Purpose:
The purpose of this document is to store our relevant findings when looking into our open questions surrounding our current implementation of storing ingested files in Chroma DD. More specifically, the goal is to understand if our current approach of using a **single Chroma collection** for a particular project's **Documentation**, and a **single Chroma collection** for a particular project's **Code** is sufficient for our use case.

Current State
In the current state of our project, we will currently create only two **Chroma Collections per Project,** one for Documentation, and one for code. We did this approach mainly because we know we will be ingesting two types of files in particular for our project: 1) documentation (i.e design docs, local setup guides, how-to's, standards, etc), and 2) code (i.e python, Java, config files, etc).

Concerns
There are multiple points of concerns that I want to address in this research

1. Ability (and feasibility) to "intelligently" select which Chroma collection a particular question belongs to
    a. Right now, a particular Chroma Collection in the context of our service represents ***only files relevant to a particular Project*** (which is essentially any file corresponding to a **DataSource** a particular project is "linked" to)
    b. In the long run, the hope is to allow for questions/conversations to be **isolated** for a particular Project, while also supporting **multi-project capabilities** (i.e a question can take files from **two separate Projects** and use this in supplementation of the answer)
    c. In order to do this, there will need to be a **system** in place that **a)** takes in the users question, **b)** determines what project(s) this question applies to, **c)** utilizes the data from these collections in support of its answer
2. Embedding the user prompted question
    a. Given the ability for specific Embedding Models to be configured for a particular Project, this means there is essentially an "infinite" number of Embeddings that a particular collection must be using
    b. In order to use the "embedding-based retrieval", we will need to embed the text passed in by query prior to "querying" a particular collection (**NOTE:** We should determine if this is done by Chroma/LlamaIndex for us)
    c. This brings up concerns when a question is touching "multiple" chroma collections, as the there really will be no "intersection of understanding" between these Chroma collections
3. The number of Chroma collections to use and how to split up documents/code
    a. As mentioned, we only have **two Chroma collections per Project,** one for documentation, and one for code

    b. This was done primarily due to the want to use a specific embedding model when dealing with code, versus a different embedding model when working with documentation
    c. The concern is that this can get **extremely cluttered** when dealing with projects with **many data sources linked** and these data sources having **thousands of files associated.** We should determine if this general architecture is sound or if we should instead use more granular collections
4. Usage of simple embedding-based retrieval
    a. The concern here is that we need a more “elegant” way to perform RAG capabilities rather than simple embedding-based retrieval
    b. The reason for this is that we may find “similar” documents to the users question, but not actually find the answer to the problem in those retrieved documents
5. Embedding Model Context Window Length
    a. The concern here is that we may need to configure our “chunking” capabilities to account for the current used context-window length
    b. We should verify we do this already for the **Documentation Ingestion** and also follow this principle when ingesting code

Research Results

1. **Optimal Number of Chroma Collections to Use**
    a. Chroma Collections are able to handle millions of ingested files, so being more granular with our number of Collections (i.e breaking down by content type, etc) is not necessary
    b. **Instead,** we should be considering making the most of the **metadata** that we provide a particular collection and the corresponding documents we put into it
2. **Intelligently Selecting Chroma Collection to Query**
    a. Multiple Projects
        i. In order to account for projects in the long run, we can build on top of the existing architecture of **two collections per project**
        ii. There should be some sort **system level collection** in order to help give the LLM context about the **entire architecture** that a particular project is **apart of**
        iii. On top of this, we can expand the attributes associated with a particular **Project** to include some additional relevant attributes
            1. The space this project belongs to
            2. The dependencies that this project has (who uses this project)
                a. I.e who are our consumers
    b. Docs vs Code Collection
        i. Options
            1. Agentic Routing
                a. We could use the LLM configured to be utilized via our RAG flow in order to make the determination as to whether

the question posted by the user requires a) investigation of docs, b) investigation of code, c) both!

        b. This involves passing this posed question (along with whatever other contextual information that is needed such as the previous messages sent to the user and the the LLMs associated responses ***if this is a conversation)*** to the LLM, and asking it to determine if this question could benefit for either documentation, code, or both

    2. Heuristic Mapping
        a. Setup simple rules that we can follow to automatically identify if this is strictly a code vs docs question
    3. Query Both
        a. A lot of questions posed by users more time than not may be best answered using context from our documentation as well as our code
        b. To account for this, we can **(***for time being)* focus on simply querying both, and then **ranking** the relevancy of documents retrieved and only use **k-most** relevant documents (i.e 5 retrieved from code, 5 from docs, but code is all we need, just use code!)

  ii. **Choice**
    1. For time being, we will query from both and simply rank the results in terms of relevancy
    2. This helps reduce the overhead of querying an LLM to choose for us, or automatically assuming a question is strictly code related despite it benefiting from viewing the design doc AS WELL as looking at a particular function

3. **Simple Embedding Based Retrieval**
    a. Relevant findings found in following article / course
        i. https://medium.com/@sulaiman.shamasna/rag-i-advanced-techniques-with-chroma-dd8c7c08d000
        ii. https://learn.deeplearning.ai/courses/advanced-retrieval-for-ai/lesson/kb5oj/introduction
    b. **Query Expansion**
        i. The big issue with simple RAG systems is that there tends to be a lot of “noise” involved in users questions that we would want to weed out
        ii. In order to do this, we can do the following
            1. Use poses question “How does authentication work in X”
            2. We pass this question to LLM, and we ask the LLM to suggest additional queries
            3. We use these queries in our RAG to retrieve results
            4. Then, we use these retrieved documents via RAG to supplement our final LLM question, which in theory should give us the

necessary context needed for our LLM to answer the question to best of abilities

iii. The only issue with the above approach is that we will have much more documents retrieved, and some of them may not be too relevant to query

iv. To account for this, we can use **Cross-Encoder Re-Ranking to score relevancy of the retrieved results for a sent query**
    1. This allows for us to score them according to relevancy for a particular query

v. Step By Step Plan
    1. Decompose query with Collection Hints **(if complexity calls for it, simple questions should just get easy responses)**
        a. EX) User passes in query, we say to break this down into 2-4 sub-queires, and have the LLM deem if their sub-query is best answered by either CODE or our DOCUMENTATION
    2. Retrieve chunks from relevant collections
        a. Based on sub-queries and suggested retrieval, retrieve chunks from collections
    3. Cross encoder re-ranking
        a. Remove duplicate chunks
        b. Use cross encoder with the **original query** and the **document chunk retrieved from each subquery**
            i. **NOTE:** This is the big benefit of cross encoding, as we are passing the **original query** and the **chunk retrieved** to deem ***how relevant this piece of text is to our original query*** by allowing the query and the chunk to “see” each other directly
        c. The output of this step should be the 10-15 most relevant chunks

vi. Technical Concern
    1. Cross-encoders have token limits, so we’ll need to make sure the retrieved chunk and the user query are able to fit into context window
    2. Slow if we choose to do this sequentially, (for each chunk retrieved, run cross encoder against original query). To account for this, we can do **batch processing** to speed this up

4. Context Length Concerns
    a. We will have **Message** and **Conversation** entities
        i. The message can either be a user created query, or an LLM generated response
        ii. Each message will have a **num tokens** associated with it
    b. In order to **count the number of tokens,** we will need to use the corresponding tokenizer associated with the particular LLM model currently configured for use.

The way to retrieve the count is different based on the LLM provider we are working with

   i. **OpenAI**
      1. Use tiktoken library
   ii. **Claude**
      1. Use claude API
   iii. **Ollama**
      1. Use Autotokenizer

c. We will need to keep track off tokens used for each of the following
   i. System Prompt
   ii. User Prompt
   iii. Context Chunks Provided
   iv. LLM Response

d. We will also need to ensure that we "reserve room" for the response each time (i.e if a user prompt gets use to 95% or something, and LLM cannot construct response without exceeding, we need to truncate or suggest new conversation)

e. We should have alerts for end-users surrounding their usage in a particular conversation

f. **Systems Practical Capability**
   i. Along with the LLM having "limits" of the context length, the **system running the LLM will also have some limits**
   ii. The limits of the particular machine is **RAM/VRAM** available (not just can the computer run the model). In other words, our system *may have enough memory to load the model, but not to support full context*
   iii. Will need to **measure available memory** on system when utilizing local LLMs (**Total VRAM - (**Model Weights + Operating System Overhead + Framework Overhead) **= Available Memory for KV Cache**
      1. **Available Memory KV Cache / Memory Per Token = Max Tokens**

5. **Multiple Embedding Models**
   a. Latency & cost concerns of having to "re-embed" a users query multiple times is negligible
   b. We can implement fairly optimal architecture to account for this, such as de-duplicating any embeddings (i.e if two collections use same embedding model, don't embed multiple times)
   c. On top of de-duplicated, we can run the embeddings in parallel for when we have a large number of unique embedding models in order to speed up processing
      i. **NOTE:** One concern with this approach is the memory availability of systems GPU in getting these embedding models into memory to be ran, so keep into account "how much is available on this system", and "how much will it cost" prior to performing parallel embeddings