

Open WebUI Web Search System Documentation

Overview

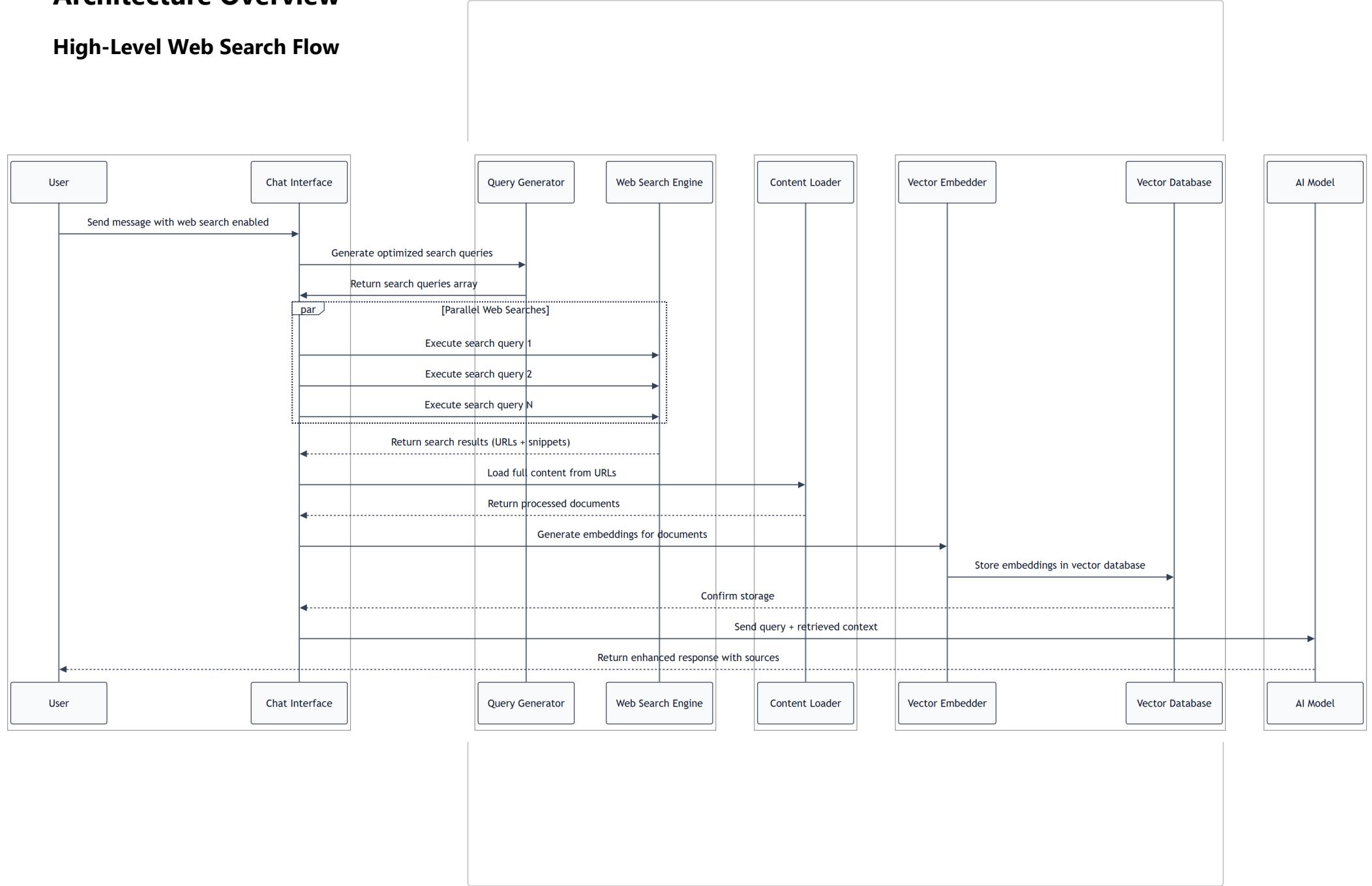
Open WebUI implements a comprehensive web search system that enables real-time information retrieval and integration into AI conversations through Retrieval Augmented Generation (RAG). The system supports 15+ search providers, intelligent query generation, content processing, and seamless integration with chat conversations.

Table of Contents

- [Overview](#)
- [Architecture Overview](#)
- [Supported Search Providers](#)
- [Configuration and Setup](#)
- [Query Generation System](#)
- [Search Execution Engine](#)
- [Content Processing Pipeline](#)
- [Vector Embedding and Storage](#)
- [RAG Integration and Context Injection](#)
- [Advanced Features](#)
- [Frontend Integration](#)
- [API Reference](#)
- [Deployment and Configuration Examples](#)
- [Troubleshooting and Best Practices](#)
- [Conclusion](#)

Architecture Overview

High-Level Web Search Flow



Supported Search Providers

Open WebUI supports a comprehensive range of search engines and APIs:

Free/Open Source Providers

- **SearXNG**: Self-hosted meta-search engine
- **YaCy**: Decentralized peer-to-peer search
- **Ducstom Google search
- **Brave Search API**: Independent search index
- **Kagi Search API**: Premium privacy-focused search
- **Mojeek Search API**: wrapper
- **SerpAPI**: Multi-provider search API aggregator
- **SearchAPI**: Google and Bing search API
- **SerpStack**: Real-time search results API
- **Serply**: Google search API service
- **Tavily**: AI-optimized search API
- **Exa**: Semantic search API for AI applications
- **Perplexity**: AI-powered search and reasoning
- **Sougou**: Chinese search engine API
- **FireCrawl**: Website crawling and search service

External/Custom Providers

- **External Web Search**: Custom API endpoint integration
- **Jina Search**: AI-powered semantic search

Configuration and Setup

Environment Variables

```
# Primary search engine selection
WEB_SEARCH_ENGINE="searxng"
WEB_SEARCH_RESULT_COUNT=10
WEB_SEARCH_CONCURRENT_REQUESTS=10
WEB_SEARCH_TRUST_ENV=true

# Default search provider
# Number of results to retrieve
# Parallel request limit
# Trust environment proxy settings

# Search result processing
ENABLE_WEB_LOADER_SSL_VERIFICATION=true # SSL verification for content loading
BYPASS_WEB_SEARCH_WEB_LOADER=false # Use snippets instead of full content
BYPASS_WEB_SEARCH_EMBEDDING_AND_RETRIEVAL=false # Skip vector processing
WEB_SEARCH_DOMAIN_FILTER_LIST="" # Comma-separated allowed domains

# Provider-specific API configurations
# SearXNG
SEARXNG_QUERY_URL="http://localhost:8080/search"

# Google PSE
GOOGLE_PSE_API_KEY=""
GOOGLE_PSE_ENGINE_ID=""

# Brave Search
BRAVE_SEARCH_API_KEY=""

# Kagi
KAGI_SEARCH_API_KEY=""

# Bing
BING_SEARCH_API_KEY=""

# Commercial API providers
SERPER_API_KEY=""
SERPAPI_API_KEY=""
SEARCHAPI_API_KEY=""
SERPSTACK_API_KEY=""
SERPLY_API_KEY=""
TAVILY_API_KEY=""
EXA_API_KEY=""
PERPLEXITY_API_KEY=""

# External custom search
EXTERNAL_WEB_SEARCH_URL=""
EXTERNAL_WEB_SEARCH_API_KEY=""
```

User Permissions

Web search functionality is controlled by role-based permissions:

```
{  
  "features": {  
    "enable_web_search": true  
  },  
  "permissions": {  
    "features": {  
      "web_search": true  
    }  
  }  
}
```

Query Generation System

Intelligent Query Generation

Open WebUI employs AI-powered query generation to optimize search effectiveness:

Query Generation Process

1. **Context Analysis:** Analyze conversation history and current user prompt
2. **Query Optimization:** Generate multiple targeted search queries
3. **Query Refinement:** Optimize queries for specific search providers
4. **Query Execution:** Execute queries in parallel for maximum efficiency

Query Generation API

```
// Frontend API call  
const queries = await generateQueries(  
  token,  
  model,          // AI model for query generation  
  messages,       // Conversation context  
  prompt,         // User's current message  
  'web_search'    // Query type  
);  
  
// Example generated queries  
[  
  "latest developments artificial intelligence 2024",  
  "AI breakthrough recent research papers",  
  "machine learning advances current trends"  
]
```

Query Generation Prompt Template

The system uses sophisticated prompts to generate effective search queries:

```
Based on the conversation context and user query, generate 1-3 optimized search queries that will help find the most relevant and current information. Format as JSON:
```

```
{  
    "queries": [  
        "specific search query 1",  
        "alternative search query 2",  
        "broader context query 3"  
    ]  
}
```

Consider:

- Current date and temporal relevance
- Specific terminology and technical terms
- Alternative phrasings and synonyms
- Broader context for comprehensive coverage

Search Execution Engine

Search Provider Selection

The system automatically selects the configured search provider and handles failover:

```
def search_web(request: Request, engine: str, query: str) -> list[SearchResult]:  
    """  
    Execute web search using configured provider  
  
    Provider Priority Order:  
    3. Google PSE (if API keys configured)  
    4. Brave Search (if API key configured)  
    5. Other commercial providers...  
    """
```

Search Result Structure

```
class SearchResult(BaseModel):  
    link: str                      # URL of the search result  
    title: Optional[str]            # Page title  
    snippet: Optional[str]          # Content excerpt/description
```

Search Provider Examples

SearXNG Integration

```
def search_searxng(
    query_url: str,
    query: str,
    count: int,
    filter_list: Optional[list[str]] = None,
    **kwargs
) -> list[SearchResult]:
    """
    Search SearXNG instance with advanced filtering

    Supports:
    - Language filtering (language="en-US")
    - Safe search levels (safesearch=0|1|2)
    - Time range filtering (time_range="2023-04-05..today")
    - Category filtering (categories=["general", "science"])
    """

```

Brave Search Integration

```
def search_brave(
    api_key: str,
    query: str,
    count: int,
    filter_list: Optional[list[str]] = None
) -> list[SearchResult]:
    """
    Search using Brave's independent search index

    Features:
    - Independent crawling (not Google-based)
    - Privacy-focused results
    - Commercial API with rate limits
    """

```

Tavily AI Search Integration

```
def search_tavily(
    api_key: str,
    query: str,
    count: int,
    filter_list: Optional[list[str]] = None
) -> list[SearchResult]:
    """
    AI-optimized search specifically designed for RAG applications

    Features:
    - AI-curated results for better relevance
    - Optimized for factual information retrieval
    - Enhanced snippet extraction
    """

```

Content Processing Pipeline

Web Content Loading

After search execution, the system processes search results through a comprehensive content loading pipeline:

Content Loading Strategies

1. Snippet-Only Mode (`BYPASS_WEB_SEARCH_WEB_LOADER=true`)

- Use search result snippets directly
- Faster processing, lower bandwidth
- Limited content depth

2. Full Content Loading (Default)

- Fetch complete web page content
- Extract structured text content
- Remove navigation, ads, and boilerplate

Web Loader Engines

```
def get_web_loader(urls, verify_ssl=True, requests_per_second=10, trust_env=True):  
    """  
        Select appropriate web content loader  
  
        Available Engines:  
        - WebBaseLoader: Standard HTTP requests with BeautifulSoup  
        - PlaywrightURLLoader: JavaScript-enabled browser automation  
        - FireCrawlLoader: Commercial web scraping service  
        - TavilyLoader: AI-optimized content extraction  
        - ExternalWebLoader: Custom API integration  
    """
```

Content Extraction and Cleaning

```
def extract_metadata(soup, url):
    """
    Extract structured metadata from web pages

    Extracted Elements:
    - Page title
    - Meta description
    - Language detection
    - Author information
    - Publication date
    - Content type
    """

def clean_content(content):
    """
    Clean and normalize web content

    Processing Steps:
    - Remove HTML tags and attributes
    - Extract main content area
    - Remove navigation, headers, footers
    - Normalize whitespace and encoding
    - Filter out advertisements and widgets
    """

```

Document Chunking and Processing

The loaded content is processed for optimal RAG integration:

Text Chunking Strategies

```
# Recursive character text splitter
RecursiveCharacterTextSplitter(
    chunk_size=1500,           # Default chunk size
    chunk_overlap=100,          # Overlap between chunks
    separators=["\n\n", "\n", " ", ""]) # Hierarchical splitting
)

# Markdown-aware splitting for structured content
MarkdownHeaderTextSplitter(
    headers_to_split_on=[
        ("#", "Header 1"),
        ("##", "Header 2"),
        ("###", "Header 3")
    ]
)

# Token-based splitting for precise control
TokenTextSplitter(
    encoding_name="cl100k_base", # GPT-4 tokenizer
    chunk_size=1500,
    chunk_overlap=100
)
```

Document Metadata Enhancement

```
class Document:
    page_content: str           # Main text content
    metadata: dict = {
        "source": str,           # Original URL
        "title": str,            # Page title
        "snippet": str,          # Search result snippet
        "link": str,             # Canonical URL
        "language": str,          # Detected language
        "author": str,            # Content author
        "published_date": str,   # Publication timestamp
        "content_type": str,     # MIME type
        "word_count": int,        # Content length
        "chunk_index": int,       # Chunk sequence number
        "total_chunks": int       # Total chunks from source
    }
```

Vector Embedding and Storage

Embedding Generation

Web search content is converted to vector embeddings for semantic search:

Embedding Models Support

```
# Local embedding models (sentence-transformers)
EMBEDDING_MODELS = [
    "sentence-transformers/all-MiniLM-L6-v2", # Default: fast, good quality
    "sentence-transformers/all-mpnet-base-v2", # High quality, slower
    "intfloat/multilingual-e5-large",        # Multilingual support
    "intfloat/multilingual-e5-base",         # Multilingual, smaller
]

# External embedding APIs
EXTERNAL_EMBEDDINGS = [
    "openai/text-embedding-ada-002",          # OpenAI API
    "openai/text-embedding-3-small",            # Latest OpenAI model
    "openai/text-embedding-3-large",            # Highest quality OpenAI
]
```

Vector Database Integration

```
# PostgreSQL with pgvector
VECTOR_DB_CLIENT = "pgvector"
DATABASE_URL = "postgresql://user:pass@host:5432/db"

# ChromaDB for local deployment
VECTOR_DB_CLIENT = "chromadb"
CHROMADB_COLLECTION_NAME = "web-search-{hash}"

# Hybrid search configuration
ENABLE_HYBRID_SEARCH = True
HYBRID_SEARCH_ALPHA = 0.5 # Balance between semantic and keyword search
```

Collection Management

Web search results are organized in temporary collections:

```
def create_web_search_collection(queries: List[str]) -> str:
    """
    Create temporary collection for web search results

    Collection Naming:
    - Format: "web-search-{hash}"
    - Hash: SHA256 of joined search queries
    - TTL: Configurable expiration (default: 24 hours)
    """
    collection_name = f"web-search-{calculate_sha256_string('-'.join(queries))}[:63]"
    return collection_name
```

RAG Integration and Context Injection

Context Retrieval

When processing user queries, the system retrieves relevant web search context:

Hybrid Search Implementation

```
async def query_collection_with_hybrid_search(
    collection_name: str,
    query: str,
    k: int = 10,
    alpha: float = 0.5
) -> List[Document]:
    """
    Perform hybrid search combining:
    - Semantic similarity (vector search)
    - Keyword matching (BM25/FTS)
    - Reranking for optimal relevance
    """

```

Context Selection Strategy

```
def select_relevant_context(
    search_results: List[Document],
    query: str,
    max_context_length: int = 4000
) -> str:
    """
    Select and format context for AI model

    Selection Criteria:
    - Relevance score threshold
    - Diversity of sources
    - Content freshness
    - Authority indicators
    - Context length optimization
    """

```

Response Generation with Citations

The system integrates web search context with AI responses:

Context Formatting

```
def format_web_context(documents: List[Document]) -> str:  
    """  
        Format web search context for AI consumption  
  
        Format:  
        <web_search_context>  
        Source: {title} ({url})  
        Published: {date}  
        Content: {content}  
  
        Source: {title2} ({url2})  
        Content: {content2}  
        </web_search_context>  
    """
```

Citation Integration

```
def generate_response_with_citations(  
    query: str,  
    context: str,  
    sources: List[Dict]  
) -> str:  
    """  
        Generate AI response with automatic source attribution  
  
        Citation Formats:  
        - Inline citations: "According to [Source 1], ..."  
        - Reference list: "Sources:\n1. Title - URL\n2. Title - URL"  
        - Footnote style: "Statement[1]" with source list  
    """
```

Advanced Features

Content Filtering and Safety

Domain Filtering

```
WEB_SEARCH_DOMAIN_FILTER_LIST = [
    "wikipedia.org",
    "reddit.com",
    "stackoverflow.com",
    "github.com"
]

def get_filtered_results(results, filter_list):
    """
    Filter search results by allowed domains

    Filtering Options:
    - Whitelist: Only allow specified domains
    - Blacklist: Exclude specified domains
    - Pattern matching: Use regex for complex rules
    """

```

Content Safety and Validation

```
def validate_web_content(url: str, content: str) -> bool:  
    """  
        Validate web content for safety and relevance  
  
        Validation Checks:  
        - URL safety (no private IPs when disabled)  
        - Content length limits  
        - Language detection  
        - Adult content filtering  
        - Malware/phishing detection  
    """
```

Performance Optimization

Concurrent Processing

```
async def process_web_search_parallel(queries: List[str]) -> List[SearchResult]:  
    """  
        Execute multiple search queries in parallel  
  
        Optimization Features:  
        - Async/await for non-blocking execution  
        - Connection pooling for HTTP requests  
        - Request rate limiting per provider  
        - Timeout handling and retries  
        - Result deduplication  
    """  
  
    search_tasks = [  
        run_in_threadpool(search_web, request, engine, query)  
        for query in queries  
    ]  
  
    search_results = await asyncio.gather(*search_tasks)  
    return search_results
```

Caching Strategy

```
@cached(ttl=3600) # Cache for 1 hour
async def cached_web_search(query: str, engine: str) -> List[SearchResult]:
    """
    Cache search results to reduce API calls

    Caching Levels:
    - Search result caching (Redis)
    - Content loading caching
    - Embedding caching
    - Collection caching
    """

```

Error Handling and Resilience

Provider Failover

```
def search_with_failover(query: str) -> List[SearchResult]:  
    """  
    Implement provider failover for reliability  
  
    Failover Chain:  
    1. Primary provider (configured)  
    2. Secondary provider (if available)  
    3. Free/fallback provider (DuckDuckGo)  
    4. Graceful degradation (empty results)  
    """
```

Rate Limiting and Quotas

```
class SearchRateLimiter:  
    """  
        Manage API rate limits across providers  
  
        Features:  
        - Per-provider rate limiting  
        - Quota tracking and warnings  
        - Automatic backoff strategies  
        - Fair usage across users  
    """
```

Frontend Integration

User Interface Components

Web Search Toggle

```
<!-- Chat input with web search toggle -->  
<script>  
    let webSearchEnabled = false;  
  
    // Auto-enable for certain query patterns  
    $: if (prompt.includes("latest") || prompt.includes("current")) {  
        webSearchEnabled = true;  
    }  
</script>  
  
<div class="web-search-toggle">  
    <button  
        class:enabled={webSearchEnabled}  
        on:click={() => webSearchEnabled = !webSearchEnabled}>  
        >  
             Web Search  
    </button>  
</div>
```

Search Provider Selection

```
<!-- Admin settings for search provider -->
<select bind:value={searchEngine}>
  <option value="searxng">SearXNG</option>
  <option value="brave">Brave Search</option>
  <option value="google_pse">Google PSE</option>
  <option value="tavily">Tavily AI</option>
</select>
```

Real-time Search Status

```
<!-- Live search progress indicator -->
{#if searchInProgress}
  <div class="search-status">
    <div class="spinner"></div>
    <span>Searching the web...</span>
    <div class="progress-bar">
      <div class="progress" style="width: {searchProgress}%"></div>
    </div>
  </div>
{/if}
```

API Reference

Web Search Endpoints

Execute Web Search

```
POST /api/v1/retrieval/process/web/search
Authorization: Bearer {token}
Content-Type: application/json

{
  "queries": [
    "artificial intelligence latest developments",
    "AI breakthrough 2024"
  ],
  "collection_name": "optional-custom-name"
}
```

Response:

```
{
  "status": true,
  "collection_name": "web-search-abc123def456",
  "filenames": [
    "https://example.com/ai-news",
    "https://example.com/ai-research"
  ],
  "docs": [
    {
      "content": "Full page content...",
      "metadata": {
        "source": "https://example.com/ai-news",
        "title": "Latest AI Developments",
        "snippet": "Search result snippet...",
        "published_date": "2024-01-15"
      }
    }
  ],
  "loaded_count": 5
}
```

Query Web Search Results

```
POST /api/v1/retrieval/query/doc
Authorization: Bearer {token}
Content-Type: application/json

{
  "collection_name": "web-search-abc123def456",
  "query": "what are the main AI breakthroughs?",
  "k": 5
}
```

Response:

```
{
  "status": true,
  "results": [
    {
      "content": "Relevant content chunk...",
      "metadata": {
        "source": "https://example.com/ai-news",
        "title": "AI Breakthroughs",
        "relevance_score": 0.92
      }
    }
  ]
}
```

Generate Search Queries

```
POST /api/v1/tasks/queries/completions
Authorization: Bearer {token}
Content-Type: application/json

{
  "model": "llama2:latest",
  "messages": [
    {"role": "user", "content": "What are the latest developments in AI?"}
  ],
  "prompt": "What are the latest developments in AI?",
  "type": "web_search"
}
```

Response:

```
{  
  "choices": [  
    {  
      "message": {  
        "content": "{\"queries\": [\"artificial intelligence latest developments  
2024\"], \"AI breakthrough recent research\", \"machine learning advances current  
trends\"]}"  
      }  
    }  
  ]  
}
```

Deployment and Configuration Examples

Docker Compose Configuration

```
version: '3.8'

services:
  open-webui:
    image: ghcr.io/open-webui/open-webui:main
    environment:
      # Web search configuration
      - WEB_SEARCH_ENGINE=searxng
      - WEB_SEARCH_RESULT_COUNT=10
      - SEARXNG_QUERY_URL=http://searxng:8080/search

      # Provider API keys
      - BRAVE_SEARCH_API_KEY=${BRAVE_API_KEY}
      - TAVILY_API_KEY=${TAVILY_API_KEY}
      - GOOGLE_PSE_API_KEY=${GOOGLE_PSE_KEY}
      - GOOGLE_PSE_ENGINE_ID=${GOOGLE_PSE_ENGINE}

      # Performance settings
      - WEB_SEARCH_CONCURRENT_REQUESTS=5
      - ENABLE_WEB_LOADER_SSL_VERIFICATION=true
    depends_on:
      - searxng
      - redis

  searxng:
    image: searxng/searxng:latest
    volumes:
      - ./searxng-config:/etc/searxng
    environment:
      - SEARXNG_BASE_URL=http://localhost:8080/
```

Kubernetes Deployment

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: web-search-config
data:
  WEB_SEARCH_ENGINE: "tavily"
  WEB_SEARCH_RESULT_COUNT: "10"
  WEB_SEARCH_CONCURRENT_REQUESTS: "10"

---
apiVersion: v1
kind: Secret
metadata:
  name: search-api-keys
type: Opaque
stringData:
  TAVILY_API_KEY: "your-tavily-api-key"
  BRAVE_SEARCH_API_KEY: "your-brave-api-key"
  GOOGLE_PSE_API_KEY: "your-google-api-key"

---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: open-webui
spec:
  template:
    spec:
      containers:
        - name: open-webui
          image: ghcr.io/open-webui/open-webui:main
          envFrom:
            - configMapRef:
                name: web-search-config
            - secretRef:
                name: search-api-keys
```

Troubleshooting and Best Practices

Common Issues and Solutions

Search Provider Connectivity

```
# Test SearXNG connectivity
curl "http://searxng:8080/search?q=test&format=json"

# Validate API keys
curl -H "X-Subscription-Token: $BRAVE_API_KEY" \
"https://api.search.brave.com/res/v1/web/search?q=test"

# Check rate limits
tail -f /var/log/open-webui/web-search.log | grep "rate limit"
```

Performance Optimization

```
# Optimal configuration for production
WEB_SEARCH_RESULT_COUNT = 8          # Balance quality vs speed
WEB_SEARCH_CONCURRENT_REQUESTS = 5    # Avoid rate limiting
CHUNK_SIZE = 1200                     # Optimize for context window
CHUNK_OVERLAP = 150                   # Ensure continuity
```

Memory and Storage Management

```
# Monitor vector database size
SELECT
    schemaname,
    tablename,
    pg_size.pretty(pg_total_relation_size(schemaname || '.' || tablename)) as size
FROM pg_tables
WHERE tablename LIKE '%embedding%';

# Clean up expired collections
DELETE FROM collections
WHERE created_at < NOW() - INTERVAL '24 hours'
    AND collection_name LIKE 'web-search-%';
```

Security Considerations

API Key Protection

```
# Secure API key storage
import os
from cryptography.fernet import Fernet

def encrypt_api_key(key: str) -> str:
    f = Fernet(os.environ['ENCRYPTION_KEY'])
    return f.encrypt(key.encode()).decode()

def decrypt_api_key(encrypted_key: str) -> str:
    f = Fernet(os.environ['ENCRYPTION_KEY'])
    return f.decrypt(encrypted_key.encode()).decode()
```

Content Validation

```
def validate_search_content(content: str, url: str) -> bool:
    """
    Validate search content for security and quality

    Checks:
    - Maximum content length
    - Malicious content detection
    - Language appropriateness
    - Source reliability scoring
    """
    return True # Implement validation logic
```