

ПОЛНЫЙ ДЕТАЛЬНЫЙ АНАЛИЗ AI-МОДУЛЯ («МОЗГ» ОРГАНИЗАЦИИ) PROMETRIC V2

СОДЕРЖАНИЕ

1. [Концепция AI как «мозга» организации](#)
2. [Архитектура AI-модуля \(Backend\)](#)
3. [Источники данных и онбординг](#)
4. [Выполнение задач \(AI Agents\)](#)
5. [Мультиязычная поддержка \(RU/KZ/EN\)](#)
6. [Варианты реализации и инструменты](#)
7. [Тестирование и валидация AI](#)
8. [Потенциальные риски и улучшения](#)
9. [Заключение](#)

КОНЦЕПЦИЯ AI «МОЗГА» ОРГАНИЗАЦИИ

AI-модуль как корпоративный мозг: В системе Prometric V2 планируется внедрить AI-ассистента, выступающего в роли «мозга» организации. Это значит, что **ИИ ассистент** будет не просто чат-ботом, а *активным помощником*, который способен **выполнять бизнес-задачи** (планирование встреч, анализ данных, автоматизация рутинных процессов) и предоставлять **консультативную поддержку** сотрудникам.

Мультиорганизационная среда: Платформа рассчитана на поддержку **нескольких компаний (тенантов)** – например, 10 компаний с 20-30 сотрудниками каждая. **Данные и знание AI должны быть строго изолированы по организациям:** ассистент каждой компании обучается **только на данных этой компании** и не «утекает» к другим. Этот принцип изоляции достигается отдельным хранением эмбедингов и фильтрацией по идентификатору компании – подобный подход (например, `group_id` в векторной базе) гарантирует, что каждый клиент видит только свои данные

1 2 .

Ролевой доступ: Внутри одной организации разные сотрудники (роли: Owner, Admin, Employee и т.д.) могут иметь **разные уровни доступа к знаниям и функциям AI**. Ассистент должен уметь учитывать роль запрашивающего пользователя: **один и тот же AI будет давать различный уровень информации и возможностей в зависимости от роли**. Например, владелец видит финансовые сводки и конфиденциальные данные, а рядовой сотрудник – только информацию, разрешённую ему. Технически это реализуется через **метаданные и фильтрацию знаний по ролям**: при поиске по базе знаний AI применяет фильтр по уровню доступа документа ³. Такой механизм гарантирует, что, например, документы с отметкой «confidential: true» доступны только руководству ³ ⁴.

AI как исполнитель, а не только собеседник: Ключевое требование – **ассистент выполняет действия**, а не ограничивается ответами в чате. Он служит *«курсором для бизнеса»*, то есть может **инициировать операции**: например, добавить контакт в CRM, запланировать звонок, сгенерировать отчет или отправить email. Для этого AI встраивается в бизнес-процессы и получает ограниченный доступ к API системы, выступая в роли *умного агента*, способного по запросу вызвать определенные функции.

АРХИТЕКТУРА AI-МОДУЛЯ (BACKEND)

Общий обзор: AI-модуль интегрируется в существующий NestJS-бэкенд как отдельный сервис (например, `AIService` + `AiController`). Архитектура включает несколько слоёв:

```
[User Query] -> [Auth Layer] -> [AI Orchestrator] -> [Knowledge Base & Tools] -> [LLM] -> [Response]
```

1. **Auth Layer:** При каждом запросе к AI backend проверяет **JWT** и определяет `userId`, `organizationId` и роль пользователя. Эта информация далее используется, чтобы ограничить данные и инструменты, доступные ассистенту ⁴. 2. **AI Orchestrator (Service):** Основной оркестратор, который принимает запрос пользователя и решает, **что должен сделать AI**: - Выполнить ли **поиск знаний** (Retrieval) в корпоративной базе знаний. - Требуется ли **инструмент/действие** (напр., вызвать функцию для отправки письма). - Сформировать **промпт** для LLM с учётом контекста (выбранные данные, формат ответа, язык). 3. **Knowledge Base & Tools: - База знаний (Vector DB):** хранилище документных эмбеддингов с разделением по `organizationId` (метаданные) для RAG. Фильтрация по организации (и опционально по разрешенному уровню) применяется при каждом запросе ¹ ³. - **Интегрированные инструменты:** каталог доступных AI функций: внешние API, внутренние endpoints, утилиты. Набор инструментов определяется **динамически по роли** пользователя – напр., сотрудники видят базовые инструменты, а владельцы дополнительные (финансовые отчёты и пр.) ³ ⁵. 4. **LLM (Large Language Model):** собственно языковая модель, генерирующая ответы. Оркестратор формирует для неё окончательный промпт, включая: - Инструкции (системное сообщение) с персоне ассистента и политиками (например, «ты корпоративный ассистент, соблюдай ролевые ограничения»). - **Контекст:** фрагменты знаний, найденные в векторной базе (через RAG), отфильтрованные по организации и роли ⁴. - **Описание доступных функций** (в формате OpenAPI function calling или другого механизма), которые AI может вызвать для выполнения задач. - **Вопрос/команда пользователя**. 5. **Response Formation:** Модель возвращает либо **прямой ответ пользователю**, либо **запрос на вызов функции** (если использован механизм функций/agents). Оркестратор обрабатывает результат: - Если была запрошена функция (например, `scheduleMeeting()`), бэкенд вызывает соответствующий сервис (с дополнительной проверкой прав) и затем передает результат обратно модели для формирования финального ответа. - Финальный ответ отправляется на фронтенд.


NestJS интеграция: - Создается `AiController` с эндпоинтом, например, `POST /ai/query`, принимающим `message` от пользователя. - `AIService.query(user, message)` выполняет шаги: определить компанию/роль, собрать контекст, вызвать LLM (через SDK/API), обработать функции. - **Модель LLM** может вызываться через SDK (OpenAI Node SDK) или через отдельный Python

микросервис (для гибкости с библиотеками вроде LangChain). На ранней стадии достаточно прямого обращения к API GPT-4. - **Данные:** Emeddings-хранилище может быть отдельным сервисом (например, Qdrant, Pinecone) или модулем в PostgreSQL (pgvector). При старте сервиса можно загружать индексы или обращаться через HTTP/клиент.

Безопасность и политика: Все запросы проходят через **контекстный фильтр** (MCP – Model Context Protocol концепция): каждый раз перед обращением к модели проверяется, кто спрашивает и **формируется строго ограниченный контекст** ⁶ ⁷. Модель **не получает ничего лишнего**, кроме разрешенных данных и инструментов. Таким образом достигается *zero-data leakage* и соблюдение ролей.

Масштабирование: 10 компаний – небольшой масштаб, поэтому можно хранить все в одном индексе с фильтрацией `organizationId`. В перспективе роста (сотни компаний) возможен вариант **отдельных коллекций/индексов на компанию** для облегчения фильтрации, либо использование встроенных фильтров в векторной БД (что поддерживается, например, Qdrant/Pinecone) ¹.

ИСТОЧНИКИ ДАННЫХ И ОНБОРДИНГ

Онбординг AI (Features Step): На этапе онбординга пользователю (особенно роли Owner) предлагается добавить источники данных для обучения AI: - **Instagram:** подключение аккаунта компании для извлечения постов, описаний, возможно комментариев. Используется Instagram Graph API для получения последних N постов, которые затем конвертируются в текст (например, описание товара, услуги компании). - **Website:** URL сайта компании – бэкенд выполняет скрейпинг страниц (например, библиотека типа Cheerio или Puppeteer для Node, либо Python + BeautifulSoup через микросервис). Полученные тексты (например, «О нас», описания продуктов) структурируются. В онбординг-коде уже заложен вызов `/api/scrape-website` ⁸ – нужно реализовать этот endpoint в бэкенде: он вытаскивает чистый текст с указанного URL, удаляя HTML/скрипты, и возвращает контент. -  **Manual Input:** пользователю предоставляется текстовое поле, куда он может вручную внести информацию о компании (миссия, ценности, описания продуктов). Эти данные сразу сохраняются. - **File Upload:** загрузка документов (PDF, DOCX, txt). Реализуется через file upload компонент -> бэкенд (например, S3 или локально) -> парсер (PDF можно распарсить через PDF.js или Python pdfminer). Полученный текст включается в набор знаний.

Структурирование знаний: После сбора всех источников, система: 1. **Чистит текст:** удаляет лишнее, объединяет однотипную информацию. 2. **Делит на чанки** (например, по 500 символов с оверлэпом) – для эффективного поиска, как и показано в примерах RAG ⁹. 3. **Генерирует эмбединги** каждой части: выбирается embedding-модель (например, OpenAI ADA для мультиязычности, либо локальная SentenceTransformers модель). *Внимание:* контент может быть на русском или казахском – поэтому предпочтительно использовать **мультиязычную модель эмбедингов**, способную в общем пространстве сравнивать тексты разных языков ¹⁰. 4. **Сохраняет** в векторное хранилище с метаданными: `{ orgId: X, role: 'any'/'owner', source: 'instagram', ... }`. Поле `role` может указывать требуемый уровень допуска. Например, если загружен документ «Финансовый отчет» и отмечен как только для владельца, метаданные `role: owner` – при поиске сотрудника эти данные исключаются.

Динамическое пополнение: AI «учится» не разово на онбординге – важна возможность **добавлять данные и после**. Предусмотрим: - Интерфейс для **добавления новых документов** в базу знаний (например, раздел «Knowledge Base» на dashboard, где owner может подгрузить файл или URL). - **Периодический скрейпинг** подключенных источников (Instagram, сайт) – например, раз в день cron-job обновляет посты или страницы, обновляя эмбединги. Это поддержит актуальность знаний. - **Версионность/удаление данных:** если информация устарела (смена описания продукта), владелец может удалить или обновить связанные записи.

Хранение оригиналов: Помимо эмбедингов, стоит хранить **оригинальный текст** контента (в PostgreSQL или S3). Это нужно, чтобы AI мог цитировать или предоставить развернутый ответ. В response можно вкладывать не только сгенерированный текст, но и ссылки на источник (напрямую показывать пользователю оригинал документа, если нужно).

Приватность данных: Все данные, загруженные компанией, **изолированы**. Векторные точки помечены идентификатором компании ¹, а файлы хранятся в частных хранилищах (пути с orgId). Это исключает пересечение информации между клиентами даже на уровне хранения.

✂ ВЫПОЛНЕНИЕ ЗАДАЧ (AI AGENTS)

Функциональный ассистент: В дополнение к ответам на вопросы из знаний, AI-ассистент способен выполнять **действия по запросу пользователя**. Для этого реализуется подход *AI Agent*: - **Каталог действий (Tools):** Определяется список функций, которые AI может вызвать. Например: - `createContact(name, phone)` – создать новый контакт в CRM. - `scheduleMeeting(date, topic)` – запланировать событие в календаре организации. - `sendEmail(to, subject, body)` – отправить шаблонное письмо. - `generateReport(type, period)` – сгенерировать отчет (продажи, KPI) через внутренний сервис. - Любые другие специфичные для бизнеса команды. - **Интеграция с LLM:** Эти функции регистрируются либо через механизм **OpenAI Function Calling**, либо посредством сторонней библиотеки (например, LangChain Agents). Модель во время генерации ответа может вернуть специальный JSON, указывающий на вызов функции. Бэкенд получит этот сигнал, выполнит функцию и вернет результат обратно модели для продолжения ответа. - **Ограничения доступа:** Каждая функция привязана к **ролям**. Например, `generateReport(financial)` доступна только `role=owner`. Перед выполнением бэкенд **проверяет права** пользователя, даже если AI по ошибке запросил функцию, не разрешенную его роли. Это дополнительная подстраховка. - **Примеры цепочки:** Пользователь (менеджер) спрашивает: «Сформируй отчет по продажам за Q1 и отправь мне на почту». 1. AI проверяет, что для этого нужно: вызвать `generateReport('sales', 'Q1')`, затем `sendEmail(me, ...)`. 2. Модель генерирует ответ с маркером вызова `generateReport`. 3. Оркестратор вызывает сервис отчета, получает PDF/сводку. 4. AI получает результат (например, цифры или файл) и следующий шаг – функция `sendEmail`. 5. После выполнения всех шагов AI формирует текст ответа: «Отчет готов и отправлен вам на почту». - **Безопасное исполнение:** Все вызываемые AI функции – **whitelist**, никаких произвольных команд. Таким образом, AI не сможет выйти за пределы разрешенного (所谓 **Secure Function Calling** подход ⁷). Например, даже если он «захочет» выполнить небезопасный код, у него просто нет такой функции. - **Модульность и тестирование:** Инструменты оформляются как отдельные сервисы или Lambda-функции. Это упрощает тестирование их отдельно и позволяет повторно использовать. Пример из AWS: знание из KB + Lambda для действий + код-интерпретатор

для вычислений ¹¹. Мы аналогично можем выделить: функции для CRM, для Email (через SMTP или внешний сервис), для календаря (например, Google Calendar API), и т.д.

Пример реализации (псевдокод OpenAI function):

```
// Определение функций для OpenAI
const functions = [
  {
    name: "scheduleMeeting",
    description: "Schedules a meeting in the company calendar",
    parameters: {
      type: "object",
      properties: {
        date: { type: "string", format: "date-time" },
        topic: { type: "string" }
      },
      required: ["date", "topic"]
    }
  },
  // ... другие функции
];

// Вызов модели с функциями
const openai = new OpenAIApi(config);
const completion = await openai.createChatCompletion({
  model: "gpt-4-0613",
  messages: [
    { role: "system", content: systemPrompt },
    { role: "user", content: userMessage }
  ],
  functions,
  function_call: "auto"
});

// Обработка ответа
if (completion.data.choices[0].finish_reason === "function_call") {
  const fnRequest = completion.data.choices[0].message.function_call;
  const fnName = fnRequest.name;
  const fnArgs = JSON.parse(fnRequest.arguments);
  // (Проверка прав доступа)
  const fnResult = await executeFunction(fnName, fnArgs, user);
  // Передача результата обратно модели для продолжения
  const secondCompletion = await openai.createChatCompletion({
    model: "gpt-4-0613",
    messages: [
      ...previousMessages,
      { role: "assistant", content: null, function_call: fnRequest },
    ]
  });
}
```

```

    { role: "function", name: fnName, content: JSON.stringify(fnResult) }
  ]
});
finalAnswer = secondCompletion.data.choices[0].message.content;
}

```

В приведенном примере AI может сам решать, вызвать ли функцию (`function_call: "auto"`). Бэкенд проверяет и выполняет, возвращает результат, затем AI продолжает ответ.

Пользовательский опыт: На фронте взаимодействие с AI остается через чат-интерфейс, но AI может сам отчитаться: например, после команды "создай таск и напomini мне через неделю", ассистент ответит "Задача создана и напоминание установлено на 7 дней.". Одновременно реально будет создана задача в системе. Таким образом, **эффективность работы возрастает**, AI становится активным помощником.

МУЛЬТИЯЗЫЧНАЯ ПОДДЕРЖКА (RU/KZ/EN)

Требование локализации: Система ориентирована на Казахстан, поэтому критично поддерживать **три языка:** русский, казахский и английский. AI-ассистент должен **понимать и отвечать** на любом из этих языков, в зависимости от предпочтения пользователя или языка вопроса.

Поддержка на уровне модели: Современные крупные LLM (GPT-4, Claude 2) обучены мультиязычно и способны воспринимать/генерировать тексты на русском и в определенной степени на казахском. Однако, для уверенности: - Выбирается **модель с мульти-языковыми возможностями**. GPT-4 показывает хорошие результаты на русском. С казахским сложнее (меньше данных), но GPT-4/3.5 тоже может, хотя возможны небольшие неточности. Альтернативно, можно подключить модели вроде **Google Translation API** как fallback: напр., если обнаруживаем язык = казахский, перевести на английский, провести поиск/генерацию, потом перевести ответ обратно на казахский. - **Embedding-модель:** Обязательно мультиязычная, иначе поиск знаний будет неточным. Например, OpenAI `text-embedding-ada-002` поддерживает многие языки (включая русский, казахский), или использовать open-source модели из топa MTEB (Multilingual Text Embedding Benchmark) ¹². Это позволит: **запрос на казахском** -> эмбеддить -> найти релевантный текст, даже если он изначально на русском (векторное пространство общее) ¹⁰. Таким образом, AI найдет информацию независимо от языка оригинала. - **Хранение данных:** Сохранять контент можно в исходном языке. Для надежности можно дополнительно сохранять **перевод важных документов** на основной язык (например, английский) – но это необязательно, если эмбединги мультиязычны.

Логика ответа: - Если вопрос задан на русском, ассистент **отвечает по-русски**; если на казахском – по-казахски; на английском – на английском. Это контролируется либо автоматически (модель сама склонна ответить на языке вопроса), либо задается явно в системном сообщении: *"Always reply in the same language as the user's query."* - В онбординге пользователь указывает предпочтительный язык интерфейса (RU/KZ/EN) – это можно использовать как индикатор, на каком языке отдавать проактивные сообщения или push-нотификации от AI.

Казахский язык: Казахский менее распространен в ML-моделях. Возможные меры: - Использовать **Kazakh NLP** модели или данные. Возможно, найти embedding-модель, обученную на казахском (на MTEB или local research). - Протестировать GPT-4 на казахских запросах; если качество недостаточное, интегрировать перевод. Например, маленькая библиотека для перевода (Google Translate API) может быть вызвана AI как один из инструментов, если ассистент не уверен в запросе. - В идеале, со временем, можно добавить в обучение AI корпоративные документы на казахском, чтобы улучшить терминологию.

UI переключение: Фронтенд уже имеет поддержку `locale` (судя по Next.js роутам `[locale]`). AI-интерфейс может менять placeholder и подсказки в зависимости от языка. Например: «Спросите меня о чем угодно...» / «Кез келген сұрақ қойыңыз...» / "Ask me anything...". Это создаст комфорт для пользователя.

Пример мультязычного взаимодействия: - Пользователь (казахский интерфейс) спрашивает: "Компаниямыздың деңгей айдағы сату көрсеткіштері қандай?" (смешанный каз-тур-текст, «Каковы показатели продаж нашей компании за прошлый месяц?»). - AI Pipeline: 1. Обнаруживает язык (можно библиотекой detect-language). 2. Эмбединг запроса (модель понимает "сату көрсеткіштері" ~ "sales figures") – находит соответствующий документ (например, финансовый отчет за прошлый месяц) независимо от языка документа. 3. LLM получает контекст (цифры продаж) и вопрос на казахском, формирует ответ на **казахском**: "Өткен айда X бірлік өнім сатылды, табыс Y теңгені құрады.". - Пользователь получает ответ на родном языке, с точными данными.

⚙️ ВАРИАНТЫ РЕАЛИЗАЦИИ И ИНСТРУМЕНТЫ

При создании AI-модуля следует рассмотреть несколько подходов и выбрать оптимальный с учетом ресурсов и требований безопасности:

1. Модель и хостинг: - *Облачные API:* Использовать OpenAI GPT-4/3.5 через API – на старте даёт лучшее качество и мультязычность без DevOps усилий. Минусы: данные отправляются внешнему провайдеру (нужно заключить DPA для конфиденциальности), стоимость при росте использования. - *Open-Source LLM on-prem:* Развернуть локально модель типа **Llama 2 70B** или аналог. Минусы: нужны мощные сервера/GPU, настройка. Плюсы: полный контроль над данными. Для 10 компаний по ~30 пользователей, нагрузка не огромная, но пиковые моменты могут быть. Возможно, промежуточный вариант – модель поменьше (Llama2 13B fine-tuned) если качество будет достаточным. - *Гибрид:* Менее чувствительные запросы идут через OpenAI, а для данных с высокой конфиденциальностью – локальная модель, обученная специфически на данных компании.

2. Vector Database: - *Managed (облачные):* Pinecone, Zilliz (Milvus Cloud), Weaviate Cloud – быстро внедряется, масштабируется. Но это внешние сервисы (вопрос данных). - *Self-hosted:* **Qdrant**, Milvus – можно поднять в Docker. Qdrant, например, легок в использовании, поддерживает фильтрацию по метаданным ¹. К тому же на NestJS можно обернуть клиент Qdrant (есть JS SDK). - *Postgres + pgvector:* небольшие объемы данных (сотни документов) можно хранить прямо в Postgres с pgvector. Это упрощает инфраструктуру, но уступает специализированным DB в скорости семантического поиска.

3. Фреймворки для RAG: - *LangChain*: популярный фреймворк для связывания LLM + данных + инструментов. Имеет модуль RetrievalQA, Agents, есть версия под Node (LangChain.js). Может ускорить разработку типовых цепочек (например, RetrievalQAChain с фильтрацией по метаданным как на StackOverflow решили ¹³). Но также можно и **самостоятельно** реализовать, что дает больший контроль. - *LlamaIndex (aka GPT Index)*: Фреймворк, облегчающий создание индексов знаний и запросов к ним. Тоже поддерживает метаданные, composition разных индексов. - *Haystack*: Open-source от deepset, больше на Python, но очень мощный для QA систем.

4. Интеграция инструментов (Agent frameworks): - *OpenAI Functions*: как описано выше, нативный способ для GPT-4/3.5. Прост в имплементации на Node/Python, без сторонних библиотек. - *LangChain Agents*: готовые реализации реактивных агентов, которые могут использовать несколько инструментов, решать когда что вызвать. Требуется тщательной настройки, чтобы не было «бродячего» поведения. - *Microsoft Semantic Kernel*: ещё один SDK, позволяющий описывать навыки (скиллы) и подключать к ним LLM, тоже стоит изучить. - *Amazon Bedrock Agents*: если бы система работала на AWS, можно использовать Bedrock и их встроенную систему inline-агентов ¹⁴ ¹⁵, но это проприетарный вариант. Наш дизайн фактически воспроизводит подобный функционал своими средствами.

5. Мониторинг и аналитика: - *Логи и трассировка*: каждая операция AI должна логироваться: запрос, отфильтрованный контекст, вызовы функций, ответ. Это важно для отладки и для **безопасности** (аудит кто что спрашивал и что получил) ¹⁶. Можно использовать Elasticsearch+Kibana для логов, либо простые текстовые логи на сервере. - *Сбор метрик*: сколько запросов, среднее время ответа, сколько случаев вызова инструментов, процент успешных/неуспешных действий. Метрики помогут понять загрузку и ценность ассистента. - *A/B тестирование*: возможно, для улучшения стоит экспериментировать: например, разные подсказки (prompt) ассистенту, разные модели (GPT-4 vs GPT-3.5) и сравнивать качество ответов.

6. Безопасность данных: - *Шифрование*: Векторное хранилище должно быть защищено, как и основная БД. В случае managed решений – включить encryption-at-rest. - *Access control*: Бэкенд-слой является стражем: даже если кто-то попытается напрямую дернуть векторную БД, там всё равно нужны ключи и идентификаторы. А сам AI доступен только через аутентифицированный запрос. - *Content Filtering*: Необходимо внедрить модуль, который проверяет ответы AI на нежелательный контент (например, OpenAI API предлагает опцию модерации контента). Это защитит от случаев, когда AI может сгенерировать нечто неподобающее или раскрыть конфиденциальное.

Каждый из этих вариантов будет оцениваться по критериям: **точность/качество** (Quality), **скорость ответа** (Performance), **себестоимость** (Cost), **простота реализации** (Development Effort), **безопасность** (Security). Наиболее сбалансированным видится: *OpenAI API + Qdrant self-hosted + LangChain (или легковесный кастом) + OpenAI Functions для инструментов* – как стартовая версия. Со временем, при росте данных и требований, можно мигрировать на более кастомные модели или расширять инфраструктуру.

ТЕСТИРОВАНИЕ И ВАЛИДАЦИЯ AI

Для уверенности, что AI-модуль работает корректно и безопасно, необходим тщательный подход к тестированию:

1. Юнит-тесты (Module tests): - Тестирование функции **поиска знаний**: на входе запрос + мокнутые данные в векторной БД -> на выходе должны быть релевантные документы. Воспроизводим кейсы: запрос точно совпадает с содержимым документа, запрос на другом языке, запрос с опечаткой – проверяем, что `AIService.findRelevant(userQuery)` возвращает правильный набор. - Тест **функций-агентов**: симулировать вызов от AI функции (например, вызвать напрямую `scheduleMeeting(date, topic)`) с разными параметрами и ролями. Убедиться, что: - Пользователь с ролью, не имеющей доступа, вызывает -> возвращается ошибка/отказ. - Валидные вызовы выполняются (моковая реализация) и возвращают ожидаемые результаты. - Тест **промт-логики**: поскольку промты сложно юнит-тестировать, можно хотя бы проверить генерацию правильной структуры сообщений перед отправкой в OpenAI (например, содержит ли системное сообщение нужные инструкции по языку).

2. Интеграционные тесты: - **Сквозной сценарий**: развернуть тестовую ВД знаний с парой документов. От имени тестового пользователя сделать запрос через API `/ai/query` и получить ответ. Проверить, что ответ содержит информацию из ожидаемого документа (например, если спросили "адрес нашей компании", а в документе "Наш адрес: ...", то ответ должен включать его). - **Разделение данных**: создать 2 тестовые организации с разными документами, отправить идентичный запрос от пользователя каждой – убедиться, что ответы разные и соответствуют их данным (ни в коем случае не смешаны). - **Role filtering**: в организацию добавить документ с флагом только для Owner. От имени Employee задать вопрос, ответ **не должен** содержать сведений из секретного документа. От имени Owner – должен. - **Выполнение действия**: имитировать запрос типа "создай контакт Иван Иванов" от роли Manager. После ответа проверить, что в системе действительно появился новый контакт "Иван Иванов" (для теста можно использовать in-memory реализацию CRM-сервиса).

3. Пользовательское тестирование (UX): - Провести Alpha-тест с несколькими пользователями от каждой роли. Пусть они пройдут онбординг (добавят данные) и позадают ассистенту вопросы: - Фактологические (из загруженных данных) – проверяется точность. - Советы/генерация (например: "придумай приветственное письмо клиенту") – проверяется адекватность тональности, язык. - Команды (например: "напомни мне позвонить клиенту завтра") – проверяется, создалось ли напоминание. - Собрать обратную связь: не отвечает ли ассистент слишком сухо или, наоборот, слишком многословно? Все ли команды понятны пользователю?

4. Автоматизированная проверка качества ответов: - Разработать набор **примеров вопросов** (по 5-10 на каждую категорию: продукция компании, политика, персональные вопросы, команды) и **эталонных ответов или критериев**. Например: вопрос: "что такое Prometric?" – ответ должен упоминать: "это платформа для ...". - Скриптом прогонять эти вопросы через API AI (в тестовой среде) после каждой значимой доработки. Анализировать ответы: - Семантическое сравнение с эталоном (через embedding схожесть). - Проверка на отсутствие запрещенных фраз. - **Regression check**: чтобы новые изменения не испортили ранее работающие кейсы.

5. Нагруженное тестирование: - Оценить производительность: имитировать, например, 50 одновременных запросов (что превышает предполагаемые пиковые 30 пользователей * 1 запрос). Проверить, не падает ли latency, нет ли ошибок таймаута. Если используем внешнее API – учесть rate limit (OpenAI requests/minute). - При необходимости – внедрить очередь запросов, чтобы разгладить пики.

6. Валидация безопасности: - Попытайтесь с разных ролей получить доступ к чужим данным (например, Employee спрашивает: "каковы продажи в компании В?" – AI не должен ответить, а лучше сказать, что не имеет таких данных). - Попробовать сгенерировать у AI нежелательный контент (ругательства, конфиденциальное, и пр.) – убедиться, что либо модель сама отказывается, либо проходит через наш фильтр. Использовать готовые промпты для jailbreak-тестов, чтобы убедиться, что ассистент соблюдает рамки. - Проверить, что если токен JWT недействителен, запрос к AI не проходит (auth guard на контроллере).

Тестирование AI-модуля – непрерывный процесс. После запуска в production, важно настроить **систему сбора фидбека**: например, кнопка «Ответ помог / Не помог», или отслеживать повторные переспрашивания (если пользователь переформулировал вопрос, возможно, ответ был неудовлетворительным). Эти данные помогут итеративно улучшать ассистента.

⚠ ПОТЕНЦИАЛЬНЫЕ РИСКИ И УЛУЧШЕНИЯ

Несмотря на продуманную архитектуру, есть ряд вызовов и возможностей для развития:

Hallucination (галлюцинации модели): LLM может сгенерировать уверенно звучащий, но неверный ответ, особенно если данных мало или вопрос выходит за рамки знаний. Митигировать: - Использовать жёсткий паттерн ответа: если уверенных данных нет, лучше пусть AI скажет "мне это неизвестно". - Включить в системный промпт предупреждение: «Если ответ не основан на данных компании, уточняй у пользователя». - В будущем – внедрить метод **«Double Check»**: ответ + цитаты из базы знаний, или вторичная проверка, чтоб убедиться, что факты в ответе есть в источниках.

Конфиденциальность и compliance: хранение и обработка данных клиентов требует соответствия требованиям (GDPR, локальные законы). Нужно обеспечить: - Возможность **удаления данных** по запросу (если клиент ушел, удалить их embeddings, файлы). - Логирование доступа (кто и когда запрашивал). MCP-подход рекомендует аудит всего взаимодействия AI ¹⁶. - Для критически важных данных, возможно, **не подключать внешний LLM**, а требовать локальную обработку (опционально).

Расширение функционала: - **Обучение на исторических данных:** Помимо загруженных документов, можно подключить данные CRM (например, сделки, клиенты) и научить AI отвечать на вопросы типа «кто наши топ-5 клиентов?». Это потребует интеграции с БД CRM и, возможно, предварительного агрегирования (LLM лучше дать итог, чем сырые данные). - **Автоматизация сценариев:** Настроить триггеры, чтобы AI сам инициировал коммуникацию. Например, увидел просроченный лид – отправил менеджеру напоминание через чат. - **Голосовой интерфейс:** Раз уже есть выбор голоса (/ /), можно добавить синтез речи: при поступлении голосового запроса (через мобильное приложение) – AI распознает (Speech-to-text), отвечает текстом и конвертирует ответ в речь (Text-to-speech) выбранным голосом.

Производительность и затраты: - Следить за расходами API (если GPT-4, токены дорогие). Оптимизировать контекст: не подставлять слишком много документов, ограничивать историю диалога, использовать более дешевые модели на простые запросы. - Кеширование: для часто задаваемых вопросов (FAQ) – можно кешировать ответы или заранее генерировать их при онбординге. - Если число пользователей резко вырастет, планировать масштабирование: несколько экземпляров AI-сервиса, распределение нагрузки, возможно, шардирование векторной БД.

Юридические аспекты AI: Генерируя советы или тексты, AI может ошибаться. Необходимо в Terms of Service указать, что это ассистент, а не юристконсульт, и финальные решения принимаются людьми. Также, если AI генерирует, например, тексты писем клиентам, нужен механизм, позволяющий пользователю просмотреть и отредактировать перед отправкой (чтобы избежать случайных некорректных сообщений).

Обратная связь и обучение: Сбор фидбека (лайк/дизлайк ответу) позволит со временем **дообучать модель** под конкретный стиль компании. Возможно, в будущем подключить **fine-tuning** модели на собранных диалогах, чтобы она лучше понимала специфику отрасли каждого клиента. Однако fine-tuning мультитенантной модели – сложный вопрос, вероятно, придется обучать отдельные под-модели или использовать **LoRA**-адаптации per company.

III ЗАКЛЮЧЕНИЕ

В данном анализе рассмотрена реализация AI-ассистента в Prometric V2, способного стать «цифровым мозгом» организации. Предложенное решение сочетает **современные подходы** (Retrieval Augmented Generation, Function Calling) с акцентом на **безопасность данных и разграничение доступа**. Ключевые особенности разработанной архитектуры:

- **Мультиаренда и изоляция:** Каждая организация имеет свой закрытый контекст знаний, исключая утечки или пересечение данных ¹. Внутри – гибкое разграничение по ролям, гарантирующее, что AI выдаст информацию согласно должностным полномочиям ³.
- **Умный функционал:** AI не ограничен ответами – интеграция с инструментами позволяет ему выполнять реальные задачи (создавать записи, отправлять уведомления, анализировать данные) и тем самым повышать эффективность рабочих процессов ⁵.
- **Локализация:** Ассистент «говорит» на языке пользователя – поддерживается русский, казахский, английский, что критично для удобства в Казахстанском бизнес-контексте. Мультиязычный подход реализован на уровне эмбеддингов и модели, обеспечивая корректность ответов на любом языке ¹⁰.
- **Интеграция в существующую систему:** Backend NestJS расширен AI-сервисом, который бесшовно встраивается в текущий flow (учитывая онбординг, профили пользователя, etc.). Предусмотрена консистентность с имеющимися модулями (Auth, Organization, etc.), включая повторное использование механизмов (например, те же JWT для идентификации пользователя при запросе к AI).
- **Безопасность и контроль:** Реализуется многоуровневая защита – от проверки прав на каждое действие AI до логирования всех обращений. Следование принципам MCP (Model Context Protocol) дает уверенность, что AI соблюдает границы каждого клиента ⁶ ⁴.

- **Масштабируемость и гибкость:** Заложены возможности для роста – от смены модели (OpenAI vs локальная) до масштабирования компонентов (векторное хранилище, сервисы). Кроме того, архитектура достаточно модульна, чтобы добавлять новые возможности (новые типы источников данных, дополнительные инструменты для AI) с минимальным влиянием на ядро.

Готовность к внедрению: Проект находится на продвинутой стадии – основные компоненты спроектированы. Требуется реализовать и протестировать описанные модули. При тщательном тестировании и постепенном запуске (сначала на 1-2 пилотных компаниях) можно ожидать, что AI-ассистент значительно усилит ценность платформы Prometric V2, предоставляя каждому клиенту **персонального интеллектуального помощника для бизнеса**.

Общий вывод: Prometric V2 с AI-модулем становится **инновационной корпоративной платформой**, сочетающей управление бизнесом с возможностями искусственного интеллекта. Это соответствует современному тренду «AI Copilot» в организациях и дает конкурентное преимущество. Главное – продолжать уделять внимание обучению AI на качественных данных конкретной организации и соблюдению приватности. Тогда ассистент действительно станет надежным «вторым мозгом» компании, разгружая сотрудников от рутины и помогая принимать обоснованные решения быстрее и эффективнее.

Документ создан: 2025-09-25

Версия AI-модуля: Проектирование v1.0

1 Multitenancy - Qdrant

<https://qdrant.tech/documentation/tutorials/multiple-partitions/>

2 6 7 16 Why MCP is Crucial for Building Multi-User, Multi-Tenant LLM Applications - Blog - Product Insights by Brim Labs

<https://brimlabs.ai/blog/why-mcp-is-crucial-for-building-multi-user-multi-tenant-llm-applications/>

3 4 5 11 14 15 Build a dynamic, role-based AI agent using Amazon Bedrock inline agents | Artificial Intelligence

<https://aws.amazon.com/blogs/machine-learning/build-a-dynamic-role-based-ai-agent-using-amazon-bedrock-inline-agents/>

8 9 10 12 How to Build a Multilingual RAG with Milvus, LangChain, and OpenAI - Zilliz blog

<https://zilliz.com/blog/building-multilingual-rag-milvus-langchain-openai>

13 python - Configure Multitenancy with Langchain and Qdrant - Stack Overflow

<https://stackoverflow.com/questions/77178335/configure-multitenancy-with-langchain-and-qdrant>