CSC490 Assignment 5

Team:
Son Nguyen (ID 1009656560), Kyle (ID 1007785229),
Daniel (ID 1008035378), Jinbo (ID 1004821419)
Project: ArXplorer - Semantic Search for arXiv

Part One

Our goals were:

- Measure where our system actually spends time.
- Identify the main bottlenecks in the embedding and search pipeline.
- Propose concrete optimizations we can implement in later parts or future work.

We focused on the embedding, indexing, and search components, since these are the core of our LLM-based semantic search system.

Profiling was done with Python 3.10 on Windows using the script:

- a5/profile_a5_part1.py

Each profiling run produced a ".prof" file, which we saved under:

- a5/profiling_results/

---

2. Functions Profiled

We profiled the following five functions:

1. _encode_text()
   - Module: src.core.pipeline.EmbeddingGenerator
   - Reason: Lowest-level "text → embedding" operation, used throughout the system.
2. generate_embeddings()
   - Module: src.core.pipeline.EmbeddingGenerator
   - Reason: Generates full embeddings for a ProcessedPaper (title + abstract, etc.). Called once per paper.
3. build_index()
   - Module: src.core.pipeline.VectorIndexer
   - Reason: Builds the FAISS vector index that powers semantic search.
4. search_papers_by_text() (mocked)
   - For Part 1, implemented via a FakeRepo inside profile_a5_part1.py.
   - Reason: Represents the search-by-text API path without requiring a real MongoDB setup.
5. get_papers_by_categories() (mocked)
   - Also implemented via FakeRepo.
   - Reason: Represents category-based search latency and async overhead.

For the last two functions we intentionally used a fake repository instead of a real MongoDB client. The goal in Part 1 is to profile code paths and runtime behavior, not to debug database configuration.

---

3. Profiling Setup

We used Python's cProfile module. The core helper in profile_a5_part1.py looks like this:

Function run_profile(label, func_str):

- Prints a header for the profiling section.
- Builds a filename such as "profile_encode_text_YYYYMMDD_HHMMSS.prof".
- Calls cProfile.run(func_str, filename).
- Loads the profile file into pstats.Stats and prints the top 20 entries, sorted by total time (tottime).

Example usage:

- run_profile("encode_text", "profile_encode_text()")
- run_profile("generate_embeddings", "profile_generate_embeddings()")
- run_profile("build_index", "profile_build_index()")
- run_profile("search_text", "asyncio.run(profile_search_by_text())")
- run_profile("search_category", "asyncio.run(profile_search_by_category())")

We kept only the latest profile file per function (to avoid clutter) and used the console output (top 20 lines sorted by to ttime) for our analysis.

---

4. Profiling Results (Per Function)

4.1 _encode_text()

Profile file: profile_encode_text_… .prof
Total runtime (latest run): approximately 0.73 seconds

Role:

- Takes a single string (e.g., title or abstract).
- Tokenizes it with a HuggingFace tokenizer.
- Runs a transformer model (sentence-transformers/all-MiniLM-L6-v2).
- Returns one embedding vector.

Top bottlenecks (by tottime):

- SSL socket reads (method 'read' of '_ssl._SSLSocket'): about 0.17 seconds on the measured run.
- File I/O for tokenizer and model files (nt.stat, io.open, io.open_code).
- torch._C._nn.linear (transformer dense layers).
- JSON parsing for model configuration.

Interpretation:

- The first run is dominated by model and tokenizer loading (and sometimes remote cache access via SSL).
- After the model is fully loaded and cached, subsequent calls are much faster and dominated by transformer fo rward pass.
- This function is fundamental; any optimization here improves all higher-level embedding functions.

4.2 generate_embeddings()

Profile file: profile_generate_embeddings_… .prof
Total runtime (latest run): approximately 0.35 seconds

Role:

- Takes a ProcessedPaper object containing:
    - cleaned_title
    - cleaned_abstract
    - extracted_keywords, etc.
- Calls _encode_text() for the relevant fields (usually title + abstract).
- Produces a joint embedding representation for the paper.

Top bottlenecks:

- SSL reads (on some runs) and transformer model state loading (especially for the first call).
- torch._C._nn.linear and torch.layer_norm.
- A few JSON decode calls and model loading functions.

Interpretation:

- generate_embeddings() is essentially a small wrapper that calls _encode_text() multiple times.
- After warm-up, most cost comes from the transformer forward passes.
- Around 0.35 seconds per paper on CPU is acceptable for small datasets, but becomes significant for large-scal e indexing.

Importance:

- This function is called for every paper we want to index.
- It directly impacts offline preprocessing time and the index-building pipeline.

4.3 build_index()

Profile file: profile_build_index_… .prof
Total runtime (latest run): approximately 2.09 seconds for a test batch of 50 papers.

Role:

- Generates dummy ProcessedPaper objects.
- Uses EmbeddingGenerator to create embeddings.

- Builds a FAISS index (e.g., FLAT index) over these embeddings.

Top bottlenecks:

- torch._C._nn.linear: about 0.81 seconds.
- torch.layer_norm: about 0.15 seconds.
- torch._C._nn.gelu: about 0.10 seconds.
- torch._C._nn.scaled_dot_product_attention.
- BERT model forward passes in transformers (modeling_bert.py).

Interpretation:

- Most time is spent generating embeddings, not inside FAISS itself.
- FAISS index creation is relatively cheap for 50 vectors, but embedding generation dominates.
- As the number of papers grows, build_index() becomes a combination of "embedding many papers" plus "FA ISS indexing".

Importance:

- Determines how fast we can build or rebuild our semantic index.
- For production, this would be part of an offline pipeline that we want to make efficient.

4.4 search_papers_by_text() (Mocked)

Profile file: profile_search_text_… .prof
Total runtime (latest run): approximately 0.002 seconds

Role in Part 1:

- In profile_a5_part1.py, we define a FakeRepo with an async method search_papers_by_text.
- The method simply returns a list of fake documents with a given limit (e.g., 10 items).
- We then profile asyncio.run(profile_search_by_text()).

Key observations from the profile:

- Almost all time is spent in asyncio event loop functions:
    - base_events.run_until_complete
    - windows_events._poll
    - socketpair and overlapped I/O operations.
- Our own application logic (creating a list of dictionaries) is negligible.

Interpretation:

- This setup measures the overhead of the async infrastructure rather than real DB latency.
- In a real deployment with MongoDB, the major cost would come from:
    - Network I/O to MongoDB.

- Query execution and index usage.

Reason for mocking in Part 1:

- Avoids failure due to missing MongoDB collections or text indexes.
- Keeps the focus of Part 1 on profiling Python code, not database configuration.

4.5 get_papers_by_categories() (Mocked)

Profile file: profile_search_category_… .prof
Total runtime (latest run): approximately 0.003 seconds

Role in Part 1:

- Similar to search_papers_by_text(), we use a FakeRepo with get_papers_by_categories.
- The mock method returns a list of fake documents where "category" is set to the first requested category.

Key observations:

- Again, most of the time shows up inside asyncio event loop internals.
- The user-level code is almost free compared to the async overhead.

Interpretation:

- As with the text search case, the real bottleneck in a production environment would be MongoDB queries and indexes, not the wrapper function.

---

5. Cross-cutting Observations

---

1. Model loading vs. steady-state performance

- The first call to embedding functions (_encode_text and generate_embeddings) is relatively slow due to:
    - Download and caching of the HuggingFace model.
    - Tokenizer and configuration file loading.
- After warm-up, runtime is dominated by the transformer forward pass in PyTorch.

2. Embedding computation dominates index building

- In our build_index test, the bulk of the time is spent calling generate_embeddings for 50 papers.
- FAISS index construction itself is relatively cheap at this scale.

3. Async search overhead is small

- With the FakeRepo, search_papers_by_text and get_papers_by_categories complete in a few milliseconds.
- The overhead is almost entirely the asyncio event loop.
- Once we incorporate MongoDB, the main additional cost will be round trips and query execution.

---

6. Optimization Ideas

Based on the profiling results, we identified several optimizations:

6.1 Cache tokenizer and model

Problem:

- If EmbeddingGenerator re-initializes models or tokenizers frequently, we pay repeated initialization costs.

Idea:

- Ensure the tokenizer and model are loaded once in EmbeddingGenerator.**init** and reused across all calls.
- Optionally, treat EmbeddingGenerator as a singleton in the application layer so the model is only created once per process.

Expected benefit:

- Removes expensive re-initialization on repeated requests.
- Reduces cold-start overhead in production.

6.2 Batch embedding for title and abstract

Problem:

- generate_embeddings calls _encode_text multiple times for different fields of the paper (title, abstract, etc.).

Idea:

- Use batch encoding with the underlying model (e.g., encode [title, abstract] in one call).
- For bulk operations like indexing, also process multiple papers per batch.

Expected benefit:

- Better CPU/GPU utilization and less Python overhead.
- Potentially 1.5x or better speedup for per-paper embedding generation.

6.3 Avoid rebuilding FAISS index from scratch

Problem:

- build_index currently assumes we generate embeddings and build the FAISS index every time we call it.

Idea:

- Move embedding generation into a separate offline preprocessing step.
- Save the FAISS index once and reload it on startup using:
    - faiss.write_index
    - faiss.read_index

Expected benefit:

- Faster service startup.
- Cheaper incremental updates (only rebuild when the corpus changes significantly).

6.4 Use more advanced FAISS index types for large scale

Problem:

- A flat L2 index (IndexFlatL2) is simple and exact but not necessarily efficient for very large datasets.

Idea:

- For large collections, migrate to IVF-based or HNSW-based indexes:
    - IndexIVFFlat with a trained quantizer.
    - Potentially HNSW for better recall/speed trade-offs.

Expected benefit:

- Faster queries and lower memory usage at the cost of slightly approximate results, which is usually acceptable in semantic search.

6.5 Warm-up and prefetching

Problem:

- First requests suffer from model loading and potential caching overhead.

Idea:

- On application startup:
    - Instantiate EmbeddingGenerator once.
    - Run a dummy call to _encode_text as a warm-up.
- This shifts the cost to startup time instead of the first user-visible request.

---

7. Next Steps

For later parts of A5 and future work, we plan to:

1. Implement at least one of the proposed optimizations in the actual codebase (for example, batching in generat e_embeddings and persisting FAISS indices).
2. Use Locust in A5 Part 2 to measure:
    - End-to-end latency for search APIs.
    - The effect of caching and warm-up on P95/P99 latency.
3. In A5 Part 3, design a scaling architecture that:
    - Separates embedding computation from query serving.

- Allows horizontal scaling of the search service.
- Uses offline jobs for index building and refreshing.

---

8. Files Included

- a5/profile_a5_part1.py
  - Profiling harness for the five target functions.
- a5/profiling_results/
  - One .prof file per function (latest run):
    - profile_encode_text_… .prof
    - profile_generate_embeddings_… .prof
    - profile_build_index_… .prof
    - profile_search_text_… .prof
    - profile_search_category_… .prof

Part Two

Part Three

1. INTRODUCTION

It describes how we would scale and harden our arXiv semantic search system as traffic and data volume grow.

we:

- Start from the current system design
- Identify likely bottlenecks conceptually,
- Propose a new scalable architecture, and
- Explain how we would scale it roughly for 10×, 100×, and 1000× more traffic and data.

2. BASELINE SYSTEM (CURRENT DESIGN)

Our current system is essentially a single-node prototype:

1. Client / Frontend
   - A simple UI or HTTP client sends requests to a /search endpoint with a text query and optional filters

(e.g., categories).

2. Search API Service (single process)
    - Python web server (e.g., FastAPI/Flask style).
    - Responsibilities:
        - Receive search requests.
        - Use EmbeddingGenerator to encode the query into a dense embedding.
        - Use VectorIndexer (FAISS) to perform nearest-neighbor search over paper embeddings.
        - Fetch full paper metadata (title, abstract, authors, etc.) from MongoDB.
        - Return results as JSON.
3. Embedding & Index Layer
    - EmbeddingGenerator:
        - Uses a HuggingFace sentence-transformers model (e.g., all-MiniLM-L6-v2) to produce embe ddings.
    - VectorIndexer:
        - Maintains a FAISS index (e.g., IndexFlatL2) for similarity search.
    - Paper embeddings may be precomputed offline or computed ad-hoc depending on the current code pa th.
4. Database Layer (MongoDB)
    - MongoDB stores:
        - Raw arXiv paper metadata,
        - Processed/cleaned paper documents,
        - Possibly some precomputed features.

Limitations:

- A **single API instance** → no horizontal scaling, single point of failure.
- FAISS index and model loaded on that process → limited by that machine's CPU and memory.
- MongoDB is likely a **single instance**, without replicas or sharding.
- Embedding and indexing work may not be fully separated between online and offline paths.

SCALING GOALS AND ASSUMPTIONS

3.1 Likely Bottlenecks

- Query embedding (transformer model inference) is CPU/GPU-heavy.
- FAISS similarity search becomes expensive as the number of papers grows.
- MongoDB reads may become a bottleneck when returning metadata for many queries.

- The single API process is a bottleneck for concurrency and availability.

3.2 Non-Functional Goals

We want an architecture that can:

- Scale to roughly:
  - 10× current data and traffic (small production),
  - 100× (serious internal deployment),
  - 1000× (hypothetical public/large-scale system).
- Improve:
  - Throughput (requests per second),
  - Latency (especially p95),
  - Availability and fault tolerance,
  - Observability (metrics and logging).

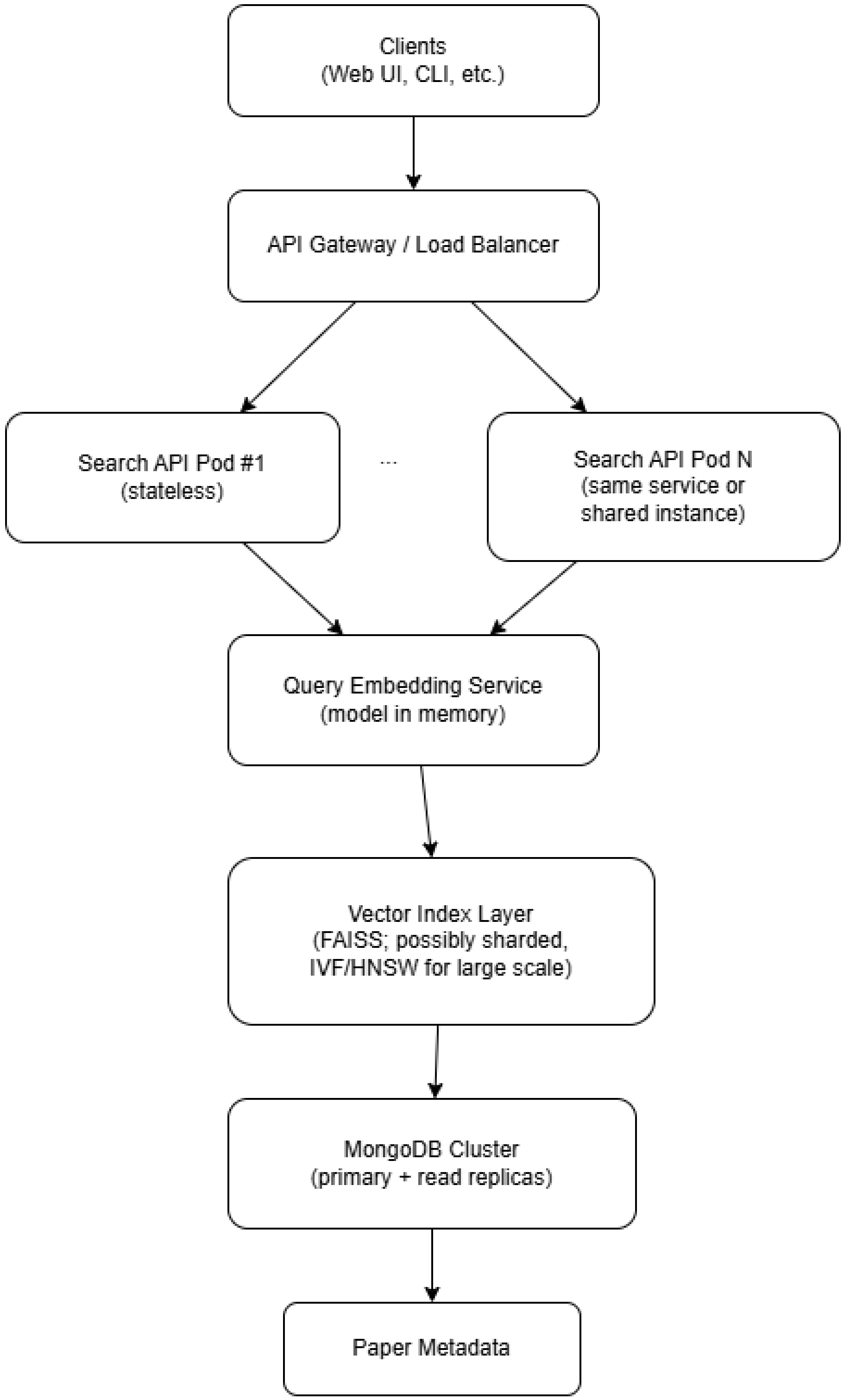
3.3 Design Principles

- Separate **online serving** from **offline preprocessing** and **index building**.
- Make the Search API **stateless and horizontally scalable**.
- Treat heavy components (embedding model, FAISS index) as shared services or managed assets.
- Use replication and snapshots to avoid single points of failure.

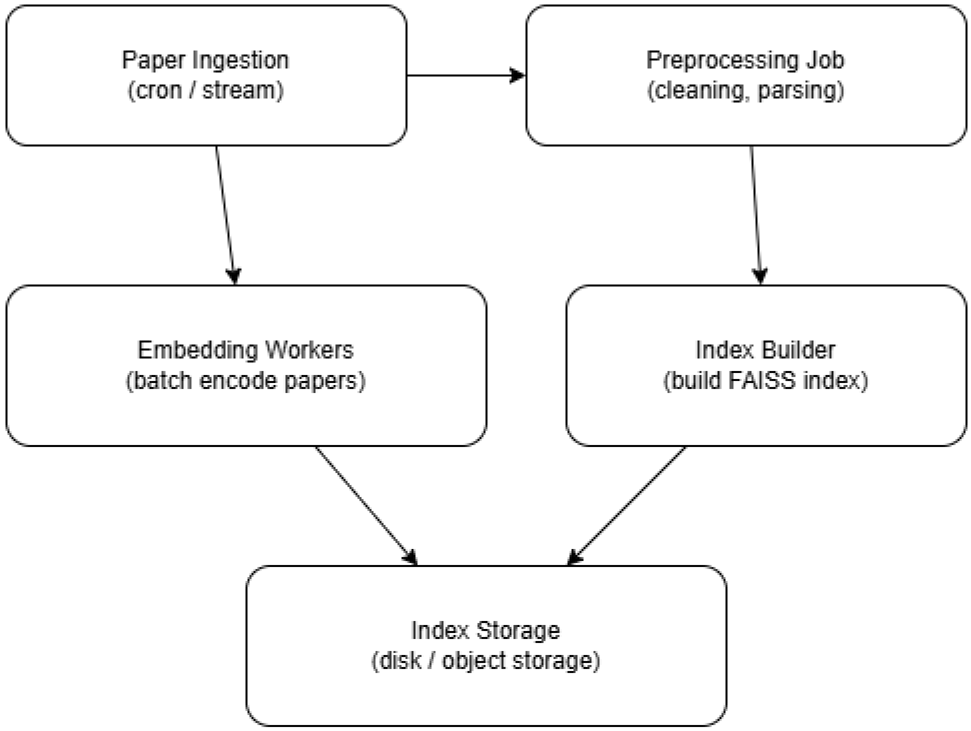
4. NEW SCALABLE ARCHITECTURE

4.1 High-Level Architecture Diagram (Text Version)

Below is a text diagram you can redraw as a figure



Offline pipeline (separate from the request path):



At runtime, each Search API pod loads the latest FAISS index (or talks to a vector index service that holds it in memory).

5. SCALING STRATEGY: 10×, 100×, 1000×

5.1 10× Scale – Small Production

Goal: modest increase in data and traffic, still relatively small.

Changes:

1. API Layer
    - Run 2–3 Search API pods behind a load balancer.
    - Each pod:
        - Is stateless,
        - Loads the model and FAISS index once at startup.
2. Embeddings
    - Precompute **all paper embeddings offline**.
    - Online path only embeds the **query**.
3. FAISS Index
    - Keep using a flat index (IndexFlatL2) in memory.
    - Load the same index into each API pod.
4. MongoDB

- Deploy a small **replica set** (1 primary, 1 secondary).
- Add indices on important fields (e.g., arxiv_id, categories).

Effect:

- Higher throughput via horizontal API scaling.
- Basic fault tolerance: if one pod fails, others continue serving traffic.
- Latency still acceptable for moderate data sizes.

5.2 100× Scale – Larger Internal Deployment

Goal: significantly more users and a much larger corpus.

Changes:

1. API Autoscaling
    - Run Search API in a container orchestrator (e.g., Kubernetes).
    - Use horizontal pod autoscaling based on CPU utilization and/or request rate.
2. Separate Query Embedding Service
    - Extract the embedding model into a dedicated internal service:
        - It holds the model in memory,
        - Accepts batched embedding requests,
        - Can optionally run on GPU.
    - API pods call this service via internal RPC/HTTP.
3. Vector Index Service
    - Instead of loading FAISS in every API pod, introduce a **Vector Index Service**:
        - Maintains FAISS index in memory,
        - Exposes a search(embedding, k) API,
        - Can be replicated and possibly sharded.
4. FAISS Improvements
    - For large N, move from flat index to approximate indexes:
        - IVF (inverted file) or HNSW.
    - Persist index snapshots to disk:
        - Use FAISS save/load routines to speed up restarts.
5. Database Scaling
    - Use a larger MongoDB cluster:
        - More replicas for scaling reads,

        - Possibly sharding by paper ID or category if the dataset grows very large.
    - Use connection pooling and tuned queries.
6. Caching
    - Add a caching layer (e.g., Redis) for:
        - Popular queries' results,
        - Frequently accessed paper metadata.

Effect:

- The system can handle much higher concurrency.
- Heavy components (embedding, index) are centralized and optimized.
- Online requests become a composition of light API logic + fast internal RPC.

5.3 1000× Scale – Hypothetical Planet-Scale

Goal: conceptual design for very high scale.

Additional ideas:

1. Multi-Region Deployment
    - Deploy the whole stack (API, embedding, index, MongoDB) in multiple regions.
    - Use a global load balancer to route users to the nearest region.
    - Keep indexes roughly synchronized via replication of embeddings and index snapshots.
2. Index Sharding
    - Shard the FAISS index by:
        - Document ID range,
        - Time (e.g., year/quarter), or
        - Category (e.g., cs.LG vs cs.CL).
    - For a query, either:
        - Route it to a relevant shard, or
        - Fan out to multiple shards and merge top-K results.
3. GPU-Heavy Embedding Infrastructure
    - Maintain a pool of GPU machines for:
        - Online query embeddings,
        - Offline batch embedding jobs.
4. Aggressive Caching and CDN
    - Cache entire search result pages for extremely popular queries.

    - Use a CDN to serve static assets and pre-rendered results.
5. Cost Optimization
    - Use autoscaling to match capacity to demand.
    - Prefer approximate search and precomputation to minimize expensive per-request work.

---

6. RELIABILITY AND FAULT TOLERANCE

Regardless of exact scale, the architecture should be resilient.

Key measures:

1. Redundancy
    - Multiple API pods.
    - Multiple index/embedding instances.
    - MongoDB replica sets.
2. Graceful Degradation
    - If embedding or index service is down:
        - Fall back to cached results where possible,
        - Return clear error messages rather than hanging.
    - Reduce result size (e.g., top 50 → top 10) under heavy load.
3. Backups and Snapshots
    - Regular MongoDB backups.
    - Regular FAISS index snapshots with versioning and rollback.
4. Timeouts and Circuit Breakers
    - Set strict timeouts for calls to embedding, index, and DB.
    - Use circuit breakers to avoid cascading failures.

7. OBSERVABILITY (METRICS, LOGGING, ALERTS)

Define what we would measure:

1. Metrics
    - API:
        - Request rate, error rate, latency (p50, p95, p99) per endpoint.
    - Embedding:
        - Average and max embedding time, model load failures.

- Index:
  - FAISS search latency, number of vectors, index load time.
- Database:
  - Query latency, connections, replica lag.

2. Logging
   - Structured logs for:
     - Errors and exceptions,
     - Slow requests (above some threshold),
     - Index build events and failures.
3. Alerts
   - Firing when:
     - Error rate exceeds a threshold,
     - Latency exceeds a threshold,
     - Index build fails or MongoDB is unreachable.

8. COST VS PERFORMANCE TRADE-OFFS

Discuss high-level trade-offs:

- Exact vs approximate search
  - Exact (flat index) is simpler but slower at large scale.
  - Approximate indexes (IVF, HNSW) are faster and cheaper for big N.
- CPU vs GPU for embeddings
  - CPU-only: cheaper, simpler, slower.
  - GPU: higher cost per node, but much higher throughput.
- Online vs offline computation
  - Precomputing paper embeddings and indexes reduces online latency.
  - On-the-fly computation is flexible but does not scale well.
- Number of replicas
  - More replicas → higher availability and capacity, but higher cost.
  - Autoscaling lets us pay only for the capacity we actually use.

Our strategy is to keep the early stages (10×) simple and only add complexity (separate services, sharding, GPUs, mul ti-region) when the scale justifies it.

---

9. CONCLUSION

In this Part 3 report, we designed a scaling and reliability strategy for our arXiv semantic search system.

We:

- Described the current single-node baseline system.
- Proposed a new scalable architecture with:
    - A load-balanced, stateless Search API layer,
    - Dedicated embedding and vector index services,
    - An offline embedding and index-building pipeline,
    - A MongoDB cluster instead of a single node.
- Explained how we would scale to approximately 10×, 100×, and 1000× more traffic and data.
- Discussed reliability, observability, and high-level cost/performance trade-offs.