## Task 4: Think about how you would scale this system to support ~Millions of objects

1. **API fetching optimization**
   1. **Asynchronous programming:** Utilizing asynchronous programming with asyncio can significantly improve throughput by allowing concurrent execution of API requests. This can be achieved using asyncio.gather() to fetch data from multiple sources concurrently. This can be particularly useful when you need to make a large number of non-blocking API calls.

| import asyncio import aiohttp  async def fetch\_data\_async(api\_client, session):  async with session.get(api\_client.api\_url) as response:  data = await response.json()  return data  async def fetch\_all\_data\_async(api\_clients):  async with aiohttp.ClientSession() as session:  tasks = [fetch\_data\_async(api\_client, session) for api\_client in api\_clients]  data\_chunks = await asyncio.gather(\*tasks)  return data\_chunks  async def run\_pipeline\_async(skip: int = None, limit: int = None):  qualys\_api\_client = QualysApiClient(API\_TOKEN, QUALYS\_API\_URL)  crowdstrike\_api\_client = CrowdstrikeApiClient(API\_TOKEN, CROWDSTRIKE\_API\_URL)   *# Fetch data asynchronously*  data\_chunks = await fetch\_all\_data\_async([qualys\_api\_client, crowdstrike\_api\_client])    *# Process data chunks*  for data\_chunk in data\_chunks:  process\_data(data\_chunk)  *# Execute the asynchronous pipeline* asyncio.run(run\_pipeline\_async()) |
| --- |

* 1. **Multithreading for Concurrent API Calls:** Use multithreading to make concurrent API requests and fetch data from multiple endpoints simultaneously. Use the concurrent.futures module to create a thread pool that can handle multiple API calls simultaneously. This can significantly reduce the time spent waiting for API responses, especially when dealing with high-latency calls.

| from concurrent.futures import ThreadPoolExecutor  def fetch\_data(api\_client):  response = requests.get(api\_client.api\_url)  return response.json()  def fetch\_all\_data(api\_clients):  with ThreadPoolExecutor() as executor:  data\_chunks = list(executor.map(fetch\_data, api\_clients))  return data\_chunks  def run\_pipeline\_threaded(skip: int = None, limit: int = None):  qualys\_api\_client = QualysApiClient(API\_TOKEN, QUALYS\_API\_URL)  crowdstrike\_api\_client = CrowdstrikeApiClient(API\_TOKEN, CROWDSTRIKE\_API\_URL)   *# Fetch data using ThreadPoolExecutor*  data\_chunks = fetch\_all\_data([qualys\_api\_client, crowdstrike\_api\_client])    *# Process data chunks*  for data\_chunk in data\_chunks:  process\_data(data\_chunk)  *# Execute the threaded pipeline* run\_pipeline\_threaded() |
| --- |

**Advantage of API optimization**

* **Improved Performance:** Achieves maximum throughput by concurrently processing and non-blocking execution allowing for efficient handling of I/O-bound tasks, enhancing performance.
* **Concurrency:** Enables multiple tasks to run concurrently within a single thread, optimizing resource usage.
* **Scalability:** Well-suited for handling large numbers of concurrent operations, facilitating scalability.
* **Simplicity:** Simplifies handling of asynchronous operations with async and await, enhancing code readability.
* **Optimal Resource Utilization:** Balances resource usage for efficient performance scalability, even under heavy workloads.
* **Responsive and Efficient:** Ensures application responsiveness while efficiently utilizing system resources for a smooth user experience.

Note: The same approach can be used during data normalization and data deduping for greater efficiency.

1. **Profiling:**

Optimizing algorithms involves identifying and improving the performance bottlenecks in your code. This process requires profiling, which involves analyzing the execution time of different parts of your code to pinpoint areas that need optimization. Profiling tools like cProfile and line\_profiler provide insights into the time spent in each function or line of code, helping you identify inefficient sections.

| *#pipelines.py:* import cProfile import normalization  def run\_pipeline(skip: int = None, limit: int = None) -> None:  *# Pipeline Code*  *# Profile the run\_pipeline function to identify performance bottlenecks* cProfile.run("run\_pipeline(skip=skip, limit=limit)") |
| --- |

| *#normalization.py:* import line\_profiler  *# Add line profiler decorator to the function to profile line-by-line* @profile def normalize\_hosts(hosts: List[dict], source: str) -> List[dict]:  normalized\_hosts = []  for host in hosts:  *# Normalization logic*  return normalized\_hosts  *# Profile the normalize\_hosts function to identify line-by-line performance* lp = line\_profiler.LineProfiler() lp\_wrapper = lp(normalize\_hosts) lp\_wrapper(qualys\_hosts, 'QUALYS') lp.print\_stats() |
| --- |

| *#clients.py:* import cProfile  *# Profile the fetch\_hosts function in QualysApiClient to identify performance bottlenecks* cProfile.runctx("qualys\_api\_client.fetch\_hosts(skip=skip, limit=limit)", globals(), locals()) |
| --- |

1. **MongoDB Optimization:**
2. **Indexing Strategies:** Proper indexing of MongoDB collections can significantly enhance query performance. Identify frequently accessed fields such as id, modifiedOn, os, etc., and create indexes on these fields using MongoDB's indexing capabilities.

| *# Check if index exists before creating it* if "id" not in collection.index\_information():  collection.create\_index("id")  if "modifiedOn" not in collection.index\_information():  collection.create\_index("modifiedOn")  if "os" not in collection.index\_information():  collection.create\_index("os") |
| --- |

1. **Sharding:** As the data volume increases, sharding the MongoDB database can help distribute data across multiple nodes, enabling horizontal scaling and improving overall throughput. Configure sharding based on the appropriate shard key, such as id or modifiedOn, to ensure efficient data distribution. However there are certain case when we should not shard. Sharding does give the capability to scale horizontally, but also comes with added complexity, not used correctly can degrade performance. Because of the Immutable nature of Shard key and the overhead of moving/balancing data/chunk between shards.

* Application database where application is still evolving.
* ETL data processing where specs might change.
* Data lake where usage pattern is not know.