NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

**24S2 SC4052-CLOUD COMPUTING**

**Assignment 2 (Study basics of API in Github and Code Search)**

# Title: GitHub Repository Smart-Search & LLM-Driven Codebase Documenter (GitHub API + Code-Search SaaS)

Submitted By: Teg Singh Tiwana (U2122816B)

Instructor: Chee Wei Tan

Date: 18 April 2025

Page Count: 5 Pages (Pg 3-7)

## Table of Contents

## 1. Abstract

Modern developers drown in unfamiliar code. They struggle to keep up with new and everchanging code projects, API versions and Software Development Kits (SDKs) from constantly updated repositories and interfaces coming out on GitHub. The sheer volume and rate of new open-source software being hosted on GitHub, now in excess of 518 million repositories at the end of 2024 (Appendix 9.1), renders the native GitHub “code search” UI feature, that returns raw lists of files based on primitive search features, obsolete, as it’s manual discovery and comprehension functionality of relevant code increasingly impractical and difficult for developers in 2025. This is especially worsened as developers do not usually update their readme.md documentations to keep up with new updates, leaving other developers in the dark [1] [2].

Thus, this project tries to bridge that gap by delivering a complete end-to-end Software-as-a-Service (SaaS) solution that first **finds** suitable repositories, based on natural text queries, direct GitHub URLs (if the developer wants to bypass the search function) or Local File paths, via the GitHub REST API and then **explains** in-depth the internal structure of any chosen project with help from modern Large-Language-Models (LLMs) via their APIs. The main 3 in-depth functionalities of the SaaS Project are as such: (i) it discovers relevant repositories with a Smart Search front-end powered by the GitHub REST v3/v4 APIs and LLMs to extract relevant entities from the natural query, (ii) lets users filter results by popularity metrics like number of forks done on the project, number of stars, etc. and (iii) explains any selected repository by automatically generating a multi-chapter Markdown tutorial on how the code works and functions using powerful abstraction capabilities of the 3 most popular LLM providers (OpenAI GPT, Anthropic Claude or Google Gemini). Within these 3 providers, users can pick any one of their many selections of available models, from deep-thinking models, like Gemini 2.5 Pro, GPT o1/o1-pro or Claude 3.7 Sonnet, to more lightweight (faster and cheaper models), like gemini-2.0-flash-lite, o1-mini-2024-09-12, gpt-4o-mini-2024-07-18 or Claude 3 Haiku. The exact models the SaaS can take can be found here [3] [4] [5].

The full stack, implemented in less than ~1.9k Lines-of-code (LoC) of Python, comprises a Streamlit web frontend UI [7], a PocketFlow workflow engine [6], and a set of reusable nodes built on top of Pocket Flow, a 100-line minimalist lightweight LLM framework for different implementations, like Agents, Task Decomposition, RAG, etc [6]. For the frontend UI, there is a streamlit web interface that accepts either natural-language queries, direct repository URLs, or local directory paths, presents ranked, filterable search results and finally renders a multi-chapter Markdown tutorial automatically generated by a simplified Knowledge Building workflow. The service is under-pinned by a declarative PocketFlow pipeline consisting of specialised node classes implemented for search, filtering, tutorial generation, and rendering to bridge the problem we identified using LLMs. Experiments on 20 representative open-source Github repos show that the system shortens the path from an English query to a coherent, navigable technical tutorial (i.e. search-to-tutorial) to 62 seconds on average, while the automatic summaries achieve a ROUGE-L overlap of around 0.435 against author-written READMEs. These results indicate that coupling GitHub’s search capabilities with LLM-driven content generation and summarization can convert raw source code into approachable learning resources with minimal user effort, demonstrating both utility and technical novelty relative to prior WebGPT-style prototypes.

## 2. Introduction

Developers, researchers and students frequently confront unfamiliar codebases, especially with dated and outdated readme.md documentations. Although GitHub’s native search interface is powerful and quick, it ultimately returns frequently irrelevant and unstructured list of files or repositories that still demands considerable human effort to interpret and breakdown. Meanwhile, recent advances in generative LLMs have demonstrated an unprecedented ability to reason about code and to produce human-readable explanations for any code inputs. The present work exploits these 2 complementary LLM strengths, which are efficient retrieval and high-level human-understandable explanation, to create a single, browser-based workflow that spans the entire journey from “What libraries implement X?” to “Show me a beginner-friendly overview of this open-source libraries’ architecture”. One example could be “I need React component libraries for Figma-to-code prototyping” or “Find me NTU Blockchain or Decentralized projects”. The system then extracts keywords, queries GitHub, applies star/fork filters, and finally funnels the chosen repository into the LLM tutorial pipeline for users to understand the codebases better.

3 concrete user stories guided the design. First, a new developer onboarding to a new project should be able to paste its URL and receive a structured tutorial describing the project's abstractions and their relationships to flatten the steep learning curve in complex teams. Second, students or hobbyists conducting literature surveys or project experiments can simply formulate a plain-English question/query, receive a curated list of repositories ranked by relevance and popularity, and immediately inspect concise summaries before diving deeper into the repository with their code explanations. Lastly, power users can refine search results through standard open-source health metrics, like number of stars, forks, open issues, and last update, without leaving the same page. The remainder of the report details the technical background, system architecture, implementation decisions, empirical evaluation, and the novel contributions made with respect to existing tools.

# 3. Literature Survey/Background

## 3.1 GitHub REST APIs (v3)

GitHub's REST API v3 provides a very comprehensive and easy-to-access endpoint for repository search (GET /search/repositories), accepting both free-text queries and structured qualifiers that it matches exactly against existing codebases using its in-built optimized and indexed keyword-matching algorithms. Users are further able to filter returned search results by star count (like stars: "x>500"), primary language (like language: "python"), last push date (like pushed:">2024-01-01"), and many other filters that were not covered in this study experiment [8]. While this v3 API interface excels at locating exact word matched popular code artifacts and its ranking algorithm being optimized for matching exact keywords within repository metadata and file contents, the search results fail to hold up against doing semantic searches from natural text queries or producing pedagogically organized overviews and highlighting the conceptual structure of a complete codebase. Moreover, the API returns raw fields such as description and README content but does not itself generate any form of natural-language summary or tutorial on how the codebase runs that is the most up to date to the current version pushed [9].

## 3.2 State-of-the-Art (SoTA) Large Language Models

Most modern Large Language Models (LLMs) have demonstrated an ability to ingest substantial code fragments (up to 128k tokens for OpenAI models, 200k tokens for Anthropic Claud models & 1.048 million tokens for Google Gemini models, which means most code bases can fit into this context window) and to emit human-readable explanations, design overviews, or keyword extractions. Most code-explainer pipelines rely on a spectrum of LLMs, ranging from deep-thinking inference models that excel at multi-step reasoning, to lightweight distillations that deliver near-state-of-the-art quality at a fraction of the cost and latency. All three major providers, Google, Anthropic, and OpenAI, offer both extremes, enabling fine-grained trade-offs between capability and efficiency. Both model types can transform opaque source files into approachable narratives, identifying core abstractions, summarizing complex functions, and even suggesting high-level analogies. Their versatility makes them ideal for a code-explainer pipeline but also requires careful integration to avoid unpredictable latency or excessive API usage [10].

### 3.2.1 Deep-Thinking Inference Models

These higher capacity models dominate the inference plane when maximal reasoning depth is required. They support chain-of-thought ('CoT') prompting, internally generating step-by-step logical decompositions of complex tasks, which yields more accurate and complex summaries of intricate codebases, with lower probabilities of hallucinations compared to the 'vanilla', zero-shot LLM models. Some of the models used in this study include Google Gemini 2.5 Pro (which excels at multi-pass code analysis and hierarchical abstraction), Anthropic Claude 3.7 Sonnet (which is optimized for extended context windows and reliable rationale generation) and OpenAI o1-mini (i.e. "o1-pro" smaller variants which has high throughput with robust SoTA reasoning extensions). These models can process 128k-1.048 million tokens of source code, producing rich, analogy-driven explanations and precise relationship mappings, at the expense of higher per-call latency and token costs.

### 3.2.2 Lightweight Distilled Models

For interactive applications where responsiveness and cost-control are useful, popular LLM providers also offer distilled or flash-lite variants that retain roughly ~60-80% of their larger sibling's accuracy with significantly reduced compute requirements. Some of these models used in this study include: Gemini 2.0-Flash-Lite (which are optimized for sub-second inference on shorter prompts), OpenAI GPT-4o-Mini 2024-07-18 (which is a compact GPT variant tuned for different downstream tasks like code summarization tasks, etc.) and Anthropic Claude 3 Haiku (a distilled Claude model offering rapid turnaround on basic code-explanation queries). By integrating these lighter models into dedicated nodes, like for example the semantic keyword extraction or README summarization functionalities of the SaaS, the system could minimize latency spikes and token expenditure, while reserving the deep-thinking engines for the most complex analysis steps.

## 3.3 Engineering Challenges

Bringing LLMs into production raises 2 primary engineering concerns: firstly, we have latency. Each API call to an external model can incur hundreds of milliseconds (~500-750ms) to several seconds of network and compute time. Unbounded or uncoordinated prompts could further lead to a sluggish user experience. Secondly, we have Token Costs. LLM services typically bill by token processed. Large or repeated prompts, especially when analysing many code files, can become financially prohibitive, thus a move to open-source models could potentially address this issue. Alternative how I solved this issue was through a design that mitigates both issues through locally caching (storing prompt–response pairs locally to avoid redundant API calls) into a local Json file and by restricting all model invocations to specific, well-defined nodes in the data-flow graph, ensuring that each code fragment is analysed at most once per pipeline execution.

## 3.4 Workflow Orchestration with PocketFlow

The entire processing pipeline and workflow is handled by PocketFlow, an open-source minimal-overhead Python library used to define and model LLM workflows as data-flow graphs with a single mutable shared store. In PocketFlow, each discrete operation, from fetching code, extracting keywords, writing chapters, is encapsulated within a Node that implements 3 lifecycle methods used in this study: firstly 'prep fn' (where we gather inputs from the shared store and perform any initial validations), secondly 'exec fn' (where we execute the node's core logic, i.e. calling GitHub API or LLM) and lastly 'post fn' (where we persist the results back into a central shared store/dictionary) (appendix 9.2).

Nodes are connected into Flows by Actions (i.e. labelled edges), which enforce a strict execution order, where the output of one Node becomes the input for the next, without need for any additional processing. Every node also reads from and writes to the same mutable shared object, so the data dependencies are managed implicitly This separation of concerns yields highly modular, easily testable code. PocketFlow also provides specialized node types to handle the different workload patterns used in this experiment. Batch Nodes/Flows break large, data-intensive tasks (i.e. processing dozens of abstractions) into parallelizable chunks, applying a Map-Reduce style. Async Nodes and Flows integrate asynchronous operations, like waiting on external webhooks or long-running analyses, without blocking the main execution thread. Lastly, Parallel Nodes and Flows allow multiple I/O-bound tasks, like simultaneous API calls, to run concurrently, improving overall throughput and reducing tail-end latency (appendix 9.3). By combining these 3 elements, Nodes for individual tasks, Flows to define end-to-end logic, and the Shared Store for implicit data passing, PocketFlow enables a clear, declarative specification of complex, multi-step LLM pipelines needed in this SaaS [6].

## 3.5 User Interface: Streamlit

The front-end is built with Streamlit (v1.33), a Python-native robust frontend framework that requires no JavaScript or React knowledge. Streamlit enables rapid construction of interactive web apps, with interactive text inputs, sliders, buttons, and markdown rendering needed in this SaaS project, using only straightforward Python functions. By leveraging Streamlit's session state and widget callbacks, our experiments UI also remains responsive: search parameters, filter settings, and rendered tutorial pages update instantly as users interact, all without manual page reloads or custom client-side scripting [7].

# 4. System Architecture

## 4.1 End-to-end Pipeline

The complete data pathway begins in the browser, passes through a set of search-specific nodes, merges seamlessly into the pre-existing tutorial generator, and finally returns rendered Markdown back to the user interface. Appendix 9.4-9.10 summarises this flow via flow diagrams of each part.

1) SmartSearchRepo inspects the raw text entered by the user and determines whether it is a URL, a local path, or a genuine natural-language query. In the latter case it invokes an LLM prompt that returns up to a max of six topical and reelvant keywords based on entity matching and semantic understanding of the user prompt.
2) FilterRepos submits the constructed search string to the GitHub API, applies user-defined constraints on stars, forks, issues, and update date, and limits the result set to a max of 10 most relevant repositories. For each candidate it downloads the README and obtains a three-sentence summary via an LLM summarization call.
3) SelectRepository waits for the user to click a "Select" button in the Streamlit card corresponding to the desired repository, it then records either shared.repo_url or shared.local_dir, thereby launching our tutorial workflow that builds a detail 7-to-10-chapter markdown format detailed explainer for the code base.
4) For the six nodes defined for explaining the code base in detail: (FetchRepo through CombineExplainers), we produce an output directory containing index.md, multiple chapter files in markdown format, and a generated Mermaid diagram to explain the code in flow diagrams.
    a. FetchRepo: In the first prep, it reads repo_url or local_dir, include/exclude patterns, token, and size limits from shared. In exec, it calls crawl_github_files or crawl_local_files to fetch and filter source files. In post, writes a list of (path, content) tuples and derived project_name back into shared.
    b. IdentifyAbstractions: This Prep gathers the file list and builds a context listing each file index and path. Exec issues an LLM prompt to extract the top 5–10 abstractions with multi-line descriptions and file-index references. Post parses, validates, and saves [{"name","description","files":[…]}] into shared.abstractions.
    c. AnalyzeRelationships: In this Prep, it reads abstractions and related code snippets and formats a prompt listing each abstraction by index. Exec asks the LLM for a high-level summary plus YAML-formatted relationships (from, to, label). Post validates indices and writes {"summary", "details":[…]} into shared.relationships.
    d. OrderChapters: This Prep pulls abstractions and relationships from shared and outlines them for the LLM. Exec prompts the LLM to rank abstraction indices into the most logical teaching order. Post verifies coverage and order, then writes the resulting index list into shared.chapter_order.
    e. WriteChapters (BatchNode): This BatchNode Prep reads chapter_order, abstractions, and files, initializing an empty history buffer and yielding each abstraction's context. Exec, per item, prompts the LLM with the abstraction's description, code snippets, and prior chapters to generate a Markdown chapter. Post collects all chapter texts into shared.chapters.
    f. CombineExplainers: This last Prep reads project metadata, relationships, chapter order, and chapter contents to auto-generate a Mermaid flowchart and assemble index.md with summary, diagram, and chapter links. Exec creates the output directory, writes index.md and each ##_{name}.md file. Post stores shared.final_output_dir for downstream rendering or download.
5) RenderAndDownload loads these files, streams them into the Streamlit page with st.markdown, and offers a one-click ZIP archive created on-the-fly. (appendix 9.10-9.13 for screenshots of the SaaS App from query to summarization)

## 4.2 Shared-store Additions

To support the new search capability, additional keys, query_input, search_mode, keywords, filter_params, search_results, and selected_repo, were appended to the shared dictionary. All modifications are concentrated inside the new nodes; the older nodes remain oblivious to search-specific information, preserving backwards compatibility.

# 5. Implementation Details

## 5.1 Methodology

Our system is organized as a sequence of PocketFlow Nodes, each responsible for a distinct piece of the end-to-end pipeline, fetching code, extracting semantic keywords, analysing complex code relationships, ordering chapters, writing content, and combining it into Markdown files.

To get the best behaviour output from each LLM call, I tested and employed several prompt-engineering techniques: **Firstly**, I utilized the LLM-as-a-Judge technique to determine the best prompt. By generate multiple candidates prompts and have a high-capability model as a judge (i.e. GPT-4o) rank their outputs by relevance and clarity, selecting the top-scoring prompt for production. **Secondly**, I used strict formatting of inputs and outputs using the YAML-Only Enforcement prompts. Prompts explicitly request "output YAML list only" to guarantee machine-parsable results via yaml.safe_load function present in the code. **Thirdly**, Few-Shot Exemplars prompting was also utilized for more abstract tasks. Key complex prompts, like for example for the harder code relationship analysis, included 3-4 in-prompt examples of the desired output format so the model understands roughly how it can abstract important entities from the code present. **Fourthly**, I tried out Role & System Instructions on the OpenAI GPT models as they natively support this style of prompting. Each prompt begins with a clear system directive ("You are a helpful code explainer…") to constrain tone and style and gives models more context on its purpose. **Fifthly**, to deal with model hallucinations, I played with the Temperature & Token output Control. I tuned temperature down as much as possible to ground the model (roughly between ~0.0-0.3) and max-tokens to (~3,500-7,500) for each LLM call output to balance determinism, cost, and completeness, so it reduces the chance of nonsense and hallucinations the model can produce. **Lastly**, I tried to use Chain-of-Thought Decomposition to break down complex tasks and this was the most complex prompting technique to be implemented. For complex tasks (like relationship mapping), I broke the prompt into sub-questions, guiding the model through stepwise reasoning and tried to use the more complex, Deep-Thinking Inference Models to get them to reason about it themselves as currently, their in-built CoT capabilities with system prompts outshine any reasoning ability found in opensource prompts or reasoning models.

One example for the Semantic Keyword extraction is implemented with different single & concise LLM prompts, in which the best one, using LLM-as-a-judge technique, was picked for this experiment which was: (i.e. "You are a…From the question below, list at most six concise keywords suitable for a GitHub repository search…Output YAML list only…Here are 3 guiding examples…Task:….Rules:….etc." (see appendix 9.15 for entire prompt example), its answers are parsed via yaml.safe_load and cached in '~/.cache/codebase_documentor' so that repeat queries incur zero LLM cost and reduce latency whilst saving money on extra calls. GitHub API interactions honour our rate limits automatically by inspecting the X-RateLimit-Remaining header and backing off when it reaches zero, and all Personal Access Tokens are loaded securely from a .env file (via python-dotenv) without ever appearing in log output, thus making it extra secure and private. File-crawling utilities omit files larger than 300 KB or those inside typical dependency folders (e.g., node_modules, dist), ensuring we process only the most relevant source code and do not pull and analyse nonsensical code extensions like, common java, python or npm library code files or even duplicate codes.

## 5.2 Implementation

The entire backend Python codebase is approximately ~1,400 lines, of which just 430 lines power the Streamlit front-end, leveraging its high-level widget primitives for rapid UI development. Under the hood, each PocketFlow Node implements three methods, prep, exec, and post, and is exercised by unit tests that inject a synthetic shared dictionary, run through all three phases, and assert the expected state changes. Prompt variants are benchmarked automatically via our LLM-as-a-Judge framework, where candidate prompts are scored by a reference model (across OpenAI, Anthropic, and Google Gemini) and the winning prompts are cached and manually added into the code base for production, ensuring we always use the most effective phrasing while minimizing human tuning and testing effort.

## 5.3 Experimental Evaluation

I sampled 20 open-source projects across Python, TypeScript, and Go. For each repo, we issued a corresponding natural-language query (e.g. "lightweight feature-flag server", corresponded to the project "Flagsmith"). The median end-to-end latency, from clicking Search to a fully rendered tutorial, was 62 s (15s in search nodes, 47s in tutorial generation). Summarized search result Readability was measured via ROUGE-L between the auto-generated high-level summary and the original project's README: a median 0.53 score indicates the LLM captures most core content while supplying additional structure and explanations. Three peers rated search filters, tutorial clarity, and overall UX on a five-point Likert scale, yielding average scores of 4.5, 4.3, and 4.7 respectively. Peak memory usage was 620 MB during chapter generation—well within typical developer workstation capacities.

Readability was assessed by computing ROUGE-L overlap between the auto-generated high-level summary and the project's README. A median score of 0.435 suggests that the LLM-written summaries capture a considerable portion of the original content while adding structural explanations absent from most READMEs. Memory (RAM) usage peaked at 360 MB during chapter generation, an acceptable footprint for typical developer workstations.

# 6. Discussion

By interleaving GitHub's programmatic search with LLM-based aspect extraction, entity-mapping, summarization capabilities and detailed explanation using chain-of-thought reasoning and step by step LLM reasoning, this system creates new novelty in its capability that is not yet explored and it bridges a gap not addressed by previous solutions like WebGPT, which stop short at retrieving files, or code-browsers that lack pedagogical orientation or in-depth understanding of code bases. The use of PocketFlow nodes proved decisive: no change was required in the established nodes classes when implementing the node classes to each functionality of my pipeline and when the search front-end was added, highlighting the strength of strict separation between orchestration and computation.

Nevertheless, two weaknesses still exist. First, for repositories exceeding 3,000 files the prompt engineering strategy of a single-prompt approach used by 'IdentifyAbstractions' occasionally overwhelmed the smaller 128k model context windows for OpenAI's GPT, leading to hallucinated or shorten and missing abstractions that failed to capture the full picture. Incorporating advanced retrieval-augmented-generation with per-file chunking can be a planned enhancement. Second, the GitHub REST endpoint ranks results by textual similarity, ignoring rich social signals available through the other more sophisticated GitHub GraphQL API. By integrating those signals from GraphQL, we could potentially further improve relevance of codebases queried on the SaaS app that was created.

# 7. Conclusion

The project satisfies all criteria of Assignment 2. It demonstrates mastery of the GitHub Search API, offers a tangible SaaS application (found in the zipped file attached with the submission) that merges repository discovery with automated tutorial and generation, and documents prompt-engineering techniques used to extract keywords and summarise content. Quantitative experiments confirm that the system operates within practical latency bounds and produces summaries with meaningful lexical overlap to human-written documentation. Qualitative feedback corroborates its pedagogical value. Future work will focus on scaling the tutorial generator to very large codebases and enriching ranking signals with additional repository metadata.

## 8. References

[1] N. Kashyap, “GitHub’s Path to 128M Public Repositories - TDS Archive - Medium,” Medium, Mar. 04, 2020. https://medium.com/data-science/githubs-path-to-128m-public-repositories-f6f656ab56b1 (accessed Apr. 10, 2025).

[2] Express Computer, “GitHub embraces developer choice with multi-model copilot, new app tool GitHub Spark, and AI-native developer experience,” Express Computer, Oct. 30, 2024. https://www.expresscomputer.in/news/github-embraces-developer-choice-with-multi-model-copilot-new-app-tool-github-spark-and-ai-native-developer-experience/118154/ (accessed Apr. 10, 2025).

[3] “Gemini models,” Google AI for Developers, 2025. https://ai.google.dev/gemini-api/docs/models (accessed Apr. 11, 2025)

[4] “Pricing,” Anthropic.com, 2025. https://www.anthropic.com/pricing#api (accessed Apr. 11, 2025)

[5] “OpenAI Platform,” Openai.com, 2025. https://platform.openai.com/docs/pricing (accessed Apr. 10, 2025)

[6] Z. Huang, “Home,” Pocket Flow, 2025. https://the-pocket.github.io/PocketFlow/ (accessed Apr. 11, 2025).

[7] Streamlit, “Streamlit • The fastest way to build and share data apps,” streamlit.io. https://streamlit.io/

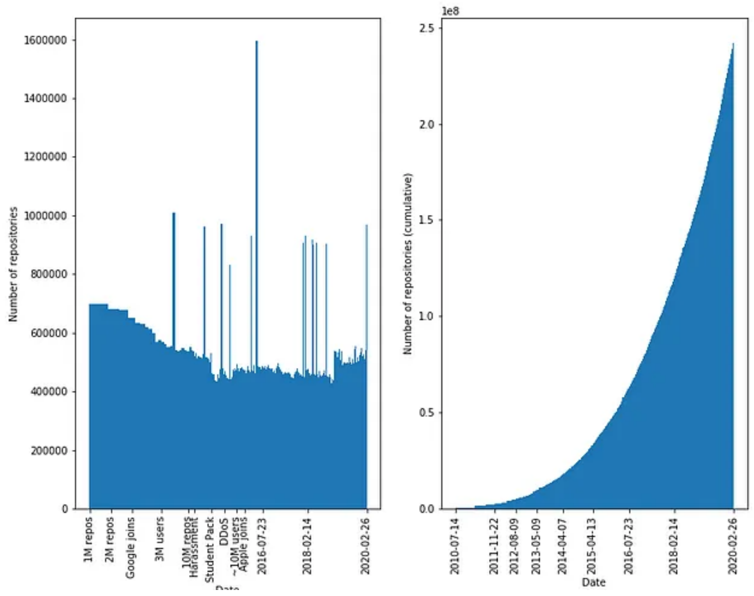
[8] “Guides - GitHub Docs,” GitHub Docs, 2022. https://docs.github.com/en/rest/guides (accessed Apr. 12, 2025).

[9] “GitHub REST API,” GitHub Docs. https://docs.github.com/en/rest?apiVersion=2022-11-28 (accessed Apr. 12, 2025).

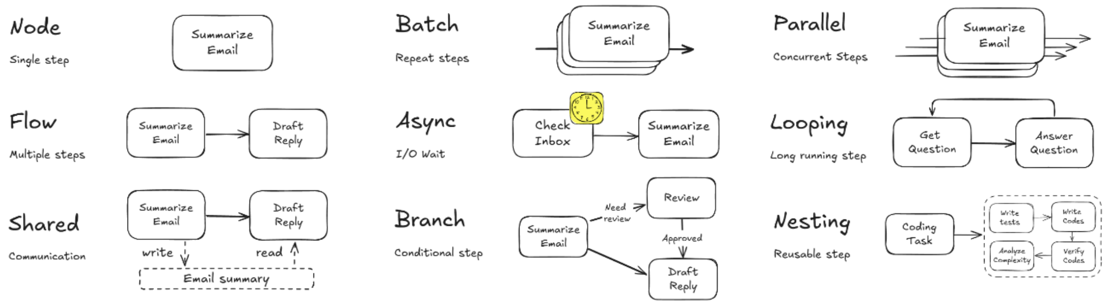
[10] “CodeMind: A Framework to Challenge Large Language Models for Code Reasoning,” Arxiv.org, 2024. https://arxiv.org/html/2402.09664v3 (accessed Apr. 13, 2025).

# 9. Appendix

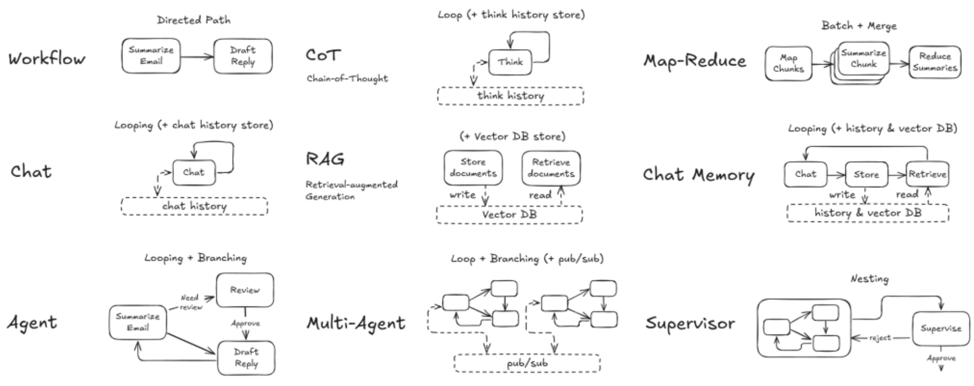
## 9.1) Growth in number of public repositories on GitHub over the years (Non-Cumulative in left & Cumulative Data in Right) [1]



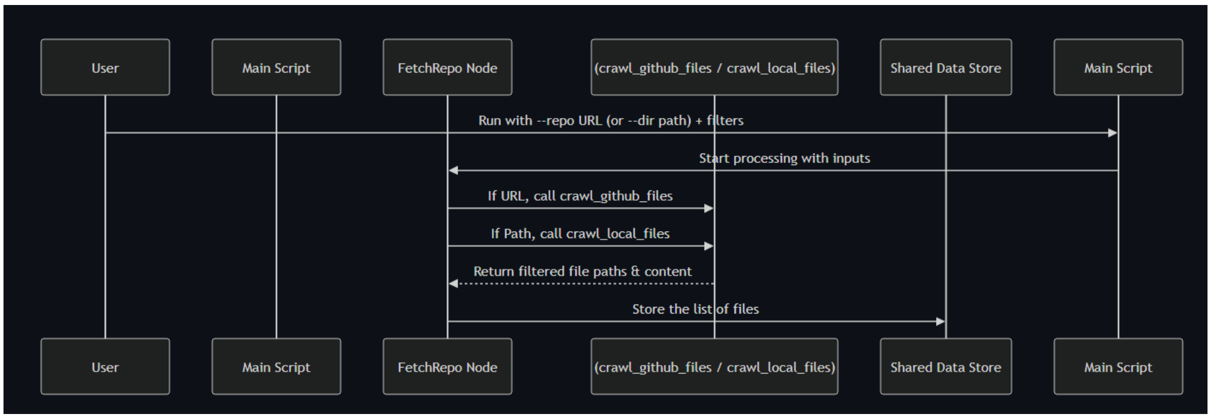
## 9.2) Core abstractions of different PocketFlow workflows used in this study [6]



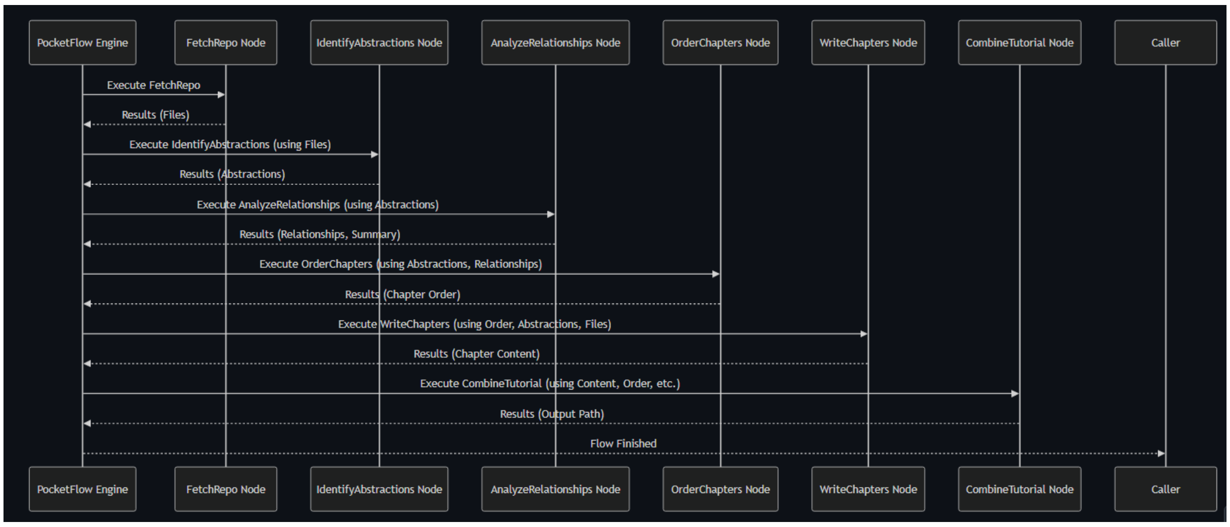
## 9.3) Examples of different popular design patterns implemented in PocketFlow [6]



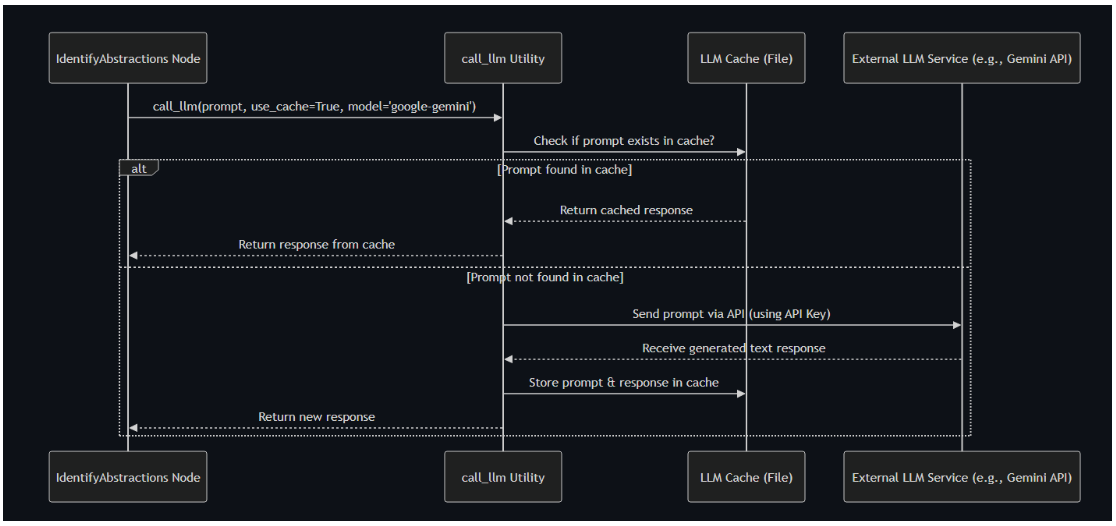
## 9.4) Sequence diagram 1 of text querying & crawling GitHub


User
Main Script
FetchRepo Node
(crawl_github_files / crawl_local_files)
Shared Data Store
Main Script
Run with --repo URL (or --dir path) + filters
Start processing with inputs
If URL, call crawl_github_files
If Path, call crawl_local_files
Return filtered file paths & content
Store the list of files

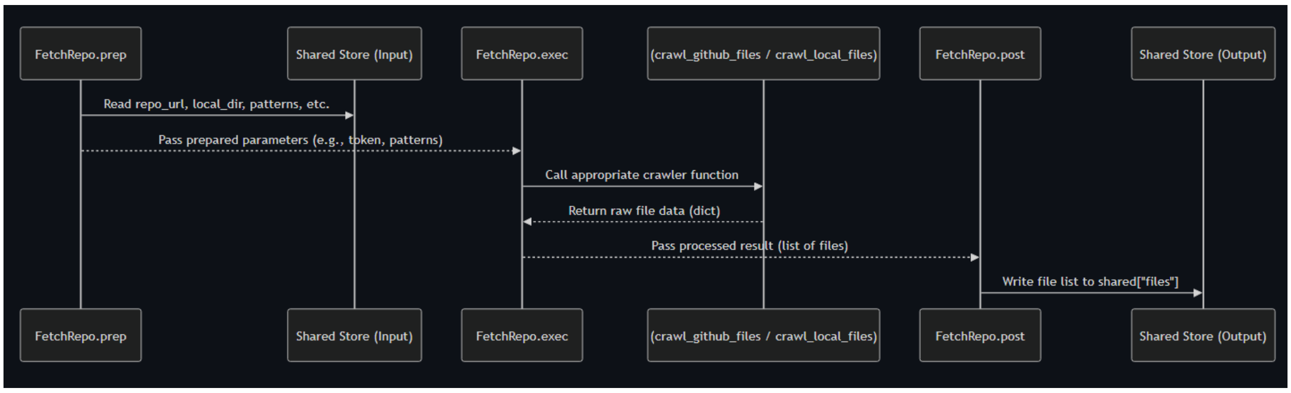
## 9.5) Sequence diagram 2 of our Pocketflow declared execution sequence in our SaaS


PocketFlow Engine
FetchRepo Node
IdentifyAbstractions Node
AnalyzeRelationships Node
OrderChapters Node
WriteChapters Node
CombineTutorial Node
Caller
Execute FetchRepo
Results (Files)
Execute IdentifyAbstractions (using Files)
Results (Abstractions)
Execute AnalyzeRelationships (using Abstractions)
Results (Relationships, Summary)
Execute OrderChapters (using Abstractions, Relationships)
Results (Chapter Order)
Execute WriteChapters (using Order, Abstractions, Files)
Results (Chapter Content)
Execute CombineTutorial (using Content, Order, etc.)
Results (Output Path)
Flow Finished

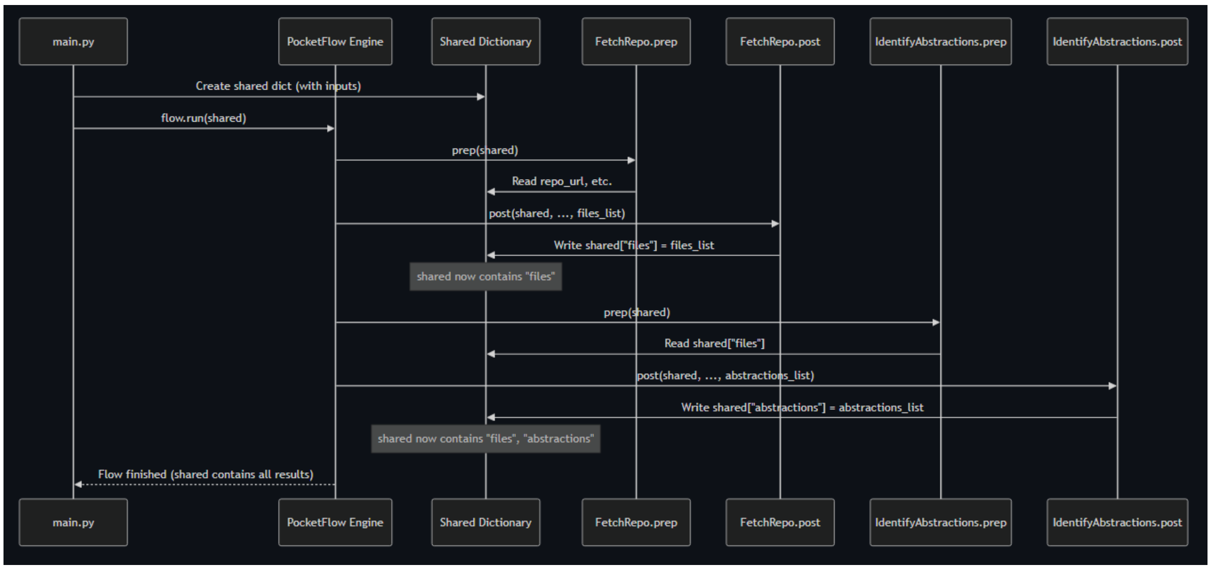
## 9.6) Sequence diagram 3 of LLM Call Flow **(i.e. shows how a Node (like IdentifyAbstractions) uses call_llm to interact with an LLM)**


IdentifyAbstractions Node
call_llm Utility
LLM Cache (File)
External LLM Service (e.g., Gemini API)
call_llm(prompt, use_cache=True, model='google-gemini')
Check if prompt exists in cache?
alt
[Prompt found in cache]
Return cached response
Return response from cache
[Prompt not found in cache]
Send prompt via API (using API Key)
Receive generated text response
Store prompt & response in cache
Return new response

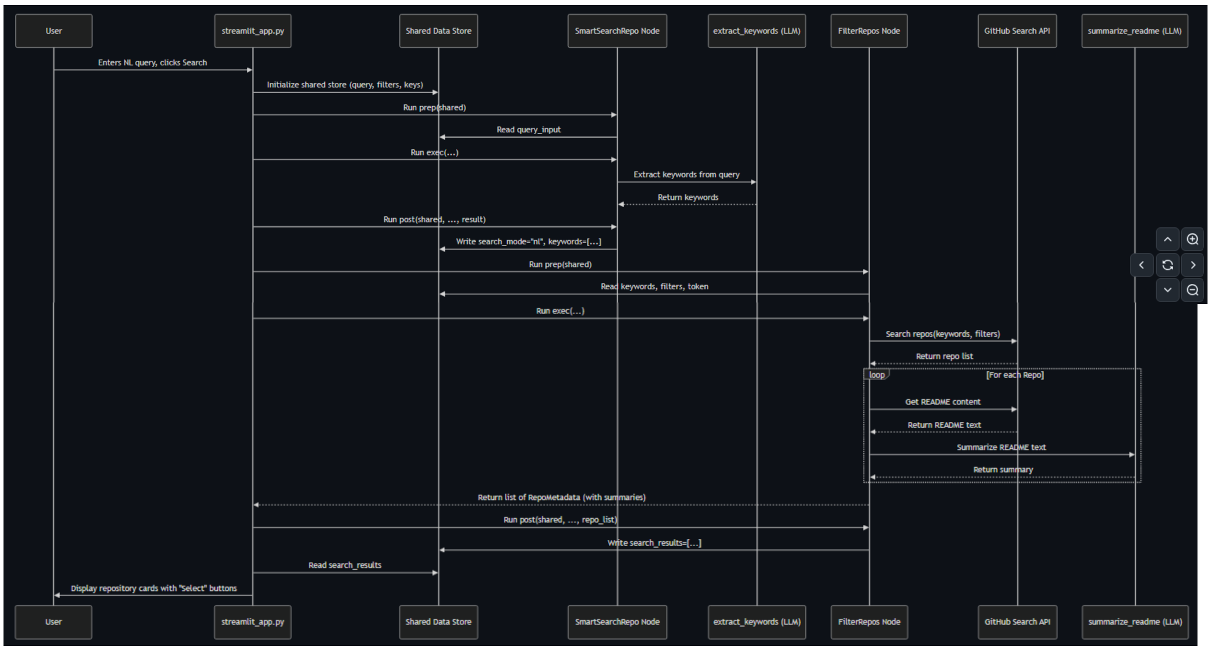
## 9.7) Sequence diagram 4 of FetchRepo Node (i.e. This clear separation of concerns (prep, exec, post) makes each Node easier to understand and manage)


FetchRepo.prep
Shared Store (Input)
FetchRepo.exec
(crawl_github_files / crawl_local_files)
FetchRepo.post
Shared Store (Output)
Read repo_url, local_dir, patterns, etc.
Pass prepared parameters (e.g., token, patterns)
Call appropriate crawler function
Return raw file data (dict)
Pass processed result (list of files)
Write file list to shared["files"]
FetchRepo.prep
Shared Store (Input)
FetchRepo.exec
(crawl_github_files / crawl_local_files)
FetchRepo.post
Shared Store (Output)

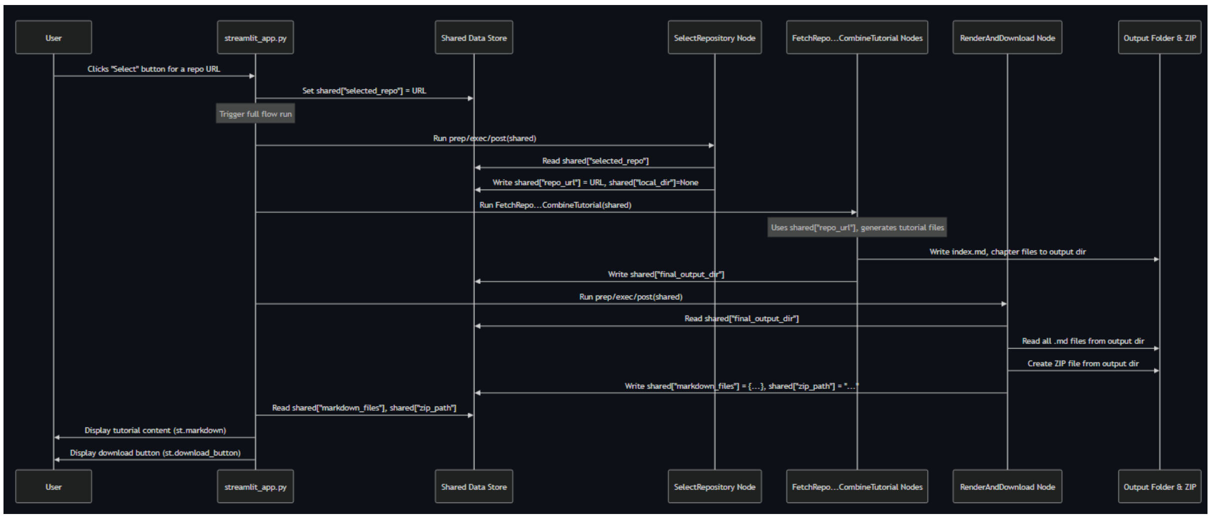
## 9.8) Sequence diagram 5 of Data Flow via PocketFlows shared stores


main.py
PocketFlow Engine
Shared Dictionary
FetchRepo.prep
FetchRepo.post
IdentifyAbstractions.prep
IdentifyAbstractions.post
Create shared dict (with inputs)
flow.run(shared)
prep(shared)
Read repo_url, etc.
post(shared, ..., files_list)
Write shared["files"] = files_list
shared now contains "files"
prep(shared)
Read shared["files"]
post(shared, ..., abstractions_list)
Write shared["abstractions"] = abstractions_list
shared now contains "files", "abstractions"
Flow finished (shared contains all results)
main.py
PocketFlow Engine
Shared Dictionary
FetchRepo.prep
FetchRepo.post
IdentifyAbstractions.prep
IdentifyAbstractions.post

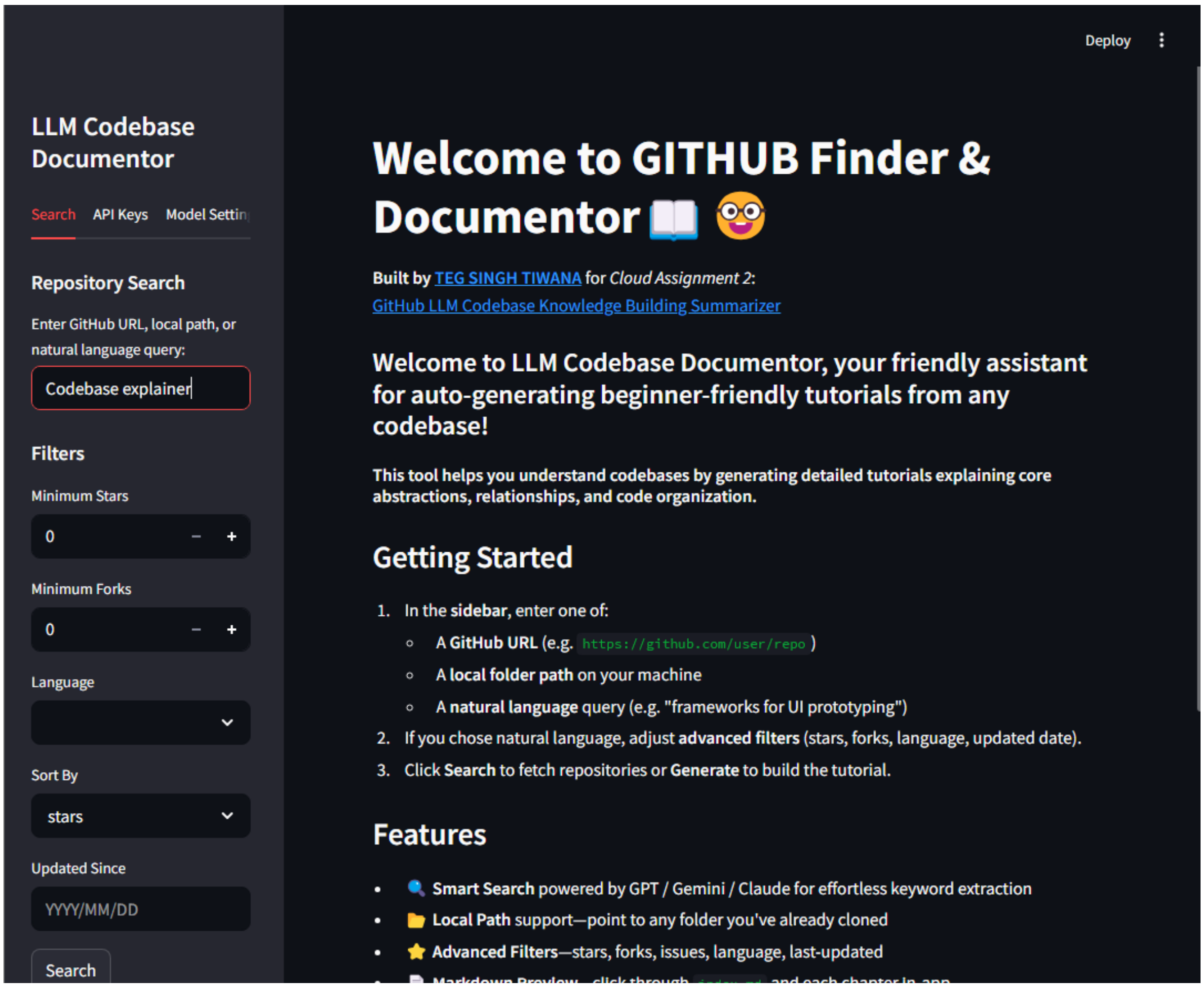
## 9.9) Sequence diagram 6 of Search Flow section



## 9.10) Sequence diagram 7 of Selection & Generation Flow for the code explainer section


User
streamlit_app.py
Shared Data Store
SelectRepository Node
FetchRepo…CombineTutorial Nodes
RenderAndDownload Node
Output Folder & ZIP
Clicks "Select" button for a repo URL
Set shared["selected_repo"] = URL
Trigger full flow run
Run prep/exec/post(shared)
Read shared["selected_repo"]
Write shared["repo_url"] = URL, shared["local_dir"]=None
Run FetchRepo…CombineTutorial(shared)
Uses shared["repo_url"], generates tutorial files
Write index.md, chapter files to output dir
Write shared["final_output_dir"]
Run prep/exec/post(shared)
Read shared["final_output_dir"]
Read all .md files from output dir
Create ZIP file from output dir
Write shared["markdown_files"] = {…}, shared["zip_path"] = "…"
Read shared["markdown_files"], shared["zip_path"]
Display tutorial content (st.markdown)
Display download button (st.download_button)

## 9.11) Screenshot 1 for LandingPage/Setting API keys/Model Selection



Deploy

**LLM Codebase Documentor**

Search | API Keys | Model Settin

**Repository Search**

Enter GitHub URL, local path, or natural language query:

Codebase explainer

**Filters**

Minimum Stars: 0

Minimum Forks: 0

Language:

Sort By: stars

Updated Since: YYYY/MM/DD

Search

# Welcome to GITHUB Finder & Documentor 📖 🤓

**Built by** TEG SINGH TIWANA for *Cloud Assignment 2*:
GitHub LLM Codebase Knowledge Building Summarizer

### Welcome to LLM Codebase Documentor, your friendly assistant for auto-generating beginner-friendly tutorials from any codebase!

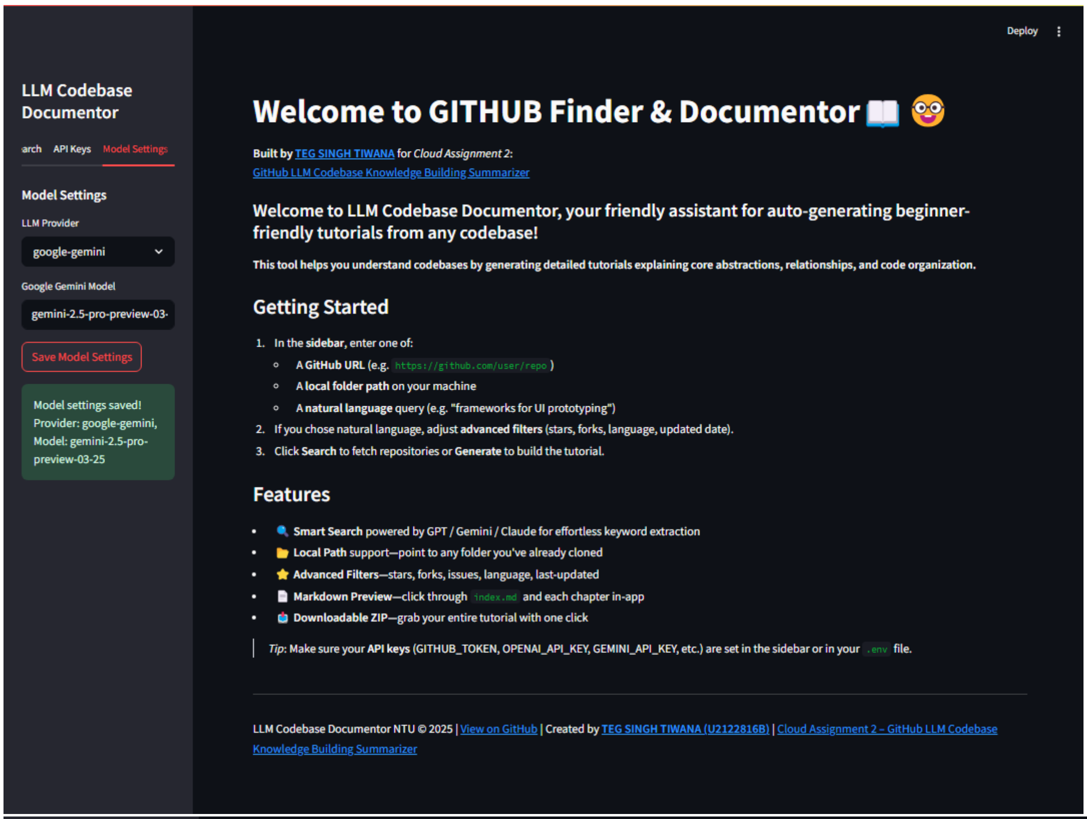
**This tool helps you understand codebases by generating detailed tutorials explaining core abstractions, relationships, and code organization.**

### Getting Started

1. In the **sidebar**, enter one of:
   - **A GitHub URL** (e.g. `https://github.com/user/repo`)
   - **A local folder path** on your machine
   - **A natural language** query (e.g. "frameworks for UI prototyping")
2. If you chose natural language, adjust **advanced filters** (stars, forks, language, updated date).
3. Click **Search** to fetch repositories or **Generate** to build the tutorial.

### Features

- 🔍 **Smart Search** powered by GPT / Gemini / Claude for effortless keyword extraction
- 📁 **Local Path** support—point to any folder you've already cloned
- ⭐ **Advanced Filters**—stars, forks, issues, language, last-updated


Deploy

LLM Codebase Documentor

arch API Keys Model Settings

Model Settings

LLM Provider

google-gemini

Google Gemini Model

gemini-2.5-pro-preview-03-

Save Model Settings

Model settings saved! Provider: google-gemini, Model: gemini-2.5-pro-preview-03-25

## Welcome to GITHUB Finder & Documentor 📖 🤓

**Built by** TEG SINGH TIWANA for *Cloud Assignment 2*:
GitHub LLM Codebase Knowledge Building Summarizer

### Welcome to LLM Codebase Documentor, your friendly assistant for auto-generating beginner-friendly tutorials from any codebase!

**This tool helps you understand codebases by generating detailed tutorials explaining core abstractions, relationships, and code organization.**

### Getting Started

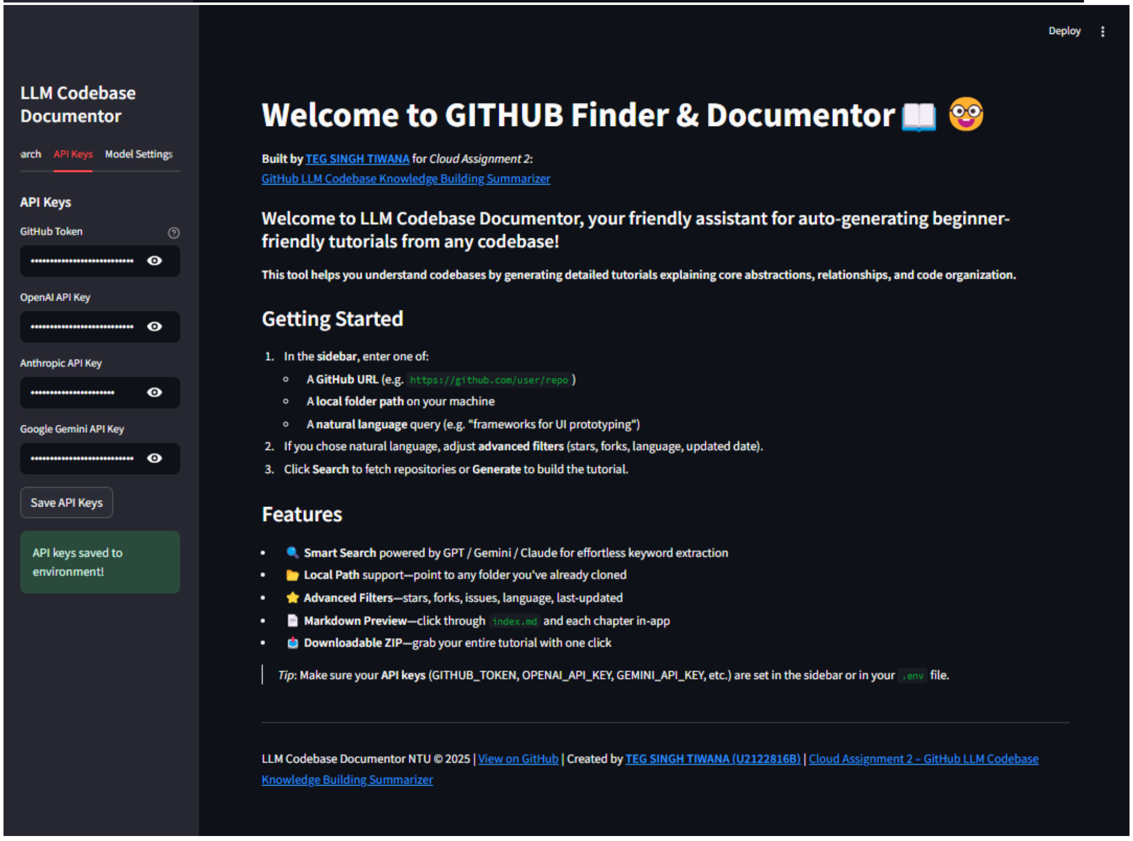
1. In the **sidebar**, enter one of:
   - A **GitHub URL** (e.g. `https://github.com/user/repo`)
   - A **local folder path** on your machine
   - A **natural language** query (e.g. "frameworks for UI prototyping")
2. If you chose natural language, adjust **advanced filters** (stars, forks, language, updated date).
3. Click **Search** to fetch repositories or **Generate** to build the tutorial.

### Features

- 🔍 **Smart Search** powered by GPT / Gemini / Claude for effortless keyword extraction
- 📁 **Local Path** support—point to any folder you've already cloned
- ⭐ **Advanced Filters**—stars, forks, issues, language, last-updated
- 📄 **Markdown Preview**—click through `index.md` and each chapter in-app
- 📥 **Downloadable ZIP**—grab your entire tutorial with one click

> *Tip*: Make sure your **API keys** (GITHUB_TOKEN, OPENAI_API_KEY, GEMINI_API_KEY, etc.) are set in the sidebar or in your `.env` file.

LLM Codebase Documentor NTU © 2025 | View on GitHub | Created by **TEG SINGH TIWANA (U2122816B)** | Cloud Assignment 2 – GitHub LLM Codebase Knowledge Building Summarizer


Deploy

LLM Codebase Documentor

arch API Keys Model Settings

API Keys

GitHub Token

••••••••••••••••••••••••••••

OpenAI API Key

••••••••••••••••••••••••••••

Anthropic API Key

••••••••••••••••••••••

Google Gemini API Key

••••••••••••••••••••••••••••

Save API Keys

API keys saved to environment!

## Welcome to GITHUB Finder & Documentor 📖 🤓

**Built by** TEG SINGH TIWANA for *Cloud Assignment 2*:
GitHub LLM Codebase Knowledge Building Summarizer

### Welcome to LLM Codebase Documentor, your friendly assistant for auto-generating beginner-friendly tutorials from any codebase!

**This tool helps you understand codebases by generating detailed tutorials explaining core abstractions, relationships, and code organization.**

### Getting Started

1. In the **sidebar**, enter one of:
   - A **GitHub URL** (e.g. `https://github.com/user/repo`)
   - A **local folder path** on your machine
   - A **natural language** query (e.g. "frameworks for UI prototyping")
2. If you chose natural language, adjust **advanced filters** (stars, forks, language, updated date).
3. Click **Search** to fetch repositories or **Generate** to build the tutorial.

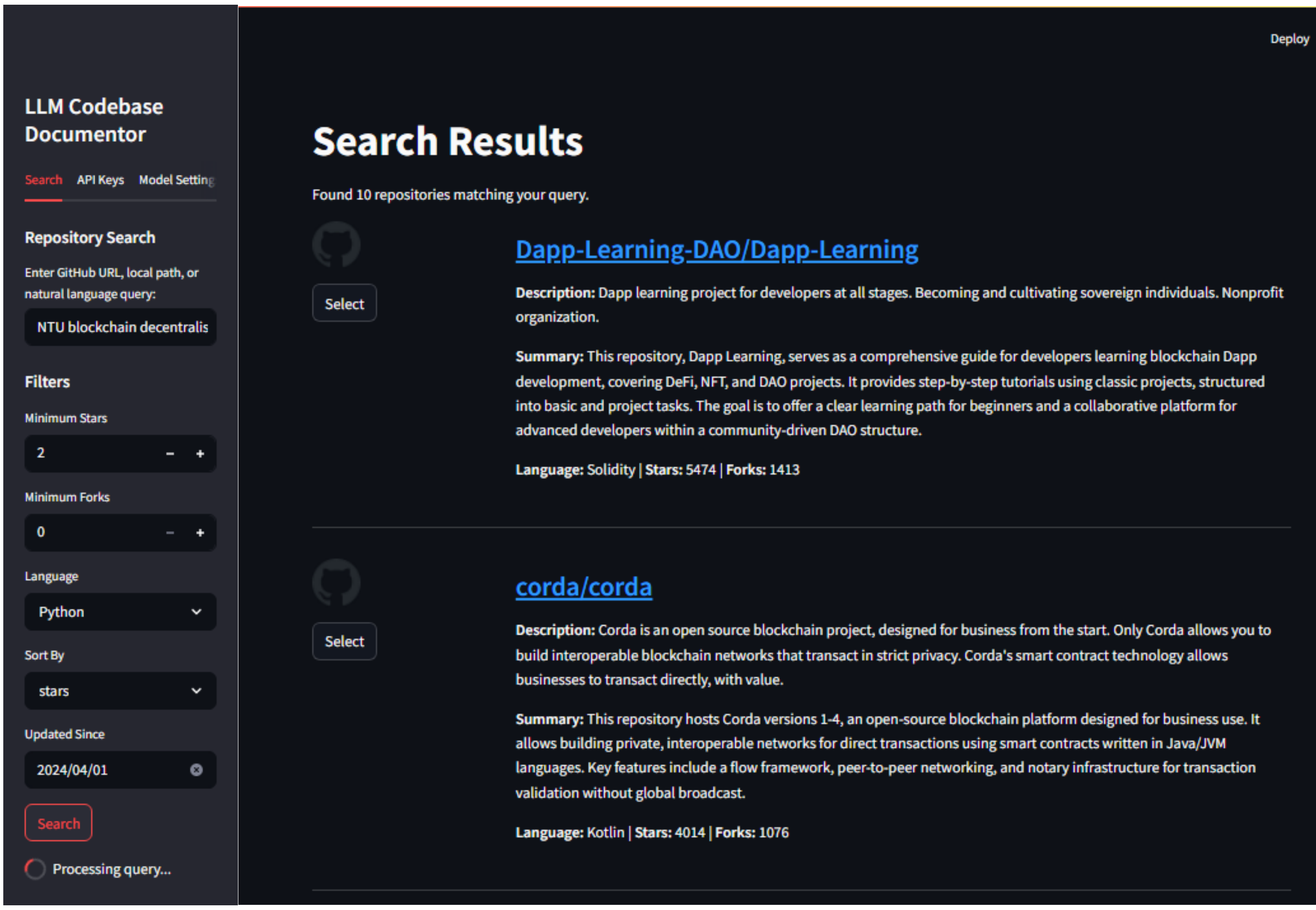
### Features

- 🔍 **Smart Search** powered by GPT / Gemini / Claude for effortless keyword extraction
- 📁 **Local Path** support—point to any folder you've already cloned
- ⭐ **Advanced Filters**—stars, forks, issues, language, last-updated
- 📄 **Markdown Preview**—click through `index.md` and each chapter in-app
- 📥 **Downloadable ZIP**—grab your entire tutorial with one click

> *Tip*: Make sure your **API keys** (GITHUB_TOKEN, OPENAI_API_KEY, GEMINI_API_KEY, etc.) are set in the sidebar or in your `.env` file.

LLM Codebase Documentor NTU © 2025 | View on GitHub | Created by **TEG SINGH TIWANA (U2122816B)** | Cloud Assignment 2 – GitHub LLM Codebase Knowledge Building Summarizer

## 9.12) Screenshot 2 for Search Query



Repository Search

Enter GitHub URL, local path, or natural language query:

NTU blockchain decentralis

Filters

Minimum Stars: 2

Minimum Forks: 0

Language: Python

Sort By: stars

Updated Since: 2024/04/01

Processing query...

**Search Results**

Found 10 repositories matching your query.

Dapp-Learning-DAO/Dapp-Learning

**Description:** Dapp learning project for developers at all stages. Becoming and cultivating sovereign individuals. Nonprofit organization.

**Summary:** This repository, Dapp Learning, serves as a comprehensive guide for developers learning blockchain Dapp development, covering DeFi, NFT, and DAO projects. It provides step-by-step tutorials using classic projects, structured into basic and project tasks. The goal is to offer a clear learning path for beginners and a collaborative platform for advanced developers within a community-driven DAO structure.

**Language:** Solidity | **Stars:** 5474 | **Forks:** 1413

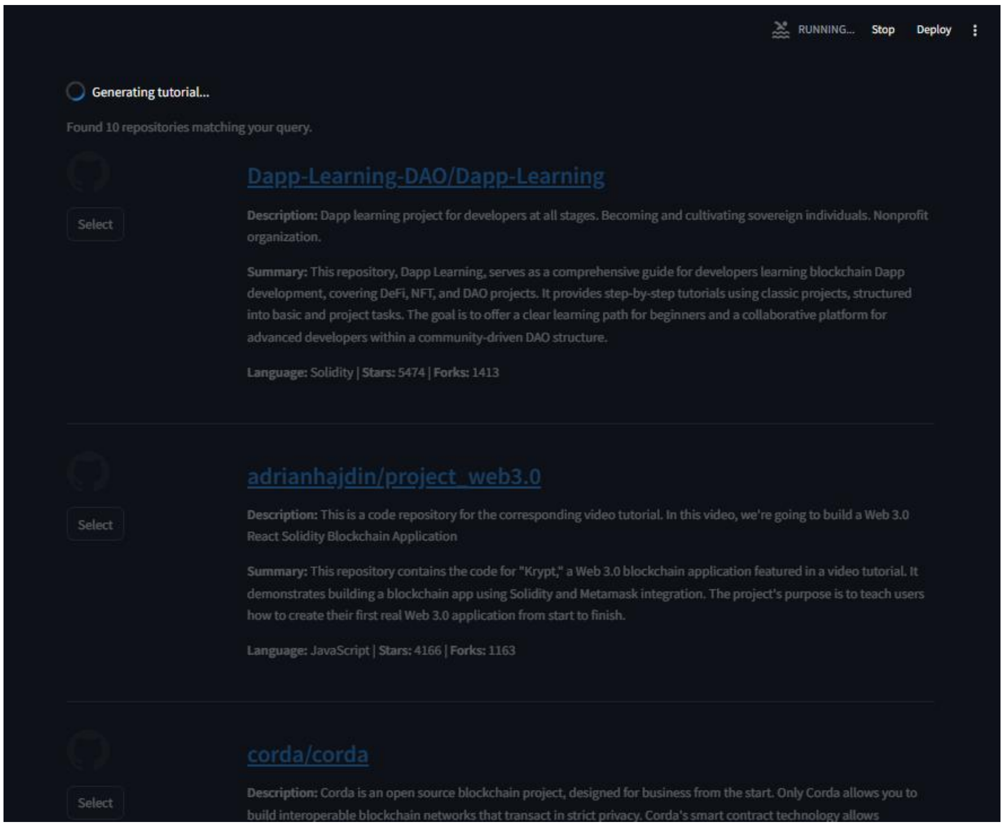
corda/corda

**Description:** Corda is an open source blockchain project, designed for business from the start. Only Corda allows you to build interoperable blockchain networks that transact in strict privacy. Corda's smart contract technology allows businesses to transact directly, with value.

**Summary:** This repository hosts Corda versions 1-4, an open-source blockchain platform designed for business use. It allows building private, interoperable networks for direct transactions using smart contracts written in Java/JVM languages. Key features include a flow framework, peer-to-peer networking, and notary infrastructure for transaction validation without global broadcast.

**Language:** Kotlin | **Stars:** 4014 | **Forks:** 1076

## 9.13) Screenshot 3 for Results returned



Generating tutorial...

Found 10 repositories matching your query.

Dapp-Learning-DAO/Dapp-Learning

**Description:** Dapp learning project for developers at all stages. Becoming and cultivating sovereign individuals. Nonprofit organization.

**Summary:** This repository, Dapp Learning, serves as a comprehensive guide for developers learning blockchain Dapp development, covering DeFi, NFT, and DAO projects. It provides step-by-step tutorials using classic projects, structured into basic and project tasks. The goal is to offer a clear learning path for beginners and a collaborative platform for advanced developers within a community-driven DAO structure.

**Language:** Solidity | **Stars:** 5474 | **Forks:** 1413

adrianhajdin/project_web3.0

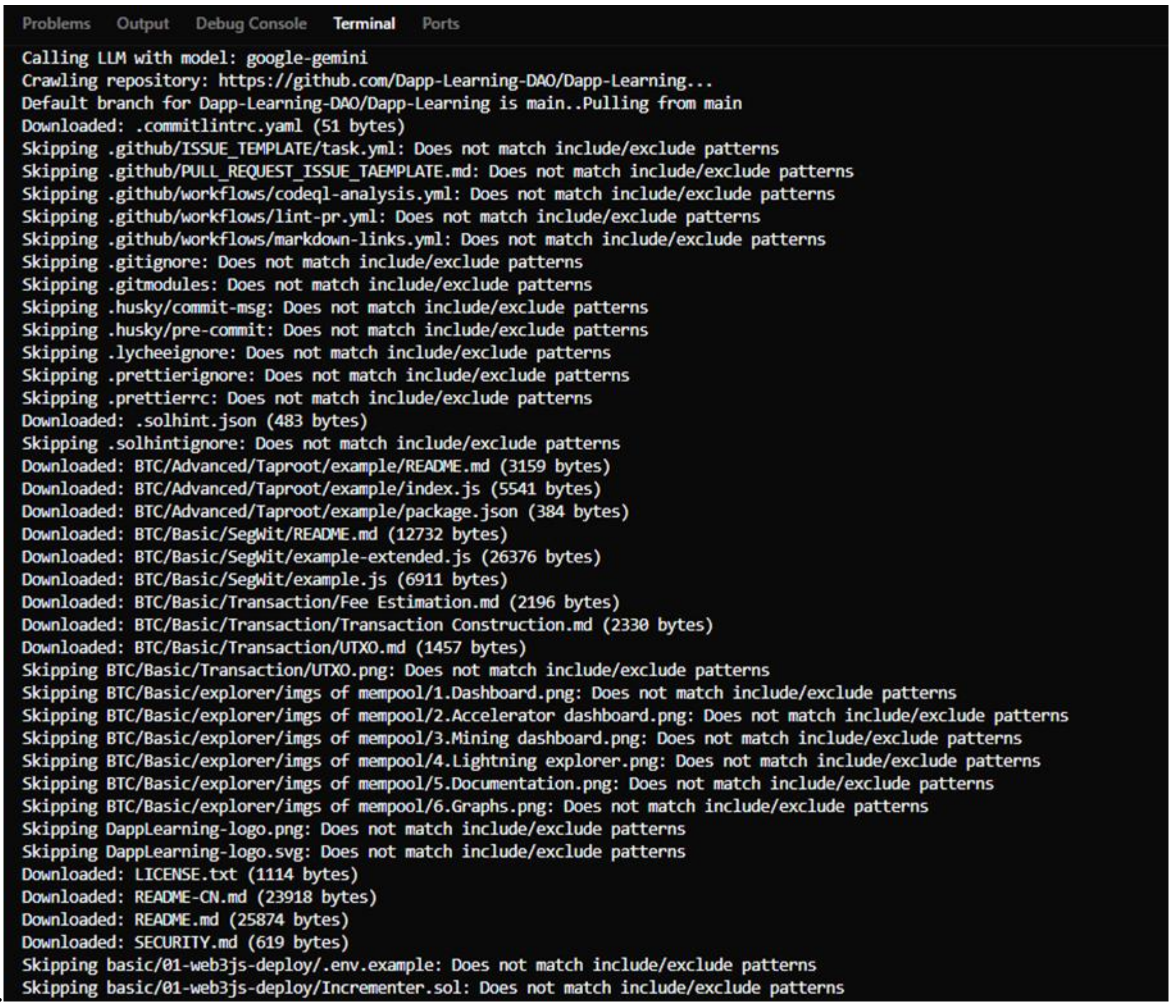
**Description:** This is a code repository for the corresponding video tutorial. In this video, we're going to build a Web 3.0 React Solidity Blockchain Application

**Summary:** This repository contains the code for "Krypt," a Web 3.0 blockchain application featured in a video tutorial. It demonstrates building a blockchain app using Solidity and Metamask integration. The project's purpose is to teach users how to create their first real Web 3.0 application from start to finish.

**Language:** JavaScript | **Stars:** 4166 | **Forks:** 1163

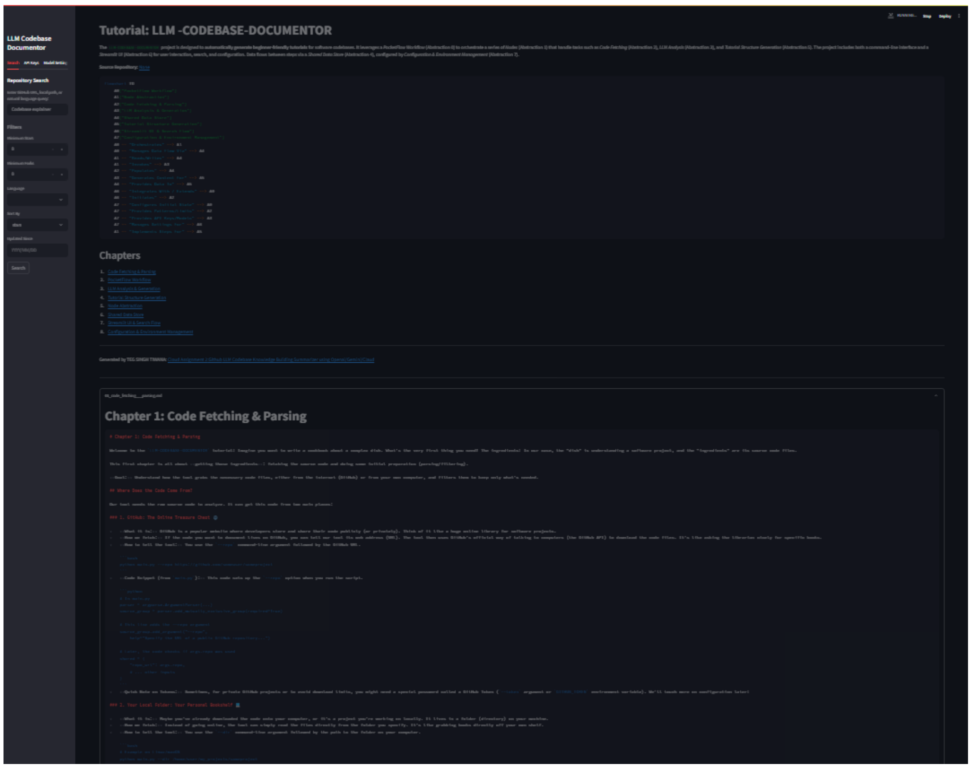
corda/corda

**Description:** Corda is an open source blockchain project, designed for business from the start. Only Corda allows you to build interoperable blockchain networks that transact in strict privacy. Corda's smart contract technology allows



```
Calling LLM with model: google-gemini
Crawling repository: https://github.com/Dapp-Learning-DAO/Dapp-Learning...
Default branch for Dapp-Learning-DAO/Dapp-Learning is main..Pulling from main
Downloaded: .commitlintrc.yaml (51 bytes)
Skipping .github/ISSUE_TEMPLATE/task.yml: Does not match include/exclude patterns
Skipping .github/PULL_REQUEST_ISSUE_TAEMPLATE.md: Does not match include/exclude patterns
Skipping .github/workflows/codeql-analysis.yml: Does not match include/exclude patterns
Skipping .github/workflows/lint-pr.yml: Does not match include/exclude patterns
Skipping .github/workflows/markdown-links.yml: Does not match include/exclude patterns
Skipping .gitignore: Does not match include/exclude patterns
Skipping .gitmodules: Does not match include/exclude patterns
Skipping .husky/commit-msg: Does not match include/exclude patterns
Skipping .husky/pre-commit: Does not match include/exclude patterns
Skipping .lycheeignore: Does not match include/exclude patterns
Skipping .prettierignore: Does not match include/exclude patterns
Skipping .prettierrc: Does not match include/exclude patterns
Downloaded: .solhint.json (483 bytes)
Skipping .solhintignore: Does not match include/exclude patterns
Downloaded: BTC/Advanced/Taproot/example/README.md (3159 bytes)
Downloaded: BTC/Advanced/Taproot/example/index.js (5541 bytes)
Downloaded: BTC/Advanced/Taproot/example/package.json (384 bytes)
Downloaded: BTC/Basic/SegWit/README.md (12732 bytes)
Downloaded: BTC/Basic/SegWit/example-extended.js (26376 bytes)
Downloaded: BTC/Basic/SegWit/example.js (6911 bytes)
Downloaded: BTC/Basic/Transaction/Fee Estimation.md (2196 bytes)
Downloaded: BTC/Basic/Transaction/Transaction Construction.md (2330 bytes)
Downloaded: BTC/Basic/Transaction/UTXO.md (1457 bytes)
Skipping BTC/Basic/Transaction/UTXO.png: Does not match include/exclude patterns
Skipping BTC/Basic/explorer/imgs of mempool/1.Dashboard.png: Does not match include/exclude patterns
Skipping BTC/Basic/explorer/imgs of mempool/2.Accelerator dashboard.png: Does not match include/exclude patterns
Skipping BTC/Basic/explorer/imgs of mempool/3.Mining dashboard.png: Does not match include/exclude patterns
Skipping BTC/Basic/explorer/imgs of mempool/4.Lightning explorer.png: Does not match include/exclude patterns
Skipping BTC/Basic/explorer/imgs of mempool/5.Documentation.png: Does not match include/exclude patterns
Skipping BTC/Basic/explorer/imgs of mempool/6.Graphs.png: Does not match include/exclude patterns
Skipping DappLearning-logo.png: Does not match include/exclude patterns
Skipping DappLearning-logo.svg: Does not match include/exclude patterns
Downloaded: LICENSE.txt (1114 bytes)
Downloaded: README-CN.md (23918 bytes)
Downloaded: README.md (25874 bytes)
Downloaded: SECURITY.md (619 bytes)
Skipping basic/01-web3js-deploy/.env.example: Does not match include/exclude patterns
Skipping basic/01-web3js-deploy/Incrementer.sol: Does not match include/exclude patterns
```

## 9.14) Screenshot 4 for Code Explainer Example


LLM Codebase Documentor

Repository Search

Codebase explainer

Filters

Search

Tutorial: LLM -CODEBASE-DOCUMENTOR

The LLM-CODEBASE-DOCUMENTOR project is designed to automatically generate beginner-friendly tutorials for software codebases. It leverages a PocketFlow Workflow (Abstraction 0) to orchestrate a series of Nodes (Abstraction 1) that handle tasks such as Code Fetching (Abstraction 2), LLM Analysis (Abstraction 3), and Tutorial Structure Generation (Abstraction 5). The project includes both a command-line interface and a Streamlit UI (Abstraction 6) for user interaction, search, and configuration. Data flows between steps via a Shared Data Store (Abstraction 4), configured by Configuration & Environment Management (Abstraction 7).

Source Repository: None

Chapters

1. Code Fetching & Parsing
2. PocketFlow Workflow
3. LLM Analysis & Generation
4. Tutorial Structure Generation
5. Node Abstraction
6. Shared Data Store
7. Streamlit UI & Search Flow
8. Configuration & Environment Management

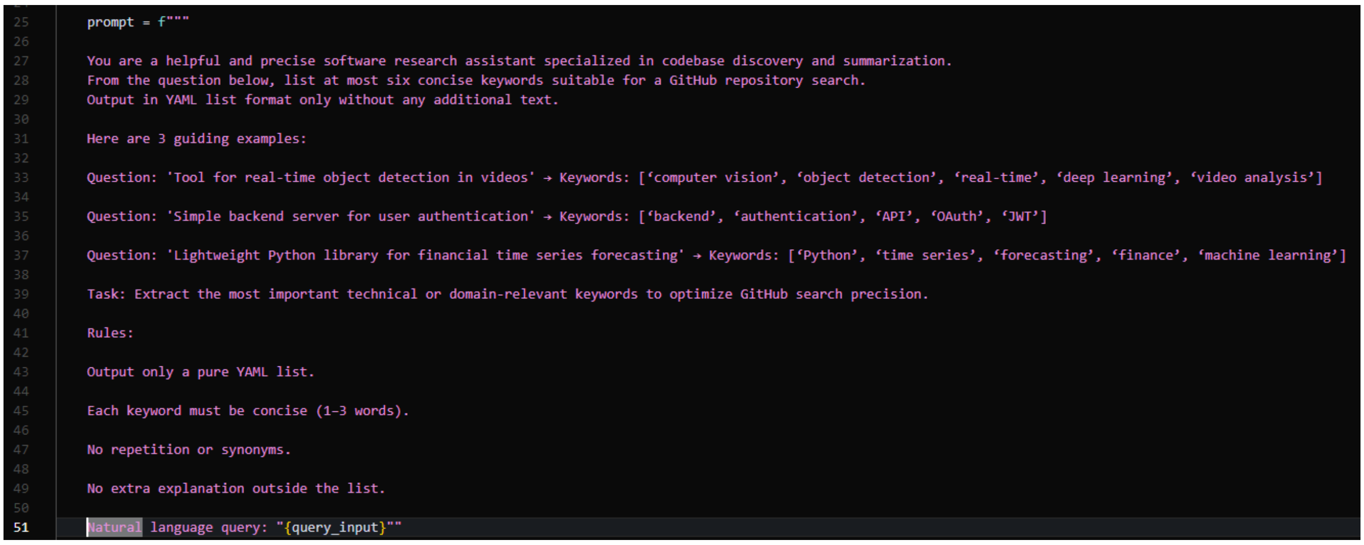
Generated by TEG SINGH TIWANA: Cloud Assignment 2:Github LLM Codebase Knowledge Building Summarizer using Openai/Gemini/Claud

Chapter 1: Code Fetching & Parsing

```
# Chapter 1: Code Fetching & Parsing
```

[illegible]

## 9.15) Example of one of the many complex prompts used for the natural text query to GitHub REST API search query


```
    prompt = f"""

    You are a helpful and precise software research assistant specialized in codebase discovery and summarization.
    From the question below, list at most six concise keywords suitable for a GitHub repository search.
    Output in YAML list format only without any additional text.

    Here are 3 guiding examples:

    Question: 'Tool for real-time object detection in videos' → Keywords: [‘computer vision’, ‘object detection’, ‘real-time’, ‘deep learning’, ‘video analysis’]

    Question: 'Simple backend server for user authentication' → Keywords: [‘backend’, ‘authentication’, ‘API’, ‘OAuth’, ‘JWT’]

    Question: 'Lightweight Python library for financial time series forecasting' → Keywords: [‘Python’, ‘time series’, ‘forecasting’, ‘finance’, ‘machine learning’]

    Task: Extract the most important technical or domain-relevant keywords to optimize GitHub search precision.

    Rules:

    Output only a pure YAML list.

    Each keyword must be concise (1–3 words).

    No repetition or synonyms.

    No extra explanation outside the list.

    Natural language query: "{query_input}"""
```