Let's break down what a vector database is, why it's essential, and specifically what Qdrant is.

**What is a Vector Database?**

A vector database is a specialized type of database designed to store, manage, and index **high-dimensional vector data**. **These "vectors" are numerical representations of various types of data (like text, images, audio, video, etc.),** **often generated by machine learning models as "embeddings."**

**What is High-Dimensional Vector Data?**

In machine learning and data science, data is often represented as **vectors**. A vector is essentially an ordered list of numbers (or features).

* **Dimension:** The "dimension" of a vector refers to the number of elements (or features) it contains.
* **High-Dimensional:** When a vector has a very large number of dimensions (e.g., hundreds, thousands, or even millions), it's considered "high-dimensional."

So, **high-dimensional vector data** is a collection of data points, where each data point is represented by a vector with a very large number of features.

Think of it this way:

* **Traditional databases** store data in structured formats (like rows and columns in a relational database) and **are great for exact matches** (e.g., "find all customers named John Smith").
* **Vector databases** store data as numerical arrays (vectors) in a multi-dimensional space. The key idea is that **similar data points will have vectors that are "close" to each other in this space.**

**This allows for similarity searches rather than exact matches.** For example, instead of searching for the exact keyword "smartphone," a vector search might return results for "cellphone" and "mobile devices" because their underlying semantic meaning (represented by their vectors) is similar.

**Why are Vector Databases Required?**

Vector databases are crucial for modern AI and machine learning applications, especially with the rise of large language models (LLMs) and generative AI. Here's why they are required:

1. **Handling Unstructured Data:** Traditional databases struggle with unstructured data (images, audio, free text). Vector databases are purpose-built to handle these complex data types by representing them as numerical vectors.
2. **Semantic Search:** This is a major driver. Instead of just keyword matching, vector databases enable searching based on the *meaning* or *context* of the data. This is vital for:
   * **Recommendation Systems:** Finding products, movies, or songs similar to what a user likes.
   * **Image and Video Search:** Searching for visually similar content.
   * **Natural Language Processing (NLP):** Understanding the nuance of language, like finding documents with similar themes or sentiment, even if they don't use the exact same words.
3. **Powering AI Models (especially LLMs/RAG):**
   * **Retrieval Augmented Generation (RAG):** LLMs often have a knowledge cutoff from their training data. Vector databases allow LLMs to access and integrate external**, up-to-date information by retrieving relevant "chunks" of data (as vectors) that can then be used to augment the LLM's response**. This improves the accuracy and relevance of AI-generated content.
   * **Long-term Memory for AI:** Machine learning models typically don't "remember" past inputs beyond their training. Vector databases act as a long-term memory, storing embeddings of data so the models can quickly retrieve and use relevant information without re-processing it every time.
4. **Efficiency and Performance:**
   * **High-Dimensional Data:** Vector databases are optimized to store and query high-dimensional data efficiently, which is challenging for traditional databases.
   * **Approximate Nearest Neighbor (ANN) Search:** They use specialized indexing techniques (like HNSW - Hierarchical Navigable Small World graphs) to quickly find "approximate" nearest neighbors, balancing speed and accuracy, even with billions of vectors.
   * **Scalability:** They are designed to scale horizontally to handle massive datasets and high query loads.
5. **Metadata Filtering:** Beyond just vector similarity, vector databases often allow you to store and filter data based on associated metadata (e.g., searching for similar shoes *from a specific brand*).

**What is Qdrant Vector Database?**

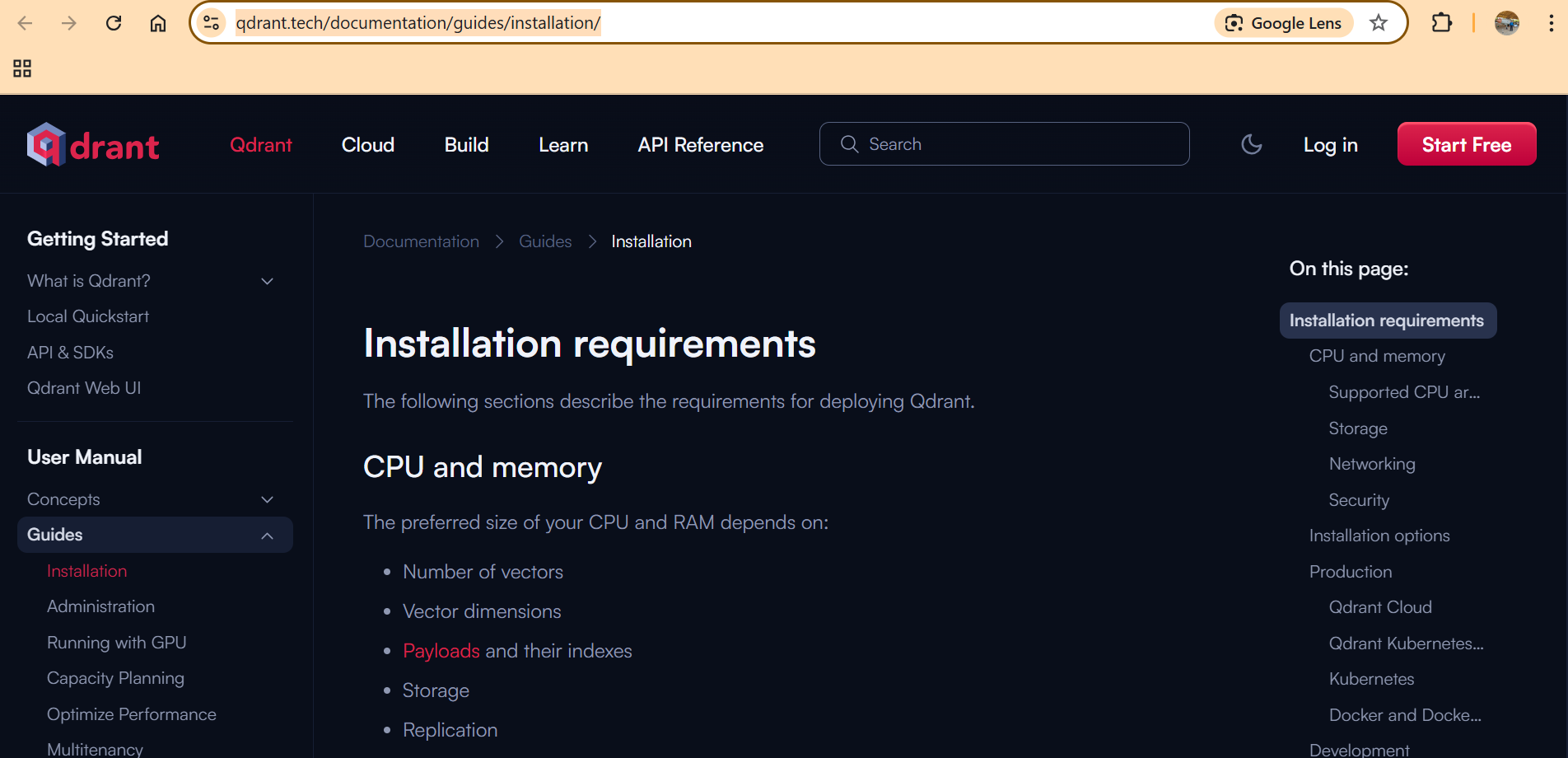
**Qdrant** is an open-source, high-performance **vector similarity search engine and vector database**. It's designed to provide a production-ready service for storing, searching, and managing high-dimensional vectors.

Here are key aspects of Qdrant:

* **Core Concept: Points:** In Qdrant, the fundamental unit of data is a "point." Each point consists of:
  + An **embedding vector**: The numerical representation of your data (e.g., an image, a document, a sound).
  + A **unique ID**: For identifying the point.
  + An **optional payload**: A JSON object that can store additional metadata about the vector (e.g., author of a text, publication date, image tags). This payload is crucial for filtering search results.
* **Collections:** Points are organized into "collections," which are analogous to tables in traditional databases. When creating a collection, you specify:
  + The **dimensionality (size)** of the vectors it will contain.
  + The **distance metric** to be used for similarity searches (e.g., Cosine Similarity, Euclidean Distance, Dot Product). The choice of metric depends on how your vectors were generated by the embedding model.
* **High Performance:**
  + **Written in Rust:** Qdrant is built in Rust, a language known for its speed, memory efficiency, and reliability, making it very performant even under heavy loads.
  + **HNSW Algorithm:** It primarily uses the Hierarchical Navigable Small World (HNSW) algorithm for Approximate Nearest Neighbor (ANN) search, which provides excellent performance for similarity queries.
  + **Quantization:** Qdrant supports quantization techniques (like Scalar Quantization and Binary Quantization) to reduce the memory footprint of vectors, which can further speed up computations at the cost of some accuracy.
* **Key Features:**
  + **Efficient Similarity Search:** Excels at finding vectors that are most similar to a given query vector.
  + **Filtering and Hybrid Search:** Allows you to combine vector similarity search with filtering based on the associated payload metadata. It also supports hybrid search, combining vector similarity with keyword search using sparse vectors.
  + **Scalability:** Designed for cloud-native deployments, supporting horizontal scaling to handle large volumes of data and queries.
  + **Flexible APIs and Clients:** Provides RESTful HTTP and gRPC APIs, along with client libraries for various programming languages (Python, TypeScript/JavaScript, Go, Rust, Java).
  + **Storage Options:** Offers in-memory storage (highest speed) and Memmap storage (balances speed and persistence).
  + **Data Management:** Provides typical database operations like inserting, updating, and deleting points.
  + **Security:** Offers features like user authentication, role-based access control, encryption, and audit trails.
* **Use Cases:** Qdrant is widely used for:
  + Semantic search
  + Recommendation systems
  + **Retrieval Augmented Generation (RAG) in LLM applications**
  + Anomaly detection
  + Image and video search
  + Personalized advertising
  + Data analysis and clustering

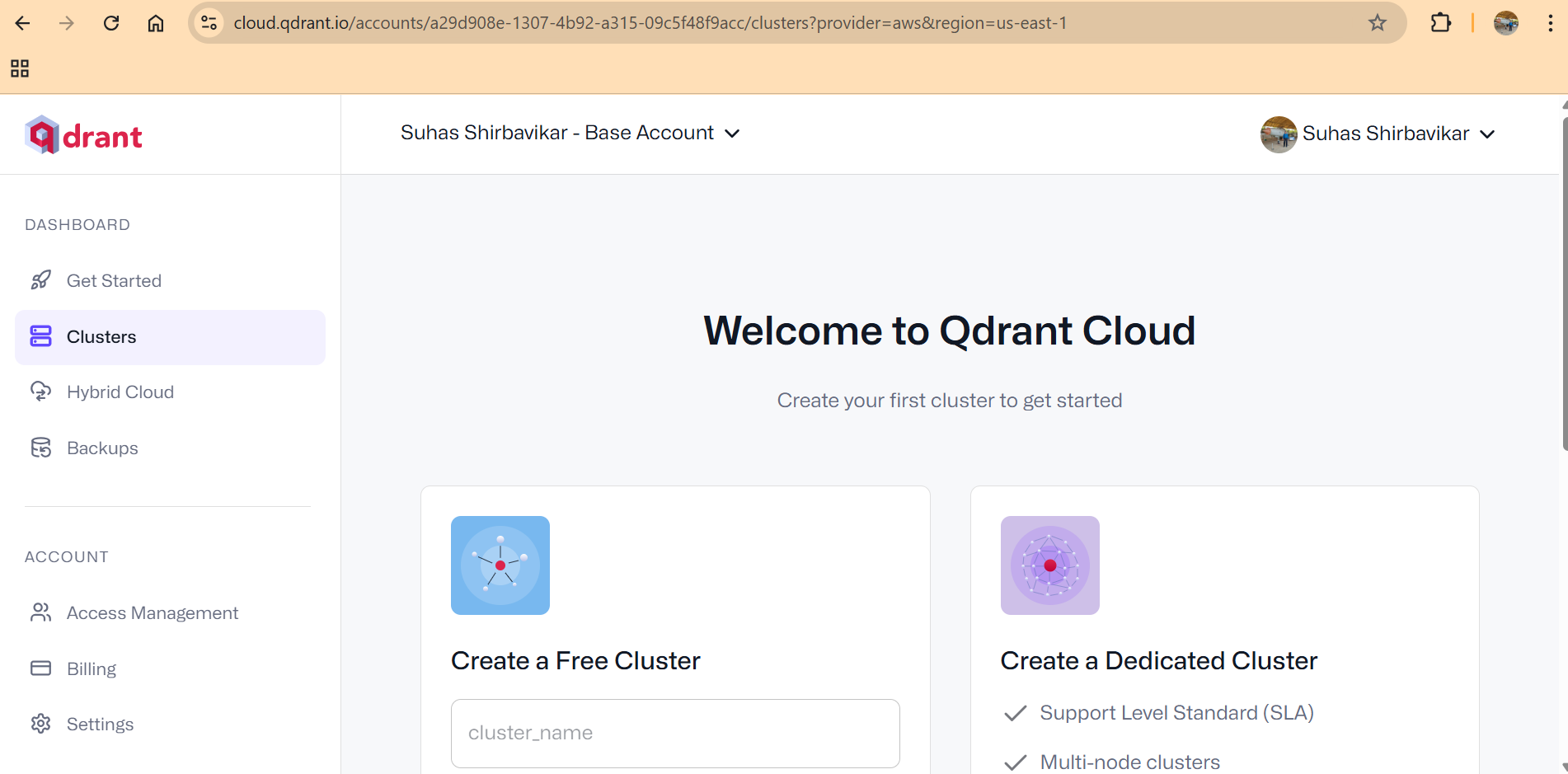
In essence, Qdrant provides a robust and efficient solution for managing and querying the vector embeddings that power many of today's most advanced AI applications

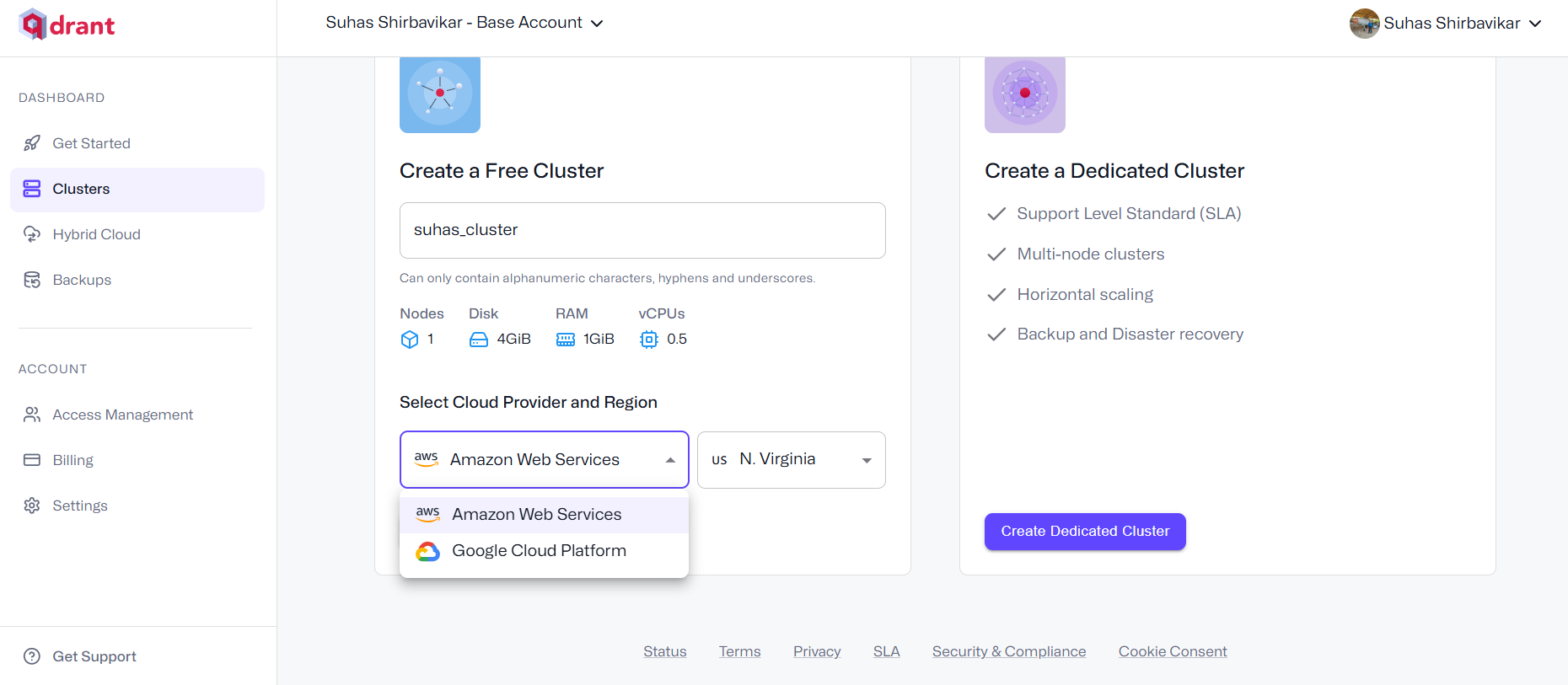
<https://qdrant.tech/documentation/guides/installation/>



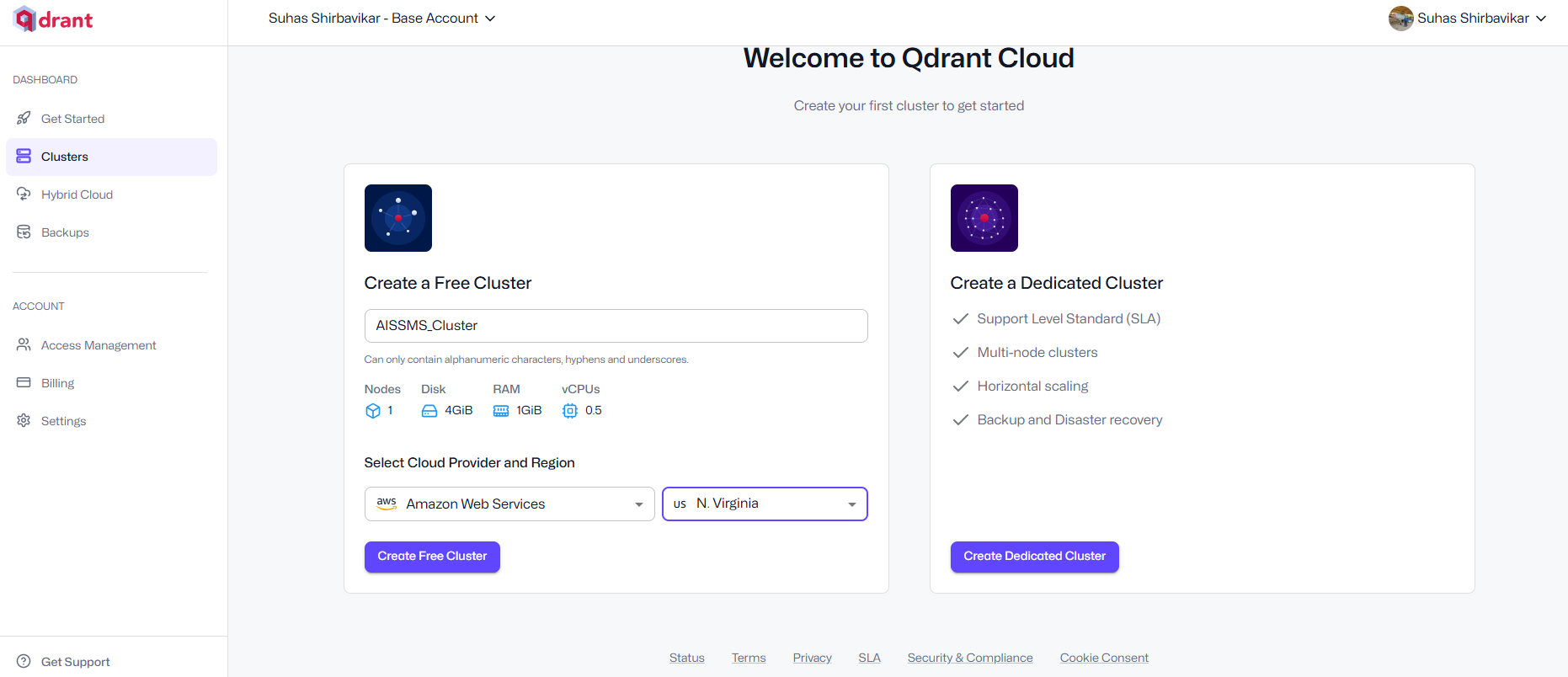
Click on Log in

Log in through gmail id

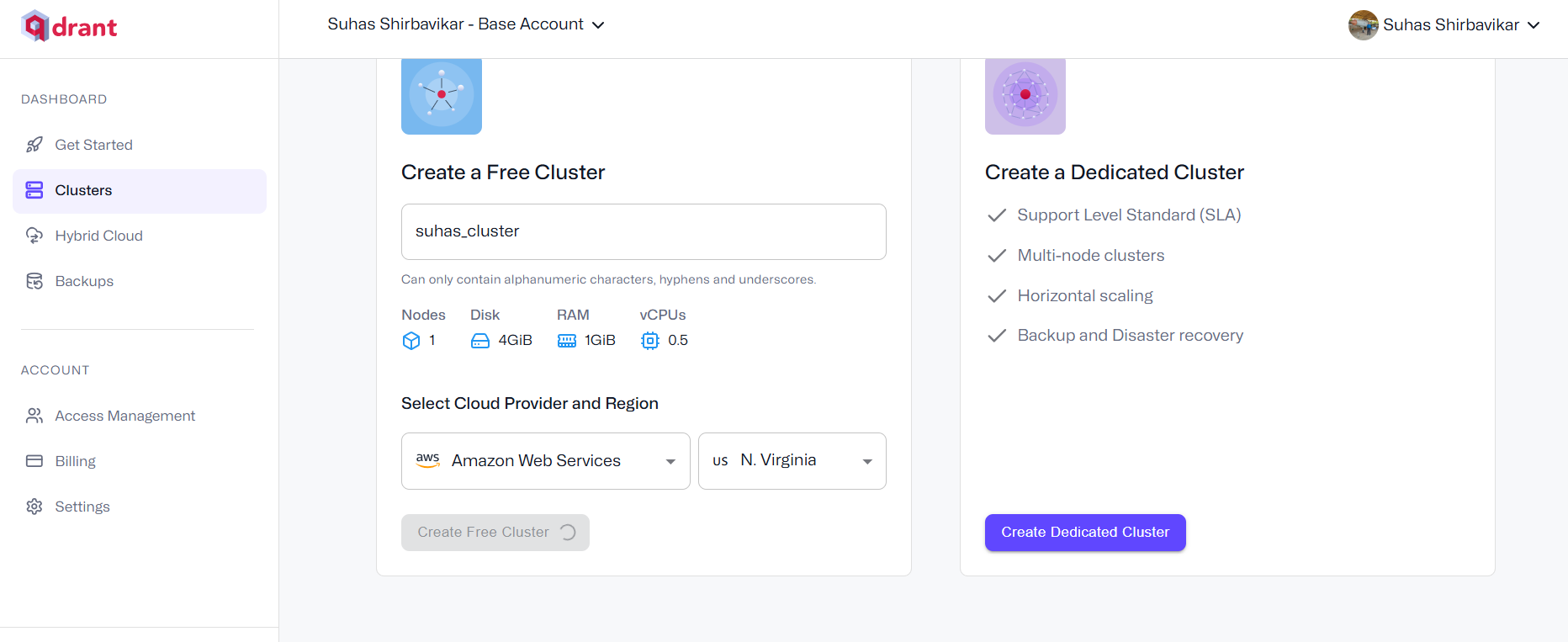


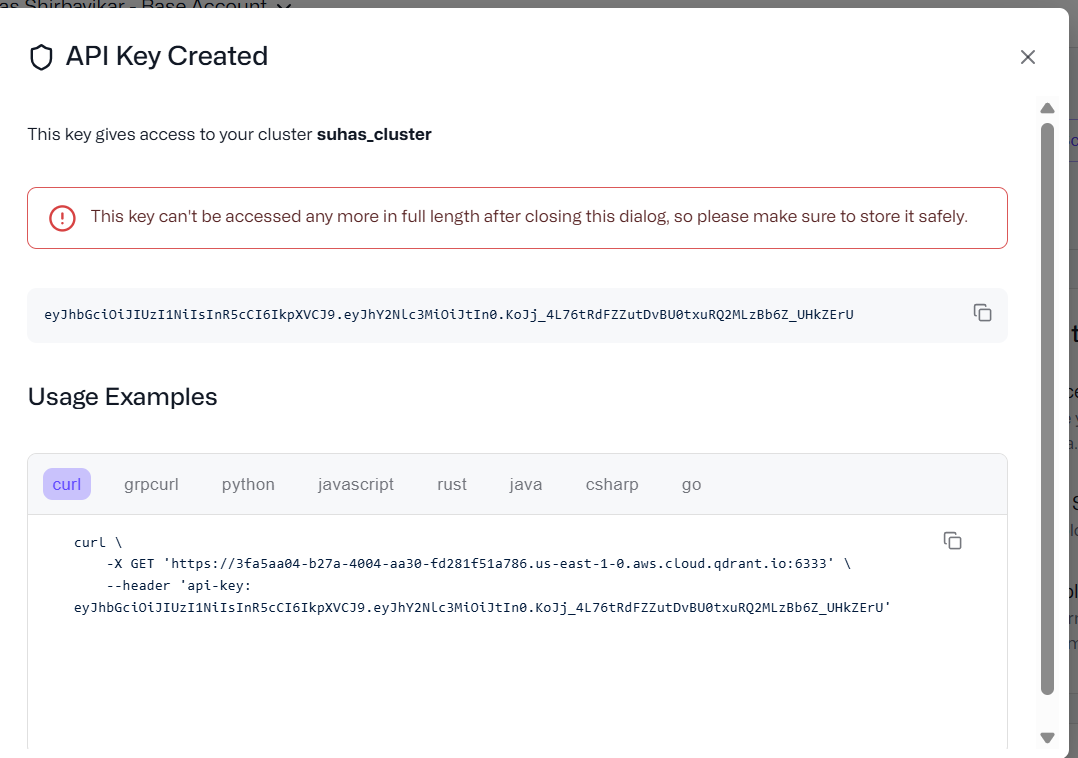


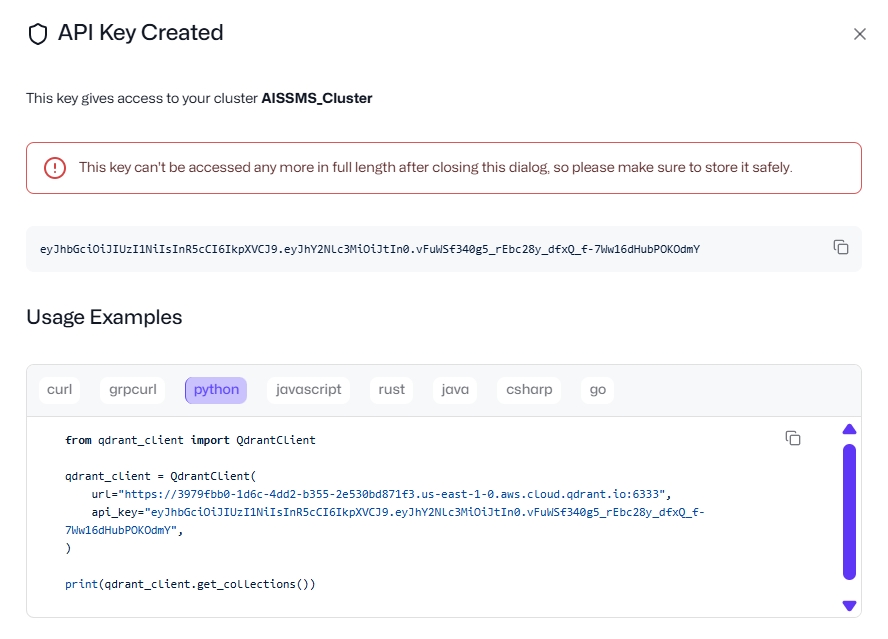
**Provide Cluster name and Select appropriate Cloud Vendor….**

****

**Click on Create Free Cluster**







**Click on python**

**API Key and URL Will be generated… 😊**

**Copy Code….**

from qdrant\_client import QdrantClient

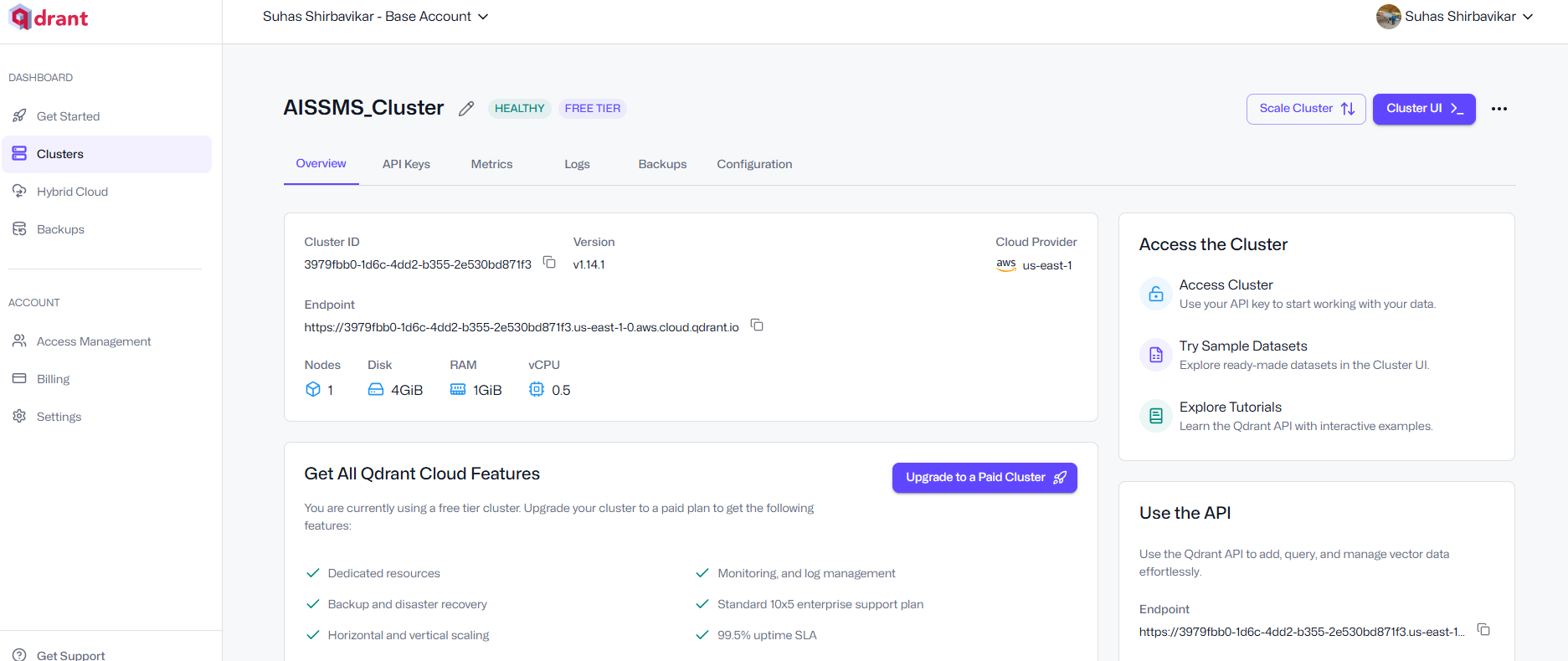
qdrant\_client = QdrantClient(

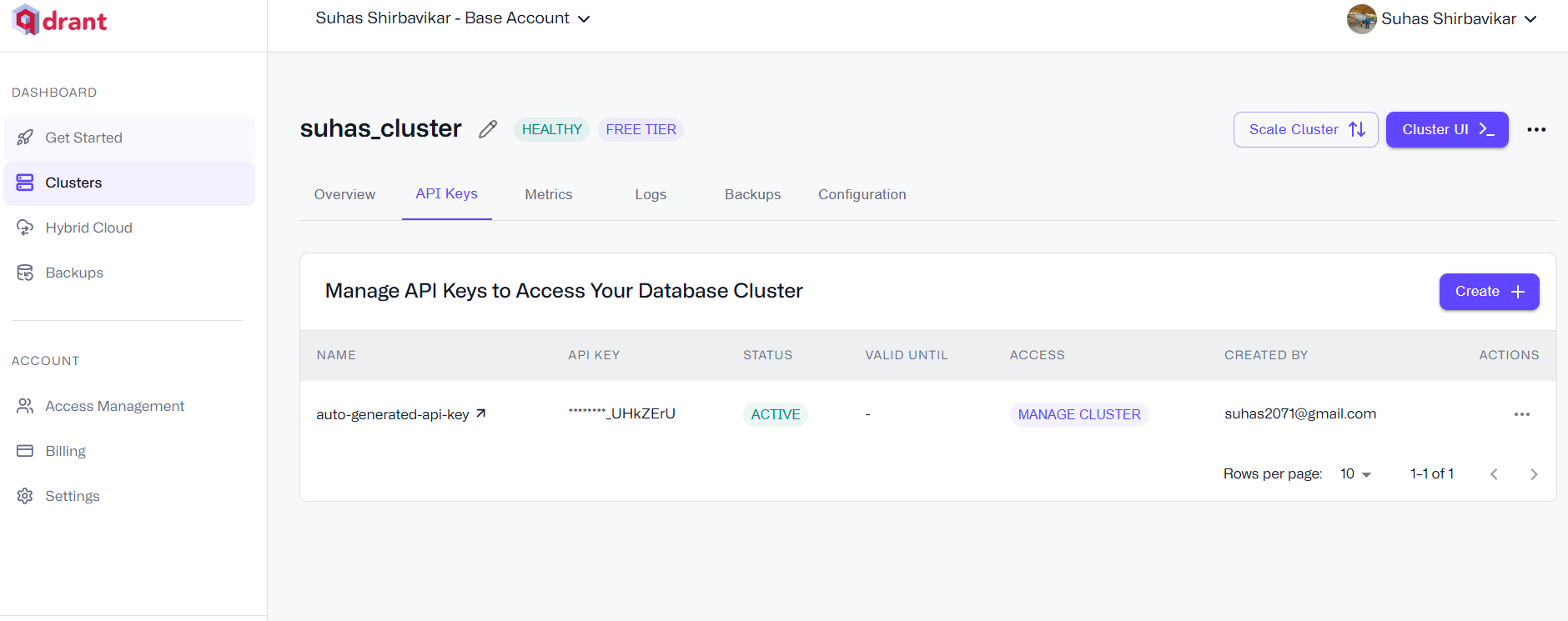
url="https://3979fbb0-1d6c-4dd2-b355-2e530bd871f3.us-east-1-0.aws.cloud.qdrant.io:6333",

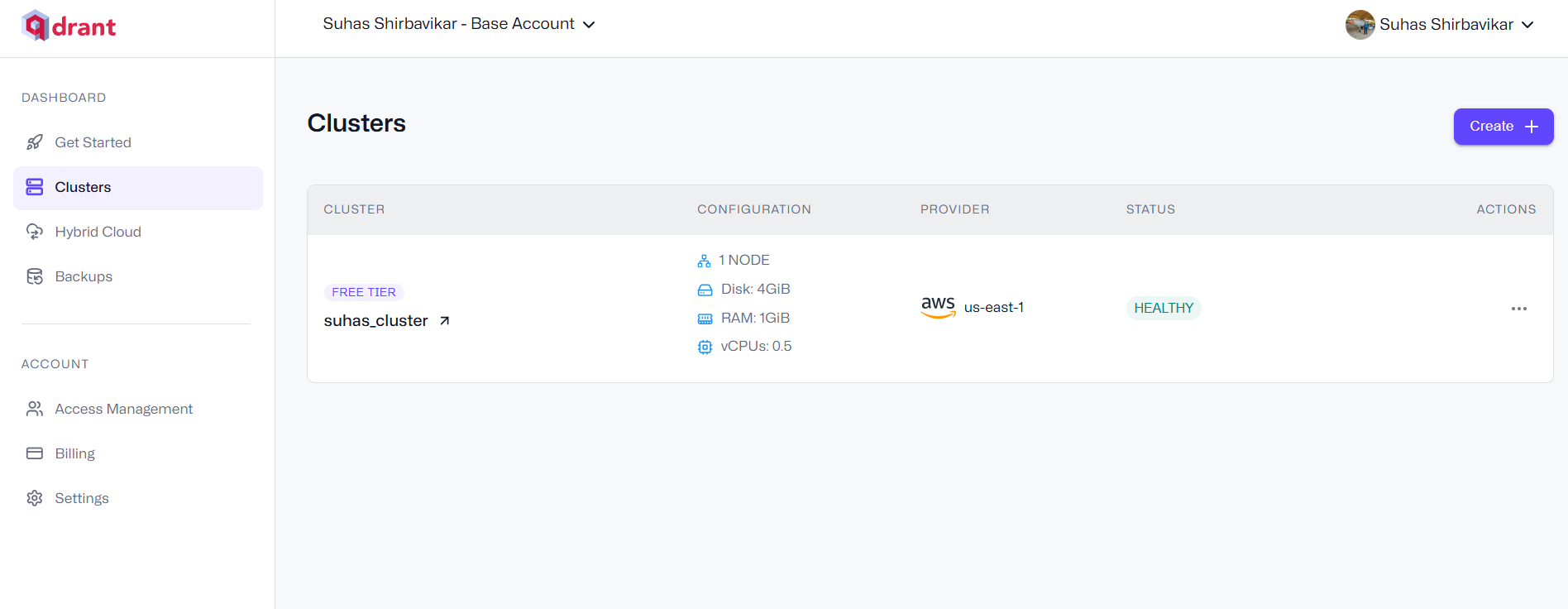
api\_key="eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJhY2Nlc3MiOiJtIn0.vFuWSf340g5\_rEbc28y\_dfxQ\_f-7Ww16dHubPOKOdmY",

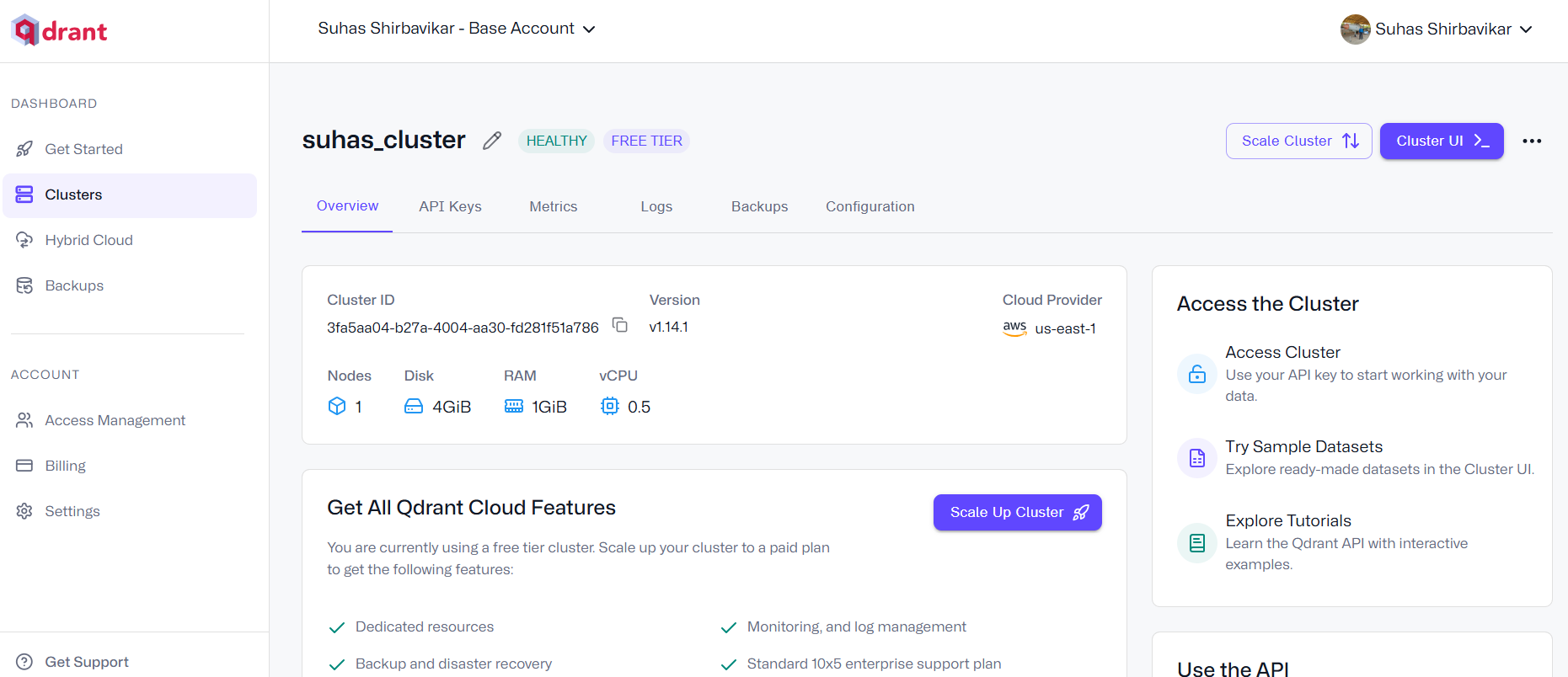
)

print(qdrant\_client.get\_collections())



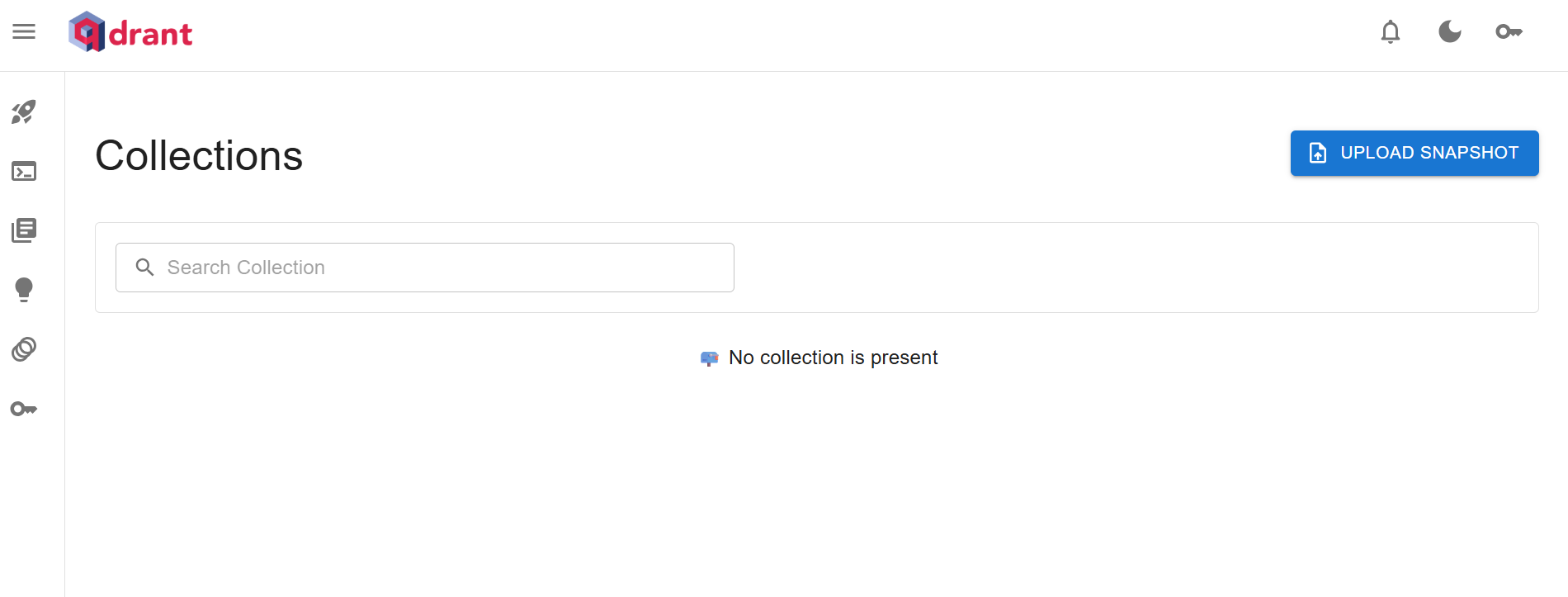




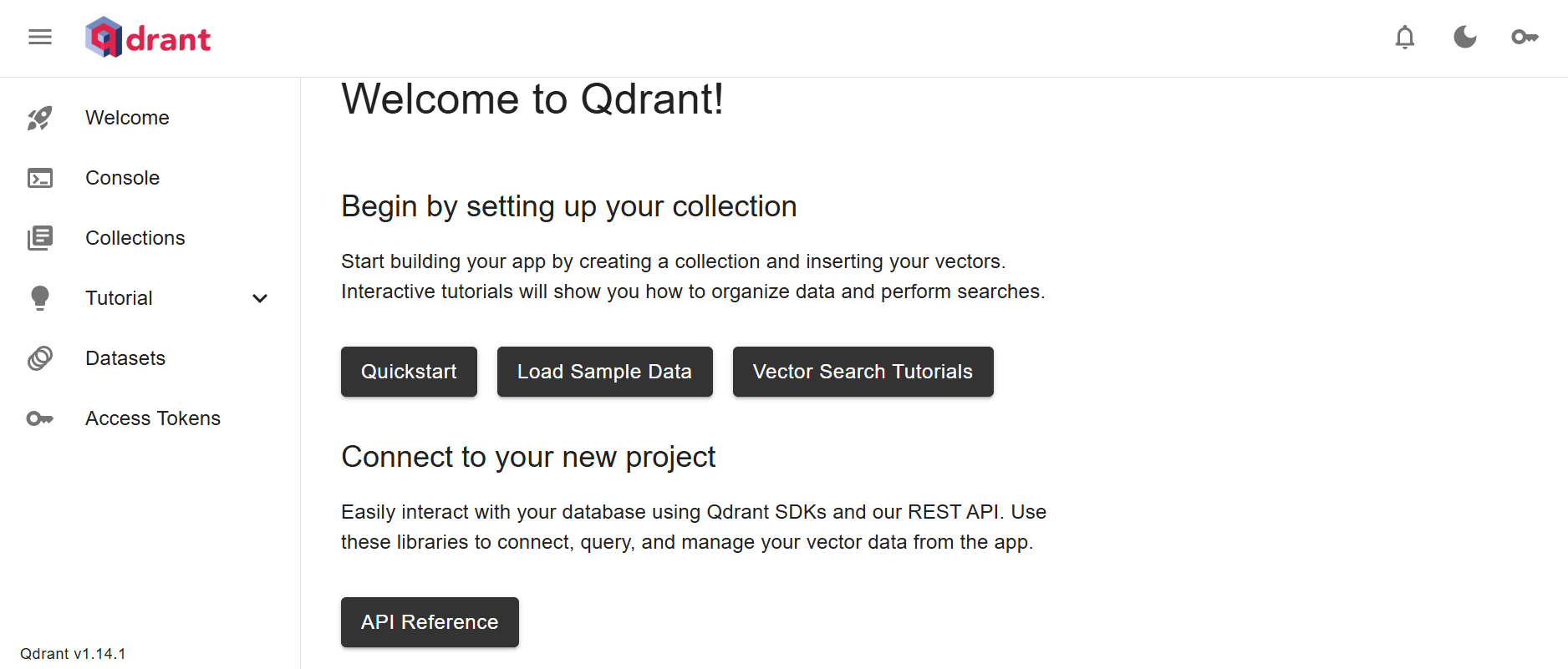


**Click on Cluster UI**

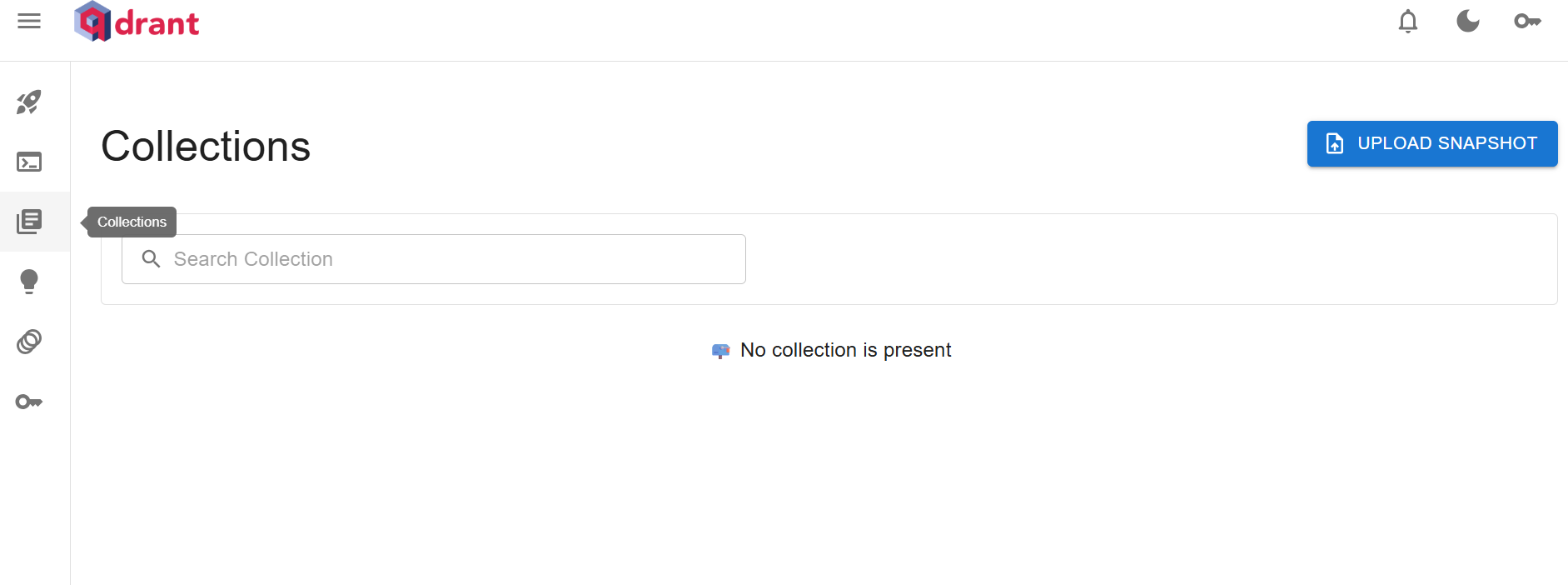
**After entering API Key…**

****

**Rest we will do from coding on Google Colab or on AWS ec2 instance…..**

****

**We finished with Qdrant part1….Click on Collections..**

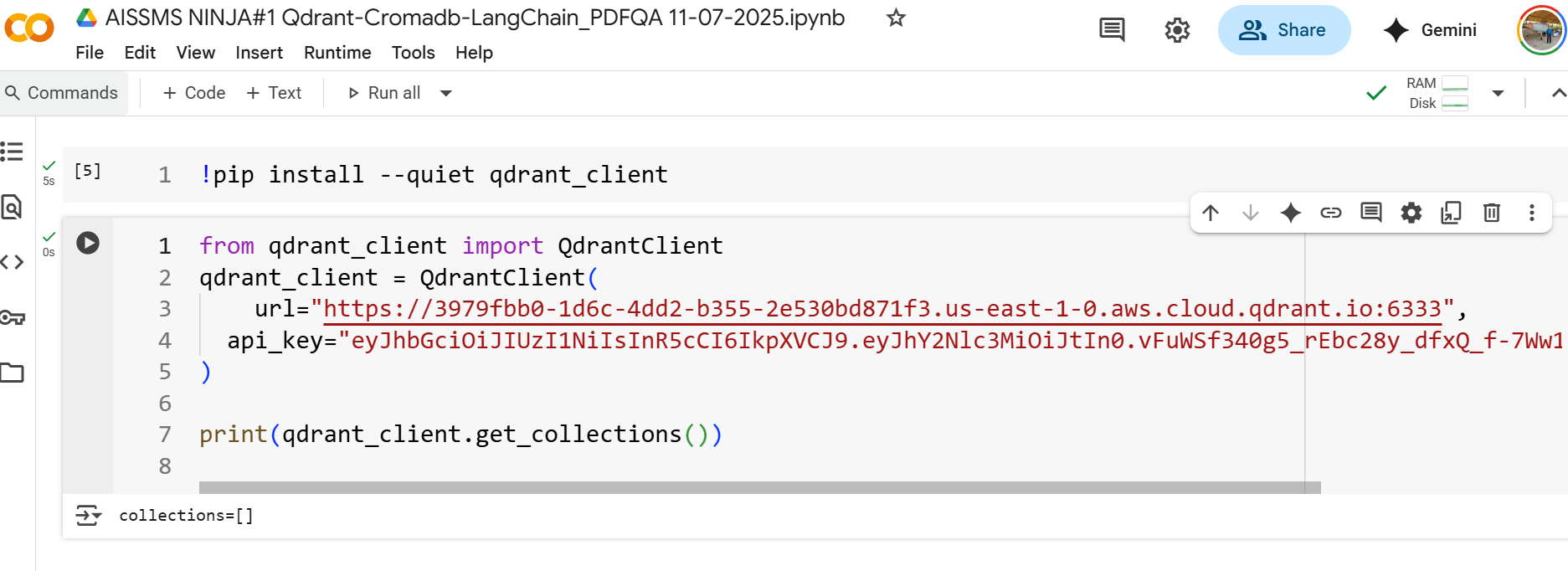
****

**Congratulations…… 😊**

-------------------==========================================---------------------------

**Part 2**

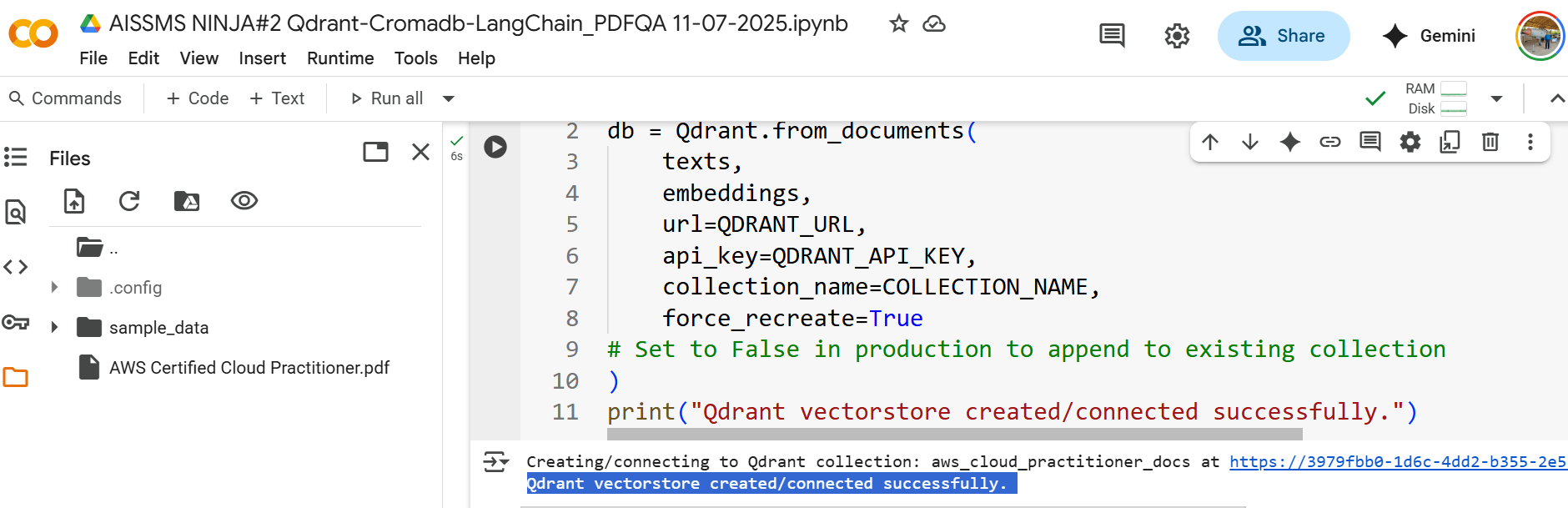
**AISSMS NINJA#1 Qdrant-Cromadb-LangChain\_PDFQA 11-07-2025.ipynb**

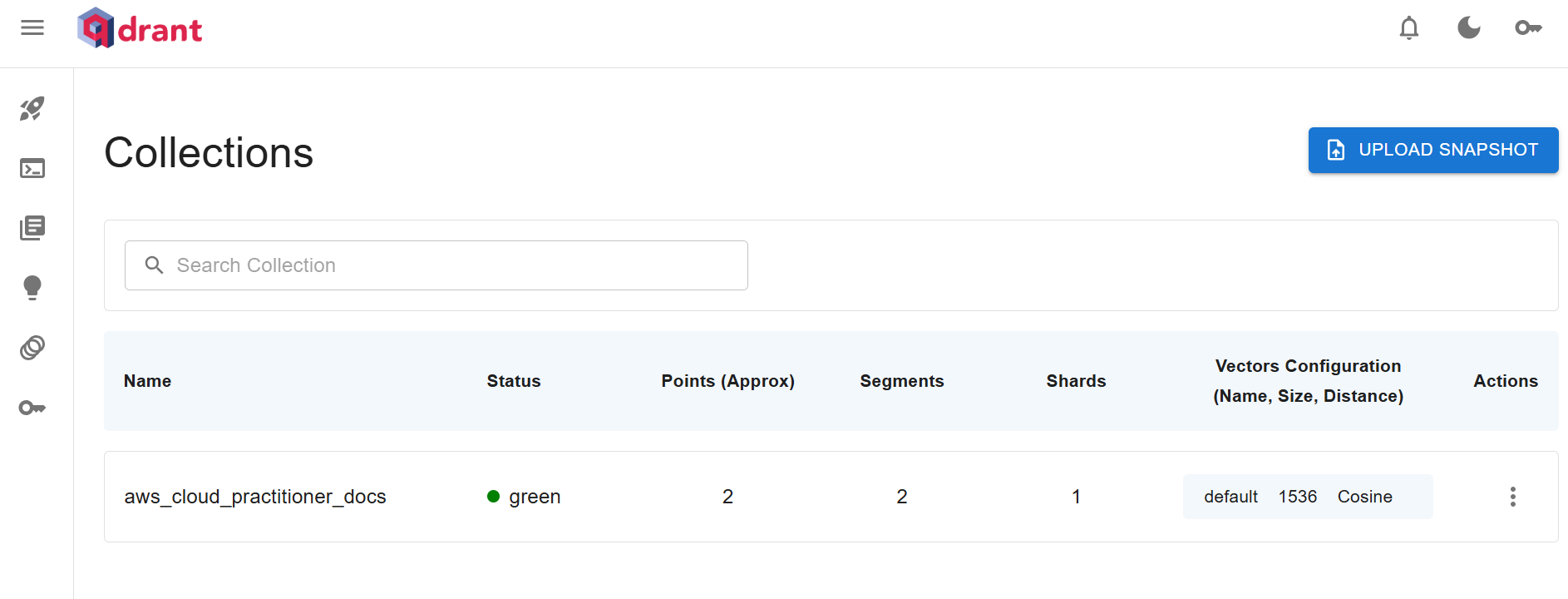
****

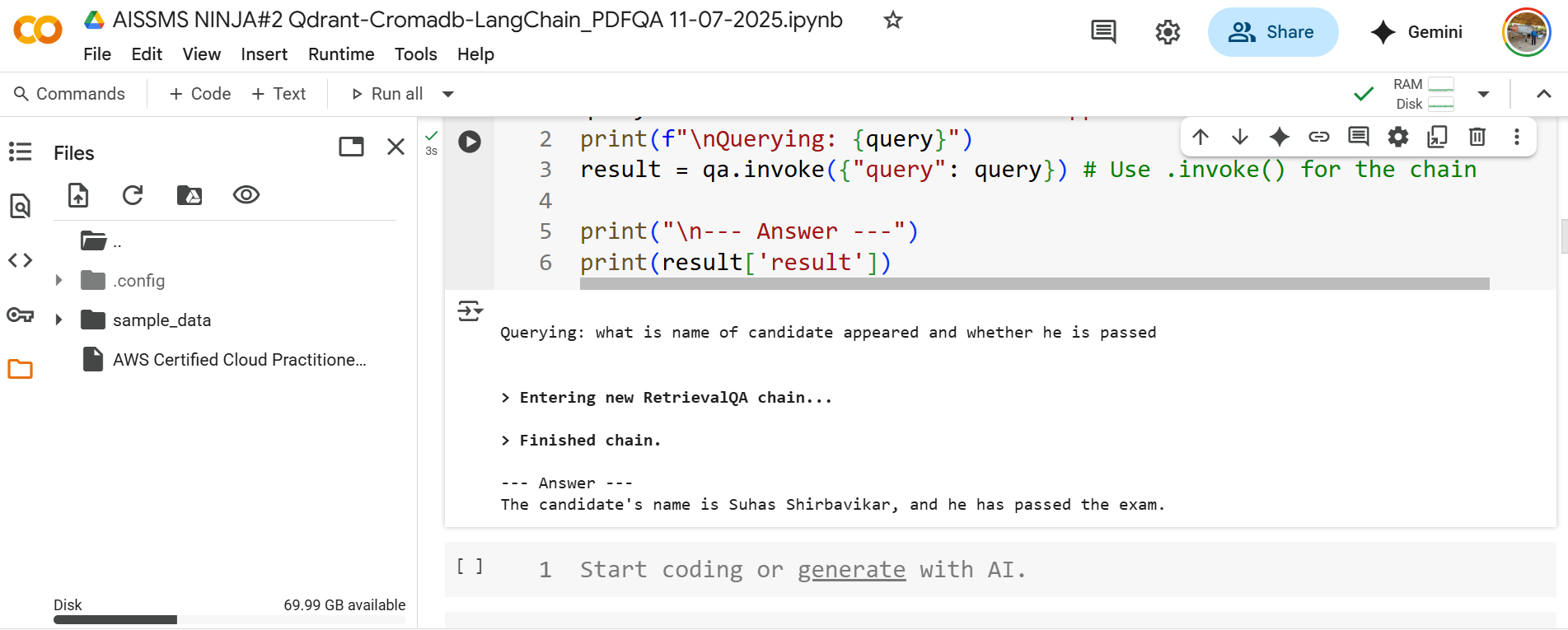
**No Collections Created yet**

**Part3**

**AISSMS NINJA#2 Qdrant-Cromadb-LangChain\_PDFQA 11-07-2025.ipynb**

****

****

****

**New OPEN AI API Key**

**import os**

**from langchain\_community.document\_loaders import PyPDFLoader**

**from langchain.chains.question\_answering import load\_qa\_chain**

**from langchain\_openai import ChatOpenAI**

**# Use ChatOpenAI for modern models**

**from langchain\_core.documents import Document**

**from langchain.chains import RetrievalQA**

**from langchain.text\_splitter import CharacterTextSplitter**

**from langchain\_openai import OpenAIEmbeddings**

# Using langchain\_openai for embeddings too

# New import for Qdrant

**from langchain\_community.vectorstores import Qdrant**

**from langchain\_core.language\_models.chat\_models import BaseChatModel**

**from langchain\_core.messages import BaseMessage, AIMessage, HumanMessage, SystemMessage**

**from langchain\_core.outputs import ChatResult, Generation, ChatGeneration**

**from typing import Any, List, Optional**

**# --- Qdrant specific connection details ---**

**# For a local Qdrant instance running in Docker:**

**QDRANT\_URL = "http://localhost:6333"**

**QDRANT\_API\_KEY = None # No API key for local instance**

**# For Qdrant Cloud, replace with your actual URL and API Key:**

**# QDRANT\_URL = "YOUR\_QDRANT\_CLOUD\_URL"**

**# QDRANT\_API\_KEY = "YOUR\_QDRANT\_CLOUD\_API\_KEY"**

# Define a collection name (Qdrant's equivalent of a table/index)

**COLLECTION\_NAME = "aws\_cloud\_practitioner\_docs"**

# 1. Load document

**loader = PyPDFLoader ("AWS Certified Cloud Practitioner.pdf")**

**documents = loader.load()**

**# 2. Split the documents into chunks**

**text\_splitter = CharacterTextSplitter(chunk\_size=200, chunk\_overlap=20) # Increased chunk size for better context**

**texts = text\_splitter.split\_documents(documents)**

**# 3. Initialize Embeddings (from langchain\_openai for consistency)**

**embeddings = OpenAIEmbeddings()**

**# 4. Create the Qdrant vectorestore to use as the index**

# Use Qdrant.from\_documents to upload texts and embeddings to Qdrant

# 'force\_recreate=True' is useful for development to clear the collection each time

**print(f"Creating/connecting to Qdrant collection: {COLLECTION\_NAME} at {QDRANT\_URL}...")**

**db = Qdrant.from\_documents(**

**texts,**

**embeddings,**

**url=QDRANT\_URL,**

**api\_key=QDRANT\_API\_KEY,**

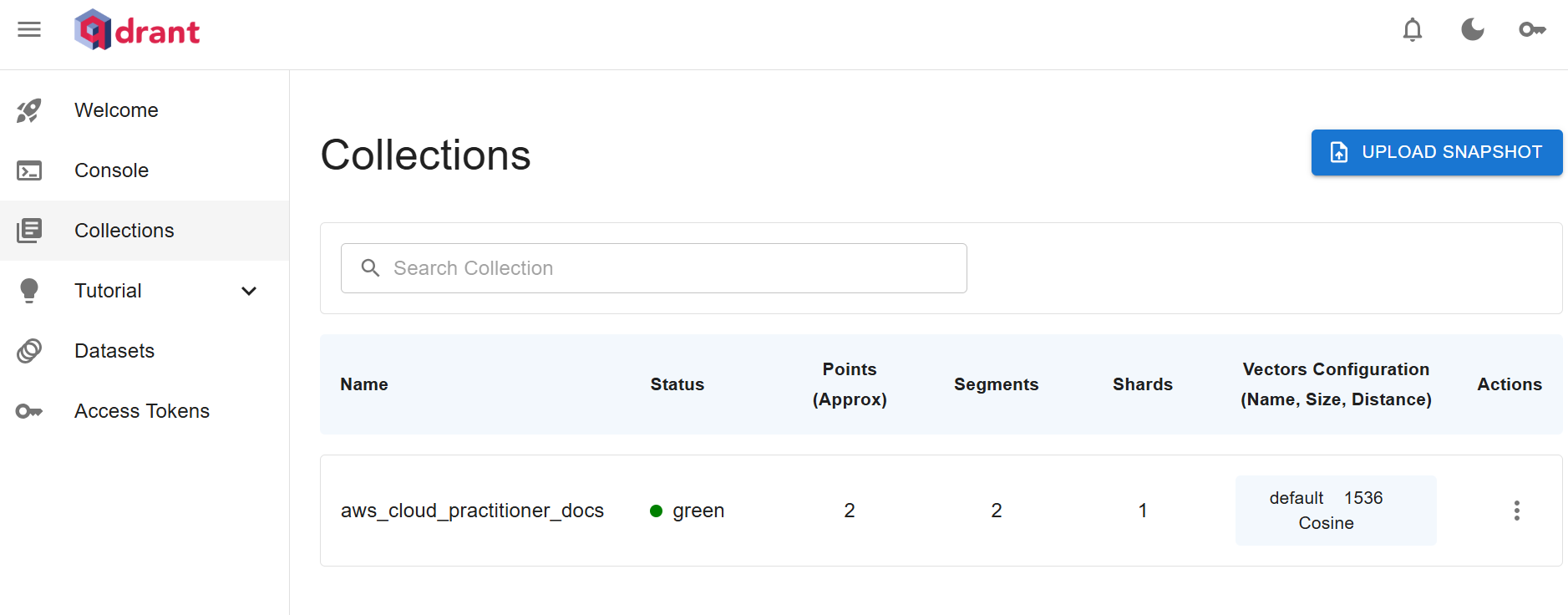
**collection\_name=COLLECTION\_NAME,**

**force\_recreate=True**

**# Set to False in production to append to existing collection**

**)**

**print("Qdrant vectorstore created/connected successfully.")**



**Created Successfully on Qdrant …Congratulations**

**# 5. Expose this index in a retriever interface**

**retriever = db.as\_retriever(search\_type="similarity", search\_kwargs={"k": 2})**

**# 6. Create a chain to answer questions**

**qa = RetrievalQA.from\_chain\_type(**

**llm= ChatOpenAI() # Using mock LLM for demo**

**chain\_type="stuff",**

**retriever=retriever,**

**return\_source\_documents=True, # Set to True to see which docs were used**

**verbose=True # Helpful for debugging**

**)**

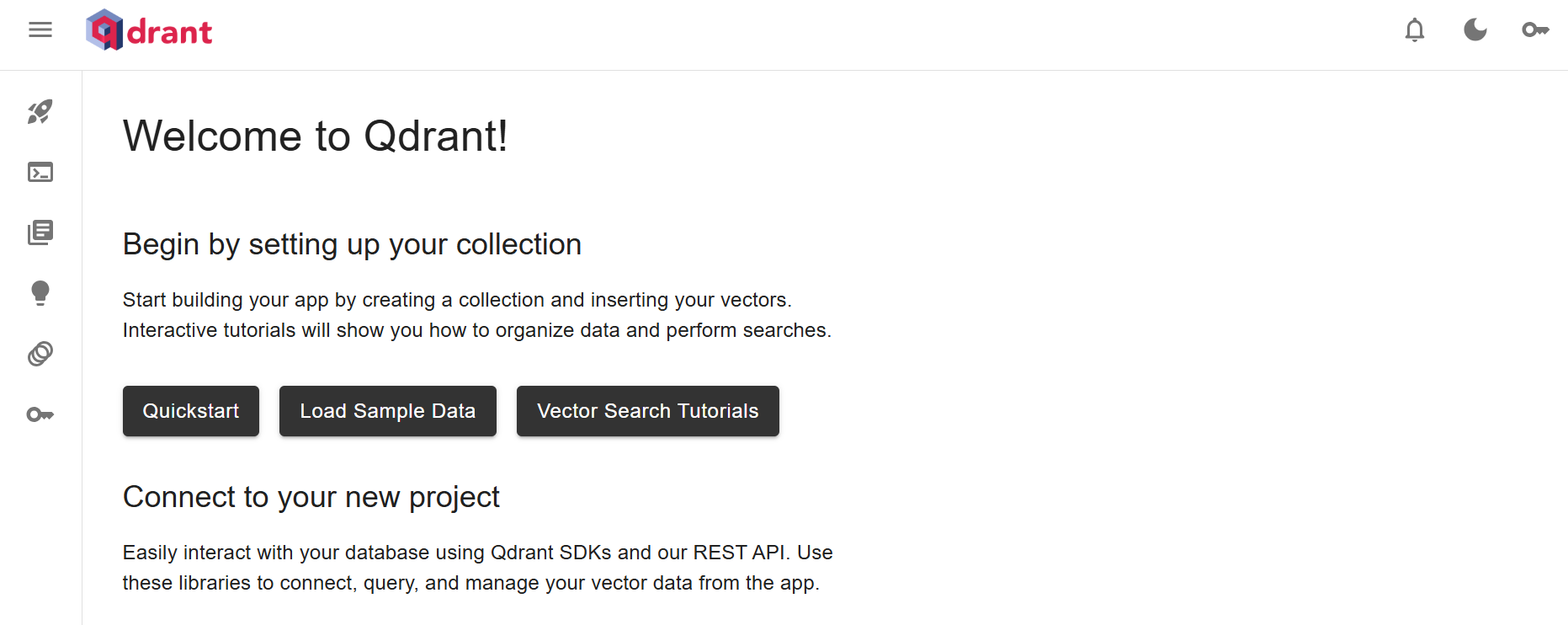
**query = "what is name of candidate appeared and whether he is passed"**

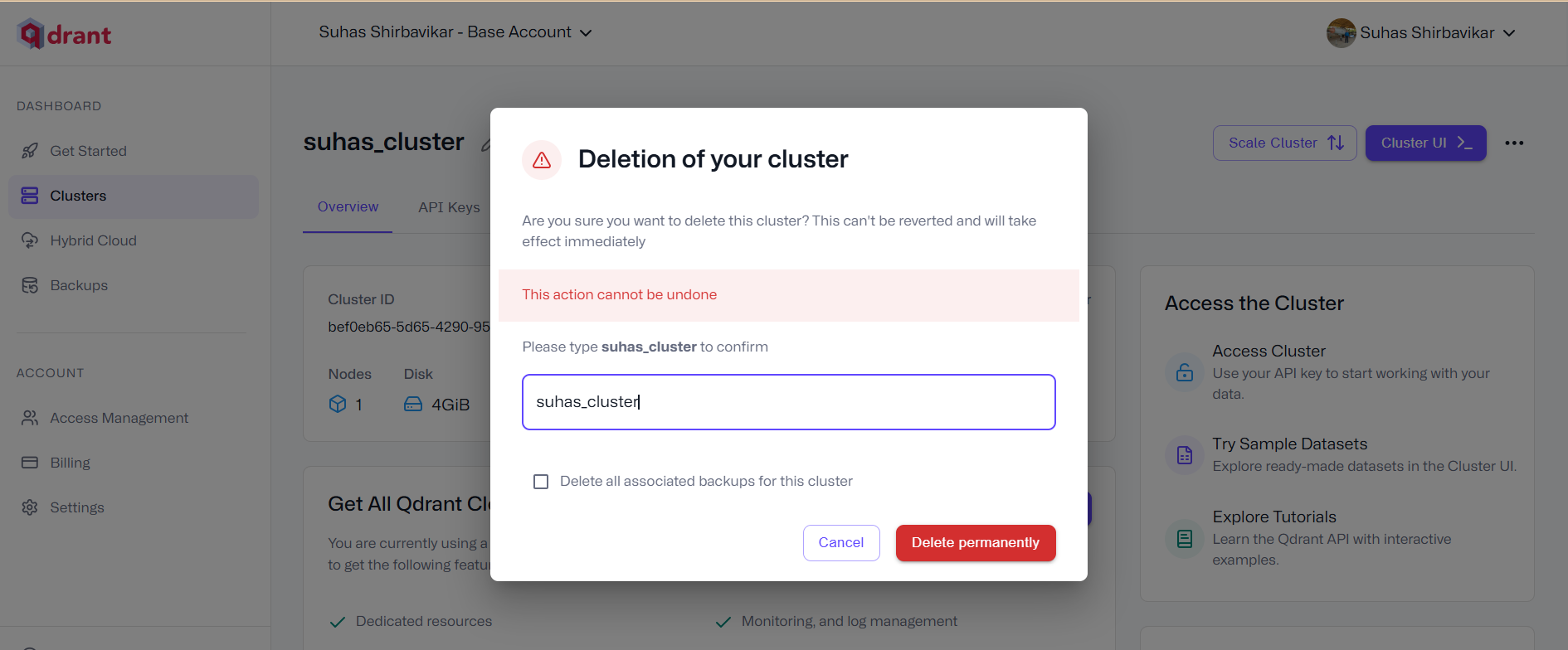
**print(f"\nQuerying: {query}")**

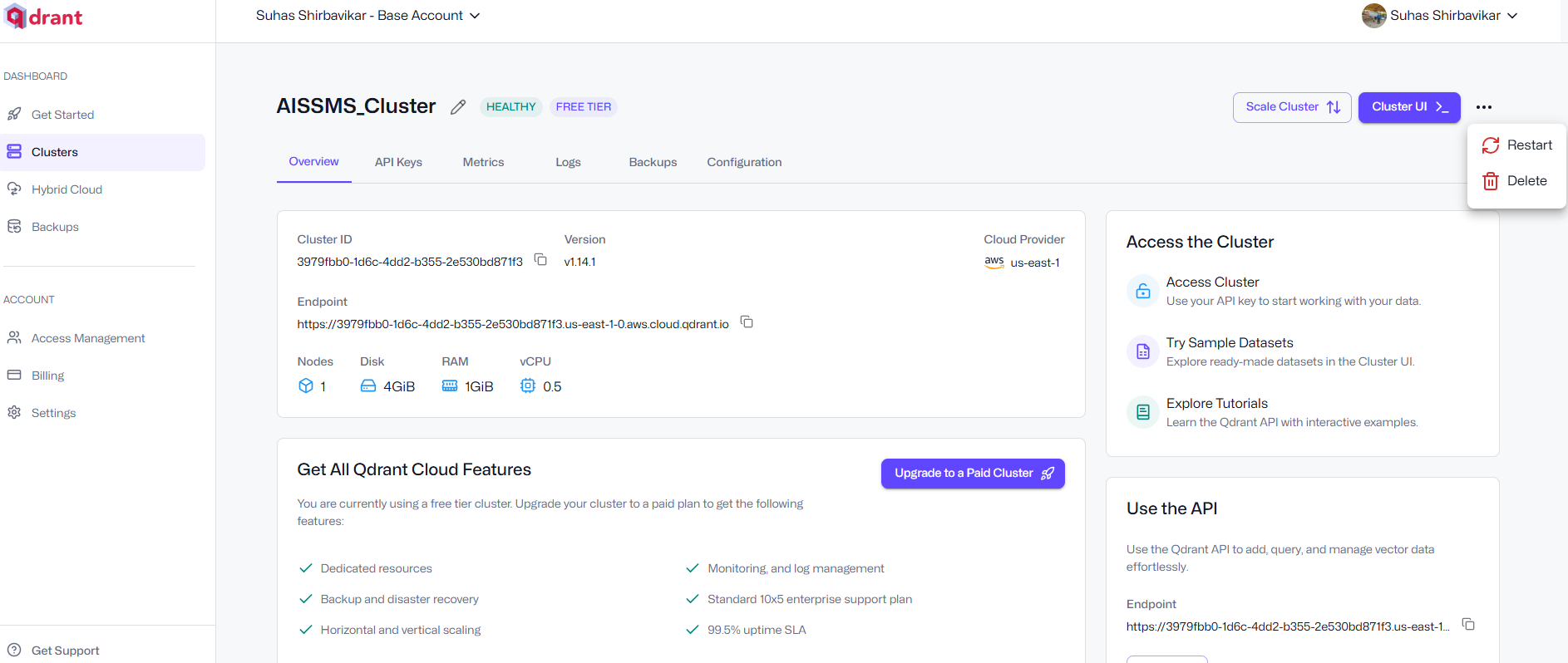
**result = qa.invoke({"query": query}) # Use .invoke() for the chain**

**print("\n--- Answer ---")**

**print(result['result'])**







Good Practice to delete Cluster…. 😊