Deep Search Integration – Technical Documentation

The Deep Search assistant was developed to seamlessly integrate advanced information retrieval and generative summarization capabilities directly into the Onyx platform. Built atop the STORM framework, this assistant is designed to support complex information seeking tasks particularly within the legal and research domains by combining retrieval augmented generation (RAG) with structured multi stage language model pipelines. These include query analysis, outline generation, content synthesis, and polishing, orchestrated through configurable LLMs (e.g., GPT-3.5, GPT-4). The assistant outputs coherent, well-cited articles that include inline references to source material, offering a research grade answer generation workflow that is tightly aligned with the needs of professional and enterprise users.

To ensure reliability, scalability, and API first interoperability, the Deep Search backend is implemented using FastAPI and exposed as asynchronous background jobs to avoid timeout issues in NGINX or Cloudflare environments. SearXNG, a self hosted meta search engine, is used as the core retriever, allowing the system to maintain search flexibility and privacy without dependence on third party APIs like Google or Bing. The Streamlit interface from the original STORM framework was deprecated in favor of a fully integrated React based frontend, enabling contextual, persona driven interactions directly within Onyx's chat environment. With features like real time polling, citation rendering, and assistant specific UI controls, Deep Search offers a production ready, full stack solution for AI-assisted web research.

1) Persona Configuration:

- **Path: onyx/backend/onyx/seeding/personas.yaml**
- **A new persona was added:**

```
- id: 5
  name: "Deep Search"
  description: >
    Advanced assistant for in-depth, real-time web search and comprehensive legal
information retrieval.
  ...
  is_default_persona: true
```

- **This persona acts as the frontend trigger for invoking the Deep Search backend logic, with visibility and priority enabled.**

2) STORM Backend Integration:

- **STORM Repo Cloning:** Cloned https://github.com/stanford-oval/storm.git into the Onyx backend directory to serve as the logic engine for Deep Search.

- STORM orchestrates multi stage LLM tasks including query refinement, outline generation, and citation grounded article creation.

- **Retriever Setup:** Integrated SearXNG as the primary retriever via .env →

  ```
  RETRIEVER="searxng"
  SEARXNG_URL="http://host.docker.internal:8087"
  ```

- All models are OpenAI-based (gpt-3.5-turbo, gpt-4-1106-preview), wired using STORMWikiLMConfigs.

- **Model Configuration:** Set OPENAI_API_KEY, OPENAI_API_TYPE=openai, and encoder settings in .env to power multiple LLM roles inside STORM.

- **Port Exposure:** Exposed 8087 (SearXNG) in Docker and ensured compatibility in run_storm_wiki_gpt.py.

- **Retriever Registration:** Added the SearXNG class to rm.py, with logic to query its /search endpoint and convert results to STORM compatible format.

- The STORM pipeline (STORMWikiRunner) was configured with retriever, LMs, and runner arguments for job execution.

3) SearXNG Setup:

- SearXNG was deployed in its own Docker Compose file with port mappings and API exposure for STORM. The key runtime environment was:

- **Port exposed:** 8087 (via Docker)

- **Accessible URL:** http://host.docker.internal:8087 (used in .env)

- **settings.yml Configuration:** Inside SearXNG's settings.yml, the following values were configured to support internal search, suppress public exposure, and ensure proper headers:

- ```
  server:
    port: 8080
    bind_address: "127.0.0.1"
  ```
  ```

```
  base_url: http://localhost:8087/
  limiter: false
  public_instance: false
  secret_key: "gB9wTAkp93uPclXAXXzb0byV9jPQfC8"
  image_proxy: false
  http_protocol_version: "1.0"
  method: "POST"
  default_http_headers:
    X-Content-Type-Options: nosniff
    X-Download-Options: noopen
    X-Robots-Tag: noindex, nofollow
    Referrer-Policy: no-referrer
```

- **docker-compose.yml:**

```
version: "3"

services:
  searxng:
    image: searxng/searxng:latest
    container_name: searxng
    ports:
      - "8087:8080"
    environment:
      - BASE_URL=http://localhost:8087/
      - INSTANCE_NAME=Techpeek AI
    volumes:
      - ./settings.yml:/etc/searxng/settings.yml
    restart: unless-stopped
```

- **Note:** The base URL was overridden using Docker and .env to ensure internal resolution via **host.docker.internal**. **format is set to json in settings.yml.**

4) STORM Engine Configuration – engine.py:

- **Path: cd onyx/backend/storm/knowledge_storm/storm_wiki/engine.py:**

- This file defines the full pipeline for **Storm's (Deep Search)** execution using the **STORMWikiRunner** class. It orchestrates every phase of the article generation lifecycle:

- LM Configuration – **STORMWikiLMConfigs**

- **The STORMWikiLMConfigs class manages language models for each pipeline stage:**

- **conv_simulator_lm – for conversation simulation**
- **question_asker_lm – for generating search questions**
- **outline_gen_lm – for outline creation**
- **article_gen_lm – for article drafting**
- **article_polish_lm – for final polishing and deduplication**

- **It supports flexible initialization using OpenAI models via LiteLLM, using a shared**

  config method: self.article_gen_lm = LitellmModel(model="gpt-4o-2024-05-13", ...)

5) Retriever Module for DeepSearch - rm.py:

- **The rm.py module implements multiple internet search retrievers to fetch relevant passages in response to user queries. These retrievers augment DeepSearch with real-time information from the web, which is critical for accurate and timely results.**

- **Attempted Retriever: Google Programmable Search (PSE)**

- **Initially, DeepSearch utilized Google PSE via the GoogleSearch class. While conceptually powerful offering structured search through a custom search engine this approach was found unsuitable for production use due to several persistent and critical issues:**

- **Frequent 403 Forbidden errors — even with a valid API key and quota remaining.**

- **SSL-related errors — including: SSL: WRONG_VERSION_NUMBER, SSL: DECRYPTION_FAILED_OR_BAD_RECORD_MAC and Certificate mismatch warnings**

- **Random aborts and crashes — including severe issues like: core dumped and malloc(): corrupted top size. These often stemmed from low-level HTTP client failures or threading conflicts during parallel URL fetching and parsing. These problems made Google PSE unstable and unreliable, especially under load or when web scraping from result URLs was involved.**

- **Final Choice: SearXNG where To address these limitations, DeepSearch transitioned to SearXNG, a self-hosted meta search engine. Fully self-hosted: Eliminates reliance on third-party rate limits and SSL quirks. Configurable backends, You can choose which search engines to use internally (e.g., Brave, Bing, Mojeek). JSON API: Provides structured and consistent results. Robust**

content extraction: Integrated with WebPageHelper for snippet generation from URLs.

6) Unified Language Model Interface for DeepSearch - lm.py:

- **The lm.py module provides a flexible and extensible abstraction layer for integrating various large language models (LLMs) into the DeepSearch system. Its primary goal is to standardize interaction across providers like OpenAI, Azure OpenAI, Groq, Anthropic (Claude), DeepSeek, Google Gemini, and Together.ai, while maintaining compatibility with DSPy and LiteLLM APIs.**

- **Usage in DeepSearch: In DeepSearch, this module powers the backend language reasoning for: Query understanding and decomposition, Document generation, Conversational response generation and Answer refinement and re-ranking.**

- **API keys and endpoints are securely sourced from environment variables:**

```
# ============ language model configurations ============
# Set up OpenAI API key.
OPENAI_API_KEY="sk-proj-ALhAekTXyqPIbh-be0j9DKzU-T09RO8dqjqDL7myTNH
HavWVQdYj1ijE0acpt-ral3MPOgPTAnT3BIbkFJ7GS_ufeliSeDaZ3HWncN1Pha3JX5
grwlB9AsfgU1x5LRTch8eALPT8MWHMuH4AzrH7r2iWdbIA"
# If you are using the API service provided by OpenAI, include the following line:
OPENAI_API_TYPE="openai"
# ============ retriever configurations ============
RETRIEVER = "searxng"
SEARXNG_API_KEY = ""
SEARXNG_URL = "http://host.docker.internal:8087"
# ============ encoder configurations ============
ENCODER_API_TYPE="openai"
```

7) Running DeepSearch: From CLI to Fullstack Integration:

- **Legacy CLI Execution (Deprecated):** **Originally, to run a DeepSearch article generation pipeline using STORM, the following command was used**

```
python examples/storm_examples/run_storm_wiki_gpt.py --output-dir results
--retriever searxng --do-research --do-generate-outline --do-generate-article
--do-polish-article
```

- **This script would Use SearXNG as the retriever and then Perform multi-stage generation: research → outline → article → polish, Save results to the results/ directory**

- **Note:** This CLI flow was useful for testing but is no longer required in production.

8) Web Integration via ChatPage.tsx:

- **Path: cd onyx/web/src/app/chat/ChatPage.tsx**

- DeepSearch is now fully integrated into the Onyx frontend, using the modern chat interface. The workflow is controlled by the onSubmit() function inside ChatPage.tsx.

- **How it works:**

- When the user sends a message through the "Deep Search" persona,
- The onSubmit() function triggers a POST request to /api/deepsearch/submit,
- Then continuously polls /api/deepsearch/status/{job_id} for the result,
- Once completed, it injects the article and citations back into the chat UI.
- This method: Enables background job processing (no browser timeouts), Provides a smooth user experience via real-time status updates, Allows reuse of existing session logic (citations, threads, messages).

- const onsubmit () function:

```
    // Route to /api/deepsearch/submit if assistant is Deep Search
    const isDeepSearch = liveAssistant?.name === "Deep Search";
    if (isDeepSearch) {
      try {
        // Step 1: Submit deepsearch job
        const submitResponse = await fetch("/api/deepsearch/submit", {
          method: "POST",
          headers: {
            "Content-Type": "application/json",
          },
          body: JSON.stringify({ query: currMessage }),
        });

        if (!submitResponse.ok) throw new Error("Failed to submit Deep Search
job");

        const { job_id } = await submitResponse.json();

        // Step 2: Poll for result
        let attempts = 0;
        let result = null;
        const maxAttempts = 100;
```

```
        const delay = (ms: number) => new Promise((res) => setTimeout(res, ms));

        while (attempts < maxAttempts) {
          const statusResponse = await fetch(`/api/deepsearch/status/${job_id}`);
          if (!statusResponse.ok) throw new Error("Failed to get job status");

          const statusData = await statusResponse.json();
          if (statusData.status === "completed") {
            result = statusData.result;
            break;
          } else if (statusData.status === "error") {
            throw new Error(statusData.error || "Deep Search failed");
          }

          await delay(3000); // wait 3s before next poll
          attempts++;
        }

        if (!result) throw new Error("Deep Search timed out");

        const newMessageId = Date.now();

        const userMessage: Message = {
          messageId: newMessageId - 1,
          message: currMessage,
          type: "user",
          files: [],
          toolCall: null,
          parentMessageId: parentMessage?.messageId || SYSTEM_MESSAGE_ID,
        };

        const assistantMessage: Message = {
          messageId: newMessageId,
          message: result.article || result.outline || "No article, outline or citations
was returned from Deep Search.",
          type: "assistant",
          citations: result.citations || {},
          files: [],
          toolCall: null,
          parentMessageId: userMessage.messageId,
        };

        upsertToCompleteMessageMap({
          messages: [userMessage, assistantMessage],
```

```
          chatSessionId: currChatSessionId,
        });

        await setMessageAsLatest(assistantMessage.messageId);
        updateChatState("input");
        resetRegenerationState();
        return;
      } catch (err) {
        console.error("Deep Search Error:", err);
        updateChatState("input");
        setPopup({
          type: "error",
          message: "Deep Search failed to respond.",
        });
        return;
      }
    }
```

9) Transition from OpenAI to LiteLLM with Gemini:

- **Due to persistent rate limiting issues and instability with the previously used LLM providers Groq (llama-3-70b) and Cerebras (llama-3.3-70b) the Deep Search backend has been migrated to use Gemini 2.0 Flash via LiteLLM as the unified inference layer. This change ensures greater reliability and flexibility while maintaining full compatibility with the STORM pipeline. The update involved patching the lm.py file to switch from OpenAI based OpenAIModel to LitellmModel, adjusting the STORMWikiLMConfig, and reconfiguring model routing in run_storm_wiki_gpt.py, engine.py, and .env. Gemini is now the default provider for all stages of the pipeline including conversation simulation, question generation, outline drafting, and article polishing. These changes also align with DSPy's abstraction and make the system more cost-effective, scalable, and vendor-agnostic moving forward.**

- **deepsearch_backend/main.py:**

```
# 1. Set up all language models using correct setter methods
lm_configs = STORMWikiLMConfigs()

conv_simulator_lm = LitellmModel(
    model="gemini/gemini-2.0-flash",
    api_key=os.getenv("LITELLM_API_KEY")
)
question_asker_lm = LitellmModel(
    model="gemini/gemini-2.0-flash",
```

```
    api_key=os.getenv("LITELLM_API_KEY")
)
outline_gen_lm = LitellmModel(
    model="gemini/gemini-2.0-flash",
    api_key=os.getenv("LITELLM_API_KEY")
)
article_gen_lm = LitellmModel(
    model="gemini/gemini-2.0-flash",
    api_key=os.getenv("LITELLM_API_KEY")
)
article_polish_lm = LitellmModel(
    model="gemini/gemini-2.0-flash",
    api_key=os.getenv("LITELLM_API_KEY")
)
```

- **.env:**

```
# ============ language model configurations ============
# LiteLLM Model Based Configurations
LITELLM_API_KEY="AIzaSyB7kJj0nswOoGCNiWEHCl9OdlRs0ECZcnA"
# ============ retriever configurations ============
RETRIEVER = "searxng"
SEARXNG_API_KEY = ""
SEARXNG_URL = "http://host.docker.internal:8087"
```

10) DeepSearch Job System — Solving Long-Running Timeout Issues:

- **Problem Long Response Times Hit Reverse Proxy & CDN Timeouts:** When running DeepSearch using the original synchronous approach via /deepsearch/, article generation could take 60–120 seconds depending on topic complexity. This triggered hard timeouts at various layers:

- **NGINX Reverse Proxy:** default proxy_read_timeout of 60 seconds
- **Cloudflare CDN:** enforced request timeout at 100 seconds (non-configurable on free tiers)

- As a result, long running DeepSearch queries would fail with 504 Gateway Timeout, even if the backend was still working.

- **Solution:** Background Job + Polling Architecture: To solve this, we implemented a background task system inside deepsearch_backend/main.py, and integrated polling logic in the onSubmit() function in ChatPage.tsx.

11) From CLI & Streamlit to Fullstack React Integration:

- **Previous STORM Interfaces: Streamlit app (createNewArticle.py, storm.py and demo_util.py) – used for interactive research and article creation. Deprecated for Production: These interfaces were useful for experimentation but were not suitable for: Authenticated user flows, Background processing, Custom personas and chat threads and Rich UI interactions (like citations and logs).**

- **DeepSearch is now fully integrated into the Onyx chat platform, using: React-based frontend (ChatPage.tsx, AgenticMessage.tsx, MemoizedTextComponents.tsx) and FastAPI backend (deepsearch_backend/main.py).**

12) UI Behavior Customizations for DeepSearch:

- **To ensure clarity and prevent user confusion, certain UI elements and actions are intentionally disabled when using the Deep Search or Case Analysis assistants. These assistants are tightly coupled with structured backend logic and preconfigured models, so allowing end user overrides (like model switching or regeneration) could break consistency.**

- **LLM Model Selection Hidden in Input Bar: In ChatInputBar.tsx, we hide the model picker (LLMPopover) when the selected assistant is either "Deep Search" or "Case Analysis":**

```
{!["Case Analysis", "Deep Search"].includes(selectedAssistant?.name) && (
  <LLMPopover
    llmProviders={llmProviders}
    llmManager={llmManager}
    currentAssistant={selectedAssistant}
    trigger={...}
  />
)}
```

- **These assistants rely on fixed, multi-model pipelines (e.g., GPT-3.5 + GPT-4 + SearXNG) defined in backend logic like STORMWikiRunner, and allowing the user to override the model would: Break intended prompt flow, Invalidate cached results or expectations and Mislead the user into thinking the selected model applies**

- **Disable Answer Regeneration: In ChatPage.tsx, we also disable the "Regenerate" button for Deep Search and Case Analysis responses:**

```
regenerate={
  liveAssistant?.name === "Case Analysis" ||
```

```
  liveAssistant?.id === -4 ||
  liveAssistant?.name === "Deep Search" ||
  liveAssistant?.id === -5
    ? undefined
    : createRegenerator({ ... })
}
```

- These assistants generate structured, multi-step outputs (e.g., outlines, citations, full articles). Regenerating: Can't reproduce the full pipeline from the frontend, May confuse users with inconsistent or partial results and Isn't supported for async jobs (like Deep Search).

13) Deep Search Backend API:

- Path: cd onyx/backend/onyx/deepsearch_backend/main.py

- Summary: The main.py file in the deepsearch_backend module is the central orchestrator that connects user queries to the STORM engine using FastAPI. It exposes Deep Search as an asynchronous job based REST API, enabling robust execution of web augmented search and article generation without frontend timeout issues.

- Language Model Initialization (STORMWikiLMConfigs)

- The file sets up five specialized language models via the OpenAIModel wrapper (from lm.py):

- Model Role: conv_simulator_lm (Simulates conversation for interactive question generation.), question_asker_lm (Actively generates diverse search queries.), outline_gen_lm (Creates a structured outline of the article.), article_gen_lm (Produces the initial article draft.) and article_polish_lm (Refines and enhances the final article.)

- OpenAI Model: gpt-3.5-turbo, gpt-3.5-turbo, gpt-4-1106-preview, gpt-4-1106-preview and gpt-4-1106-preview

- Each is configured with:

  temperature=1.0
  top_p=0.9
  max_tokens=... (500 to 4000 depending on role)

- **These models are assigned using:**

  lm_configs.set_conv_simulator_lm(conv_simulator_lm) # ...and so on for all stages

- **The SearXNG class (from rm.py) is used as the retriever. It hits the self-hosted SearXNG instance running at http://host.docker.internal:8087.**

  rm = SearXNG(
  ** searxng_api_url=SEARXNG_URL,**
  ** k=engine_args.search_top_k,**
)

- **SearXNG performs federated web search across multiple engines (e.g., Google, Bing, Mojeek) and returns structured results in JSON. These are used during the knowledge curation phase to build factual grounding for the article.**

- **All configurations are passed to STORMWikiRunner, the pipeline engine defined in engine.py: runner = STORMWikiRunner(engine_args, lm_configs, rm).**

- **This runner encapsulates the entire generation process in sequential modules:**

- **Knowledge Curation**
- **Outline Generation**
- **Draft Generation**
- **Article Polishing**
- **Citation Link Injection (add_inline_citation_links): After the article is generated, this function parses all [n] references and injects HTML links like [1] This helps render clickable citations in the frontend.**

- **Asynchronous Job Execution: The backend supports async job handling using FastAPI's BackgroundTasks.**

- **/submit Endpoint: Accepts a POST request with the search query. Generates a job_id and Runs background_deepsearch() in the background.**

  @router.post("/submit")
  async def submit_deepsearch_job(...):
  ** job_id = str(uuid.uuid4())**
  ** background_tasks.add_task(background_deepsearch, request.query, job_id)**

- **/status/{job_id} Endpoint: Allows the frontend to poll for job status (pending, completed, or error). Once complete, returns**

- **article (with inline links)**
- **outline**
- **citations (url_to_info.json)**
- **conversation_log (dialogue history)**

- **Output File Management: The STORM runner writes artifacts to**

- **./results/api_run/{topic} →**
- **storm_gen_article_polished.txt**
- **storm_gen_outline.txt**
- **url_to_info.json**
- **conversation_log.json**

- **These are read using: def safe_read(path, is_json=False)**

- **Authentication: All endpoints depend on user=Depends(current_user) This ensures access is restricted to authenticated users only.**

- **Functionality: Wraps the STORM engine and exposes it via an asynchronous background job system. Returns structured outputs including:**

- **Final article (with inline citations)**
- **Outline**
- **Raw citation metadata**
- **Conversation log**

- **Core API Endpoints: POST /deepsearch/submit → Starts a background job for the given query and GET /deepsearch/status/{job_id} → Polls job status and fetches result if complete.**

- **Citation Linking: add_inline_citation_links() function replaces [1], [2] style citations in the generated article with actual <a href> links from the url_to_info.json.**

14) Output Directory:

- **When a Deep Search job runs, STORM generates the following files in ./results/api_run/{topic}/**

- **storm_gen_article_polished.txt**
- **storm_gen_outline.txt**
- **url_to_info.json**
- **conversation_log.json**

- **These are used to construct the frontend response.**

15) Router Integration:

- **Path: cd onyx/backend/onyx/main.py**

- **from onyx.deepsearch_backend.main import router as deepsearch_router**
- **include_router_with_global_prefix_prepended(application, deepsearch_router)**

- **Ensures Deep Search is fully exposed under Onyx's unified API structure.**

16) Conclusion:

- **The Deep Search integration represents a key milestone in transforming STORM from a CLI/Streamlit prototype into a scalable, production ready module within the Onyx platform. By rearchitecting the system as an asynchronous, API driven service using FastAPI, we resolved critical timeout limitations posed by NGINX and Cloudflare, enabling support for long running, multi stage generation tasks without disrupting user experience.**

- **Ibrahim Sultan**