Phase 3

CSE354, Distributed Computing

Name:	Ali Tarek Abdelmonim	ID:	21P0123
Name:	Mohamed Mostafa Mamdouh	ID:	21P0244
Name:	Mohamed Walid Helmy	ID:	21P0266
Name:	Tsneam Ahmed Eliwa Zky	ID:	21P0284

Presented To: Dr Ayman Bahaa Dr Hossam Mohamed Eng. Alaa Hamdy
Eng. Mostafa Ashraf

Phase No: (3) Group No.(11)

Date: 2 / 5 /2025

Table of Contents

Table of figures

Introduction

In the modern digital era, efficient web crawling and indexing are crucial for information retrieval. As web content continues to grow exponentially, traditional centralized crawling methods struggle to keep up with scalability demands. This report presents the **enhanced indexing, fault tolerance, and monitoring mechanisms** implemented in a **Distributed Web Crawling and Indexing System using Cloud Computing**. By transitioning to **Elasticsearch**, improving **fault tolerance mechanisms**, and incorporating **robust monitoring features**, this phase aims to optimize performance, resilience, and scalability.

This phase focuses on:

- **Enhanced Indexing:** Adopting Elasticsearch for advanced indexing and search capabilities.
- **Fault Tolerance:** Implementing heartbeat monitoring and task re-queueing to handle node failures.
- **Monitoring:** Introducing logging and status tracking for improved visibility of system performance.

Through these enhancements, the system ensures high availability, data integrity, and efficient search functionality, ultimately contributing to the effectiveness of large-scale web crawling and indexing operations.

Enhanced Indexing, Fault Tolerance, and Monitoring

Transition to Elasticsearch for Robust Indexing

Objective: Replace basic indexing with Elasticsearch for scalability and advanced search capabilities.

Elasticsearch Configuration: The indexer initializes a connection to Elasticsearch and creates an index with optimized mappings

```
# --- Initialize Elasticsearch ---
try:

    ES_URL = f"https://{ES_USERNAME}:{ES_PASSWORD}@{ES_HOST}"

    es_client = Elasticsearch(
        ES_URL,
        verify_certs=True
    )
    if not es_client.ping():
        raise ValueError("Elasticsearch connection failed")
    logging.info(f"Connected to Elasticsearch at {ES_HOST}:{ES_PORT}")

    if not es_client.indices.exists(index=ES_INDEX_NAME):
        mapping = {
            "mappings": {
                "properties": {
                    "url": {"type": "keyword"},
                    "content": {"type": "text", "analyzer": "standard"}
                }
            }
        }
```

```
        es_client.indices.create(index=ES_INDEX_NAME, body=mapping)
        logging.info(f"Created Elasticsearch index '{ES_INDEX_NAME}'")
    except Exception as e:
        logging.error(f"Failed to initialize Elasticsearch client: {e}", exc_info=True)
        exit(1)
```

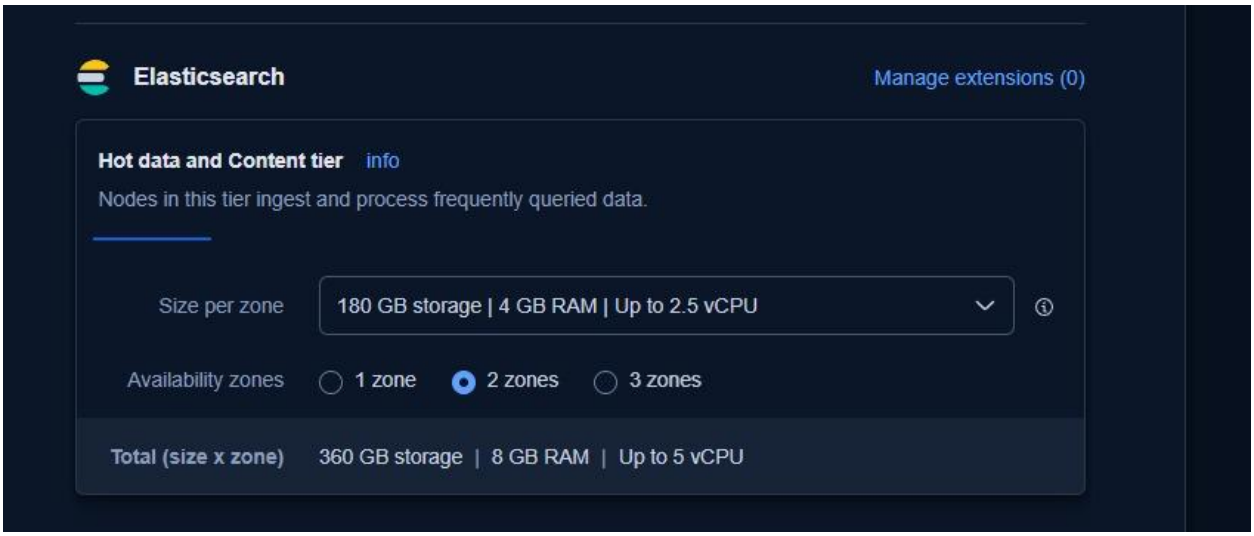

Figure 1 Creation of ElasticSearch deployment with 4 availability zones.

Note that the elastic search deployment has 2 availability zones for higher reliability and fault tolerance.

Enhanced Content Processing

Text Extraction: The download_from_gcs function retrieves processed text from Google Cloud Storage (GCS), which is already cleaned by the crawler (e.g., HTML stripped)

```
# --- Download Text from GCS ---
def download_from_gcs(bucket_name, blob_path):
    try:
        bucket = storage_client.bucket(bucket_name)
        blob = bucket.blob(blob_path)
        return blob.download_as_text()
    except exceptions.NotFound:
        logging.error(f"GCS object not found: gs://{bucket_name}/{blob_path}")
```

```
        return None
    except Exception as e:
        logging.error(f"Failed to download from GCS path gs://{bucket_name}/{blob_path}:
{e}")
        return None
```

Fault Tolerance Implementation for Crawler Nodes

Heartbeat Monitoring

```
def start_health_heartbeat():
    def loop():
        while True:
            publish_health_status()
            time.sleep(30)
    threading.Thread(target=loop, daemon=True).start()

def publish_health_status():
    health_msg = {
        "node_type": "crawler",
        "hostname": socket.gethostname(),
        "status": "online",
        "timestamp": datetime.utcnow().isoformat()
    }
    publish_message(metrics_topic_path, health_msg)

# --- Main Execution ---
def main():
    logging.info("Crawler node starting...")
    logging.info(f"Project ID: {PROJECT_ID}")
    logging.info(f"Listening for tasks on subscription: {subscription_path}")
    logging.info(f"Publishing index data to topic: {index_topic_path}")
    logging.info(f"Publishing new URLs to topic: {new_url_topic_path}")
    logging.info(f"Storing data in GCS Bucket: {GCS_BUCKET_NAME}")
    logging.info(f"Max Crawl Depth: {MAX_DEPTH}")
    start_health_heartbeat()
```

Figure 2 Implementation of heartbeat tracking in code

Note that heartbeat monitoring was also set within the google cloud console for the crawler and indexer managed instance groups, as well as the master node. With a set interval of 180 seconds, if the VM does not generate a pulse for 2 consecutive periods of 100 seconds, then the VM will be considered dead and a new VM will be be created to replace it.

Task Timeout

File: src/master/master_node.py

```
# Modify publish_crawl_task to accept parameters
def publish_crawl_task(url, depth=0, domain_restriction=None, source_job_id=None):
    """Publishes a single URL crawl task to Pub/Sub."""
    global total_crawled
    task_id = str(uuid.uuid4())
    message_data = {
        "task_id": task_id,
        "url": url,
        "depth": depth,
        "domain_restriction": domain_restriction, # Pass along
        "source_job_id": source_job_id # Optional: Link back to UI job
    }
    data = json.dumps(message_data).encode("utf-8")

    try:
        future = publisher.publish(crawl_topic_path, data)
        total_crawled += 1
        publish_metric("urls_crawled", total_crawled)
        future.result(timeout=30)
        logging.info(f"Published task {task_id} for URL: {url} (From Job: {source_job_id}, Depth:
{depth}, Domain: {domain_restriction})")
        return True
    except exceptions.GoogleAPICallError as e:
        logging.error(f"API error publishing task for URL {url}: {e}")
        return False
    except Exception as e:
        logging.error(f"Unexpected error publishing task for URL {url}: {e}", exc_info=True)
        return False
```

In the publish_crawl_task() function, the line future.result(timeout=30) introduces a critical timeout mechanism that enhances the reliability of the task publishing process. When a crawl task is sent to Google Cloud Pub/Sub, the publish() method returns a Future object representing the asynchronous operation of message delivery. By invoking .result(timeout=30), the system waits for a maximum of 30 seconds for the publishing confirmation. This ensures that the master node does not block indefinitely in the event of network latency or service delays. If the publish operation does not complete within the specified timeout, a TimeoutError is raised, allowing the system to handle it gracefully in the surrounding try-except block. This mechanism not only prevents system hangs but also provides an opportunity to log, retry, or escalate failures appropriately, thereby improving fault tolerance and operational robustness in the task distribution pipeline.

Logging

To ensure observability, debuggability, and maintainability of the distributed crawling system, the master node includes a comprehensive logging mechanism using Python's built-in logging module. Logging is crucial for tracking system events, capturing errors, and monitoring the flow of operations across crawling tasks.

```
# --- Setup Logging ---
logging.basicConfig(
   level=logging.INFO,
   format='%(asctime)s - Master - %(levelname)s - %(message)s',
   datefmt='%Y-%m-%d %H:%M:%S'
)
```

Logging Use Cases in Master Node

File: src/master/master_node.py

1. **Error Reporting for Missing Environment Variables**

```
missing_vars = [k for k, v in essential_vars.items() if v is None]
if missing_vars:
      logging.error(f"Error: Missing essential environment variables:
      {','.join(missing_vars)}")
   logging.error("Please set them in your environment or a .env file.")
```

```
    exit(1)
```

These logs help identify configuration issues early during startup.

2. **Task Publishing Feedback**

```
logging.info(f"Published task {task_id} for URL: {url} (From Job: {source_job_id}, Depth: {depth}, Domain: {domain_restriction})")

logging.error(f"Unexpected error publishing task for URL {url}: {e}", exc_info=True)
```

When crawl tasks are dispatched via Pub/Sub, success or failure is clearly logged, allowing operators to verify system activity or troubleshoot errors.

3. **Job Handling and URL Processing**

During job consumption from Pub/Sub and processing of GCS blobs:

```
# --- Handle Incoming Crawl Job Requests ---
def handle_new_job(message: pubsub_v1.subscriber.message.Message):
    global total_jobs_received
    try:
        data_str = message.data.decode("utf-8").strip()
        if not data_str:
            logging.error("Received empty message from Pub/Sub.")
            message.ack()
            return


        try:
            job_meta = json.loads(data_str)
        except json.JSONDecodeError as e:
            logging.error(f"Failed to parse JSON: {e}")
    message.ack()
    return

task_id = job_meta.get("task_id")
gcs_path = job_meta.get("gcs_path")
total_jobs_received += 1
publish_metric("crawl_jobs_received", total_jobs_received)
```

```
    if not task_id or not gcs_path:
        logging.error("Missing task_id or gcs_path in the message.")
        message.ack()
        return

    logging.info(f"Received task {task_id} from {gcs_path}")

    if not gcs_path.startswith("gs://"):
        logging.error(f"Invalid GCS path: {gcs_path}")
        message.ack()
        return

    # Safe split: remove 'gs://' and split at first slash
    try:
        parts = gcs_path[5:].split("/", 1)
        bucket_name, blob_path = parts
    except ValueError:
        logging.error(f"Failed to parse GCS path: {gcs_path}")
        message.ack()
        return

    bucket = storage_client.bucket(bucket_name)
    blob = bucket.blob(blob_path)

    blob_content = blob.download_as_text()
    if not blob_content.strip():
        logging.error(f"GCS blob at {gcs_path} is empty.")
        message.ack()
        return

    job_data = json.loads(blob_content)
    seed_urls = job_data.get("seed_urls", [])
    depth_limit = job_data.get("depth", 1)
    domain_restriction = job_data.get("domain_restriction")

    if not isinstance(seed_urls, list) or not seed_urls:
        logging.warning(f"No seed URLs found in job {task_id}. Skipping.")
        message.ack()
```

```
            return

        for url in seed_urls:
            publish_crawl_task(
                url,
                depth=0,
                domain_restriction=domain_restriction,
                source_job_id=task_id
            )
            time.sleep(0.05)

        logging.info(f"Published crawl tasks for job {task_id}")
        message.ack()

    except json.JSONDecodeError as e:
        logging.error(f"Failed to parse JSON: {e}")
        message.ack()
    except Exception as e:
        logging.error(f"Failed to process incoming crawl job: {e}", exc_info=True)
        message.nack()
```

These messages trace job lifecycle events, from receipt to processing and re-dispatching of seed URLs.

4. **Unhandled Exceptions and Recovery**

   The script uses:

   ```
   except Exception as e:
   logging.error(..., exc_info=True)
   ```

   to log stack traces during unexpected exceptions, making post-mortem analysis easier.

5. **Shutdown and Graceful Exit**

   ```
   logging.info("KeyboardInterrupt received. Shutting down gracefully...")
   ```

   Ensures that application termination is also logged for operational clarity.

Logging Use Cases in Crawler Node

File: src/scripts/crawler_node.py

1. **System Initialization**

   logging.info("Crawler node starting...")

   Logs essential system configuration, such as the project ID, GCS bucket, and Pub/Sub topic paths.

2. **Message Validation**

   logging.warning(f"Received invalid task data (missing/invalid URL): {data_str}")

   Logs issues when invalid or malformed messages are received, ensuring they are acknowledged or rejected without crashing the crawler.

3. **Task Processing**

   logging.info(f"Received task {task_id}: Crawl URL: {url} at depth {depth}")

   Each received crawl task is logged with its ID, target URL, and depth level to help trace the crawl path.

4. **HTTP Request Failures**

```
except requests.exceptions.Timeout:
    logging.warning(f"Timeout fetching URL: {url}")
    message.nack() # Let Pub/Sub redeliver later or move to dead-letter
    return
except requests.exceptions.RequestException as e:
    logging.error(f"Request failed for URL: {url} - {e}")
    message.ack()
```

   If a request fails due to timeout or other network issues, the error is captured to differentiate between temporary and permanent problems.

5. **Content Processing and Storage**

   ```
   logging.warning(f"No text content extracted from {final_url}")
   logging.info(f"Successfully processed and queued for indexing: {final_url}")
   ```

   Provides feedback on content extraction and GCS storage success/failure, helping to verify what content is crawled and passed to the indexer.

6. **New URL Discovery**

   ```
   logging.info(f"Found {new_urls_found} new URLs from {final_url}")
   ```

   Tracks link extraction and ensures that discovered URLs are logged before being sent back to the master node.

7. **Error Handling and Debugging**

   ```
   logging.error(f"Unexpected error processing message for task {task_id}: {e}",
   exc_info=True)
   ```

   Any unhandled or unexpected error is logged with full traceback information (exc_info=True), helping developers debug runtime issues without losing task context.

8. **Shutdown and Exception Monitoring**

   ```
   logging.info("Crawler node shutting down.")
   logging.error(f"Subscriber error: {e}", exc_info=True)
   ```

   Logs controlled shutdowns (e.g., via KeyboardInterrupt) and unexpected subscriber failures for graceful recovery.

Logging Use Cases in Indexer Node

File: src/scripts/indexer_node.py

1. **Startup and Configuration Verification**

```
logging.info("Indexer node starting...")
```

```
logging.info(f"Project ID: {PROJECT_ID}")
logging.info(f"Listening on: {subscription_path}")
logging.info(f"GCS Bucket: {GCS_BUCKET_NAME}")
logging.info(f"Elasticsearch: {ES_HOST}:{ES_PORT}, Index: {ES_INDEX_NAME}")
```

Logs essential environment configurations such as project ID, subscription path, Elasticsearch host, and GCS bucket.

2. **Elasticsearch Initialization**

```
logging.info(f"Connected to Elasticsearch at {ES_HOST}:{ES_PORT}")
logging.info(f"Created Elasticsearch index '{ES_INDEX_NAME}'")
logging.error(f"Failed to initialize Elasticsearch client: {e}", exc_info=True)
```

Confirms Elasticsearch connectivity and index creation or logs errors with full stack traces if initialization fails.

3. **Message Handling**

```
logging.info(f"Received task: Index content_id {content_id} for URL {url} from {gcs_path}")
logging.warning("No URL found in message.")
logging.warning(f"Invalid task data (missing URL or GCS path): {data_str}")
```

Logs every incoming task message, clearly stating what is being indexed, or highlights incomplete or malformed messages.

4. **GCS Download Failures**

```
logging.error(f"GCS object not found: gs://{bucket_name}/{blob_path}")
logging.error(f"Failed to download from GCS path gs://{bucket_name}/{blob_path}: {e}")
```

Informs when content cannot be fetched from Google Cloud Storage, allowing operators to identify and address missing or inaccessible files.

5. **Indexing Success and Failures**

```
logging.info(f"Successfully indexed document ID: {url}")
logging.error(f"Error indexing URL {url}: {e}", exc_info=True)
```

Reports successful additions to the Elasticsearch index or details failures during the indexing process with full error context.

6. **Subscriber Lifecycle Events**

```
logging.info("Indexer node shutting down.")
logging.error(f"Subscriber error: {e}", exc_info=True)
```

Tracks graceful shutdowns (e.g., from a KeyboardInterrupt) and handles unexpected errors during message streaming.

Client Interface Development

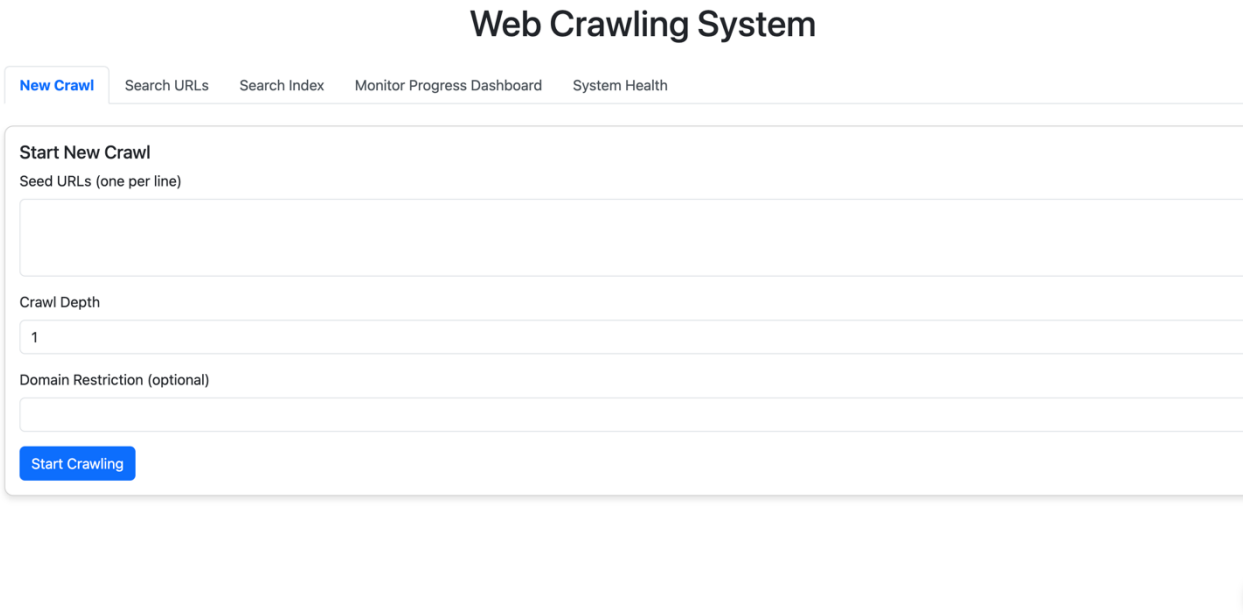

Figure 3 Start New Crawl Interface

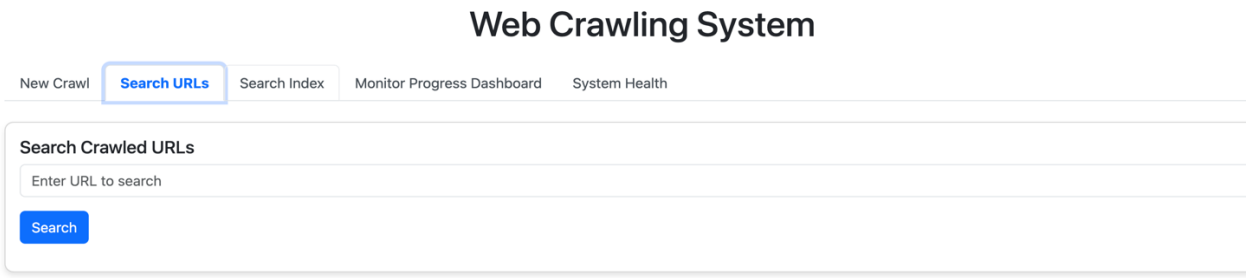

Figure 4 Search Crawled URLs Interface

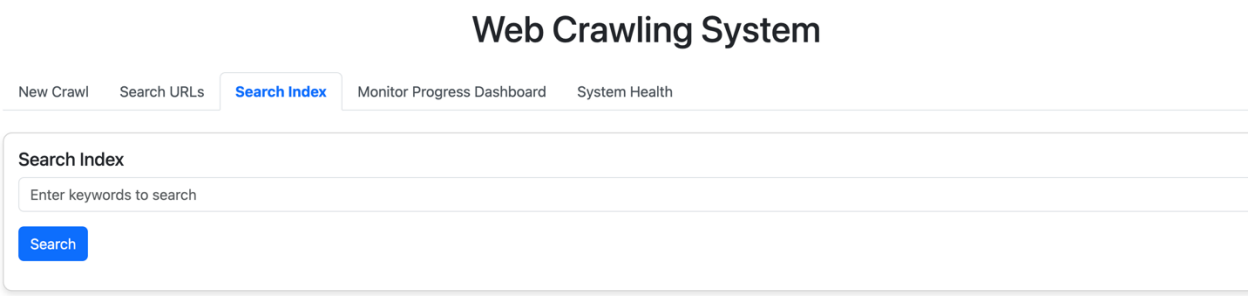

Figure 5 Search Index Interface

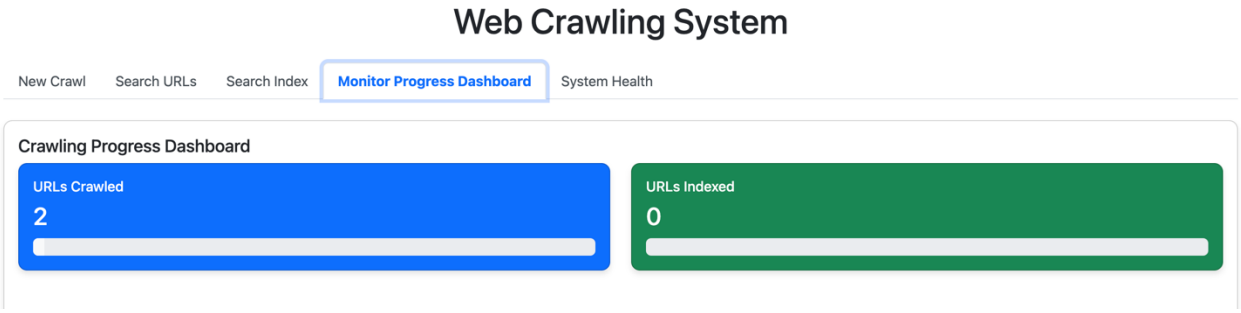

Figure 6 Crawling Progress Dashboard

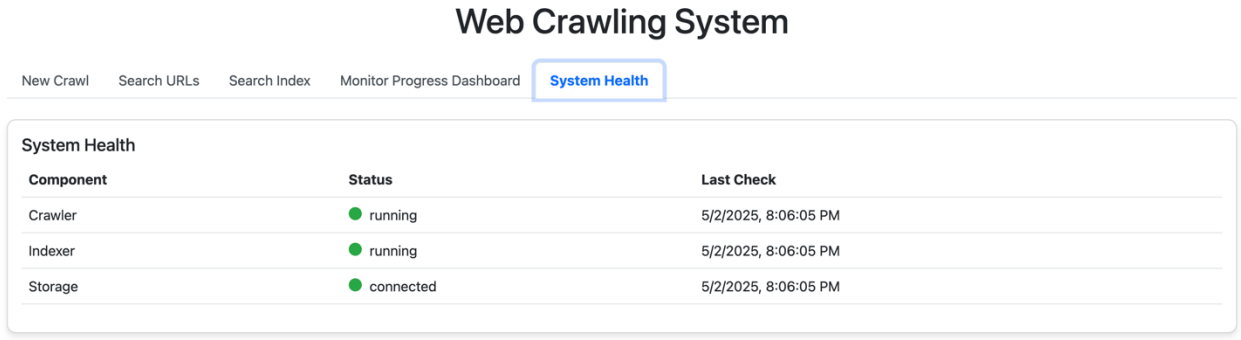

Figure 7 System Health Status

Testing and Validation

We are currently using a managed instance group for crawlers and indexers. Instance templates of each of those has autoscaling set, starting with 1 all the way up to 8. To test the autoscaling feature here, we deleted the solo virtual machine running as a crawler and within 30 seconds a new crawler virtual machine was up and running, replacing the deleted one.

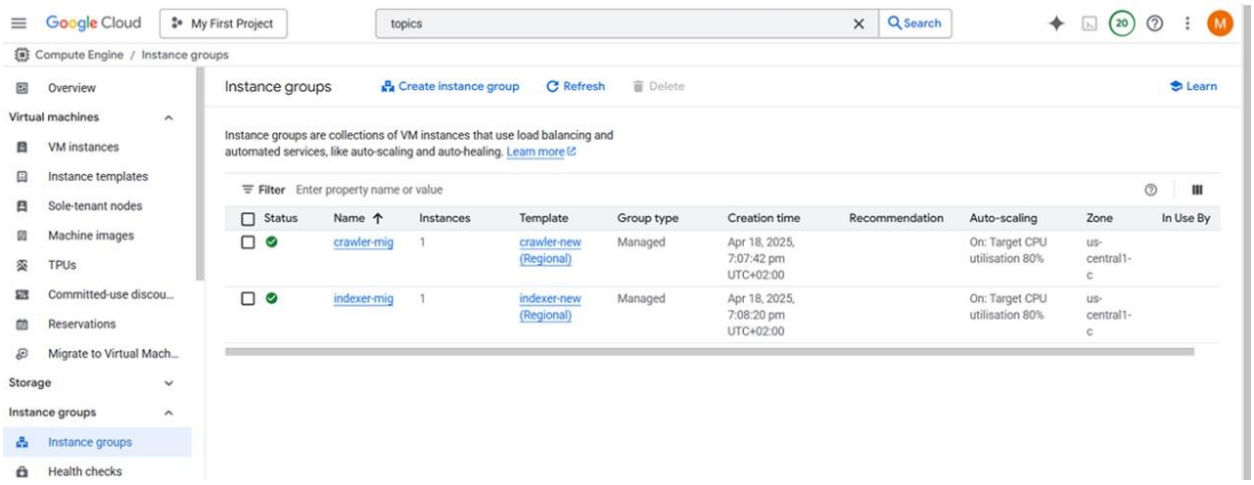

Figure 8:Testing and Validation using managed instance groups

Conclusion

The implementation of enhanced indexing, fault tolerance, and monitoring in this phase significantly improves the reliability and efficiency of the Distributed Web Crawling and Indexing System. Transitioning to Elasticsearch optimizes indexing processes, ensuring fast and scalable search operations. The integration of heartbeat monitoring, task timeout handling, and error recovery mechanisms strengthens the system's resilience against node failures. Additionally, real-time logging and monitoring tools enhance system transparency and facilitate troubleshooting.

These advancements collectively ensure that the system is robust, scalable, and fault-tolerant, capable of operating effectively in cloud-based distributed environments. Moving forward, further refinements in scalability testing, system optimization, and extended search functionalities will help elevate the system’s efficiency, preparing it for larger-scale deployment.

References

https://cloud-based_web_crawler_architecture_cloud_lab_ucm.pdf

https://cloud.google.com/architecture/framework/reliability/build-highly-available-systems

https://www.robotstxt.org/

https://www.crummy.com/software/BeautifulSoup/

https://cloud.google.com/?hl=en

https://developer.hashicorp.com/terraform

https://cloud.google.com/pubsub?hl=en

https://cloud.google.com/storage?hl=en

https://cloud.google.com/appengine?hl=en