ChatGPT

NovaDocs – Modern Collaborative Wiki Platform

Architecture Overview

```
flowchart LR
    User["Web Browser (React client)"] -->|HTTPS| Frontend["**Next.js 13**
(React 19)"]:::frontend
    Frontend -->|GraphQL & REST| BackendAPI["**FastAPI** + Strawberry
GraphQL"]:::backend
    User -->|WebSocket| CollabServer["FastAPI Yjs Collab Server"]:::backend
    BackendAPI -->|SQL| Postgres[(PostgreSQL 16 + pgvector)]:::storage
    BackendAPI -->|Cache| Redis[(Redis 7.2 Pub/Sub)]:::storage
    BackendAPI -->|OIDC| Keycloak[[Keycloak (SSO)]]:::service
    BackendAPI -->|File Storage| MinIO[(MinIO S3 Bucket)]:::storage
    BackendAPI -->|Telemetry| Prometheus[(Prometheus)]:::service
    Prometheus --> Grafana[Grafana Dashboard]:::service

    classDef frontend fill:#cdeaff,stroke:#333,stroke-width:1px;
    classDef backend fill:#ffefd2,stroke:#333,stroke-width:1px;
    classDef storage fill:#ffdede,stroke:#333,stroke-width:1px;
    classDef service fill:#e2e2e2,stroke:#333,stroke-width:1px;
```

Figure: High-Level Architecture. NovaDocs is structured as a **self-hosted web app** with a modern React frontend and a scalable Python backend. Clients use a browser-based **Next.js 13** application (TypeScript & React 19) for a rich editing UI. The frontend communicates with the backend over **HTTPS** using a **GraphQL API** (for most data operations) and some REST endpoints (for file uploads, health checks, etc.). Real-time collaborative editing is powered by **WebSocket** connections to a collaboration service on the backend (using CRDT via **Yjs**). The backend is built with **FastAPI** (Python 3.12) and **Strawberry GraphQL**, handling both GraphQL queries/mutations and REST routes.

On the data layer, NovaDocs uses **PostgreSQL 16** as the primary database, taking advantage of extensions like pg_trgm for full-text search and **pgvector** for embedding-based semantic queries [1] [2]. A **Redis 7.2** instance is used for caching and pub/sub messaging to enable features like ephemeral collaboration presence and job queues. Binary assets (images, files) are stored via an S3-compatible service; in development NovaDocs uses **MinIO** on premise [3]. **Keycloak** (OpenID Connect) is integrated for single sign-on and identity management [4], allowing enterprise-grade authentication and user provisioning. The backend also includes instrumentation for observability: metrics are collected with **Prometheus** and visualized in **Grafana**, and the codebase is prepared to emit OpenTelemetry traces. This architecture is containerized for easy deployment.

In summary, the design follows a **microservice-like modular** approach within a monolithic app: the frontend app (Next.js) and backend API (FastAPI) operate separately but are deployed together. The backend serves the GraphQL/REST API and static files (for attachments), and manages real-time editing state via WebSockets. This separation of concerns, coupled with containerization, allows NovaDocs to **scale horizontally** – e.g. multiple FastAPI instances behind a load balancer (with sticky sessions or a distributed pub/sub for collaboration events). All components are self-hostable via Docker and Kubernetes for teams that want full control.

Monorepo Directory Structure

NovaDocs is organized as a **monorepo** for a seamless developer experience. Key directories include:

```
novadocs/              # Repository root (monorepo)
├── apps/
│   ├── frontend/      # Next.js 13 frontend app (React + TypeScript)
│   └── backend/       # FastAPI backend (Python 3.12 + Strawberry GraphQL)
├── packages/          # Shared packages (e.g. utils, types) for frontend/
backend
├── docs/              # Documentation (architecture, development,
deployment, API)
├── infrastructure/
│   └── docker/        # Docker compose files, Dockerfiles, Helm charts for
deployment
├── .github/workflows/ # CI/CD pipeline definitions (GitHub Actions)
├── package.json       # NPM workspaces config (frontend, etc.)
├── pyproject.toml     # Python project config (backend)
└── ...                # Config files (README, Makefile, .env.example, etc.)
```

Figure: Monorepo Structure. The repository is split into an `apps` directory for separate frontend and backend codebases, plus a `packages` directory for any shared libraries (for example, common TypeScript types or utility functions shared between frontend and backend). The `docs` folder contains supporting documentation such as architecture overviews and API references. An `infrastructure` folder holds deployment configurations (Docker Compose for development, and Helm charts or K8s manifests for production). A root Makefile provides common commands to build, test, and run the whole system. For instance, the Makefile uses Docker Compose to spin up Postgres, Redis, MinIO, etc., then starts the Next.js and FastAPI servers for development [5]. Both the frontend and backend can be developed and run locally with hot-reload, while sharing code via the monorepo setup. This structure ensures **fast DX** (developer experience) with clearly separated concerns and the ability to extend shared logic easily.

Sources: Monorepo layout inferred from repository (`apps/frontend` *and* `apps/backend` *exist* [6]*, and Docker setup is under* `infrastructure/docker` [7]*).*

Data Model and Relationships (ERD)

```
erDiagram
    User ||--o{ Workspace : "owns"
    Workspace ||--|| User : "owned by"
    User ||--o{ WorkspaceMember : "memberships"
    Workspace ||--o{ WorkspaceMember : "members"
    Workspace ||--o{ Page : "pages"
    Page ||--o{ Page : "subpages"
    Page ||--o{ Block : "blocks"
    Block ||--o{ Block : "sub-blocks"
    Page ||--o{ Database : "databases"
    Database ||--o{ DatabaseRow : "rows"
    Workspace ||--o{ Asset : "assets"
    User ||--o{ Asset : "uploads"
    Workspace ||--o{ Permission : "permissions"
    User ||--o{ Permission : "permissions"
```

Figure: Entity-Relationship Diagram. NovaDocs organizes content in **Workspaces**, each owned by a User and containing many Pages. Users can belong to multiple workspaces through a membership table, with roles per workspace (e.g. admin, editor, viewer). Pages are hierarchical (a page can have sub-pages infinitely nested) and contain structured content as blocks. The **Page** entity has a self-referential relationship to model a page tree (one parent page to many child pages) [8]. Each Page can also contain one or more **Blocks** (which represent rich text content or embedded objects); Blocks themselves support nesting (for example, a list block may have child blocks for each list item) [9]. Some pages can serve as **Database** containers – analogous to Notion's database pages – which are special pages that have associated **Database** records. A Database defines a schema (custom fields/columns) and holds many **DatabaseRow** entries (each row is one record in that database) [10] [11].

Other supporting entities include **Asset** (files uploaded to the workspace, such as images or documents) which are linked to both the user who uploaded them and the workspace they belong to [12]. There is also a **Comment** entity (not shown above for brevity) linking users to specific page or block content for inline discussions. Access control is handled by a flexible **Permission** model: permissions can be granted at the workspace or page level. A Permission record ties a user (or group/role) to a resource (workspace or page) with a certain access level (read, write, admin, etc.) [13] [14]. Additionally, NovaDocs supports shareable links: a **ShareLink** entity (not shown in diagram) represents a token granting temporary read-access to a specific page or resource, with an expiry time and limited permissions.

In summary, the data model is centered on **Wiki-style pages and sub-pages**, augmented by **structured data** (databases with rows), and all tied together with a robust workspace/membership and permission system for multi-tenant isolation. This design ensures that all pages, assets, and data rows are **partitioned by workspace** – i.e., content in one workspace cannot be accessed by users outside of it, unless explicitly shared via a permission or share link.

Sources: The SQLAlchemy models define these relationships (e.g., a `Workspace` *has many* `pages` [15]*; a* `Page` *has* `children` *and* `blocks` [16]*; a* `Database` *has many* `rows` [17]*; etc.). The unique constraint on pages (*`uq_workspace_slug`*) ensures each page slug is unique per workspace* [18]*, reinforcing workspace isolation. Permission model links user/workspace to resource id* [19] [14].

API Specification

NovaDocs provides both a **GraphQL API** for most application operations and a minimal **REST API** for certain endpoints (file uploads, health checks, etc.). Real-time collaboration uses a WebSocket protocol (separately documented in the next section). Below is an outline of the GraphQL schema (SDL) covering authentication, page CRUD, and live update subscriptions, followed by examples of key REST endpoints.

GraphQL Schema (SDL)

```
# Queries (read operations)
type Query {
  me: User                       # Current logged-in user info (or null if
not authenticated)
  workspace(slug: String!): Workspace      # Get a workspace by slug (must
be a member)
  page(id: UUID!): Page
# Fetch a page by ID (if user has access)
  searchPages(query: String!, workspaceId: UUID!): [Page!]!    # Full-text
search within pages
  searchBlocks(query: String!, workspaceId: UUID!): [Block!]! # Full-text
search within blocks
}

# Mutations (write operations)
type Mutation {
  # Workspace management
  createWorkspace(input: CreateWorkspaceInput!): Workspace!
  updateWorkspace(id: UUID!, input: UpdateWorkspaceInput!): Workspace!
  deleteWorkspace(id: UUID!): Boolean!
  # Page (wiki) CRUD
  createPage(input: CreatePageInput!): Page!
  updatePage(id: UUID!, input: UpdatePageInput!): Page!
  deletePage(id: UUID!): Boolean!
  movePage(id: UUID!, parentId: UUID, position: Int!): Page!    # Re-parent
or reorder a page
  # Block (content) CRUD
  createBlock(input: CreateBlockInput!): Block!
  updateBlock(id: UUID!, input: UpdateBlockInput!): Block!
  deleteBlock(id: UUID!): Boolean!
  # Database and Row CRUD
  createDatabase(input: CreateDatabaseInput!): Database!
  updateDatabase(id: UUID!, input: UpdateDatabaseInput!): Database!
  createDatabaseRow(input: CreateDatabaseRowInput!): DatabaseRow!
  updateDatabaseRow(id: UUID!, input: UpdateDatabaseRowInput!): DatabaseRow!
  deleteDatabaseRow(id: UUID!): Boolean!
  # Comments and Permissions
  createComment(input: CreateCommentInput!): Comment!
  updateComment(id: UUID!, input: UpdateCommentInput!): Comment!
  deleteComment(id: UUID!): Boolean!
  grantPermission(input: GrantPermissionInput!): Permission!    # Share page
or set workspace role
  createShareLink(input: CreateShareLinkInput!): ShareLink!
}

# Subscriptions (real-time updates)
type Subscription {
  pageUpdated(pageId: UUID!): Page!           # Stream updates when a page's
content or metadata changes
```

```
  blockUpdated(pageId: UUID!): Block!        # Stream updates for blocks in
a page (e.g., new block added)
  commentAdded(pageId: UUID!): Comment!      # Notify when a new comment is
added to a page
  cursorUpdate(pageId: UUID!): CursorUpdate! # Real-time cursor movement in
a collaborative editing session
}

# Key type definitions
type User {
  id: ID!
  name: String!
  email: String!
  avatarUrl: String
  workspaces: [Workspace!]!       # Workspaces this user is a member of
}

type Workspace {
  id: ID!
  name: String!
  slug: String!
  description: String
  members: [WorkspaceMember!]!    # Users and their roles in this workspace
  pages: [Page!]!                 # Top-level pages in this workspace
  owner: User!                   # Workspace owner (creator)
}

type Page {
  id: ID!
  title: String!
  slug: String!
  workspace: Workspace!
  parent: Page        # Parent page (for nested pages; null if top-level)
  children: [Page!]!  # Subpages nested under this page
  blocks: [Block!]!   # Content blocks in this page (text, embeds, etc.)
  databases: [Database!]!  # Structured databases defined in this page (if
any)
  createdBy: User!
  metadata: JSON      # Additional page metadata (e.g., icon, cover image,
etc.)
  isTemplate: Boolean!
  createdAt: DateTime!
  updatedAt: DateTime!
}

type Block {
  id: ID!
  page: Page!
  parentBlock: Block         # Parent block if this block is nested
  childBlocks: [Block!]!     # Any nested sub-blocks
  type: String!              # e.g., "paragraph", "todo", "image", etc.
  data: JSON!                     # Content data (text or embed payload)
  properties: JSON               # Additional properties (e.g., formatting,
attributes)
  createdAt: DateTime!
  updatedAt: DateTime!
}

type Database {
  id: ID!
  page: Page!                     # The page that contains this database
  name: String!
  schema: JSON!                   # Definition of columns/fields (name, type,
etc.)
  views: JSON!                    # Saved views (table, kanban, calendar
configurations)
  rows: [DatabaseRow!]!
}

type DatabaseRow {
  id: ID!
  database: Database!
  data: JSON!                     # Key-value data matching the database schema
  position: Int
  createdAt: DateTime!
  updatedAt: DateTime!
}

type Comment {
  id: ID!
  page: Page
  block: Block
  user: User!
  content: String!
  createdAt: DateTime!
}

type WorkspaceMember {
  id: ID!
  workspace: Workspace!
  user: User!
  role: String!     # e.g., "admin", "editor", "viewer"
  joinedAt: DateTime!
}

type Permission {
  id: ID!
  resourceType: String!    # e.g., "page" or "workspace"
  resourceId: ID!          # ID of the page or workspace
  workspace: Workspace!    # Workspace context for the resource
  user: User              # User who has this permission (null for public
link)
```

```
  permissionType: String!  # e.g., "read", "write", "admin"
}

type ShareLink {
  id: ID!
  token: String!
  resourceType: String!
  resourceId: ID!
  permissions: JSON!       # Permissions granted by this link (e.g., read-
only)
  expiresAt: DateTime
}

type CursorUpdate {
  userId: ID!
  pageId: ID!
  position: Int!           # Caret position or selection start
  selection: JSON!         # Selection range or additional cursor metadata
}

# Input types for mutations (examples)
input CreateWorkspaceInput {
  name: String!
  slug: String!
  description: String
}

input CreatePageInput {
  workspaceId: ID!
  parentId: ID
  title: String!
  isTemplate: Boolean
  position: Int
}
```

Figure: GraphQL schema (excerpt). The GraphQL API is the primary interface for clients. It includes **query** fields for fetching data (the current user, pages, workspaces, and search results), **mutation** fields for creating/updating/deleting content, and **subscription** fields for receiving live updates. All operations require appropriate authentication and permission checks. For example, the `me` query returns the currently authenticated user (or null if not logged in), using the request's auth token to identify the user [20]. Most mutations require the user to be a workspace member or page collaborator; if a user is not authenticated or lacks access, a `PermissionError` is raised by the resolver [21].

Notable GraphQL types: - **User, Workspace, WorkspaceMember** reflect the collaboration model. A `Workspace` has a slug and list of members with roles. - **Page** is the core content node (with hierarchical children). Each Page stores its content in two forms: as structured blocks and (for search) as plain text or vector embeddings (the `contentText` and `contentVector` fields in the model [22] [18], not all exposed directly in GraphQL). - **Block** represents an individual content block (paragraph, heading, image, etc.), possibly nested under another block. - **Database** and **DatabaseRow** implement the notion of structured data within pages: a Database is essentially a collection of rows with a uniform schema (similar to a table). - **Comment** allows attaching discussion to a page or block. - **Permission** and

ShareLink support the permissions system (though typically, permission management might also be done via workspace membership and shareable links rather than direct GraphQL objects). - **CursorUpdate** is a custom type for real-time collaboration, describing a user's cursor position in a page for presence indication.

Real-time subscriptions: The schema defines several subscription channels. For example, clients may subscribe to `pageUpdated(pageId: ID!)` to be notified whenever a given page's content changes. Under the hood, the backend yields updated Page objects on this channel (likely triggered by collaboration events or manual edits) [23] . Similarly, `commentAdded(pageId: ID!)` streams new comments, and `cursorUpdate(pageId: ID!)` streams live cursor movements (for showing collaborators' cursors) [24] . The subscription resolvers are implemented with Strawberry's async generator pattern, and would be backed by a pub/sub system (e.g., Redis or NATS) to broadcast updates from one user's edit to all other subscribers [25] . (In the current code, these yield dummy placeholders and would be completed in production with actual pub-sub integration.)

Authentication & Authorization: The GraphQL endpoint expects a JWT or OIDC token for authenticated requests (e.g., in the `Authorization` header). FastAPI and Strawberry GraphQL integrate to populate `info.context["request"]` with user info, so each resolver can enforce auth. For instance, the `workspace` query fetches a workspace by slug only after confirming the requester is logged in and a member of that workspace [26] . Most mutations similarly do `get_current_user` and raise errors if not permitted. This ensures **role-based access control** at the API layer in addition to the database level checks.

REST API (Representative Endpoints)

While GraphQL covers most data operations, NovaDocs includes a few RESTful endpoints for specific purposes like file upload and health checks (and possibly for compatibility with external tools). Below are examples of important REST endpoints:

```python
# Health check (liveness) endpoint
@router.get("/health")
async def health_check():
    return {
        "status": "healthy",
        "version": "1.0.0",
        "timestamp": "<server-time>",
        "services": {
            "database": "healthy",
            "redis": "healthy",
            "storage": "healthy",
        }
    }
```

Example: A GET request to `/api/v1/health` returns a JSON payload indicating the application status (useful for Kubernetes liveness checks or monitoring) [27] . It reports the overall status, version, timestamp, and the health of sub-services (database, cache, storage, etc.). This does not require authentication.

```
# File upload endpoint (multipart/form-data)
@router.post("/upload")
async def upload_file(
    file: UploadFile = File(...),
    workspace_id: str = Form(...),
    current_user: User = Depends(get_current_user)
):
    """Upload file endpoint."""
    # ... (save file to storage, omitted)
    return {
        "id": "<generated-file-id>",
        "filename": file.filename,
        "url": f"/uploads/{file.filename}",
        "size": file.size,
        "mime_type": file.content_type
    }
```

Example: A POST to `/api/v1/upload` (authenticated) allows uploading a file. The client sends a multipart form with the file and target workspace. The server (after validating `current_user` via dependency injection) would save the file to the storage backend (MinIO or cloud S3 bucket) and respond with a JSON containing the file's metadata and a URL [28]. Currently this is implemented as a stub (simply echoing the file info) – in a real deployment, it would generate a unique ID/path, store the file, and perhaps perform virus scanning or image resizing as needed.

Other REST endpoints could include an export function (e.g. GET `/pages/{id}/export?format=markdown` to download a page as Markdown or PDF – a stub for this exists [29]) and possibly webhooks or integration callbacks if `FEATURE_WEBHOOKS_ENABLED` is true. However, the core philosophy is to expose most functionality via GraphQL for a unified API experience.

Key Code Samples and Implementation Details

This section provides key implementation snippets corresponding to NovaDocs's unique features, to guide developers in understanding and extending the system.

Real-Time Collaboration (CRDT + WebSockets)

NovaDocs uses **Conflict-free Replicated Data Types (CRDTs)** via **Yjs** for real-time collaborative editing. Each page's content is maintained as a Yjs document (which can be serialized/stored in the `content_yjs` field in the database [22]). Clients use the Yjs client library with a WebSocket provider (e.g., Hocuspocus or `y-websocket`) to sync edits. On the backend, a WebSocket server coordinates these updates. Below is a simplified snippet of the WebSocket collaboration server using FastAPI:

```
# apps/backend/src/api/websocket/collaboration.py (excerpt)
@router.websocket("/collaboration/{room_id}")
async def ws_collaboration(websocket: WebSocket, room_id: str, token: str =
None):
    # Accept connection and add to room
    user = authenticate_token(token)  # (Pseudo-code: validate the token if
provided)
    user_info = { "id": user.id if user else "anon", "name": user.name if
user else "Guest", "color": "#3B82F6" }
    await collaboration_manager.connect(websocket, room_id, user_info)
    try:
        while True:
            data = await websocket.receive_text()
            message = json.loads(data)
            # Broadcast cursor movements to others
            if message.get("type") == "cursor_update":
                await collaboration_manager.broadcast_to_room(
                    room_id,
                    { "type": "cursor_update", "user": user_info, "position":
message.get("position"), "selection": message.get("selection") },
                    exclude=websocket
                )
            # Broadcast content changes to others
            elif message.get("type") == "content_update":
                await collaboration_manager.broadcast_to_room(
                    room_id,
                    { "type": "content_update", "user": user_info, "content":
message.get("content") },
                    exclude=websocket
                )
            # ... handle other message types (e.g., awareness, comments) ...
    except WebSocketDisconnect:
        collaboration_manager.disconnect(websocket)
```

In this code, clients connect to `ws://<server>/ws/collaboration/{pageId}` (with an auth token) to join a collaboration "room" for that page. The server keeps track of active connections in `collaboration_manager`. When a client sends a `cursor_update` or `content_update` message, the server relays it to all other connections in the same room [30] [31]. This allows all collaborators to see each other's edits and cursors in real time. The actual content merging is handled by Yjs on the client side – each `content_update` might contain a diff or Yjs update that the clients apply to their local document state. The server here is essentially a relay; it doesn't interpret the document (thus it's scalable and fairly stateless). Persisting the CRDT state is done by saving Yjs updates periodically to the `Page.content_yjs` field (for example, on editor blur or a snapshot interval).

Integration with Yjs: In development, NovaDocs currently uses the **Hocuspocus** WebSocket provider (a Node.js server specialized for Yjs) on the client side [32] [33]. The Python snippet above shows how one could implement a Yjs-compatible server in FastAPI. To scale collaboration across multiple backend instances, one would use a shared pub/sub (like Redis or NATS) so that a message from any node is broadcast to subscribers on all nodes. (This is hinted in the code with comments about using Redis pub/sub for the subscription implementation [25].) In a production setup, you might run a separate collaboration service (e.g., a Node.js Hocuspocus server or Yjs's `y-websocket` server) alongside FastAPI, or extend the FastAPI app with such capabilities using `python-socketio` or similar. The current design keeps the WebSocket handling lightweight – essentially funneling messages and relying on CRDT for consistency – which makes horizontal scaling feasible (with sticky sessions or a message broker to synchronize state).

Finally, the backend app is launched via Uvicorn to serve HTTP and WS endpoints. For example, the entrypoint runs `uvicorn.run("src.main:app", host=..., port=..., reload=...)` to start the server [34]. Thus, all GraphQL, REST, and WebSocket routes are served by one FastAPI application for simplicity.

Tiptap Editor Extension – “DatabaseView” Node

On the frontend, NovaDocs uses **Tiptap (a rich-text editor toolkit)** to implement the editor with custom block types. A notable custom extension is the **DatabaseView** node, which represents an embedded database/table within a page. This extension allows users to view and manipulate structured data (rows) in the context of a page, similar to Notion’s database blocks. Below is a simplified version of how a Tiptap node extension for a database might look:

```
import { Node, mergeAttributes } from '@tiptap/core'

export const DatabaseView = Node.create({
  name: 'databaseView',
  group: 'block',       // Behaves like a block-level node
  atom:
true,           // Treated as an indivisible unit (no child content in
ProseMirror terms)
  addAttributes() {
    return {
      databaseId: { default: null },       // Reference to the database
entity
      viewType: { default: 'table' }       // e.g., 'table', 'kanban',
'calendar'
    }
  },
  parseHTML() {
    return [
      {
        tag: 'div[data-database-view]',    // Recognize this node from a
specific HTML tag/attribute
        getAttrs: dom => ({
          databaseId: dom.getAttribute('data-database-id'),
          viewType: dom.getAttribute('data-view-type') || 'table'
        })
      }
    ]
  },
  renderHTML({ node, HTMLAttributes }) {
    return ['div',
      {
        'data-database-view': '',
        'data-database-id': node.attrs.databaseId,
        'data-view-type': node.attrs.viewType,
        ...HTMLAttributes
      },
      0
    ]
  },
  addNodeView() {
    // This is where you'd integrate a frontend framework component for the
node's UI
    return ({ node, getPos, editor }) => {
      const container = document.createElement('div')
      container.dataset.databaseView = ''
      container.contentEditable = 'false'           // Make the whole node
non-editable as one unit
      // You could mount a React component here that takes
node.attrs.databaseId and displays the database
      container.innerText = `📊 Database [id=${node.attrs.databaseId}]`
      return { dom: container }
    }
  }
})
```

In this snippet, we define a new Tiptap node called `databaseView`. It’s an **atom** (no editable content within it, the entire node is managed as one piece). It carries attributes like a `databaseId` to know which database it represents and a `viewType` to switch between table/kanban/calendar views. The `parseHTML` and `renderHTML` functions tell Tiptap how to serialize this node to HTML (using a custom `div` with data attributes) so it can be saved/loaded. The `addNodeView` method is key: it can inject a custom React/Vue component. In NovaDocs, this would mount a **React component** that fetches the database rows and displays a table or board UI within the editor. For example, the node view might render a table with rows and columns as specified by `schema`. Because it’s marked `contentEditable=false`, user interactions (like editing a cell) might be handled by the React component and then propagate changes (e.g., through controlled inputs that call GraphQL mutations to update a DatabaseRow). This mechanism makes the editor extensible – the “slash command” menu (triggered by typing `/` in an empty block [35]) likely offers an option to insert a DatabaseView node, among other blocks.

Overall, the custom extension system allows NovaDocs to support **embedded dynamic content**. Similar extensions could be made for other block types (like a Kanban board view, a code block with syntax highlight, Mermaid diagrams, etc.). The provided structure ensures that these blocks are collaborative (since their data ultimately resides in the backend, multiple users can see updates) and easily serializable.

Database Schema Migration (Pages & Blocks)

NovaDocs uses Alembic for database migrations. The initial migration defines tables for all core entities. Below is an excerpt of the SQL DDL for the **Pages** and **Blocks** tables from the Alembic migration [36] [37]:

```
op.create_table('pages',
    sa.Column('id', postgresql.UUID(as_uuid=True), primary_key=True,
default=uuid.uuid4),
    sa.Column('title', sa.String(500), nullable=False),
    sa.Column('slug', sa.String(200), nullable=False),
    sa.Column('workspace_id', postgresql.UUID(as_uuid=True),
sa.ForeignKey('workspaces.id', ondelete='CASCADE'), nullable=False),
    sa.Column('parent_id', postgresql.UUID(as_uuid=True),
sa.ForeignKey('pages.id', ondelete='CASCADE')),
    sa.Column('created_by_id', postgresql.UUID(as_uuid=True),
sa.ForeignKey('users.id', ondelete='CASCADE'), nullable=False),
    sa.Column('metadata', postgresql.JSONB(), server_default='{}',
nullable=False),
    sa.Column('content_yjs', sa.Text()),          # Yjs CRDT document
(optional, for collaborative content)
    sa.Column('content_text', sa.Text()),         # Plain text content (for
full-text search indexing)
    sa.Column('content_vector', postgresql.ARRAY(sa.Float)),  # Vector
embeddings for semantic search
    sa.Column('position', sa.Integer, server_default='0', nullable=False),
    sa.Column('is_template', sa.Boolean, server_default='false',
nullable=False),
    sa.UniqueConstraint('workspace_id', 'slug', name='uq_workspace_slug'),
    # created_at & updated_at timestamps...
)

op.create_table('blocks',
    sa.Column('id', postgresql.UUID(as_uuid=True), primary_key=True,
default=uuid.uuid4),
    sa.Column('page_id', postgresql.UUID(as_uuid=True),
sa.ForeignKey('pages.id', ondelete='CASCADE'), nullable=False),
    sa.Column('type', sa.String(50), nullable=False),
    sa.Column('data', postgresql.JSONB(), server_default='{}',
nullable=False),
    sa.Column('properties', postgresql.JSONB(), server_default='{}'),
    sa.Column('position', sa.Integer, server_default='0', nullable=False),
    sa.Column('parent_block_id', postgresql.UUID(as_uuid=True),
sa.ForeignKey('blocks.id', ondelete='CASCADE')),
    # created_at & updated_at timestamps...
)
```

Explanation: Each **Page** has a UUID primary key, a title and slug, and foreign keys linking it to its workspace, its parent page (for hierarchy), and the user who created it [38]. We store `metadata` as a JSONB for arbitrary page properties (this could include icon, cover image, or other Notion-like page settings). The `content_yjs` field (Text) can store the serialized Yjs document state of the page for persistence, while `content_text` is used to store a plaintext representation (updated via triggers or application logic) to facilitate full-text search (with a GIN index on this column [18]). There is also a `content_vector` which is an array of floats – this will be converted to the PostgreSQL **VECTOR** type (1536 dimensions, e.g., using OpenAI embeddings) to enable semantic search queries [39] [40]. We index this with ivfflat for efficient similarity search. The `position` field is for ordering pages under the same parent, and `is_template` marks template pages.

The **Block** table similarly has an `id`, and a foreign key `page_id` to the page it belongs to [41]. Blocks also have an optional `parent_block_id` for nested blocks (allowing blocks tree two levels or more deep). The `type` field differentiates block kinds (text, image, etc.), and `data` and `properties` are

JSONB columns where the block's content and settings are stored (JSONB allows flexible schema for different block types). For example, for a text block, `data` might contain the text string; for an image block, `data` could contain a file reference; for a to-do block, `properties` might include a "checked" boolean.

All tables include `created_at` and `updated_at` timestamp columns (with timezone) via a Timestamp mixin [42], which are set to default to current time and auto-update on modifications. Foreign keys use `ondelete="CASCADE"` so that deleting a workspace will cascade to its pages, and deleting a page cascades to its sub-pages, blocks, etc., ensuring referential integrity. There are also indexes not shown above (see the migration file) to optimize common queries: e.g., an index on `pages(parent_id)` to quickly fetch children pages, on `blocks(page_id)` to fetch all blocks of a page, and a GIN index on `content_text` for full-text search [43] [44].

This database schema is designed to balance **flexibility** (using JSON for variable fields and content) with **queryability** (key fields are indexed and properly normalized for relationships). As the product evolves (e.g., adding more structured field types or references between entities), migrations will adjust this schema accordingly, but the core entities (Workspace, Page, Block, etc.) provide a stable backbone.

Offline-First Sync – `usePage` React Hook

To create a seamless user experience, NovaDocs employs an **offline-first** approach in the frontend. The React app can optimistically update the UI and cache changes locally, then synchronize with the server in the background. A good example is the custom React hook `usePage(pageId)` which encapsulates page state fetching and syncing logic. Below is a simplified version highlighting how it handles optimistic updates, auto-save, and real-time sync:

```
function usePage(pageId: string) {
  const queryClient = useQueryClient()
  const [localChanges, setLocalChanges] = useState<Partial<Page>>({})
  const [saveTimeout, setSaveTimeout] = useState<NodeJS.Timeout | null>(null)
  const [isOnline, setIsOnline] = useState(navigator.onLine)

  // Fetch page data (initial load) using react-query (omitted for brevity)

  // Mutation to update page on server
  const updatePageMutation = useMutation(async (updates: UpdatePageInput) =>
{
      // Send GraphQL mutation to update page on server...
      const updatedPage = await graphqlClient.request(UPDATE_PAGE_MUTATION,
{ id: pageId, input: updates })
      return updatedPage
    }, {
      onSuccess: (updatedPage) => {
        // Merge server response into cache
        queryClient.setQueryData(['page', pageId], (old: Page) =>
({ ...old, ...updatedPage }))
        setLocalChanges({})  // clear localChanges after successful save
        // Notify other clients about the update via WebSocket
        send('page_updated', { pageId, updates: updatedPage, userId:
currentUser.id })
```

```
      },
      onError: () => {
        console.error("Failed to update page")
        // On error, rollback cache to server state
        queryClient.invalidateQueries(['page', pageId])
      }
  })

  // Optimistic update function
  const updatePage = useCallback((updates: Partial<Page>) => {
    if (optimistic) {
      // Apply optimistic updates locally
      queryClient.setQueryData(['page', pageId], (old: Page) => ({
        ...old,
        ...updates,
        updatedAt: new Date().toISOString()
      }))
      setLocalChanges(prev => ({ ...prev, ...updates }))
    }
    if (autoSave) {
      // Schedule a debounced save after some delay
      if (saveTimeout) clearTimeout(saveTimeout)
      const timeout = setTimeout(() => {
        if (isOnline) {
          updatePageMutation.mutate(updates)
        } else {
          queueOfflineUpdate(updates)  // store in IndexedDB or localStorage
for later sync
        }
      }, saveDelay)
      setSaveTimeout(timeout)
    }
  }, [pageId, optimistic, autoSave, isOnline, saveTimeout])

  // Sync offline updates when back online
  useEffect(() => {
    const handleOnline = () => { setIsOnline(true); processOfflineQueue() }
    const handleOffline = () => setIsOnline(false)
    window.addEventListener('online', handleOnline)
    window.addEventListener('offline', handleOffline)
    return () => {
      window.removeEventListener('online', handleOnline)
      window.removeEventListener('offline', handleOffline)
    }
  }, [pageId])

  // Subscribe to collaboration updates via WebSocket
  useEffect(() => {
    if (!pageId) return
    const unsubscribe = subscribe('page_updated', data => {
      if (data.pageId === pageId) {
        // Merge updates from other users/collaborators into local state
        queryClient.setQueryData(['page', pageId], (old: Page) => ({
          ...old,
          ...data.updates
        }))
      }
    })
    return () => { unsubscribe() }
  }, [pageId, subscribe, queryClient])

  // ... return page state and the updatePage function ...
}
```

How it works: When `usePage` is used in a component, it will initially fetch the page's data via GraphQL (using `react-query` for caching). The hook defines an `updatePage` function that UI code can call to apply edits (like changing the page title or content). This function immediately applies the changes to the local React state and cache (that's the optimistic update: it calls `setQueryData` to merge changes into the cached page data) [45]. It also records the changes in a `localChanges` state in case we need to rollback. Then, if auto-save is enabled, it starts a timer (debounce via `setTimeout` for, say, 1 second) to actually send these changes to the server [46]. If additional changes come in quickly, the timer resets to avoid flooding the server.

If the app is offline (`navigator.onLine` is false), `updatePage` will not attempt a network mutation; instead it queues the update locally (e.g., using `localStorage` or IndexedDB) for later syncing. The hook listens to browser online/offline events to trigger `processOfflineQueue()` when connectivity is restored, which will replay any queued updates by sending them to the server [47] [48]. This allows a user to continue editing even without network, and their changes will sync once back online.

The hook also integrates with the collaboration WebSocket: it subscribes to `"page_updated"` messages for the same pageId [49]. When another user's changes come in (the backend or one of the clients will emit a WebSocket event on saves), the hook merges those external updates into the local state, so the current user sees others' edits in near real-time. This is effectively **eventual consistency**: each client optimistically applies its own edits and also applies others' edits as they arrive, keeping the page state in sync across collaborators. Conflicts are minimized by the CRDT at the text editor level (for block content), and by last-write-wins on discrete fields like title or metadata (the backend updates `updatedAt` to help ordering).

Result: The UI remains responsive and data is never lost. Users get immediate feedback for their own actions and up-to-date views of teammates' actions. If something fails (e.g., a save fails on the server), the code invalidates the query to refetch authoritative data from the server, ensuring eventual correctness. This offline-first approach, combined with CRDT for the editor, enables a **robust collaborative experience** similar to Notion or Google Docs – even with flaky internet, users can keep working and trust that everything will sync and merge properly.

Sources: The logic above is adapted from the actual `usePage` *implementation* [45] [46] *which shows optimistic cache updates and delayed save, and the WebSocket subscription for* `"page_updated"` [49] *that merges remote updates. The offline queuing mechanism uses localStorage to store pending updates when offline* [47] .

DevOps & Deployment

Docker Compose (Development) Setup

For development and testing, NovaDocs provides a **Docker Compose** setup that brings up all dependent services quickly. The compose file defines services for Postgres, Redis, MinIO, Keycloak, etc., so that a developer can run `docker-compose -f infrastructure/docker/development/docker-compose.yml up` and have all backend dependencies running locally [5]. Key services from the compose file include:

```
services:
  postgres:
    image: pgvector/pgvector:pg16         # PostgreSQL 16 with pgvector
extension pre-installed
    environment:
      POSTGRES_DB: novadocs
      POSTGRES_USER: novadocs
      POSTGRES_PASSWORD: novadocs
    ports:
      - "5432:5432"
    volumes:
      - postgres_data:/var/lib/postgresql/data
    # ... healthcheck omitted ...
  redis:
    image: redis:7.2-alpine
    ports:
      - "6379:6379"
    command: redis-server --appendonly yes --maxmemory 512mb --maxmemory-
policy allkeys-lru
    volumes:
      - redis_data:/data
  minio:
    image: minio/minio:latest
    environment:
      MINIO_ROOT_USER: minioadmin
      MINIO_ROOT_PASSWORD: minioadmin
    ports:
      - "9000:9000"        # S3 API
      - "9001:9001"        # MinIO console
    volumes:
      - minio_data:/data
    command: server /data --console-address ":9001"
  keycloak:
    image: quay.io/keycloak/keycloak:22.0
    environment:
      KEYCLOAK_ADMIN: admin
      KEYCLOAK_ADMIN_PASSWORD: admin
      KC_DB: postgres
      KC_DB_URL: jdbc:postgresql://postgres:5432/keycloak
      KC_DB_USERNAME: novadocs
      KC_DB_PASSWORD: novadocs
      # ... other Keycloak configs ...
    ports:
      - "8080:8080"
    depends_on:
      postgres:
        condition: service_healthy
    command: start-dev
  # Additional services: Elasticsearch (optional for advanced search),
Prometheus, Grafana for monitoring
volumes:
  postgres_data:
  redis_data:
  minio_data:
  keycloak_data:
  # ...
```

When running in development mode, developers typically start this stack (`docker-compose up -d`) then run `npm run dev` for the Next.js app and `uvicorn src.main:app --reload` for the FastAPI app (the Makefile automates this [50]). With the above services: - **Postgres** is running on localhost:5432 with a ready database (the compose mounts an init script to set up extensions like uuid-ossp, pg_trgm, vector) [1]. - **Redis** is on localhost:6379 for caching/pubsub. - **MinIO** is on localhost:9000 (console on 9001) with default credentials, providing an S3 API endpoint for file uploads (the app is configured with these via environment variables). - **Keycloak** on localhost:8080 provides an authentication UI and token issuance. In dev, Keycloak is configured with a demo realm/client for NovaDocs; the backend would validate JWTs from this IdP using the OIDC config (client id/secret and JWKS, configured via environment variables in `.env`) – this enables testing SSO flows locally.

The compose file also includes **Elasticsearch** (on 9200) – while Postgres covers search, an optional Elasticsearch container is present if one wanted to use Elastic for advanced full-text or as a vector store (this might be experimental or for future use). Additionally, **Prometheus** (9090) and **Grafana** (Grafana on 3001 in dev) are included to monitor the application. Grafana is preconfigured (via a provisioned datasource config) to pull metrics from Prometheus, which scrapes metrics from the FastAPI app (if `PROMETHEUS_ENABLED=true`, the app likely exposes metrics at `/metrics`). This setup lets developers observe performance and load even in dev.

For production, NovaDocs can be containerized and orchestrated on Kubernetes. A Helm chart is likely provided (or can be created) to deploy the `frontend` and `backend` containers, plus the dependencies as either external services or in-cluster pods. The architecture supports scaling the backend horizontally (stateless FastAPI processes with shared Postgres/Redis) and similarly the Next.js frontend can be served from a CDN or Node server.

CI/CD Pipeline (GitHub Actions)

NovaDocs employs a **CI pipeline** to ensure code quality and automate deployments. The pipeline (e.g., a GitHub Actions workflow) would include the following steps:

- **Lint & Test:** On each push or pull request, run the frontend linter/tests and backend linters/tests. For example, use Node 18 to install and run `npm run lint && npm run test` in `apps/frontend`, and Python 3.12 to run `flake8`, `black --check`, `isort --check` and `pytest` for `apps/backend` [51] [52]. A Postgres service can be spun up in CI to run integration tests (with `pytest` connecting to it; since the tests likely use an in-memory or a Docker DB, GitHub Actions allows adding `services: postgres:...` easily). These tests ensure that no commit breaks existing functionality.

- **Build & Migrate:** If tests pass, the pipeline can run database migrations (for example, `alembic upgrade head` to ensure the latest schema applies, possibly on a staging database) [53]. This verifies that migrations are up-to-date. Then, build the production Docker images for frontend and backend. For instance, use a Docker Build action to build `Dockerfile.frontend` and `Dockerfile.backend` and tag them (e.g., `novadocs-frontend:commitSHA` and `novadocs-backend:commitSHA`).

- **Push & Deploy:** If on the `main` branch (or a tagged release), push the Docker images to a registry (e.g., GitHub Container Registry or Docker Hub). The workflow might log in to the registry and use `docker/build-push-action` to push both images with `:latest` and `:git-sha` tags. After pushing, an optional deploy step could kick in: for example, if using Kubernetes, trigger a Helm upgrade or a `kubectl apply` to update the running cluster. Alternatively, if using a platform like Docker Compose in production, the new images can be pulled by the server and restarted. The CI could also publish the frontend as a static build (Next.js can export static pages if mostly static, but given SSR and dynamic collab, likely it's a Node server mode).

Pseudo-snippet of a GitHub Actions workflow (simplified for brevity):

```
name: CI
on: [push, pull_request]
jobs:
  build-test:
    runs-on: ubuntu-latest
    services:
      postgres:
        image: postgres:16-alpine
        env: { POSTGRES_USER: novadocs, POSTGRES_PASSWORD: novadocs,
POSTGRES_DB: novadocs }
        ports: ["5432:5432"]
        options: --health-cmd "pg_isready" --health-interval 5s
    steps:
      - uses: actions/checkout@v3
      - name: Set up Node & PNPM
        uses: actions/setup-node@v3
        with: { node-version: 18 }
      - run: pnpm install && pnpm run lint && pnpm run test --filter apps/
frontend
      - name: Set up Python
        uses: actions/setup-python@v4
        with: { python-version: "3.12" }
      - run: pip install poetry && poetry install
      - run: poetry run flake8 && poetry run black --check . && poetry run
pytest
      - name: Run migrations
```

```
        run: poetry run alembic upgrade head
  docker-build:
    runs-on: ubuntu-latest
    needs: build-test
    steps:
      - uses: actions/checkout@v3
      - name: Build and push Docker images
        uses: docker/build-push-action@v3
        with:
          push: true
          tags: ghcr.io/your-org/novadocs-frontend:latest, ghcr.io/your-org/
novadocs-backend:latest
          # ... (build args, Dockerfile paths for frontend/backend)
```

In practice, the CI would be split into multiple jobs (build/test, then publish) and could use caching for node_modules and poetry venv to speed up builds. It would also collect test coverage, and possibly run security scans (e.g., Dependabot or Snyk to check for vulnerabilities in dependencies).

For CD (continuous deployment), if NovaDocs is an internal tool, deployment might be manual or triggered on certain tags. If it's open source, instructions would be provided for users to deploy on their own infrastructure (for instance, a **Helm chart** could be included in `infrastructure/helm` for Kubernetes deployment, making it easy to set up Postgres, Redis, etc., and run the app with configurable values).

Security & Scalability Considerations

Threat Model & Security Hardening: NovaDocs is designed with **security in mind**, especially since it can handle sensitive team knowledge. Key aspects: - **XSS Protection:** All user-generated content (pages, comments, etc.) is treated as untrusted and is sanitized or encoded on rendering. The frontend uses React, which by default escapes HTML in user content (and any intentional HTML embeddings would be by safe design). Additionally, the backend sets security headers on responses: e.g., `X-Content-Type-Options: nosniff`, `X-Frame-Options: DENY`, `X-XSS-Protection: 1; mode=block`, and a strict Content Security Policy defaulting to self [54]. These headers mitigate common injection vectors and clickjacking. - **CSRF:** Since the API is mostly accessed via token in Authorization header (for GraphQL and REST) and not via cookies, CSRF risk is lower. If any session cookie is used (for example, if the frontend uses a cookie from Keycloak for API calls), the backend would enforce CSRF tokens or SameSite cookies. In general, using OAuth/OIDC tokens means requests are authenticated via Bearer token, so an attacker cannot replay a user's token without possession. Still, it's important the app checks the `Origin/Referer` on sensitive requests if cookies are used, and includes CSRF tokens for form submissions if any. - **Granular Permissions:** The permissions system ensures users only access what they should. At the API layer, every resolver and REST endpoint that returns workspace- or page-scoped data verifies the current user's access. For example, the `PageService.get_by_id` likely filters by page ID and also checks that the user is a member of the workspace that page belongs to (or that a share link exists) – if not, it returns not found or permission error [55]. This prevents **data leakage** across tenants. We also ensure that one workspace's pages have unique slugs only within that workspace [18], so guessing URLs across workspaces is not possible. - **Encryption & Secrets:** All communication is over HTTPS (TLS). At rest, if needed, the database can use encryption and the object storage (MinIO) can encrypt files. Secrets like JWT signing keys and database passwords are loaded from environment variables (never hard-coded) [56] [57]. In production, these should be stored in a secrets manager or environment config with appropriate access controls. - **Input Validation:** The backend uses

Pydantic for schema validation of inputs (e.g., Strawberry GraphQL inputs are defined as Pydantic models or dataclasses, ensuring types are correct). This prevents malformed data from causing errors or security issues. File upload endpoints restrict file types and size – e.g., `MAX_UPLOAD_SIZE` is set (100 MB by default) and only certain extensions are allowed [58], to avoid attacks via huge files or disallowed formats.

Rate Limiting & Brute-Force: A basic rate limiting middleware is included to thwart brute-force attacks or abusive clients [59]. By default it allows 60 requests per minute per IP in memory [60] (this could be expanded to use Redis for a distributed rate limit store in a multi-node scenario). This primarily protects login attempts or any unauthenticated endpoints. Authenticated heavy operations (like full-text search with large results) could further be rate-limited per user to prevent abuse. Additionally, the system could integrate with an IDS/IPS or at least log suspicious patterns (the structured logging can feed into monitoring for unusual activity).

Horizontal Scaling: NovaDocs is built to scale **horizontally** for higher load: - **Backend API Scaling:** The FastAPI app is stateless (except WebSocket connections). Multiple instances can run behind a load balancer. Sticky sessions are not strictly required if using a distributed collaboration setup (see below), but might simplify WebSocket handling (ensuring a user's WebSocket traffic goes to the same instance). The database (Postgres) will naturally be the main stateful component; it can be scaled read-wise via replicas if needed, but writes (especially from collaborative editing) all go to the primary. We should monitor and optimize queries (use EXPLAIN, ensure indexes like we have on search fields). - **Real-time Collaboration Scaling:** For multiple app servers handling WebSockets, there must be a mechanism to sync CRDT updates and awareness. Options include using **Redis Pub/Sub** or **NATS** as a central event bus. For example, when a user on instance A makes an edit, instance A's server could publish the Yjs update to a Redis channel for that document; all other instances subscribe and relay that update to their local WebSocket clients. This prevents needing sticky sessions – any client can connect to any server and still receive all updates. The code is designed to accommodate this: the GraphQL subscription placeholders explicitly note using Redis or WebSockets for implementation [25]. Another approach is to deploy a dedicated stateful collaboration service (like Hocuspocus) that all clients connect to (thus offloading CRDT sync from the FastAPI app entirely). In either case, NovaDocs can support dozens or hundreds of concurrent editors on the same page, given Yjs's efficiency (the limiting factor would be network bandwidth for broadcasting updates). - **Caching & Performance:** Response caching could be added for read-heavy endpoints (e.g., caching the rendered page or expensive queries in Redis). The current design fetches fresh data on each request (which is simpler and ensures real-time accuracy). Given the use of Postgres and proper indexing (and possibly vector indexes for semantic search), this should be fine for medium-sized teams. For very large deployments, one might introduce an ElasticSearch for even faster full-text search than pg_trgm, but pg_trgm is usually sufficient up to millions of records.

Tenant Isolation: Since multiple workspaces (tenants) share the database, it's critical to enforce workspace scoping on all queries. By design, almost every table has a `workspace_id` and queries filter by it (the services layer methods always require a workspace or user context). Moreover, Role-Based Access Control is enforced such that even if a user guesses an ID of a page from another workspace, the service will not return it because that user lacks a Permission for it. Each workspace's data can also be exported or backed up separately if needed (e.g., one could dump all pages where workspace_id = X). If stronger isolation is needed, one could even deploy separate instances per workspace (but that loses cross-workspace user convenience).

Dependency Security: The stack uses recent versions of frameworks (FastAPI, Next.js, etc.) and will be monitored for vulnerabilities. The CI can run security scanners, and we advise regular updates (the use of Poetry and lockfiles ensures reproducible builds). Keycloak handles the heavy lifting of auth security (password storage, MFA, etc.), so NovaDocs doesn't implement those itself (reducing potential security pitfalls).

Adoption Roadmap

NovaDocs is a ambitious project, and a phased approach helps deliver value early:

- **MVP (≈90 days):** Focus on core features that provide a working collaborative wiki:
- **Pages & Rich Text Editing:** Implement the page hierarchy, the rich text editor with basic block types (headings, paragraphs, lists, checkboxes, code blocks, etc.), and real-time collaboration on page content (multiple cursors, live updates via Yjs). Ensure editing works with minimal latency for 2-3 concurrent users.
- **Workspaces & Auth:** Set up user accounts (integrate Keycloak or simple JWT auth), workspace creation, and membership invites or management (could start with everyone as an editor in the workspace, roles can be simplified initially). Basic permission model: only workspace members can access pages in it.
- **Backlinks & Navigation:** Support linking pages within content (e.g., @ or [[to search pages) and showing backlinks. Also implement a left sidebar or tree view for the page hierarchy for easy navigation.
- **Databases (Basic):** Allow creating a simple table within a page with a few column types (text, number, date, maybe single-select). Support adding/editing rows in a basic table view. (Advanced views like Kanban or Calendar could be postponed).
- **Search:** Implement basic full-text search across pages in a workspace using PostgreSQL text search. The MVP can skip vector search or use a simple trigram search for now.
- **File uploads:** Enable attaching images or files to pages (front-end UI to upload and backend storing them on MinIO and rendering links). Possibly integrate images into the editor content.
- **Responsive UI:** Ensure the web UI is usable on various screen sizes (desktop and tablet at least). A mobile-friendly view can be limited (maybe read-only or basic editing) for MVP.
- **Performance & Bugfix:** Before MVP release, test with sample data and ensure acceptable performance (typing latency, load times under a second for normal pages, etc.). Fix critical bugs (especially around collaboration conflicts, auth flows, and data consistency).

MVP Cuts / Deferrals: Advanced database features (formula fields, rollups), Kanban/Calendar views, sharing pages publicly (share links) might be skipped in MVP. Also, plugin system and webhooks would be deferred. The MVP is essentially a self-hosted Notion-like core: pages, editing, and basic tables, with real-time multi-user editing working reliably.

- **Phase 2 (Next 3-6 months):** Build on the solid foundation:
- **Mobile App:** Develop a React Native or Flutter app for mobile, or at least optimize the web app for mobile browsers. This may involve creating an offline-capable mobile client that syncs via the same APIs. Because the backend is already offline-first friendly, a mobile app can store a local cache (perhaps using SQLite or directly Yjs persistent storage).
- **Advanced Databases:** Add more field types (relations between tables, formula fields that can do computations or lookups, roll-up of linked data). Implement Kanban board view (group by select field), Calendar view (by date field), and Timeline view for databases. These features will make NovaDocs more project-management friendly.
- **AI Integration:** Using the `OPENAI_API_KEY` integration points, add AI-powered features like page summarization, querying the knowledge base in natural language, or auto-generating content. For example, a "Summarize page" action could call OpenAI API to generate a summary and store it in the page's metadata or a summary block. Or an "Ask NovaDocs" search using

embeddings: the backend could generate embeddings for each page (already stored in `content_vector`) and use pgvector to find relevant pages answering a user's question.
- **Plugin/Extension API:** Define a formal way for third-party plugins to extend NovaDocs. This could be client-side (e.g., allow injecting custom Tiptap extensions or custom React components via a plugin registry) and server-side (webhook triggers or custom GraphQL resolvers). Possibly provide a CLI or config to register new block types or integration (for instance, integration with external issue trackers or calendars).
- **Scalability & Clustering:** If not already done, introduce NATS or Redis-based signaling for WebSocket scale-out, and document how to run NovaDocs in a cluster (multiple app pods, with a Redis and Postgres). Also implement sticky sessions or state-sharing needed for collab at scale.
- **Polish & UX:** Address UX improvements like drag-and-drop page tree reordering, a better slash-command palette with all block types, in-place page linking autocompletion, undo/redo across sessions, etc. Improve the theming system to allow easy custom CSS/theme override (the groundwork with Tailwind and CSS variables is already in place, so just expose theme config).
- **Monitoring & Analytics:** Add admin dashboards for system usage (how many pages, active users, etc.). Possibly include an optional analytics module to track user engagement (if enabled) and performance metrics. Also set up Sentry or a similar error tracking service for the frontend and backend to catch and aggregate runtime errors.

By the end of Phase 2, NovaDocs would be a robust platform with feature parity close to mainstream collaborative wiki tools, plus the advantage of self-hosting. The focus would then shift to community contributions, plugin ecosystem, and maybe enterprise features (like LDAP support via Keycloak, backup/restore utilities, and so on).

Throughout these phases, **community feedback** and real-world testing will guide prioritization. Since NovaDocs is open-source, early adopters might contribute improvements or request features — the roadmap can adapt to ensure the most impactful enhancements are delivered. With a solid architecture and clean modular codebase (monorepo), adding new capabilities (be it an AI feature or a new block type) should be relatively straightforward without breaking existing functionality. NovaDocs aims to evolve into the "open-source Notion" for teams that value privacy and control, and this roadmap ensures continuous progress towards that vision.

---

1 2 36 37 38 41 44 001_initial_schema.py
https://github.com/stevross-git/NovaDocs/blob/2e869f366e93c5f068ea61ff4803450087075527/novadocs/apps/backend/src/migrations/001_initial_schema.py

3 4 docker-compose.yml
https://github.com/stevross-git/NovaDocs/blob/2e869f366e93c5f068ea61ff4803450087075527/novadocs/infrastructure/docker/development/docker-compose.yml

5 6 7 50 51 52 53 Makefile
https://github.com/stevross-git/NovaDocs/blob/2e869f366e93c5f068ea61ff4803450087075527/novadocs/Makefile

8 9 10 11 12 13 14 15 16 17 18 19 22 39 40 42 43 models.py
https://github.com/stevross-git/NovaDocs/blob/2e869f366e93c5f068ea61ff4803450087075527/novadocs/apps/backend/src/core/models.py

20 21 23 24 25 26 55 schema.py
https://github.com/stevross-git/NovaDocs/blob/2e869f366e93c5f068ea61ff4803450087075527/novadocs/apps/backend/src/api/graphql/schema.py

27 28 29 routes.py
https://github.com/stevross-git/NovaDocs/blob/2e869f366e93c5f068ea61ff4803450087075527/novadocs/apps/backend/src/api/rest/routes.py

30 31 collaboration.py
https://github.com/stevross-git/NovaDocs/blob/2e869f366e93c5f068ea61ff4803450087075527/novadocs/apps/backend/src/api/websocket/collaboration.py

32 33 35 CollaborativeEditor.tsx
https://github.com/stevross-git/NovaDocs/blob/2e869f366e93c5f068ea61ff4803450087075527/novadocs/apps/frontend/src/components/editor/CollaborativeEditor.tsx

34 main.py
https://github.com/stevross-git/NovaDocs/blob/2e869f366e93c5f068ea61ff4803450087075527/novadocs/apps/backend/src/main.py

45 46 47 48 49 usePage.ts
https://github.com/stevross-git/NovaDocs/blob/2e869f366e93c5f068ea61ff4803450087075527/novadocs/apps/frontend/src/hooks/usePage.ts

54 59 60 middleware.py
https://github.com/stevross-git/NovaDocs/blob/2e869f366e93c5f068ea61ff4803450087075527/novadocs/apps/backend/src/core/middleware.py

56 57 58 config.py
https://github.com/stevross-git/NovaDocs/blob/2e869f366e93c5f068ea61ff4803450087075527/novadocs/apps/backend/src/core/config.py