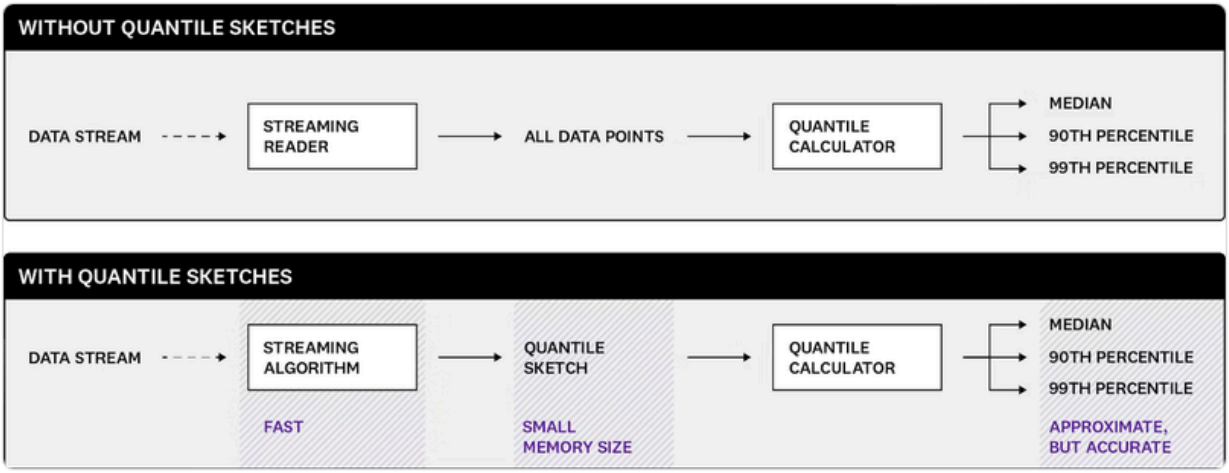
Rahul Nair
CMSC818Q
Final Project Report

Evaluating DDSketch for Real-Time Monitoring Workloads

# Introduction

Quantile metric estimation is essential in enterprise monitoring pipelines for summarizing large data streams. When dealing with tens or hundreds of millions of data points, computing quantiles directly becomes infeasible because each query requires resorting or reprocessing the streamed data.

Sketch algorithms offer a practical alternative. By accepting a bounded margin of error, users can construct a compact data structure that captures the key distributional characteristics of a dataset. Unfortunately, many existing sketch algorithms suffer from key trade-offs. Some do not provide strict relative error guarantees. This can be problematic in large-scale data streams. Others are not fully mergeable, meaning that combining multiple sketches from distributed nodes, for example, may produce biased or inaccurate results. This is a limitation in systems that rely on sketch algorithms for aggregating metrics across distributed services and regions.




Moreover, this is where Datadog's DDSketch (Distributed Distribution Sketch) shines. DDSketch is fast, fully mergeable, and offers provable relative-error bounds for quantile approximation. It uses logarithmic binning to provide consistent accuracy across scales and maintains a small, fixed-size memory footprint regardless of input volume. These properties make it particularly well-suited for production monitoring and observability systems.

# Motivations

Prior to this project, I had no experience working with sketch algorithms or Prometheus. After reading the DDSketch paper, I was fascinated by how enterprises can represent massive volumes of data using these compact, mergeable sketches. Initially, I was skeptical that a single data structure could deliver accurate quantiles, low memory footprint, and quick queries, so I decided to evaluate it myself. This project focuses on profiling DDSketch along three main areas: memory efficiency, quantile accuracy, and ingestion & query performance. The paper discussed the theoretical advantages of DDSketch, and I wanted to quickly validate those derivations. DDSketch's performance has been benchmarked against alternatives like t-digest in the original paper, so this project focuses on evaluating its performance in real-time ingestion scenarios and streaming settings. **My ultimate goal for this project was to prove that DDSketch is super useful in a sliding window approach for time relevant metrics.** The rest of this report will cover my project milestones, results, challenges, and ideas for future work.

# Implementation (my code can be found here)

## DDSketch Wrapper Design

I built a Go wrapper around the DDSketch implementation in Datadog's official sketches-go library. The goal was to simplify the interface for core sketch operations while maintaining the algorithm's original performance guarantees. Eventually, this version could prove useful in promsketch. The wrapper removed unnecessary configurations and error propagation patterns. I tried returning direct values wherever possible. It also exposed helper methods to retrieve basic stats like count, min, max, and sum, which made it easier to run benchmarks.

## Static and Scenario Testing Setup

Before testing DDSketch in a Prometheus workload, I conducted a static analysis of the algorithm. The promsketch repository had real-world datasets with distributed distributions. I ran simple benchmarking of DDSketch on the Google Cluster dataset. After that, I created a scenario test using three synthetic data distributions: uniform, normal, and exponential. Each test involved inserting 10 million values into a new sketch and measuring its memory usage, query accuracy across standard quantiles. After, I merged the 3 sketches to get overarching quantiles.

## Prometheus Ingestion Pipeline

I built a simplified ingestion pipeline that mimics a production-style Prometheus setup to simulate how DDSketch might behave in an enterprise monitoring system. The pipeline includes a metric exporter generating synthetic values from various distributions, a Prometheus server

scraping from the exporter, and a custom receiver that accepts remote writes and encodes the incoming data into DDSketch structures.

This setup enabled my evaluation of DDSketch under realistic ingestion patterns, with endpoints for quantile summaries and performance metrics. Although it wasn't fully integrated into PromSketch, it served as a functional testbed for validating ingestion logic and sketch accuracy in a Prometheus-compatible environment. I also had trouble computing accurate memory footprints, but this limitation was not that significant since my static analysis did a good job profiling DDSketch memory (see challenges and limitation section).

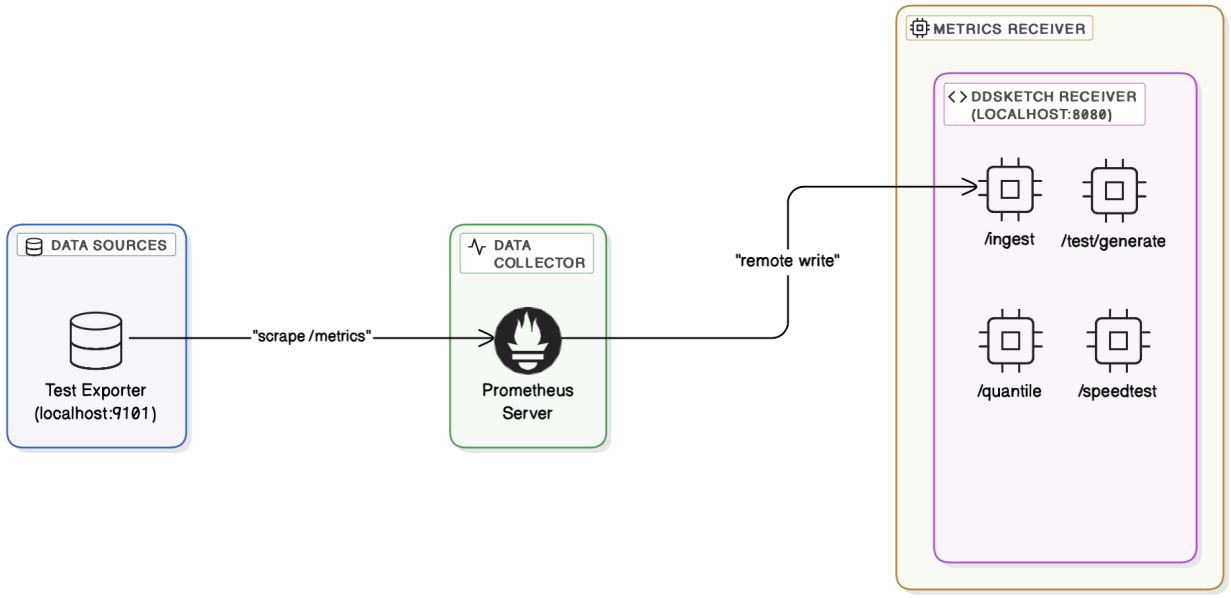


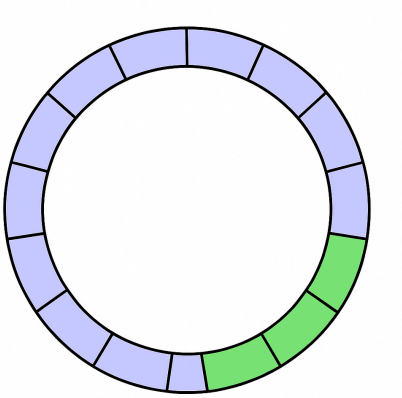

**Figure 1: Prometheus Ingestion Pipeline**

## Windowed DDSketch Proof-of-Concept

I wanted to simulate a real-time sketching system that estimates quantiles over recent streaming data using a sliding window of timesteps. This proof of concept maintains a rotating buffer of DDSketches, where each sketch captures a short interval of incoming values. New values are added to the active sketch, and once the timestep ends, the buffer rotates and begins writing to the next one.

At regular intervals, the system merges a group of recent sketches and computes quantiles over the combined window, and this enables efficient rolling summaries with predictable memory usage and bounded relative error. The configuration parameters are per-sketch window size, number of sketches in the ring buffer, **k** (number of sketches to merge per window), and **n** (total number of values to ingest). The diagram on the next page provides a visual description of sliding window DDSketch.

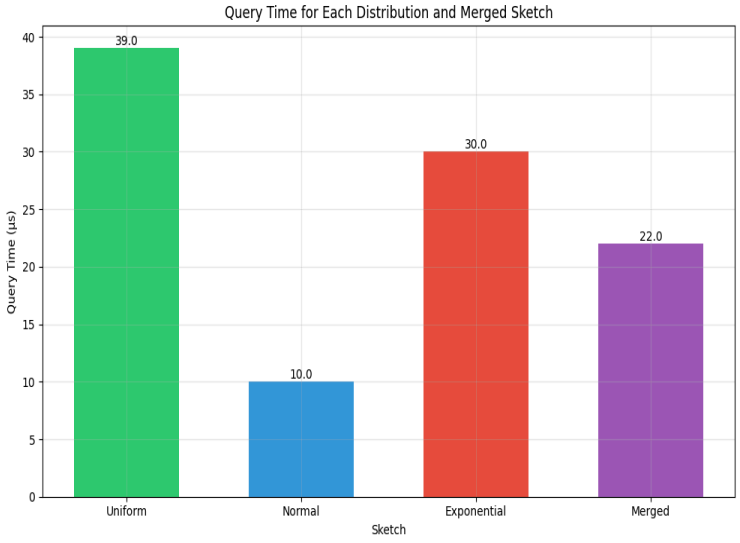
The ring represents a circular buffer that stores DDSketches for the 15 most recent fixed timesteps. Each segment holds one DDSketch corresponding to a single timestep, such as one hour. If we want quantiles over a rolling 3-hour window, the system continuously ingests streaming data into the active DDSketch for the current hour. Once a sketch falls outside the window of interest, it is discarded. At any point, we can merge the sketches from the last three timesteps to compute quantiles over that window. For example, at 7 PM, we can retrieve quantile estimates for data spanning from 4 PM to 7 PM. This approach is memory efficient, accurate, and well-suited for time-sensitive monitoring.



# Results

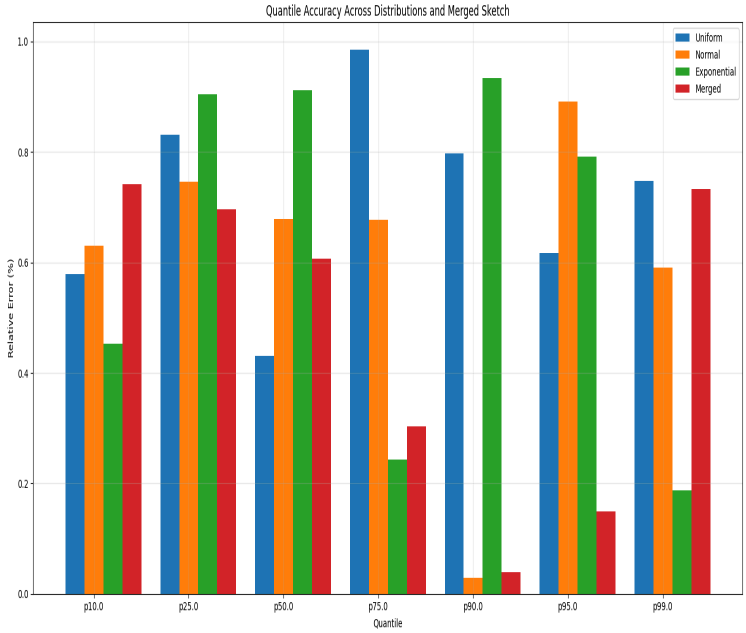
## Scenario Test Results: Accuracy and Query Performance

This section evaluates DDSketch on synthetic datasets representing uniform, normal, and exponential distributions. Each sketch was built from 10 million values, and a fourth sketch was created by merging the three. The evaluation focuses on query latency and quantile accuracy to verify DDSketch's performance under varied data shapes and validate its mergeability guarantees.




This bar chart shows the time (in microseconds) required to query seven standard quantiles from the DDSketches. Query times remain consistently fast, with all values under 40 µs, even for sketches built from 10 million data points. The merged sketch performs on par with or better than the individual sketches, demonstrating that merging has little to no impact on query latency.

**Figure 2: Static Analysis of Query Time**




This graph shows the relative error (%) for each quantile (p10-p99) across the 4 sketches. All sketches maintain low relative error, consistently below the configured 1% threshold. The merged sketch exhibits similar or even slightly better accuracy, confirming that DDSketch preserves its accuracy guarantees even after merging. This supports its use in distributed systems where sketches are aggregated across sources.

**Figure 3: Static Analysis of Quantile Accuracy**

DDSketch achieves over 99.99% memory savings across all distributions. Even the merged sketch representing 30 million values is under 8 KB, confirming that memory usage remains effectively constant regardless of input size. This demonstrates DDSketch's suitability for high-volume, memory-constrained environments.The memory savings are best represented by the results table below and line up with the original DDSketch paper.

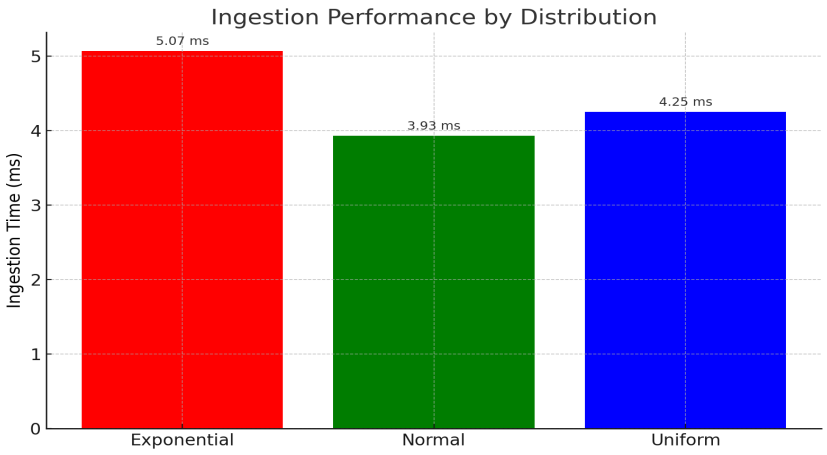
Memory Savings

| Distribution | Values | Sketch Size | Per-Value Memory | Raw Size | Memory Savings |
|---|---|---|---|---|---|
| Uniform | 10,000,000 | 7.09 KB | 0.0007 bytes | 76.29 MB | 99.99% |
| Normal | 10,000,000 | 2.03 KB | 0.0002 bytes | 76.29 MB | 100.00% |
| Exponential | 10,000,000 | 8.06 KB | 0.0008 bytes | 76.29 MB | 99.99% |
| Merged | 30,000,000 | 7.97 KB | 0.0003 bytes | 228.88 MB | 100.00% |

**Figure 4: Memory Savings with DDSketch**

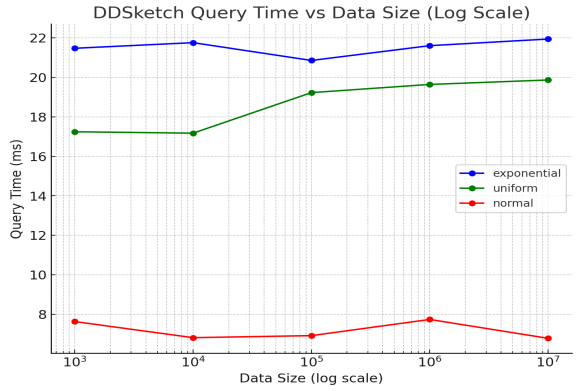
## Prometheus Ingestion Pipeline

Since the sketches in this setup were built the same way as in the static scenario tests, the primary new variable that I wanted to test was ingestion performance when data points were streamed through the Prometheus collector. My goal was to evaluate how efficiently the receiver could process and sketch incoming data points under realistic ingestion patterns.




All ingestion times are under 6 milliseconds for 10 million values, with normally distributed data performing the fastest at 3.93 ms. The results demonstrate DDSketch's ability to handle high-throughput data efficiently, regardless of distribution shape.

**Figure 5: Pipeline Ingestion Performance**



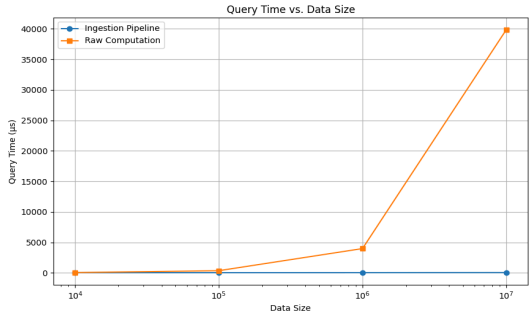




**Figure 6 & 7: Sketch Query Time as Data Size Grows**

I also wanted to see how query performance changed as data size grew. As shown in the results, query time stays roughly constant regardless of the size of the dataset.I think it's interesting that sketches built from normally distributed data consistently return faster query times than those from uniform or exponential distributions (see figure 6). This is probably due to the tighter clustering of values around the mean, which would make quantile estimation more efficient. Additionally, Figure 7 shows that while raw query times grow linearly, DDSketch queries are approximately constant.

## Time-Based Analysis with Sliding Window

I simulated a 24-hour sliding window setup where 100 million data points are ingested per hour. The goal was to maintain a rolling summary of the last 3 hours at any point in time, resulting in a window size of 300 million data points. This configuration (which would typically require 10-30 seconds for quantile computation due to full sorting on a personal computer) was implemented using 24 DDSketches (one per hour) with 3 sketches per sliding window and a total of 2.4 billion data points.

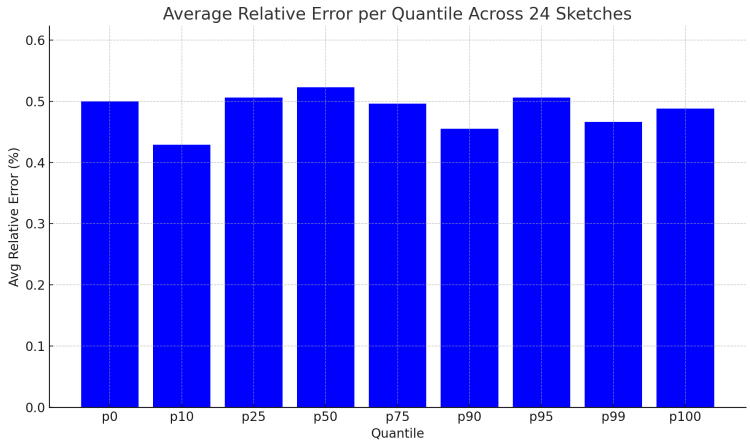



Figure 8 shows the quantile errors averaged over the 24 individual sketches (100M data points each). As expected, the relative errors remain consistently low across quantiles. All of the quantile error averages hover around 0.3-0.6% relative error, which is below the 1% threshold. Error is relatively consistent across quantiles.

**Figure 8: Average Quantile Error for Sliding Window Sketches**

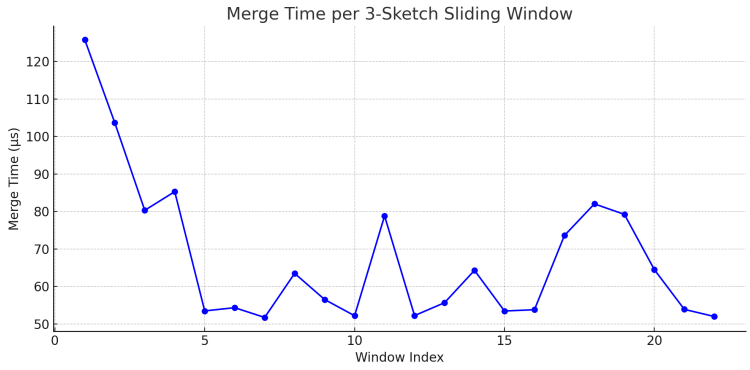



Figure 9 shows the merge time (in microseconds) for each 3-sketch sliding window. Merge times remain consistently low (typically between 50-90 microseconds). This shows that DDSketch merging is a lot more efficient than repeatedly sorting raw values for recent data in addition to having smaller memory footprint (see Figure 4).

**Figure 9: Average Merge Time for k-Sketch Window**

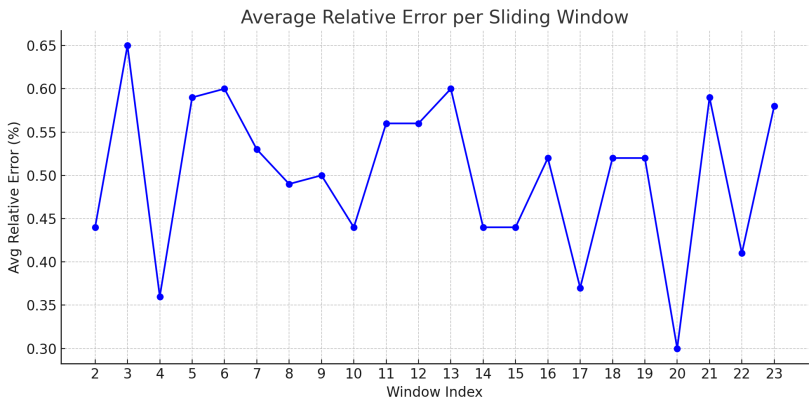



Figure 10 shows the average relative error for each 3-sketch sliding window. While there's no clear trend, every window's average merged sketch quantile accuracy stays between 0.3% and 0.6%. This is below the 1% threshold, showing that merging sketches from different timesteps doesn't degrade overarching quantile accuracy for windows.

**Figure 10: Average Relative Error for Each Window**

## Evaluation Summary

To recap, I tested DDSketch across static datasets, my simple Prometheus ingestion pipeline, and the sliding window setup. All benchmarks were run locally on a 2019 MacBook Pro with a 6-core Intel Core i7 processor, 16 GB of RAM, and macOS. In every case, DDSketch gave fast queries (under 40 µs) and kept quantile errors below 1%. This even applies to the merged sketches, proving that DDSketch doesn't lose quantile accuracy after merges. The static analysis also showed that the sketches had a very lightweight memory footprint (~98%+ memory savings over raw data storage). The sliding window proof of concept showed that I could maintain accurate, low-latency quantile summaries over 2.4 billion points without sorting or storing raw data. I was especially impressed with this due to the data scale and production applications.

## Challenges and Limitations

I ran into two main issues during the project. The first was measuring memory usage in the ingestion pipeline. In the static tests, it was easy to check the sketch footprint since nothing else was running. But in the Prometheus setup, Go's heap had too much background activity, and I couldn't isolate how much memory the DDSketches were using. Since the sketches behave the same regardless of how data arrives, I relied on the static memory measurements, which I think still fairly represent the sketch's footprint.

The second issue was streaming. After a lot of trial and error, I couldn't get Prometheus to scrape from an exporter that streamed continuously instead of exposing data at fixed intervals. I suspect production setups use extra tools to handle this, but I didn't set that up here. To stay

close to how a real Prometheus pipeline works, I prepopulated the exporter with data and had Prometheus scrape from it as usual. The data then went to the receiver, where DDSketches were built. For the same reason, I didn't stream live data for the sliding window DDSketch analysis. Instead, I used large synthetic datasets to simulate how well the setup would scale.

## Conclusion & Future Work

This project convinced me that DDSketch is a great fit for large-scale, time-sensitive monitoring workloads. It's fast, memory efficient, and accurate even when sketches representing huge data volumes are merged. These are exactly the kinds of properties you'd want in real-world monitoring systems.

There are a few areas I'd explore if I had more time. First, I'd try adding real streaming support to the pipeline using something like OpenTelemetry, since Prometheus alone can't scrape from exporters that emit data continuously. I also think it would be useful to visualize the sliding window results through a simple dashboard to see how quantiles change over time more intuitively.

Another direction for future work is CPU benchmarking. Right now, all the testing was done on my personal computer with a standard Intel chip. Running it on less powerful hardware or in containerized environments would help test whether DDSketch stays efficient under tight resource constraints. On the flip side, I'd love to run it on higher-end systems to explore the upper limits for ingestion speed and sketch memory usage.

Before this project, I had never worked with sketch algorithms or Prometheus. Now, I understand how a well-designed sketch algorithm can make real-time analysis scalable and practical.