NovusAI - Technical Documentation

Version: 1.0
Last Updated: Augest 2025
Author: Technical Documentation Team

Table of Contents

Architecture Overview

NovusAI is a full-stack web application built with a React.js frontend and dual backend servers (Flask for Python-based AI operations and Node.js for database operations). The application provides AI-powered document processing, email integration, and intelligent chat capabilities.

Technology Stack

Frontend: - React.js 19.1.1 with functional components and hooks - React Router for navigation - Sass for styling - KaTeX for mathematical notation rendering - ReactMarkdown for rich text formatting - Lucide React for icons

Backend: - Flask (Python) for AI operations and API routes - Node.js with Express for database operations - PostgreSQL for data persistence - OpenAI GPT-4 integration for AI responses - Gmail API integration with OAuth2

Authentication & Security: - JWT (JSON Web Tokens) for session management - BCrypt for password hashing - OAuth2 for Gmail integration - CORS enabled for cross-origin requests

---

Backend Runtime Architecture

The Flask service inside `novusai/src/backend/app` is structured around blueprints so each functional surface (auth, chat, billing, RAG, etc.) remains independently deployable.

Execution flow: - `app/__init__.py` wires the global `Flask` instance, registers blueprints under `/api`, and initialises shared extensions (Redis cache, database connection pool, colorised logging). - Every request passes through `utils/auth_helperes.token_required` when the route demands authentication; the decorator decodes JWTs, loads cached user metadata, and attaches a `request.user` dict for downstream handlers. - Database access is handled through the shared `psycopg2` connection (`db.conn`) for fast query execution; high-concurrency tasks open short-lived connections via `db.new_conn()`.

Cross-cutting concerns: - **Caching:** `utils.Cache.cache` fronts Redis and powers usage counters, org subscription flags, and memoised lookups (e.g., document metadata in RAG flows). - **Usage gating:** Billing logic (`billing.Subtract_Usage`, `billing.getUsage`, `billing.is_org_subscription_active`) is called before expensive operations like RAG queries, streaming responses, and PDF generation to enforce organisation quotas. - **Observability:** System events funnel through `utils.system_logger` while user actions are captured with `utils.user_helpers.log`, enabling per-tenant audits. - **Streaming:** Long-running responses (e.g., `/api/broad/citations/stream`) leverage Flask's `stream_with_context`, yielding SSE events so the React client can render partial outputs without blocking.

Frontend Components (React.js)

App.js - Main Application Controller

Location: `novusai/src/App.js`

Purpose: Serves as the root component managing application state, routing, and user authentication flow.

Key Functions:

State Management

```
const [user, setUser] = useState(null);           // Current authenticated user
const [activeTab, setActiveTab] = useState('Home'); // Current active view
const [messages, setMessages] = useState([]);      // Chat message history
const [apiKey, setApiKey] = useState('');          // OpenAI API key
const [model, setModel] = useState('gpt-3.5-turbo'); // AI model selection
const [temperature, setTemperature] = useState(0.7); // AI response creativity
```

Core Functions `addMessage(msg)` - **Purpose:** Adds new messages to the chat history - **Parameters:** `msg` (Object) - Message object containing sender and text - **Returns:** None (updates state) - **Usage:** Called when user sends message or AI responds

`clearMessages()` - **Purpose:** Resets chat history for new conversations - **Parameters:** None - **Returns:** None (clears messages state) - **Usage:** Triggered by chat clear button or new session

Component Routing Logic: The App component conditionally renders different views based on `activeTab` state: - `Home`: Landing page with application introduction - `Upload`: File upload interface for document processing - `Viewer`: Document viewing and analysis interface - `Tasks`: Task management and processing status - `Search`: Document search and retrieval - `Chat`: AI-powered conversation interface - `Settings`: User preferences and API configuration - `Account`: Authentication forms and user management - `Dashboard`: Analytics and Gmail integration

---

Chat.jsx - AI Conversation Interface

Location: `novusai/src/Frontend/components/Chat.jsx`

Purpose: Provides an interactive chat interface with AI, supporting rich text, mathematical notation, and conversation history.

Key Functions `generateChatHash()` - **Purpose:** Creates unique identifiers for chat sessions - **Returns:** String - Unique hash combining random characters and timestamp - **Algorithm:** `Math.random().toString(36) + Date.now().toString(36)` - **Usage:** Called when starting new chat sessions

`renderMessageContent(text)` - **Purpose:** Processes and renders message text with LaTeX math support - **Parameters:** `text` (String) - Raw message text with potential LaTeX notation - **Returns:** JSX elements with formatted content - **Features:** - Detects LaTeX expressions in `\[...\]` (block math) and `\(...\)` (inline math) - Renders remaining text as Markdown - Supports complex mathematical expressions using KaTeX

Chat Message Management:

```
// Message structure
{
  sender: 'user' | 'bot',
  text: string,
  timestamp: Date
}
```

`handleSendMessage()` - **Purpose:** Processes user input and triggers AI response - **Flow:** 1. Validates user input 2. Adds user message to conversation 3.

Calls backend API for AI response 4. Updates conversation history 5. Handles error states

Features: - Real-time typing indicators - Message timestamp tracking - LaTeX mathematical notation support - Markdown formatting for rich text - Chat session persistence - Demo mode for API key-less operation

---

Dashboard.jsx - User Analytics and Gmail Integration

Location: `novusai/src/Frontend/components/Dashboard.jsx`

Purpose: Displays user statistics, manages Gmail integration, and provides account overview.

Key Functions `fetchDashboardStats()` - **Purpose:** Retrieves user analytics from backend API - **API Endpoint:** `GET /api/dashboard` - **Returns:** Object containing user statistics - **Data Structure:**

```
{
  emails_processed: number,     // Total emails processed
  documents_uploaded: number,   // Total documents uploaded
  ai_task_completed: number,    // Total AI tasks completed
  monthly_usage: number         // Monthly usage percentage
}
```

`fetchGmailLink()` - **Purpose:** Retrieves Gmail OAuth authorization URL - **API Endpoint:** `GET /api/gmail/link` - **Returns:** OAuth authorization URL for Gmail integration - **Flow:** 1. Requests authorization URL from backend 2. Presents link to user for OAuth consent 3. Handles OAuth callback and token storage

`fetchLinkedAccounts()` - **Purpose:** Retrieves list of linked Gmail accounts - **API Endpoint:** `GET /api/emails` - **Returns:** Array of linked email addresses - **Usage:** Displays connected Gmail accounts in dashboard

Dashboard Metrics **Email Processing Statistics:** - Tracks total emails processed through the system - Updates automatically when emails are analyzed - Displays processing success/failure rates

Document Upload Tracking: - Monitors document upload activity - Tracks file types and sizes processed - Shows recent upload history

AI Task Completion: - Records all AI-powered operations - Includes chat responses, document analysis, email processing - Provides performance metrics and usage patterns

Monthly Usage Monitoring: - Calculates usage based on API costs and processing time - Displays percentage of monthly limits consumed - Provides

usage forecasting and alerts

Upload.jsx - File Processing Interface

Location: `novusai/src/Frontend/components/Upload.jsx`

Purpose: Handles file uploads and document processing initialization.

Key Functions Current Implementation:

```
const Upload = () => (
  <div className="card">
    <h2> Uploaded Documents</h2>
    <p>Drag & drop files (PDF, DOCX, PPTX, XLSX, TXT).</p>
    <input type="file" multiple />
  </div>
);
```

Supported File Types: - **PDF:** Portable Document Format files - **DOCX:** Microsoft Word documents - **PPTX:** Microsoft PowerPoint presentations - **XLSX:** Microsoft Excel spreadsheets - **TXT:** Plain text files

Future Enhancement Areas: - File validation and size checking - Upload progress indicators - Drag-and-drop functionality - File preview capabilities - Processing status tracking - OCR integration for scanned documents

---

Settings.jsx - Configuration Management

Location: `novusai/src/Frontend/components/Settings.jsx`

Purpose: Manages user preferences and application configuration.

Note: Currently implemented as empty component, designed for future configuration options.

Planned Features: - OpenAI API key management - Model selection (GPT-3.5, GPT-4, custom models) - Temperature and creativity settings - User interface preferences - Data retention policies - Privacy settings - Export/import configurations

---

Home.jsx - Landing Page

Location: `novusai/src/Frontend/components/Home.jsx`

Purpose: Provides application introduction and navigation entry point.

Key Functions `useEffect` for Visual Effects: - Manages overflow settings for animated background - Prevents horizontal scrolling during animations - Cleans up styles on component unmount

Visual Features: - Animated star field background (100 randomly positioned particles) - Dynamic particle scaling and opacity - Responsive circular design element - Smooth CSS transitions and animations

Content Structure: - Application branding and name display - Feature description and value proposition - Call-to-action button for chat navigation - Inspirational quote display

---

login.jsx - Authentication Interface

Location: `novusai/src/Frontend/components/login.jsx`

Purpose: Handles user authentication including login and registration.

Key Functions:

User Registration: - Form validation for username and password - Password strength requirements - Duplicate username checking - Account creation with secure password hashing

User Login: - Credential validation against database - JWT token generation and storage - Session management and persistence - Login state management

Authentication Flow: 1. User submits credentials 2. Frontend validates input format 3. API call to backend authentication endpoint 4. Backend verifies credentials and generates JWT 5. Frontend stores token and updates application state 6. User is redirected to dashboard or requested page

---

FileStorage.jsx - Document Library & Viewer

Location: `novusai/src/Frontend/components/FileStorage.jsx`

Purpose: Lists every asset the signed-in user has uploaded, exposes inline previews, and coordinates delete/download flows against the `filestorage` API.

Highlights: - Hydrates table state from `GET /api/files/getUserFiles`, normalising payloads and surfacing aggregate counts. - Provides in-browser previews by toggling `view=1` on download URLs and embedding returned blobs in an `<iframe>` modal. - Wraps destructive actions in confirmation modals (`ModernConfirm`) and keeps optimistic UI state in sync with backend responses.

Fetch & action dispatch:

```
const res = await fetch(`${API}/files/getUserFiles`, {
  headers: token ? { Authorization: `Bearer ${token}` } : {},
});
```

- Reuses the viewer state machine (`viewerOpen`, `viewerMime`, `viewerUrl`) so the same payload supports inline preview or traditional download.

Rag.jsx - Retrieval Playground

Location: `novusai/src/Frontend/components/Rag.jsx`

Purpose: Lets power users query their personal RAG index with custom retrieval modes before surfacing the answer stream inside Chat.

Highlights: - Switches between summary and full-text retrieval while exposing `top_k` and `auto_case` controls. - Renders grounding metadata (`matches`) alongside the model answer so users can inspect source coverage.

Backend handshake:

```
const res = await fetch(`${API}/rag/answer`, {
  method: 'POST',
  headers: { 'Content-Type': 'application/json', Authorization: `Bearer ${token}` },
  body: JSON.stringify({ query, retrieval_mode: retrievalMode, top_k: Number(topK), auto_cas
});
```

- Errors are surfaced inline and the UI disables submission while the request is in flight to avoid duplicate invocations.

Payments.jsx - Stripe Billing Console

Location: `novusai/src/Frontend/components/Payments.jsx`

Purpose: Embeds Stripe Elements to manage subscriptions, one-off top-ups, and stored payment methods while mirroring backend usage counters.

Highlights: - Auto-detects organization status from `GET /api/billing/org/subscription` and branches into subscription or deposit flows. - Lazily loads Stripe.js, caches the `Stripe` instance, and mounts/unmounts Elements as intents change. - Provides a guided top-up experience that maps dollars to internal usage credits (`USAGE_PER_DOLLAR`).

Creating a top-up intent:

```
const res = await fetch('/api/billing/create-payment-intent', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json', ...(token ? { Authorization: `Bearer ${toke
```

```
  body: JSON.stringify({ amount: cents, currency: 'usd', metadata: { purpose: 'topup' } }),
});
```

- After confirmation, the component polls `GET /api/billing/payment-intent/<id>` to reflect status and unlock the dashboard when funds settle.

SshConsole.jsx - Data & Backup Console

Location: `novusai/src/Frontend/components/SshConsole.jsx`

Purpose: Presents a lightweight control panel for Qdrant, Postgres, and user-file backups exposed by the data manager API.

Highlights: - Tabbed navigation reuses the same fetch pipeline while swapping URL roots (`/api/data/qdrant`, `/api/data/sql`, `/api/data/userfiles`). - Restore operations include destructive warnings and stream status updates back into the UI.

Retrieving backup sets:

```
const res = await fetch(`${API}/${activeTab}/backups`, { credentials: 'include' });
```

- The console surfaces success/error banners and exposes one-click retries to simplify recovery workflows.

UserManagment.jsx - Org Roster & Audit Tools

Location: `novusai/src/Frontend/components/UserManagment.jsx`

Purpose: Gives administrators a consolidated view of organization members, role assignments, and scoped log downloads.

Highlights: - Pairs `GET /api/users/org` with cached JWT identity lookups to present editable role pickers per user. - Surfaces admin-only log listings via `GET /api/logs/list` and streams individual files into a modal viewer. - Wraps irreversible actions (role changes, deletions) in confirmation prompts that require an emailed code where appropriate.

Org roster retrieval:

```
const res = await fetch(`${API}/users/org`, { headers: authHeader() });
const data = await res.json();
setUsers(Array.isArray(data.users) ? data.users : []);
```

- Local `edits` state tracks unsaved role adjustments so the UI can batch updates without leaving users in inconsistent states.

Backend API Routes (Flask)

auth.py - Authentication Management

Location: `novusai/src/backend/app/routes/auth.py`

Purpose: Handles user authentication, registration, and JWT token management.

Key Functions `register_user()` - **Route:** `POST /api/register` - **Purpose:** Creates new user accounts with secure password storage - **Parameters:** - `username` (String): Unique username for the account - `password` (String): Raw password to be hashed - **Process:** 1. Validates input parameters for completeness 2. Hashes password using Werkzeug security functions 3. Inserts new user record into PostgreSQL database 4. Handles duplicate username errors - **Returns:** Success message or error details - **Security:** Uses `generate_password_hash()` with salt for secure storage

`login_user()` - **Route:** `POST /api/login` - **Purpose:** Authenticates users and provides JWT access tokens - **Parameters:** - `username` (String): User's login identifier - `password` (String): Raw password for verification - **Process:** 1. Retrieves user record from database by username 2. Verifies password using `check_password_hash()` 3. Generates JWT token with user information and expiration 4. Fetches Gmail credentials and recent emails for dashboard 5. Returns authentication token and user details - **JWT Payload:**

```
{
  'id': user_id,
  'username': username,
  'exp': datetime.utcnow() + timedelta(hours=2)
}
```

- **Returns:** JWT token, user data, and success confirmation

`get_user_info()` - **Route:** `GET /api/user` - **Purpose:** Retrieves current user information from JWT token - **Authentication:** Requires valid JWT token via `@token_required` decorator - **Returns:** User ID and username from token payload - **Usage:** Profile display, user validation, session verification

Security Features **JWT Token Management:** - 2-hour expiration policy for security - HS256 algorithm for token signing - Secret key protection via environment variables - Automatic token validation on protected routes

Password Security: - Werkzeug password hashing with automatic salt generation - Protection against rainbow table attacks - Secure comparison functions to prevent timing attacks

---

chat.py - Chat Session Management

Location: `novusai/src/backend/app/routes/chat.py`

Purpose: Manages AI chat sessions, message persistence, and conversation history.

Key Functions `start_chat()` - **Route:** `POST /api/chat/start` - **Purpose:** Initializes new chat sessions with unique identifiers - **Authentication:** Requires JWT token - **Parameters:** - `chathash` (String): Unique session identifier - `chatname` (String): Human-readable chat title - **Process:** 1. Extracts user information from JWT token 2. Creates initial chat data structure 3. Inserts new session record into database 4. Handles database transaction rollback on errors - **Database Schema:**

```
INSERT INTO user_chat_sessions (chathash, username, chat_data)
VALUES (hash, username, {
  "chatname": string,
  "user": [],
  "bot": []
})
```

`chat()` - **Route:** `POST /api/chat` - **Purpose:** Processes user messages and generates AI responses - **Authentication:** Requires JWT token - **Parameters:** - `chathash` (String): Session identifier - `chatname` (String): Chat title - `usermsg` (String): User's message content - **Process:** 1. Validates user authentication and message content 2. Calls OpenAI API through `chatgpt_response()` utility 3. Retrieves existing chat session or creates new one 4. Appends user message and bot response to conversation 5. Updates database with new message history 6. Handles error states and database rollbacks - **Returns:** AI response text and conversation update confirmation

`rename_chat(chathash)` - **Route:** `PATCH /api/chat/<chathash>/rename` - **Purpose:** Updates chat session titles for organization - **Authentication:** Requires JWT token - **Parameters:** - `chathash` (String): Session identifier from URL - `new_name` (String): Updated chat title - **Process:** 1. Validates chat ownership and existence 2. Updates chat_data JSON with new title 3. Commits changes to PostgreSQL database - **Returns:** Rename confirmation or error message

`delete_chat(chathash)` - **Route:** `DELETE /api/chat/<chathash>` - **Purpose:** Removes chat sessions and associated history - **Authentication:** Requires JWT token - **Security:** Ensures users can only delete their own chats - **Process:** 1. Validates chat ownership via username check 2. Removes chat record from database 3. Handles foreign key constraints and dependencies - **Returns:** Deletion confirmation

`get_user_chats()` - **Route:** `GET /api/chat` - **Purpose:** Retrieves all chat sessions for authenticated user - **Authentication:** Requires JWT token - **Pro-**

cess: 1. Queries database for user's chat sessions 2. Transforms JSON chat data into structured format 3. Interleaves user and bot messages chronologically 4. Handles unbalanced conversations (more user or bot messages) - **Returns:** Array of chat objects with messages and metadata - **Message Format:**

```
{
  'id': session_id,
  'title': chat_name,
  'messages': [
    {'sender': 'user', 'text': message_content},
    {'sender': 'bot', 'text': ai_response}
  ],
  'hash': unique_identifier
}
```

Chat Data Structure Database Storage: Chat conversations are stored as JSON objects in PostgreSQL, allowing flexible message structures while maintaining relational benefits.

JSON Schema:

```
{
  "chatname": "Session Title",
  "user": ["User message 1", "User message 2"],
  "bot": ["Bot response 1", "Bot response 2"]
}
```

This parallel array structure maintains message order and conversation flow while enabling efficient database queries and updates.

---

gmail.py - Gmail Integration

Location: `novusai/src/backend/app/routes/gmail.py`

Purpose: Manages Gmail OAuth integration and email processing capabilities.

Key Functions `get_user_credentials(user_id)` - **Purpose:** Retrieves stored Gmail OAuth credentials for authenticated users - **Parameters:** `user_id` (Integer): Database user identifier - **Process:** 1. Queries `gmail_tokens` table for user credentials 2. Deserializes JSON credential data 3. Creates Google OAuth2 Credentials object 4. Handles missing or expired credentials - **Returns:** Google Credentials object ready for API calls - **Error Handling:** Raises exception if no credentials found

`list_user_emails(service, max_results=10)` - **Purpose:** Fetches recent emails from user's Gmail account - **Parameters:** - `service` (Google API Service): Authenticated Gmail API client - `max_results` (Integer): Maximum

number of emails to retrieve - **Process:** 1. Calls Gmail API messages.list() endpoint 2. Retrieves message details for each email 3. Extracts subject lines and snippets from headers 4. Handles API rate limits and errors - **Returns:** List of tuples containing (subject, snippet) - **Usage:** Dashboard email preview and processing initialization

Gmail API Integration OAuth2 Flow: 1. User initiates Gmail linking from dashboard 2. Application redirects to Google OAuth consent screen 3. User grants permissions for email access 4. Google returns authorization code to callback URL 5. Application exchanges code for access and refresh tokens 6. Tokens stored securely in database for future API calls

Supported Gmail Operations: - **Email Listing:** Retrieve recent messages with metadata - **Message Reading:** Full email content access including attachments - **Header Parsing:** Extract sender, recipient, subject, and timestamp information - **Snippet Generation:** Automatic email preview text creation

Security Considerations: - Minimal scope requests (read-only email access) - Secure token storage with encryption at rest - Token refresh handling for expired credentials - User consent verification and revocation support

---

dashboard.py - User Analytics and Account Management

Location: `novusai/src/backend/app/routes/dashboard.py`

Purpose: Provides user statistics, analytics, and account management endpoints.

Key Functions `get_dashboard()` - **Route:** `GET /api/dashboard` - **Purpose:** Retrieves comprehensive user analytics and usage statistics - **Authentication:** Requires JWT token via `@token_required` decorator - **Process:** 1. Extracts user ID from JWT token payload 2. Queries dashboard table for user metrics 3. Handles missing dashboard records (creates default) 4. Returns formatted analytics data - **Returns:** JSON object with user statistics - **Data Structure:**

```
{
  "emails_processed": 0,      // Total emails analyzed
  "documents_uploaded": 0,    // Total files processed
  "ai_task_completed": 0,     // Total AI operations
  "monthly_usage": 0.00       // Usage percentage of monthly limit
}
```

`get_linked_gmail_accounts()` - **Route:** `GET /api/emails` - **Purpose:** Lists all Gmail accounts linked to user's profile - **Authentication:** Requires JWT token - **Process:** 1. Queries `gmail_tokens` table for user's linked accounts 2. Extracts email addresses from credential records 3. Orders results by most

recently linked 4. Handles database connection errors - **Returns:** Array of linked email addresses - **Usage:** Dashboard account management and email selection

Analytics Tracking Usage Metrics: The dashboard system tracks several key metrics to provide users with insights into their application usage:

Email Processing Metrics: - Incremented when emails are analyzed by AI - Tracks success/failure rates - Monitors processing time and performance

Document Upload Tracking: - Records file uploads and processing attempts - Tracks file types and sizes - Monitors storage usage

AI Task Completion: - Counts all AI-powered operations - Includes chat responses, document analysis, email processing - Tracks API costs and token usage

Monthly Usage Calculation: - Calculates percentage of monthly limits consumed - Based on API costs, processing time, and storage usage - Provides usage forecasting and limit warnings

---

broad_questions.py - RAG Orchestrator & Humanizer

Location: `novusai/src/backend/app/routes/broad_questions.py`

Purpose: Serves "broad question" traffic by chaining the RAG service, optional humanisation, citation streaming, and billing enforcement.

Highlights: - Proxies payloads to `/api/rag/answer` with preserved cookies/headers so downstream decorators continue to see the caller's identity. - Adds specialist endpoints for Markdown repair, citation retrieval (`/citations` & `/citations/stream`), and per-user LlamaIndex conversations. - Humanised modes rewrite raw answers into structured briefs (executive, investment, legal, policy, etc.), persist them to chat history, and debit organisation usage.

Representative flow:

```
rag = _call_rag_answer(
    query_text=query,
    chathash=chathash,
    chatname=chatname,
    top_k=int(top_k) if isinstance(top_k, int) else None,
    auto_case=bool(auto_case) if isinstance(auto_case, bool) else None,
    retrieval_mode=retrieval_mode,
)
```

- After the base answer is produced, `_strip_metadata_tags` scrubs source markers before the humaniser runs, and `compute_usage_from_messages` drives real-time usage deductions.

- `citations()` and `citations/stream` layer on usage gating (`is_org_subscription_active`, `getUsage`) before calling RAG, then hydrate citation bodies either from cached RAG summary blocks or direct database lookups when needed.
- The streaming handler buffers raw chunks, cleans Markdown with `fix_common_md`, and emits structured SSE events so the React client can render partial answers and usage metrics while the model writes.
- `htmlString(...)` transforms investment briefs into HTML pamphlets via the OpenAI mini model and feeds the result into `generate_pamphlet`, returning both the raw JSON and rendered HTML back to the client.

Rag.py - Retrieval-Augmented Answer Service

Location: `novusai/src/backend/app/routes/Rag.py`

Purpose: Implements hybrid BM25/vector retrieval over per-user Qdrant collections and synthesises grounded answers, citations, and graphs.

Highlights: - Caches LlamaIndex handles and Qdrant query engines so successive requests avoid expensive rebuilds. - Offers `/search`, `/answer`, `/citations`, `/raw`, and `/graph` endpoints, allowing downstream services to pull either structured hits or rendered HTML. - Annotates retrieved context with alias markers and validates model citations against the actual source IDs before returning them.

Key internals: - `retrieve_hybrid(...)` merges BM25 hits from Postgres with Qdrant vector scores using Reciprocal Rank Fusion. Each hit is a dict containing `id_tag`, `source_id`, `subject`, `body`, `score`, and the raw payload retrieved from storage. - `_ensure_embed_model_once()` pins the LlamaIndex embedding model to match the configured OpenAI embedding so repeated calls do not reconfigure global state. - `_CACHE_LOCK` protects per-user index caching; the module stores `VectorStoreIndex` objects in `_INDEX_CACHE` keyed by Qdrant collection name. - `build_context_full_sources(...)` truncates concatenated context to `MAX_CTX_CHARS` (or `FULLTEXT_CTX_CHARS` for full-text mode) and injects alias annotations so the downstream LLM can cite sources deterministically. - `check_grounding_types(...)` inspects the generated answer, verifying that every cited `email_id:` or `doc_id:` exists in the retrieved context before marking `grounding_ok`.

Answer pipeline:

```
if USE_HYBRID:
    hits = retrieve_hybrid(query, case_label=case_label, top_k=top_k, user_id=user_id, retri
ctx = build_context_full_sources(hits, user_id=user_id, max_chars=ctx_cap, retrieval_mode=re
ans = call_llm(query, ctx_annot)
```

- If a case filter removes too much context, the service retries without it to keep answer quality high.

- When citations are requested, `_call_rag_citations` reuses the same retrieval stack but returns the raw chunk metadata; the streaming variant serialises SSE messages (`token`, `usage`, `meta`, `done`) to keep UI progress responsive.

---

upload.py - Multi-Format Document Ingestion

Location: `novusai/src/backend/app/routes/upload.py`

Purpose: Normalises uploaded assets (PDF, DOCX, images, emails, etc.), runs OCR and summarisation, and indexes both metadata and embeddings into Postgres/Qdrant.

Highlights: - Configurable chunk size/overlap with optional title injection improves downstream retrieval relevance. - Generates per-document summaries that feed a dedicated "summary" vector track for broad-mode queries. - Falls back to manual Qdrant upserts when LlamaIndex writes omit required metadata, guaranteeing every point keeps `doc_id` and `user_id` payload fields. - Runs best-effort OCR via `pdf2image` + `pytesseract` for image-heavy PDFs, and routes DOCX, EML, and legacy formats through specialised parsers (python-docx, BytesParser). - For emails, combines header metadata (`extract_metadata_from_eml`) with sanitised bodies (`sanitize_bodies`) before indexing to maintain structured search. - Usage counters are incremented at each major milestone (`set_progress`, `Subtract_Usage`), allowing the frontend to render fine-grained ingestion status updates.

Embedding & storage guard:

```
vector_store = QdrantVectorStore(client=qdrant, collection_name=collection)
VectorStoreIndex.from_documents([doc], storage_context=storage_context, embed_model=li_embed
```

- On failure, `openai_embedding(text)` provides a manual vector and the code upserts via `qdrant.upsert`, preserving payload integrity.

---

filestorage.py - Secure Asset Delivery

Location: `novusai/src/backend/app/routes/filestorage.py`

Purpose: Handles authenticated downloads, inline previews, and audit logging for user-managed documents.

Highlights: - Normalises paths and enforces directory boundaries to prevent traversal attempts. - Converts Word documents to PDF on demand with headless LibreOffice when `view=1` is supplied, streaming the result directly to the browser. - Logs every interaction through `user_audit_log` and `log_system_event`, recording mode (view/download) and organisation context.

- Delete operations cascade into `qdrant_helper.delete_document_points_by_ids` so vector payloads remain in sync with the relational document catalogue.

Inline conversion path:

```
if view_mode and ext in ("doc", "docx"):
    proc = subprocess.run(cmd, stdout=subprocess.PIPE, stderr=subprocess.PIPE, timeout=120)
    with open(pdf_path, "rb") as f:
        data = f.read()
    resp = send_file(io.BytesIO(data), mimetype="application/pdf", as_attachment=False, down
```

- Temporary directories are cleaned up immediately after streaming to keep the filesystem tidy.

---

pdf.py - HTML-to-PDF Rendering Service

Location: `novusai/src/backend/app/routes/pdf.py`

Purpose: Uses Playwright-controlled Chromium to convert HTML fragments into downloadable PDFs for investor briefs and reports.

Highlights: - Waits for fonts/media to load before rendering, ensuring the PDF matches the screen layout. - Returns binary responses with explicit `Content-Length` and `Content-Disposition` headers for reliable browser handling.

Renderer:

```
with sync_playwright() as p:
    browser = p.chromium.launch()
    page = browser.new_page()
    page.set_content(html, wait_until="networkidle")
    pdf_bytes = page.pdf(format=format, landscape=landscape, print_background=True, prefer_
```

- Errors are surfaced as JSON responses so the frontend can display actionable feedback.

---

billing.py - Stripe Subscriptions & Usage Accounting

Location: `novusai/src/backend/app/routes/billing.py`

Purpose: Centralises subscription lifecycle management, Stripe billing hooks, and the usage ledger consumed by AI workloads.

Highlights: - Persists subscription status and usage counters in Postgres while mirroring hot values in Redis for fast reads. - Supports both recurring plans (`/create-subscription`) and ad-hoc top-ups via PaymentIntents. - Deducts AI costs from organisational balances through `Subtract_Usage`, powering

gating logic across chat, RAG, and broad flows. - `get_org_subscription` translates Stripe states into internal status codes (`1=active`, `2=inactive`, `3=needs_topup`) so the React dashboard can branch flows deterministically. - `grant_subscription_usage` seeds new organisations with the configured credit block (default 15,000 units) immediately after a successful billing event.

Payment intent creation:

```
pi = stripe.PaymentIntent.create(
    amount=amount,
    currency=currency,
    automatic_payment_methods={"enabled": True},
    **({"receipt_email": receipt_email} if receipt_email else {}),
)
return jsonify({"clientSecret": pi.client_secret, "payment_intent": pi.id})
```

- Successful intents trigger subscription activation and top up organisational usage credits.

---

backups.py - Qdrant, SQL, and User File Backups

Location: `novusai/src/backend/app/routes/backups.py`

Purpose: Manages vector store, Postgres, and user file backups with both modern and legacy URL namespaces.

Highlights: - Streams Qdrant collections into tarballs, saves PCA visualisations to Postgres, and restores datasets atomically. - Automates Postgres dumps and enforces pruning of the oldest user-file backups to control storage. - Re-exports every endpoint under `/api/data/*` for compatibility with older frontends. - `qdrant_visualize` reads vectors in batches of 1,000, reduces them to 2-D with PCA, and stores the Plotly HTML snippet in the `qdrant_graph` table for the requesting user. - Backups are staged to `/tmp` before upload to S3 helpers (`utils.s3_helper`), minimising downtime while large archives are generated.

Compatibility mapping:

```
compat_bp.add_url_rule("/qdrant/backups", view_func=qdrant_list_backups, methods=["GET"])
compat_bp.add_url_rule("/sql/restore",    view_func=sql_restore,         methods=["POST"])
compat_bp.add_url_rule("/userfiles/restore", view_func=userfiles_restore, methods=["POST"])
```

- Shared implementations keep logging and validation consistent across both namespace styles.

---

gmail.py - Google OAuth & Inbox Snapshotting

Location: `novusai/src/backend/app/routes/gmail.py`

Purpose: Handles Gmail OAuth, token storage, and lightweight inbox previews for dashboard consumption.

Highlights: - Persists OAuth state tokens to defend against CSRF and stores credentials per user in Postgres. - Builds Gmail service clients on demand to fetch subject/snippet previews without re-running the heavy ingestion pipeline. - Respects environment-provided redirect URIs so the same code can run locally or in production. - Token blobs are serialised as JSON (`credentials_json`) and rehydrated into `google.oauth2.credentials.Credentials` objects before each API call. - Helper `_pick_col` detects schema differences between deployments, allowing the service to resolve the correct column names (e.g., `gmail_id` vs `google_id`).

Authorization URL issuance:

```
flow = _flow()
auth_url, state = flow.authorization_url(access_type="offline", include_granted_scopes="true
cur.execute("INSERT INTO oauth_states (state, user_id, provider) VALUES (%s, %s, 'google') (
```

- The callback exchanges the code for tokens and stores them via `_new_conn()` to avoid interfering with the global connection pool.

---

logs.py - Organisation Log Browser

Location: `novusai/src/backend/app/routes/logs.py`

Purpose: Gives administrators scoped access to archived application logs.

Highlights: - Enforces admin-only access and normalises requested filenames to prevent traversal. - Supports inline previews (`view=1`) or forced downloads and logs every action for auditability. - Uses `os.scandir` sorted by mtime to surface the freshest logs first and gracefully handles missing directories (returns empty list). - Download routes guard against directory traversal by requiring the resolved path to remain inside the organisation-specific folder before delegating to `send_file`.

Admin list:

```
ok, resp = _require_admin()
if not ok:
    return resp
org_dir = _org_log_dir()
entries = _serialize_entries(org_dir)
return jsonify({"logs": entries})
```

- Uses compatibility fallbacks when calling `send_file` so the route works across Flask versions.

---

users.py - Organisation Membership Management

Location: `novusai/src/backend/app/routes/users.py`

Purpose: Powers the user management console by exposing safe role updates, deletion workflows, and organisation rosters.

Highlights: - Validates both actor and target ranks before applying changes, preventing privilege escalation or self-demotion. - Issues email-based delete codes, caches pending authorisations, and requires verification before removing users and their vector data. - Provides public organisation listings for onboarding flows via `/api/users/organizations`. - `update_user_role` double-checks organisation equality (actor vs target) and forbids self updates, ensuring even admins cannot accidentally lock themselves out. - Deletion flow: `/<user_id>/delete/request` generates a 6-digit code, emails it via `utils.gmail_helpers.send_email`, stores the request in Redis, and `/<user_id>/delete/verify` validates the code before invoking `Admin_Tools.Qdrant_Manager.delete_collections_for_user`.

Role update enforcement:

```
if actor_rank <= target_rank:
    return _deny("Forbidden: target is not lower rank", 403)
if new_role > actor_rank:
    return _deny("Forbidden: cannot assign role above your rank", 403)
with conn.cursor() as cur:
    cur.execute("UPDATE users SET role = %s WHERE id = %s", (int(new_role), str(user_id)))
```

- Successful updates log both user-facing and system events so audit trails remain complete.

Backend Utilities

openai_client.py - AI Integration

Location: `novusai/src/backend/utils/openai_client.py`

Purpose: Handles OpenAI API integration, cost tracking, and AI response generation.

Key Functions `chatgpt_response(user_message, user_id)` - **Purpose:** Generates AI responses using OpenAI's GPT models with cost tracking - **Parameters:** - `user_message` (String): User's input text for AI processing -

`user_id` (Integer): Database user identifier for usage tracking - **Process:** 1. **Prompt Engineering:** Constructs system prompt with specific instructions - Sets AI personality as technical assistant - Specifies response format (natural paragraphs with Markdown/LaTeX) - Includes user message in structured format 2. **API Call:** Sends request to OpenAI with configured parameters - Model: `gpt-4.1-nano-2025-04-14` (cost-optimized) - Single-turn conversation format - Error handling for API failures and rate limits 3. **Usage Tracking:** Records API consumption and costs - Extracts token usage from response - Calculates cost based on current pricing - Updates user's monthly usage statistics 4. **Database Updates:** Maintains user analytics - Increments AI task completion counter - Updates monthly usage percentage - Creates dashboard record if needed

Cost Calculation Token Pricing Model:

```
cost = (prompt_tokens * 0.0001 + completion_tokens * 0.0004) / 1000
```

Usage Tracking: - Prompt tokens: Input text tokenization count - Completion tokens: Generated response tokenization count - Cost calculation: Based on OpenAI's current pricing structure - Monthly usage: Calculated as percentage of $25 monthly limit

New cost helpers:

```
usage = compute_usage_from_messages(
    messages=[{"role": "system", "content": systemprompt}, {"role": "user", "content": user_
    completion_text=response_text,
    model_key="mini",
)
```

- `estimate_message_cost` and `compute_usage_from_messages` now underpin every AI call, providing consistent USD costs that billing routes subtract from organisation balances.

Cache.py - Redis-Backed Memoisation Layer

Location: `novusai/src/backend/utils/Cache.py`

Purpose: Replaces the legacy in-process cache with a Redis-backed implementation that survives worker restarts and scales across containers.

Highlights: - Serialises arbitrary Python objects with pickle/base64 and prefixes keys with `CACHE_REDIS_PREFIX` to avoid collisions. - Provides `set_many`, `get_many`, and namespaced memoisation helpers so routes can cache expensive lookups without bespoke Redis wiring. - Falls back gracefully when cached values cannot be deserialised, returning the raw payload instead of failing the request.

Example usage:

```
mem_cache.set(f"user:role:{username}", role_name, ttl=ttl_seconds)
keys = mem_cache.keys(prefix="org:usage:")
```

- The same instance is imported across routes (auth, billing, broad questions) to keep cache semantics consistent.

---

redis_client.py - Central Redis Connection Factory

Location: `novusai/src/backend/utils/redis_client.py`

Purpose: Supplies configured Redis clients (default + secondary DBs) based on environment variables used by the cache and payment subsystems.

Highlights: - Exposes `get_redis_client()` for general caching and `redis_client/JWT_REDIS_PREFIX` for auth token revocation. - Supports password-protected deployments and configurable database selections without code changes.

---

auth_helperes.py - JWT Guard Rails

Location: `novusai/src/backend/utils/auth_helperes.py`

Purpose: Provides the `@token_required` decorator and helper utilities that secure every protected route.

Highlights: - Extracts Bearer tokens from headers and validates them with PyJWT using the configured algorithm and secret (`JWT_SECRET`, `JWT_ALG`). - Mirrors session state into Redis (`JWT_REDIS_PREFIX`) so revocations or TTL expiries take effect immediately across workers. - On successful decode, enriches `request.user` with Redis claims (role, organisation, etc.) and ensures downstream handlers always have an `id` field.

Decorator usage:

```
@token_required
def answer_route():
    user_id = request.user["id"]
    ...
```

- `dashboard_user_required` wraps routes that need convenient access to the current user ID while retaining the original response signature.

---

user_helpers.py - User Context & Audit Trail

Location: `novusai/src/backend/utils/user_helpers.py`

Purpose: Supplies helpers for reading cached user metadata, resolving organisation slugs, and writing per-user audit logs.

Highlights: - `get_user_role()` and `get_organization()` cache lookups in Redis for 5 minutes to avoid repeated Postgres reads on hot paths. - `log(action)` writes both a tenant-wide `Master.log` entry and a per-user log file under `/Logs/<org>/<user>.log`, tagging each line with timestamps and usernames. - Provides lightweight accessors (`get_user_id`, `get_username`) that lean on the `request.user` payload inserted by `token_required`.

---

system_logger.py - Structured System Logging

Location: `novusai/src/backend/utils/system_logger.py`

Purpose: Centralises infrastructure-level logging for operational events that are not tied to a single end user.

Highlights: - `log_system_event` appends JSON-serialised metadata to `Master.log` with UTC timestamps and severity levels. - Mirrored console output uses colour-coded severity (`INFO`, `WARNING`, `ERROR`, `DEBUG`) for quick scanning inside container logs. - Helper wrappers (`log_warning`, `log_error`, `log_debug`) keep call sites terse and consistent.

---

qdrant_helper.py - Vector Store Management

Location: `novusai/src/backend/utils/qdrant_helper.py`

Purpose: Wraps Qdrant client calls for creating per-user collections, inserting embeddings, and evicting cached RAG artefacts.

Highlights: - `ensure_user_collection` guarantees the collection dimension matches the active embedding model; if it drifts, the helper rebuilds the collection with the correct vector size. - `invalidate_user_rag_cache` sweeps Redis keys (`rag:{user_id}:*`) whenever documents are deleted so subsequent queries see fresh context. - Provides high-level helpers (`delete_document_points_by_ids`, `upsert_point`, `search`) consumed by the upload pipeline, memory glue, and RAG services.

---

memory_glue.py - Conversational Memory Service

Location: `novusai/src/backend/utils/memory_glue.py`

Purpose: Maintains long-lived chat memory in Qdrant so assistants can recall prior messages on subsequent turns.

Highlights: - Embeds each utterance with OpenAI's `text-embedding-3-small` and stores it in the per-user `chat_memory` collection through `qdrant_helper.upsert_point`. - `retrieve_context` re-embeds the incoming query, filters by `session_id`, and returns the top-`k` historical messages ranked by cosine similarity. - `build_messages_with_memory` injects the retrieved snippets into the system prompt so the downstream model receives an ordered memory block.

End-to-end helper:

```
answer = chat_with_memory(user_id, session_id, user_input)
```

- The routine saves user/assistant turns, fetches relevant context, queries OpenAI, and persists the response to maintain continuity.

AI Response Configuration System Prompt: The function uses a carefully crafted system prompt to ensure consistent, high-quality responses: - Technical assistant persona for professional interactions - Natural paragraph formatting for readability - Markdown support for rich text formatting - LaTeX integration for mathematical expressions - Context-aware responses based on application domain

Model Selection: - Primary: `gpt-4.1-nano-2025-04-14` for cost efficiency - Fallback options can be configured for different use cases - Model parameters optimized for legal and technical document processing

---

gmail_helpers.py - Gmail API Operations

Location: `novusai/src/backend/utils/gmail_helpers.py`

Purpose: Provides utility functions for Gmail API integration and email processing.

Key Functions `get_user_credentials(user_id)` - **Purpose:** Retrieves and deserializes stored Gmail OAuth credentials - **Parameters:** `user_id` (Integer): Database user identifier - **Process:** 1. Queries `gmail_tokens` table for user's credential data 2. Deserializes JSON credential string 3. Creates Google OAuth2 Credentials object 4. Validates credential integrity and expiration - **Returns:** `google.oauth2.credentials.Credentials` object - **Error Handling:** Raises exception for missing or invalid credentials - **Usage:** Called before any Gmail API operations

`list_user_emails(creds, max_results=10)` - **Purpose:** Fetches and processes recent emails from Gmail account - **Parameters:** - `creds` (Credentials):

Authenticated Google OAuth2 credentials - `max_results` (Integer): Maximum number of emails to retrieve (default: 10) - **Process:** 1. **Service Creation:** Builds authenticated Gmail API service client 2. **Message Listing:** Calls Gmail API to retrieve message IDs 3. **Detail Retrieval:** Fetches complete message data for each email 4. **Header Parsing:** Extracts relevant information from email headers - Subject line extraction with fallback for missing subjects - Snippet generation for preview purposes - Timestamp and sender information processing 5. **Data Formatting:** Returns structured email data - **Returns:** List of tuples: `[(subject, snippet), ...]` - **Error Handling:** Manages API rate limits, network errors, and authentication issues

Gmail API Integration Details **Authentication Flow:** 1. OAuth2 credentials stored securely in database 2. Credentials include access token, refresh token, and scope information 3. Automatic token refresh handling for expired credentials 4. Secure credential storage with JSON serialization

Email Processing Capabilities: - **Subject Extraction:** Parses email headers for subject information - **Snippet Generation:** Creates preview text from email content - **Header Analysis:** Extracts metadata including sender, recipient, date - **Attachment Handling:** Framework for processing email attachments - **Thread Support:** Capability for conversation thread processing

API Scope Management: - Minimal required permissions for security - Read-only access to preserve user privacy - Scope validation during OAuth consent process

---

auth_helpers.py - Authentication Utilities

Location: `novusai/src/backend/utils/auth_helpers.py`

Purpose: Provides authentication decorators and JWT token validation utilities.

Key Functions **`@token_required` Decorator** - **Purpose:** Validates JWT tokens and protects API endpoints - **Functionality:** Function decorator that wraps protected routes - **Process:** 1. **Token Extraction:** Retrieves token from Authorization header - Expects format: `Bearer <jwt_token>` - Handles missing or malformed headers 2. **Token Validation:** Decodes and verifies JWT token - Uses configured JWT secret key - Validates token signature and structure - Checks token expiration timestamp 3. **User Data Injection:** Adds decoded user information to request - Sets `request.user` with token payload - Includes user ID, username, and other claims 4. **Error Handling:** Returns appropriate HTTP responses for failures - 401 Unauthorized for missing tokens - 401 Unauthorized for expired tokens - 401 Unauthorized for invalid tokens

Usage Example:

```python
@auth_bp.route('/protected', methods=['GET'])
@token_required
def protected_route():
    user_id = request.user['id']
    username = request.user['username']
    # Protected functionality here
```

`@dashboard_user_required` Decorator - **Purpose:** Enhanced authentication for dashboard-specific operations - **Functionality:** Combines token validation with user ID extraction - **Process:** 1. Validates JWT token using standard process 2. Extracts user ID from token payload 3. Passes user ID as first parameter to wrapped function 4. Handles missing user ID in token payload - **Usage:** Simplifies dashboard route implementations

JWT Configuration Token Structure:

```
{
  'id': user_database_id,
  'username': user_login_name,
  'exp': expiration_timestamp,
  'iat': issued_at_timestamp
}
```

Security Features: - HS256 algorithm for token signing - Configurable expiration times (default: 2 hours) - Secret key protection via environment variables - Automatic token validation on all protected routes

Error Response Format:

```
{
  "error": "Token is missing" | "Token expired" | "Invalid token"
}
```

---

user_helpers.py - User Data Utilities

Location: `novusai/src/backend/utils/user_helpers.py`

Purpose: Provides convenience functions for accessing user information from request context.

Key Functions `get_user_id()` - **Purpose:** Retrieves current user's database ID from JWT token - **Returns:** Integer user ID - **Prerequisites:** Must be called within `@token_required` decorated route - **Usage:** Simplifies access to user ID in protected routes - **Implementation:** `return request.user.get("id")`

`get_username()` - **Purpose:** Retrieves current user's username from JWT token - **Returns:** String username - **Prerequisites:** Must be called within

`@token_required` decorated route - **Usage:** User identification and logging - **Implementation:** `return request.user.get("username")`

Usage Context These utility functions are designed to be used within Flask routes that are protected by the `@token_required` decorator. They provide convenient access to user information without requiring direct token parsing in application code.

Example Usage:

```
@chat_bp.route('/example', methods=['POST'])
@token_required
def example_route():
    user_id = get_user_id()      # Gets current user's database ID
    username = get_username()    # Gets current user's login name
    # Route logic here
```

Email Polling System

Location: `novusai/src/backend/polling.py`

Purpose: Implements a sophisticated, continuous email monitoring and processing service that transforms Gmail content into searchable, AI-analyzable data structures.

System Architecture

The email polling system serves as a critical bridge between Gmail's API and NovusAI's AI-powered analysis capabilities. It operates as a long-running daemon service that:

- **Monitors Multiple Gmail Accounts:** Multi-tenant processing with isolated contexts per user
- **Extracts Rich Metadata:** Comprehensive parsing of headers, content, and attachments
- **Generates Vector Embeddings:** AI-powered semantic representations using OpenAI embeddings
- **Maintains Dual Storage:** Structured data in PostgreSQL, vectors in Qdrant
- **Ensures Data Consistency:** ACID-compliant transactions with checkpoint-based recovery

Core Components 1. Email Discovery and Authentication

```
# Multi-user OAuth2 credential management
for user_id in get_all_user_ids():
```

```
    creds = get_user_credentials(user_id)
    gmail_service = build('gmail', 'v1', credentials=creds)
```

2. Incremental Processing Pipeline

```
# Checkpoint-based resumable processing
last_seen = get_last_seen_email_id(user_id)
messages = gmail_service.users().messages().list(userId='me', maxResults=5)
```

3. Content Extraction and Parsing

```
# Comprehensive metadata extraction
meta = extract_email_metadata(msg_data, headers)
email_string = format_email_to_string(user_id, email_id, meta)
```

4. Dual-Storage Architecture

```
# Relational storage for structured queries
save_email_to_db(user_id, email_id, meta)
# Vector storage for semantic search
save_embedding(user_id, email_id, email_string)
```

Email Processing Pipeline

Stage 1: Discovery and Authentication

- **User Enumeration:** Query database for users with active Gmail integration
- **Credential Retrieval:** Load OAuth2 tokens from secure database storage
- **API Authentication:** Establish authenticated Gmail API service connections
- **Rate Limit Management:** Respect Google API quotas and usage limits

Stage 2: Email Retrieval and Filtering

- **Message Listing:** Fetch recent email message IDs with configurable batch sizes
- **Checkpoint Comparison:** Compare against last processed email to avoid duplicates
- **Incremental Processing:** Process only new emails since last successful run
- **Error Isolation:** User-specific failures don't affect other users

Stage 3: Content Extraction and Parsing

- **Header Analysis:** Extract all RFC 2822 compliant email headers
- **Multipart Processing:** Handle complex email structures with attachments
- **Content Decoding:** Base64 decode email bodies with error recovery

- **HTML Sanitization:** Safe conversion of HTML content to searchable text
- **Attachment Metadata:** Extract file information without downloading content

Stage 4: Data Transformation and Storage

- **Metadata Normalization:** Transform Gmail API responses to consistent schema
- **Content Formatting:** Optimize email content for AI analysis and embeddings
- **Database Storage:** Insert structured data into PostgreSQL with ACID guarantees
- **Vector Generation:** Create high-dimensional embeddings using OpenAI models
- **Checkpoint Updates:** Mark processing progress for resumable operations

Performance Characteristics

Throughput and Scalability

- **Processing Rate:** 5 emails per user per 30-second cycle (150 emails/hour/user)
- **Multi-User Capacity:** Handles hundreds of users with isolated processing
- **Memory Efficiency:** Single-email processing prevents memory accumulation
- **API Efficiency:** Minimal API calls through incremental checkpoint system

Error Handling and Recovery

- **Graceful Degradation:** Individual user failures don't stop service
- **Automatic Recovery:** Checkpoint system enables seamless service restart
- **Data Consistency:** Transaction rollbacks prevent partial processing states
- **Comprehensive Logging:** Detailed error context for debugging and monitoring

Resource Management

- **Connection Pooling:** Efficient database connection reuse across users
- **Memory Management:** Processes emails individually to limit peak usage
- **API Quota Management:** Configurable rate limiting for Gmail API compliance

- **CPU Optimization:** Efficient text processing and vector operations

Configuration and Deployment

Service Configuration

```
# Current hardcoded values (should be environment-configurable)
POLLING_INTERVAL = 30        # Seconds between polling cycles
MAX_RESULTS_PER_USER = 5     # Emails per user per cycle
EMBEDDING_MODEL = "openai-text-embedding-3-large"
VECTOR_DIMENSIONS = 3072     # Model-specific embedding size
```

Environment Requirements

- **Python Dependencies:** Flask, psycopg2, google-api-python-client, qdrant-client
- **External Services:** Gmail API access, OpenAI API key, Qdrant instance
- **Database Schema:** PostgreSQL with emails, email_chunks, gmail_tokens tables
- **Network Access:** Outbound HTTPS for APIs, inbound connections to databases

Deployment Considerations

- **Process Management:** Use systemd, supervisor, or PM2 for daemon operation
- **Resource Monitoring:** Track memory usage, API quotas, and processing rates
- **Log Management:** Structured logging for operational visibility and debugging
- **Health Checks:** Monitor database connectivity, API availability, and processing status
- **Scaling Strategy:** Horizontal scaling through multiple service instances with user sharding

Security and Compliance

Data Security

- **OAuth2 Token Management:** Secure storage and refresh of Gmail access credentials
- **Multi-Tenant Isolation:** Complete data separation between user accounts
- **Content Sanitization:** Safe HTML processing prevents XSS and injection attacks
- **API Security:** Parameterized queries prevent SQL injection vulnerabilities

Privacy Protection

- **Minimal Data Exposure:** Process only necessary email metadata and content
- **Secure Storage:** Encrypted database connections and credential storage
- **Access Control:** User-specific data isolation throughout processing pipeline
- **Audit Trails:** Complete logging of data access and processing operations

Compliance Features

- **Data Retention:** Configurable email storage and deletion policies
- **Processing Logs:** Comprehensive audit trails for regulatory requirements
- **Right to Deletion:** Support for user data removal and account deactivation
- **International Compliance:** Timezone-aware processing and storage

Integration Points

Frontend Integration

- **Dashboard Analytics:** Email processing statistics and user metrics
- **Search Interface:** Vector-powered semantic email search capabilities
- **Account Management:** Gmail integration status and configuration options

Backend Services

- **Chat System:** Semantic search integration for AI-powered email queries
- **Analytics Engine:** Processing metrics and usage statistics
- **Notification System:** Real-time updates on email processing status

External APIs

- **Gmail API:** Primary data source for email content and metadata
- **OpenAI API:** Embedding generation for semantic search capabilities
- **Qdrant API:** Vector storage and similarity search operations

Monitoring and Observability

Operational Metrics

- **Processing Rate:** Emails processed per hour per user
- **Error Rate:** Failed processing attempts and recovery statistics
- **API Usage:** Gmail and OpenAI API consumption and quota utilization
- **Storage Growth:** Database and vector storage utilization trends

Health Indicators

- **Service Availability:** Continuous operation status and uptime tracking
- **Database Connectivity:** PostgreSQL and Qdrant connection health monitoring
- **API Connectivity:** External service availability and response time tracking
- **Resource Utilization:** Memory, CPU, and network usage patterns

Alert Conditions

- **Processing Failures:** Consecutive failures for individual users
- **API Quota Exhaustion:** Approaching Gmail or OpenAI usage limits
- **Storage Issues:** Database connection failures or capacity constraints
- **Performance Degradation:** Significant increases in processing time

---

Vector Database Integration (Qdrant)

Location: Qdrant integration spans `polling.py`, `qdrant_t.py`, and utility functions

Purpose: Provides high-performance vector storage and semantic similarity search capabilities for AI-powered email analysis.

Qdrant Architecture Overview

Qdrant serves as NovusAI's vector database solution, enabling semantic search across email content through high-dimensional vector embeddings. The integration provides:

- **Vector Storage:** Efficient storage of 3072-dimensional OpenAI embeddings
- **Similarity Search:** Sub-second cosine similarity queries across millions of vectors
- **Metadata Filtering:** Combined vector and attribute-based search capabilities
- **Horizontal Scaling:** Distributed deployment support for large-scale operations

Vector Collection Structure **Collection Name:** `email_chunks` **Vector Configuration:**

```
VectorParams(
    size=3072,                    # OpenAI text-embedding-3-large dimensions
    distance=Distance.COSINE      # Cosine similarity for semantic relationships
)
```

Point Structure:

```
PointStruct(
    id=uuid4_string,              # Unique identifier for cross-system reference
    vector=embedding_vector,       # 3072-dimensional semantic representation
    payload={
        "user_id": str,           # Multi-tenant isolation
        "email_id": str,          # Cross-reference to PostgreSQL
        "embedding_model": str    # Model version tracking
    }
)
```

Integration with Email Processing

Embedding Generation Pipeline

1. **Content Preparation:** Email content formatted by `format_email_to_string()`
2. **API Request:** OpenAI embedding generation via `embeddings()` utility
3. **Vector Validation:** Dimension verification and model consistency checks
4. **Collection Management:** Automatic collection creation and configuration
5. **Point Storage:** Vector insertion with metadata payload
6. **Reference Storage:** PostgreSQL metadata linking for relational queries

Search and Retrieval Operations

```
# Semantic search across user's emails
search_results = qdrant.search(
    collection_name="email_chunks",
    query_vector=query_embedding,
    query_filter=Filter(
        must=[
            FieldCondition(
                key="user_id",
                match=MatchValue(value=user_id)
            )
        ]
    ),
    limit=10
)
```

Performance Characteristics

Query Performance

- **Search Latency:** Sub-100ms similarity searches on million+ vector collections
- **Throughput:** Thousands of concurrent search operations per second

- **Accuracy:** High-precision semantic matching with configurable similarity thresholds
- **Filtering:** Efficient metadata filtering without performance degradation

Storage Efficiency

- **Vector Compression:** Automatic compression for reduced storage footprint
- **Memory Management:** Intelligent caching for frequently accessed vectors
- **Disk Optimization:** Efficient storage layout for large-scale deployments
- **Index Management:** Automatic indexing for optimal search performance

Scalability Features

- **Horizontal Scaling:** Multi-node deployment support for large datasets
- **Load Balancing:** Automatic query distribution across cluster nodes
- **Replication:** Data redundancy and failover capabilities
- **Backup Support:** Snapshot and restore operations for data protection

Configuration and Deployment

Qdrant Server Configuration

```yaml
# qdrant.yaml example configuration
service:
  host: 0.0.0.0
  port: 6333

storage:
  storage_path: "./qdrant_storage"

log_level: INFO

cluster:
  enabled: false  # Enable for distributed deployment
```

Connection Management

```
# Application connection configuration
qdrant = QdrantClient(
    host="localhost",      # Qdrant server host
    port=6333,            # Default Qdrant port
    timeout=30            # Connection timeout
)
```

Collection Management

- **Automatic Creation:** Collections created on first use with proper configuration
- **Schema Evolution:** Support for embedding model changes and dimension updates
- **Maintenance Operations:** Automatic optimization and cleanup procedures
- **Monitoring Integration:** Performance metrics and health status reporting

Security and Access Control

Network Security

- **Connection Security:** TLS encryption for client-server communication
- **API Authentication:** Token-based authentication for production deployments
- **Network Isolation:** Private network deployment recommendations
- **Firewall Configuration:** Port restrictions and access control lists

Data Protection

- **Tenant Isolation:** User-specific data separation through payload filtering
- **Access Patterns:** Application-level access control and audit logging
- **Data Encryption:** At-rest encryption for sensitive vector data
- **Backup Security:** Encrypted backup storage and secure recovery procedures

Integration with Chat System

Query Processing Flow

1. **User Query:** Natural language input from chat interface
2. **Query Embedding:** Convert user query to vector using same embedding model
3. **Similarity Search:** Find semantically similar email content in Qdrant
4. **Result Filtering:** Apply user-specific filters and access controls
5. **Content Retrieval:** Cross-reference vector results with PostgreSQL metadata
6. **Response Generation:** Provide relevant email context to AI chat system

Semantic Search Capabilities

- **Intent Understanding:** Semantic matching beyond keyword-based search

- **Context Awareness:** Understanding of email relationships and threading
- **Multilingual Support:** Cross-language semantic matching capabilities
- **Temporal Filtering:** Time-based constraints combined with semantic similarity

---

Database Schema

Database Configuration

Database System: PostgreSQL
Connection Management: psycopg2 (Python) and pg (Node.js)
Location: `novusai/src/backend/db.py`

Connection Details

```
conn = psycopg2.connect(
    dbname=os.getenv("PGDATABASE"),
    user=os.getenv("PGUSER"),
    password=os.getenv("PGPASSWORD"),
    host=os.getenv("PGHOST"),
    port=os.getenv("PGPORT"),
    sslmode=os.getenv("PGSSLMODE", "require")
)
```

Table Schemas

users Table **Purpose:** Stores user authentication and profile information

Column	Type	Constraints	Description
id	SERIAL	PRIMARY KEY	Auto-incrementing user identifier
username	VARCHAR(255)	UNIQUE, NOT NULL	User login identifier
password	VARCHAR(255)	NOT NULL	BCrypt hashed password
created_at	TIMESTAMP	DEFAULT NOW()	Account creation timestamp

user_chat_sessions Table **Purpose:** Stores chat conversation history and metadata

Column	Type	Constraints	Description
id	SERIAL	PRIMARY KEY	Auto-incrementing session identifier
chathash	VARCHAR(255)	UNIQUE, NOT NULL	URL-safe session identifier

Column	Type	Constraints	Description
username	VARCHAR(255)	FOREIGN KEY	Reference to users.username
chat_data	JSONB	NOT NULL	Conversation history and metadata
created_at	TIMESTAMP	DEFAULT NOW()	Session creation timestamp

chat_data JSON Structure:

```
{
  "chatname": "Human-readable session title",
  "user": ["User message 1", "User message 2"],
  "bot": ["AI response 1", "AI response 2"]
}
```

gmail_tokens Table **Purpose:** Stores OAuth credentials for Gmail integration

Column	Type	Constraints	Description
id	SERIAL	PRIMARY KEY	Auto-incrementing token identifier
user_id	INTEGER	FOREIGN KEY	Reference to users.id
email_address	VARCHAR(255)	NOT NULL	Gmail account email address
credentials_json	TEXT	NOT NULL	Serialized OAuth2 credentials
linked_at	TIMESTAMP	DEFAULT NOW()	Account linking timestamp
last_seen_email_id	TEXT		Last processed email for resumable polling

emails Table **Purpose:** Stores comprehensive email metadata and content from Gmail API processing

Column	Type	Constraints	Description
user_id	UUID	NOT NULL	Multi-tenant isolation key
gmail_msg_id	TEXT	PRIMARY KEY	Gmail's unique message identifier
thread_id	TEXT		Gmail conversation thread identifier

Column	Type	Constraints	Description
subject	TEXT		Email subject line
from_addr	TEXT		Sender email address and display name
to_addrs	TEXT		Primary recipient addresses
cc_addrs	TEXT		Carbon copy recipient addresses
bcc_addrs	TEXT		Blind carbon copy recipients
date	TIMESTAMPTZ		Original email timestamp with timezone
snippet	TEXT		Gmail-generated preview text
body	TEXT		Consolidated email content for search
labels	TEXT[]		Gmail labels as PostgreSQL array
created_at	TIMESTAMPTZ	DEFAULT NOW()	Record creation timestamp
updated_at	TIMESTAMPTZ	DEFAULT NOW()	Last modification timestamp
email_id	TEXT		Compatibility field (duplicate of gmail_msg_id)

email_chunks Table **Purpose:** Stores metadata for email vector embeddings in Qdrant

Column	Type	Constraints	Description
id	UUID	PRIMARY KEY	Unique identifier matching Qdrant point ID
user_id	INTEGER	FOREIGN KEY	Reference to users.id
email_id	TEXT	FOREIGN KEY	Reference to emails.gmail_msg_id
embedding_dim	INTEGER	NOT NULL	Vector dimension count
embedding_model	VARCHAR(255)	NOT NULL	Embedding model identifier
created_at	TIMESTAMPTZ	DEFAULT NOW()	Record creation timestamp

Column	Type	Constraints	Description
updated_at	TIMESTAMP WITH TIME ZONE	DEFAULT NOW()	Last modification timestamp

Composite Unique Index: `(user_id, email_id)` for deduplication

dashboard Table **Purpose:** Tracks user analytics and usage statistics

Column	Type	Constraints	Description
id	INTEGER	PRIMARY KEY	Reference to users.id
emails_processed	INTEGER	DEFAULT 0	Total emails analyzed
documents_uploaded	INTEGER	DEFAULT 0	Total files processed
ai_task_completed	INTEGER	DEFAULT 0	Total AI operations
monthly_usage	DECIMAL(10,4)	DEFAULT 0	Monthly usage percentage

entries Table (Node.js Server) **Purpose:** General data storage for Node.js operations

Column	Type	Constraints	Description
id	SERIAL	PRIMARY KEY	Auto-incrementing entry identifier
text	TEXT	NOT NULL	Entry content
created_at	TIMESTAMP	DEFAULT NOW()	Entry creation timestamp

Database Operations

Connection Management

- Persistent connections maintained throughout application lifecycle
- Automatic reconnection handling for database failures
- Connection pooling for performance optimization
- SSL/TLS encryption for data transmission security

Transaction Handling

- Explicit transaction management for data consistency
- Rollback operations for error recovery
- ACID compliance for all database operations
- Connection-level transaction isolation

Server Configuration

Flask Server (Python)

Location: `novusai/src/backend/server.py`

Purpose: Main Python application server handling AI operations and API routes.

Server Configuration `create_app()` Function: - **Purpose:** Factory function creating configured Flask application instance - **Components:** 1. **Flask Instance:** Creates base application with default configuration 2. **CORS Setup:** Enables cross-origin requests for frontend integration 3. **Secret Key:** Configures JWT signing key (environment variable recommended) 4. **Blueprint Registration:** Mounts API route modules - **Returns:** Configured Flask application ready for deployment

Blueprint Registration:

```
app.register_blueprint(auth_bp)       # Authentication routes (/api/register, /api/login)
app.register_blueprint(chat_bp)       # Chat management (/api/chat/*)
app.register_blueprint(gmail_bp)      # Gmail integration (/api/gmail/*)
app.register_blueprint(dashboard_bp) # Dashboard analytics (/api/dashboard, /api/emails)
```

Development Server Configuration: - **Debug Mode:** Enabled for development with auto-reload - **Port:** 5001 (configurable via environment) - **Host:** Localhost binding for local development

Security Configuration CORS Policy: - Enables cross-origin requests for React frontend - Configurable origins for production deployment - Credential support for authenticated requests

JWT Configuration: - Secret key management via environment variables - Token expiration policies - Signature validation algorithms

---

Node.js Server (Express)

Location: `novusai/src/backend/server.js`

Purpose: Dedicated server for PostgreSQL database operations and general API endpoints.

Server Configuration Express Application Setup:

```
const app = express();
app.use(cors());                    // Cross-origin request handling
app.use(express.json());            // JSON request body parsing
```

Database Connection Pool:

```
const pool = new Pool({
  user: process.env.PGUSER,
  host: process.env.PGHOST,
  database: process.env.PGDATABASE,
  password: process.env.PGPASSWORD,
  port: process.env.PGPORT,
  ssl: { rejectUnauthorized: false }
});
```

API Endpoints `POST /api/entries` - **Purpose:** Creates new database entries - **Parameters:** `text` (String) - Entry content - **Process:** Inserts data into entries table - **Returns:** HTTP 200 on success, error details on failure

`GET /api/entries` - **Purpose:** Retrieves all entries in reverse chronological order - **Returns:** JSON array of entry objects with ID, text, and timestamps - **Ordering:** Most recent entries first (ORDER BY id DESC)

Performance Configuration **Connection Pooling:** - Maintains pool of database connections for efficiency - Automatic connection management and recycling - Configurable pool size and timeout settings

SSL Configuration: - SSL/TLS encryption for database connections - Certificate validation configuration - Production-ready security settings

---

API Endpoints Reference

Authentication Endpoints

POST /api/register **Purpose:** User account creation
Authentication: None required
Request Body:

```
{
  "username": "string",
  "password": "string"
}
```

Response: Success message or error details
Status Codes: 200 (success), 400 (validation error), 500 (server error)

POST /api/login **Purpose:** User authentication and token generation
Authentication: None required
Request Body:

```
{
  "username": "string",
```

```
  "password": "string"
}
```

Response:

```
{
  "message": "Login successful",
  "token": "jwt_token_string",
  "user": {
    "id": 123,
    "username": "string"
  }
}
```

GET /api/user **Purpose:** Current user information retrieval
Authentication: JWT token required
Response: User ID and username from token

Chat Management Endpoints

POST /api/chat/start **Purpose:** Initialize new chat session
Authentication: JWT token required
Request Body:

```
{
  "chathash": "unique_session_id",
  "chatname": "Chat Title"
}
```

Response: Session creation confirmation

POST /api/chat **Purpose:** Send message and receive AI response
Authentication: JWT token required
Request Body:

```
{
  "chathash": "session_id",
  "chatname": "Chat Title",
  "usermsg": "User message content"
}
```

Response:

```
{
  "response": "AI generated response"
}
```

GET /api/chat **Purpose:** Retrieve all user chat sessions
Authentication: JWT token required
Response: Array of chat objects with messages and metadata

PATCH /api/chat//rename **Purpose:** Update chat session title
Authentication: JWT token required
Request Body:

```
{
  "new_name": "Updated Chat Title"
}
```

DELETE /api/chat/ **Purpose:** Remove chat session and history
Authentication: JWT token required
Response: Deletion confirmation

Dashboard Endpoints

GET /api/dashboard **Purpose:** User analytics and usage statistics
Authentication: JWT token required
Response:

```
{
  "emails_processed": 0,
  "documents_uploaded": 0,
  "ai_task_completed": 0,
  "monthly_usage": 0.00
}
```

GET /api/emails **Purpose:** List linked Gmail accounts
Authentication: JWT token required
Response:

```
{
  "email_addresses": ["email1@gmail.com", "email2@gmail.com"]
}
```

Database Endpoints (Node.js)

POST /api/entries **Purpose:** Create database entry
Authentication: None required
Request Body:

```
{
  "text": "Entry content"
}
```

GET /api/entries **Purpose:** Retrieve all entries
Authentication: None required
Response: Array of entry objects with ID, text, and timestamps

---

Setup and Installation

Prerequisites

System Requirements: - Node.js 16+ and npm - Python 3.8+ and pip - PostgreSQL 12+ database server - Qdrant vector database server - OpenAI API account and API key - Gmail API credentials (for email integration)

Frontend Setup

1. Navigate to Frontend Directory:

```
cd novusai/
```

2. Install Node.js Dependencies:

```
npm install
```

3. Install Additional Dependencies:

```
npm install react react-dom react-scripts
npm install react-katex katex
npm install react-markdown remark-math rehype-katex
npm install lucide-react sass
```

4. Start Development Server:

```
npm start
```

Application will be available at `http://localhost:3000`

Backend Setup

1. Navigate to Backend Directory:

```
cd novusai/src/backend/
```

2. Create Python Virtual Environment:

```
python -m venv venv
source venv/bin/activate  # On Windows: venv\Scripts\activate
```

3. Install Python Dependencies:

```
pip install flask flask-cors
pip install psycopg2-binary python-dotenv
pip install werkzeug jwt
```

```
pip install openai google-auth google-auth-oauthlib google-auth-httplib2 google-api-python-c
pip install qdrant-client colorama uuid
```

4. Install and Configure Qdrant Vector Database:

```
# Using Docker (recommended)
docker run -d --name qdrant -p 6333:6333 qdrant/qdrant

# Or install locally following Qdrant documentation
# https://qdrant.tech/documentation/install/
```

4. Configure Environment Variables: Create `.env` file in backend directory:

```
# Database Configuration
PGDATABASE=your_database_name
PGUSER=your_database_user
PGPASSWORD=your_database_password
PGHOST=localhost
PGPORT=5432
PGSSLMODE=require

# JWT Configuration
JWT_SECRET=your-super-secret-jwt-key

# OpenAI Configuration
OPENAI_API_KEY=your-openai-api-key

# Gmail API Configuration
GOOGLE_CLIENT_ID=your-gmail-client-id
GOOGLE_CLIENT_SECRET=your-gmail-client-secret

# Qdrant Configuration
QDRANT_HOST=localhost
QDRANT_PORT=6333
```

5. Start Python Server:

```
python server.py
```

API will be available at `http://localhost:5001`

6. Start Email Polling Service (separate terminal):

```
python polling.py
```

This starts the continuous email monitoring and processing service.

7. Start Node.js Server (separate terminal):

```
node server.js
```

Database API will be available at `http://localhost:5001`

Database Setup

1. Create PostgreSQL Database:

```
CREATE DATABASE novusai_db;
CREATE USER novusai_user WITH PASSWORD 'secure_password';
GRANT ALL PRIVILEGES ON DATABASE novusai_db TO novusai_user;
```

2. Create Database Tables:

```
-- Users table
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    username VARCHAR(255) UNIQUE NOT NULL,
    password VARCHAR(255) NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Chat sessions table
CREATE TABLE user_chat_sessions (
    id SERIAL PRIMARY KEY,
    chathash VARCHAR(255) UNIQUE NOT NULL,
    username VARCHAR(255) REFERENCES users(username),
    chat_data JSONB NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Gmail tokens table
CREATE TABLE gmail_tokens (
    id SERIAL PRIMARY KEY,
    user_id INTEGER REFERENCES users(id),
    email_address VARCHAR(255) NOT NULL,
    credentials_json TEXT NOT NULL,
    linked_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    last_seen_email_id TEXT  -- Checkpoint for resumable email processing
);

-- Email storage table
CREATE TABLE emails (
    user_id UUID NOT NULL,
    gmail_msg_id TEXT PRIMARY KEY,
    thread_id TEXT,
    subject TEXT,
    from_addr TEXT,
    to_addrs TEXT,
```

```
    cc_addrs TEXT,
    bcc_addrs TEXT,
    date TIMESTAMPTZ,
    snippet TEXT,
    body TEXT,
    labels TEXT[],
    created_at TIMESTAMPTZ DEFAULT NOW(),
    updated_at TIMESTAMPTZ DEFAULT NOW(),
    email_id TEXT
);

-- Email vector embeddings metadata table
CREATE TABLE email_chunks (
    id UUID PRIMARY KEY,
    user_id INTEGER REFERENCES users(id),
    email_id TEXT,
    embedding_dim INTEGER NOT NULL,
    embedding_model VARCHAR(255) NOT NULL,
    created_at TIMESTAMPTZ DEFAULT NOW(),
    updated_at TIMESTAMPTZ DEFAULT NOW(),
    UNIQUE(user_id, email_id)  -- Prevent duplicate embeddings
);

-- Dashboard statistics table
CREATE TABLE dashboard (
    id INTEGER PRIMARY KEY REFERENCES users(id),
    emails_processed INTEGER DEFAULT 0,
    documents_uploaded INTEGER DEFAULT 0,
    ai_task_completed INTEGER DEFAULT 0,
    monthly_usage DECIMAL(10,4) DEFAULT 0
);

-- Entries table for Node.js server
CREATE TABLE entries (
    id SERIAL PRIMARY KEY,
    text TEXT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Gmail API Setup

1. Google Cloud Console Configuration: - Create new project in Google Cloud Console - Enable Gmail API for the project - Create OAuth 2.0 credentials - Configure authorized redirect URIs

2. OAuth Consent Screen: - Configure application name and user support

email - Add authorized domains for your application - Request minimal scopes for Gmail read access

3. Download Credentials: - Download client configuration JSON - Extract client ID and secret for environment variables

---

Deployment Guide

Production Environment Setup

1. Environment Configuration: Update environment variables for production:

```
NODE_ENV=production
JWT_SECRET=production-secret-key-128-chars-minimum
PGHOST=production-database-host
PGSSLMODE=require
OPENAI_API_KEY=production-openai-key
QDRANT_HOST=production-qdrant-host
QDRANT_PORT=6333
```

2. Service Deployment: - **Flask API Server:** Main application API on port 5001 - **Node.js Server:** Database operations on port 3001 - **Email Polling Service:** Background daemon for email processing - **Qdrant Vector Database:** Vector storage and search service - **Process Management:** Use systemd, PM2, or Docker for service management

3. Database Security: - Use connection pooling for performance - Enable SSL/TLS for all database connections - Configure firewall rules for database access - Set up automated backups and monitoring - **Qdrant Security:** Configure authentication and network isolation

3. Frontend Build:

```
cd novusai/
npm run build
```

4. Server Deployment: - Use process managers (PM2, systemd) for server reliability - Configure reverse proxy (Nginx) for load balancing - Set up SSL certificates for HTTPS - Implement logging and monitoring - **Email Polling Service:** Deploy as systemd service for automatic restart - **Qdrant Deployment:** Use Docker or native installation with persistence

Example systemd service for email polling:

```
[Unit]
Description=NovusAI Email Polling Service
After=network.target postgresql.service
```

```
[Service]
Type=simple
User=novusai
WorkingDirectory=/path/to/novusai/src/backend
ExecStart=/path/to/venv/bin/python polling.py
Restart=always
RestartSec=10

[Install]
WantedBy=multi-user.target
```

Security Considerations

Authentication Security: - Use strong JWT secret keys (minimum 128 characters) - Implement token refresh mechanisms - Set appropriate token expiration times - Use HTTPS for all authentication endpoints

Database Security: - Use parameterized queries to prevent SQL injection - Implement proper database user permissions - Enable connection encryption and authentication - Regular security updates and patches

API Security: - Implement rate limiting for API endpoints - Use CORS policies appropriate for production - Validate all input data and sanitize outputs - Monitor for suspicious activity and implement logging

Performance Optimization

Frontend Optimization: - Code splitting for reduced bundle sizes - Asset compression and minification - CDN integration for static assets - Progressive Web App (PWA) features

Backend Optimization: - Database query optimization and indexing - Caching strategies for frequently accessed data - Asynchronous processing for heavy operations - Load balancing for high availability

Monitoring and Maintenance

Application Monitoring: - Error tracking and alerting systems - Performance metrics and analytics - User activity monitoring - API usage and cost tracking

Maintenance Procedures: - Regular security updates and patches - Database maintenance and optimization - Backup verification and recovery testing - Capacity planning and scaling assessments

---

Docker Deployment Progress

Current Deployment Status

The NovusAI application has been successfully deployed using Docker containers with the following services:

Successfully Running Services **Backend API Server (Flask)** - **Status:** Healthy and Running - **Port:** 5001 - **Health Check:** http://localhost:5001/api/health

Echo API Endpoint - **Status:** Working - **Purpose:** Request diagnostics and CORS testing - **Endpoint:** http://localhost:5001/api/echo

PostgreSQL Database - **Status:** Healthy and Running - **Port:** 5432 - **Database:** novusai_db - **User:** novus_user

Qdrant Vector Database - **Status:** Running - **Port:** 6333-6334 - **Version:** 1.15.3 - **Purpose:** Vector search engine for AI embeddings

Ngrok Tunnel Service - **Status:** Running - **Purpose:** External access to backend API - **Domain:** mighty-musical-turtle.ngrok-free.app

Refresh In Progress **React Frontend** - **Status:** Refresh Container building but experiencing dependency issues - **Port:** 3000 (when operational) - **Issue:** npm package installation timeouts in container environment - **Solution:** Frontend dependencies are being resolved

Available API Endpoints

The backend Flask server is fully operational with the following endpoints:

- `GET /api/health` - Service health check
- `GET /api/echo` - Request diagnostics
- `/api/auth/*` - Authentication endpoints

- `/api/chat/*` - Chat functionality
- `/api/gmail/*` - Gmail integration
- `/api/dashboard/*` - Analytics dashboard
- `/api/rag/*` - Retrieval Augmented Generation

Docker Container Status

```
NAME             STATUS                      PORTS
novus_backend    Up (healthy)                0.0.0.0:5001->5001/tcp
novus_postgres   Up (healthy)                0.0.0.0:5432->5432/tcp
novus_qdrant     Up                          0.0.0.0:6333-6334->6333-6334/tcp
novus_ngrok      Up                          4040/tcp
novus_frontend   Restarting (dep issues)     0.0.0.0:3000->3000/tcp
```

Next Steps

1. **Frontend Resolution:** Complete React dependency installation
2. **UI Testing:** Access full web interface at http://localhost:3000
3. **Integration Testing:** Test complete user workflows
4. **Production Deployment:** Configure for production environment

---

This documentation provides comprehensive coverage of the NovusAI application architecture, implementation details, and deployment procedures. For additional support or questions, refer to the inline code comments and error handling implementations throughout the codebase.