другие модели ресусы например тот же groq для ускорения

Короткий ответ: **да, можно ускорить embeddings, но Groq — плохой кандидат именно для embeddings.**




Ниже — разбор без мифов и с практическими вариантами.

## 🔴 Главное сразу (важно)

---

**Groq = ускоритель LLM-инференса, а не embedding-платформа.**

На сегодня Groq:

-  отлично подходит для **чат-LLM (70B, reasoning, parsing)**
-  **не предназначен для массовой генерации embeddings**
-  не даёт стабильного, дешёвого embedding-API как OpenAI / Cohere

👉 Использовать Groq для embeddings **архитектурно неправильно**, даже если технически получится.

## Почему у тебя сейчас медленно

---

Ты написал:

```
env
```

```
LLM_EMBEDDINGS_PROVIDER=ollama  
EMBED_MODEL=nomic-embed-text
```

И embeddings считаются **локально**.

## Возможные причины тормозов:

---

1. CPU без AVX2 / AVX512
2. Ollama запущен **без GPU**
3. Embeddings считаются **по одному**, синхронно
4. Embeddings пересчитываются слишком часто

## ? Можно ли использовать Groq как EMBEDDINGS PROVIDER?

---

### Технически:

---

- Groq API не позиционирует embeddings как основной продукт
- нет гарантии:
  - стабильности размерности
  - одинаковых векторов со временем
  - совместимости с pgvector индексами

### Архитектурно:

---

✗ плохо

- embeddings = инфраструктурный слой
- Groq = runtime reasoning слой

👉 Это разные задачи.

### ✓ Правильные варианты ускорения (по приоритету)

---

🏆 **Вариант 1: Оставить Ollama, но сделать правильно (РЕКОМЕНДУЮ)**

---

#### 1 Убедись, что embeddings считаются НЕ LLM'ом

---

Очень частая ошибка:

- ✗ embeddings через llama-3.3-70b
- ✓ embeddings через nomic-embed-text

Проверь, что ты **НЕ** используешь chat completion для embeddings.

## 2 Включи GPU для Ollama (если есть)

---

```
bash
```

```
ollama run nomic-embed-text
```

Проверь:

```
bash
```

```
ollama ps
```

Если GPU не используется — embeddings будут медленные.

## 3 Батчинг (x5–x20 ускорение)

---

❌ Плохо:

```
ts
```

```
for (item of items) {  
  await embed(item.text)  
}
```

✅ Хорошо:



ts

```
embedBatch(texts: string[])
```

Через Ollama можно слать **массив запросов** (или параллелить).

#### 4 **Делай embeddings асинхронно**

---

-  В runtime пользователя
-  Через очередь / worker

sql

```
INSERT equipment
→ enqueue embedding job
→ worker
→ update embedding
```

#### **Вариант 2: Вынести embeddings в внешний API (гибрид)**

---

Если тебе **важна скорость**, но без больших затрат:

##### ◆ **OpenAI `text-embedding-3-small`**

---

- очень быстрый
- дешёвый
- стабильный
- 1536 → можно хранить

 Минус: платный, но копейки.



## ◆ Cohere embeddings

---

- отличны для поиска
- поддержка RU
- быстрые

## 🏆 Вариант 3: Локальная embedding-модель НЕ через Ollama

---

Для продакшена часто делают:

- Python service
- `sentence-transformers`
- ONNX / CUDA / CPU AVX

Это **быстрее Ollama**, но сложнее.

## 🧠 Очень важная архитектурная мысль

---

Embeddings считаются **редко**,  
поиск — **часто**.

Поэтому:

- embeddings **НЕ** должны быть ультрабыстрыми
- они должны быть **стабильными и качественными**

LLM (Groq):

- быстрый
- вызывается часто
- не сохраняет состояние

Embeddings:

- медленные допустимы
- считаются batch'ами
- сохраняются навсегда

## 🔥 Идеальная схема для тебя (рекомендую)

---

text

LLM (Groq)  
└─ parsing, reasoning, MCP tools

Embeddings  
├─ nomic-embed-text  
├─ batch  
├─ async worker  
└─ pgvector

Если станет узко:

- заменить embedding provider
- НЕ трогая LLM
- НЕ трогая MCP

## ❌ Чего точно не делать

---

- ❌ embeddings через 70B LLM
- ❌ embeddings в runtime user search
- ❌ смешивать Groq и embeddings в одном слое
- ❌ пересчитывать embeddings при каждом запросе

## Хочешь — могу дальше

---

- 1 Проверить твой pipeline и указать **точно где тормозит**
- 2 Дать **код батчинга embeddings** под Ollama
- 3 Предложить **гибрид: локально + API fallback**
- 4 Рассчитать **стоимость embeddings** при платном API
- 5 Помочь вынести embeddings в отдельный microservice

Скажи, куда идём 🧡

тогда давай обсудим такое момент как формировать текст для embedding у нас каталог такого вида

```
CREATE TABLE public.equipment ( id serial4 NOT NULL, url
varchar(500) NOT NULL, "name" varchar(300) NOT NULL, category varchar(100)
NULL, subcategory varchar(100) NULL, brand varchar(100) NULL, region varchar(100)
NULL, description text NULL, main_parameters jsonb DEFAULT '{}':jsonb NULL,
additional_parameters jsonb DEFAULT '{}':jsonb NULL, price varchar(100) NULL,
photo_links jsonb DEFAULT '[]':jsonb NULL, created_at timestamp DEFAULT
CURRENT_TIMESTAMP NULL, updated_at timestamp DEFAULT
CURRENT_TIMESTAMP NULL, embedding public.vector NULL, is_active bool
DEFAULT true NOT NULL, search_vector tsvector NULL, CONSTRAINT
equipment_pkey PRIMARY KEY (id), CONSTRAINT equipment_url_key UNIQUE (url));
CREATE INDEX equipment_brand_idx ON public.equipment USING btree (brand)
WHERE (is_active = true); CREATE INDEX equipment_category_idx ON
public.equipment USING btree (category) WHERE (is_active = true); CREATE INDEX
equipment_embedding_idx ON public.equipment USING ivfflat (embedding
vector_cosine_ops) WITH (lists='29'); CREATE INDEX equipment_is_active_idx ON
public.equipment USING btree (is_active); CREATE INDEX equipment_region_idx ON
public.equipment USING btree (region) WHERE (is_active = true); CREATE INDEX
equipment_search_vector_idx ON public.equipment USING gin (search_vector)
WHERE (is_active = true); CREATE INDEX equipment_subcategory_idx ON
public.equipment USING btree (subcategory) WHERE ((is_active = true) AND
(subcategory IS NOT NULL)); -- Table Triggers create trigger
equipment_search_vector_trigger before insert or update on public.equipment for each
row execute function equipment_search_vector_update() поиск будет вестись скорее
всего сначала по выбору типа оборудования в категории субкатегории, category
subcategory, затем по характеристикам main_parameters jsonb,
additional_parameters jsonb, и по описанию description, названию name и по
производителю как лучше формировать текст для embedding
```

Отлично, это **самый важный практический вопрос во всём RAG/каталоге**.

От того, **как ты сформируешь embedding-текст**, зависит до **70% качества поиска**, а не от модели.

Ниже — **проверенная схема именно под твой каталог**, не абстрактная.



## Главный принцип (запомни)

Embedding должен представлять “смысл оборудования”,  
а не копию строки из БД.

Мы не дублируем `search_vector`,  
мы кодируем семантику.



## Цель embedding в твоей схеме

Embedding используется для:

- неточного запроса
- вопросов “что подойдёт”
- поиска по характеристикам словами
- синонимов и перефразов

- ✗ Не для точной фильтрации
- ✗ Не для диапазонов чисел
- ✗ Не для региона / цены

## Стратегия: 3 слоя текста

---

Я рекомендую строго такую структуру:

csharp

```
[IDENTITY]
[CAPABILITIES]
[PARAMETERS (NORMALIZED)]
```

### 1 Identity — “что это вообще”

---

Это самый важный блок.

text

Экскаватор CAT 320.  
Тип оборудования: экскаватор.  
Подтип: гусеничный экскаватор.  
Производитель: Caterpillar.

Источник:

- name
- category
- subcategory
- brand

 Почему:

пользователь почти всегда начинает с типа

## 2 Capabilities — “для чего используется”

---

Берётся из:


- description
- иногда из параметров (если есть смысл)

text

Назначение: земляные и строительные работы.

Подходит для рытья котлованов, траншей, погрузки грунта.

Используется на строительных площадках и инфраструктурных проектах.

 Почему:

запросы: “для стройки”, “для котлована”, “для демонтажа”

## 3 Parameters (NORMALIZED) — ключевая часть

---

**! Важно**

---

JSON нельзя просто stringify.

Embedding-модель плохо понимает JSON.

## Плохо

---

json

```
{"weight":21,"engine_power":110,"type":"crawler"}
```

## Хорошо (человеческий язык)

---

text

Технические характеристики:

Масса: 21 тонна.

Мощность двигателя: 110 киловатт.

Тип хода: гусеничный.

## Правило нормализации параметров

---

### 1 Только важные параметры

---

Не всё подряд.

Примеры важного:

- масса
- грузоподъёмность
- мощность
- глубина копания
- тип (гусеничный / колёсный)

- ✗ серийный номер
- ✗ артикул
- ✗ URL
- ✗ дата создания

## 2 Единицы измерения ОБЯЗАТЕЛЬНЫ

---

text

Масса: 21 тонна.

а не:

text

Масса: 21000.

## 3 Преобразуй названия параметров

---

JSON key	Текст
max_weight	Максимальная масса
engine_kw	Мощность двигателя
dig_depth_m	Глубина копания

## Итоговый пример embedding-текста

---

text

Экскаватор CAT 320.

Тип оборудования: экскаватор.

Подтип: гусеничный экскаватор.

Производитель: Caterpillar.

Назначение: земляные и строительные работы.

Подходит для рытья котлованов и траншей, погрузки грунта.

Технические характеристики:

Масса: 21 тонна.

Мощность двигателя: 110 киловатт.








Тип хода: гусеничный.

Глубина копания: 6.7 метра.

 Это идеальный embedding-текст.

## Что НЕ включать в embedding

---

-  region
-  price
-  url
-  photo\_links
-  created\_at
-  updated\_at
-  is\_active

 Это фильтры, а не семантика.

## Связь с FTS (**search\_vector**)

---

Поле	Использование
name	FTS + embedding
description	FTS + embedding
category	фильтр + embedding



Поле	Использование
subcategory	фильтр + embedding
brand	фильтр + embedding
JSON параметры	embedding
region	SQL
price	SQL

## Как LLM участвует (тонко)

---

LLM может:

- помочь **один раз** создать шаблон преобразования JSON → текст
- помочь с **mapping параметров**
- НЕ участвует в runtime embedding

## Где это реализовать

---

Лучшее место:

```
text
```

```
CatalogService
└─ EmbeddingTextBuilder
    └─ build(equipment): string
```

Один детерминированный метод.

## Рекомендую тестировать так

---

Возьми 10 реальных вопросов:

- “экскаватор до 25 тонн”
- “техника для рытья котлована”
- “гусеничный экскаватор cat”

И смотри:

- что попадает в top-3
- без LLM
- только vector search

## Краткий чек-лист

---

- ✓ Человеческий язык
- ✓ Структура
- ✓ Нормализованные параметры
- ✓ Без шума
- ✓ Один embedding на объект

## Хочешь дальше?

---

Следующие логичные шаги:

- 1 Дам **реальный код EmbeddingTextBuilder (TS)**
- 2 Покажу **SQL hybrid search** под твою схему
- 3 Как хранить **версию embedding-шаблона**
- 4 Как безопасно пересчитывать embedding при изменениях
- 5 Как сделать **re-ranking без платных API**

Скажи, что делаем дальше 💪

но как нам хранить поля я параметров если надо будет в итоге делать SQL запрос из полученного JSON

Отличный вопрос.

Ты сейчас упёрся в **самую важную границу архитектуры** — и это очень хорошо.

Коротко:

**Embedding ≠ фильтрация ≠ SQL**

Это три разных представления одних и тех же данных.

Ниже объясняю как хранить поля параметров так, чтобы:

- embedding был качественный
- SQL был точный
- LLM ничего не ломал

## Ключевая идея

---

| У параметра всегда 2 формы (иногда 3):

1. Структурированная — для SQL
2. Нормализованная текстовая — для embedding
3. (опционально) человеко-читаемая — для ответа

Они **НЕ** должны совпадать.



## Как хранить параметры в БД (ты уже почти всё сделал)

---

У тебя уже есть:

```
sql
```

```
main_parameters jsonb
additional_parameters jsonb
```

Это **правильно**.

Пример:

```
json
```

```
{
  "weight_t": 21,
  "engine_power_kw": 110,
  "drive_type": "crawler",
  "dig_depth_m": 6.7
}
```



Это **canonical-форма**.

## Где происходит магия преобразования

---

- ! HE в LLM
- ! HE в SQL
- ! HE в embedding-модели

## В backend-коде

---

pgsql

```
equipment row
├─ JSON parameters (canonical)
├─ EmbeddingTextBuilder → text
└─ SQL filters ← SearchQuery
```

## Как связать JSON ↔ SQL ↔ embedding

---

### 1 Parameter Dictionary (обязательно)

---

Создай **единый словарь параметров**:

ts

```
const PARAMS = {
  weight_t: {
    label: "Масса",
    unit: "тонна",
    type: "number",
    sql: "main_parameters->>'weight_t'",
  },
  engine_power_kw: {
    label: "Мощность двигателя",
    unit: "киловатт",
    type: "number",
  },
  drive_type: {
    label: "Тип хода",
    type: "enum",
    values: {
      crawler: "гусеничный",
      wheeled: "колёсный"
    }
  }
}
```

👉 Один источник истины.

## 2 EmbeddingTextBuilder (JSON → текст)

---

ts

```
function buildEmbeddingText(eq: Equipment): string {
  const lines = [];

  lines.push(`${eq.name}.`);
  lines.push(`Тип оборудования: ${eq.category}.`);
  if (eq.subcategory) lines.push(`Подтип: ${eq.subcategory}.`);
  if (eq.brand) lines.push(`Производитель: ${eq.brand}.`);

  lines.push(`Технические характеристики:`);

  for (const [key, value] of Object.entries(eq.main_parameters)) {
    const meta = PARAMS[key];
    if (!meta) continue;

    if (meta.type === "number") {
      lines.push(`${meta.label}: ${value} ${meta.unit}.`);
    }

    if (meta.type === "enum") {
      lines.push(`${meta.label}: ${meta.values[value]}.`);
    }
  }

  return lines.join("\n");
}
```

 Здесь JSON → текст

 Без SQL

 Без LLM

## 3 SearchQuery JSON (LLM → backend)

---

LLM возвращает **НЕ SQL**, а **структуру**:

json

```
{
  "category": "excavator",
  "filters": {
    "weight_t": { "lte": 25 },
    "drive_type": "crawler"
  }
}
```

👉 LLM **НЕ** знает, как это превратится в SQL.

#### 4 SQLBuilder (SearchQuery → SQL)

---

ts

```
if (filters.weight_t?.lte) {
  where.push(`
    (main_parameters->>'weight_t')::float <= ${i++}
  `);
}
```

- 📌 SQL строится **детерминированно**
- 📌 Без участия LLM

#### 🔄 Что НЕ делать

---

- ❌ хранить параметры как текст
- ❌ парсить параметры из embedding
- ❌ извлекать параметры из description
- ❌ позволять LLM писать SQL

## Почему это масштабируется

---

- хочешь добавить новый параметр → словарь
- хочешь изменить embedding → builder
- хочешь новый фильтр → SQLBuilder
- LLM остаётся неизменной

## Пример полного цикла

---

vbnet

DB:

```
main_parameters.weight_t = 21
```

Embedding:

```
"Масса: 21 тонна."
```

User:

```
"экскаватор до 25 тонн"
```

LLM:

```
weight_t <= 25
```

SQL:

```
(main_parameters->>'weight_t')::float <= 25
```

 Всё сошлось, без магии.

## Итог (самое важное)

---

- JSON параметры = **источник истины**
- embedding = **человеческая проекция**
- SQL = **машинная логика**
- LLM = **переводчик**, не исполнитель



## Хочешь дальше?

---

Логичные продолжения:

- 1 Как **версионировать параметры**
- 2 Как делать **range-фильтры быстро** (GIN vs expression index)
- 3 Как **объединять FTS + vector + JSON фильтры**
- 4 Как MCP описывает параметры, чтобы LLM их не путал
- 5 Как автоматически проверять, что LLM не запросил несуществующий параметр

Скажи, что дальше 🔥

```
{"Мощность": "132 л.с.", "Рабочий вес": "13500 кг", "Тип питания": "Дизельный",  
"Режим нагрева": "Электрический нагрев плиты", "Транс. размеры":  
"6300×2100×3030 мм", "Модель двигателя": "DONGFENG CUMMINS QSB4.5",  
"Рабочая скорость": "1,5~30 м/мин", "Частота Вибрации": "0 ~ 25 Гц",  
"Производительность": "250 т/ч", "Тип рабочего органа": "Механическое уширение  
плиты", "Макс. ширина укладки": "3.1 м", "Общая макс. мощность": "97 кВт",  
"Экологический класс": "Tier III", "Преодолеваемый уклон": "≥20 %", "Номинальная  
амплитуда": "4 мм", "Вес при транспортировке": "13500 кг", "Скорость движения  
вперед": "0~2,8 км/ч", "Угол наклона передних колес": "0~20 км/ч", "Макс. толщина  
дорожного покрытия": "25 см", "Номинальная частота вращения двигателя": "1800  
об/мин"} вот такие данные находятся в параметрах и находится в полях  
main_parameters jsonb, additional_parameters jsonb каких можно обработать и  
сохранить чтоб ыработал лучше схема поиска и надо ли это делать
```

Отличный, **очень показательный пример**.

Ты попал ровно в ту точку, где **95% каталогов ломаются**, если сделать неправильно.

Ниже — **чёткий, практический ответ**:

- 👉 что реально нужно обрабатывать,
- 👉 что оставить как есть,
- 👉 зачем это делать вообще,
- 👉 и когда можно НЕ делать.

## 🧠 Ключевой вывод сразу

---

**НЕ нужно нормализовать ВСЁ.**

Нужно нормализовать **только параметры, по которым реально ищут и фильтруют**.

Остальное:

- хранить как есть
- использовать только для embedding и отображения

## Разбор твоего JSON (по категориям)

Я разобью параметры на 4 группы.

### 1 КРИТИЧЕСКИЕ (обязательно нормализовать)

Это параметры, которые:

- часто встречаются в запросах
- используются с диапазонами
- влияют на выбор техники

#### Нормализовать в **main\_parameters**

Исходное поле	Canonical key	Тип
Рабочий вес	weight_kg	number
Вес при транспортировке	transport_weight_kg	number
Общая макс. мощность	power_kw	number
Мощность	power_hp	number
Производительность	capacity_tph	number
Макс. ширина укладки	max_width_m	number
Макс. толщина дорожного покрытия	max_thickness_cm	number
Преодолеваемый уклон	max_slope_percent	number

 Почему:

“до 15 тонн”, “не менее 200 т/ч”, “ширина укладки 3 м”

### 2 ENUM / КАТЕГОРИАЛЬНЫЕ (нормализовать)

Используются как **фильтры** или **уточнения**.

Исходное поле	Canonical key	Значения
Тип питания	fuel_type	diesel, electric
Экологический класс	emission_class	Tier II / III / IV
Тип рабочего органа	working_body_type	enum

 Пример:

json

```
"fuel_type": "diesel"
```

### 3 ПОИСКОВЫЕ, НО НЕ ФИЛЬТРУЕМЫЕ

---

Хороши для:

- embedding
- описаний
- сравнений

✗ Не нужны в SQL.

#### Поле

Модель двигателя

---

Режим нагрева

---

Транс. размеры

---

Рабочая скорость

---

Частота Вибрации

---

Номинальная амплитуда

---

Номинальная частота вращения двигателя

👉 Оставлять **как есть**, использовать:

- в embedding
- в ответе пользователю

### 4 СОМНИТЕЛЬНЫЕ / ШУМ

---

Редко ищут, часто шумят.

## Поле

Угол наклона передних колес (подозрительно, возможно ошибка)

---

Скорость движения вперед (дублирует рабочую скорость?)

📌 Можно:

- хранить
- не индексировать
- не фильтровать

## 🧩 Как это хранить (рекомендую)

---

main\_parameters (ТОЛЬКО нормализованные)

---

json

```
{
  "weight_kg": 13500,
  "power_kw": 97,
  "power_hp": 132,
  "capacity_tph": 250,
  "max_width_m": 3.1,
  "max_thickness_cm": 25,
  "fuel_type": "diesel",
  "emission_class": "Tier III"
}
```

## additional\_parameters (сырьё + остальное)

---

json

```
{  
  "Модель двигателя": "DONGFENG CUMMINS QSB4.5",  
  "Режим нагрева": "Электрический нагрев плиты",  
  "Частота Вибрации": "0~25 Гц",  
  "Транс. размеры": "6300×2100×3030 мм"  
}
```

### Надо ли это делать вообще?

---

#### Если НЕ делать:

---

- embedding будет хуже
- SQL-фильтрация невозможна
- всё только “на глаз” через LLM

#### Если делать:

---

- точные фильтры
- быстрый поиск
- меньше галлюцинаций
- понятные ответы

### ВАЖНОЕ ПРАВИЛО

---

| **Никогда не извлекай параметры обратно из embedding.**

Embedding = read-only semantic hint.

## Когда и как делать нормализацию

---

### 1 При импорте данных (лучше всего)

---

- один раз
- можно с помощью LLM
- сохранить результат

### 2 Никогда в runtime пользователя

---

## Как делать нормализацию (практика)

---

Pipeline:

```
javascript
```

```
RAW JSON
↓
Normalizer (rules + regex + units)
↓
Canonical JSON
↓
main_parameters
```

LLM можно использовать:

- на этапе импорта
- с ограниченным schema
- с валидацией

## Итог (чеклист)

---

- ✓ Нормализовать 20–30% параметров
- ✓ Остальное оставить “как есть”
- ✓ Чёткий словарь параметров
- ✓ SQL только по canonical JSON
- ✓ Embedding из нормализованного + текстового

## Хочешь дальше?

---

Я могу:

- 1 Сделать таблицу canonical-параметров под спецтехнику
- 2 Написать код нормализации (regex + units)
- 3 Показать prompt для LLM-нормализации при импорте
- 4 Показать SQL-индексы под JSON параметры
- 5 Сделать MCP-schema для параметров

Скажи, что выбираем 🔥

а если нам сохранить исходные данные в одном поле а нормализованные в другом

Да 👍

Это **не просто можно**, это **ЛУЧШАЯ архитектура** для каталога с поиском и LLM.

Ты сейчас предложил **правильное “двухслойное хранение”** — именно так делают production-каталоги.

Ниже — почему, как и что именно делать.

### Коротко (суть)

---

| Сырые данные ≠ поисковые данные

Мы **НИКОГДА** не теряем исходные данные,  
но **НИКОГДА** не используем их напрямую для SQL.

### Рекомендуемая схема хранения

---

#### 1 Исходные данные (RAW)

---

```
sql
```

```
raw_parameters jsonb
```

Содержит:

- данные “как пришли”

- все строки
- любые единицы
- ошибки источника
- локализацию (RU / EN)

Пример:

json

```
{  
  "Мощность": "132 л.с.",  
  "Рабочий вес": "13500 кг",  
  "Тип питания": "Дизельный",  
  "Модель двигателя": "DONGFENG CUMMINS QSB4.5"  
}
```

 Используется для:

- отображения карточки
- отладки
- повторной нормализации
- контроля качества данных

## **2** Нормализованные параметры (CANONICAL)

---

sql

main\_parameters jsonb



Содержит:

- ТОЛЬКО известные ключи
- ТОЛЬКО числовые / enum значения
- стандартные единицы

Пример:

json

```
{
  "power_hp": 132,
  "weight_kg": 13500,
  "fuel_type": "diesel"
}
```




Используется для:

- SQL фильтров
- сортировки
- логики подбора

### Текст для embedding (DERIVED)

---

 НЕ храним в БД (или храним версию)

Генерируется:

- из canonical + важных raw
- детерминированно

javascript

```
RAW JSON
  ↓
Normalizer (rules / LLM / hybrid)
  ↓
Canonical JSON
  ↓
EmbeddingTextBuilder
  ↓
embedding vector
```

## Почему это идеально

---

### 1. Никогда не теряешь данные

---

- можно пересчитать всё
- можно улучшить нормализацию
- можно сравнить старое / новое

### 2. Поиск всегда стабильный

---

- SQL работает ТОЛЬКО с canonical
- LLM не влияет на фильтрацию
- нет “магических” значений

### 3. Embedding можно улучшать независимо

---

- поменял шаблон
- пересчитал embedding
- БД и логика не меняются

## ✓ 4. Удобно для MCP

---

MCP может:

- читать canonical
- показывать raw пользователю
- не путаться

## 🧩 Рекомендованные поля в таблице

---

sql

```
ALTER TABLE equipment  
ADD COLUMN raw_parameters jsonb DEFAULT '{} '::jsonb;
```

Итого:

Поле	Назначение
raw_parameters	исходные данные
main_parameters	нормализованные
additional_parameters	редкие / неиспользуемые
embedding	семантика
search_vector	FTS

## 🔧 Как делать нормализацию (правильно)

---

### Этап 1 — правила (80%)

---

- regex
- словари
- единицы измерения

## Этап 2 — LLM (20%)

---

- сложные строки
- нестандартные поля
- только при импорте

✗ никогда в runtime поиска

### Пример нормализации

---

ts

```
if (raw["Рабочий вес"]) {  
    canonical.weight_kg = parseNumber(raw["Рабочий вес"]);  
}
```

### Чего НЕ делать

---

- ✗ переписывать raw
- ✗ хранить нормализованные данные только текстом
- ✗ извлекать фильтры из embedding
- ✗ позволять LLM менять canonical без валидации

### Итог (золотой стандарт)

---

**RAW → CANONICAL → EMBEDDING**

три разных слоя, три разных назначения

Ты предложил **архитектуру уровня production**, без шуток.

## Хочешь дальше?

---

Можем:

- 1 Спроектировать **NormalizerService**
- 2 Сделать **JSON Schema** для canonical параметров
- 3 Добавить **версионирование нормализации**
- 4 Показать **SQL индексы** под JSONB
- 5 Связать это с **MCP contract**

Скажи, что делаем следующим шагом 🚀

Спроектировать NormalizerService

Отлично. Давай спроектируем **NormalizerService** так, как его реально делают в production

(без магии, с контролем, версиями и возможностью пересчёта).

Я буду опираться на твой стек (**NestJS + PostgreSQL + JSONB + LLM опционально**).

## 🎯 Цель NormalizerService

---

Преобразовать **raw\_parameters** → **canonical (main\_parameters)**

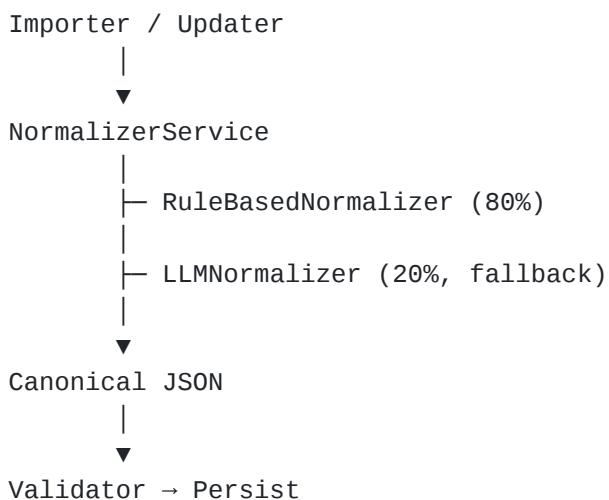
При этом:

- ❌ не портить исходные данные
- ✅ работать детерминированно
- ✅ быть расширяемым
- ✅ позволять пересчёт при изменении логики
- ✅ изолировать LLM

## 🧠 Общая архитектура

---

matlab





## 1 RawInput

---

ts

```
export type RawParameters = Record<string, string | number | null>;
```

## 2 CanonicalParameters (строго)

---

ts

```
export interface CanonicalParameters {  
  power_hp?: number;  
  power_kw?: number;  
  weight_kg?: number;  
  engine_volume_l?: number;  
  fuel_type?: 'diesel' | 'petrol' | 'electric' | 'hybrid';  
  drive_type?: '2wd' | '4wd';  
}
```



### Важно:

- только числа и enum
- никаких строк типа "132 л.с."

## NormalizationResult

---

ts

```
export interface NormalizationResult {
  canonical: CanonicalParameters;
  unresolved: RawParameters;
  confidence: number;           // 0..1
  version: string;
}
```

## NormalizerService — публичный API

---

ts

```
@Injectable()
export class NormalizerService {
  normalize(
    raw: RawParameters,
    context?: NormalizationContext
  ): NormalizationResult
}
```

## Внутренние компоненты

---

### RuleBasedNormalizer (основа)

---

#### Принцип

---

- словари
- regex

- единицы измерения
- 100% детерминирован

## Пример словаря

---

ts

```
const FIELD_ALIASES = {  
  power_hp: ['мощность', 'power', 'л.с.', 'hp'],  
  weight_kg: ['масса', 'вес', 'weight'],  
};
```

## Пример правила

---

ts

```
normalizePower(raw: RawParameters): number | undefined {  
  const value = findByAliases(raw, FIELD_ALIASES.power_hp);  
  if (!value) return;  
  
  const num = extractNumber(value);  
  if (value.includes('кв')) return Math.round(num * 1.36);  
  return num;  
}
```



## 2 UnitParser

---

ts

```
class UnitParser {
  parseWeight(value: string): number | null {
    if (value.includes('т')) return parseFloat(value) * 1000;
    if (value.includes('кг')) return parseFloat(value);
    return null;
  }
}
```

## 3 EnumNormalizer

---

ts

```
const FUEL_MAP = {
  'дизель': 'diesel',
  'diesel': 'diesel',
  'бензин': 'petrol',
  'электрo': 'electric',
};
```

## LLMNormalizer (опционально)

---

! Используется ТОЛЬКО если:

- остались важные поля
- rule-based не справился

## Контракт

---

ts

```
export interface LLMNormalizer {  
  extract(  
    raw: RawParameters,  
    missing: string[]  
  ): Partial<CanonicalParameters>;  
}
```

## Prompt (упрощённо)

---

mathematica

You are a data normalization engine.  
Extract numeric and enum values ONLY.  
Return valid JSON matching schema.

Missing fields:

- power\_hp
- weight\_kg

Raw input:

{...}

### Правило безопасности

LLM-результат → всегда через:

- JSON Schema validation
- диапазоны значений

## Валидация (обязательно)

---

ts

```
validate(canonical: CanonicalParameters) {  
  if (canonical.weight_kg && canonical.weight_kg < 100) return false;  
  if (canonical.power_hp && canonical.power_hp > 3000) return false;  
}
```

## Версионирование

---

ts

```
const NORMALIZER_VERSION = '1.3.0';
```

В БД:

sql

```
normalization_version text  
normalized_at timestamp
```

 Позволяет:

- массовый пересчёт

- A/B логики

## Persist слой

---

ts

```
UPDATE equipment
SET
  main_parameters = :canonical,
  normalization_version = :version,
  normalized_at = now()
WHERE id = :id;
```

## Re-normalization pipeline

---

ts

```
SELECT id, raw_parameters
FROM equipment
WHERE normalization_version != '1.3.0';
```

## Уровни уверенности

---

confidence	значение
1.0	всё rule-based
0.7	часть LLM
<0.5	слабая нормализация

## ✗ Что LLM делать НЕ может

---

- ✗ менять единицы без указаний
- ✗ придумывать параметры
- ✗ писать текст
- ✗ влиять на SQL

## ● Итоговая схема

---

SCSS

```
raw_parameters
  ↓
RuleBasedNormalizer
  ↓
LLM (fallback)
  ↓
Validator
  ↓
main_parameters
```

## ➡ SOON Следующие шаги (логично)

---

- 1 JSON Schema для CanonicalParameters
- 2 Индексы PostgreSQL под main\_parameters
- 3 EmbeddingTextBuilder (из canonical)
- 4 Async-нормализация через RabbitMQ
- 5 UI для просмотра raw vs canonical

👉 Что берём следующим?

- 1 JSON Schema для CanonicalParameters 2 Индексы PostgreSQL под main\_parameters 3 EmbeddingTextBuilder (из canonical) 4 Async-нормализация через RabbitMQ 5 UI для просмотра raw vs canonical объясни каждый пункт

Отличный список — это **ровно тот набор, который превращает RAG-каталог в промышленную систему**, а не «поиск по вектору».

Ниже — **подробно, по каждому пункту**, с привязкой к твоей текущей схеме (equipment, jsonb, embedding, search\_vector, NestJS).




## 1 JSON Schema для CanonicalParameters

---

### Зачем

---

JSON Schema решает 3 критичные задачи:

1.  Гарантия формы данных
2.  Защита от галлюцинаций LLM
3.  Безопасный re-normalization

Без схемы:

- LLM может вернуть "power": "много"
- rule-based может пропустить мусор
- БД станет свалкой

## Пример схемы (Draft 2020-12)

---

json

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "type": "object",
  "additionalProperties": false,
  "properties": {
    "power_hp": {
      "type": "number",
      "minimum": 1,
      "maximum": 3000
    },
    "power_kw": {
      "type": "number",
      "minimum": 1,
      "maximum": 2500
    },
    "weight_kg": {
      "type": "number",
      "minimum": 50,
      "maximum": 200000
    },
    "engine_volume_l": {
      "type": "number",
      "minimum": 0.1,
      "maximum": 50
    },
    "fuel_type": {
      "type": "string",
      "enum": ["diesel", "petrol", "electric", "hybrid"]
    }
  }
}
```

## Где используется

---

Место	Как
NormalizerService	<code>ajv.validate(schema, result)</code>
LLM output	reject если не валиден

Место	Как
Tests	snapshot валидации
Migration	проверка старых данных

🔴 Если не прошло schema → данные не сохраняются

## 2 Индексы PostgreSQL под `main_parameters`

### Зачем

Чтобы:

- делать **фильтрацию SQL**, а не через LLM
- комбинировать `WHERE + vector search`
- не грузить embedding по мелочам

### Базовые GIN-индексы

sql

```
CREATE INDEX equipment_main_params_gin
ON equipment
USING gin (main_parameters jsonb_path_ops);
```

Используется для:

sql

```
WHERE main_parameters @> '{"fuel_type":"diesel"}'
```



## Частичные B-Tree (🔥 важно)

---

sql

```
CREATE INDEX equipment_weight_idx
ON equipment ((main_parameters->>'weight_kg'))
WHERE is_active = true;
```

Но лучше — привести к числу:

sql

```
CREATE INDEX equipment_weight_num_idx
ON equipment (((main_parameters->>'weight_kg')::numeric))
WHERE is_active = true;
```

Пример запроса:

sql

```
WHERE (main_parameters->>'weight_kg')::numeric BETWEEN 10000 AND 20000
```

## Почему не отдельные колонки

---

Подход	Причина
100 колонок	боль миграций
jsonb + индекс	гибко
hybrid	лучший вариант

## 3 EmbeddingTextBuilder (из canonical)

---

### Зачем

---

- ✗ Не embedding всего подряд
- ✗ Не embedding сырых строк
- ✓ embedding **смыслового представления**

### Принцип

---

arduino

canonical → controlled text → embedding

## Пример Builder

---

ts

```
build(c: CanonicalParameters): string {  
  return [  
    c.fuel_type && `fuel: ${c.fuel_type}`,  
    c.power_hp && `power ${c.power_hp} hp`,  
    c.weight_kg && `weight ${c.weight_kg} kg`  
  ].filter(Boolean).join(' ');  
}
```

 Результат:

makefile

fuel: diesel. power 132 hp. weight 13500 kg

## Почему так лучше

---

**Было**

"Мощность 132 л.с."

**Стало**

power 132 hp

"Вес при транспортировке"

weight 13500 kg

мусор

стабильный embedding

 **Embedding становится воспроизводимым**

## 4 Async-нормализация через RabbitMQ

---

### Зачем

---

- импорт → быстрый
- нормализация → тяжёлая
- LLM → медленный

### Схема

---

csharp

```
Importer
  ↓
save raw
  ↓
publish event
  ↓
Normalizer Worker
  ↓
update canonical + embedding
```

### Очереди

---

ts

```
equipment.normalize
equipment.reembed
```

## Payload

---

json

```
{
  "equipmentId": 123,
  "force": false
}
```

## Worker

---

ts

```
@RabbitSubscribe({
  exchange: 'equipment',
  routingKey: 'normalize'
})
async handle(msg) {
  const eq = await repo.find(msg.id);
  const normalized = normalizer.normalize(eq.raw_parameters);
  await repo.update(eq.id, normalized);
}
```

📌 Можно:

- throttle
- retry
- DLQ

## 5 UI: raw vs canonical

---

### Зачем (очень важно)

---

Без UI:

- ты не видишь ошибок
- нельзя обучать правила
- нельзя доверять поиску

### Минимальный UI

---





yaml

RAW		CANONICAL
Мощность: 132 л.с.	→	power_hp: 132
Вес: 13.5 т	→	weight_kg: 13500
Тип: дизель	→	fuel_type: diesel

confidence: 0.93

### Что можно делать

---

-  ручная правка
-  флаг "ошибка нормализации"
-  re-run normalize
-  смотреть версию

pgsql

```
Raw params
↓
Normalizer
↓
JSON Schema
↓
main_parameters
↓
EmbeddingTextBuilder
↓
Embedding
```

## Главное резюме

---

Компонент	Зачем
JSON Schema	защита от мусора
Индексы	быстрый SQL
EmbeddingTextBuilder	стабильный поиск
RabbitMQ	масштаб
UI	контроль качества

Если хочешь — следующим шагом можем:

- спроектировать **гибридный SQL + vector search**
- разобрать **prompt для LLM-normalizer**
- выбрать **embedding provider (Groq / OpenAI / Cohere)**
- сделать **reindex** стратегию

 Что идём делать дальше?

### **Вы достигли лимита плана Free для GPT-5.**

Вам необходим GPT-5, чтобы продолжить этот чат, поскольку в нем есть вложение. Ваш лимит сбрасывается после 14:54.

ChatGPT может допускать ошибки. Проверяйте важную информацию. Смотрите [настройки cookie-файлов](#).