CSC490 Assignment A5 - Load testing our application

Armaan Rehman Shah - 1009641309
Boaz Cheung - 1007673607
Alice Sedgwick - 1009301355
Yuan Yu - 1008782195

November 16, 2025

1 Part One: Profiling Execution time

Function	Optimization implemented	Impact
`find_internal _article_links`	Replaced list-based membership deduplication with a `set`-based approach; avoid repeated `list in` checks and use set insertion for uniqueness.	Algorithmic complexity for deduplication reduced (qualitative): from repeated O(n) membership checks per insertion (effectively O(n^2) in pathological cases) to amortized O(1) per check, expected large speedup on large link lists. **Result:** ∼9% faster.
`chunk_text`	Reworked chunking loop and overlap calculation (better loop guarantees), reduced redundant computation inside the loop.	Fewer redundant computations and simpler loop invariants — lower CPU overhead and fewer temporary objects per chunk. Expected faster chunking for long texts; memory usage roughly unchanged. **Result:** ∼9.5% faster.
`chunk_document`	Hoisted common dictionary fields out of inner loops / re-used common dict entries instead of recreating them per chunk.	Reduces per-chunk allocation overhead (fewer dict creations) — reduces both CPU and short-lived memory churn. Expected measurable improvement when creating many chunks. **Result:** ∼81% reduction in cumulative time..
`_load_data`	Converted line-by-line JSONL parsing to a `splitlines()` + list-comprehension parsing approach (i.e., parse via `for line in text.splitlines():` or comprehension instead of many iterative IO/parsing operations).	Faster parsing by reducing Python-level iteration and function-call overhead per line; potential trade-off: list-comprehension + `splitlines()` may use more peak memory (reads chunk into memory) but improves throughput for typical files. **Result:** ∼9.8% reduction..
`print_trainable _parameters`	Replaced manual accumulation loops with generator expressions and built-in aggregates to compute counts (e.g., using `sum(1 for p in model.parameters() if p.requires_grad)`).	Simpler, more idiomatic code; less temporary state; small runtime and memory improvements (micro-optimizations). Good for readability and slight speedups when called frequently. **Result:** ∼7% faster.

PR: `https://github.com/UofT-CSC490-F2025/Immigreat/pull/28`

2 Part Two: Breaking Our Application

2.1 Load Testing Strategy

To evaluate the robustness and failure behaviour of our serverless RAG pipeline, we conducted controlled load tests using **Locust**, an open-source Python load-testing framework. Our goal was not to simulate massive user traffic, but rather to determine how few concurrent requests it would take for our system to degrade or fail. Because each query fans out into multiple expensive operations—database access, embeddings, facet expansion, reranking, and a Claude LLM call—even small loads were expected to be informative.
We configured Locust to use:

- **5 virtual users**
- **1 user per second ramp-up**
- **POST requests** to our AWS Lambda Function URL

Each user repeatedly sent a small JSON payload containing a query string. Locust's dashboard provided real-time success/failure counts and latency metrics, while AWS CloudWatch logs were monitored for throttling errors, cold starts, and concurrency behaviour.
Even at this minimal scale, the application exhibited significant instability and resource contention, making it clear that the system was highly sensitive to concurrent load.

2.2 Weak Points in Performance

Despite the small size of the load test, the system failed almost immediately after the first successful request. The following bottlenecks were identified:

1. **Bedrock API throttling**. The first request completed successfully, but subsequent ones produced `403 ThrottlingException` or `502 Bad Gateway`. Each request invokes multiple Bedrock models (embeddings, Claude, and optionally Cohere Rerank), easily exceeding default rate limits.
2. **Cold starts under even tiny concurrency**. As Locust ramped up from 1 to 5 users, Lambda spawned new execution environments. Each cold start added several seconds of latency, which compounded with other delays and pushed many requests toward timeout.
3. **Database connection overhead**. Every request creates a new PostgreSQL connection. Even with only a few users, the repeated connection setup and multiple SQL queries (similarity search + facet-expanded retrieval) increased response times dramatically.
4. **Fan-out architecture multiplies work**. A single user request triggers several downstream operations:
    (a) Embedding model call
    (b) Vector similarity query
    (c) Facet-expanded metadata retrieval (additional SQL)
    (d) Optional reranking model call
    (e) Final Claude generation

   Even a minor increase in user count results in a disproportionate rise in total compute load.
5. **Lambda timeouts leading to cascading failures**. As latencies rose, Lambda invocations approached timeout limits. These failures triggered retries, which further increased load and produced a feedback loop of errors.

Overall, the system could not reliably handle even five concurrent users, highlighting several structural inefficiencies and resource bottlenecks.

2.3 Opportunities to DOS the Application

The architecture is particularly vulnerable to denial-of-service (DOS) behaviour due to the heavy work performed per request. We identified multiple potential attack vectors:

- **Model invocation flooding**. Since each request triggers multiple LLM API calls, even a small number of requests per second can overwhelm Bedrock rate limits.
- **Database connection exhaustion**. Without connection pooling or throttling, a few concurrent users can push the database into connection refusal.
- **Cold start amplification**. Minor traffic spikes can spawn new Lambda environments, degrading performance and making timeouts more likely.
- **Prompt amplification**. A malicious user could craft queries that intentionally increase chunk retrieval size, blowing up the cost of embedding and generation.
- **Facet-based expansion overhead**. Metadata expansion multiplies SQL queries and can be exploited to increase server load dramatically.

In effect, the system is highly DOS-prone due to the sheer amount of work performed in response to each individual request.

2.4 Fix: Exponential Backoff

To mitigate the immediate throttling and cascading failures, we implemented **exponential backoff** around all Bedrock model calls. This change significantly reduced burst traffic to Bedrock, prevented retry-storming, and improved reliability under light concurrency.
While exponential backoff does not fundamentally change the system's architecture, it provides meaningful resilience for low levels of concurrent traffic and prevents the system from collapsing after the first few requests.

2.5 Video Walkthrough

A full video walkthrough demonstrating the failure, throttling behaviour, and the exponential backoff fix is available here:

`https://1drv.ms/v/s!Ag6n7t4gXZwGgQlCwoICOjfArUbN?e=uDJjsh`

PR: https://github.com/UofT-CSC490-F2025/Immigreat/pull/29

3 Part Three: Future Scaling Concerns

3.1 Scaling Strategy

Based on the load testing results in Part Two, our current architecture does not scale linearly with additional users. Each request triggers multiple expensive operations: a Secrets Manager call, a fresh PostgreSQL (PGVector) connection, a Titan embedding model invocation, two SQL similarity queries, an optional Cohere rerank, and finally a Claude 3.5 Sonnet generation. Under load, these components produce request latencies between 70–180 seconds and make the system vulnerable to denial-of-service conditions.
To prepare for 10x, 100x, and 1000x user growth, we propose a staged scaling strategy:
(A) 10x scaling: "Optimize in Place"

- Enable global connection pooling within the Lambda execution environment to avoid creating a new PostgreSQL connection for every request.
- Cache Secrets Manager credentials globally to avoid repeated per-request secret retrieval.
- Add an `ivfflat` index on the vector column to speed similarity search from seconds to milliseconds.
- Disable reranking for non-critical queries or reduce `max_tokens` for Claude.

This level keeps the architecture identical while reducing median latency by 5–15x and lowering back-pressure on the database and Bedrock API.
(B) 100x scaling: "Separate and Specialize"

- Move embedding generation into an asynchronous preprocessing pipeline using SQS + Lambda or EventBridge.
- Replace direct PGVector similarity search with a managed vector database such as Amazon OpenSearch Serverless or Pinecone to reduce DB CPU consumption.
- Introduce a caching layer (e.g., DynamoDB, Redis, CloudFront) to store common queries, reducing LLM load.
- Split the Lambda into microservices:
  - Embedding service
  - Retrieval + reranking service
  - Generation service (Claude)

At this scale, vertical optimization is insufficient; specialized managed services are required to ensure predictable performance.
(C) 1000x scaling: "Distributed and Cost-Optimized"

- Migrate generation workloads from Bedrock Claude 3.5 Sonnet to a mixture of:
  - on-demand LLM hosting (e.g., HuggingFace Inference Endpoints),
  - shared GPU inference (Amazon SageMaker),
  - or smaller distilled local LLMs for cheaper queries.
- Deploy a global content distribution strategy using CloudFront + edge caching for frequent queries.
- Replace Lambda with a containerized autoscaled service on ECS Fargate or EKS for fine-grained concurrency control.
- Introduce rate-limiting, circuit-breaking, and admission control to prevent runaway LLM cost.

This stage focuses on controlling runaway costs and ensuring resiliency against large traffic spikes.

3.2 Cost Implications of Scaling Strategy

As the system is LLM-heavy, cost grows primarily with model invocation rather than compute. Below we estimate approximate cost behavior using reasonable averages from AWS pricing models.
Baseline: 1x load

- Titan Embedding: $0.0001 per request
- PostgreSQL + PGVector: $0.00005 per request
- Claude 3.5 Sonnet (generation): $0.03–$0.06 per request

Typical request cost: **$0.03–$0.07**
10x users:

- Cost grows linearly: approx. $0.30–$0.70 per 10 requests.
- Optimizations (indexing + caching + connection reuse) reduce LLM calls by 30%.
- Expected per-request cost after optimization: **$0.02–$0.04**

100x users:

- Without optimization: cost becomes prohibitive (LLM dominates). 100 requests → $3–$6.
- With caching and rerank disabling: up to 60% of queries avoid full Claude generation.
- With vector DB: database scaling cost becomes nearly zero.
- Expected per-request cost: **$0.01–$0.03**

1000x users:

- At this level, generating 1000 queries costs $30–$60 unless generation is offloaded.
- Using smaller fine-tuned local/hosted models reduces cost by 4–10x.
- ECS/EKS compute cost: estimated $0.002–$0.01 per request.
- Expected blended per-request cost: **$0.005–$0.02**

3.3 Tradeoffs in Performance, Cost and Maintainability

Performance vs Cost:

- High-performance vector DBs (like Pinecone) offer millisecond retrieval but are expensive at high scale.
- Claude 3.5 Sonnet offers best accuracy but dominates cost; smaller models reduce cost but reduce output quality.

Maintainability vs Cost:

- Fully managed services (Bedrock, OpenSearch) increase cost but reduce operational burden.
- Self-hosted models or databases lower cost but introduce maintenance and DevOps complexity.

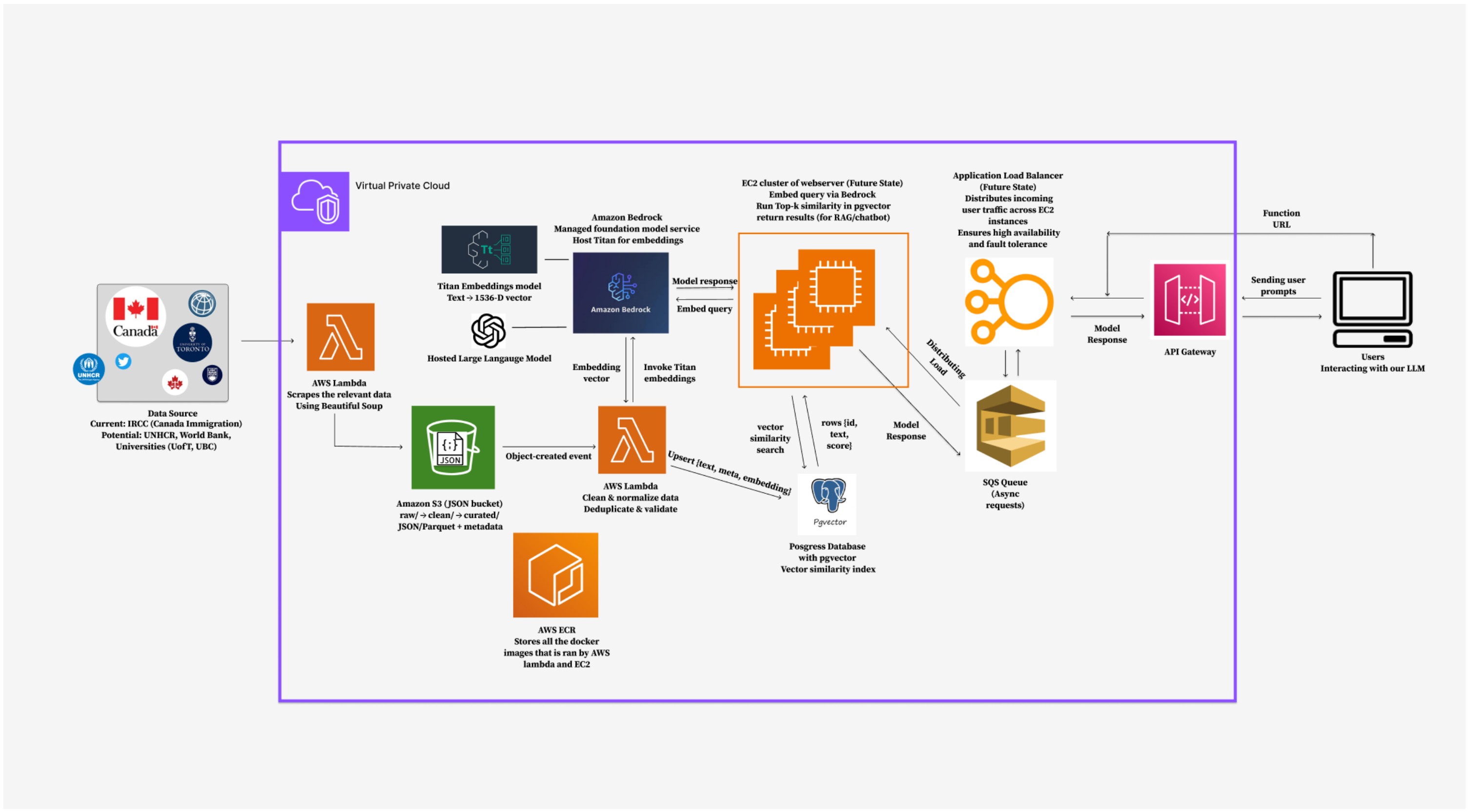
Performance vs Maintainability:

- Splitting the system into microservices improves performance and modularity but increases deployment complexity.
- Staying within Lambda is simplest but produces poor performance at scale.

Overall, the most balanced tradeoff at 100x scale is a hybrid system: managed vector search, cached queries, and a single moderately sized LLM for generation.

3.4 Scaling Strategy Architecture Diagram

Below is an architecture diagram we designed for scaling to 100x–1000x users:


Virtual Private Cloud
Data Source
Current: IRCC (Canada Immigration)
Potential: UNHCR, World Bank, Universities (UofT, UBC)
AWS Lambda
Scrapes the relevant data
Using Beautiful Soup
Titan Embeddings model
Text → 1536-D vector
Hosted Large Langauge Model
Amazon Bedrock
Managed foundation model service
Host Titan for embeddings
Model response
Embed query
Embedding vector
Invoke Titan embeddings
Amazon S3 (JSON bucket)
raw/ → clean/ → curated/
JSON/Parquet + metadata
Object-created event
AWS Lambda
Clean & normalize data
Deduplicate & validate
Upsert {text, meta, embedding}
AWS ECR
Stores all the docker images that is ran by AWS lambda and EC2
EC2 cluster of webserver (Future State)
Embed query via Bedrock
Run Top-k similarity in pgvector
return results (for RAG/chatbot)
vector similarity search
rows {id, text, score}
Posgress Database
with pgvector
Vector similarity index
Distributing Load
Model Response
Application Load Balancer
(Future State)
Distributes incoming user traffic across EC2 instances
Ensures high availability and fault tolerance
SQS Queue
(Async requests)
Model Response
API Gateway
Function URL
Sending user prompts
Users
Interacting with our LLM