ChatGPT

Архитектура полнофункционального TypeScript-приложения (аналог Notion)

Введение. Проектирование приложения, подобного Notion, – сложная инженерная задача. Необходимо обеспечить одновременную работу множества пользователей над древовидными документами в режиме реального времени, поддерживать интерактивные AI-ассистенты, и при этом гарантировать безопасность и масштабируемость. Такой сервис сочетает в себе богатый редактор (блоки, форматирование, вложенность), мгновенную синхронизацию изменений (коллаборация без конфликтов), управление доступом (workspace, страницы, роли) и интеграцию ИИ для анализа и автодополнения контента. Чтобы реализовать все эти возможности, требуются продуманная архитектура фронтенда и бэкенда, а также выбор подходящей инфраструктуры. Ниже представлен детальный обзор ключевых компонентов.

Архитектура фронтенда

Фреймворк и рендеринг. В качестве основы фронтенда рационально использовать React с метафорой компонетов и состоянием, а также фреймворк Next.js для серверного рендеринга и маршрутизации. Next.js позволит реализовать гибрид SSR/SPA-подход: можно предварительно рендерить публичные страницы (например, опубликованные документы) для SEO, и одновременно использовать клиентскую навигацию для основного приложения. React-компоненты представляют блоки документа, панели инструментов, меню и пр., что упрощает повторное использование кода. При планировании интерфейса важно уделить внимание отзывчивости (responsive design), чтобы приложение корректно работало как на десктопах, так и на мобильных устройствах. Как правило, макет строится с учетом адаптивности: боковые панели сворачиваются на малых экранах, инструменты редактора перестраиваются для удобного касания, обеспечивается прокрутка длинных страниц и т.д. Кроме того, Next.js облегчает внедрение темизации – можно применить CSS-переменные или библиотеки (например, Styled Components или Tailwind) для поддержки светлой/темной темы интерфейса, а также кастомных тем рабочих пространств.

Управление состоянием. Несмотря на то, что Convex (рассматривается ниже) обеспечивает реактивные запросы, на уровне клиента потребуется управление локальным состоянием: текущий пользователь, настройки UI, кэш открытых документов, несохраненные правки и т.д. В React-приложениях для этого часто применяют легковесные стораны (zustand, Jotai) или контекст + `useReducer`, либо Redux/MobX для более сложных сценариев. Например, можно завести Zustand-стор для пользователя и сессии: при входе пользователя стор хранит его профиль и токен; при выходе – сбрасывает. Иной стор может отвечать за кэширование структуры текущего документа в редакторе (например, чтобы отрисовывать изменения мгновенно без повторного запроса). Использование React-состояния и сторов помогает мгновенно отражать действия пользователя (optimistic UI) до получения подтверждения с сервера. Это важно для ощущения **реального времени**: когда пользователь вводит текст или перемещает блок, он сразу видит результат локально, а затем синхронизация приносит подтверждение или поправки от сервера. В целом, благодаря React+Next можно добиться плавной, интерактивной работы приложения – однако это требует аккуратной организации стейта, чтобы при поступлении обновлений по WebSocket/UI реакции происходили без лишних перерисовок всего дерева компонентов.

Редактор документов (блоки, текст, форматирование). Ключевой элемент фронтенда – редактор, позволяющий редактировать *древовидный* документ. Документ Notion-подобного приложения состоит из блоков: каждый блок – это либо параграф текста, либо элемент списка, заголовок, изображение, встроенная база данных и т.п. Блоки могут содержать дочерние блоки, образуя иерархическое дерево (структура вложенных списков, страниц внутри страниц и т.д.). На практике такие структуры реализуются либо на основе **контентаeditable** и rich-text (как ProseMirror, TipTap, Lexical), либо с помощью блоковых редакторов (EditorJS, Slate). Современные решения включают **Lexical** (от Meta) и **TipTap** (на базе ProseMirror) – оба предоставляют высокоуровневое API для работы с документом как с деревом JSON-нод. Например, TipTap поддерживает пользовательские *extensions* для различных типов блоков, всплывающие меню для форматирования, и имеет модуль *Collaboration* для реального времени [1] [2]. Lexical также позволяет определять кастомные узлы (текст, списки, таблицы и т.п.) и плагинами обрабатывать командные вводы (слэши, хоткеи). Выбор редактора зависит от требований: TipTap предлагает более готовое решение с UI-компонентами, Lexical – более низкоуровневый и производительный. В любом случае, редактор должен поддерживать совместное редактирование: это обычно достигается интеграцией CRDT/OT (подробнее ниже) или отправкой изменений по событиям. Например, можно использовать TipTap + **Y.js**: TipTap предоставляет `CollaborationKit`, использующий CRDT Y.js для синхронизации – разработчик подключает `yjs` как источник правок, и редактор автоматически мерджит изменения, а также отображает **курсыры других пользователей** и их выделения (caret). Аналогично Lexical – для него существуют экспериментальные плагины или можно интегрировать CRDT вручную. Важно обеспечить плавное редактирование: операции вставки/удаления текста, перемещения блоков, форматирования должны применяться локально и затем реплицироваться на другие клиенты через сеть.

Реал-тайм коллаборация на фронтенде (CRDT vs OT). В многопользовательском редакторе основная сложность – согласованно применять concurrent-изменения. Два распространенных подхода: **OT (Operational Transform)** – трансформации операций относительно друг друга, и **CRDT (Conflict-Free Replicated Data Type)** – специальные структуры данных, обеспечивающие коммутативность операций. Многие современные редакторы отходят от классического OT (как в Google Docs) в пользу CRDT, либо гибридных решений. Например, Y.js (CRDT) широко используется благодаря простоте: изменения представляют собой операции вставки/удаления с уникальными идентификаторами позиций, которые можно применить в любом порядке и получить один результат. CRDT автоматически разруливает конфликты, но несет оверхед (дополнительные метаданные по каждому символу, например). OT, напротив, требует наличия центрального сервера для согласования – сервер получает операции, применяет их к текущей версии документа, и отсылает трансформированные операции другим клиентам. В архитектуре с централизованным бэкендом можно применить упрощенный подход: **последняя запись побеждает** (LWW) под контролем сервера. Интересно, что сам Notion, по словам инженеров, **не использует сложный CRDT для текста** – вместо этого текст разбит на небольшие блоки (параграфы), и при конфликте сервер просто берет последнюю версию блока [3]. Такой Last-Write-Wins на уровне мелких блоков работает приемлемо: уменьшение размера элементов, редактируемых одним пользователем, резко снижает вероятность конфликта [3]. Конкурентные изменения разных пользователей обычно затрагивают разные блоки, поэтому система часто может принять все изменения. Если же двое одновременно правят один и тот же абзац, побеждает одна правка – другая теряется – однако, учитывая малый объем параграфа, потери невелики и заметны сразу (пользователь видит, что его текст перезаписан, и может вручную вернуть). **CRDT-подход** позволяет избежать потери данных, но сложнее в реализации. Он хранит весь набор внесенных символов/блоков с уникальными идентификаторами позиций, что обеспечивает автоматически-мерджинг изменений. **OT-подход** требует вычислять *трансформации* операций при их применении не в исходном порядке: например, если один

пользователь вставил текст в начало параграфа, а другой – в конец, то операция вставки в конец должна быть трансформирована (смещена) при применении после первой. При большом числе одновременных пользователей OT приводит к взрывному росту комбинаций трансформаций (сложность O(n^2) для n одновременных операций) [4]. CRDT же применяет все операции как есть (коммутативно), но платой может быть рост размера структуры данных и сложность очистки истории. Многие гибридные решения используют *серверный контроль порядка*: например, операции принимаются сервером и рассылаются в порядке прихода, что по сути превращает их в последовательность с last-write-wins при конфликте. Для текстового контента популярны *позиционные идентификаторы*: каждый символ/элемент списка имеет уникальный ID, сохраняющий упорядоченность (Fractional indexing, например) [5]. Это типичный CRDT-подход: вставка символа не зависит от текущего индекса, а указывает позицию между двумя ID – даже если другие символы вставлены параллельно, каждый имеет свой ID и может быть упорядочен без двусмысленности [5]. В итоге, на фронтенде может быть реализовано: либо полный CRDT (через Y.js/Automerge), либо тонкий клиент, отправляющий изменения на сервер (через WebSocket) и получающий обратно финальные изменения (OT на сервере). Оба подхода приемлемы. **Выбор подхода** часто диктуется библиотеками: TipTap/Y.js – CRDT, ProseMirror built-in collab – ближе к OT (сервер сериализует/rebase), Lexical – предоставляет low-level API, можно интегрировать CRDT или делать LWW. В целом, **реал-тайм движок фронтенда** отвечает за: отправку событий изменений, получение и применение чужих изменений, а также за показ *присутствия*.

Присутствие и курсоры. Пользователи ожидают видеть друг друга при совместной работе: кто сейчас на странице, где находится курсор каждого, что выделено. Для этого фронтенд ведет *состояние присутствия* – обычно отдельный объект, который обновляется при активности пользователя. Например, при перемещении курсора или выделении текста приложение отправляет по каналу присутствия сообщение с координатой курсора и ID пользователя. Другие клиенты, получив это, визуализируют маркер (обычно цветной курсор с именем пользователя или аватаром). Такие *live cursors* и *live selections* существенно улучшают восприятие коллаборации [2]. Реализация: можно использовать возможности CRDT-библиотеки (Y.js имеет *Awareness API* для обмена метаданными типа курсора), или простой канал WebSocket для присутствия. В некоторых BaaS (например, Liveblocks, Supabase Realtime) присутствие встроено – разработчик лишь обновляет координаты. В UI курсоры отображаются поверх редактора: как слои со позиционированием в соответствующей позиции текста (нередко – вставкой специального невидимого элемента-контейнера). Помимо курсора, отображаются *границы выделения* текста другими (например, подсветка от начала до конца выделенного фрагмента другим цветом). Также рядом с курсором обычно показывают имя или аватар редактора. Все эти элементы обновляются динамически по данным, приходящим по WebSocket. Если пользователь неактивен, его курсор через некоторое время может исчезать. **UI управления доступом** тоже является частью фронтенда: например, кнопка «Share» на странице открывает модальное окно, где можно добавить участников, назначить им роли (чтение, комментирование, редактирование), или скопировать публичную ссылку. Эта часть представляет тонкий слой над бэкендом: фронтенд собирает список текущих участников и прав (через API), отображает, позволяет изменить, и отправляет изменения на сервер. Также UI должен различать разные статусы: например, если пользователь имеет только комментаторские права, элементы редактирования контента должны быть задизейблены, а интерфейс должен явно показать режим «Comment only». Всё это достигается комбинацией проверок состояния (например, `if (userRole !== 'editor') disable editing controls`) и визуальных подсказок.

Публикация и внешний доступ. Notion позволяет публиковать страницу в web, предоставив публичный доступ по URL. В нашем приложении аналогично – потребуется механизм «publish». На фронтенде может быть реализован отдельный рендер страницы в режиме просмотра без

необходимости логина. Технически с Next.js можно сделать специальную серверную отрисовку: при включении флага «опубликовать» документ становится доступным по URL вида `https://app/doc/<hash>` или пользовательскому слогу. Next.js getServerSideProps может подтянуть содержимое документа из БД и отрисовать HTML статически. Статический рендер хорош для SEO, но нужно обеспечить обновления: если страница правится после публикации, обновленная версия должна автоматически стать доступной. Альтернативно, можно раздавать публичный контент как полностью клиентское приложение, которое по открытому токену грузит данные через публичный API (ограниченного доступа). Выбор подхода зависит от требований безопасности и кэширования: статический export (SSG) дает возможность отдавать страницы через CDN, но сложнее делать инкрементальные обновления; динамический SSR – даёт свежие данные на каждый запрос, но нагружает сервер. Кроме того, фронтенд должен учитывать открытие документов гостями: например, если пользователь без аккаунта открывает публичную страницу, ему показывается режим «только чтение» без возможности редактирования, и без лишних элементов UI (навигация, боковые панели). Поддержка публикации также включает мета-теги (для социальных превью), опцию включения индексации (Notion позволяет запретить поисковикам индексировать).

Итого по фронтенду: Фронтенд архитектура сочетает **Next.js/React** для модульности и производительности, **мощный редактор** (TipTap/Lexical) с расширениями под наши нужды (блоки, slash-команды, таблицы, упоминания и т.д.), и **механизмы коллаборации** (WebSocket + CRDT/OT). Особое внимание уделяется UX: мгновенные отклики, отсутствие блокировок, индикация присутствия, плавная работа на мобильных устройствах. Все это создает интерфейс, в котором команда пользователей может одновременно создавать контент, видя изменения друг друга в реальном времени.

Архитектура бэкенда

Стек и реалтайм-сервер. На стороне сервера критично обеспечить **низкие задержки** и согласованность данных при одновременных запросах. Вариант, который набирает популярность – платформа **Convex** (Reactive backend as a service на TypeScript). Convex предлагает готовый механизм хранения данных, запросов и автоматических real-time подписок: разработчик пишет функции на TS, а Convex сам отслеживает зависимости и пушит обновления клиентам [6] [7]. Однако можно реализовать и **кастомный Node.js** backend. Рассмотрим второй вариант для гибкости: бэкенд на Node/TypeScript (например, фреймворк Fastify или NestJS для структурированности) + БД PostgreSQL. Бэкенд должен держать открытые WebSocket-соединения с клиентами для мгновенной отправки обновлений. Это можно сделать встроив WebSocket-сервер (например, на базе `ws` или Socket.io). При использовании Next.js можно задействовать серверные функции Vercel (но они работают по HTTP, для WS понадобится либо сторонний сервис, либо запуск отдельного Node-процесса). Convex, кстати, избавляет от ручного управления WS: он поддерживает **reactive queries** – клиент подписывается на запрос, и Convex шлет обновления, когда соответствующие данные меняются. Аналогичного эффекта можно достичь с библиотеками типа **Hasura** или Supabase (у них под капотом logical replication, см. ниже). В общем случае сервер будет централизованно принимать операции редактирования, валидировать и сохранять их, а затем рассылать слушателям. Бэкенд также обрабатывает обычные REST/HTTP запросы: вход в систему, загрузка начальных данных, запросы AI-ассистента и пр. Можно организовать два взаимодействующих сервера: один – **API-сервер** (HTTP REST/GraphQL), второй – **Realtime-сервер** (WS). Но чаще в Node это объединено: например, при старте приложения HTTP-сервер и WebSocket сервер работают на одном порту (HTTP upgrade). В контексте TypeScript-бэкенда логично использовать один кодовый проект, где endpoints и WS-обработчики сосуществуют, чтобы разделять логику (например, обновление документа и рассылка событий).

Схема данных и хранение документов. Одно из ключевых решений – как хранить содержимое документов (иерархию блоков) в базе. Существуют два подхода [8] [9]:

- **Блочный (Notion-style):** каждый блок – отдельная запись в БД, со ссылками на родительский блок/страницу и на дочерние блоки. Например, таблица `Blocks` с полями: `id, parent_id, parent_page_id, type, content, order, ...`. При запросе страницы выбираются все блоки с `parent_page_id = pageId` и затем рекурсивно дочерние. Такой подход используют Notion, Roam Research, Workflowy и др., т.к. он отражает древовидную структуру и позволяет загружать/сохранять отдельные блоки. Преимущество: **мелкозернистые операции** – изменение текста в одном блоке затрагивает одну запись (нет переписывания всей страницы) [10], можно подгружать части документа (ленивая загрузка вложенных узлов по мере раскрытия или прокрутки) [11], легко реализовать историю версий блоков или реестр ссылок. Недостаток: выборка страницы требует рекурсивного сбора множества записей, потенциально десятков или сотен запросов (или одного, но большого JOIN). Также сложнее полнотекстовый поиск по документу – требуется склеивать блоки.
- **Документный (Google Docs-style):** хранение всего содержимого страницы целиком, обычно как текст (Markdown/HTML/JSON). Например, таблица `Pages` с полем `content` (JSON-структура или текст). Преимущество: **простота выборки** – вся страница одним запросом [10], легко сохранить/отдать одним куском, удобно индексировать полнотекстово. Недостатки: изменение даже маленького абзаца приводит к перезаписи всего документа (при большой странице – накладно) [12], проблемы с одновременным редактированием (два пользователя поменяли разные места – придется мержить), нет естественной структуры для вложенных страниц. Evernote исторически использовал монолитные заметки (XML), что осложнило внедрение коллаборации [13].

Notion и схожие современные продукты выбрали **блочный подход** ради гибкости и коллаборации. Внутренне «страница» – это тоже блок особого типа (Page), который может иметь child-блоки. Таким образом, каждая страница хранится как дерево блоков рекурсивно. В базе можно представить это несколькими таблицами: `Pages(id, workspace_id, title, ...metadata...)` и `Blocks(id, page_id, parent_block_id NULLABLE, type, content, props...)`. Поле `content` может содержать текст (для параграфов) или JSON (например, для таблиц, списков – специфичные настройки). *Порядок* блоков задается либо отдельным полем `order` (целое или дробное значение для упорядочивания), либо неявно через список children (н-р, хранить массив children-идентификаторов в parent-блоке – но в реляционной БД это неудобно). Чаще используют поле `order` или естественный порядок по `created_at` при сортировке. Дочерние блоки выбираются запросом `SELECT * FROM Blocks WHERE parent_block_id = X ORDER BY order`. Чтобы получить всё дерево, выполняется либо рекурсивный CTE (в Postgres WITH RECURSIVE), либо несколько запросов по уровням. В памяти на сервере можно составить целый JSON дерева и отдать клиенту. Кеширование здесь очень важно: при открытии страницы сотни мелких блоков могут породить нагрузку, поэтому стоит кешировать собранный JSON документа (инвалидируя при обновлениях).

Хранение версий и изменений. Для функций истории (“значок изменений”, откат изменений, восстановление удаленных) бэкенд должен фиксировать версии. Можно подходить по-разному:

- *Полные снимки:* сохранять копию документа (всех блоков) при каждом значимом сохранении. Это дороговато (дублирование данных), но просто для восстановления – берется сохраненная версия JSON.
- *Журнал изменений:* хранить лог операций (например, Operational Transform ops или CRDT-deltas). Тогда версию можно получить, проиграв операции до нужного момента. Этот

подход экономит место для текстов (каждая правка относительно маленькая), но усложняет логику (нужно уметь проигрывать ops, иногда применять обратные ops для undo).

- *Гибрид:* хранить снапшоты раз в N изменений, между ними – диффы. Тогда восстановление = последний снапшот + диффы до нужной точки.

Notion, насколько известно, хранит историю на уровне блоков (возможно, с помощью двоичного журнала). Мы можем реализовать упрощенно: создать таблицу `BlockRevisions` (block_id, timestamp, content_before, content_after, user). Каждое изменение блока пишет запись. Для отката блока можно взять предыдущее content. Для отката всей страницы – применяются все последние версии блоков на нужное время. Альтернативно, Convex или Supabase (с Logical replication) можно использовать как источник истории – они логируют все транзакции. В вопросе также упоминается *корзина/восстановление*: это, скорее, про удаленные страницы или блоки. Решение: иметь флаг `deleted` и помечать удаление, а восстановление – просто снятие флага. Для страниц, возможно, хранить их в отдельной таблице `Trash` (но обычно достаточно флага и фильтрации).

CRDT-синхронизация на бэкенде. Если фронтенд использует CRDT (например, Y.js), то бэкенд может быть вовсе **бестранзакционным** для контента: клиенты сами договариваются о состоянии, а сервер только хранит текущее. Однако обычно сервер тоже должен понимать изменения, например, для записи в БД. Один из вариантов – *хранить CRDT-структуру* (например, Y.js документ) прямо в базе, в сериализованном виде (у Y.js есть `encodeStateAsUpdate(doc)` -> Uint8Array). Тогда при подключении нового клиента сервер отдает ему текущий бинарный doc, а при изменениях получает от клиентов updates и мержит. Это приближает архитектуру к **сервер-как-реле**: логику мержинга делают CRDT-библиотеки на клиенте и сервере, а сервер обеспечивает доставку и сохранность. Например, с Convex можно сочетать Automerge CRDT: клиент шлет изменение (patch), Convex-функция применяет его к хранимому документу и сохраняет [14] [15]. При этом Convex автоматически разошлет обновленное состояние подписавшимся клиентам. **Альтернативный подход** – не хранить CRDT, а хранить финальный текст/блоки, а concurrent-редактирование решать другим способом. Если идем по пути LWW (last write wins), то бэкенд при получении изменения блока просто проверяет метку времени/версию: если опережает текущую – записывает и ретранслирует. При конфликте можно либо отвергнуть старую операцию, либо всё равно записать (с поверхностным слиянием). Многи централизованные приложения предпочитают простые стратегии, потому что наличие сервера позволяет **упростить** алгоритмы (строгие CRDT-алгебраические свойства не обязательно нужны, если сервер обеспечит порядок или явно выберет победителя) [16]. В контексте редактора форматированного текста, *идеальным* решением считается CRDT (чтобы не терять ни символа). В наших блоках LWW тоже работает не идеально, но приемлемо. Можно комбинировать: для *текстовых блоков* внедрить CRDT (например, каждый параграф – отдельный Y.Text, тогда конфликты внутри абзаца решаются CRDT-механизмом), а для операций над блоками (создание, перемещение, удаление) – простой порядок или last-win. CRDT-библиотеки (Y.js) это поддерживают: структура JSON с текстовыми листьями. **Вывод:** бэкенд либо содержит CRDT-движок (через стороннюю либу) для сложного мержинга, либо выступает арбитром с простой логикой (серверный OT: принимать, трансформировать или отбрасывать опоздавшие операции).

Ролевая модель доступа и аутентификация. В многоарендном приложении (multi-tenant) каждый пользователь принадлежит как минимум к одному **workspace (организация)**. У Notion есть роли на уровне воркспейса (Owner, Admin, Member, Guest) и права доступа на уровне страниц (Can edit/comment/view, либо Full Access). Мы должны спроектировать схему хранения ACL (Access Control List). Можно ввести таблицы: `Workspaces`, `Users`, `WorkspaceMembers(user_id, workspace_id, role)`, `PagePermissions(page_id, user_id, access_level)` для гостевых

или индивидуальных шарингов и `PagePermissions(page_id, workspace_role, access_level)` для групповых прав (например, “все участники воркспейса по умолчанию могут просматривать”). При запросе данных сервер **проверяет право**: у пользователя должна быть хотя бы право чтения на запрошенный ресурс, иначе 403. Проверки нужны и в real-time каналах: если пользователь пытается подписаться на обновления документа, не имея доступа, сервер должен отказать в подписке. Это важно для безопасности – ни через один интерфейс не должны утечь данные чужого workspace. Для удобства, можно использовать **RBAC (role-based access control)**: например, роли Owner/Admin автоматически имеют редактирование на всех страницах, Member – возможно только на своих или по приглашению, Guest – только явно расшаренные. Authentication (идентификация пользователя) лучше доверить готовым провайдерам: обычно OAuth через Google/Microsoft/GitHub для корпоративных приложений. Можно интегрировать **NextAuth.js** на фронте или Supabase Auth, или Auth0. В любом случае, после логина пользователь получает JWT или сессию. Сервер при каждом запросе валидирует токен, узнает user_id + workspace_id и роль. Для WebSocket-соединения – либо токен передается при установке соединения, либо используется cookie. **Интеграция с OAuth**: при self-hosted решении можно воспользоваться библиотекой passport.js или firebase auth, но проще – использовать Supabase Auth или аналог, чтобы не хранить самим пароли. Если нужно поддержать email+пароль – тоже можно использовать, но хранить хеши с солью, обеспечить сброс пароля и т.п. – это дополнительные трудозатратные детали, поэтому OpenID Connect провайдеры предпочтительны.

Хранение файлов и медиа. Документы могут включать вложения: изображения, видео, файлы PDF, иконки страниц, обложки. Лучше не хранить блобы непосредственно в PostgreSQL (хотя небольшие можно в bytea). Обычно делают интеграцию с облачным хранилищем: AWS S3 или аналог (Supabase Storage, Google Cloud Storage). Архитектура: фронтенд запрашивает у сервера *временную ссылку* (presigned URL) для загрузки файла; потом загружает файл прямо в облако, минуя наш сервер, чтобы не создавать трафик через него. Сервер получает уведомление или фронтенд сообщает, что файл загружен, после чего можно сохранить ссылку на файл. Альтернативно – сервер принимает файл (через multipart upload), сохраняет куда-то (на диск или тоже в облако), и возвращает URL. Это проще для фронта, но нагружает бэкенд (нежелательно, если файлы большие). В контексте TypeScript-стека можно использовать пакет AWS SDK для выдачи presigned URL. Файлы могут версионироваться (но обычно нет, достаточно последней версии). Также стоит внедрить ограничение прав: ссылки можно делать временными (в S3 – expiring URLs) или публичными в readonly. Дополнительно, можно генерировать превью изображений (thumbnails) для быстрого отображения галерей. БД хранит записи о файлах: `Files(id, owner_type(page/block), owner_id, url, metadata...)` или просто в JSON контента блока (например, блок-изображение хранит URL).

События и уведомления. Помимо основных данных, бэкенду часто нужна система событий: например, триггеры на изменение документов (для лога активности), уведомления пользователям (кто-то поделился с вами страницей, или упомянул вас в комментарии). Это можно реализовать через таблицу `Notifications` и фоновые джобы. Например, при вставке комментария сервер создает уведомление адресату. Так как у нас real-time приложение, можно уведомления тоже раздавать мгновенно через WS. Многие используют Redis Pub/Sub для таких задач (проще, чем пытаться через Postgres NOTIFY для масштабных случаев). Сервис Convex или Firebase облегчает это: они могут вызывать серверные функции по расписанию или по триггерам. Но на self-hosted Node лучше внедрить планировщик (agenda.js, или просто setInterval для периодических) и оповещения через WS канал.

Масштабирование и разделение по рабочим пространствам. В многоарендной (multi-tenant) системе важно изолировать данные одного workspace от других. Самый надежный, но дорогой способ – **отдельная база на клиента** (tenant-per-database): тогда полная изоляция, можно делать

бэкапы и миграции отдельно, и нагрузка разных клиентов не влияет напрямую. Однако это трудно масштабировать на тысячи клиентов – множится число баз и соединений [17] [18]. Практичнее – общая база с **полем tenant_id** во всех ключевых таблицах (Pages, Blocks, etc.) [19]. Тогда все запросы должны фильтровать по tenant_id. PostgreSQL можно настроить с Row-Level Security (RLS) для дополнительной защиты: при подключении сессии установить `current_setting('tenant_id')` и прописать политика, разрешающая доступ только к строкам с этим tenant_id. Так поступает, например, Postgres-движок Nile для multi-tenant RAG – они используют `SET nile.tenant_id = ...` и в последующих SQL он автоматически фильтруется [20] [21]. Важный момент – **производительность**: в рамках одного инстанса БД все клиенты делят ресурсы, но запросы по индексу tenant_id достаточно эффективны пока индекс не слишком разряжен. Для гигантских клиентов (очень больших рабочих пространств) можно выделить им отдельную базу или шардинг. Но на старте общий подход с tenant_id – проще администрировать и масштабировать, все миграции применяются к одной схеме.

Real-time подписки и трансляции. Backend должен организовать рассылку обновлений всем клиентам, работающим над одним документом (или в одном workspace). Есть несколько реализаций (подробнее в следующем разделе), здесь опишем общую логику: при изменении данных (например, пользователь набрал текст в блоке) сервер применяет это в хранилище и затем отправляет обновление другим активным пользователям этого же ресурса. Отправка – через WebSocket сообщение, обычно в определенный *канал* (комнату) соответствующую документу или странице. Например, используя Socket.io, можно сделать комнаты: `doc:<pageId>` – и клиенты при открытии документа присоединяются к этой комнате, а сервер делает `io.to(room).emit('block_updated', {...})` при изменении блока. Так, только клиенты на этой странице получат событие. Для масштабирования, если серверов несколько, потребуется координация: тут вступают в игру внешние брокеры (Redis pub/sub, etc.) – чтобы серверы обменивались событиями. Convex, если используется, скрывает эту механику: его реактивная БД сама *знает*, какие сессии слушают какой запрос, и отправляет им изменения [6]. При собственном решении – нужно либо использовать БД-события, либо вручную в коде после записи в БД вызывать рассылку. **Доставка** должна быть надежной: WS обычно работают поверх TCP, так что доставляются, но клиент может временно отключиться. Обычно, при ре-подключении, клиент запрашивает актуальное состояние документа (т.к. мог пропустить события). Можно оптимизировать, храня на сервере буфер последних оповещений, но проще – просто рефреш документа.

Работа AI-ассистента на сервере. Мы подробно обсудим интеграцию AI ниже, но отметим, что бэкенд играет роль посредника между LLM-сервисами и данными. Когда фронтенд запрашивает, например, автосуммаризацию страницы, бэкенд получает запрос и выполняет следующие шаги: 1. Проверяет права (AI ассистент тоже не должен выдавать данные, если у пользователя нет доступа к ним). 2. Достает из хранилища контекст (например, полный текст страницы и связанных документов) или выполняет **векторный поиск** по embedding-индексу, чтобы найти наиболее релевантные разделы [22]. 3. Формирует prompt для LLM: включает найденный контент, возможно инструкции, имя пользователя, язык. 4. Отправляет запрос внешнему сервису (OpenAI, Azure OpenAI, локальный модельный сервер) и получает сгенерированный ответ. 5. Ответ возвращается фронтенду, а при необходимости – сохраняется (например, как новый блок-резюме, если это функция «сгенерировать резюме страницы»).

Бэкенд также может обновлять векторный индекс при изменении документов: например, после каждого сохранения страницы заново вычислять её embedding и обновлять запись в векторном хранилище (если используется внешняя векторная БД). Можно делать это лениво – по запросу от ассистента, или периодически в фоне.

Интеграция AI как агент. Если реализуется «AI Agent» внутри workspace, он может иметь собственные «функции», которые вызываются моделью (например, через OpenAI Functions или сделанный самостоятельно парсер команд). Например, ассистент может решать, когда ему нужно позвать функцию `searchDocuments(query)` для выполнить поиск по базе знаний workspace, или `createPage(title, content)`. Это требует на сервере реализации API для этих функций и оркестрации диалога. Можно построить это на существующих решениях (Langchain Agents, AI SDK). Но ключевое – **изоляция контекста по workspace**: чтобы агент случайно не раскрыл данные из чужого пространства, все его функции должны фильтровать результаты только по текущему workspace. Метаданные (например, embedding -> какая страница, кому принадлежит) должны содержать tenant_id, и поиск выполняется с условием tenant_id. Как отмечается в исследовании, multi-tenant RAG должен в каждом запросе ограничивать область поиска текущим клиентом – это повышает и безопасность, и эффективность поиска [22]. Если архитектура предусматривает хранение embedding'ов в общем векторном хранилище, должна быть либо отдельная коллекция/индекс на workspace, либо как минимум атрибут для фильтра. Многие векторные базы (Pinecone, Weaviate) поддерживают namespace или фильтрацию по метаданным – это и нужно использовать.

Поддержка контекста и задержки. Важный аспект AI-ассистента – **latency**. Взаимодействие с LLM зачастую медленное (сотни миллисекунд или секунды). Чтобы не блокировать основной поток приложения, делается асинхронно: фронтенд вызывает API -> сервер отвечает сразу «принято, работаю» или устанавливает поток (например, для стриминга токенов). Если используем стрим (Server-Sent Events или WebSocket), то бэкенд может постепенно отправлять ответ ассистента пользователю. Например, OpenAI API позволяет стримить, и Node-бэкенд ретранслирует стрим на веб клиенту. Это улучшает восприятие, даже если полный ответ занимает 5 сек, первые слова появятся через 1-2 сек. **Контекстность диалога**: если ассистент поддерживает multi-turn диалог с пользователем, надо хранить историю запросов. Можно держать в памяти сеанса (например, в Redis или Convex), или требовать, чтобы клиент каждый раз присылал весь исторический контекст (но это нагружает). В идеале – хранить последние N обменов в базе `Conversations(user, agent, messages)` и при новом запросе подгружать для prompt. Однако это риск для token limit, поэтому иногда старые ходы выкидывают или суммаризуют. **Заземление (grounding)** означает, что ассистент должен отвечать с опорой на факты из базы, а не от себя. Для этого в prompt явно включают найденные документы (или их фрагменты) и просят использовать только их. Например: `"Вот выдержки из документов: ... Вопрос пользователя: ... Дай ответ на основе текста."`. Некоторые реализации также требуют от модели давать ссылки на источник (Notion AI пока не даёт ссылки, но другой продукт мог бы). Grounding и контроль достигаются проверкой ответа: можно с помощью регэкспов или доп. модели проверять, не упоминаются ли данные, выходящие за рамки предоставленного контекста. В enterprise-кейсе, где важно отсутствие галлюцинаций, возможна post-processing: неуверенные утверждения модель может пометить. Но это за пределами архитектуры, скорее вопрос настройки LLM.

Опции инфраструктуры real-time (Convex, WebSocket, Postgres, Redis)

Архитектуру реального времени можно реализовать разными способами. Рассмотрим перечисленные варианты, их плюсы и минусы:

1. Convex (Reactive backend). Convex – это облачная (есть и open-source) платформа, сочетающая базу, функции и real-time синхронизацию. Вы пишете TypeScript-функции для запросов и мутаций, а Convex обеспечивает автоматическое отслеживание зависимостей: при изменении данных

Convex сам переисполнит соответствующие запросы и пришлет клиенту новые результаты [6] [7]. Проще говоря, Convex предоставляет **подписки на запросы**: например, клиент делает `await query("getPage", {id})` и получает данные + остается подписанным; когда кто-то вызовет `mutation("editBlock", { ... })`, Convex обновит БД и увидит, что запрос "getPage" зависит от этих данных, и пришлет обновление. Плюсы Convex: не надо самому писать логику WebSocket, рассылки – все из коробки, плюс транзакционность и консистентность (Convex последовательно применяет мутации). Он поддерживает >=80 OAuth-провайдеров для auth, и позволяет вызывать внешние API (например, AI). Более того, Convex недавно опубликовал open-source ядро – можно хостить самому [23]. Главный плюс – **быстрота разработки** и избавление от головной боли с кешем/сокетами. Минусы: привязка к собственному API (не SQL база, а своя схема на основе JSON-хранилища или key-value), ограниченная гибкость сложных запросов (нет join, только скрипты). При высоких нагрузках – надо масштабировать через Convex (вероятно, автоматически). Convex хорош для realtime мелких/средних приложений, но потенциально сложно сделать *on-premise* (хотя есть self-hosted, но молодой).

2. Собственный WebSocket-сервер. Классический подход: развернуть WS-сервер (например, на Node.js + `ws` или Socket.io). Клиенты при подключении проходят аутентификацию (например, по токену) и потом присоединяются к нужным «комнатам» или сообщают, какие документы их интересуют. Сервер хранит список подключений и их подписок. При изменении данных приложение (например, код внутри REST API handler'a) делает `websocketServer.broadcast(docId, message)`. Плюсы: **полный контроль** – можно реализовать любую логику (например, при обновлении блока сразу слать diff), оптимизировать формат сообщений, сжимать payload. Масштабирование: можно сделать несколько инстансов WS-сервера и держать информацию о подписках в Redis (pub/sub: один инстанс опубликовал, все читают; или использовать sticky sessions via load balancer). WebSocket обеспечивает дуплексную связь: клиент также может слать серверу события (используемо для отправки правок). Это гибко – можно по тому же сокету и правки, и presence, и команды пересылать. Минусы: нужно самостоятельно решать проблемы надежности – реконнект клиентов, обработка отсутствия ack (если важна доставка), шифрование (wss), а также обеспечивать отказоустойчивость (в идеале – если один сервер падает, клиенты переподключаются к другому). Кроме того, stateful nature: вебсокеты лучше держать на отдельных серверах или использовать sticky-sessions, т.к. соединение привязано к конкретному серверу. Это усложняет контейнеризацию и autoscaling – понадобится либо **session affinity**, либо внешнее решение (например, опять же Ably/Pusher – готовые WS SaaS). **Вывод:** свой WebSocket-сервер подходит, если нужно строго подогнать под свои требования и есть ресурсы на поддержку. Многие компании сначала идут по этому пути.

3. PostgreSQL Logical Replication (Supabase Realtime). Интересный вариант – использовать саму базу данных как источник событий. Postgres умеет логическую репликацию: транзакции транслируются в виде потока изменений. Проект Supabase построил на этом **Realtime**: специальный сервер подписывается на логическую репликацию (через слот) и получает поток изменений (WAL) для указанных таблиц, преобразует их в JSON и раздает по WebSocket клиентам [24]. Это фактически push-нотификации из БД: если кто-то изменил запись – клиент сразу узнает, **без опроса**. Плюсы: минимализм – не надо писать сервер кода для этого, достаточно настроить Supabase (или сторонний репликатор). Вся логика фильтрации может быть сделана на основе SQL – например, клиент подписывается только на изменения в таблице `Blocks` с `page_id = X`. Supabase Realtime как раз позволяет клиенту подписаться: `"SELECT * FROM Blocks:page_id=eq.X"`, и получать Insert/Update/Delete в эту выборку. Это удобно и согласованно: то, что в базе, то и ушло клиентам. К тому же, Postgres RLS (row-level security) применима – Supabase делает так, что клиенты получают только разрешенные строки (они недавно добавили RLS поддержку в realtime) [25]. Минусы: такая система **почти real-time**, но не мгновенная – изменения проходят через WAL, что может добавить десятки-сотни миллисекунд

задержки. Также throughput ограничен – WAL будет включать все изменения, и если много таблиц/тенантов, поток большой. Supabase решил это путем запуска отдельного репликатора на каждого tenant_id (в свежих версиях). Кроме того, форматы wal2json не супер оптимальны, и сложнее отправлять *произвольные* сообщения (не из БД). Но можно комбинировать: для изменений данных использовать PG realtime, а для таких событий как «пользователь печатает...» (что не сохраняется в БД) – свой WS канал. Зато разработчику меньше кода писать: вставил строку – все подключенные сразу получили через общий механизм. **Итог:** отличный выбор для быстрого старта, особенно если уже используете PostgreSQL. Supabase предоставляет готовый сервер, но можно и самостоятельно (есть open-source supabase/realtime).

4. Redis Streams / PubSub. Redis – легковесный и быстрый брокер сообщений, который можно использовать для трансляции событий. Есть 2 режима: Pub/Sub – простая рассылка в канал (не храня историю, “огонь и забыли”), или Streams – **журнал событий** с сохранением и возможностью ack. Redis Streams можно воспринимать как встроенный "Kafka-lite": продюсеры добавляют сообщения в именованный стрим, а потребители (группы) читают их в своем темпе, можно хранить историю, повторно читать [26]. Для нашего real-time, вероятно, достаточно Pub/Sub, т.к. нам нужно просто фан-аут сообщений всем слушателям в моменте. Например, клиент подписан на канал `doc:123`, Redis Pub/Sub распространяет сообщение всем подписчикам (через Redis -> потом сервер WS их получит). Но есть архитектура, где клиент **напрямую** подключается к Redis Streams (через a lightweight WebSocket gateway) – однако обычно лучше держать WS-сервер, который уже коннектится к Redis. Преимущества Redis: **низкая задержка, высокая производительность** для умеренных нагрузок, простота использования если Redis уже есть в системе [26]. Конфигурация: WS-сервер при старте подключается к Redis. Когда нужно выслать сообщение, либо сам же WS-сервер публикует (`XADD` в stream или `PUBLISH`), и **другие ноды** через Redis получают (subscribe). Это позволяет масштабировать на несколько серверов: Redis выступает шиной событий. При очень большом количестве сообщений Redis может стать узким местом, но по сравнению с HTTP polling – это огромный шаг вперед. Недостатки: нет встроенной знания о данных, т.е. нужно вручную вкладывать payload (JSON, diff). Также Redis PubSub не гарантирует доставку если клиент оффлайн (но Streams может хранить, если настроить). Для collab-редактора, где важен текущий state, обычно offline клиенты при возвращении просто запрашивают snapshot, поэтому журнал не обязателен. **Latency vs Throughput:** Redis обеспечивает очень низкие задержки (сотни микросекунд внутри DC), но макс. throughput может быть ниже, чем у масштабируемых систем типа Kafka – хотя Redis 6+ Streams вполне тянут тысячи сообщений в секунду на узел. Как отмечают инженеры, Kafka оптимизирован под огромный поток (млн сообщений/сек), но ценой немного большей задержки; Redis Streams – быстрее по задержке, но на экстремальных объемах может уступать по общей пропускной способности [26]. Для нашего приложения (документ-редактор) нагрузка – десятки событий в секунду на документ (даже если 5 чел печатают быстро, это не мессенджер). Так что Redis справится, а Kafka была бы избыточна. Операционное преимущество: если уже используем Redis для кешей/квот, добавить Streams – не сложно, не нужно заводить новую технологию.

Вывод по инфраструктуре: Каждый подход имеет сценарии. **Convex** – минимальный custom-code, быстро получить результат, но lock-in и ограниченная низкоуровневая оптимизация. **Свой WebSocket** – максимум контроля, гибкость, но требует поддержки (особенно multi-server). **Postgres Logical (Supabase)** – великолепно интегрируется с базой, идеально если основная нагрузка – изменения данных в PG; но не покрывает не-базовые события, и добавляет небольшую задержку через WAL. **Redis** – универсальный event-bus, хорошо подходит для распределенной архитектуры, добавляет компонент, но надежный и проверенный. В реальности, можно и комбинировать: например, **Supabase Realtime** для CRUD операций + **свой WS** для presence, или Convex для всего real-time а Postgres для heavy данных. В данном обзоре, чтобы быть полнофункциональным, мы рассматриваем *симбиоз*: основной бэкенд на Node/PG, с

WebSocket-сервером, использующим Redis для масштабирования. Это достаточно типично: Node-приложение (например, NestJS + Socket.io) и Redis pubsub для координации.

Интеграция AI-ассистента

Теперь подробно о том, как AI-ассистент (генеративный, работающий с текстами workspace) встраивается в систему. Предположим, ассистент умеет отвечать на вопросы по содержимому документов, помогать с редактированием (например, переписать текст, найти контент). Для этого используется подход **RAG (Retrieval-Augmented Generation)**: модель дополнена возможностью поиска по данным пользователя.

Пайплайн эмбеддингов и индексирование. RAG начинается с превращения документов пользователя в векторное представление (*embeddings*). Бэкенд (или офлайн-процесс) берет каждый документ или логический фрагмент (например, каждый блок или абзац, либо каждые N предложений) и пропускает через модель эмбеддинга (например, OpenAI ADA или локальную SentenceTransformer) чтобы получить вектор – набор чисел размерности d (e.g. 1536 для ADA). Эти векторы сохраняются в **векторное хранилище**. Популярные варианты: сторонние managed-сервисы (Pinecone, Weaviate, Vespa) или self-hosted (модуль pgvector в Postgres, Milvus, ElasticSearch vectors). Для упрощения интеграции можно использовать PostgreSQL с pgvector: тогда не нужно отдельную базу, в таблице хранятся embeddings и через SQL можно искать ближайшие соседи. Например, создать таблицу `Vectors(tenant_id, doc_id, embedding vector(1536), metadata JSON)`. Как упоминалось, *мультиарендность* требует изоляции: либо делать отдельную таблицу/пространство на каждый workspace, либо, как Nile/pgvector, использовать колонку tenant_id и обязательно фильтровать по ней при запросе [22]. Второй способ проще, но важно иметь индекс или механизм, чтобы не пробегать по всем векторам всех клиентов (что может быть огромным числом). К счастью, pgvector поддерживает индексы HNSW, а Pinecone/Weaviate – разделение по namespace. **Метаданные** при индексировании: сохраняем не только вектор, но и информацию – к какому документу/блоку он относится, позиция, возможно категория. Это чтобы, получив результат nearest neighbors, можно было вывести ответ с указанием источника (например: "На странице X найдено: ...").

Поиск и RAG-взаимодействие. Когда пользователь задает вопрос ассистенту (например: "Объясни мне содержание этой страницы" или "Найди где упоминается X в наших документах"), происходит следующий цикл: (1) сервер получает запрос с указанием контекста (текущая страница или весь workspace); (2) сервер формирует embedding запроса (тем же механизмом, что документы, либо специальной Query-моделью); (3) выполняет поиск по vector DB среди embedding'ов документов данного workspace. Обычно берутся топ-K ближайших (например, 5 или 10 фрагментов) [22]. Если workspace маленький, можно позволить себе и точный поиск (без ANN, просто вычислить расстояния до каждого, что не страшно для тысячи документов), это дает максимальную точность [22]. (4) Полученные топ-K фрагментов (с их оригинальным текстом) фильтруются при необходимости (можно отбросить совсем нерелевантные по порогу расстояния) и включаются в prompt. Prompt может иметь вид: *"Контекст: [фрагмент1] [фрагмент2] ... Вопрос: [вопрос пользователя]. Ответь, используя только информацию из контекста."*; (5) Этот prompt отправляется в LLM (например, GPT-4 via API); (6) Модель генерирует ответ. Сервер возвращает ответ клиенту, возможно снабдив ссылками на документы (если предусмотрено, можно пост-обработкой привязать предложения ответа к источникам).

Изоляция по Workspace. Как отмечалось, абсолютно необходимо, чтобы в шаге (3) поиск был *ограничен одним арендатором* – иначе ассистент может случайно достать куски чужих данных. Архитектурно это выполняется либо отдельными индексами, либо runtime-фильтрацией.

Например, если используется OpenAI Embeddings + Pinecone, то можно при upsert указывать `namespace=workspace_id`, а при query – тот же namespace [27]. В pgvector – просто `SELECT ... FROM Vectors WHERE tenant_id = ... ORDER BY embedding <-> query_vec LIMIT K`. Multi-tenant RAG не только безопаснее, но и быстрее: поиск среди данных одного клиента гораздо быстрее, чем по глобальной базе [22]. В статье AWS и Nile отмечается, что ограничение области поиска даёт возможность делать **точный поиск** вместо approximate для небольших наборов, а значит и recall идеальный [22]. В случае Notion-ассистента, очевидно, он не полезет за пределы workspace, но внутри workspace тоже стоит ограничивать контекст текущей страницы, если пользователь работает с ней – так ответы будут более конкретными.

Вызов функций и агентный подход. AI-ассистент может не ограничиваться Q&A. Например, если пользователь попросил "Создай новую страницу с таблицей на основе данных X", ассистент может вызвать функцию `createTable`. Современные LLM API (OpenAI GPT-4, GPT-3.5) поддерживают **Function Calling**: бэкенд регистрирует набор функций (с названиями, сигнатурами) и модель может в ответ вернуть не завершенный ответ, а "вызов функции с такими аргументами". Это позволяет делать **agent**, который сам решит, какие инструменты задействовать. Для нашего приложения можно предоставить функции: поиск документов (для реализации RAG внутри самого диалога), создание/редактирование страницы, получение списка задач и т.п. Архитектура: сервер описывает функции и при каждом запросе, если модель вернула function_call, сервер выполняет соответствующий метод (например, обращается к БД, получает данные) и затем продолжает диалог, возвращая модели результат функции, чтобы та сформировала окончательный ответ. Такой цикл может повториться (многоходовый агент). Прелесть в том, что модель сама контролирует логику. Однако важно не забывать про **безопасность**: функции должны тоже проверять права, и не возвращать лишнего. Например, если реализована функция `searchDocuments(query) -> returns text of top results`, она должна сама использовать текущего пользователя/тенанта для фильтрации. Иначе, если по ошибке она выдаст весь индекс, модель может получить лишнюю информацию. Также лучше ограничить объем данных, которые функция может вернуть, чтобы модель не утонула в них.

Обновление эмбеддингов. Документы не статичны – пользователи постоянно правят их. Значит, embeddings устаревают. Нужно продумать механизм обновления векторного индекса: либо *синхронно* при каждом сохранении документа пересчитывать embedding (что дорого, если документ большой и частые правки), либо *асинхронно*. Асинхронный подход: поставить задачу на фон (например, через очередь задач) – "через 1-2 минуты после последней правки пересчитать embedding". Или batch-обновления ночью. Если ассистент пытается отвечать по недавно обновленному куску текста, а embedding старый, он может дать неактуальный ответ. Впрочем, обычно изменения затрагивают част документа, и даже старый embedding может содержать схожий контент. **Практика**: можно порционно хранить embeddings для каждого абзаца – тогда при изменении блока пересчитываем только его vector, остальные остаются. Это локализует обновления. Convex, кстати, анонсировал инструменты для AI, возможно, помогающие с автоматическим обновлением embeddings (например, триггер функций при изменении docs).

Задержки и производительность AI. AI-запросы могут быть одними из самых медленных операций в системе. Если обычная правка транслируется за 50мс, то запрос к GPT может занять 2 секунды. Это допустимо, но UX нужно продумать: на фронте показывать индикатор "AI is thinking...", возможно, отображать частичный прогресс (стриминг токенов). Нагрузку от LLM API нельзя недооценивать: если много одновременных запросов, можно быстро потратить бюджет (OpenAI берёт $ за 1K токенов). Поэтому архитектура должна включать **квотирование и кэширование**. Квоты: например, не более N запросов от одного пользователя в час; кэш: если два пользователя задали очень похожий вопрос по одинаковому контексту, можно сохранить и

вернуть одинаковый ответ (хотя это редкий случай). Либо кэш на уровне embedding: хранить результаты векторного поиска для недавних запросов, но это тоже специфично.

Отдельный AI-сервис. При высоких нагрузках на AI возможно выносить его в отдельный сервис. Например, микросервис `ai-assistant` который общается с LLM API, а основной бэкенд через очередь делегирует ему сложные запросы. Это классическое разграничение: не держать тяжелые вычисления в том же потоке, что обработка UI-запросов. Однако для простоты на старте можно встроить в тот же Node-сервер вызов OpenAI API (с таймаутами, retry). Если LLM self-hosted (например, локальный LLM типа Llama2-run), тогда точно придется отдельный сервис, т.к. он будет на Python с GPU.

Логирование и обучение. С точки зрения DevOps, запросы к ассистенту стоит логировать: вопрос пользователя, отданные моделью контексты, полученный ответ. Это помогает разбираться в проблемах (почему ассистент дал такой ответ, какие данные использовал). Возможно, в будущем для улучшения ассистента можно fine-tune на накопленных диалогах, но это отдельная тема.

Сквозные аспекты архитектуры

Безопасность и изоляция данных. В многоарендном приложении приоритет – никаких утечек данных между клиентами. На всех уровнях внедряются проверки tenant_id. На бэкенде это контроллеры, SQL-запросы с WHERE tenant_id, RLS. На фронтенде – маршруты, IDs всегда привязаны к workspace пользователя. В идеале, даже если баг в клиенте сделает запрос к чужому ресурсу, сервер ответит отказом. Кроме того, все приватные данные должны храниться и передаваться шифрованно: включаем HTTPS/WSS, включаем шифрование БД (если требуется compliance). Notion заявляет, что шифрует данные в покое и в транзите [28] – это стандарт: включить encryption-at-rest на уровне облачного провайдера (например, для Postgres – Transparent Data Encryption или хотя бы шифрование диска, для S3 – включить AES256). Передача – только TLS 1.2+.

Управление доступом end-to-end. Сквозные права доступа означают, что каждый компонент уважает ограничения. Например, если у пользователя только комментаторский доступ, фронтенд должен заблокировать UI редактирования, но *даже если* он обойдет это (через DevTools), сервер все равно не применит `editBlock` от него (проверит роль). То же касается AI – ассистент, как обсуждалось, ограничен данным workspace. Еще пример: при публикации страницы публично – генерируется токен/ссылка. Пользователь без аккаунта заходя по ней должен иметь доступ **только к данной странице**, и никаким другим (даже через API). Реализация: публичный просмотр может происходить через ограниченного технического пользователя или через особый режим: токен в URL, сервер валидирует токен -> находит страницу -> отдает контент только ее, любые попытки нагрузить что-то еще (например, соседние страницы по API) – запрещаются, т.к. у токена нет такой привилегии. **Модульность безопасности**: хорошей практикой будет вынести проверку прав в middleware/декоратор в коде – чтобы не забыть ни в одном эндпоинте. Также полезно иметь централизованное логирование отказов доступа, чтобы видеть, не пытаются ли неавторизованные запросы что-то получить (это может быть как атака, так и баг клиента).

Бюджет задержки и производительность. Для ощущения мгновенной работы цель – большинство операций <100 мс до отрисовки результата. Это достижимо: локальные операции (ввод текста) – сразу; сохранение на сервер – в фоне; получение подтверждения – через 50мс; UI уже обновлен. Но есть более тяжелые операции: открытие большой страницы, поиск, загрузка

файла, ответ AI. Они могут занять сотни мс или секунды. Их нужно **асинхронизировать**: при открытии страницы можно показать скелетон-экран на 0.3с, а затем постепенно загружать содержимое. Большие страницы – кандидат для виртуализации (как делает Notion: длинные списки или базы данных лениво подгружаются при прокрутке). Разработчик должен определить, какие задержки критичны. Например, коллаборативный ввод: задержка более 200 мс между вводом одного пользователя и появлением символа у другого уже заметна. В WebSocket + хорошее соединение обычно можно уложиться в 50мс внутри региона. Но если пользователи глобально распределены – возникнет сетевой лаг. Решение – много региона (см ниже) или хотя бы оптимизация пакетов (накопление правок и отправка батчем каждые 30мс, чтобы не делать слишком частые вызовы). **Latency budget** – концепция, когда мы распределяем, сколько времени может уходить на каждый этап. Например: UI обработка 5ms, сеть 50ms, серверная обработка 20ms, рендер у клиента 30ms => итого ~100ms. Этого можно достичь, если сервер не делает тяжелых синхронных операций. В Node.js важно не блокировать event loop: никаких длительных синхронных вычислений (шифрование, больших JSON) – все либо стримить, либо выносить в Worker Threads. В бэкенде на типичных нагрузках узким местом станет БД. Нужны индексы на поля фильтрации (tenant_id, page_id, parent_id). Желательно держать часто читаемые данные (структуры страниц) закешированными (Redis, in-memory). Некоторые используют **dataloader** паттерны, чтобы за один запрос выбирать сразу несколько нужных вещей (Batching). Например, при открытии страницы можно одним джойн-запросом получить и страницу, и ее блоки, и авторов блоков, вместо множества отдельных query.

Масштабирование в среде с несколькими арендаторами. Multi-tenancy накладывает, что одни клиенты могут быть очень активны, другие нет. Архитектура должна избежать ситуации, когда один «шумный сосед» тормозит всех. На уровне БД: крупные workspace (с тысячами страниц) – их запросы должны по возможности использовать индексы и не сканировать всю таблицу. Partitioning по tenant_id – опция, если одна таблица блоков станет очень большой (миллионы записей). Postgres поддерживает партиционирование, можно сделать по tenant (если ограниченное число) или ключу, но это заранее стоит планировать. На уровне сервисов: можно масштабировать по функционалу – отдельные инстансы для real-time редактирования, отдельные для AI, отдельные для serve статических страниц. Обычно начинают с монолита, а потом вытаскивают части, когда потребуется. **Horizontal scaling**: фронтенд (Next.js) можно легко масштабировать, т.к. он статический или серверлесс (Vercel Functions масштабируются автоматически под нагрузку). Бэкенд WebSocket – сложнее, как отмечено: нужно обеспечить координацию (Redis pub/sub) и sticky sessions. Node.js хорошо масштабируется до определенного числа соединений на процессе (десятки тысяч), далее можно запустить несколько. Балансировщик (AWS ALB или Nginx) должен направлять повторные подключения одного клиента на тот же сервер либо мы должны иметь механизм переноса состояния. Как альтернатива – рассмотреть **serverless WebSockets** (например, Ably, Pusher – которые снимают задачу масштабирования соединений). Multi-region: если у вас клиенты по всему миру, задержки real-time растут. Вариант – деплоить сервера/концентраторы в основных регионах (US, EU, Asia) и привязывать конкретные workspace к региону (напр., клиент выбирает регион своего воркспейса при регистрации, как в Confluence Cloud). Cross-region collaboration – сложно (двое из разных регионов на одной странице – либо одному из них придется подключаться к далекому серверу, либо нужен меж-регионный синхрон, что тяжело). Многие SaaS решают, что workspace "живёт" в одном регионе. Если бизнес требует гео-распределения, можно реализовать eventual consistency между регионами (типа каждый документ CRDT-мерджится меж датацентров, как Figma многорегионально). Но это очень сложно и выходит за рамки MVP.

Балансировка нагрузки. В нашем многоуровневом приложении имеются разные типы нагрузки: статика (HTML/JS/CSS), API запросы, постоянные WebSocket, фоновые AI-вычисления. Их желательно разделять, чтобы, например, скачивание крупного JS-бандла не мешало обработке

быстрого API. Используют CDN для статики – Vercel автоматом это делает, или можно свой CDN/NGINX. Для API – автоскейлинг по CPU/RAM. Для WS – по количеству соединений. Оркестрация Kubernetes позволяет задать раздельные deployments. Если не использовать K8s, можно ручным образом поднимать отдельные процессы на разных портах. Для AI – возможно держать отдельную очередь (RabbitMQ, etc.) и воркеры.

Кэширование. Кеш играет роль на нескольких уровнях:

- *Фронтенд кэш:* Next.js и браузер могут кэшировать данные. Например, после первой загрузки страницы хранить JSON страницы в LocalStorage или IndexDB – если юзер открыл и закрыл страницу, а потом снова, можно показать сразу последний кеш и параллельно обновить. Это сложнее реализовать, но Convex, например, дает ощущение offline-first. Кэш GraphQL/REST запросов (Apollo, React Query) тоже помогает снизить лишние запросы.
- *CDN caching:* опубликованные страницы можно отдать через CDN с публичным cache (так как они не требовательны к авторизации). Так, внешние пользователи будут получать копию напрямую с edge-сервера, разгружая наш origin.
- *Бэкенд кэш:* Redis как in-memory кэш для дорогих запросов. Например, результаты сложного аналитического запроса (если в будущем появятся дашборды), или просто кэш структур страницы. Можно кэшировать и embeddings (но там лучше обновлять сразу). Также кэш короткоживущих данных – список страниц в меню (меняется редко, 5 минут кеша).
- *Кэш БД:* на уровне СУБД Postgres имеет буфер-пул, и при достаточной памяти часто многие индексы и часто используемые таблицы (Pages, Workspace, small indices) будут всегда в памяти. Но Redis все равно быстрее для тяжелых JSON, т.к. не нужно каждый раз парсить SQL.

Нельзя, однако, слишком агрессивно кешировать динамический контент для авторизованных пользователей – они ожидают реального времени. Поэтому основное – именно push обновлений, а не консистентность через TTL. В нашем случае, каждый документ в активной работе будет постоянно меняться, так что кеш на него смысла не имеет – проще запрашивать актуальный снапшот по открытию. А вот неактивные вещи (список всех пользователей воркспейса) – можно кеш.

Индексирование для поиска. Помимо AI-поиска (векторного) возможно потребуется классический текстовый поиск (по заголовкам страниц, по строкам текста). Для этого, если объем позволяет, можно использовать встроенный full-text Postgres (GIN индекс на tsvector) по всем блокам, или завести ElasticSearch/OpenSearch. Notion, вероятно, индексирует в Elastic, т.к. нужно мгновенное ранжирование по релевантности. В нашем проекте можно поступить так: завести сервис (можно тоже в Node, или использовать MeiliSearch) и при изменениях документов отправлять туда обновления. Опять же, multi-tenant filter. Поисковой сервис должен быть синхронен: например, пользватель вбил слово – мы отправляем запрос на search-сервис с параметром tenant, получаем результаты с ссылками на страницы/блоки. В случае небольшого проекта можно обойтись SQL: создать материализованное представление или даже просто запрос с `to_tsvector(content)` по таблице Blocks. Но в блоковом подходе надо объединять блоки одной страницы, иначе слово, разбитое на два блока, не найдется как цельный контекст [29]. В StackOverflow вопросе отмечалась проблема, что блоковые менеджеры плохо ищут, когда часть запроса в одном блоке, часть – в другом [29]. Решение – либо склеивать текст страницы при индексировании, либо при поиске собирать результаты с нескольких блоков. Это сложный нюанс, но для MVP можно проигнорировать.

Стратегия деплоя. Контейнеризация – де-факто стандарт. Можно держать всё в одном репозитории (монорепо) и даже одном контейнере (вместе Node API, WS, Next.js SSR). Но удобнее разделить: фронтенд (Next) как статический билд можно деплоить на Vercel или S3+CloudFront, а бэкенд – как Node container на сервис типа Railway/Heroku или Kubernetes. Решение зависит от команды: **Managed-подход** (Vercel, Supabase, Railway) позволяет меньше думать о инфраструктуре, но может иметь ограничения:

- *Vercel:* отлично подходит для Next.js фронта – push -> автодеплой превью и прод. Также позволяет серверные безсерверные функции (Edge Functions) на Cloudflare edge. Однако Vercel Functions не держат постоянных соединений, поэтому WebSocket сервер туда не разместить (он мгновенно завершится). Значит, все реальное время либо через внешние SaaS (Ably etc.), либо свой сервер. Vercel может и хостить static+client, а API делать внешним (в Vercel можно указать rewrite к внешнему хосту). В целом, Vercel удобен для фронтенда.
- *Supabase:* это набор облачных услуг на основе Postgres. Оно предоставляет: Postgres DB, Auth, Storage, Realtime, Edge Functions (Deno-based). Если использовать Supabase, можно закрыть большую часть потребностей: БД + реалтайм в одном, хранение файлов, а аутентификация – буквально несколькими настройками (соц. логины, почта). Тогда backend-кода минимум. Supabase Edge Functions (функции на Deno) могут выполнять серверную логику (но они не persistent, для WS не подойдут). Supabase хорошо масштабируется по вертикали (увеличить план). Managed Supabase избавляет от администва PG, но дороговат на больших данных. Self-hosted Supabase можно, но теряется часть магии.
- *Railway:* популярный PaaS для разворачивания Docker образов или Node приложений. Можно там запустить Postgres, Redis, и Node. Он удобен простотой (пуш в гит – деплой), но на больших нагрузках может быть дороже AWS своих. Аналоги – Render.com, Fly.io, или DigitalOcean App Platform. Плюсы: быстрее, чем настраивать Kubernetes, минусы: возможно ограничение гибкости, чуть меньше автоматики чем Vercel. Но для бэкенда Node – вполне.
- *Self-Hosted (на своих серверах или облачных VM):* максимальный контроль, можно настроить все что угодно (k8s, сервис-сеть, etc). Минус – требуется DevOps-опыт, настройка CI/CD, мониторинга. Если продукт для внешних клиентов, вероятно, лучше довериться managed в начале, чтобы не тратить время на поддержку инфраструктуры.

Сравнение: Managed-сервисы сокращают *время разработки*, но иногда за счет *стоимости* (тарифы), *lock-in* (например, Supabase специфичен), и не всегда подходят под требования безопасности (некоторым клиентам нужно on-prem). Self-hosted – вы тратите больше времени, но контролируете данные и расходы на уровне сырья (серверы, трафик). Возможен комбинированный путь: например, продукт может быть развёрнут как SaaS в нашем облаке (managed для нас), но для крупных enterprise клиентов предлагать self-hosted вариант (например, Docker Compose пакет или Helm chart). Такой вариант потребует абстрагировать некоторые сервисы (например, возможность вместо Supabase Realtime использовать собственный Redis, и т.п.).

Работа с конкретными сервисами: - **При использовании Vercel + Supabase:** фронтенд деплоится на Vercel, API-запросы (за данными) прямо идут к Supabase (Supabase предоставляет JS SDK для запросов к БД и realtime из браузера, минуя наш бэкенд!). Это радикально упрощает: по сути, можно обойтись вообще без собственного бэкенда – Next.js страницами да облачной БД. Но на практике понадобится хоть тонкий proxy-бэкенд для AI или сложной бизнес-логики. Supabase Edge Functions могут играть эту роль. Такой архитектурный стиль (Jamstack + Supabase) – минимум сервера, много логики в фронте. Однако, для корпоративного приложения логика доступа/AI на фронте – небезопасно и сложно. Лучше писать backend TS функции (в Supabase edge or Next API). - **При использовании Railway/Render:** вы бы зашили всё (DB, Node, maybe front) на эти

контейнеры. Нужно настроить CI (GitHub Actions, или они предоставляют), environment variables, volume для DB (или managed add-on). WebSockets: на Railway Node Express+Socket.io – нет проблем, они открывают порт. - **При Kubernetes:** настроите раздельные deployments: e.g. `frontend` (Next on Node serving SSR, or static on Nginx), `api` (Node for rest), `ws` (Node for websockets, scaled by connections), `pg` (StatefulSet), `redis` etc. K8s дает гибкость rolling updates (можно трафик режиссировать, etc). Но это значительный overhead для стартапа, обычно.

CI/CD и развертывание. Вне зависимости от платформы, настроим pipeline: - при пуше в main/master – прогон автотестов (юнит, интеграционных), линтер, сборка. - Затем деплой: на Vercel это автоматически, на Railway/Heroku – тоже интеграция, на k8s – через ArgoCD или kubectl apply. - Важно иметь **staging окружение**: например, dev и prod. В Vercel каждый PR деплоится превью, это удобно для фронта. Для полного продукта нужен staging, где мы тестим интеграцию end-to-end с реальной БД (можно copy prod db nightly). - Миграции БД – нужно их автоматизировать (применять при деплое). Используют инструменты: Prisma Migrate, Knex migrations, Flyway. В CI можно запускать `prisma migrate deploy` на прод DB. - **Rollout стратегии:** если изменение крупное, можно делать *canary* релизы – развернуть новую версию для части пользователей. В веб-приложениях сложнее, но можно на стороне фронта переключать API endpoint по user group. Либо завести feature flags: включать новый функционал через флаги (гейтить и фронт, и бэк). Тогда можно выпустить код, но неактивный, и включить постепенно. - **Blue-green deployment:** держать две версии среды параллельно, переключить роутинг с старой на новую, откатиться быстро если что. Это обычно уже в Kubernetes делается или на продвинутых PaaS. - **Monitoring & Logging:** После деплоя, нужно мониторить состояние. Инструменты: **Prometheus/Grafana** для метрик (загрузка CPU, mem, кол-во запросов, ошибок 500). **APM (Application Performance Monitoring)** как Datadog, NewRelic – дают детальный трейс запросов, профилирование DB. **Error tracking**: Sentry – чтобы ловить исключения JS (фронт и бэк) и иметь алерты. **Logs**: должны собираться централизованно – либо тот же Sentry (для ошибок), либо ELK stack (ElasticSearch + Kibana) или cloud-specific log explorer. Важно, чтобы логи индексировались: по incident можно поискать всю активность конкретного пользователя, или последние действия перед падением. - **Alerts:** Настраиваются оповещения на метрики: высокий процент ошибок (>1% 5xx в минуту), увеличение latency API, многократные перезапуски контейнеров, заполнение диска, рост времени запросов БД, и конечно, слова "ERROR" в логах. Алерты отправляются в Slack/почту on-call разработчикам, чтобы быстро реагировать. Например, если real-time сервер упал и никто не получает обновления – должна сработать метрика (возможно, по количеству активных соединений или heartbeat). - **DevOps automation:** Terraform или аналог для описания инфраструктуры желательны, если не используем полностью managed (который сам все создает). Например, описать VPC, базы, Redis, kubernetes deployment. Helm charts, if k8s, to define our application. - **Testing environments:** Помимо staging, возможно, нужны feature-specific env (особенно если много параллельной разработки). Docker-compose локально – минимум, чтобы разработчик мог поднять PG, Redis, Node локально и потестить. Хорошо написать скрипты заполнения тестовыми данными, чтобы не делать все вручную через UI.

Self-hosted vs Managed recap: - Self-hosted: больше контроля над данными (можно разместить на корпоративных серверах клиента), возможность тонкой оптимизации (настроить Postgres, co-locate services), потенциально дешевле при крупном масштабе. Минусы – ваша команда отвечает за обновления, безопасность, бекапы. Это оправдано для enterprise клиентов, которые требуют изоляции. - Managed (SaaS): быстрее обновления и фичи выкатывать, меньше усилий на поддержку. Минус – клиенты зависят от вашего uptime, вам надо очень надежно все сделать (N+1 резервирование, DR plans). Но обычно для продукта стартуют как SaaS, затем по требованию делают вариант для on-prem (лицензионный).

Интеграция компонентов: поток данных сквозь систему

Чтобы понять, как все части сочетаются, проследим **жизненный цикл** нескольких сценариев:

1. Редактирование документа в реальном времени:

- Пользователь A открывает страницу. Фронтенд (Next.js) выполняет запрос за данными страницы (через API или прямой запрос к Convex/PG). Бэкенд проверяет права и возвращает структуру страницы (JSON дерево блоков). Клиент отрисовывает содержимое в редакторе. Одновременно устанавливается WebSocket-соединение и клиент подписывается на обновления этой страницы (например, `joinRoom(pageId)` событие). Теперь пользователь начинает ввод – при каждом изменении (скажем, каждые 50-100 мс при вводе текста) Lexical/TipTap генерирует delta (в CRDT это автоматом, или onChange callback). Эта delta отправляется на сервер: если CRDT, то как бинарный update; если дифф, то как сообщение `{op: "insert_text", block: 5, offset: 10, text: "A"}`. Сервер принимает: через тот же WS (Socket.io message handler) или HTTP (less likely). На сервере происходит обработка: проверка валидности (существует ли блок 5 и может ли A редактировать), затем применение – например, обновление в БД (Block id=5 контент меняется). Далее сервер формирует обновление для рассылки: может быть тот же формат (CRDT update blob или `{block:5, newText: ...}`) и отправляет **всем в комнате, кроме отправителя** (чтобы не отправлять обратно автору, хотя это не критично, автор может просто игнорировать). Пользователь B, находящийся на той же странице, через WS получает сообщение. На его фронтенде TipTap collab экзекутор применяет изменение: текст блока 5 изменяется, на экране появляется новая буква – практически сразу (с задержкой сети ~50мс). Если используется CRDT/Yjs, то Yjs самостоятельно синхронизирует, а приложение уже обновлено реактивно (TipTap интегрирован с Yjs). Пока все в онлайне, все правки мгновенно видны. Если кто-то офлайн (вышел из сети и вернулся), при переподключении клиент запрашивает текущее состояние документа заново (или синхронизирует CRDT state vector), и получает последние данные.

- В то же время, A и B видят курсоры друг друга: при каждом движении курсора A фронт посылает сообщение presence типа `{cursor: position, selection: [from,to]}`. Сервер можно не сохранять эти ephemeral данные, а сразу ретранслировать другим. Клиент B получает `{cursor: ..., user: A}` и рисует на соответствующей позиции. Если A бездействует N секунд, можно послать `cursor:null` (признак ушел). Это не критично сохранять – presence статусы восстанавливаются только в момент, когда все онлайн.

- Если A удаляет целый блок, процесс аналогичен: фронт отправляет `delete block 5`. Сервер удаляет запись из БД (или помечает удаленным), шлет всем команду удалить этот блок. Клиенты обновляют состояние редактора (удаляют DOM элемента).

- Обработка конфликтов: если A и B *одновременно* изменили один абзац. В CRDT случае – оба изменения дойдут, CRDT объединит (например, оба вставили по символу в разные позиции – получится строка с обоими символами). В LWW случае – сервер может принять сначала A, затем B и перезаписать текст B поверх. Тогда у A его изменение потеряется: но A клиент тоже получит по WS обновление (с текстом от B) и перерисует. Пользователи видят финальный текст. Может быть, ассистент (AI) или логика заметит конфликт и сообщит, но чаще просто “тихий” override.

- В бэкенде, запись изменений: транзакция на PG фиксирует новые данные блока. Можно также тут же создать запись в `BlockRevisions` (с old/new content). Если нужно, через event system можно уведомить, например, сервис индексирования (чтобы обновить search index).

- Вся эта цепочка происходит обычно быстро: DB commit – миллисекунды, WS отправка – миллисекунды. Так достигается **соглашение**: все клиенты синхронно видят одно и то же содержимое.

2. Добавление комментария и уведомление:

- Пользователь C оставляет комментарий на странице (например, к блоку 7). Фронтенд либо через REST, либо realtime-канал отправляет `addComment(block7, "Hello")`. Сервер создает запись в таблице `Comments` (id, block_id, author, text, timestamp). При успехе отправляет по WS (или по Convex subscription) событие всем на странице: `comment_added` с данными. Клиенты A, B, D на этой странице получают и либо сразу рендерят новый комментарий (например, иконку в интерфейсе), либо обновляют счетчик комментариев.

- Кроме того, сервер определяет: владелец страницы – другой пользователь (скажем, X, не онлайн сейчас). Тогда сервер создает `Notification(user:X, type:"NewComment", page: id, from: C)`. Когда X зайдет в систему позже, он увидит уведомление. Чтобы в реальном времени уведомлять, можно при создании нотификации сразу проверить: есть ли WS-сессия для user X сейчас? Если да, отправить через нее событие `notification: NewComment...`. Если нет – отправить email, или ждать пока зайдет. Это уже бизнес-детали.

3. AI ассистент – пример запроса:

- Пользователь A открыл страницу и нажимает «Ask AI» – вводит вопрос на естественном языке. Фронтенд вызывает API (REST POST `/ask`) с `{workspace: wid, page: pid, question: "В чем резюме этой страницы?"}`. Сервер принимает, проверяет, что user A имеет доступ хотя бы чтения к page pid (иначе отказ). Сервер собирает контекст: например, решаем, что ассистент ограничится текущей страницей. Берем текст страницы (из БД, либо у нас в объекте после правок, можно даже CRDT state преобразовать). Возможно, режем на куски по 1000 символов, делаем embedding запроса и каждого куска, сравниваем – или даже проще: берем весь текст, если он небольшой (<N tokens) и кормим напрямую в prompt без векторного поиска. Предположим, страница большая – тогда:
- Вычисляем embedding вопроса через модель (например, функция `getEmbedding(question)` – может вызывать внешнее API). Это 1 доп. сетевой вызов (~100ms, платный).
- В БД `Vectors` делаем `SELECT text FROM Vectors WHERE tenant_id = wid AND page_id = pid ORDER BY embedding <-> query_vec LIMIT 5`. Получаем 5 фрагментов текста.
- Формируем prompt: `"Вот выдержки из документа:\n1. ...\n2. ...\nВопрос: ... \nОтветь по существу."`.
- Отправляем в LLM API (OpenAI). Ждем ответ (например, стримим).
- Как только получили первые токены, можем переслать фронту (если установлено SSE или websockets). Если API не стримит, просто ждем полный.

- Фронт получает ответ (или постепенно отображает, как печатает ассистент). В результате А видит сгенерированный текст. При желании, А может нажать "Вставить в документ" – тогда фронт отправит событие через collab редактирование (создать новый блок с таким текстом), и это опять пойдет по обычной коллаборационной цепочке всем.

- Весь диалог хранится? Если у нас интерфейс как чат сбоку, то имеет смысл сохранять вопрос/ответ в таблицу `AiDialog` (user, page, question, answer, timestamp). А также, асинхронно, можно залогировать для разработчиков (таблица `AiLogs` с prompt и ID ответа).

- Если ассистент поддерживает функции: например, А спрашивает: "Найди мне страницу, где описывается бюджет проекта". Модель может решить вызвать функцию `searchDocuments("бюджет проекта")`. Сервер видит function_call, выполняет поиск по всем страницам workspace (embedding или полнотекст). Находит, допустим, страницу "Проект X Финансы". Отдает результат модели. Модель теперь отвечает: "Вот страница 'Проект X Финансы' содержит раздел по бюджету.". Сервер возвращает ответ А. И, может, предлагает ссылку (так как функция вернула конкретную страницу, можно фронтенду знать ID и дать ссылку-ку).

4. Публикация страницы:

- Пользователь-редактор нажимает "Share -> Publish to web". Фронтенд посылает API `POST /page/<id>/publish {public: true}`. Сервер проверяет: user имеет FullAccess на страницу? Если да, генерирует случайный `public_share_id` (UUID) и сохраняет в `Pages.public_share_id`. Это значит, что страница доступна по URL `/public/<uuid>`. Сервер может также сохранить настройки: `Pages.is_public = true, allowSearchIndex = false/true`.
- Теперь кто угодно по этому URL может GET страницу: фронтенд Next.js имеет динамический route, который по `getServerSideProps` определяет `public_share_id` из URL, стучится к API `GET /public/page/<uuid>`, сервер находит страницу и возвращает контент (без треб. auth, но проверяя uuid). Next.js рендерит HTML (возможно даже SSG, т.к. контент не изменится часто; но надо будет инвалдиировать при изменениях).
- Если страница изменится автором, должны изменения отражаться на публичной версии. Если SSR – они отразятся сразу при новом запросе. Если SSG – нужно либо регенерировать (ISR – Incremental Static Regeneration) либо заставить при изменении страницы дернуть Vercel revalidate API.
- Если в странице публичной есть вложения (изображения), нужно решить: делать их тоже публичными? Обычно – да, если страница опубликована, ее изображения тоже видны. Это значит или делать их URL публичными (S3 bucket policy open for those files), или проксировать через свой сервер, но это лишняя нагрузка. Проще – помечать файл как public при публикации страницы.

5. Удаление и восстановление:

- Пользователь удаляет страницу. Можно сразу удалить записи (каскадно все блоки, комменты). Но лучше флаг `deleted`. Например, `Pages.deleted_at = now()`. Клиент UI переносит страницу в раздел "Trash". Реализация: фронт может просто отфильтровывать deleted, а спецраздел "Корзина" показывает их.
- При восстановлении: `PUT /page/<id>/restore` -> сервер обнуляет `deleted_at`.

- Принудительно очистить корзину: удалить окончательно все, что старше N дней. Это либо крон-джоба (раз в день), либо по требованию пользователя "Empty Trash".
- Удаление блоков можно сразу из БД – они мелкие. Но возможно, стоит и для них `deleted`, если нужно на UI "undo". TipTap/Lexical обычно имеют client-side undo (Ctrl+Z) на недолгое время. Для глобального undo через историю версий у нас `BlockRevisions`.

6. Просмотр активности и аудита:

- Возможно, для пользователей полезен лог изменений страницы (кто что отредактировал). Это выводится на основе либо `BlockRevisions`, либо CRDT history. Можно сделать endpoint `/page/<id>/history` -> сервер агрегирует изменения по времени, группирует.
- Также, workspace администратор может иметь журнал аудита: "user X экспортировал данные", "user Y приглашен" – такие события логируются.

В итоге, все компоненты работают сообща: **фронтенд** обеспечивает удобный интерфейс и мгновенную отзывчивость, **бэкенд** гарантирует целостность данных, синхронизацию и доступ, **инфраструктура real-time** – доставка изменений без задержек, **векторные и AI сервисы** – интеллектуальные возможности, а **общие механизмы** – безопасность, логирование, масштабируемость – делают систему надежной и готовой к росту.

CI/CD, DevOps и эксплуатация

Разработав приложение, нужно наладить процессы сборки, доставки обновлений и мониторинга:

- **Контроль версий и сборка:** Используется репозиторий (GitHub/GitLab). Настроен pipeline CI: на каждый коммит или Pull Request запускаются автотесты (юнит-тесты на фронт и бэк, если есть, линтеры ESLint, компиляция TypeScript чтобы не было ошибок). В идеале, и e2e тесты (Cypress или Playwright) на основные сценарии – создание страницы, совместное редактирование (можно имитировать два пользователя с разными браузерами). Авто-тесты гарантируют, что изменения не сломали базовый функционал.

- **Непрерывная доставка (CD):** После прохождения тестов, изменения сливаются в основную ветку, что триггерит деплой. В Vercel это встроено – деплой фронта. Для бэкенда – может быть GitHub Actions: например, сборка Docker-образа и пуш на registry, затем вызов deploy (если Railway – через CLI, если k8s – `kubectl rollout`). При деплое, выполнять миграции БД (н-р, `prisma migrate`). Обязательно иметь **стратегию отката**: если новая версия вызывает много ошибок, уметь быстро вернуть предыдущую. На Kubernetes можно хранить старый replicaset, на Heroku – просто re-deploy старого slug, на Docker – иметь tag previous. В Vercel можно откатиться, переключив прод домен на предыдущий deployment (они хранят историю).

- **Фича-флаги:** Реализация feature toggle – через конфигурацию (ENV var) или сервис типа LaunchDarkly – позволяет включать/выключать функционал для % пользователей. Это помогает безопасно выкатить большие изменения: сначала включить для внутренней команды, потом 10% клиентов, потом всем. В коде это условные блоки.

- **DevOps: мониторинг и алертинг:** Как обсуждалось, внедряется централизованный лог (например, с помощью услуги Papertrail или Graylog, или встроенного в платформу – Railway предлагает простые логи). Метрики: если на Kubernetes – установить Prometheus + Grafana, иначе использовать cloud monitoring (Datadog metrics). Следить за CPU (если CPU близок к лимиту, приложение может не успевать обрабатывать WS, значит

масштабировать), RAM (чтобы не было утечек), DB connections (чтобы не исчерпались). В WebSocket-сервере можно мерить число подключений, и тоже репортить. AI – отслеживать частоту запросов, затраты (OpenAI $). Alerting: настроить оповещения (PagerDuty/Slack) на критичные события: падение сервера, рост 500 ошибок, DB недоступна. Также на безопасность: аномальные логины, множ. 401 ответов (возможен перебор паролей).

- **Обновление зависимостей и патчинг:** Вредно забывать обновлять библиотеки – фронтенд (React, etc.), бэкенд (Node security patches). Процесс: раз в спринт/месяц обновлять, прогонять тесты. В случае критичных уязвимостей (например, Log4Shell-стайл, или уязвимость XSS в редакторе) – быстро патчить и деплоить hotfix. Notion-стейк подразумевает много сторонних пакетов (Yjs, TipTap extensions, etc.), их тоже мониторить.

- **Бэкапы и восстановление:** Для БД настроить регулярные бэкапы (ежедневно полные + лог транзакций, если PG). Проверять, что можно восстановить (тестовые restore на staging). Файлы – если S3, там версии включить или отдельное хранение. Векторную базу – можно пересоздать из данных, но embeddings тоже желательно бэкапить (хотя их можно пересчитать, но долго).

- **Тестирование на нагрузку:** Перед расширением аудитории стоит провести load testing: имитировать, например, 100 одновременных пользователей редактируют 10 страниц – смотреть, как поведет себя система (CPU spikes? WS lag?). Профилировать узкие места (возможно, блокировки в DB – тогда оптимизировать транзакции; либо Node event loop – тогда расшивать на воркеры).

- **Развитие архитектуры:** По мере роста, могут потребоваться: разделение сервисов (microservices), например, вынести AI-сервис отдельно, вынести notifications отдельным процессом. Также, возможно, multi-region deployment. Еще одна тема – **desktop/mobile приложения**: Notion имеет и electron-приложение, и мобильные native. Наш архитектурный стек ориентирован на веб, но API и realtime можно повторно использовать: Electron просто будет встраивать ту же веб-страницу или использовать те же WS, mobile – можно на React Native подключаться к нашим WS и API. Поэтому важно спроектировать API стабильно, документировать его (OpenAPI/Swagger).

- **Dev/test data:** Для разработки удобно иметь генератор фейковых данных: скрипт, который создаст 5 воркспейсов, в каждом по 10 страниц со случайными блоками. Это позволит и в UI посмотреть поведение, и тесты наполнить. Use cases: например, большой документ (1000 блоков) – проверить производительность.

- **Пример DevOps-тулчейн:**

- Код: TypeScript (front/back), хранение GitHub.
- CI: GitHub Actions (npm run test, etc.).
- Container: Dockerfile multi-stage (build front, serve via Node/Express or static).
- Deploy: Kubernetes on cloud (AWS EKS). IaC: Terraform (создать DB RDS, Elasticache Redis, etc). CI/CD доставляет образы, ArgoCD применяет manifests.
- Monitoring: EKS + Prom/Grafana, Alerts via AlertManager -> Slack. Logging via ELK or AWS CloudWatch Logs.
- Sentry for error tracking front/back.

- On-call rotation if team >1, to handle incidents.

- **Analytics и продуктовые метрики:** Помимо системных метрик, стоит внедрить сбор событий: какие функции используют, сколько страниц создают, время активной работы. Это нужно продактам для улучшения. Можно использовать Mixpanel/Amplitude, либо собственное (собирая анонимно). Но осторожно с privacy – нужно пользовательское согласие, а для enterprise – опция отключить.

Заключая, построение аналога Notion – масштабный проект, затрагивающий множество технологий. Мы рассмотрели архитектуру **фронтенда** (сложный редактор + real-time взаимодействие), **бэкенда** (хранение дерева блоков, синхронизация, права, файлы, история), варианты инфраструктуры real-time (от Convex до кастомных WS+Redis), **AI-интеграцию** (pipelines embeddings, поиск по данным, многопользовательский контекст), а также **сквозные требования** (безопасность, многотенантность, масштабирование, DevOps). Все компоненты должны быть спаяны в единое целое: когда пользователь нажимает клавишу или спрашивает ассистента – система через несколько уровней тут же реагирует и выдает результат, оставаясь надежной и безопасной. Такой технический проект требует грамотной инженерной организации, но реализуем при использовании современных инструментов и подходов, обсужденных выше [30] [31].

---

1 2 Tiptap Collaboration - Collaborative Real Time Editor
https://tiptap.dev/product/collaboration

3 So Last-Write-Wins (LWW) basically _is_ a CRDT, but not in the sense that anyone... | Hacker News
https://news.ycombinator.com/item?id=38292682

4 5 16 Architectures for Central Server Collaboration - Matthew Weidner
https://mattweidner.com/2024/06/04/server-architectures.html

6 7 Realtime, all the time
https://www.convex.dev/realtime

8 9 10 11 12 13 29 31 javascript - Choosing DB model for an app similar to Notion, Block-based ("paragraphs") or document-based? - Stack Overflow
https://stackoverflow.com/questions/71024175/choosing-db-model-for-an-app-similar-to-notion-block-based-paragraphs-or-do

14 15 Going local-first with Automerge and Convex
https://stack.convex.dev/automerge-and-convex

17 18 19 20 21 22 Building successful multi-tenant RAG applications
https://www.thenile.dev/blog/multi-tenant-rag

23 get-convex/convex-backend: The open-source reactive database for ...
https://github.com/get-convex/convex-backend

24 Self-Hosting Realtime - Supabase
https://supabase.com/docs/reference/self-hosting-realtime/introduction

25 Realtime Postgres RLS now available on Supabase
https://supabase.com/blog/realtime-row-level-security-in-postgresql

26 Beyond the Hype: Why We Chose Redis Streams Over Kafka for Microservices Communication - DEV Community
https://dev.to/mtk3d/beyond-the-hype-why-we-chose-redis-streams-over-kafka-for-our-microservices-dmc

27 Scaling RAG Application to Production - Multi-tenant Architecture ...
https://www.reddit.com/r/Rag/comments/1n21nq1/scaling_rag_application_to_production_multitenant/

28 Security practices – Notion Help Center
https://www.notion.com/help/security-and-privacy

30 Let’s Build a Notion-Style Editor with Real-Time Collaboration Using Velt | by Mr. Ånand | JavaScript in Plain English
https://javascript.plainenglish.io/lets-build-a-notion-style-editor-with-real-time-collaboration-using-velt-cb3644ef6e51?gi=a62940d8c374