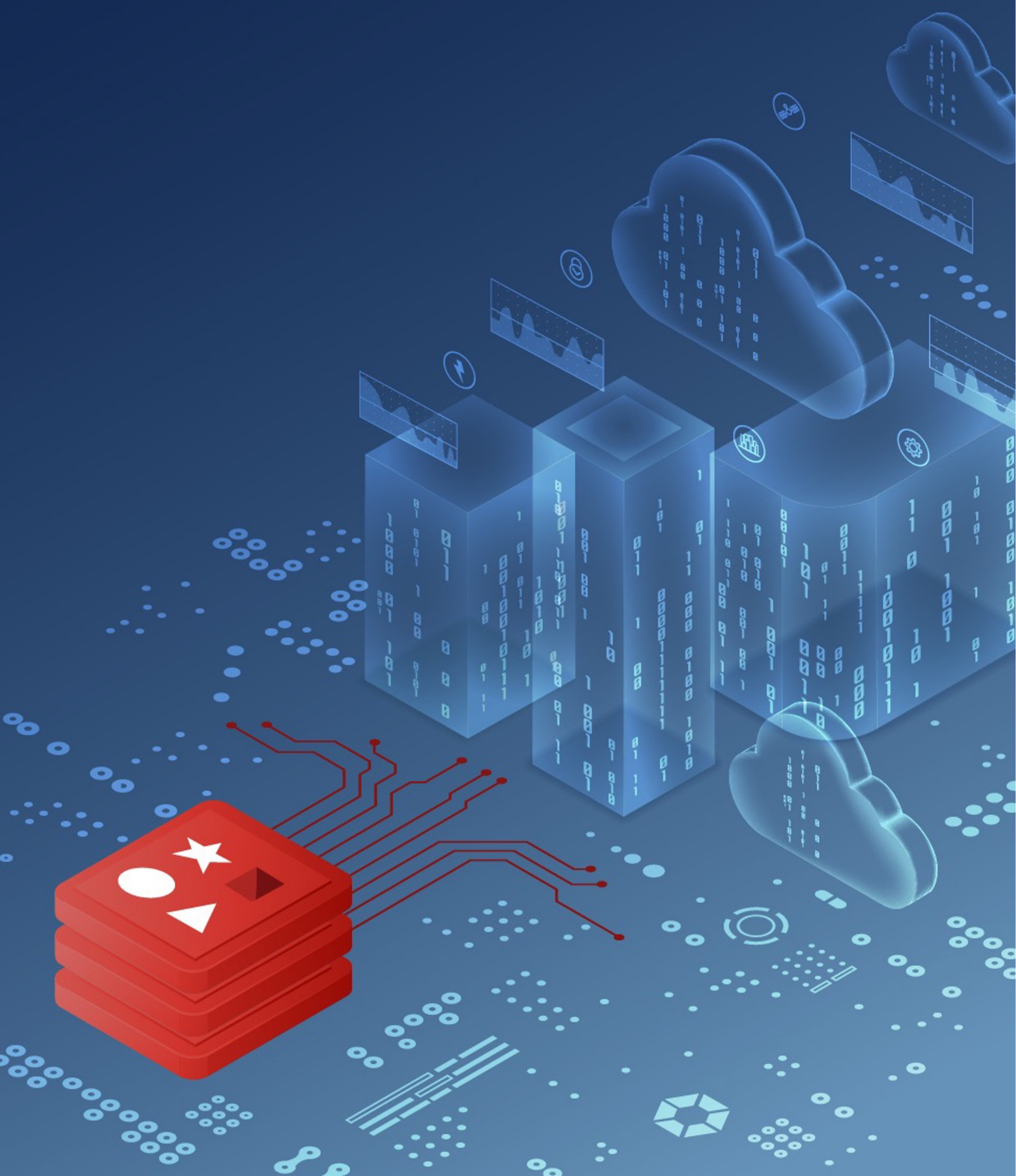
redis

Optimizing Data Systems

for NVIDIA Merlin and Triton

March 2023



Sam Partee
Principal Engineer – Redis – Applied AI

Tyler Hutcherson
Senior Engineer – Redis – Applied AI

Agenda

Table of Contents

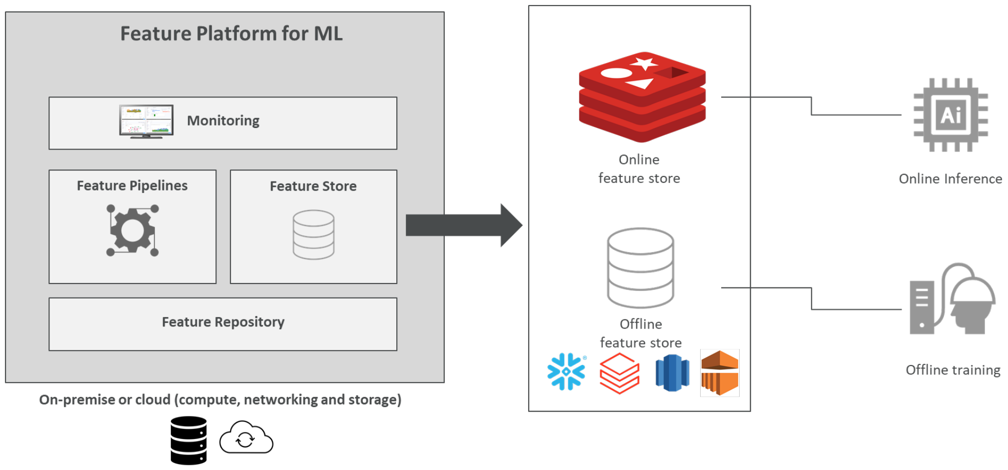
- Introduction to Redis in ML Pipelines
- Recommender System Pipelines with **Redis** and **NVIDIA Merlin**
  - Introduction
  - Benchmarking setup
  - Baseline architecture and performance
  - Data pipeline optimizations
    1. Ensemble stage consolidation
    2. Direct Redis communication
    3. Vector search feature retrieval
    4. Inference caching
- Performance Summary and Next Steps

Redis in ML Pipelines

Redis for Real-Time ML Data

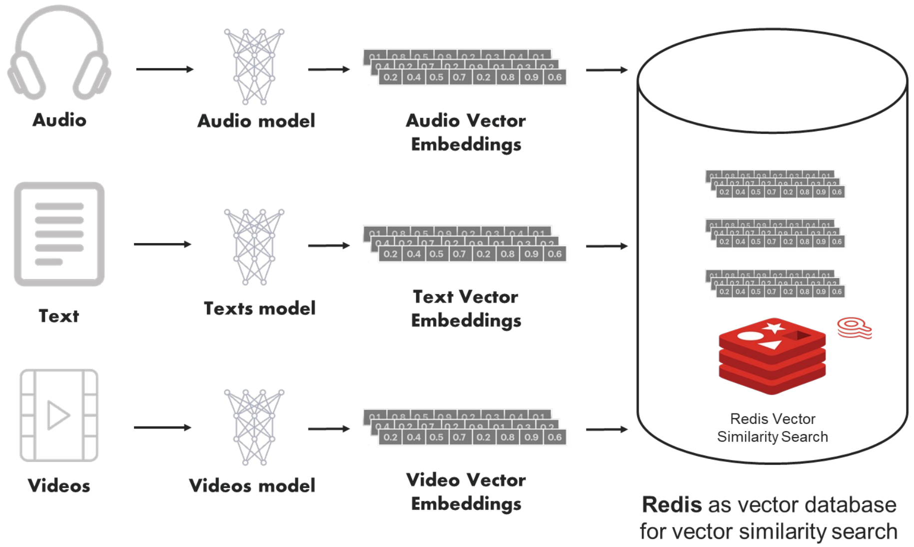
Single-digit millisecond Feature Retrieval

Redis Feature Store

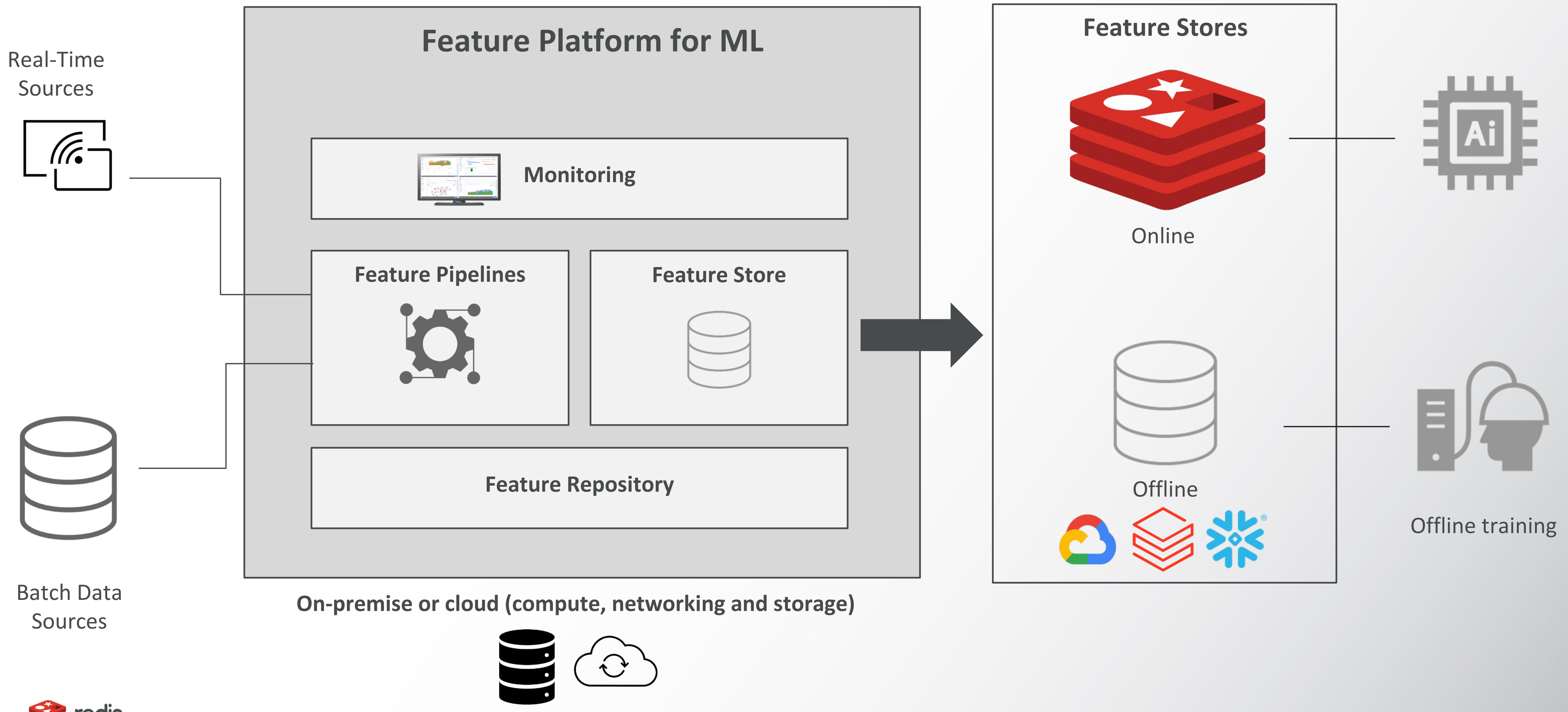


A Composable Platform for Intelligent Applications

Redis Feature Store for Vectors



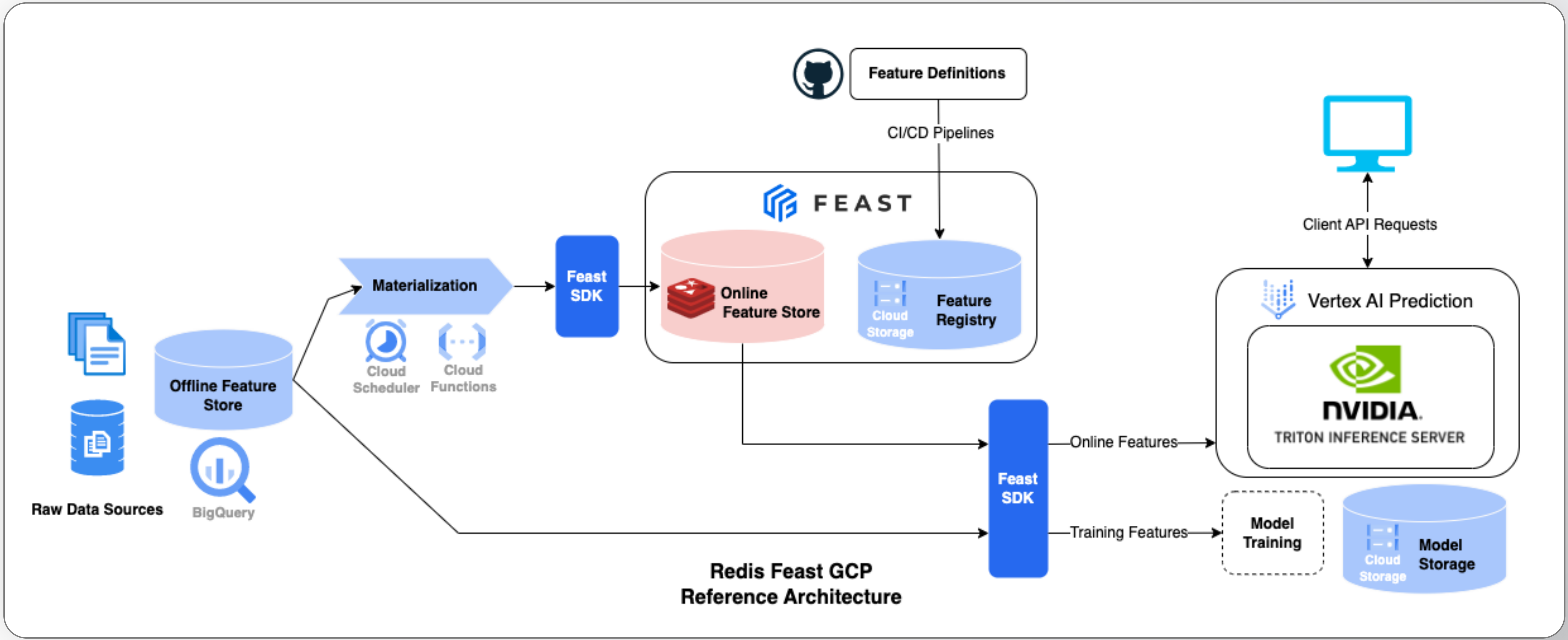
Redis – Online Feature Store



redis

Feature Store – GCP Reference Architecture

End-to-end feature pipeline using Redis, Triton, and Feast on GCP

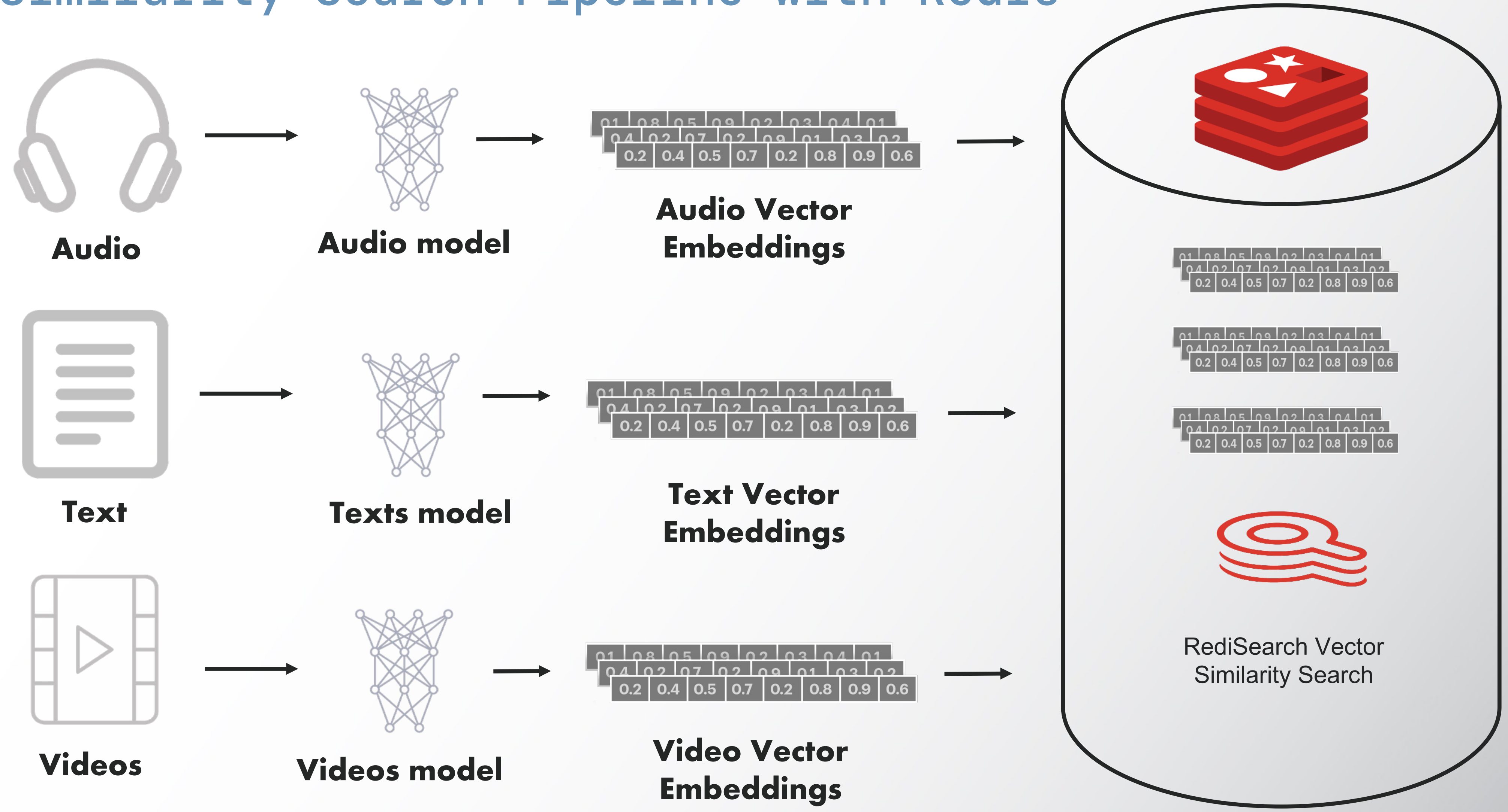


redis

Code: https://github.com/RedisVentures/redis-feast-gcp

Redis as a Vector Database

Vector Similarity Search Pipeline with Redis



Redis – Vector Similarity Search

Feature Set

Redis Vector Similarity Search – RediSearch

- **Redis**: Low-latency, scalable, in-memory database
- Indexing methods
  - HSNW (ANN)
  - Flat (KNN)
- Distance metrics
  - L2, Cosine, internal product
- Support for hybrid queries
  - Vector search + filtering by text, geo, etc.
- Store vectors in JSON (new in 2.6)

Recommender System Pipelines with Redis and NVIDIA Merlin

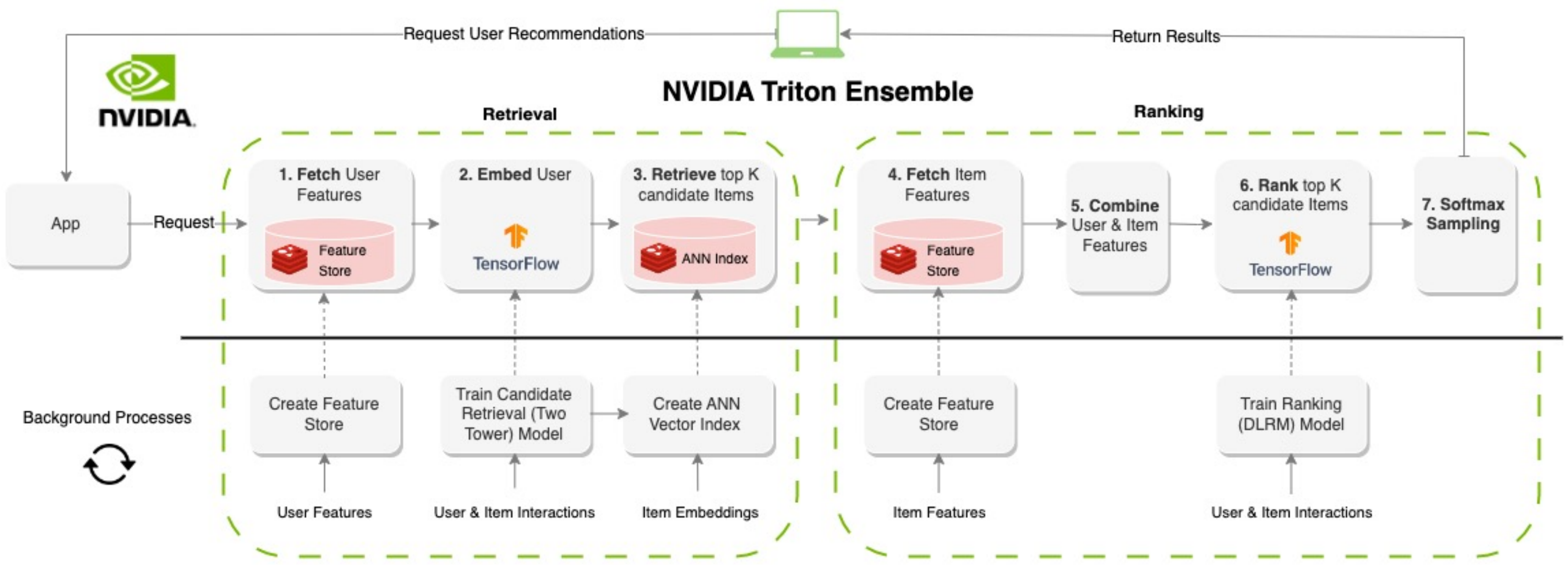
Data Systems – Multi-stage Recommender

Introduction

- Two Primary Stages
  - **Retrieval** – emphasis on speed and efficiency. A relevant subset is selected from a large pool of potential candidates. ANN algorithms commonly used.
  - **Ranking** - emphasis on precision and accuracy. Models are more computationally complex (e.g. DLRM) and rank subset of items based on likelihood of user interaction.
- Serving in Real-Time
  - **Triton Ensemble** enables multi-stage DAG processing per inference call. Variety of supported backends (e.g. Tensorflow, PyTorch, FIL, Python)
  - Low latency & high throughput is key.
- Merlin Building Blocks
  - **NVTabular –** Distributed GPU Data Processing
  - **Triton –** Model Serving
  - **Merlin Systems –** Utilities for RecSys
  - **HugeCTR** - Distributed RecSys Model Library

Data Systems – Multi-stage Recommender

Architecture - Baseline



7 Stages of RecSys Pipeline

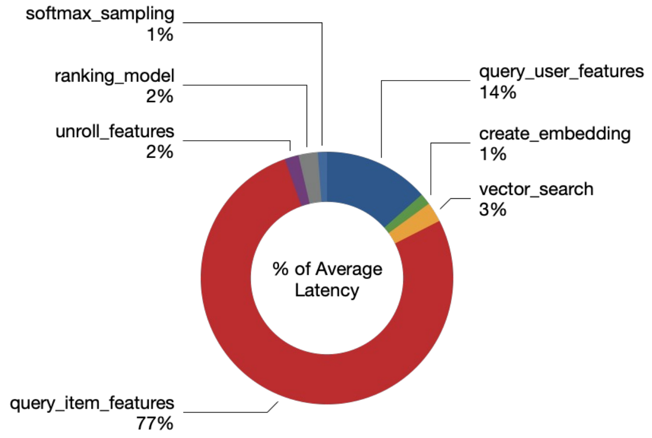
1. User Feature Retrieval
2. Create User Embedding
3. Vector Search for TopK Items
4. TopK Item Feature Retrieval
5. Unroll Features (processing)
6. Ranking Model (DLRM)
7. Softmax Sampling

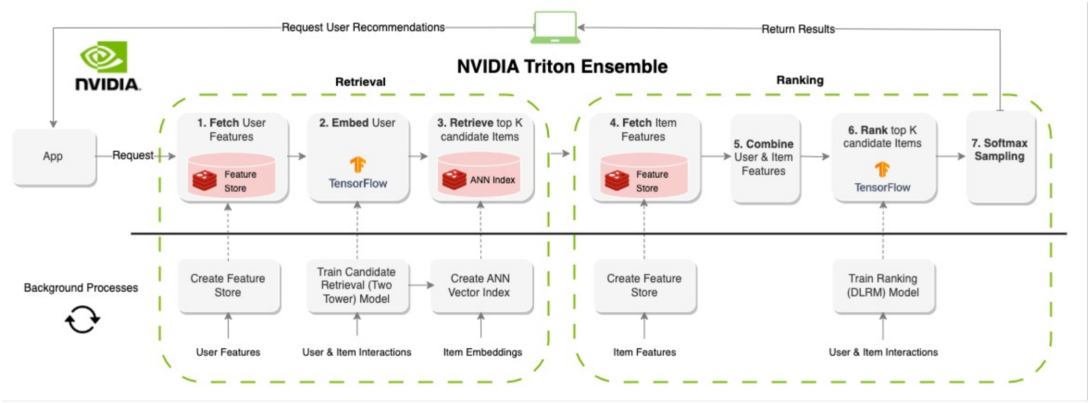
Code: https://github.com/RedisVentures/Redis-Recsys

Data Systems – Performance

Performance - Baseline

- **Importance of baseline**
  - Don't start with model optimizations
  - Ensure data pipeline first
- **Approach to Optimization**
  - Start with smaller models
  - Over-emphasize data pipeline
  - Optimize data movement
  - Then scale and optimize models
- **Feature retrieval accounts for** ~90% of latency.



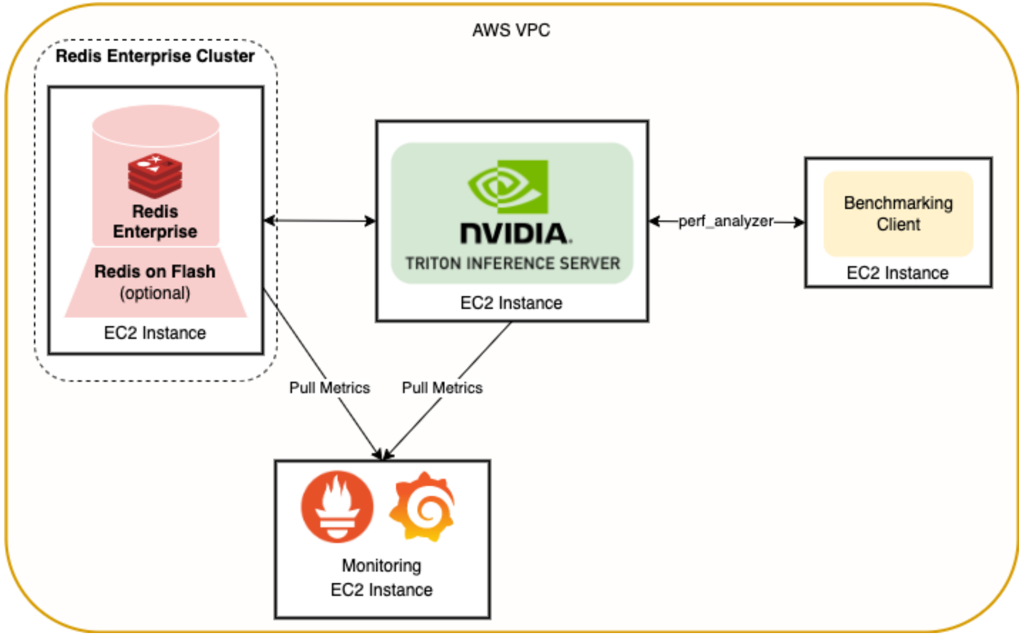


redis

Data Systems – Multi-stage Recommender

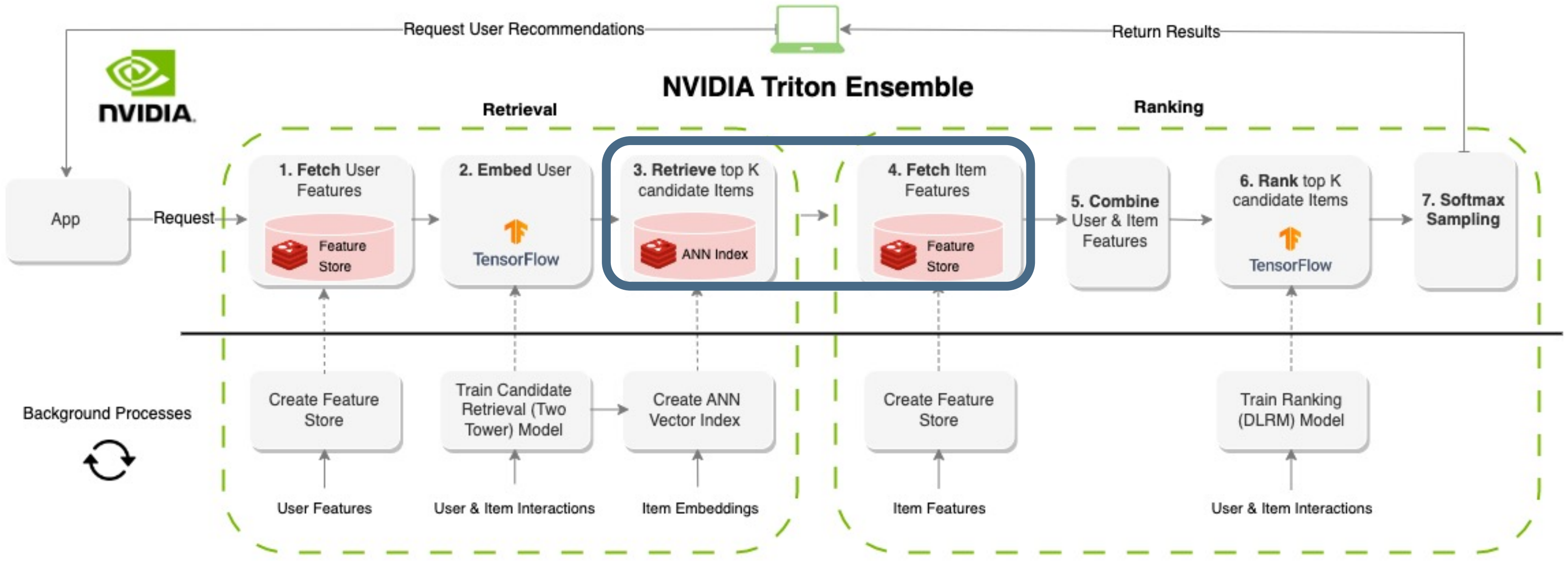
Benchmarking Setup

- **Triton Inference Server**
  - Instance: g4dn.xlarge, 4 vCPU, T4 16Gb GPU
- **Redis Enterprise Database**
  - Instance: i3.xlarge, 4 vCPU, 32Gb RAM
  - Shard Count: 1 master, 1 replica (highly available)
- **Benchmarking Client with Perf Analyzer**
  - Instance: t2.2xlarge, 8 vCPU
  - Concurrency: 16
- **Grafana & Prometheus**
  - Instance: t3.micro, 2 vCPU



Data Systems – Performance

Architecture - Optimization 1 - Stage Consolidation

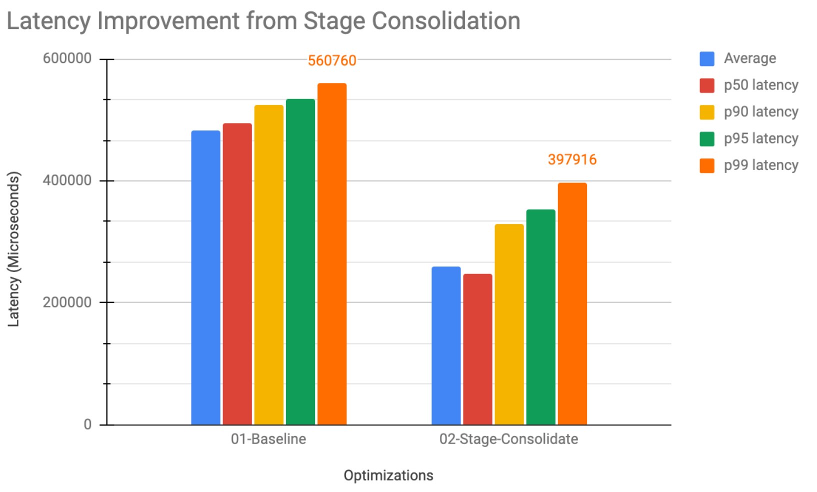


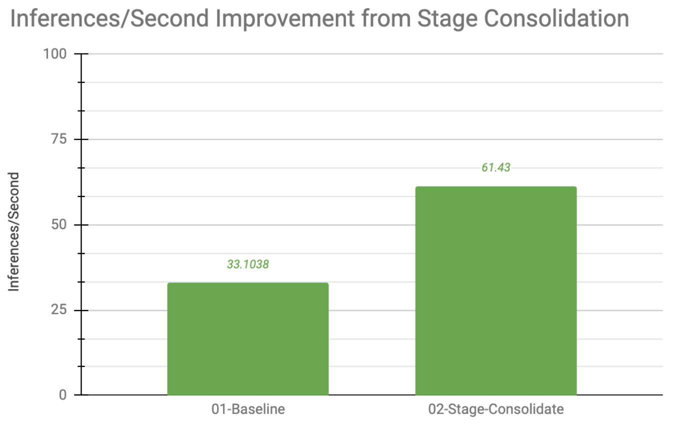
6 Stages of RecSys Pipeline

1. User Feature Retrieval
2. Create User Embedding
3. VSS and Item Feature Retrieval
4. ~~TopK Items Feature Retrieval~~
5. Unroll Features (processing)
6. Ranking Model
7. Softmax Sampling

Data Systems – Performance

Performance - Optimization 1 - Stage Consolidation





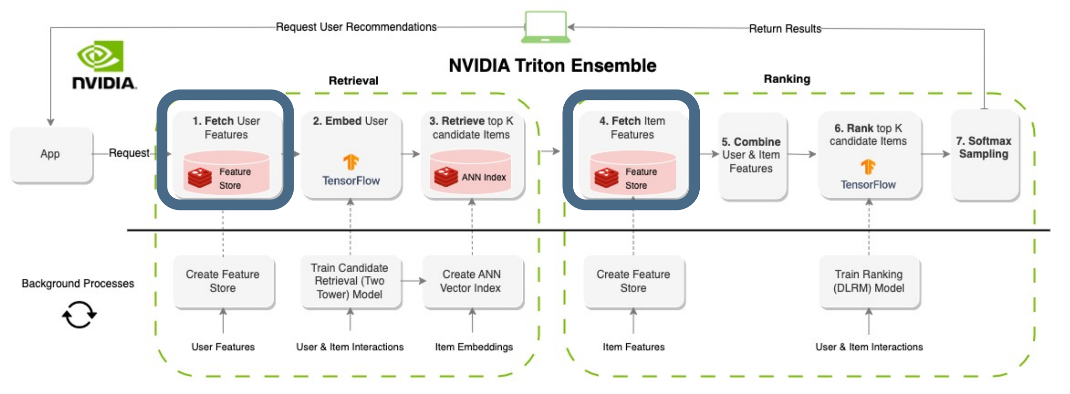
Results of Stage Consolidation

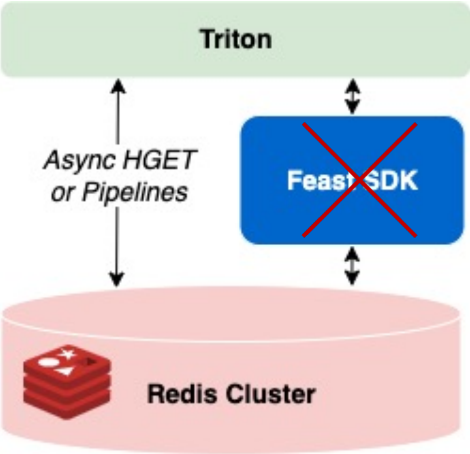
- Change: Reduce number of ensemble stages by one, combining VSS and item retrieval
- Compared to baseline
  - 85.57% increase in throughput
  - 46.15% decrease in Avg latency
  - **29.22%** decrease in p99 Latency

redis

Data Systems – Performance

Architecture - Optimization 2 - Direct Redis Communication (Remove Feast)





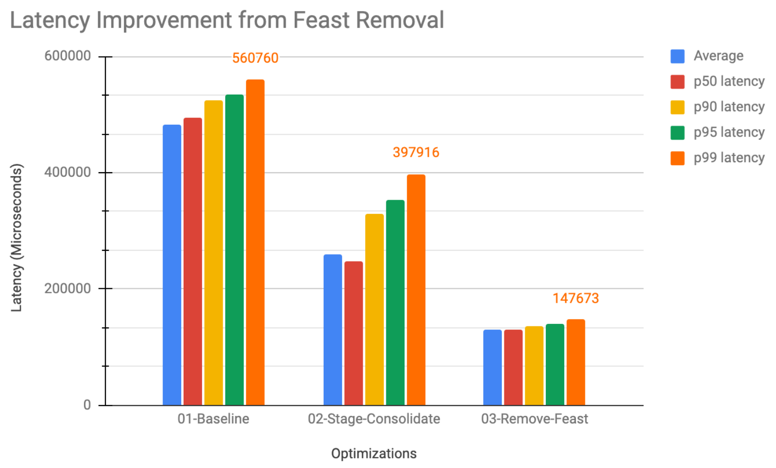
Feast is useful for feature management, orchestration, and cataloging.

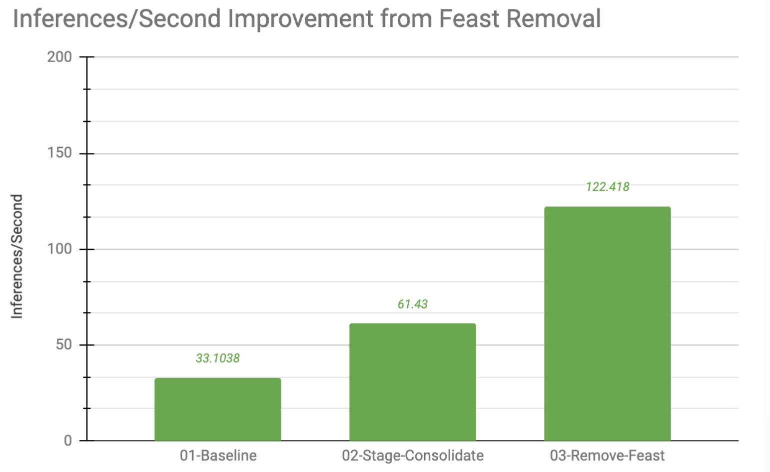
Drawbacks

- Serialization can add too much overhead for high-throughput applications. (protobuf)
- Direct communication with Redis client allows for async calls and pipelines.
- Enables future optimizations in combining feature retrieval and vector search.

Data Systems – Performance

Performance - Optimization 2 - Direct Redis Communication (Remove Feast)





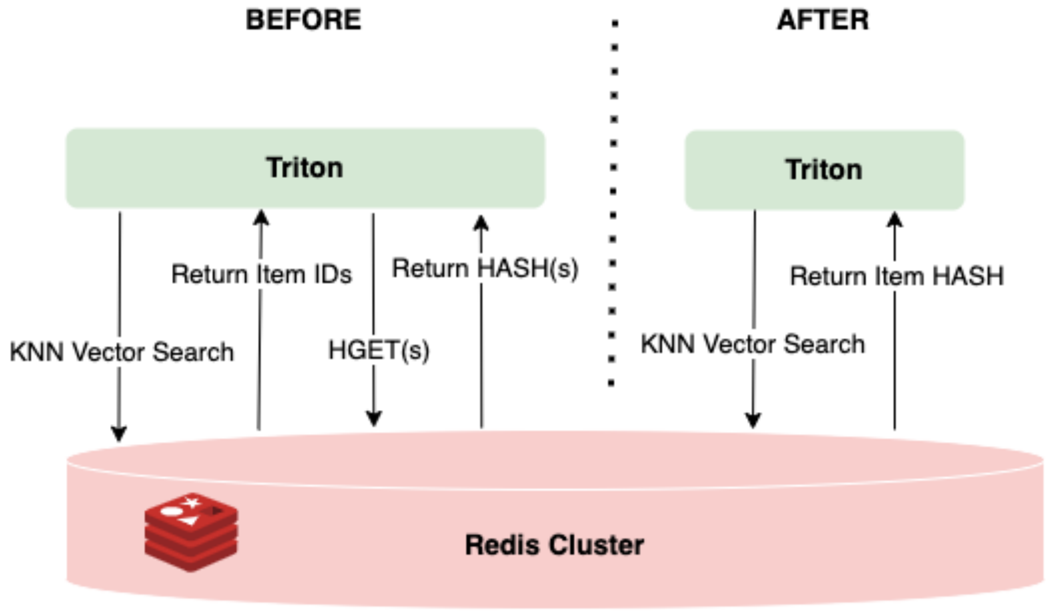
Results of Feast Removal

- Change: **Remove Feast SDK Layer**
- Item features retrieved by Redis client directly communicating with Redis
- Compared to previous optimization
  - 99.28% increase in throughput
  - 49.81% decrease in Avg latency
  - **62.88%** decrease in p99 Latency

redis

Data Systems – Performance

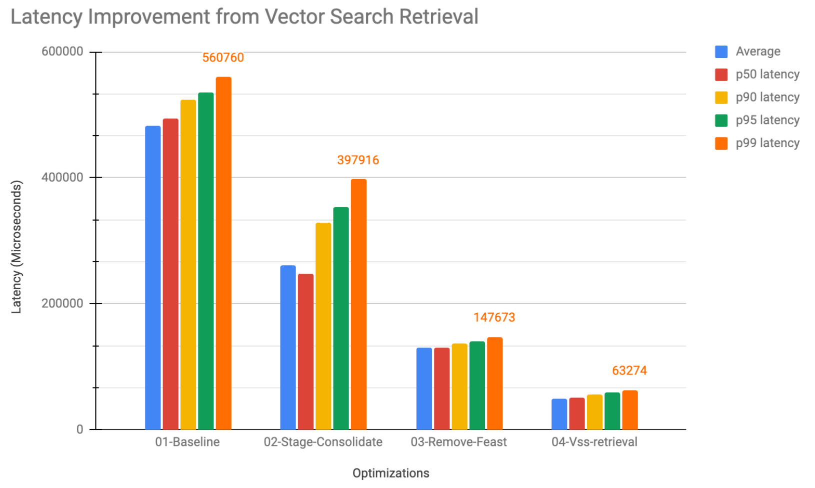
Architecture - Optimization 3 - Vector Search Retrieval of Item Features

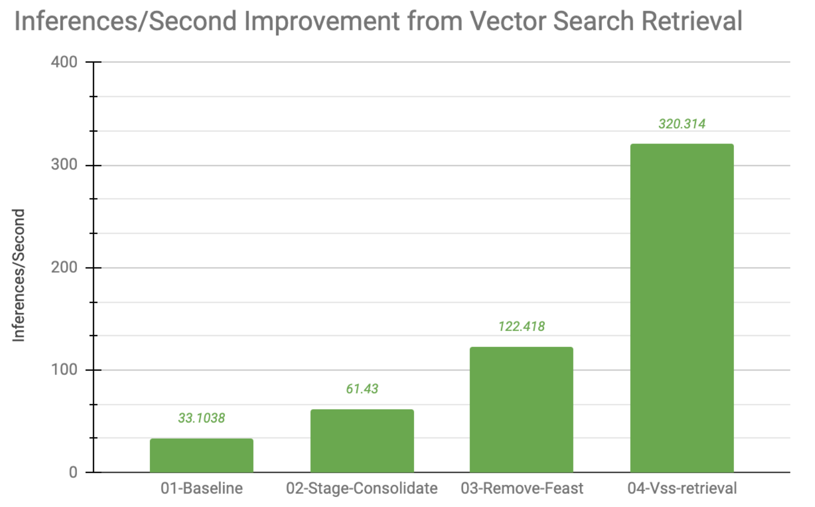


- Built-In hash Retrieval
  - **RediSearch** includes hash data in response object.
  - Eliminates need for separate database calls to retrieve item features for ranking
- Enabled by using Redis hash instead of Feast protobuf format
- RediSearch can also use JSON format instead of HASH

Data Systems – Performance

Performance - Optimization 3 - Vector Search Retrieval of Item Features





Results of Vector Search Feature Retrieval

- Change: **Item features retrieved by vector search instead of HGETALL**
- K–1 reduction in calls to Redis where K is the number of items retrieved.
- Compared to previous optimization
  - 161.66% increase in throughput
  - 61.77% decrease in Avg latency
  - 57.15% decrease in p99 Latency

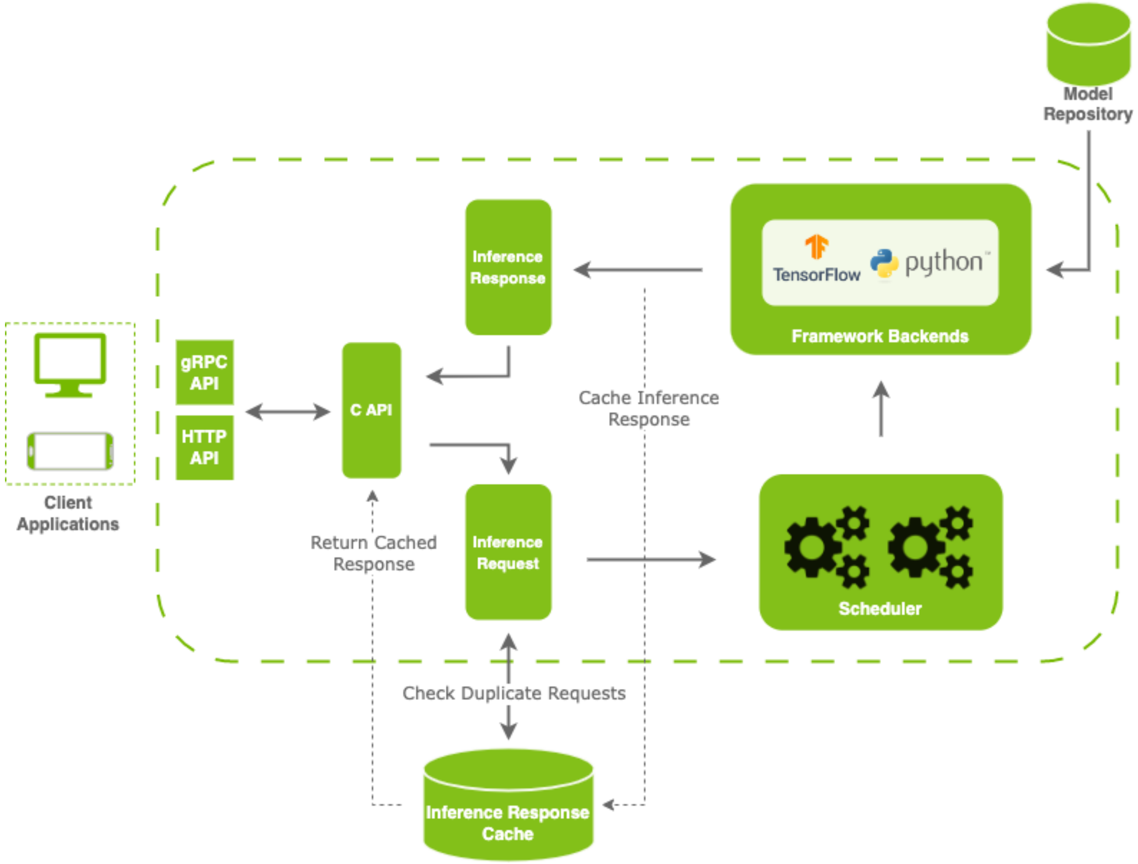
redis

Data Systems – Performance

Architecture - Optimization 4 - Caching Inference Requests

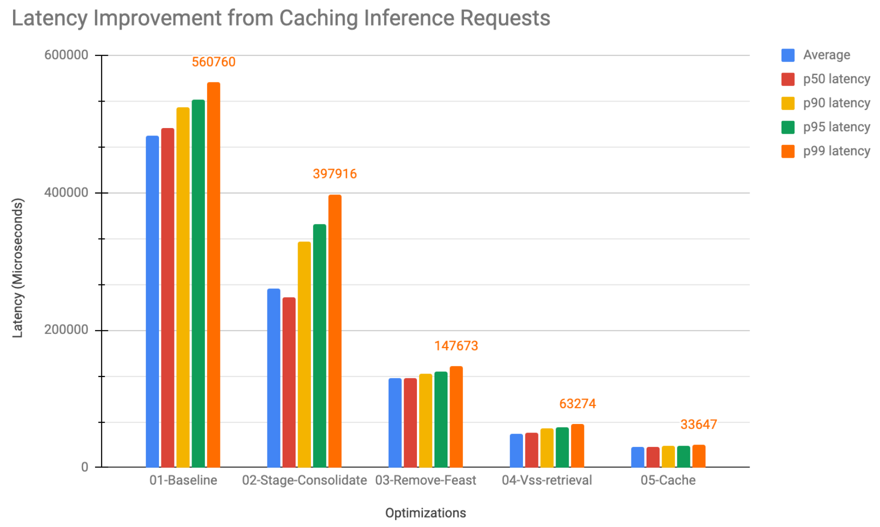
- Triton inference Response Cache
- Enabled in ensemble model configuration
- Cached Stages in Ensemble
  - User feature
  - VSS and Item feature retrieval

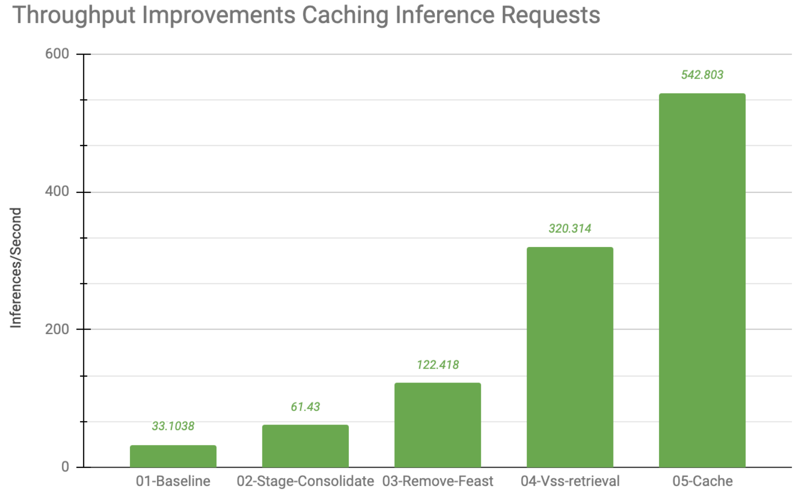
```
dynamic_batching {}
response_cache {
  enable: True
}
instance_group [{ kind: KIND_CPU, count: 4 }]
```



Data Systems – Performance

Performance - Optimization 4 - Caching Duplicate Requests





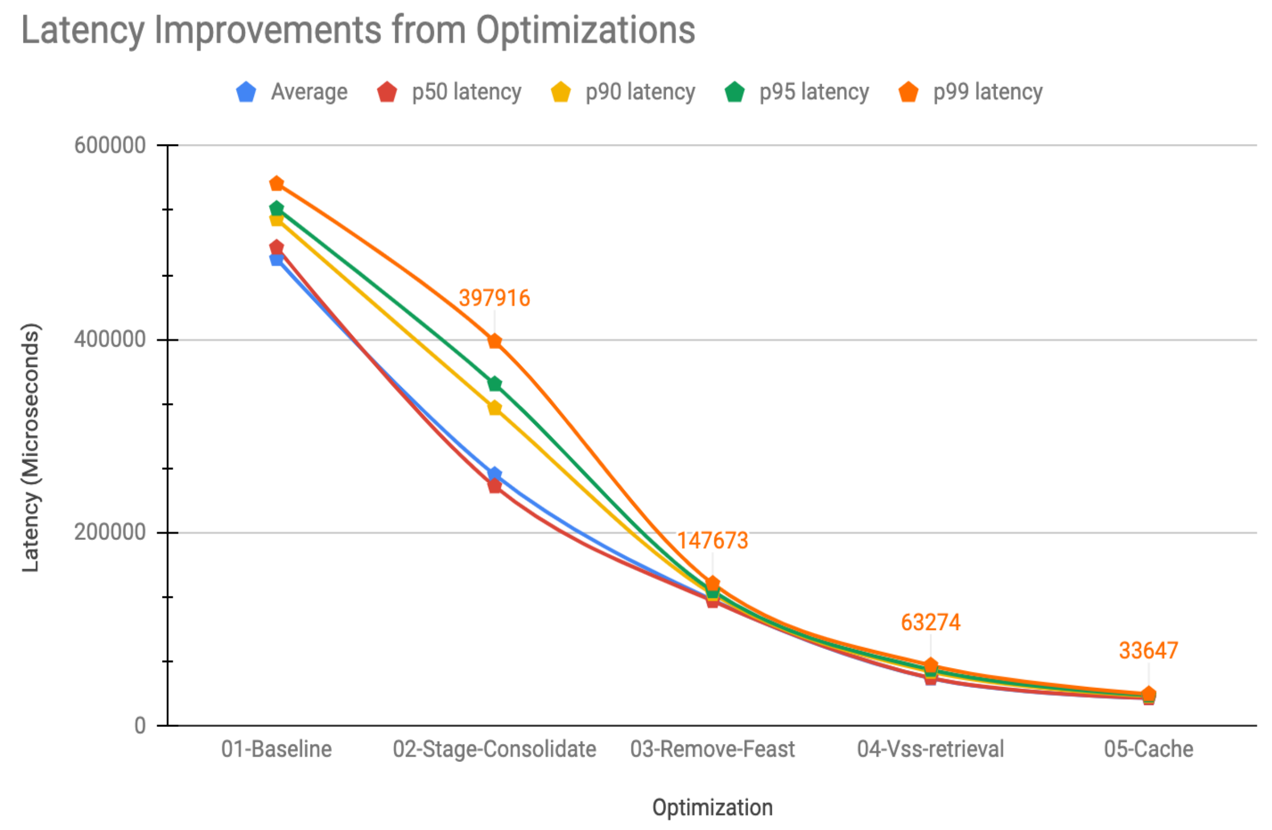
Results of Optimization

- Change: **Caching inference requests**
- ***All duplicate requests***
  - Cached retrieval stage. **100% cache hits for test**
  - Increase in throughput for duplicate requests
- Benefits (for duplicate requests)
  - 69.5% increase in throughput
  - 40.98% decrease in Average latency
  - 46.82% decrease in p99 Latency

redis

Data Systems – Performance Summary

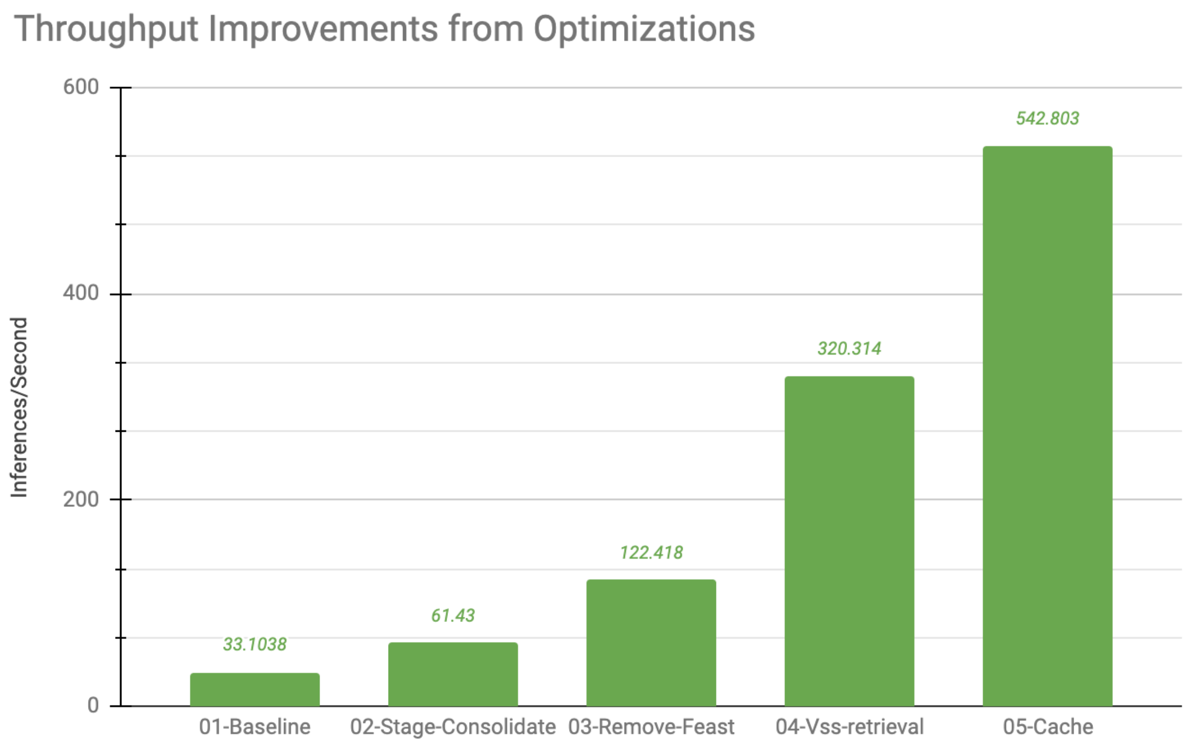
Performance Improvements from Data Pipeline Optimizations

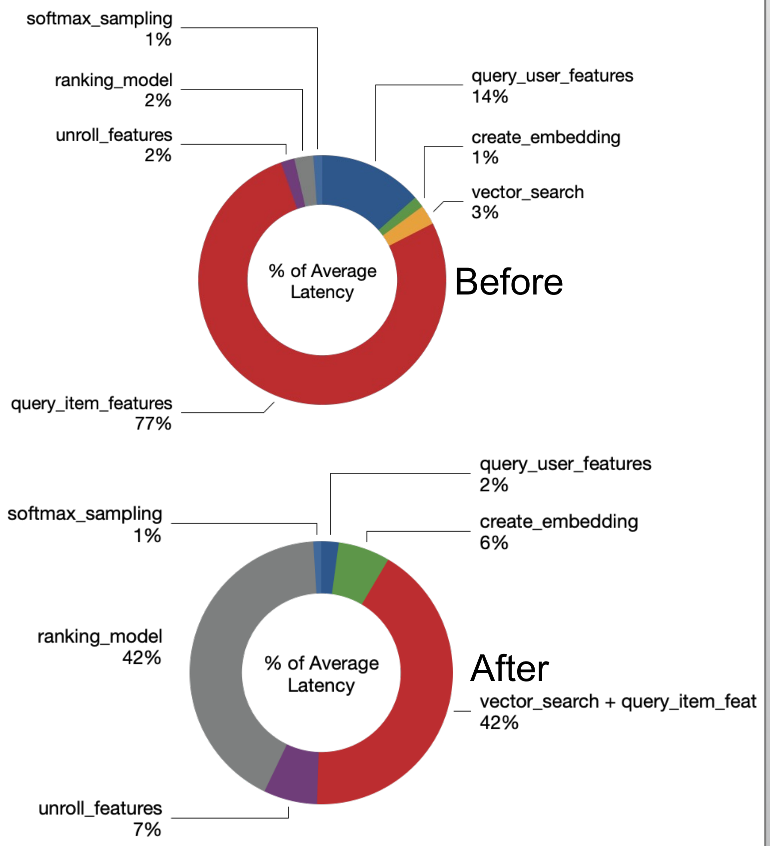


redis

Data Systems – Performance Summary

Performance Improvements from Data Pipeline Optimizations





redis

Data Systems – Performance Summary

Performance Improvements from Data Pipeline Optimizations

- Optimization approach: Improve data system prior to increasing model capacity/performance
- Performance Improvements over Baseline
  - Avg Latency: **88.72% decrease** (94.01% for duplicate requests)
    - 483ms to 49ms
  - Throughput: ~**9x inferences/second** (~16x for duplicate requests)
    - 33 infer/sec to 320 infer/sec
- **7.3x GPU Utilization** compared to baseline
- Future Work - Redis-based Triton Response Cache

Code: https://github.com/RedisVentures/Redis-Recsys

Thank you!

For more information please contact: sam.partee@redis.com or follow on Twitter @sampartee