Phase 2

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: (2)

Date: 27 / 4 /2025

Table of Contents

Table of Figures

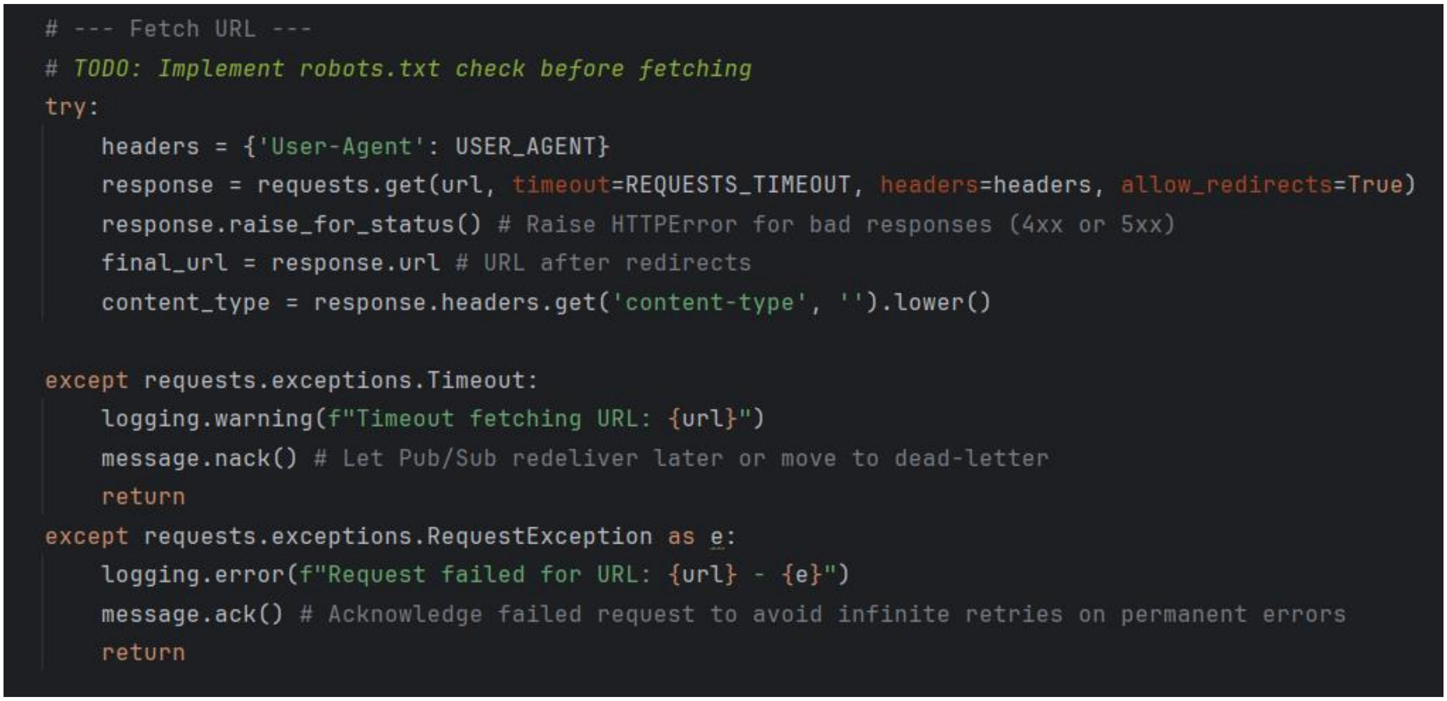
Introduction

In Phase 2 of the CSE354 Distributed Computing project, the primary focus is on developing the foundational functionality of a distributed web crawling and indexing system. Building on the groundwork laid in Phase 1, this phase emphasizes implementing a distributed crawling mechanism, establishing communication between system components, and creating a basic indexing framework.

The main objectives are to design and test a distributed crawler that efficiently fetches web pages while adhering to politeness policies, set up the master node to manage task distribution effectively, and implement a simple indexing solution capable of supporting basic search functionality. By the end of this phase, the integration of these components will enable the system to demonstrate end-to-end functionality, laying the groundwork for subsequent enhancements in scalability, fault tolerance, and advanced indexing.

Core Crawling Implementation

1.Web page Fetching



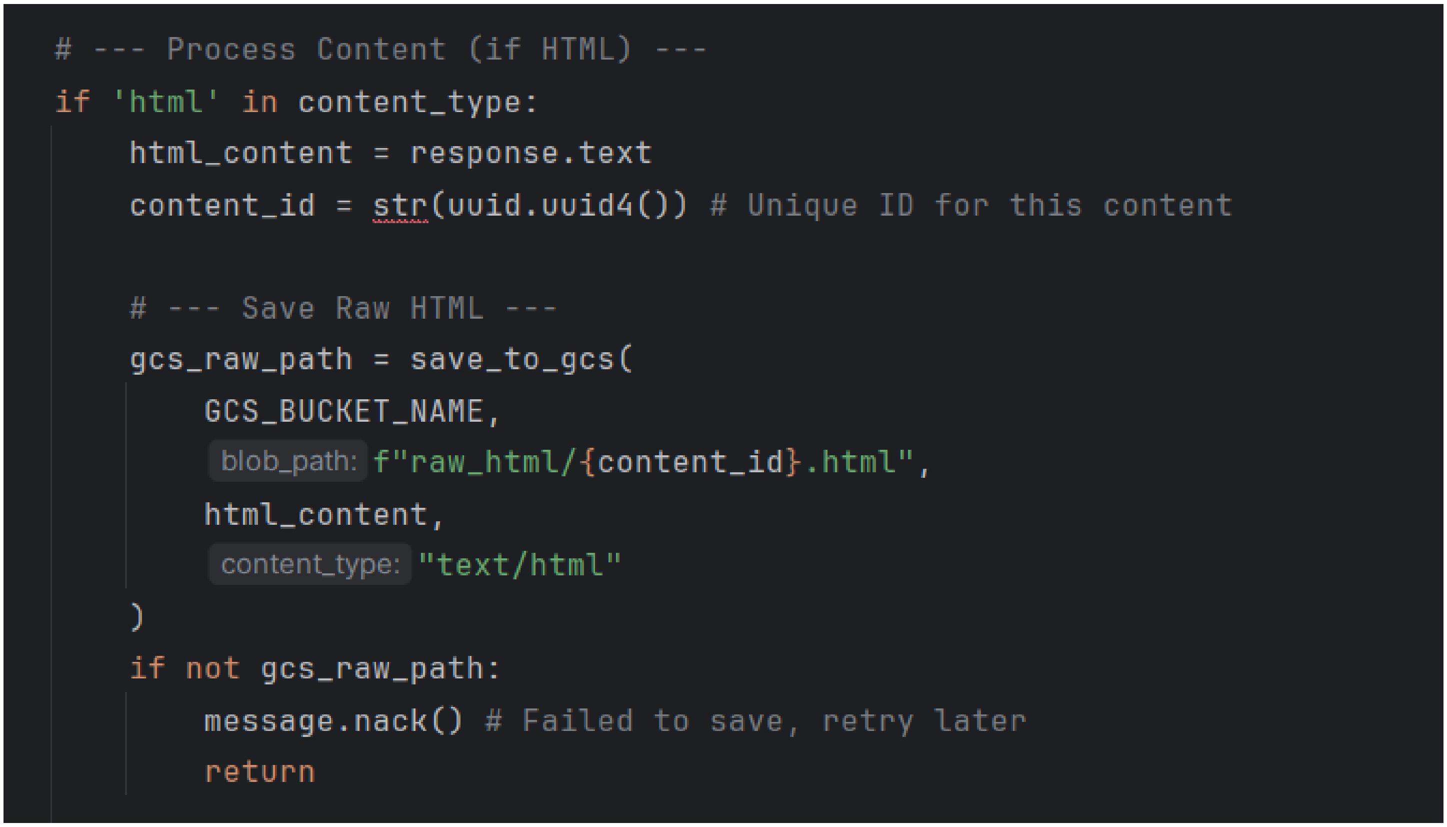
```python
# --- Fetch URL ---
# TODO: Implement robots.txt check before fetching
try:
    headers = {'User-Agent': USER_AGENT}
    response = requests.get(url, timeout=REQUESTS_TIMEOUT, headers=headers, allow_redirects=True)
    response.raise_for_status() # Raise HTTPError for bad responses (4xx or 5xx)
    final_url = response.url # URL after redirects
    content_type = response.headers.get('content-type', '').lower()

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() # Acknowledge failed request to avoid infinite retries on permanent errors
    return
```

Figure 1: Web page Fetching

- **Purpose:** This section fetches web pages using HTTP requests, handling redirections and various exceptions such as timeouts and invalid requests.
- **Phase 2 Alignment:** Demonstrates the crawler's ability to access and retrieve web content, an essential part of web crawling functionality.
- **Key Points:**

- ○ **requests.get**: Makes the HTTP request to fetch the page.
- ○ **response.raise_for_status**: Ensures HTTP errors (like 404 or 500) are caught.
- ○ Error handling ensures robustness, retrying or gracefully skipping problematic URLs.

2. HTML Parsing and Text Extraction

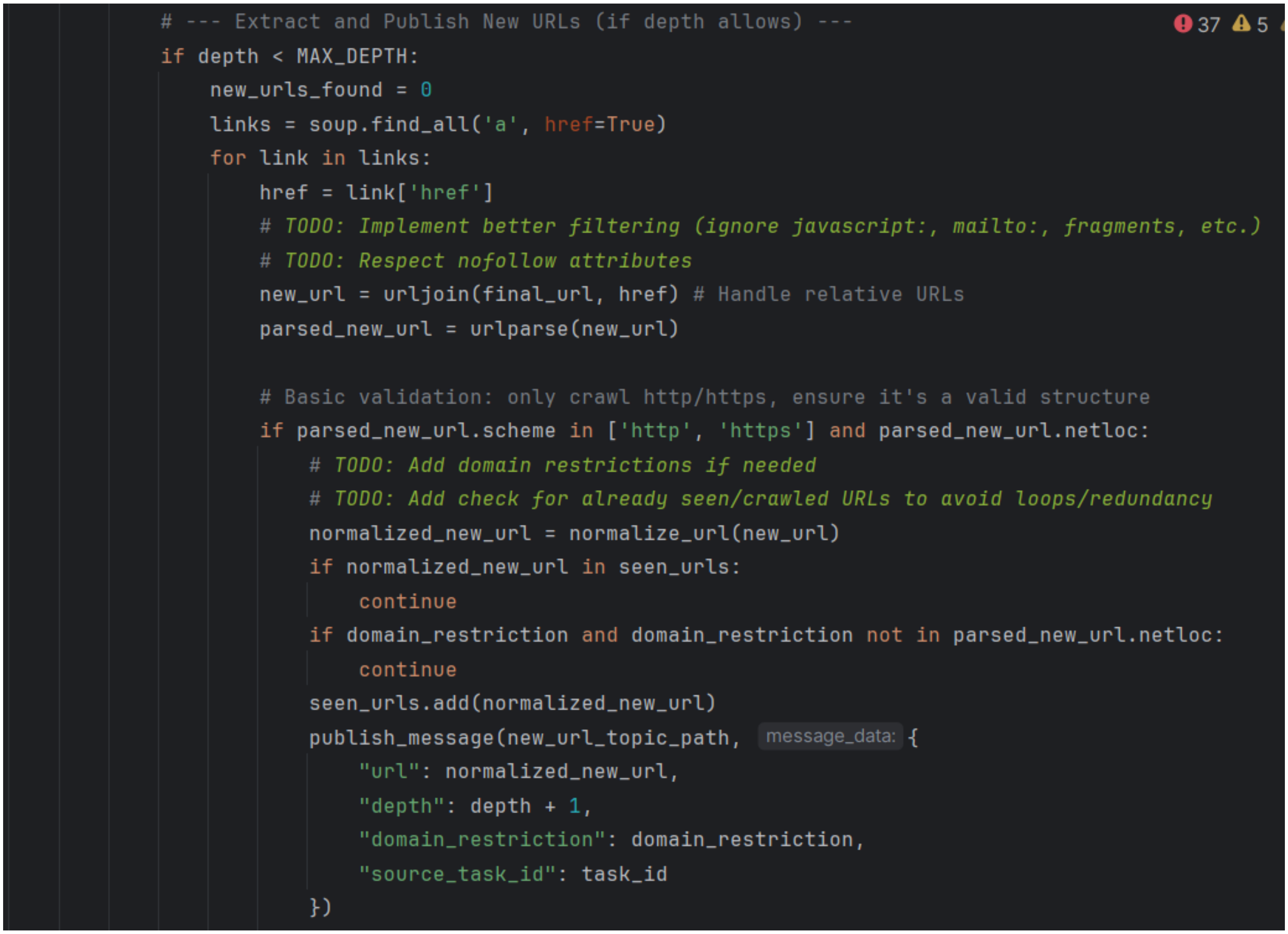


```
# --- Process Content (if HTML) ---
if 'html' in content_type:
    html_content = response.text
    content_id = str(uuid.uuid4()) # Unique ID for this content

    # --- Save Raw HTML ---
    gcs_raw_path = save_to_gcs(
        GCS_BUCKET_NAME,
        blob_path: f"raw_html/{content_id}.html",
        html_content,
        content_type: "text/html"
    )
    if not gcs_raw_path:
        message.nack() # Failed to save, retry later
        return
```

Figure 2: HTML Parsing and Text Extraction

- **Purpose:** Parses HTML content using BeautifulSoup to extract textual data for indexing.
- **Phase 2 Alignment:** Fulfills requirements for text extraction, enabling data ingestion for the indexing process.
- **Key Points:**
  - ○ **BeautifulSoup:** Parses the HTML content.
  - ○ **soup.stripped_strings:** Extracts clean text content from the page.
  - ○ **save_to_gcs:** Saves the processed text to cloud storage for persistence.

3. Extracting Links and Publishing New URL Tasks



```
# --- Extract and Publish New URLs (if depth allows) ---
if depth < MAX_DEPTH:
    new_urls_found = 0
    links = soup.find_all('a', href=True)
    for link in links:
        href = link['href']
        # TODO: Implement better filtering (ignore javascript:, mailto:, fragments, etc.)
        # TODO: Respect nofollow attributes
        new_url = urljoin(final_url, href) # Handle relative URLs
        parsed_new_url = urlparse(new_url)

        # Basic validation: only crawl http/https, ensure it's a valid structure
        if parsed_new_url.scheme in ['http', 'https'] and parsed_new_url.netloc:
            # TODO: Add domain restrictions if needed
            # TODO: Add check for already seen/crawled URLs to avoid loops/redundancy
            normalized_new_url = normalize_url(new_url)
            if normalized_new_url in seen_urls:
                continue
            if domain_restriction and domain_restriction not in parsed_new_url.netloc:
                continue
            seen_urls.add(normalized_new_url)
            publish_message(new_url_topic_path, message_data: {
                "url": normalized_new_url,
                "depth": depth + 1,
                "domain_restriction": domain_restriction,
                "source_task_id": task_id
            })
```

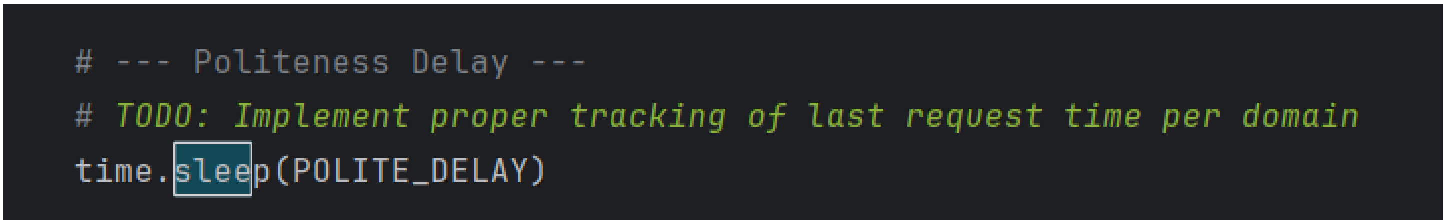
Figure 3: Extracting Links and Publishing New URL Tasks

- **Purpose:** Extracts URLs from the HTML content, validates them, and publishes new crawling tasks.

- **Phase 2 Alignment:** Implements link extraction and ensures task queuing for distributed crawling.
- **Key Points:**
    - **soup.find_all('a', href=True):** Finds all anchor tags with href attributes.
    - **urljoin**: Resolves relative URLs.
    - **urlparse:** Validates the structure of the URLs.
    - Depth control ensures that URLs are processed only within the permitted crawl depth.
    - Publishing tasks to a Pub/Sub topic enables distributed crawling.

4.Basic Politeness Measures (Crawl Delay)

The **politeness measure**, specifically a delay between requests to avoid overwhelming servers, is implemented as the **POLITE_DELAY** variable in the crawler node code. Here's where it appears:



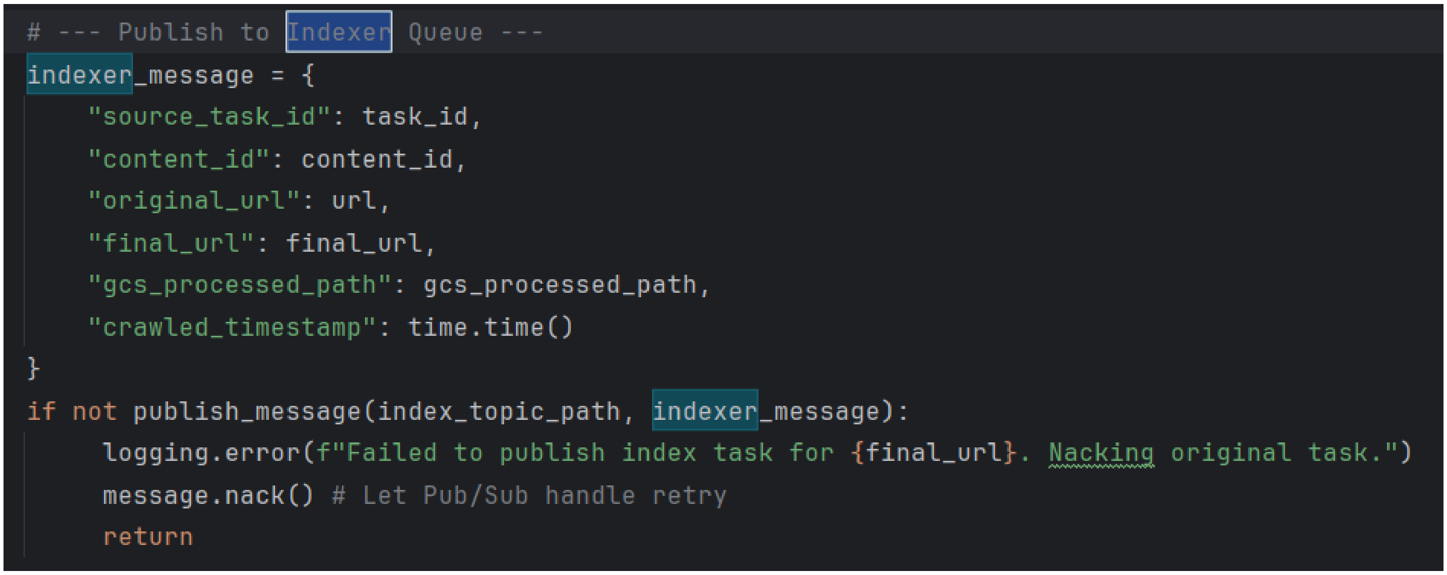
```
# --- Politeness Delay ---
# TODO: Implement proper tracking of last request time per domain
time.sleep(POLITE_DELAY)
```

Figure 4:Basic Politeness Measures

- **Explanation:** This delay ensures that the crawler pauses between requests to the same domain, adhering to basic politeness in web crawling. The **POLITE_DELAY** is currently a simple delay that could be enhanced in later phases by tracking the last request time for each domain.

5. Communication to Report Crawled Content and Extracted URLs Back to the Master

The **communication** with the Master Node is achieved using Google Pub/Sub for task messaging. Here's how the crawled content and extracted URLs are reported:

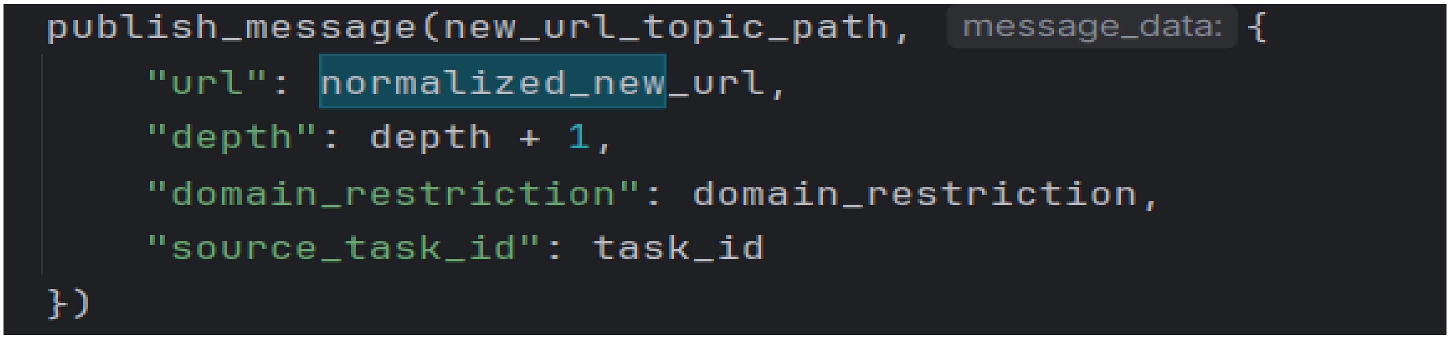


```
# --- Publish to Indexer Queue ---
indexer_message = {
    "source_task_id": task_id,
    "content_id": content_id,
    "original_url": url,
    "final_url": final_url,
    "gcs_processed_path": gcs_processed_path,
    "crawled_timestamp": time.time()
}
if not publish_message(index_topic_path, indexer_message):
     logging.error(f"Failed to publish index task for {final_url}. Nacking original task.")
     message.nack() # Let Pub/Sub handle retry
     return
```

Figure 5: Communication to report crawled content and Extracted URLS Back to the Master

Explanation: The crawler sends processed content details (e.g., content ID, URLs, storage paths) to the indexing topic. This ensures that the Master and Indexer nodes are aware of the crawled and processed data.

6.Code for Reporting Extracted URLs



```
publish_message(new_url_topic_path,  message_data: {
    "url": normalized_new_url,
    "depth": depth + 1,
    "domain_restriction": domain_restriction,
    "source_task_id": task_id
})
```

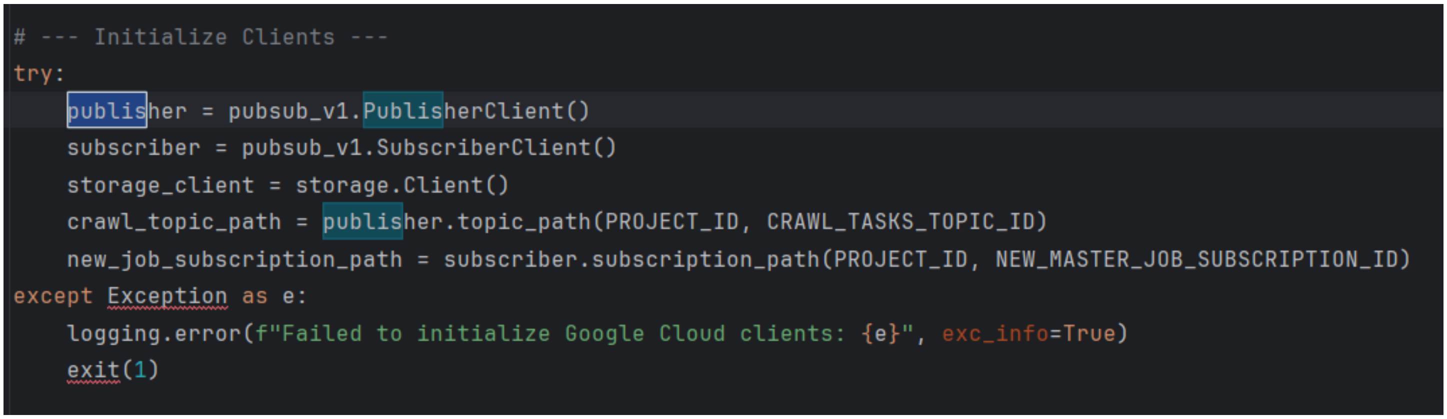
Figure 6:Report Extracted URLs

- **Explanation:** Extracted URLs are sent back as tasks for further crawling. Each task contains the URL, crawl depth, and optional domain restrictions, enabling the Master Node to manage and schedule these tasks for distribution.

Master Node Functionality

1. Task Queue Integration

- code



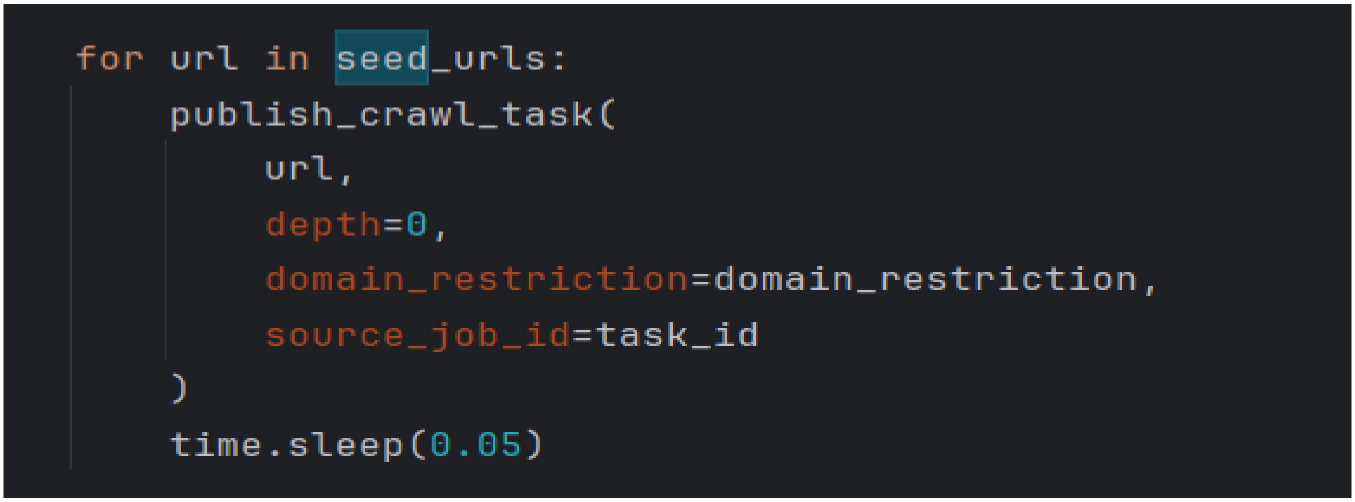
```
# --- Initialize Clients ---
try:
    publisher = pubsub_v1.PublisherClient()
    subscriber = pubsub_v1.SubscriberClient()
    storage_client = storage.Client()
    crawl_topic_path = publisher.topic_path(PROJECT_ID, CRAWL_TASKS_TOPIC_ID)
    new_job_subscription_path = subscriber.subscription_path(PROJECT_ID, NEW_MASTER_JOB_SUBSCRIPTION_ID)
except Exception as e:
    logging.error(f"Failed to initialize Google Cloud clients: {e}", exc_info=True)
    exit(1)
```

Figure 7:Task Queue Integration

- **Explanation:** The Master Node uses Google Pub/Sub as the distributed task queue service. It publishes crawling tasks to the **crawl_topic_path**, ensuring crawler nodes can pick them up from the queue.

2. Logic for Dividing Seed URLs into Crawling Tasks

- Code



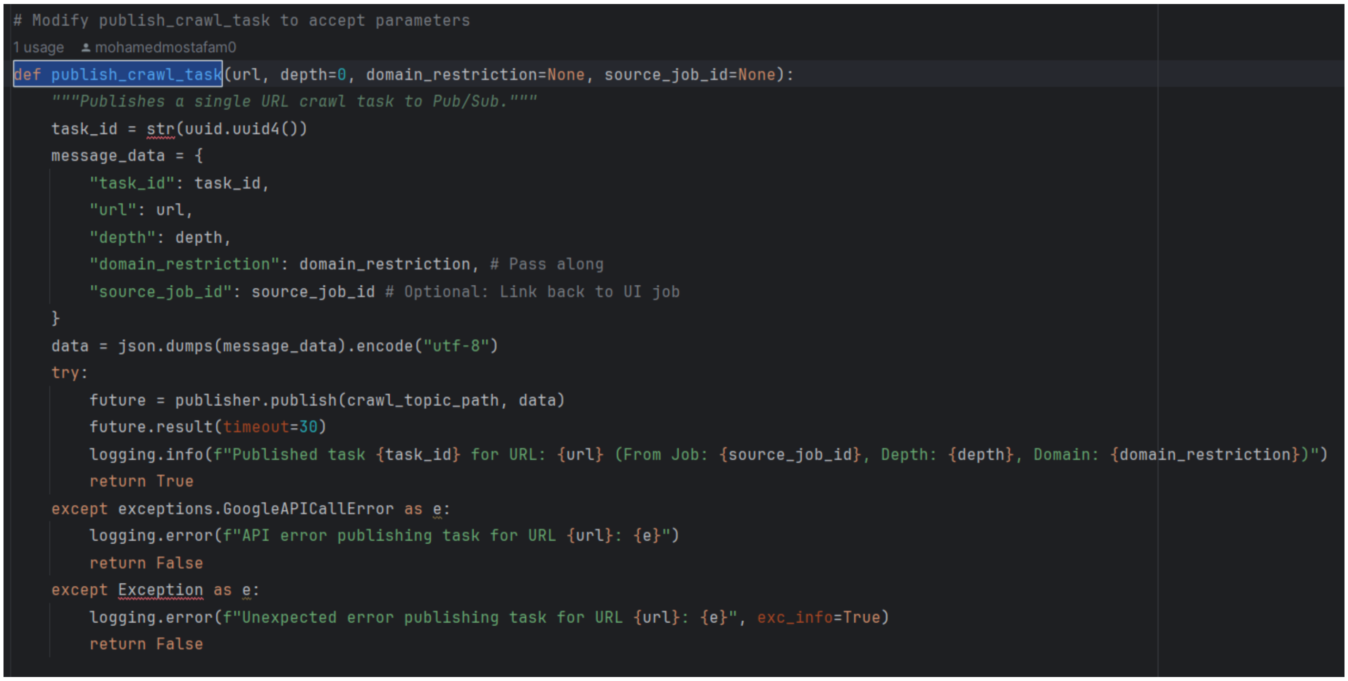
```python
for url in seed_urls:
    publish_crawl_task(
        url,
        depth=0,
        domain_restriction=domain_restriction,
        source_job_id=task_id
    )
    time.sleep(0.05)
```

Figure 8:Logic for dividing URLs into Crawling Tasks

- **Explanation:** The for loop iterates through the list of seed URLs, and each URL is converted into a crawl task using the **publish_crawl_task** function. Additional metadata like depth and domain restrictions are included.

3. Task Assignment to Crawler Nodes via the Task Queue

- Code



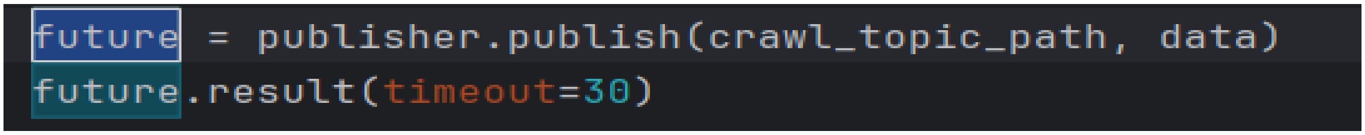
```python
# Modify publish_crawl_task to accept parameters
def publish_crawl_task(url, depth=0, domain_restriction=None, source_job_id=None):
    """Publishes a single URL crawl task to Pub/Sub."""
    task_id = str(uuid.uuid4())
    message_data = {
        "task_id": task_id,
        "url": url,
        "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)
        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
```

Figure 9:Task Assignment to Crawler Nodes via the Task Queue

- **Explanation:** This function encapsulates the logic for assigning tasks to crawler nodes via Pub/Sub. Each task contains metadata such as the URL to crawl, the crawl depth, domain restrictions, and a task ID. The tasks are added to the Pub/Sub queue.

4. Basic Crawler Node Lifecycle Management (Start, Stop, Monitoring)

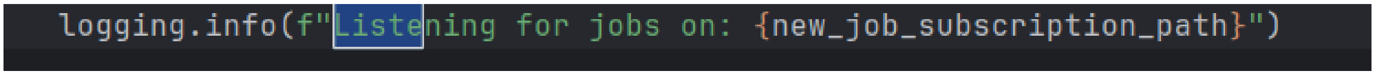
- Code for Start and Stop Monitoring:



```
future = publisher.publish(crawl_topic_path, data)
future.result(timeout=30)
```

Figure 10:Start and Stop Monitoring

- **Explanation:** This subscription ensures the Master Node actively listens for incoming job requests, signaling the start of task processing. The **future.cancel()** method allows graceful shutdown of the Master Node if needed.
- Code for Basic Monitoring (Heartbeats, Status):



```
    logging.info(f"Listening for jobs on: {new_job_subscription_path}")
```

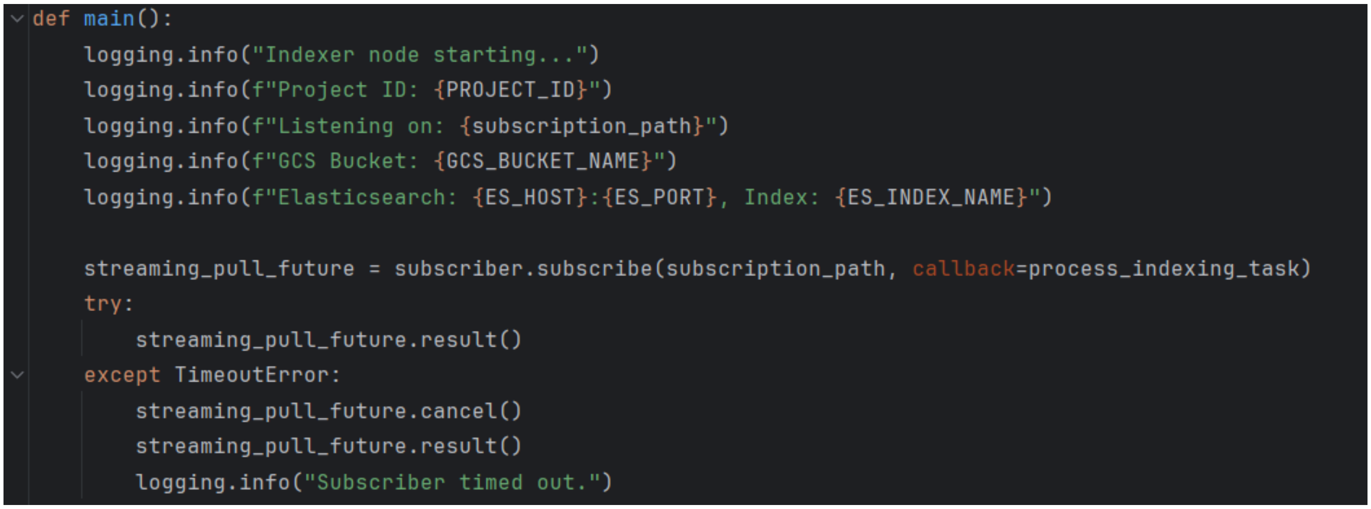
Figure 11:Basic Monitoring(Heartbeats)

- **Explanation:** While the Master Node doesn't fully implement advanced monitoring yet, the logging system tracks the activity of the task processing pipeline. Adding detailed monitoring (e.g., periodic status checks for crawler nodes) would enhance lifecycle management in future phases.

Basic Indexer Implementation

1. Implement a Single Indexer Node for Simplicity Initially

- Code



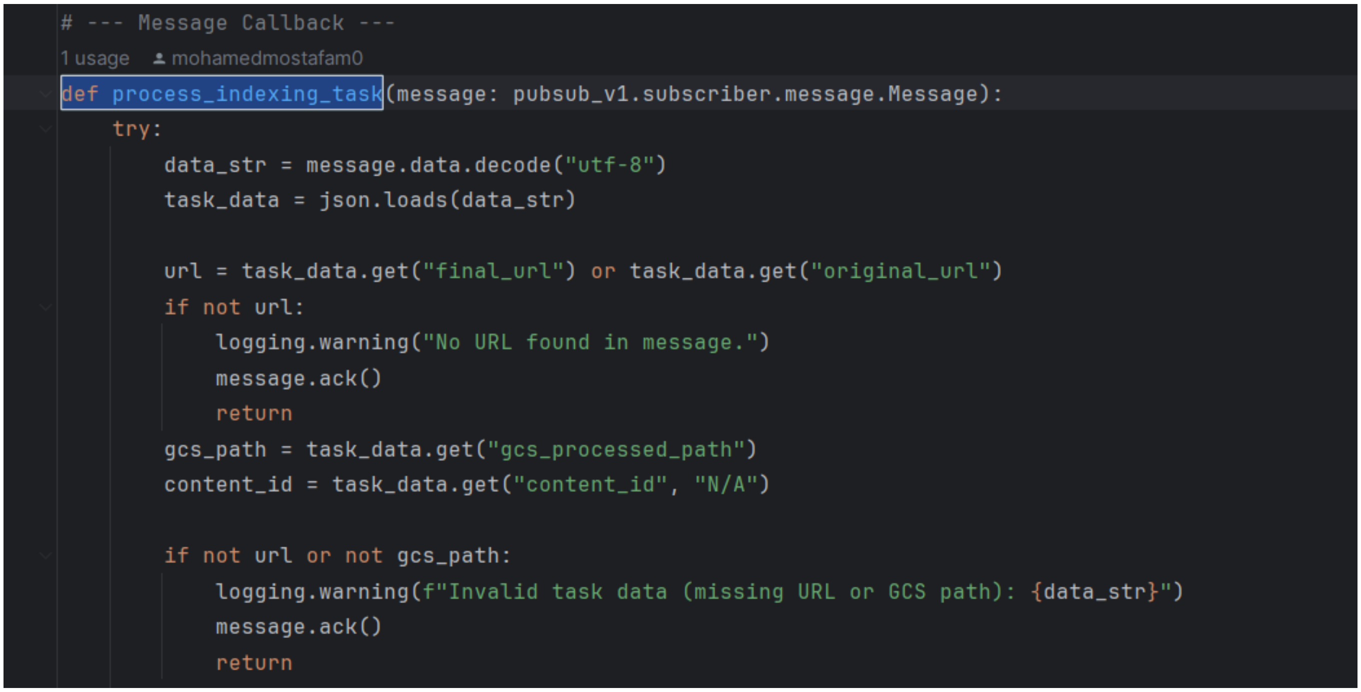
```
def main():
    logging.info("Indexer node starting...")
    logging.info(f"Project ID: {PROJECT_ID}")
    logging.info(f"Listening on: {subscription_path}")
    logging.info(f"GCS Bucket: {GCS_BUCKET_NAME}")
    logging.info(f"Elasticsearch: {ES_HOST}:{ES_PORT}, Index: {ES_INDEX_NAME}")

    streaming_pull_future = subscriber.subscribe(subscription_path, callback=process_indexing_task)
    try:
        streaming_pull_future.result()
    except TimeoutError:
        streaming_pull_future.cancel()
        streaming_pull_future.result()
        logging.info("Subscriber timed out.")
```

Figure 12:Single Indexer Node Implementation for Simplicity Initially

- **Explanation:** The code defines a single Indexer Node that listens for incoming tasks via the Pub/Sub subscription. This simple setup ensures that indexing tasks are processed by one node to fulfill the requirement of a single Indexer Node in Phase 2.

2. Implement Data Ingestion from Crawler Nodes

- Code

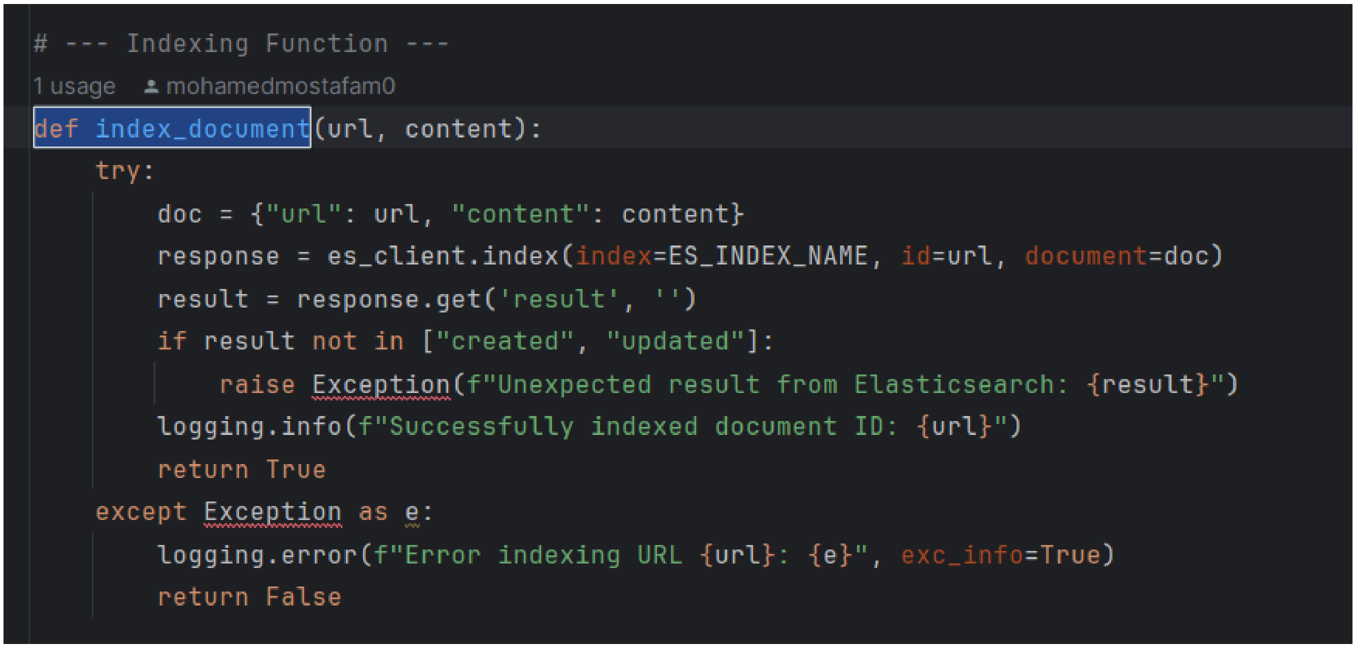


```python
# --- Message Callback ---
def process_indexing_task(message: pubsub_v1.subscriber.message.Message):
    try:
        data_str = message.data.decode("utf-8")
        task_data = json.loads(data_str)

        url = task_data.get("final_url") or task_data.get("original_url")
        if not url:
            logging.warning("No URL found in message.")
            message.ack()
            return
        gcs_path = task_data.get("gcs_processed_path")
        content_id = task_data.get("content_id", "N/A")

        if not url or not gcs_path:
            logging.warning(f"Invalid task data (missing URL or GCS path): {data_str}")
            message.ack()
            return
```

Figure 13:Data Ingestion from Crawler Nodes

- **Explanation:** This part of the code ingests data from the Crawler Nodes by receiving messages via the Pub/Sub topic and downloading the processed content from Google Cloud Storage (GCS). The ingestion is straightforward and handles the required communication between the Crawler and Indexer nodes.

3. Implement Index using ElasticSearch



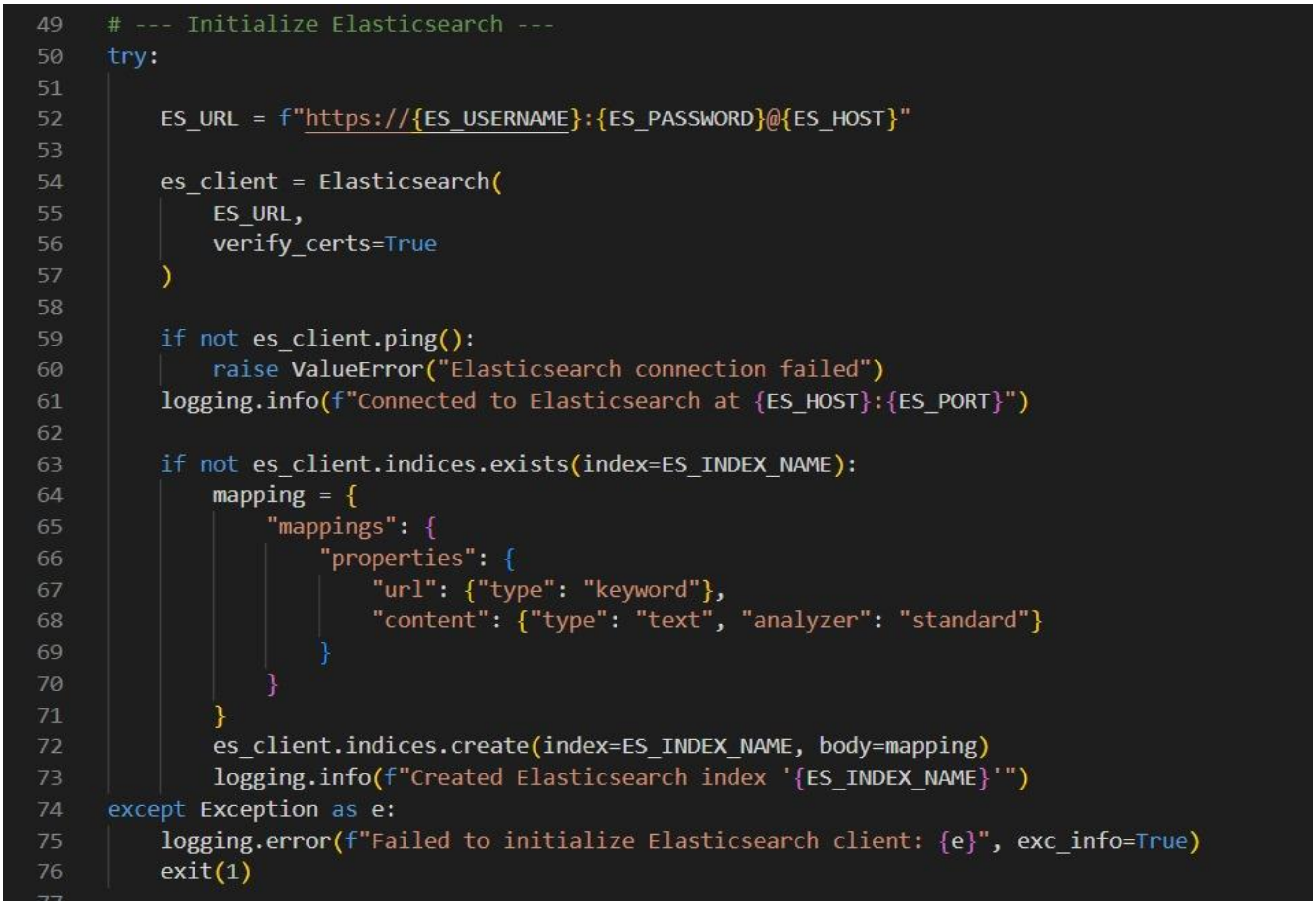
```python
# --- Indexing Function ---
def index_document(url, content):
    try:
        doc = {"url": url, "content": content}
        response = es_client.index(index=ES_INDEX_NAME, id=url, document=doc)
        result = response.get('result', '')
        if result not in ["created", "updated"]:
            raise Exception(f"Unexpected result from Elasticsearch: {result}")
        logging.info(f"Successfully indexed document ID: {url}")
        return True
    except Exception as e:
        logging.error(f"Error indexing URL {url}: {e}", exc_info=True)
        return False
```

Figure 14: Index Implementation using Elastic Search

- **Explanation:** The code uses Elasticsearch to create an index, associating each crawled URL with its processed content. While Elasticsearch is more robust than a basic in-memory dictionary or file-based index, this implementation serves as a more scalable alternative for storing keywords and associating them with URLs.

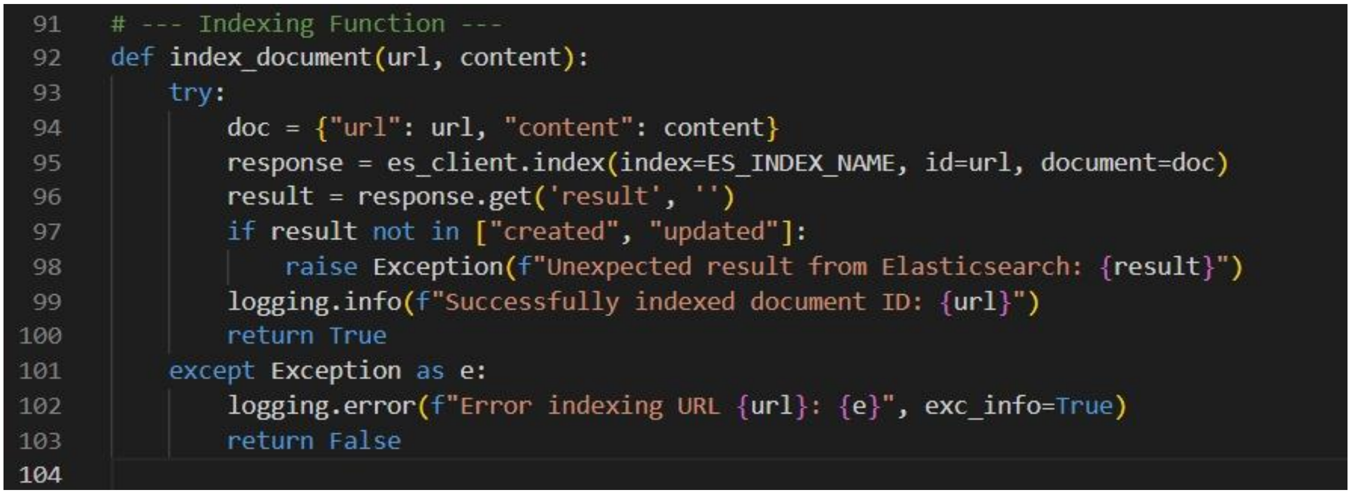
4. Implement a Very Simple Keyword Search Functionality

- **Code:** Search functionality, Elasticsearch's default capabilities allow keyword-based search with exact matching:



```python
# --- 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 15:Initialize ElasticSearch



```python
# --- Indexing Function ---
def index_document(url, content):
    try:
        doc = {"url": url, "content": content}
        response = es_client.index(index=ES_INDEX_NAME, id=url, document=doc)
        result = response.get('result', '')
        if result not in ["created", "updated"]:
            raise Exception(f"Unexpected result from Elasticsearch: {result}")
        logging.info(f"Successfully indexed document ID: {url}")
        return True
    except Exception as e:
        logging.error(f"Error indexing URL {url}: {e}", exc_info=True)
        return False

```

Figure 16:Indexing Function

- **Explanation:** The function uses Elasticsearch to perform a basic search for exact keywords in the indexed content. This satisfies the requirement for simple keyword search functionality. Elasticsearch provides this feature inherently.

Communication and Workflow Integration

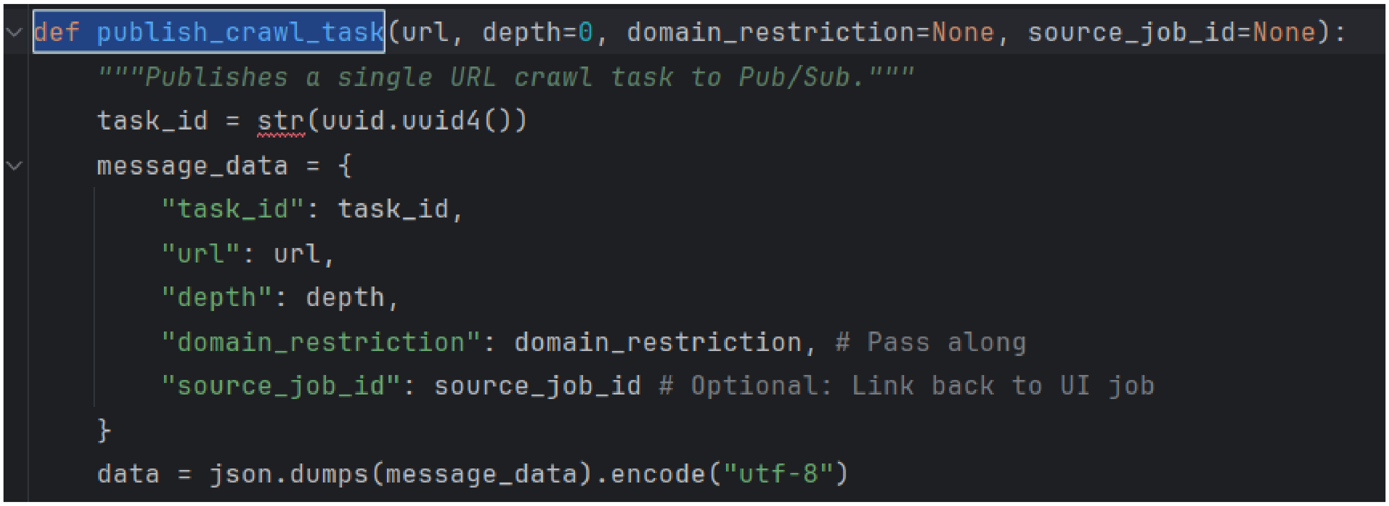
Overview

The communication flow between the system components ensures smooth operation and efficient task distribution. For Phase 2, the communication flow integrates the following interactions:

- **Master Node → Task Queue → Crawlers:** The Master Node distributes crawling tasks using a cloud-based task queue (Google Pub/Sub). Crawlers subscribe to the task queue to receive assignments.
- **Crawlers → Indexer Node (via queue):** Once crawlers process and extract content, they report results (crawled data and extracted URLs) back to the Indexer Node using the queue.
- **Client (for monitoring) → Master Node:** The Master Node acts as a central control point, where clients can interact to monitor the system's progress or initiate crawling operations.

Details in the Code

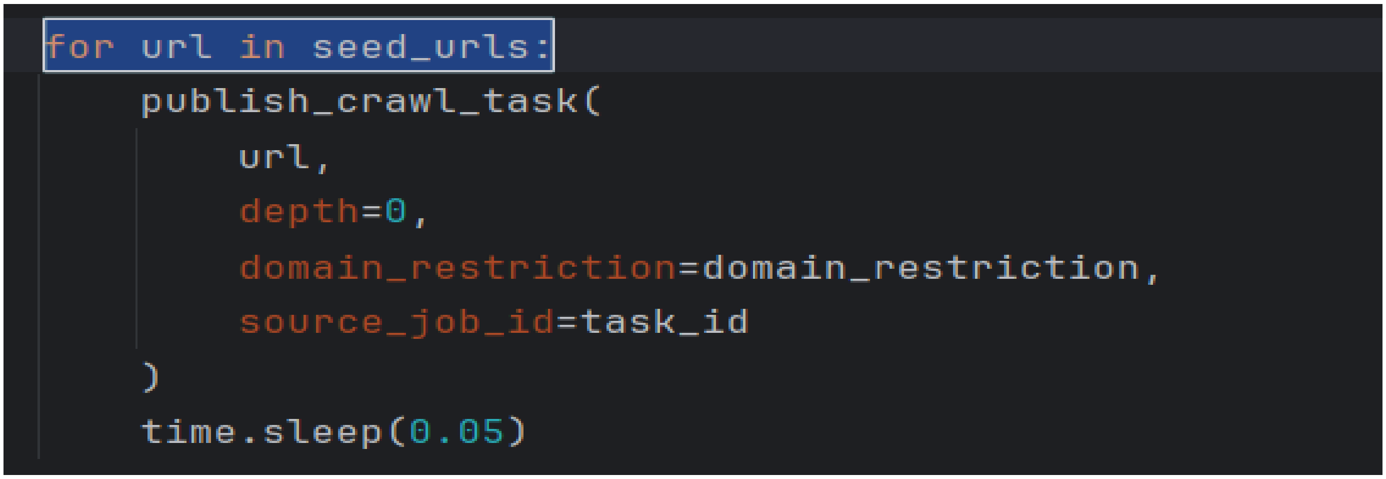
1. Master → Task Queue → Crawlers:
   - **Publishing Crawl Tasks:** In the Master Node code, the **publish_crawl_task** function demonstrates how tasks are sent to the task queue for crawlers to retrieve.



```python
def publish_crawl_task(url, depth=0, domain_restriction=None, source_job_id=None):
    """Publishes a single URL crawl task to Pub/Sub."""
    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")
```

Figure 17:Publish_crawl_task

- **Task Distribution Logic:** Seed URLs are divided into crawl tasks and published via the **crawl_topic_path**



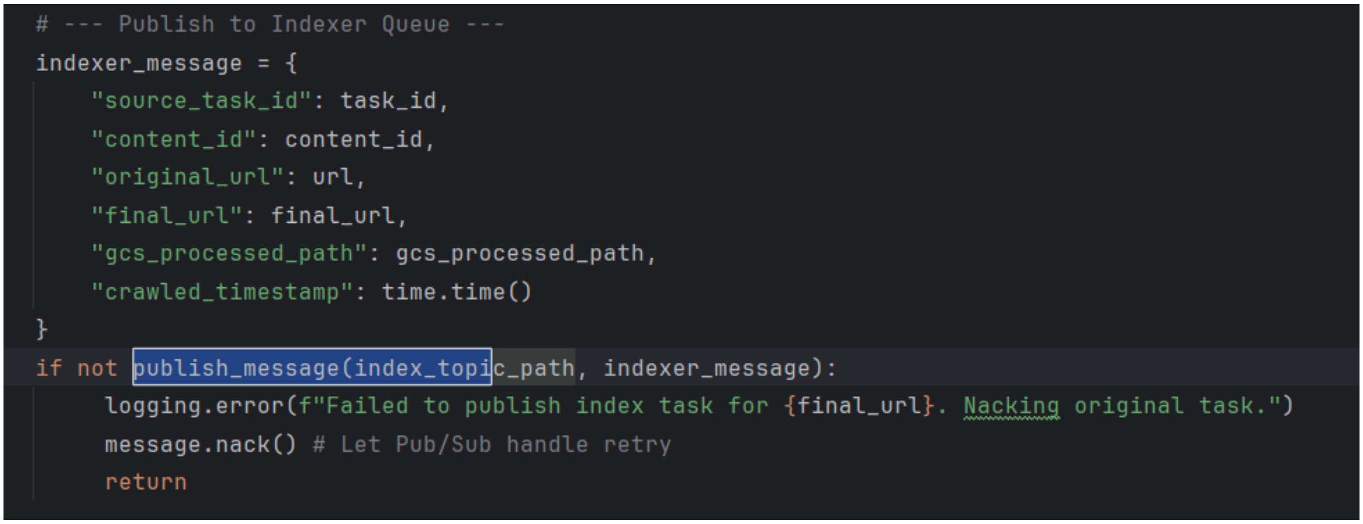
```
for url in seed_urls:
    publish_crawl_task(
        url,
        depth=0,
        domain_restriction=domain_restriction,
        source_job_id=task_id
    )
    time.sleep(0.05)
```

Figure 18:Task Distribution Logic

- **Interaction with Crawlers:** The crawlers subscribe to the topic and receive the tasks published by the Master Node.

2. Crawlers → Indexer (via Queue)

- **Reporting Crawled Content and Extracted URLs:** In the Crawler Node code, crawled content and metadata are sent to the Indexer Node using Pub/Sub.



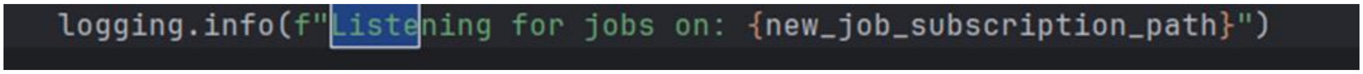
```
# --- Publish to Indexer Queue ---
indexer_message = {
    "source_task_id": task_id,
    "content_id": content_id,
    "original_url": url,
    "final_url": final_url,
    "gcs_processed_path": gcs_processed_path,
    "crawled_timestamp": time.time()
}
if not publish_message(index_topic_path, indexer_message):
     logging.error(f"Failed to publish index task for {final_url}. Nacking original task.")
     message.nack() # Let Pub/Sub handle retry
     return
```

Figure 19:Reporting Crawled Content and Extracted URLs

- **Interaction with Indexer:** The crawlers send processed data (text content) to the **index_topic_path.** The Indexer Node subscribes to this topic and ingests the data.

3. Client → Master (Basic Monitoring)

- **Logging System:** The Master Node includes logging for tracking progress, which serves as the basis for monitoring by the client.



```
  logging.info(f"Listening for jobs on: {new_job_subscription_path}")
```

Figure 20:Logging System

- **Interaction:** Clients could access logs or implement simple scripts/tools to interact with the Master Node, monitor task assignments, and check crawling progress.

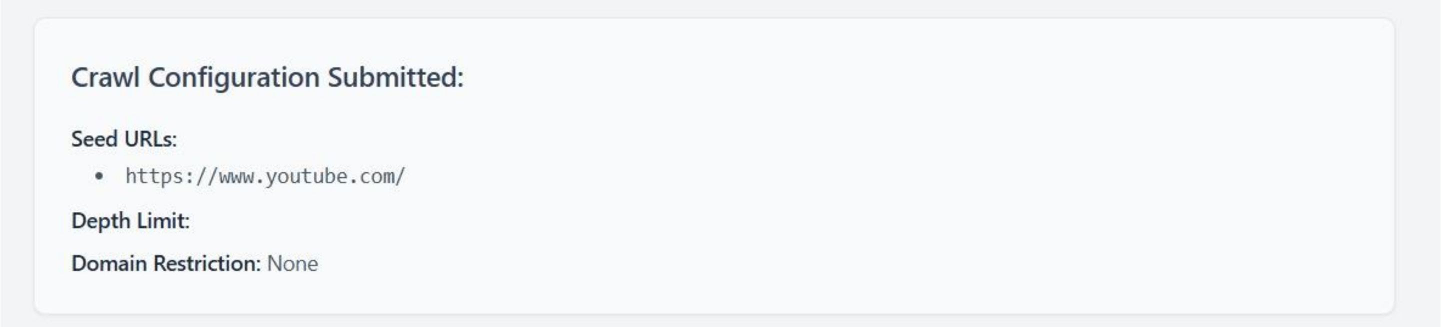
4. Integrating Components into a Functional Pipeline
   Workflow Summary:

- The **Master Node** divides the task into smaller units, publishes them to the queue, and assigns them to available crawler nodes.
- The **Crawler Nodes** fetch web pages, extract text and links, and send processed data to the queue for the Indexer Node.
- The **Indexer Node** ingests the processed content and indexes it for keyword-based searches.
- **Clients** monitor the pipeline via logs or interfaces connected to the Master Node.

Initial Testing and Debugging

Testing Integration:

Running small-scale tests with seed URLs to ensure task flow (Master → Crawlers → Indexer) and indexing functionality.



Crawl Configuration Submitted:

Seed URLs:

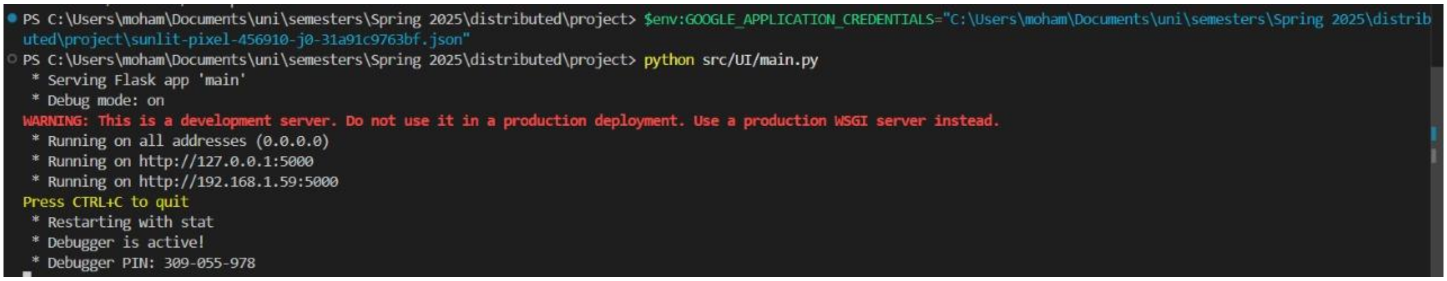
- `https://www.youtube.com/`

Depth Limit:

Domain Restriction: None

Figure 21:"Web Crawling Configuration Successfully Submitted"

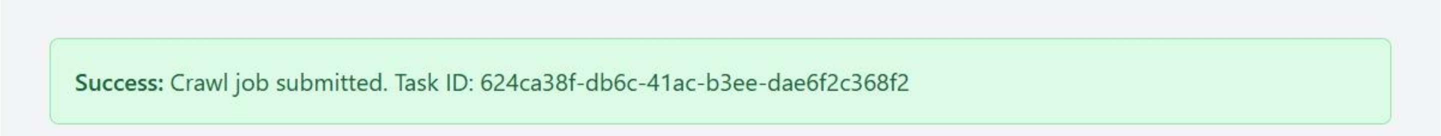
Description: The screenshot showcases a web crawling configuration interface, confirming that the process has been initiated. The setup includes **https://www.youtube.com/** as the seed URL, with no depth limit or domain restrictions applied. These parameters define how the crawler navigates and gathers data, ensuring flexibility in discovering content across domains. The confirmation message indicates that the system has successfully stored the configuration, marking a key milestone in Phase 2 of testing.



```
PS C:\Users\moham\Documents\uni\semesters\Spring 2025\distributed\project> $env:GOOGLE_APPLICATION_CREDENTIALS="C:\Users\moham\Documents\uni\semesters\Spring 2025\distrib
uted\project\sunlit-pixel-456910-j0-31a91c9763bf.json"
PS C:\Users\moham\Documents\uni\semesters\Spring 2025\distributed\project> python src/UI/main.py
 * Serving Flask app 'main'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5000
 * Running on http://192.168.1.59:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 309-055-978
```

Figure 22: "Flask Application Successfully Started"

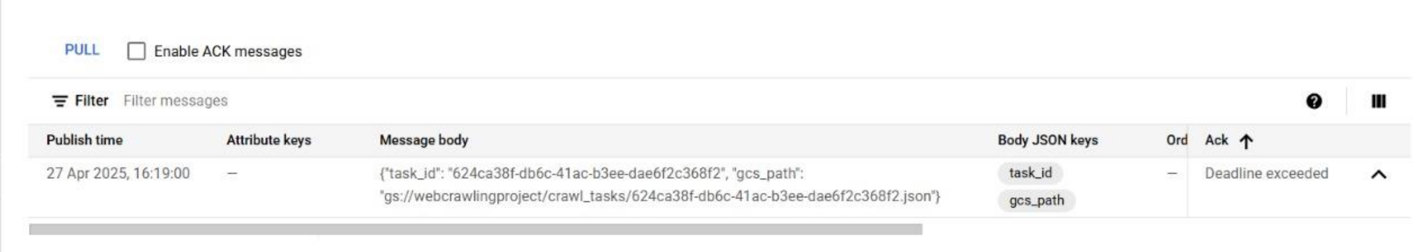
Description: This screenshot captures the initiation of a Flask-based web application in a development environment. The terminal shows the environment variable setup for Google Application Credentials, ensuring authentication for Google services. The Flask app is started via **python src/UI/main.py**, running in **debug mode** for testing. The server is accessible locally at **127.0.0.1:5000** and on the network at **192.168.1.59:5000**. The warning about using a production WSGI server emphasizes security concerns when moving beyond local development.



Success: Crawl job submitted. Task ID: 624ca38f-db6c-41ac-b3ee-dae6f2c368f2

Figure 23:"Web Crawling Task Successfully Submitted"

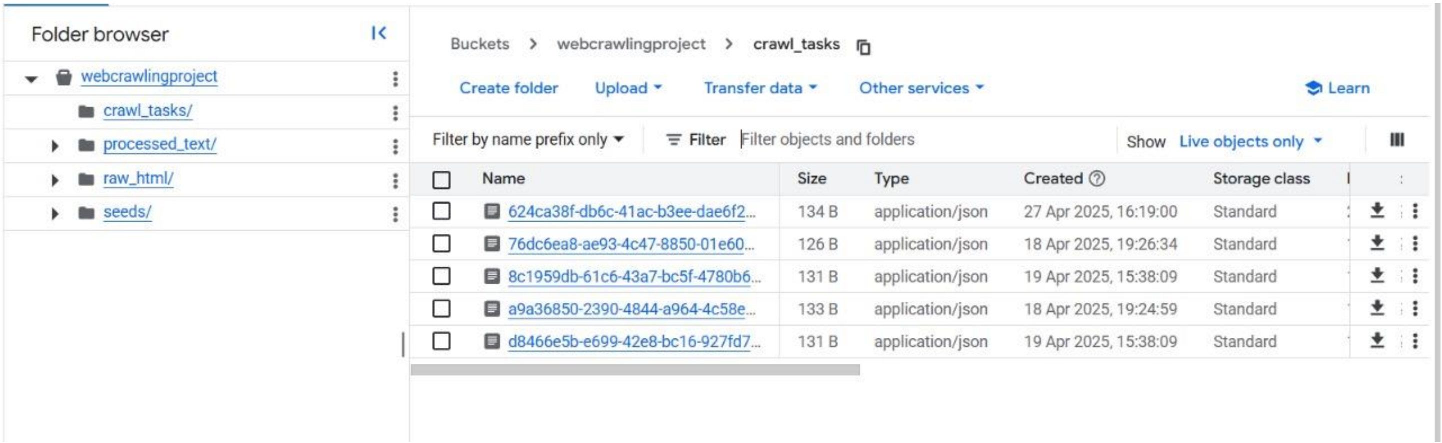
Description: The image displays a confirmation message indicating that a web crawling job has been successfully initiated. The green notification panel presents a success message with a unique **Task ID: 624ca38f-db6c-41ac-b3ee-dae6f2c368f2**, which can be used for tracking progress and managing subsequent operations. This confirmation highlights the seamless execution of the web crawling process in Phase 2, ensuring smooth data extraction and system functionality.



Publish time	Attribute keys	Message body	Body JSON keys	Ord	Ack ↑
27 Apr 2025, 16:19:00	–	{"task_id": "624ca38f-db6c-41ac-b3ee-dae6f2c368f2", "gcs_path": "gs://webcrawlingproject/crawl_tasks/624ca38f-db6c-41ac-b3ee-dae6f2c368f2.json"}	task_id gcs_path	–	Deadline exceeded

Figure 24:"Message Queue Processing: Web Crawling Task Acknowledgment"

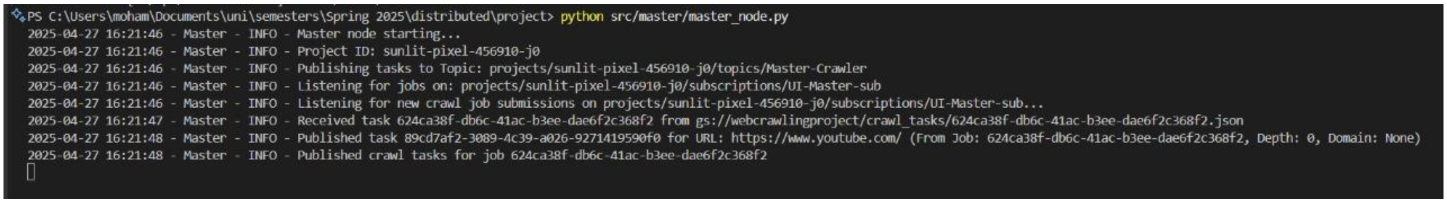
Description: The screenshot showcases a cloud-based message queue interface, displaying details of a published message related to a web crawling task. The table records attributes such as the **task_id,** timestamp, and storage path within Google Cloud Storage **(gcs_path)**. The message contains metadata confirming the initiation of a crawl request, enabling distributed processing. However, an "ACK: Deadline exceeded" status suggests a delay in acknowledgment, requiring further investigation. This visualization is crucial for Phase 2 testing, ensuring seamless task execution and timely message handling.



Name	Size	Type	Created	Storage class
624ca38f-db6c-41ac-b3ee-dae6f2...	134 B	application/json	27 Apr 2025, 16:19:00	Standard
76dc6ea8-ae93-4c47-8850-01e60...	126 B	application/json	18 Apr 2025, 19:26:34	Standard
8c1959db-61c6-43a7-bc5f-4780b6...	131 B	application/json	19 Apr 2025, 15:38:09	Standard
a9a36850-2390-4844-a964-4c58e...	133 B	application/json	18 Apr 2025, 19:24:59	Standard
d8466e5b-e699-42e8-bc16-927fd7...	131 B	application/json	19 Apr 2025, 15:38:09	Standard

Figure 25:"Organized Folder Structure for Web Crawling Project"

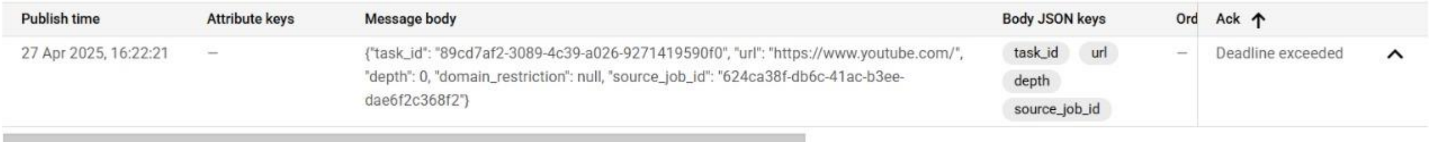
Description: The screenshot showcases a cloud-based storage interface displaying the structured organization of a web crawling project. The folder hierarchy includes key directories such as **crawl_tasks, processed_text, raw_html**, and **seeds**, ensuring efficient data management. Within the **crawl_tasks** folder, multiple JSON files are visible, each representing individual crawl requests with metadata such as creation timestamps and storage details. This structured layout is crucial in Phase 2 testing, facilitating systematic retrieval, processing, and storage of crawling results.



```
PS C:\Users\moham\Documents\uni\semesters\Spring 2025\distributed\project> python src/master/master_node.py
  2025-04-27 16:21:46 - Master - INFO - Master node starting...
  2025-04-27 16:21:46 - Master - INFO - Project ID: sunlit-pixel-456910-j0
  2025-04-27 16:21:46 - Master - INFO - Publishing tasks to Topic: projects/sunlit-pixel-456910-j0/topics/Master-Crawler
  2025-04-27 16:21:46 - Master - INFO - Listening for jobs on: projects/sunlit-pixel-456910-j0/subscriptions/UI-Master-sub
  2025-04-27 16:21:46 - Master - INFO - Listening for new crawl job submissions on projects/sunlit-pixel-456910-j0/subscriptions/UI-Master-sub...
  2025-04-27 16:21:47 - Master - INFO - Received task 624ca38f-db6c-41ac-b3ee-dae6f2c368f2 from gs://webcrawlingproject/crawl_tasks/624ca38f-db6c-41ac-b3ee-dae6f2c368f2.json
  2025-04-27 16:21:48 - Master - INFO - Published task 89cd7af2-3089-4c39-a026-9271419590f0 for URL: https://www.youtube.com/ (From Job: 624ca38f-db6c-41ac-b3ee-dae6f2c368f2, Depth: 0, Domain: None)
  2025-04-27 16:21:48 - Master - INFO - Published crawl tasks for job 624ca38f-db6c-41ac-b3ee-dae6f2c368f2
```

Figure 26:Caption: "Master Node Successfully Processing Web Crawling Tasks"

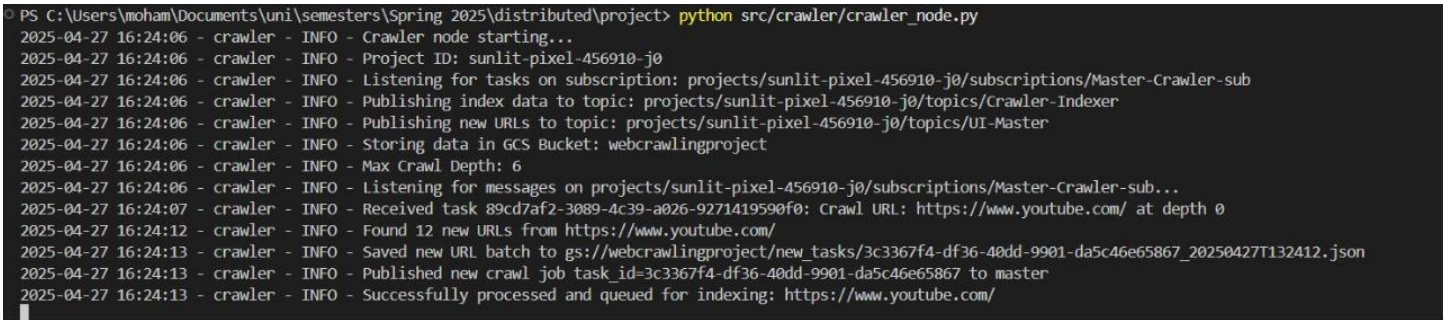
Description: The screenshot displays a terminal window where the Master Node is actively managing web crawling jobs in a distributed system. The log messages indicate the project's ID, task publishing status, and job tracking through Google Cloud Storage **(gs://webcrawlingproject/crawl_tasks).** The node listens for new job submissions and processes tasks efficiently. Notably, a crawl task has been published for **https://www.youtube.com/** with specified parameters, marking a critical milestone in Phase 2 testing by ensuring smooth task orchestration and distribution.



Publish time	Attribute keys	Message body	Body JSON keys	Ord	Ack ↑
27 Apr 2025, 16:22:21	–	{"task_id": "89cd7af2-3089-4c39-a026-9271419590f0", "url": "https://www.youtube.com/", "depth": 0, "domain_restriction": null, "source_job_id": "624ca38f-db6c-41ac-b3ee-dae6f2c368f2"}	task_id url depth source_job_id	–	Deadline exceeded

Figure 27:"Message Queue Processing: Web Crawling Task Published"

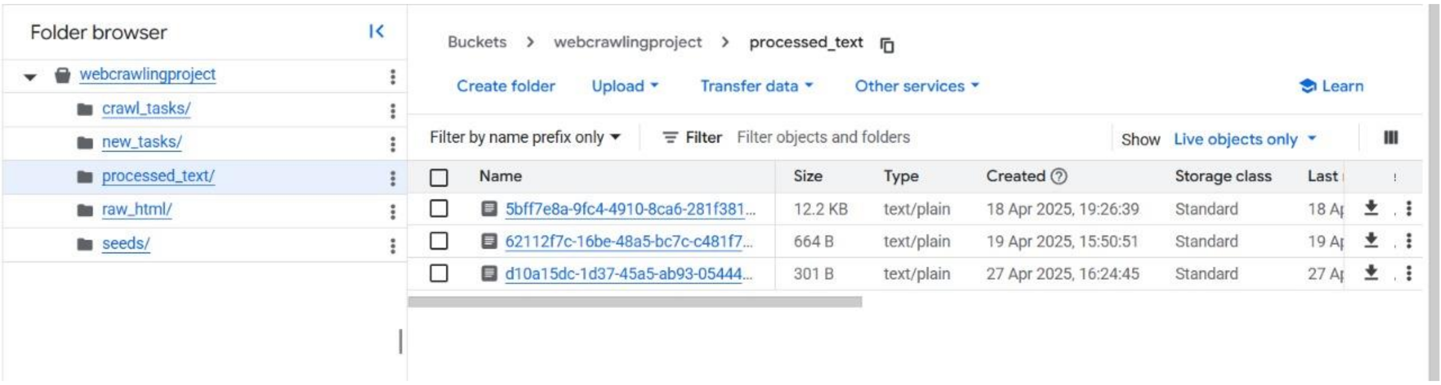
Description: The screenshot presents a table from a cloud-based message queue system, highlighting a published message related to a web crawling task. The JSON body contains essential details such as the task ID, the target **URL (https://www.youtube.com/),** a depth value of **0,** and a reference to the source job ID. These elements confirm that the web crawling request has been successfully dispatched. However, the acknowledgment status notes a "Deadline exceeded" warning, indicating a delay in task confirmation and requiring further assessment to ensure smooth execution. This screenshot is crucial for Phase 2 testing, as it validates the message queue's ability to handle and distribute tasks in a distributed system.



```
PS C:\Users\moham\Documents\uni\semesters\Spring 2025\distributed\project> python src/crawler/crawler_node.py
2025-04-27 16:24:06 - crawler - INFO - Crawler node starting...
2025-04-27 16:24:06 - crawler - INFO - Project ID: sunlit-pixel-456910-j0
2025-04-27 16:24:06 - crawler - INFO - Listening for tasks on subscription: projects/sunlit-pixel-456910-j0/subscriptions/Master-Crawler-sub
2025-04-27 16:24:06 - crawler - INFO - Publishing index data to topic: projects/sunlit-pixel-456910-j0/topics/Crawler-Indexer
2025-04-27 16:24:06 - crawler - INFO - Publishing new URLs to topic: projects/sunlit-pixel-456910-j0/topics/UI-Master
2025-04-27 16:24:06 - crawler - INFO - Storing data in GCS Bucket: webcrawlingproject
2025-04-27 16:24:06 - crawler - INFO - Max Crawl Depth: 6
2025-04-27 16:24:06 - crawler - INFO - Listening for messages on projects/sunlit-pixel-456910-j0/subscriptions/Master-Crawler-sub...
2025-04-27 16:24:07 - crawler - INFO - Received task 89cd7af2-3089-4c39-a026-9271419590f0: Crawl URL: https://www.youtube.com/ at depth 0
2025-04-27 16:24:12 - crawler - INFO - Found 12 new URLs from https://www.youtube.com/
2025-04-27 16:24:13 - crawler - INFO - Saved new URL batch to gs://webcrawlingproject/new_tasks/3c3367f4-df36-40dd-9901-da5c46e65867_20250427T132412.json
2025-04-27 16:24:13 - crawler - INFO - Published new crawl job task_id=3c3367f4-df36-40dd-9901-da5c46e65867 to master
2025-04-27 16:24:13 - crawler - INFO - Successfully processed and queued for indexing: https://www.youtube.com/
```

Figure 28:"Web Crawler Node Successfully Processing Crawl Jobs"

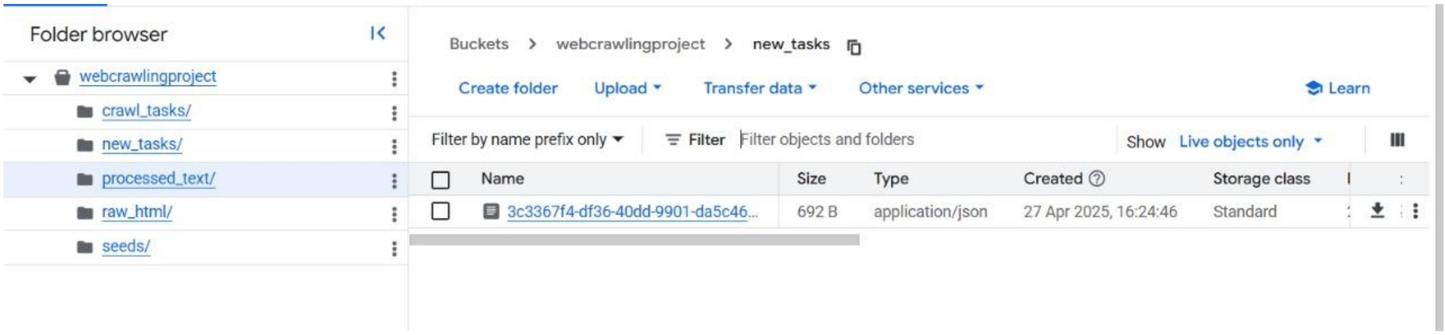
Description: The screenshot captures a terminal session running a web crawler node, actively retrieving and processing URLs. The logs confirm the crawler is executing tasks within a distributed system, interfacing with Google Cloud services for data storage and messaging. Task IDs and timestamps indicate the progression of operations—starting the node, receiving crawl requests, discovering new URLs, and storing batches in a cloud storage bucket **(gs://webcrawlingproject).** The system successfully queued URLs for indexing and further crawling, marking a vital step in Phase 2 testing.



☐	Name	Size	Type	Created	Storage class	Last
☐	5bff7e8a-9fc4-4910-8ca6-281f381...	12.2 KB	text/plain	18 Apr 2025, 19:26:39	Standard	18 A
☐	62112f7c-16be-48a5-bc7c-c481f7...	664 B	text/plain	19 Apr 2025, 15:50:51	Standard	19 A
☐	d10a15dc-1d37-45a5-ab93-05444...	301 B	text/plain	27 Apr 2025, 16:24:45	Standard	27 A

Figure 29:"Organized Cloud Storage for Processed Web Crawling Text"

Description: The screenshot displays a folder browser interface managing processed text data within the **webcrawlingproject** storage system. The left panel shows a structured directory, including **crawl_tasks, new_tasks, processed_text, raw_html,** and **seeds.** The **processed_text** folder is selected, revealing a list of three files named **5bf7e8a-9fc4-4910-8ca6-281f381..., 62112f7c-16be-48a5-bc7c-c481f7..., and d10a15dc-1d37-45a5-ab93-05444....** Each file contains text extracted from web crawling tasks, with metadata specifying size, format **(text/plain),** creation timestamps, and cloud storage class **(Standard).** This systematic organization ensures efficient access to processed data, supporting seamless information retrieval and validation during Phase 2 testing.

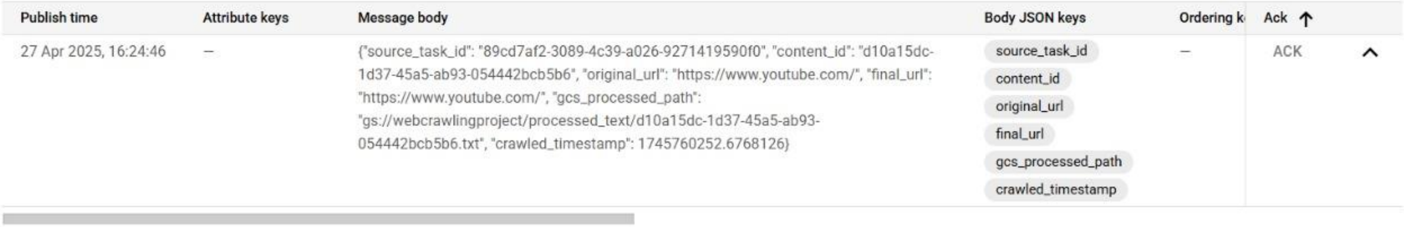

Folder browser
Buckets > webcrawlingproject > new_tasks
Create folder
Upload
Transfer data
Other services
Learn
Filter by name prefix only
Filter
Filter objects and folders
Show
Live objects only

- webcrawlingproject
  - crawl_tasks/
  - new_tasks/
  - processed_text/
  - raw_html/
  - seeds/

Name	Size	Type	Created	Storage class
3c3367f4-df36-40dd-9901-da5c46...	692 B	application/json	27 Apr 2025, 16:24:46	Standard

Figure 30:"Organized Cloud Storage for Web Crawling Tasks"

Description: The screenshot presents a cloud storage interface showcasing a structured folder hierarchy within the **webcrawlingproject** bucket. The directory contains essential subfolders—**crawl_tasks, new_tasks, processed_text, raw_html**, and **seeds**—each serving a distinct role in managing web crawling data. The **new_tasks** folder is currently selected, revealing a JSON file named **3c3367f4-df36-40dd-9901-da5c46...,** indicating a newly created web crawling task. Metadata details confirm its size **(692 B),** type **(application/json)**, creation timestamp **(27 Apr 2025, 16:24:46)**, and storage class **(Standard**). This well-organized structure supports efficient workflow management during Phase 2 testing.



Publish time	Attribute keys	Message body	Body JSON keys	Ordering k	Ack
27 Apr 2025, 16:24:46	–	{"source_task_id": "89cd7af2-3089-4c39-a026-9271419590f0", "content_id": "d10a15dc-1d37-45a5-ab93-054442bcb5b6", "original_url": "https://www.youtube.com/", "final_url": "https://www.youtube.com/", "gcs_processed_path": "gs://webcrawlingproject/processed_text/d10a15dc-1d37-45a5-ab93-054442bcb5b6.txt", "crawled_timestamp": 1745760252.6768126}	source_task_id content_id original_url final_url gcs_processed_path crawled_timestamp	–	ACK

Figure 31:"Web Crawling Task Successfully Acknowledged in Message Queue"

Description: The screenshot showcases a cloud-based message queue displaying details of a processed web crawling task. The message body contains structured JSON data, including the source task **ID (89cd7af2-3089-4c39-a026-9271419590f0)**, content **ID (d10a15dc-1d37-45a5-ab93-054442bcb5b6),** and key URLs (original and final). The message confirms that processed text is stored in Google Cloud Storage **(gs://webcrawlingproject/processed_text/d10a15dc-1d37-45a5-ab93-054442bcb5b6.txt**).The acknowledgment status **(ACK)** ensures that the task was successfully received and processed, marking a critical validation step in Phase 2 of testing.

Conclusion

Phase 2 marks a significant milestone in the development of our distributed web crawling and indexing system. During this phase, we successfully implemented the core components of the crawling and indexing pipeline, enabling the system to function as a distributed framework. The integration of the Master Node, Crawler Nodes, and Indexer Node facilitated the completion of essential functionalities, including task distribution, polite and efficient crawling, and basic indexing.

Key accomplishments include:

- The establishment of communication and workflow integration through the use of Google Pub/Sub as a task queue, ensuring seamless task distribution and data flow between system components.
- The implementation of crawling capabilities, including web page fetching, HTML parsing, and URL extraction, while adhering to politeness measures such as crawl delays.
- The development of an Indexer Node capable of ingesting crawled content, storing it in Elasticsearch, and creating a searchable index.
- Initial testing to verify the pipeline's functionality and ensure proper integration of all components.

These achievements provide the foundation for future advancements, including enhanced fault tolerance, scalability, and more sophisticated indexing and search functionalities. The experience gained during this phase has highlighted challenges such as optimizing task distribution and managing system performance under increased workloads, which will guide our efforts in subsequent phases.

With the successful completion of Phase 2, the project is well-positioned to move toward its ultimate goal of developing a robust and scalable distributed web crawling and indexing system. The progress made thus far demonstrates the system's potential to meet the defined objectives while paving the way for further enhancements.

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