

AI Knowledge Workspace (RAG-based SaaS)

Complete Design Documentation

Executive Summary

AI Knowledge Workspace is a production-grade, Retrieval Augmented Generation (RAG) system designed to enable users to upload documents, ask contextual questions, and receive AI-powered answers grounded in their own knowledge base. The system integrates modern cloud technologies, asynchronous processing, and vector databases to provide a scalable, secure, and intelligent document search and chat interface.

This document provides comprehensive architectural, design, and implementation guidelines for the entire system.

1. Project Overview

1.1 Project Name (Conceptual)

AI Knowledge Workspace (RAG-based SaaS)

1.2 What It Does

- Users upload documents (PDF, TXT, DOCX)
- Users ask questions in a chat interface
- System retrieves relevant document chunks using semantic search
- AI generates answers grounded in retrieved content
- Users manage workspaces, chats, and documents
- Enterprise-grade security and multi-tenancy support

1.3 Key Features

- Multi-tenant workspace isolation
- Real-time chat with AI
- Document management and versioning
- Vector-based semantic search
- Role-based access control (RBAC)
- Rate limiting and session management
- Asynchronous document processing
- Production-ready scalability

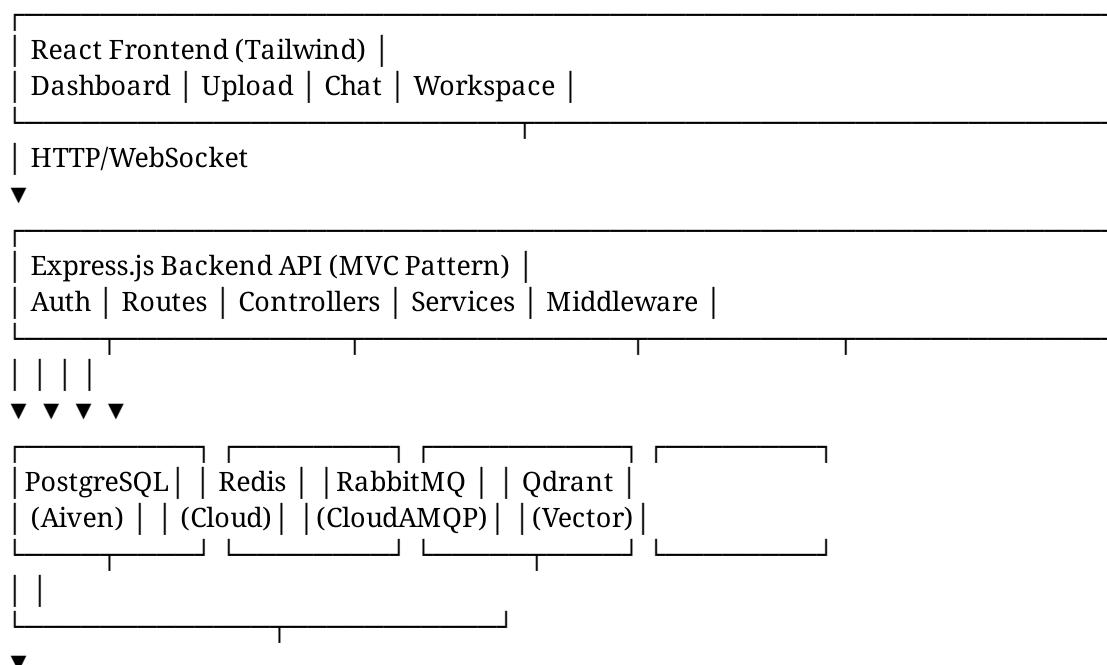
1.4 Technology Stack Summary

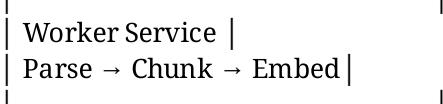
Layer	Technology	Purpose
Frontend	React + Tailwind	UI/UX - Dashboard, Chat, Upload
Backend API	Express.js	RESTful API, Orchestration
Relational DB	PostgreSQL (Aiven)	Metadata, Users, Chats, Documents
Cache Layer	Redis (Cloud)	Sessions, Rate Limits, Caching
Job Queue	RabbitMQ (CloudAMQP)	Async Document Processing
Vector DB	Qdrant	Embeddings, Semantic Search
Background Jobs	Worker Service	Document Parsing, Chunking, Embedding
LLM APIs	Gemini / OpenAI	NLP, RAG Generation

2. High-Level Design (HLD)

2.1 Architecture Overview

The system follows a modern microservices-inspired architecture with clear separation of concerns:





2.2 Component Responsibilities

React + Tailwind CSS (Frontend)

Purpose: User-facing interface for all system interactions

Responsibilities:

- Dashboard: Workspace overview, document management
- File Upload: Drag-and-drop upload, progress tracking
- Chat UI: Real-time chat with AI, message history
- Workspace Manager: Create/manage workspaces
- Settings: User profile, permissions, integrations

Why React:

- Component-based architecture
- Industry standard for SaaS
- Easy integration with WebSocket for real-time chat
- Rich ecosystem (libraries, tools)

Why Tailwind:

- Rapid UI development
- No CSS file bloat
- Consistent design tokens
- Dark mode support out-of-the-box

Express.js (Backend API)

Purpose: The orchestration hub connecting all services

Responsibilities:

- Handle authentication (JWT)
- Expose REST API endpoints
- Validate inputs
- Orchestrate complex workflows
- Connect frontend to backend services
- Manage error handling and logging

Architecture Pattern: MVC

Routes (Express)

- Controllers (Request Handling)
- Services (Business Logic)
- Data Access / Integrations

Why Express:

- Lightweight and flexible
- Minimal overhead
- Perfect middleware ecosystem
- Production-proven

PostgreSQL (Aiven)

Purpose: System of record for all relational data

Data Stored:

- Users (authentication, profiles)
- Workspaces (isolation, ownership)
- Documents (metadata, status, versioning)
- Chats (conversation history, references)
- Messages (user and AI messages)
- Audit logs (compliance)

What NOT to Store:

- ✗ Embeddings (use Qdrant)
- ✗ Large text blobs for search (use Qdrant)
- ✗ Session tokens (use Redis)

Why PostgreSQL:

- ACID transactions
- Relational integrity
- Strong consistency
- Works perfectly with Sequelize ORM

Redis (Redis Cloud)

Purpose: High-speed in-memory caching and session management

Use Cases:

Use Case	Example	Benefit
Session Store	JWT tokens, user sessions	Fast auth, microsecond latency
Rate Limiting	API request counts per user	Protect against abuse
Caching	Chat responses, permissions	Reduce DB/LLM calls
Job Status	Document processing status	Fast polling for UI

Key Patterns:

Cache-aside pattern:

1. Check Redis
2. If miss, fetch from DB
3. Store in Redis with TTL
4. Return to user

Session pattern:

1. Generate JWT
2. Store in Redis with expiration
3. Validate on each request

Why Redis:

- Microsecond latency
- Distributed availability
- Automatic expiration (TTL)
- Atomic operations for counters

RabbitMQ (CloudAMQP)

Purpose: Asynchronous job queue for long-running tasks

Jobs Queued:

- Document parsing (extract text from PDFs, DOCX)
- Text chunking (split into semantic chunks)
- Embedding generation (convert chunks to vectors)
- Vector database insertion (bulk Qdrant inserts)

Queue Architecture:

Producer (Express) → RabbitMQ Queue → Consumer (Worker)

Why Async Processing:

- API responds faster (fire-and-forget)

- Workers scale independently
- Failure isolation (one job doesn't crash API)
- Better resource utilization

Why RabbitMQ:

- Reliable message delivery
- Message acknowledgment
- Dead-letter queues for failed jobs
- Distributed, scalable

Worker Service (Background Processor)

Purpose: Consume and process asynchronous jobs

Workflow:

1. Listen on RabbitMQ queue
2. Receive document upload job (document_id, workspace_id)
3. Extract text from document
4. Split text into semantic chunks
5. Generate embeddings for each chunk
6. Insert chunks + embeddings into Qdrant
7. Update PostgreSQL document status to "processed"
8. Acknowledge job completion

Scaling Strategy:

- Multiple workers can run in parallel
- Horizontal scaling: add more worker instances
- Each worker is stateless

Qdrant (Vector Database)

Purpose: Semantic search engine for document retrieval

Data Structure:

Collection: document_chunks

```

└── id: unique chunk ID
└── vector: embedding (1536 dimensions for Gemini)
└── document_id: reference to document
└── workspace_id: multi-tenancy
└── chunk_text: original text
└── chunk_index: order in document
└── metadata: {source, page_number, ...}

```

Search Mechanism:

1. User asks: "What is the refund policy?"
2. Convert question to embedding
3. Query Qdrant: "Find top 5 most similar chunks"
4. Retrieve chunk texts
5. Pass to LLM with prompt

Why Qdrant:

- Purpose-built for semantic search
- HNSW indexing (fast approximate nearest neighbor)
- Multi-tenancy support
- Filtering by metadata (workspace_id)
- Scales to billions of vectors

LLM APIs (Gemini / OpenAI)

Purpose: Natural language understanding and generation

RAG Generation Flow:

User Question

↓

1. Embed question
2. Search Qdrant (retrieve top chunks)
3. Build prompt:
System: "You are a helpful assistant..."
Context: [Retrieved chunks]
User: "Original question"
4. Call LLM API
5. Stream/return response

Why Gemini:

- Cost-effective pricing
- Good reasoning capabilities
- Fast API response
- Easy to integrate
- Supports streaming responses

2.3 Data Flow Diagrams

Document Upload Flow

React UI

| Upload file

▼

Express API (/upload endpoint)

| Validate file

|— Store in PostgreSQL (document metadata)

|— Publish job to RabbitMQ

|— Return response

Redis (optional caching of document status)

RabbitMQ

| Message: {document_id, workspace_id, file_path}

▼

Worker Service

- | Pull from queue
- | Extract text (PDF/DOCX parser)
- | Chunk text (semantic chunks)
- | Generate embeddings
- | Insert into Qdrant

PostgreSQL

- | Update document status
- | document.status = "processed"
- | document.processed_at = now()

Qdrant

- | Store vectors + metadata
- | Ready for search

Chat Query Flow

React Chat UI

- | User types: "What is..."

▼

Express API (/chat endpoint)

- | Authenticate user
- | Validate workspace access
- | Save message to PostgreSQL
- | Check Redis cache for similar queries
(optional optimization)
- | Embed user question
- | Query Qdrant
 - | Get top 5 relevant chunks
- | Build LLM prompt
 - | System instructions
 - | Retrieved context
 - | User question
- | Call Gemini API
 - | Stream response (or return full)
- | Save AI response to PostgreSQL
 - | messages.text = AI response
- | Return to frontend

Frontend

- | Display response in chat
- | Store in message history

2.4 Security Principles

Authentication & Authorization:

- JWT tokens (stateless)
- Refresh token rotation
- RBAC: Owner, Editor, Viewer roles
- Workspace isolation (query filters)

Data Security:

- TLS 1.3 for all transport
- Secrets in environment variables (.env)
- API key management for LLM services
- Database encryption at rest (Aiven managed)

API Security:

- Rate limiting (Redis-based counters)
- Input validation (sanitization)
- CORS configuration
- Request logging and monitoring

Compliance:

- Audit logging (who accessed what, when)
- Data retention policies
- GDPR considerations (document deletion)

2.5 Scalability Strategy

Horizontal Scaling:

- Stateless Express API (scale replicas)
- Worker service (scale independent of API)
- PostgreSQL read replicas (for read-heavy queries)
- Redis cluster (for distributed cache)
- Qdrant cloud (managed scaling)

Caching Strategy:

- Redis caching of frequently accessed data
- Cache invalidation on updates
- TTL-based expiration

Rate Limiting:

- Per-user API rate limits (Redis counters)
 - Per-workspace document upload limits
 - LLM API call throttling
-

3. Low-Level Design (LLD)

3.1 Database Schema (PostgreSQL)

Users Table

```
CREATE TABLE users (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    email VARCHAR(255) UNIQUE NOT NULL,
    password_hash VARCHAR(255) NOT NULL,
    full_name VARCHAR(255),
    role VARCHAR(50) DEFAULT 'user',
    is_active BOOLEAN DEFAULT true,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    deleted_at TIMESTAMP
);
```

Workspaces Table

```
CREATE TABLE workspaces (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    owner_id UUID NOT NULL REFERENCES users(id),
    name VARCHAR(255) NOT NULL,
    description TEXT,
    plan VARCHAR(50) DEFAULT 'free',
    max_documents INT DEFAULT 100,
    is_active BOOLEAN DEFAULT true,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    deleted_at TIMESTAMP
);
```

Workspace Members (RBAC)

```
CREATE TABLE workspace_members (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    workspace_id UUID NOT NULL REFERENCES workspaces(id),
    user_id UUID NOT NULL REFERENCES users(id),
    role VARCHAR(50) DEFAULT 'viewer', -- owner, editor, viewer
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    UNIQUE(workspace_id, user_id)
);
```

Documents Table

```
CREATE TABLE documents (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    workspace_id UUID NOT NULL REFERENCES workspaces(id),
    filename VARCHAR(255) NOT NULL,
    file_size INT,
    file_type VARCHAR(50), -- pdf, docx, txt
    file_path VARCHAR(255), -- S3 or local path
);
```

```
status VARCHAR(50) DEFAULT 'pending', -- pending, processing, processed, failed
error_message TEXT,
total_chunks INT,
created_by UUID NOT NULL REFERENCES users(id),
created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
processed_at TIMESTAMP,
deleted_at TIMESTAMP
);
```

Chats Table

```
CREATE TABLE chats (
id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
workspace_id UUID NOT NULL REFERENCES workspaces(id),
created_by UUID NOT NULL REFERENCES users(id),
title VARCHAR(255),
is_archived BOOLEAN DEFAULT false,
created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
deleted_at TIMESTAMP
);
```

Messages Table

```
CREATE TABLE messages (
id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
chat_id UUID NOT NULL REFERENCES chats(id),
sender_id UUID REFERENCES users(id), -- NULL if AI message
content TEXT NOT NULL,
is_ai BOOLEAN DEFAULT false,
tokens_used INT, -- For billing
latency_ms INT, -- Response time
retrieved_chunks INT, -- How many chunks were used
created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
deleted_at TIMESTAMP
);
```

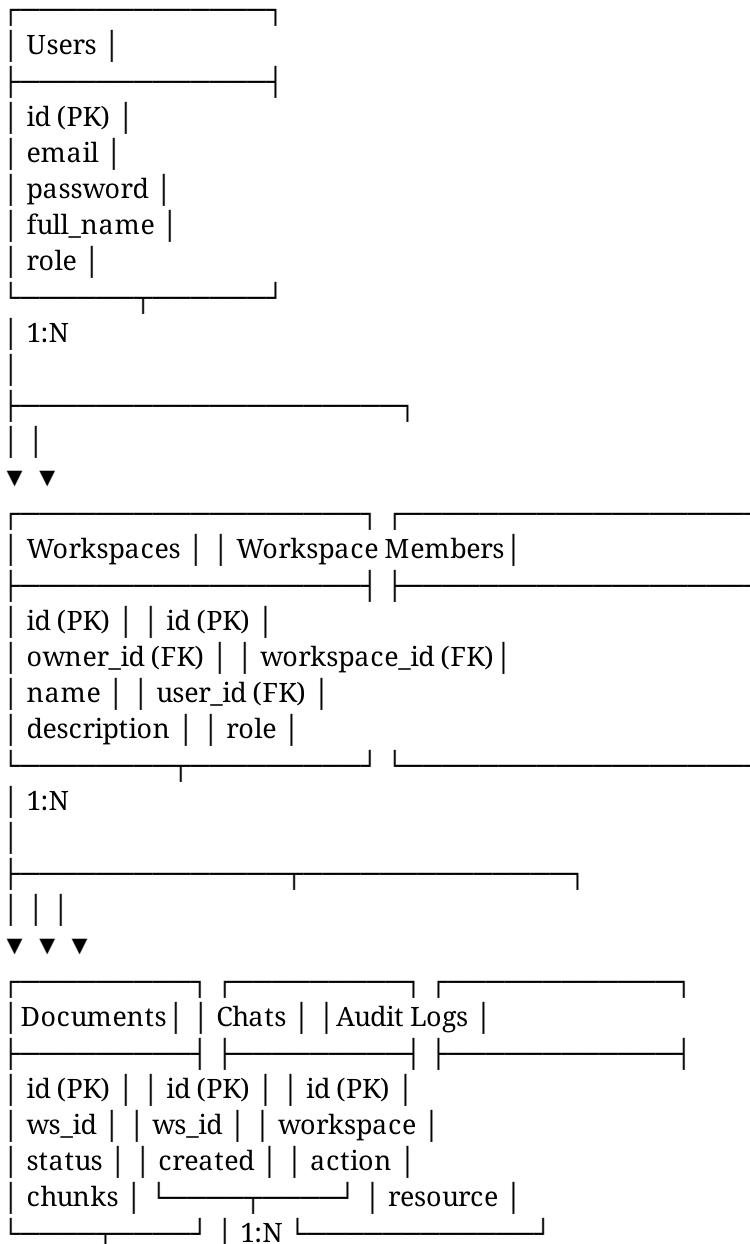
Document Chunks Table (Qdrant Reference)

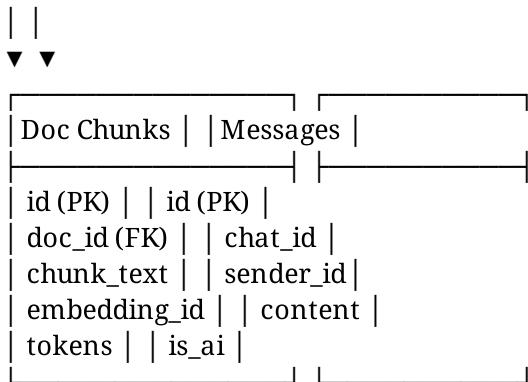
```
CREATE TABLE document_chunks (
id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
document_id UUID NOT NULL REFERENCES documents(id),
workspace_id UUID NOT NULL REFERENCES workspaces(id),
chunk_index INT,
chunk_text TEXT,
embedding_id UUID, -- Reference to Qdrant vector ID
tokens INT,
created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Audit Log

```
CREATE TABLE audit_logs (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    workspace_id UUID REFERENCES workspaces(id),
    user_id UUID REFERENCES users(id),
    action VARCHAR(50), -- upload, chat, delete, etc.
    resource_type VARCHAR(50), -- document, chat, etc.
    resource_id UUID,
    ip_address INET,
    user_agent TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

3.2 Entity Relationship Diagram (ER)





3.3 Express.js Module Structure

Recommended Folder Structure:

```

backend/
  └── src/
      ├── config/ # Configuration files
      │   ├── database.js # PostgreSQL/Sequelize
      │   ├── redis.js # Redis client
      │   ├── rabbitmq.js # RabbitMQ connection
      │   ├── qdrant.js # Qdrant client
      │   └── llm.js # LLM API config
      ├── models/ # Sequelize Models
      │   ├── User.js
      │   ├── Workspace.js
      │   ├── Document.js
      │   ├── Chat.js
      │   ├── Message.js
      │   └── AuditLog.js
      ├── routes/ # Express routes
      │   ├── auth.routes.js
      │   ├── workspace.routes.js
      │   ├── document.routes.js
      │   ├── chat.routes.js
      │   └── index.js
      ├── controllers/ # Request handlers
      │   ├── auth.controller.js
      │   ├── workspace.controller.js
      │   ├── document.controller.js
      │   ├── chat.controller.js
      │   └── health.controller.js
      └── services/ # Business logic
          ├── auth.service.js
          ├── workspace.service.js
          ├── document.service.js
          └── chat.service.js
  
```

```
    └── rag.service.js # RAG orchestration
    └── embedding.service.js
    └── vector-search.service.js
    └── cache.service.js

    └── integrations/ # External API clients
        ├── qdrant.integration.js
        ├── gemini.integration.js
        ├── rabbitmq.producerjs
        ├── redis.client.js
        └── file-storage.js # S3 or local

    └── middleware/ # Express middleware
        ├── auth.middleware.js
        ├── rbac.middleware.js
        ├── rate-limit.middleware.js
        ├── error-handler.middleware.js
        └── validation.middleware.js

    └── utils/ # Helper functions
        ├── logger.js
        ├── validators.js
        ├── error-codes.js
        └── constants.js

    └── migrations/ # Sequelize migrations
        ├── 001-create-users.js
        ├── 002-create-workspaces.js
        └── ...

    └── app.js # Express app setup

    └── worker/ # Background job processor
        └── src/
            └── workers/
                ├── document-parserworker.js
                ├── embedding-generatorworker.js
                └── vector-indexerworker.js
            └── integrations/
            └── services/
                └── index.js # Worker startup

    └── .env # Secrets (not committed)
    └── .env.example # Example template
    └── package.json
    └── docker-compose.yml
    └── README.md
```

3.4 Key Service Layer Implementation

RAG Service (Core Intelligence)

```
// rag.service.js
class RAGService {
  constructor(qdrantClient, geminiClient, embeddingService) {
    this.qdrant = qdrantClient;
    this.gemini = geminiClient;
    this.embedding = embeddingService;
  }

  async generateAnswer(question, workspaceId, topK = 5) {
    // 1. Embed the question
    const questionEmbedding = await this.embedding.embed(question);

    // 2. Search Qdrant
    const chunks = await this.qdrant.search(
      workspaceId,
      questionEmbedding,
      topK
    );

    // 3. Build prompt
    const context = chunks.map(c => c.chunk_text).join('\n\n');
    const prompt = this.buildPrompt(question, context);

    // 4. Call LLM
    const response = await this.gemini.generate(prompt);

    return {
      answer: response.text,
      chunks: chunks,
      tokensUsed: response.tokens,
      latencyMs: response.latencyMs
    };
  }

  buildPrompt(question, context) {
    return `You are a helpful assistant. Answer the user's question based on the provided context.
If the context doesn't contain relevant information, say so.
${question}
${context}`
  }
}
```

```
CONTEXT:  
${context}  
  
QUESTION:  
${question}  
  
ANSWER: `;  
}  
}  
  
module.exports = RAGService;
```

Chat Controller

```
// chat.controller.js  
class ChatController {  
constructor(chatService, ragService, messageService) {  
this.chatService = chatService;  
this.ragService = ragService;  
this.messageService = messageService;  
}  
  
async sendMessage(req, res) {  
try {  
const { chatId, message } = req.body;  
const { userId } = req.user;  
  
// Save user message  
const userMsg = await this.messageService.create({  
chatId,  
senderId: userId,  
content: message,  
isAi: false  
});  
  
// Check cache  
const cached = await this.cacheService.getResponse(message);  
if (cached) {  
return res.json({ message: cached });  
}  
  
// Get workspace context  
const chat = await this.chatService.getById(chatId);  
  
// Generate answer using RAG
```

```

const result = await this.ragService.generateAnswer(
  message,
  chat.workspaceId
);

// Save AI response
const aiMsg = await this.messageService.create({
  chatId,
  content: result.answer,
  isAi: true,
  tokensUsed: result.tokensUsed,
  latencyMs: result.latencyMs,
  retrievedChunks: result.chunks.length
});

// Cache for future
await this.cacheService.setResponse(message, result.answer);

res.json({
  message: aiMsg,
  sources: result.chunks
});
} catch (error) {
  res.status(500).json({ error: error.message });
}

}

module.exports = ChatController;

```

3.5 Worker Service Implementation

Document Processing Worker

```

// worker/src/workers/document-parser-worker.js
const amqp = require('amqplib');

class DocumentParserWorker {
  async start() {
    const connection = await amqp.connect(process.env.RABBITMQ_URL);
    const channel = await connection.createChannel();

```

```
await channel.assertQueue('document_processing', { durable: true });

channel.consume('document_processing', async (msg) => {
  try {
    const job = JSON.parse(msg.content.toString());
    await this.processDocument(job);
    channel.ack(msg);
  } catch (error) {
    console.error('Job failed:', error);
    channel.nack(msg, false, true); // Requeue
  }
});

}

async processDocument(job) {
  const { documentId, workspaceId, filePath } = job;

  // 1. Extract text
  const text = await this.extractText(filePath);

  // 2. Chunk text
  const chunks = this.chunkText(text);

  // 3. Generate embeddings
  const embeddings = await this.generateEmbeddings(chunks);

  // 4. Insert into Qdrant
  await this.qdrant.insertChunks(workspaceId, {
    documentId,
    chunks: chunks.map((text, idx) => ({
      id: `${documentId}_${idx}`,
      text,
      embedding: embeddings[idx],
      metadata: { documentId, workspaceId, chunkIndex: idx }
    }))
  });
}
```

```
// 5. Update database
await this.updateDocumentStatus(documentId, 'processed');

}

chunkText(text, chunkSize = 500, overlap = 50) {
const chunks = [];
for (let i = 0; i < text.length; i += chunkSize - overlap) {
chunks.push(text.slice(i, i + chunkSize));
}
return chunks;
}
}

module.exports = DocumentParserWorker;
```

3.6 API Endpoint Examples

Authentication Endpoints

POST /api/auth/register

Request: { email, password, fullName }

Response: { user, token, refreshToken }

POST /api/auth/login

Request: { email, password }

Response: { user, token, refreshToken }

POST /api/auth/refresh

Request: { refreshToken }

Response: { token }

POST /api/auth/logout

Response: { success: true }

Document Endpoints

GET /api/workspaces/:workspaceId/documents

Response: [{ id, filename, status, created_at, ... }]

POST /api/workspaces/:workspaceId/documents/upload

Request: FormData { file }

Response: { documentId, status: 'pending' }

GET /api/documents/:documentId

Response: { id, filename, status, chunks, ... }

DELETE /api/documents/:documentId

Response: { success: true }

Chat Endpoints

POST /api/workspaces/:workspaceId/chats

Request: { title }

Response: { chatId, title, created_at }

POST /api/chats/:chatId/messages

Request: { message }

Response: { messageId, response, sources, latencyMs }

GET /api/chats/:chatId/messages

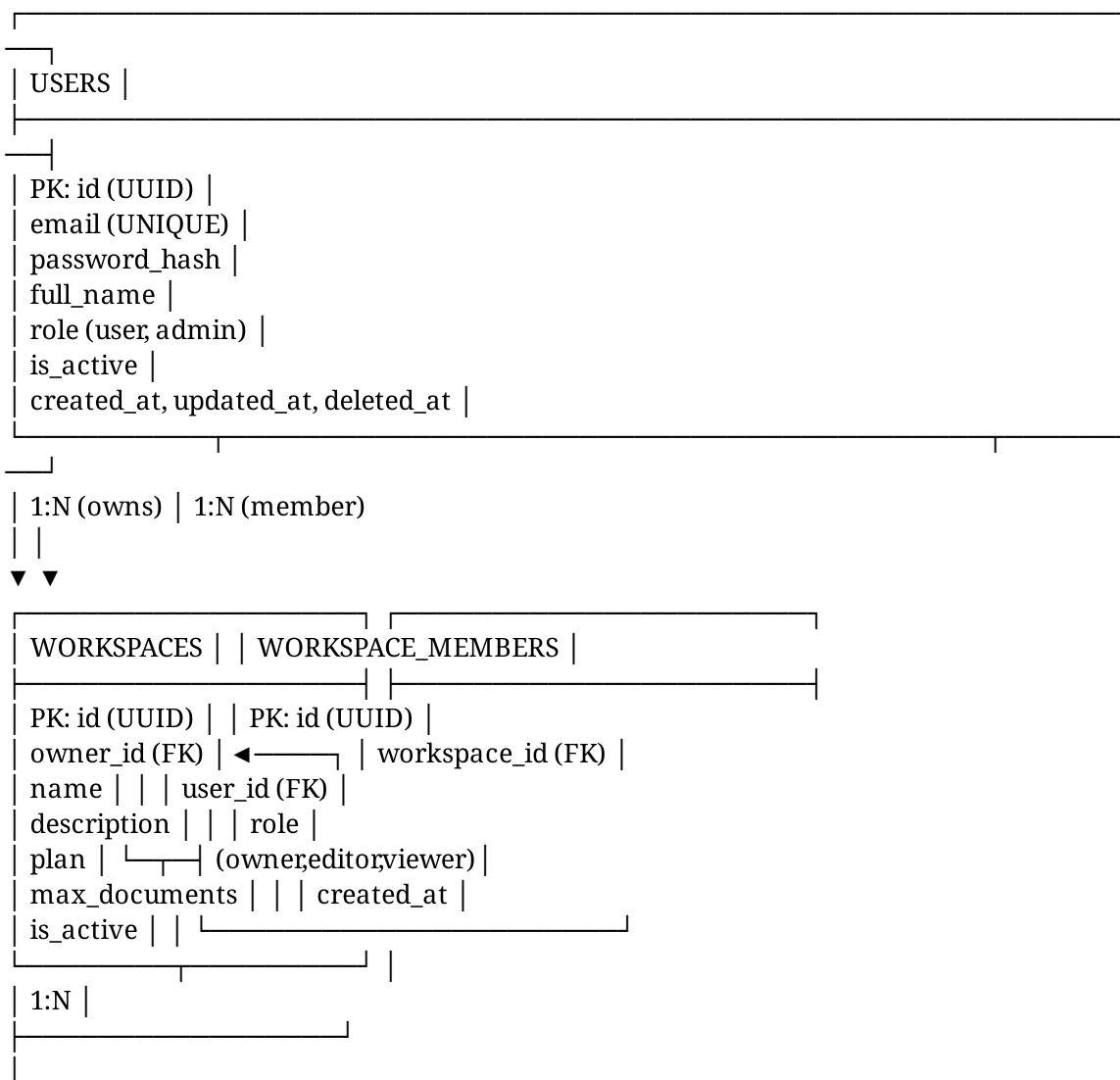
Response: [{ id, sender, content, isAi, created_at, ... }]

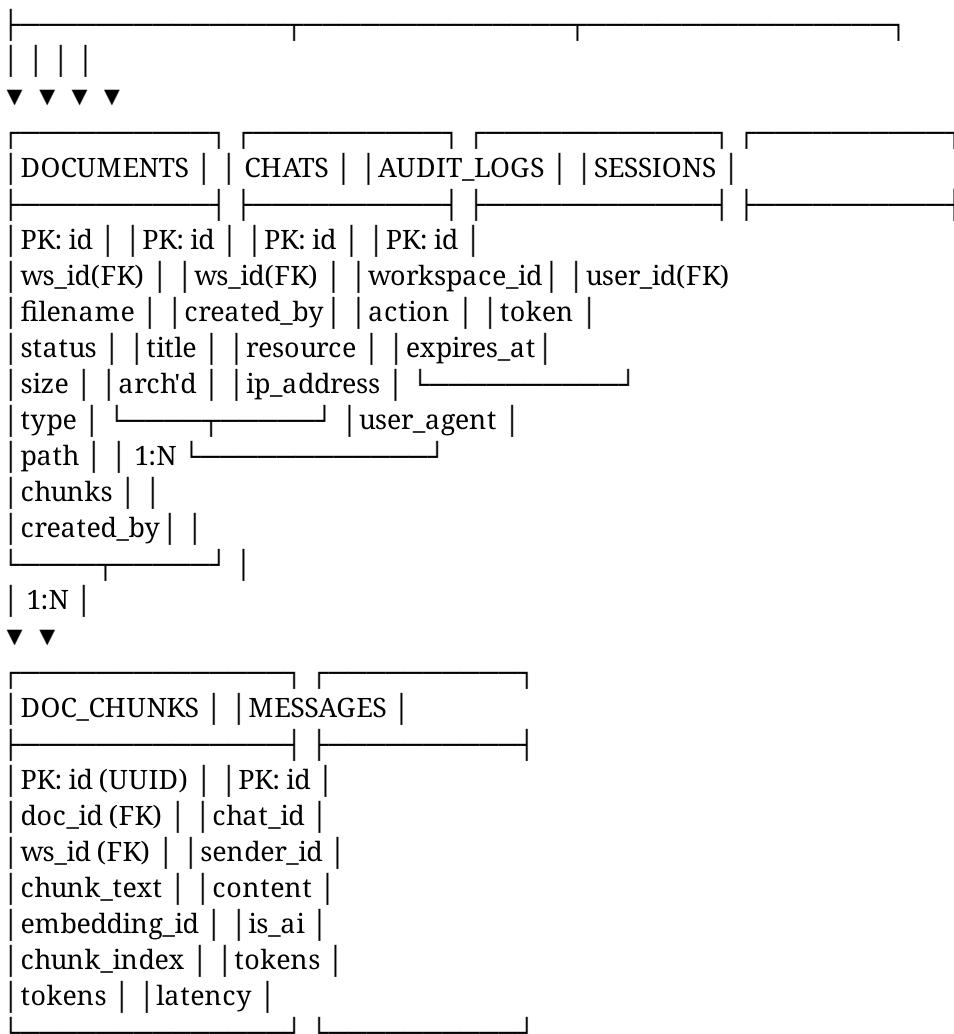
GET /api/chats/:chatId/sources

Response: [{ chunkId, documentId, text, similarity_score }]

4. Entity Relationship Diagram (ER)

4.1 Complete ER Model



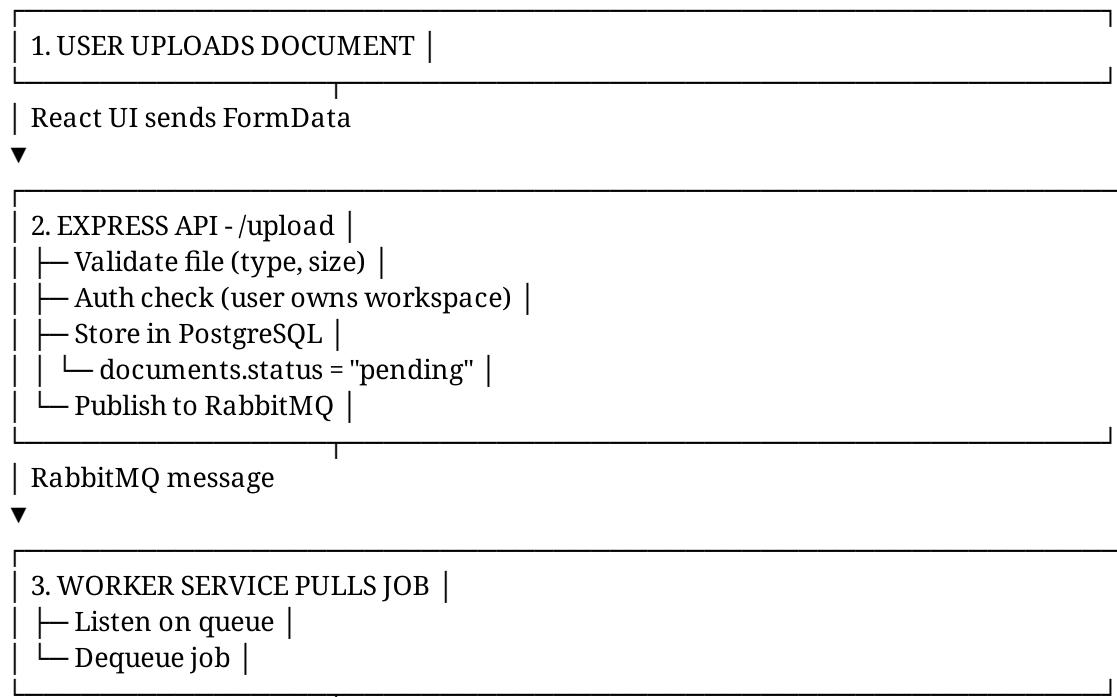


4.2 Key Relationships

Relationship	Type	Description
User → Workspaces	1:N	User owns multiple workspaces
User → Workspace Members	1:N	User is member of multiple workspaces
Workspace → Documents	1:N	Workspace contains multiple documents
Workspace → Chats	1:N	Workspace contains multiple chats
Workspace → Audit Logs	1:N	Track all workspace activities
Document → Doc Chunks	1:N	Document split into chunks
Chat → Messages	1:N	Chat contains conversation messages
Message → (references chunks)	N:M	Message cites document chunks

5. System Flows

5.1 Document Upload and Processing Flow



| Start processing



```
4. EXTRACT TEXT |  
  └─ PDFKit for PDF |  
  └─ Mammoth for DOCX |  
  └─ FS for TXT |
```

| Raw text



```
5. CHUNK TEXT |  
  └─ Semantic chunking (500 chars, 50 overlap) |  
  └─ Generate chunk metadata |
```

| [Chunk1, Chunk2, ...]



```
6. GENERATE EMBEDDINGS |  
  └─ Call embedding model (Gemini API) |  
  └─ Get vectors (1536 dimensions) |
```

| Vectors + chunks



```
7. INSERT INTO QDRANT |  
  └─ Create collection if not exists |  
  └─ Insert points with vectors |  
  └─ Index for fast search |
```

| Success



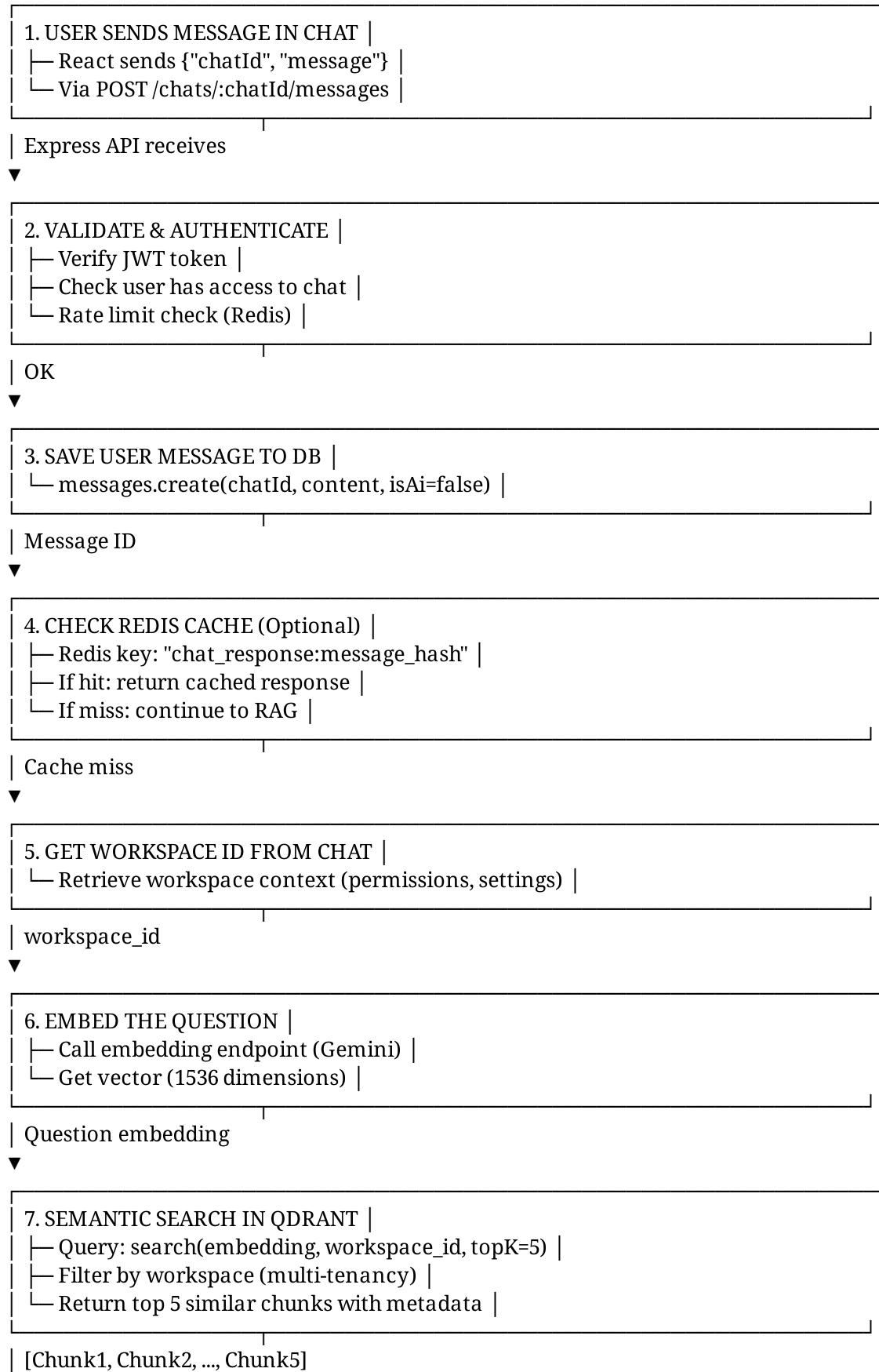
```
8. UPDATE DATABASE STATUS |  
  └─ documents.status = "processed" |  
  └─ documents.total_chunks = count |  
  └─ documents.processed_at = now() |
```

| Complete



```
9. UI POLLING (Optional) |  
  └─ Frontend periodically checks document status |  
  └─ Displays "processed" when complete |
```

5.2 Chat Query and RAG Generation Flow



8. BUILD LLM PROMPT |
 └ System: "You are a helpful assistant" |
 └ Context: Concatenate chunk texts |
 └ User Question: Original message |

| Final prompt

9. CALL GEMINI API |
 └ Stream response (optional) |
 └ Track tokens used |

| AI response text, tokens

10. SAVE AI RESPONSE TO DATABASE |
 └ messages.create(chatId, aiResponse, isAi=true) |
 └ Store metadata: tokensUsed, latencyMs, chunks |
 └ Return to frontend |

| Response object

11. CACHE IN REDIS (Optional) |
 └ Set cache key with TTL (1 hour) |
 └ For future identical queries |

| Complete

12. RETURN TO FRONTEND |
 └ Message object (id, content, created_at) |
 └ Source chunks (for transparency) |
 └ Metrics (latencyMs, tokensUsed) |

6. Technology Stack Deep Dive

6.1 Frontend (React + Tailwind)

Core Libraries:

- react@18.x - UI framework
- react-router-dom - Routing
- axios - HTTP client
- tailwindcss - Styling
- zustand or redux - State management
- react-query or swr - Data fetching

- framer-motion - Animations

Key Components:

- Dashboard (workspace overview)
- DocumentUploader (drag-drop, progress)
- ChatInterface (message list, input, streaming)
- WorkspaceManager (create, manage, share)
- DocumentViewer (view uploaded documents)

6.2 Backend (Express.js)

Core Dependencies:

```
{
  "express": "^4.18.2",
  "sequelize": "^6.35.0",
  "pg": "^8.11.0",
  "redis": "^4.6.0",
  "amqplib": "^0.10.0",
  "jsonwebtoken": "^9.1.0",
  "bcrypt": "^5.1.1",
  "dotenv": "^16.3.1",
  "cors": "^2.8.5",
  "helmet": "^7.1.0",
  "joi": "^17.11.0"
}
```

Middleware Stack:

- helmet - Security headers
- cors - CORS handling
- express.json() - Body parsing
- express-rate-limit - Rate limiting
- Custom auth middleware
- Custom RBAC middleware
- Error handling middleware

6.3 Database (PostgreSQL)

Connection Pool:

- Min connections: 5
- Max connections: 20
- Idle timeout: 30 seconds
- Connection timeout: 5 seconds

Optimization:

- Indexes on foreign keys
- Indexes on frequently queried fields
- Partitioning for large tables (future)
- Read replicas for horizontal read scaling

6.4 Cache Layer (Redis)

Key Patterns:

session:{userId} → JWT token
rate:{userId}:{endpoint} → request count
cache:chat_response:{messageHash} → cached response
cache:permissions:{userId}:{workspaceId} → user role
job_status:{jobId} → processing status

Expiration:

- Sessions: 24 hours
- Rate limits: 1 hour
- Cache: 1 hour (configurable)
- Job status: 24 hours

6.5 Message Queue (RabbitMQ)

Queues:

document_processing
→ Worker processes uploads
→ Dead-letter queue for failures

embedding_generation
→ Generate vectors for chunks
→ Retry logic for API failures

vector_indexing
→ Bulk insert into Qdrant
→ Ensures consistency

Producer (Express):

```
const publishJob = (queue, job) => {
  const serialized = JSON.stringify(job);
  channel.sendToQueue(queue, Buffer.from(serialized), {
    persistent: true,
    contentType: 'application/json'
  });
};
```

Consumer (Worker):

```
channel.consume(queue, async (msg) => {
  const job = JSON.parse(msg.content.toString());
  try {
    await processJob(job);
    channel.ack(msg);
  } catch (error) {
    channel.nack(msg, false, !shouldRetry(error));
  }
});
```

6.6 Vector Database (Qdrant)

Collection Configuration:

```
{  
  "name": "document_chunks",  
  "vectors": {  
    "size": 1536,  
    "distance": "Cosine"  
  },  
  "payload_schema": {  
    "document_id": "keyword",  
    "workspace_id": "keyword",  
    "chunk_text": "text",  
    "chunk_index": "integer"  
  }  
}
```

Search Query:

```
const searchChunks = (embedding, workspaceId, topK = 5) => {  
  return qdrant.search('document_chunks', {  
    vector: embedding,  
    limit: topK,  
    filter: {  
      must: [  
        { key: 'workspace_id', match: { value: workspaceId } }  
      ]  
    }  
  });
};
```

6.7 LLM Integration (Gemini)

API Call Pattern:

```
const generateResponse = async (prompt) => {  
  const response = await gemini.generateContent({  
    contents: [{  
      role: 'user',  
      parts: [{ text: prompt }]  
    }],  
    generationConfig: {  
      temperature: 0.7,  
      topK: 40,  
      topP: 0.95,  
      maxOutputTokens: 2048  
    }  
  });
  return response.text;
};
```

Streaming Response:

```
const streamResponse = async (prompt, callback) => {  
  const stream = gemini.generateContentStream(prompt);
```

```
for await (const chunk of stream) {  
  callback(chunk.text);  
}  
};
```

7. Implementation Roadmap

Phase 1: Core Backend (Weeks 1-3)

- [] Set up Express.js with Sequelize
- [] PostgreSQL schema and migrations
- [] JWT authentication
- [] RBAC (role-based access control)
- [] API endpoints (auth, workspace, documents)
- [] Error handling and logging

Phase 2: Document Processing (Weeks 4-6)

- [] RabbitMQ producer/consumer setup
- [] Document parser (PDF, DOCX, TXT)
- [] Text chunking logic
- [] Embedding generation integration
- [] Qdrant vector database setup
- [] Worker service

Phase 3: Chat & RAG (Weeks 7-9)

- [] Chat endpoints
- [] RAG pipeline implementation
- [] Gemini API integration
- [] Semantic search from Qdrant
- [] Response generation and caching

Phase 4: Frontend (Weeks 10-12)

- [] React setup with Tailwind
- [] Dashboard and workspace UI
- [] Document upload interface
- [] Chat interface with real-time updates
- [] Document viewer

Phase 5: DevOps & Production (Weeks 13-15)

- [] Docker containerization
 - [] Docker-compose for local development
 - [] Kubernetes manifests (optional)
 - [] CI/CD pipeline (GitHub Actions)
 - [] Monitoring and logging (ELK stack)
 - [] Performance optimization
-

8. Security Considerations

8.1 Authentication & Authorization

- JWT with refresh tokens
- Secure password hashing (bcrypt)
- RBAC middleware for workspace access
- API key management for integrations

8.2 Data Protection

- TLS 1.3 for all transport
- Database encryption at rest (managed by Aiven)
- Environment variables for secrets
- Input validation and sanitization

8.3 API Security

- Rate limiting (Redis-based)
- CORS configuration
- CSRF protection
- Helmet.js for HTTP headers

8.4 Compliance

- Audit logging for all operations
- GDPR-compliant document deletion
- Data retention policies
- Regular security audits

9. Performance Optimization

9.1 Caching Strategy

- Cache frequently accessed data in Redis
- Cache RAG responses for identical queries
- Cache user permissions
- TTL-based expiration

9.2 Database Optimization

- Index on foreign keys and query columns
- Connection pooling
- Query optimization
- Read replicas for scaling

9.3 Worker Scaling

- Multiple worker instances for parallel processing
- Job prioritization in queue
- Batch processing for bulk operations

9.4 LLM Optimization

- Prompt optimization for efficiency
- Token-aware response truncation
- Caching of repeated queries
- Stream responses for better UX

10. Monitoring & Logging

10.1 Application Monitoring

- Request/response timing
- Error rate tracking
- API endpoint performance
- Worker job success/failure rates

10.2 Infrastructure Monitoring

- Database query performance
- Redis memory usage
- RabbitMQ queue depth
- Qdrant search latency

10.3 Logging Stack

- Application logs (Winston/Pino)
- Structured logging (JSON)
- Centralized logging (ELK or CloudWatch)
- Audit logs for compliance

11. Cost Optimization

11.1 Cloud Service Costs

- **PostgreSQL:** Use read replicas only when needed
- **Redis:** Right-size instance based on usage
- **RabbitMQ:** Monitor queue depth
- **Qdrant:** Vector storage cost depends on vectors size
- **Gemini API:** Optimize prompts, cache responses

11.2 Strategies

- Cache heavily to reduce LLM calls
 - Batch document processing
 - Compress document storage
 - Archive old chats
-

12. Key Takeaways & Learning Outcomes

By implementing this system, you will gain expertise in:

✓ Production-Grade Architecture

- Microservices patterns
- Asynchronous processing
- Scalable API design
- Cloud service integration

✓ RAG System Implementation

- Vector databases and semantic search
- Prompt engineering
- LLM API integration
- Context retrieval optimization

✓ Full-Stack Development

- React frontend development
- Express.js backend
- PostgreSQL relational database design
- Real-time chat systems

✓ DevOps & Deployment

- Docker containerization
- CI/CD pipelines
- Monitoring and observability
- Production best practices

✓ Advanced Topics

- Multi-tenancy architecture
 - Rate limiting and caching
 - Security and compliance
 - Performance optimization
-

13. References & Resources

RAG Systems:

- [RAG Best Practices ([Orkes.io](#))][web:21]
- [Enterprise RAG Implementation (Intelliarts)][web:25]
- [RAG Evaluation Guide (Maxim AI)][web:23]

Express.js:

- [Express.js Production Best Practices][web:34]
- [Express Architecture Patterns][web:29]
- [Clean Architecture in Express.js][web:36]

Database & ORM:

- [Sequelize Getting Started][web:35]
- [Sequelize Best Practices][web:30]

Vector Databases:

- [Qdrant Documentation](#)

LLM APIs:

- [Google Gemini API](#)
- [OpenAI API](#)

Document Version History

Version	Date	Changes
1.0	2025-01-18	Initial comprehensive design document

Author: AI System Design Team

Last Updated: December 18, 2025

Status: Production-Ready Design
