Distributed Web Crawling and Indexing System using Cloud Computing phase 4

Team 10

Name	ID
Alaa Yasser Fathy Bekhit	21P0408
Mohamed Ahmed Abdelraouf	2001038
Adham Osama Mohamed	22P0071
Seif Elden Samir Mohamed	2101524

Supervised By: Prof.Ayman Mohamed Bahaa Eldin

Google Drive Folder:
`drive.google.com/drive/folders/1TqauejRmq6-jAea4jWtBrJKI-em7ndAX`

Contents

List of Figures

1 Introduction to Distributed Crawling System

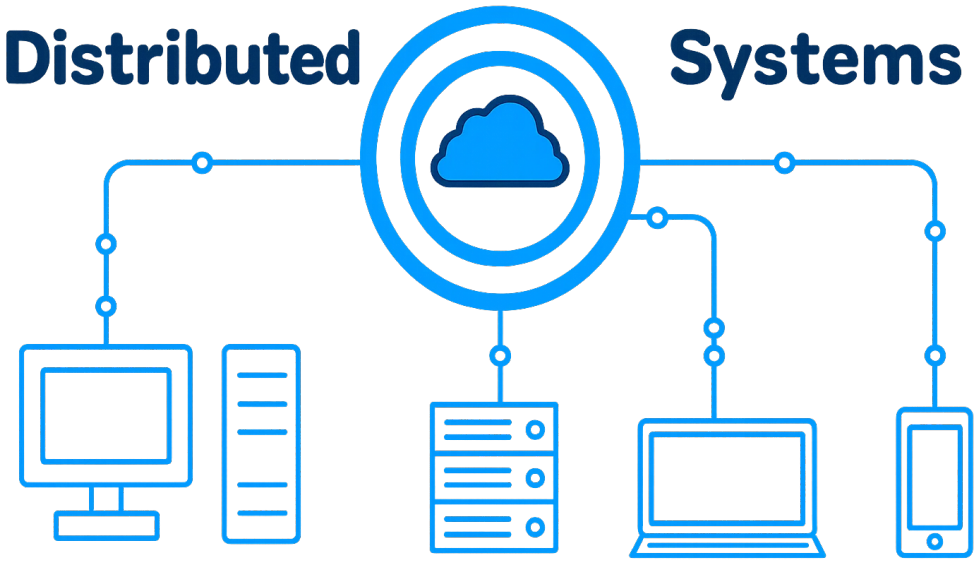


Figure 1: Intro to Distributed Systems

In the era of information overload, efficient and scalable web crawling systems have become essential for collecting, indexing, and retrieving relevant data from the vast landscape of the World Wide Web. This project presents the design and implementation of a distributed web crawling and indexing system built using cloud-native technologies such as AWS SQS, DynamoDB, and S3, combined with Python-based processing and indexing tools. The system is architected to support large-scale, fault-tolerant, and extensible operations through a modular composition of nodes: a centralized Master Node, distributed Crawler Nodes, and an Indexer Nodes.

Designed for scalability and resilience, the system incorporates heartbeat monitoring and dynamic task reassignment to recover from node failures. Furthermore, it supports user interaction via a client web interface for submitting seed URLs and querying indexed content, and a monitoring dashboard for real-time visualization of system performance and crawler status. This architecture lays the groundwork for a robust, cloud-based information retrieval platform adaptable to diverse web data collection needs.

2 Detailed Project Description

2.1 Overview

This project, titled **Distributed Web Crawling and Indexing System using Cloud Computing**, is a robust, scalable solution designed to efficiently collect, store, and index vast amounts of web data. Built using a modular architecture and cloud-native services, it supports distributed task execution, fault tolerance, and interactive user interfaces for seamless web crawling and information retrieval.

The core objective is to design a system capable of crawling and indexing data from the internet in a decentralized and fault-resilient manner. The architecture supports real-time monitoring, intelligent task distribution, and extensible indexing mechanisms to serve as a foundation for large-scale data analytics and search systems.

2.2 Key Features

- **Distributed Architecture**: Modular separation of responsibilities across *Master*, *Crawler*, and *Indexer* nodes allows concurrent execution and system scalability.
- **Cloud Integration**: Utilizes AWS services such as **SQS** for decoupled task queuing, **S3** for persistent storage of raw data and index backups, and **DynamoDB** for real-time metadata management.
- **Indexing and Search**: Employs the **Whoosh** full-text search library to create a powerful and efficient local search index, supporting multi-field queries (e.g., title, description, keywords).
- **Fault Tolerance**: Integrates heartbeat signals and status monitoring to detect crawler node failures, allowing the Master to reassign tasks dynamically.
- **Asynchronous Communication**: Implements non-blocking communication through AWS SQS queues to handle large volumes of tasks and responses without system bottlenecks.
- **User Interaction Interfaces**:
  - **Client Web Interface** for submitting crawl requests and configuring crawl depth.
  - **Search Interface** to perform real-time queries over indexed content.
  - **Monitoring Dashboard** for live updates on system health, crawler progress, and data flow.
- **Content Filtering and Processing**: HTML content is sanitized and parsed for relevant text extraction before indexing, reducing noise and improving search quality.
- **Scalability**: New crawler nodes can be added without modifying existing components, enabling horizontal scaling as crawl demand increases.
- **Persistent Storage and Backup**: Raw crawl data and indexes are securely archived to S3 for future retrieval and disaster recovery.

2.3 System Modules

- **Master Node**: Central coordinator responsible for managing the URL frontier, distributing tasks, processing results, and handling system metadata.
- **Crawler Nodes**: Execute crawl tasks by downloading web pages, extracting content and links, and reporting results/status back to the Master node.
- **Indexer Node**: Processes crawled content into a structured, searchable format using Whoosh. Also performs backup of index files to cloud storage.
- **Client Web Interface**: Web-based frontend for users to initiate crawl jobs and specify parameters.
- **Search Interface**: Provides search functionality over the indexed documents, returning high-relevance results with titles, URLs, and descriptions.
- **Monitoring Dashboard**: Displays live system metrics including active nodes, processed URLs, system throughput, and crawler health status.

3 Beneficiaries of the Project

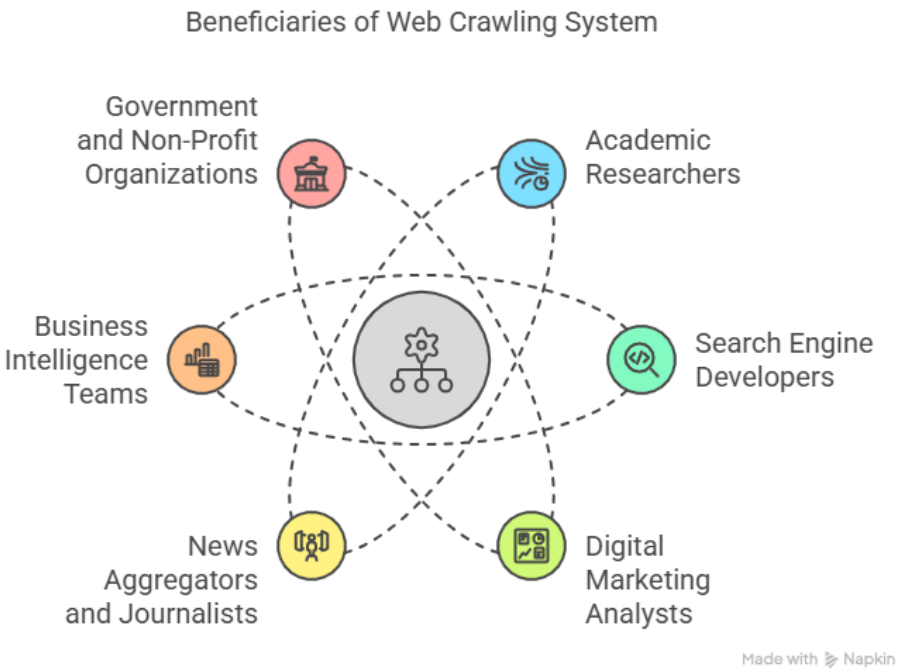


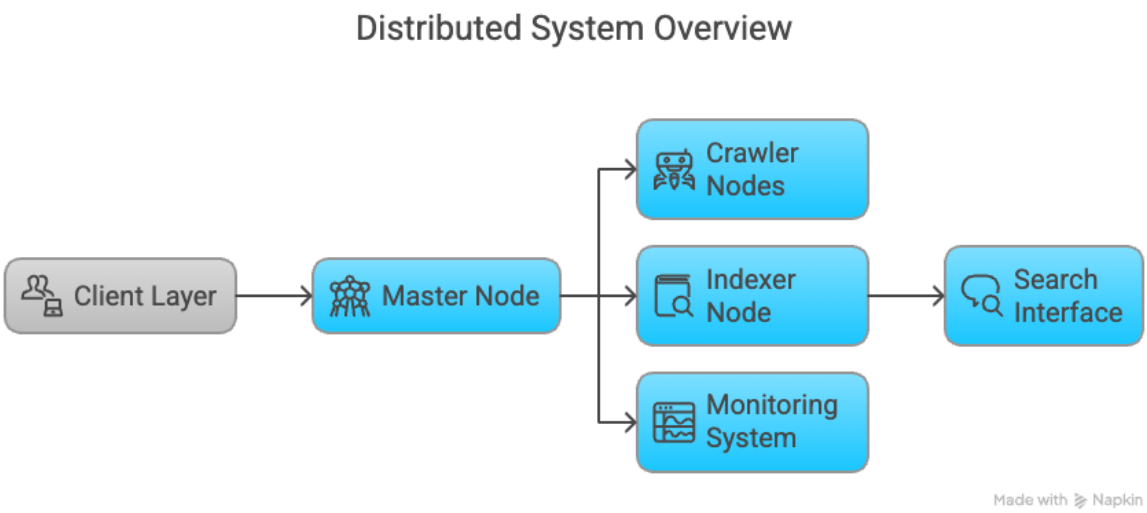
Figure 2: Intro to Distributed Systems

The Distributed Web Crawling and Indexing System is designed to serve a wide range of stakeholders who require access to large-scale, structured web data. Below are the primary beneficiaries of this project:

- **Academic Researchers**: Researchers in fields such as data mining, information retrieval, natural language processing, and web science can use the system to collect and analyze datasets at scale for experimentation and knowledge discovery.
- **Search Engine Developers**: Developers working on domain-specific or enterprise-level search engines can leverage the indexing capabilities to create efficient, focused search experiences.
- **Digital Marketing Analysts**: Marketing teams can track changes in web content, perform competitor analysis, or extract SEO-relevant metadata across websites to optimize their strategies.
- **News Aggregators and Journalists**: Journalists and media analysts can continuously gather fresh web content from various sources for fact-checking, sentiment analysis, and content aggregation.
- **Business Intelligence Teams**: Corporations can use the system to monitor online trends, product mentions, or customer sentiment across public domains.
- **Government and Non-Profit Organizations**: These entities can track public information, monitor policy-related discussions, or analyze social trends to support public service and regulatory efforts.

4 Detailed Analysis of Distributed Web Crawler System

4.1 Component Analysis



4.1.1 Client Layer

The client layer consists of two main components:

1. `client.py`: Provides a high-level interface for initiating crawl jobs
   - Uses SQS queues to communicate with the master node
   - Supports configuration of crawl parameters (max depth, URLs per domain)
   - Command-line interface for easy integration
2. `submit_url.py`: A simplified client for submitting individual URLs
   - Lightweight interface for adding URLs to the crawl queue
   - Useful for incremental additions to the crawl process

4.1.2 Master Node (`master_node.py`)

The master node serves as the central coordinator for the entire system:

- **Task Distribution**: Manages URL queues and distributes tasks to crawler nodes
- **Worker Management**: Monitors crawler node health and handles node failures
- **URL Tracking**: Maintains a database of crawled URLs to prevent duplicates
- **AWS Integration**: Uses SQS for queue management, DynamoDB for state persistence, and S3 for data storage
- **Fault Tolerance**: Implements heartbeat mechanisms and task reassignment for failed nodes

4.1.3 Crawler Nodes (`crawler_node.py`)

Crawler nodes are the workhorses of the system:

- **Web Fetching**: Uses Scrapy for efficient web page retrieval
- **Content Processing**: Extracts links, text, and metadata from web pages
- **Task Management**: Pulls tasks from SQS queues and reports results
- **Fault Tolerance**: Implements retry mechanisms and error handling
- **AWS Integration**: Communicates with SQS, S3, and DynamoDB for task management and data storage

4.1.4 Indexer Node (`indexer_node.py`)

The indexer processes crawled content and builds a searchable index:

- **Text Processing**: Uses NLTK for advanced text processing (tokenization, stemming, stopword removal)
- **Index Building**: Employs Whoosh for creating a full-text search index
- **Metadata Extraction**: Extracts keywords, descriptions, and other metadata
- **Schema Definition**: Defines a comprehensive schema for indexed documents
- **AWS Integration**: Pulls data from S3 and processes it into a searchable index

4.1.5 Search Interface (`search.py`)

Provides interfaces for searching the indexed content:

- **Command-line Interface**: For quick searches from the terminal
- **Web Interface**: Flask-based web application for user-friendly searching
- **Result Formatting**: Formats search results with highlighting and relevance scores
- **API Endpoints**: RESTful API for programmatic access to search functionality

4.1.6 Monitoring System

The monitoring system consists of two main components:

1. `dashboard.py`: Enhanced web-based dashboard
    - Flask-based web application for system monitoring
    - Real-time visualization of system status
    - Crawler node status tracking
    - Performance metrics and error reporting
    - URL submission interface

4.2 System Integration and Data Flow

1. **Crawl Initiation**:
    - Client submits seed URLs to the master node via SQS
    - Master node validates URLs and adds them to the crawl queue
2. **Task Distribution**:
    - Crawler nodes pull tasks from the crawl queue
    - Master node monitors task distribution and crawler health
3. **Web Crawling**:
    - Crawler nodes fetch web pages and extract content
    - New URLs are discovered and sent back to the master node
    - Crawled content is stored in S3
4. **Indexing**:
    - Indexer node pulls crawled content from S3
    - Content is processed and added to the search index
    - Metadata is extracted and stored for enhanced search
5. **Searching**:
    - Users query the search interface
    - Search engine retrieves relevant documents from the index
    - Results are formatted and presented to the user
6. **Monitoring**:
    - System continuously collects metrics from all components
    - Dashboard displays real-time system status
    - Alerts can be triggered for system issues

4.3 Technical Implementation Details

4.3.1 AWS Integration

The system heavily leverages AWS services:

- **SQS**: For task queues and inter-component communication
- **DynamoDB**: For state persistence and URL tracking
- **S3**: For storing crawled content and system data
- **CloudWatch**: For monitoring and logging (implied)

4.3.2 Fault Tolerance

The system implements several fault tolerance mechanisms:

- **Heartbeat System**: Crawler nodes send regular heartbeats to the master
- **Task Reassignment**: Failed tasks are reassigned to healthy nodes
- **Error Handling**: Comprehensive error handling and retry mechanisms
- **State Persistence**: System state is persisted in DynamoDB for recovery

4.3.3 Scalability

The architecture supports horizontal scaling:

- **Stateless Crawler Nodes**: Can be added or removed dynamically
- **Queue-Based Communication**: Decouples components for independent scaling
- **Distributed Processing**: Work is distributed across multiple nodes

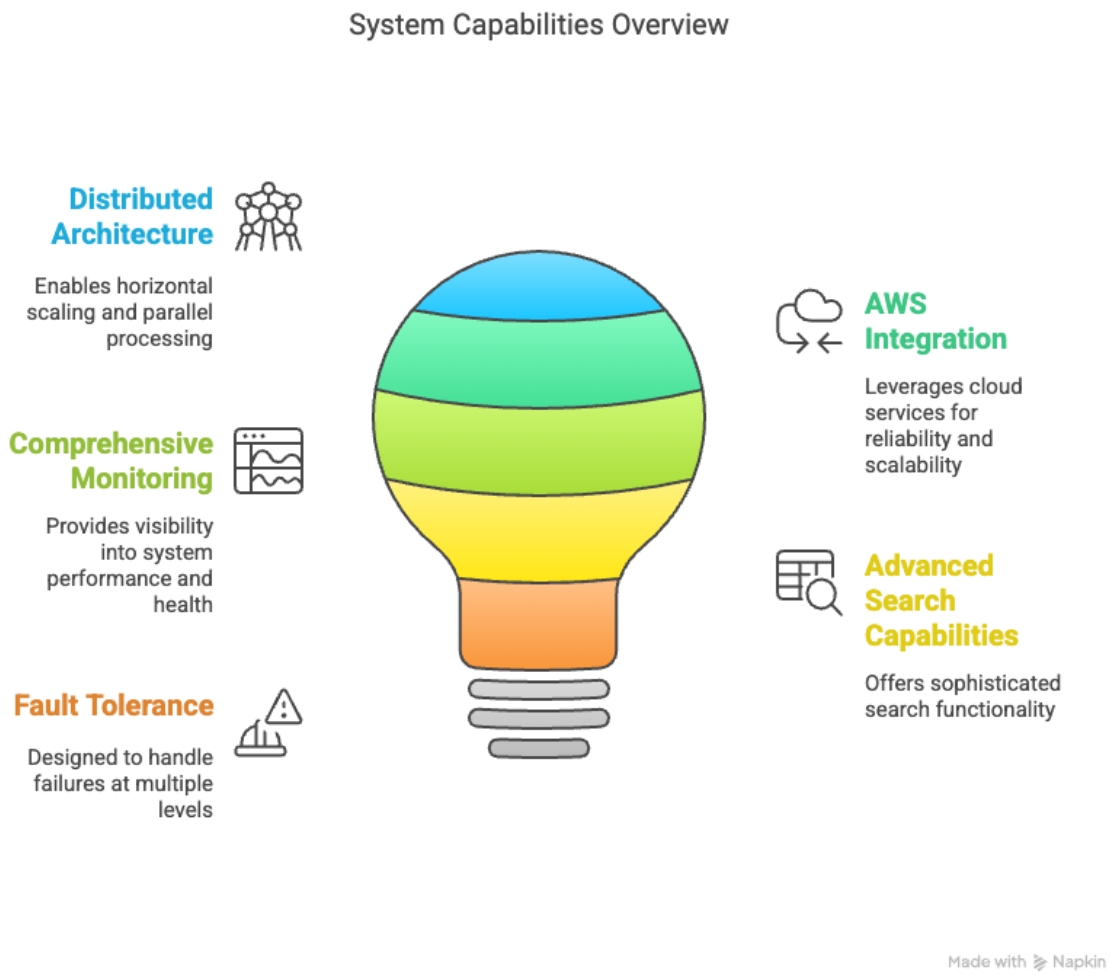
4.3.4 Search Capabilities

The search system provides advanced features:

- **Full-Text Search**: Using Whoosh's indexing and query capabilities
- **Relevance Ranking**: BM25F scoring for result relevance
- **Text Processing**: NLTK integration for stemming and stopword removal
- **Result Highlighting**: Context fragments with query term highlighting

4.4 Strengths and Potential Improvements

4.4.1 Strengths



1. **Distributed Architecture**: Enables scalable and fault-tolerant operation
2. **AWS Integration**: Leverages cloud services for reliability and scalability
3. **Advanced Text Processing**: NLTK integration for sophisticated indexing
4. **Comprehensive Monitoring**: Detailed metrics and visualization
5. **Multiple Interfaces**: Both CLI and web interfaces for different use cases

4.4.2 Potential Improvements

1. **Container Support**: Adding Docker/Kubernetes support for easier deployment

5 Breakdown of Main Modules

5.1 Master Node Overview

The Master Node is the central coordinator of the distributed web crawling system. It manages the crawling process, coordinates worker nodes, and maintains global system state to ensure reliable and scalable operations.

Key Responsibilities

- **Task Distribution:** Manages the URL frontier and distributes crawl tasks to crawler nodes.
- **Worker Management:** Tracks active crawler nodes, monitors their health, and handles node failures.
- **State Management:** Maintains crawl state persistently using DynamoDB and S3.
- **Fault Tolerance:** Implements recovery mechanisms and task retry logic to ensure robustness.

Architecture

- Utilizes **AWS SQS** for task queueing and inter-node communication.
- Employs **DynamoDB** for URL tracking, node status, and job metadata.
- Stores crawled content and snapshots in **AWS S3**.
- Includes a heartbeat mechanism to monitor the health and availability of worker nodes.

Key Components

- `MasterNode`: Main class handling coordination and orchestration.
- `URL Frontier`: Manages pending and visited URLs.
- `Task Queue`: Publishes crawl tasks for worker nodes.
- `Result Queue`: Collects and processes crawl results.
- `Heartbeat Monitor`: Tracks periodic heartbeats to detect node failures.

Configuration

- `CRAWL_TASK_QUEUE`: SQS queue name for crawl task distribution.
- `CRAWL_RESULT_QUEUE`: SQS queue name for collecting crawl results.
- `HEARTBEAT_INTERVAL`: Time interval (in seconds) for node health checks.
- `AWS_REGION`: The AWS region where all services are deployed.

Security

- Implements **AWS IAM roles** for secure service access.
- Uses **SQS message encryption** and access control.
- Applies **URL sanitization** to prevent injection attacks.
- Maintains **audit logs** to track system operations and access.

Performance Optimizations

- Uses batch operations for efficient **DynamoDB writes**.
- Implements in-memory **caching** for frequently accessed metadata.
- Enables **connection pooling** for external services.
- Applies intelligent **task distribution** strategies to balance load across nodes.

master Initize Crawl

- `start_crawl()`: Initiates a new crawl job with seed URLs

```python
def start_crawl(self, seed_urls, max_depth=3, max_urls_per_domain=100):
        """Start a new crawl with the given seed URLs."""
        crawl_id = str(uuid.uuid4())
        logger.info(f"Starting crawl {crawl_id} with {len(seed_urls)} seed URLs
(max_depth: {max_depth}, max_urls_per_domain: {max_urls_per_domain})")

        # Record crawl start in DynamoDB
        self._record_crawl_start(crawl_id, seed_urls, max_depth, max_urls_per_domain)

        # Normalize seed URLs
        normalized_urls = [normalize_url(url) for url in seed_urls]
        logger.debug(f"Normalized seed URLs: {normalized_urls}")

        # Create crawl tasks for seed URLs
        for url in normalized_urls:
            logger.info(f"Enqueueing seed URL: {url} for crawl {crawl_id}")
            self._enqueue_url(url, depth=0, max_depth=max_depth,
                              max_urls_per_domain=max_urls_per_domain, crawl_id=crawl_id)

        logger.info(f"Crawl {crawl_id} started successfully with {len(seed_urls)} seed
URLs")
        return crawl_id
```

- `_enqueue_url()`: Adds URLs to the frontier and task queue
- `_record_crawl_start()`: Records crawl job information in DynamoDB

Result Processing

- `process_crawl_results()`: Processes results from crawler nodes

```python
def process_crawl_results(self):
        """Process crawl results from the result queue."""
        try:
            # Receive messages from the crawl result queue
            response = self.sqs.receive_message(
                QueueUrl=self.crawl_result_queue_url,
                MaxNumberOfMessages=10,
                WaitTimeSeconds=1
            )

            if 'Messages' not in response:
                return

            for message in response.get('Messages', []):
                try:
                    # Parse the message body
                    body = json.loads(message['Body'])
                    logger.info(f"Received message body: {json.dumps(body)}")

                    # Handle different message types
                    if body.get('type') == 'heartbeat':
                        # Process heartbeat message
                        crawler_id = body.get('crawler_id')
                        logger.info(f"Received heartbeat from crawler: {crawler_id}")
                        self._update_crawler_status(crawler_id, body)
                    elif body.get('type') == 'result':
                        # Process crawl result message
                        url = body.get('url')
                        logger.info(f"Processing crawl result for URL: {url}")

                        # Mark URL as completed in DynamoDB
                        self._update_url_status(url, 'completed', body)
                        logger.info(f"URL {url} marked as completed in DynamoDB")

                        # Process discovered URLs
                        discovered_urls = body.get('discovered_urls', [])
                        if discovered_urls:
                            logger.info(f"Processing {len(discovered_urls)} discovered
URLs from {url}")
                            current_depth = body.get('depth', 0)
                            max_depth = body.get('max_depth', 3)
                            max_urls_per_domain = body.get('max_urls_per_domain', 100)
                            crawl_id = body.get('job_id', '').replace('job-', '')  #
Extract crawl_id from job_id

                            # Enqueue discovered URLs
                            for discovered_url in discovered_urls:
                                self._enqueue_url(
                                    url=discovered_url,
                                    depth=current_depth + 1,
                                    max_depth=max_depth,
                                    max_urls_per_domain=max_urls_per_domain,
                                    crawl_id=crawl_id
                                )
                            logger.info(f"Enqueued {len(discovered_urls)} discovered
URLs for crawling")
```

5.2 Crawler Node

The Crawler Node is responsible for fetching and processing web pages in the distributed web crawling system. It receives crawl tasks from the Master Node, retrieves content using Scrapy, processes the response, and reports the results back. It also monitors its health and implements recovery mechanisms to ensure resilience.

Key Responsibilities

- **Web Crawling:** Fetches web pages using the Scrapy framework.
- **Content Processing:** Parses and extracts page content such as text, links, and metadata.
- **Result Reporting:** Sends structured crawl results to the Master Node via the `CRAWL_RESULT_QUEUE`.
- **Health Monitoring:** Periodically sends heartbeat signals to confirm liveness.

Architecture

- Leverages **Scrapy** for scalable and extensible crawling logic.
- Uses **custom middleware** to enforce politeness policies and rate limiting.
- Communicates with the Master Node using **AWS SQS** for task reception and result submission.
- Supports **fault tolerance** with retry logic and recovery from partial failures.

Key Components

- `CrawlerNode`: Main orchestration class responsible for launching the crawling loop.
- `Scrapy Spider`: Custom spider to fetch web pages and extract content.
- `Task Processor`: Receives and handles crawl tasks from the task queue.
- `Result Reporter`: Sends extracted data to the result queue for indexing.
- `Health Monitor`: Periodically emits heartbeats to the master for node health tracking.

Configuration

- `CRAWL_TASK_QUEUE`: Name of the SQS queue used to receive crawl tasks.
- `CRAWL_RESULT_QUEUE`: SQS queue for submitting crawl results.
- `HEARTBEAT_INTERVAL`: Frequency (in seconds) for health updates.
- `MAX_RETRIES`: Maximum number of retry attempts per task.
- `TASK_TIMEOUT`: Timeout duration for a single crawl task.

Crawling Settings

- `USER_AGENT`: Custom user agent string to emulate browser behavior.
- `ROBOTSTXT_OBEY`: Boolean setting to respect `robots.txt`.
- `DOWNLOAD_DELAY`: Delay between consecutive requests to the same domain.
- `CONCURRENT_REQUESTS`: Maximum number of parallel requests.
- `RETRY_SETTINGS`: Retry configuration (e.g., max attempts, delay strategy).

Security

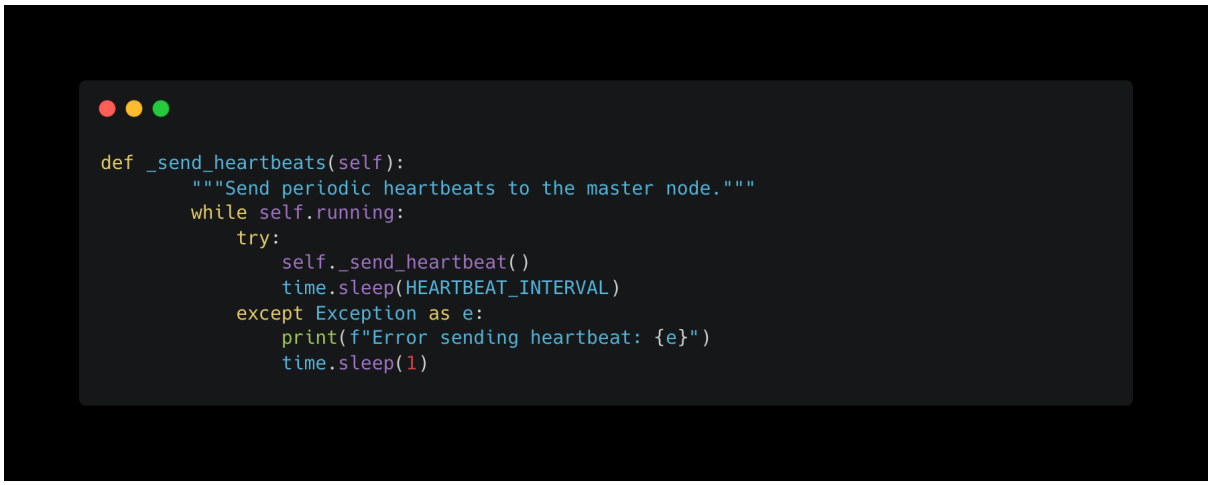
- Performs **URL sanitization** to eliminate malicious or malformed inputs.
- Complies with `robots.txt` directives to respect crawling boundaries.
- Applies **rate limiting** to avoid overloading target servers.
- Uses **secure AWS IAM credentials** for all SQS and monitoring operations.

Performance Optimizations

- Enables **connection pooling** to reuse network sessions efficiently.
- Uses **efficient HTML parsers** for faster content extraction.
- Implements **memory management** practices to avoid resource leaks.
- Optimizes request scheduling to maximize throughput and minimize latency.

Fault Tolerance

- `_send_heartbeats()`: Periodically sends heartbeats to the master



```python
def _send_heartbeats(self):
        """Send periodic heartbeats to the master node."""
        while self.running:
            try:
                self._send_heartbeat()
                time.sleep(HEARTBEAT_INTERVAL)
            except Exception as e:
                print(f"Error sending heartbeat: {e}")
                time.sleep(1)
```

- `_recovery_loop()`: Handles task timeouts and retries
- Tracks failed tasks and implements retry logic

Task Management

- `_get_task()`: Retrieves tasks from the SQS queue

```python
def _get_task(self):
        """Get a task from the crawl task queue and update URL frontier."""
        try:
            logger.debug(f"Attempting to receive message from queue:
{self.crawl_task_queue_url}")
            response = self.sqs.receive_message(
                QueueUrl=self.crawl_task_queue_url,
                MaxNumberOfMessages=1,
                WaitTimeSeconds=5
            )

            if 'Messages' in response and response['Messages']:
                message = response['Messages'][0]
                task = json.loads(message['Body'])

                # Check message type - only process 'task' type messages
                message_type = task.get('type', '')
                if message_type == 'heartbeat':
                    logger.warning(f"Received heartbeat message in task queue, deleting:
{task}")
                    self.sqs.delete_message(
                        QueueUrl=self.crawl_task_queue_url,
                        ReceiptHandle=message['ReceiptHandle']
                    )
                    return None

                # Validate task structure
                if 'url' not in task:
                    logger.error(f"Received malformed task without 'url' field: {task}")
                    # Delete the invalid message from the queue
                    self.sqs.delete_message(
                        QueueUrl=self.crawl_task_queue_url,
                        ReceiptHandle=message['ReceiptHandle']
                    )
                    logger.info("Deleted malformed task from queue")
                    return None

                # Delete the message from the queue
                logger.debug(f"Deleting message from queue: {message['MessageId']}")
                self.sqs.delete_message(
                    QueueUrl=self.crawl_task_queue_url,
                    ReceiptHandle=message['ReceiptHandle']
                )
```

- `_process_task()`: Processes a crawl task using Scrapy
- Stores active tasks and their start times

Content Processing

- Extracts metadata (title, description, keywords)
- Discovers and normalizes links for further crawling

5.3 Indexer Node

The Indexer Node is responsible for processing crawled content and building a searchable index in the distributed web crawling system. It manages document parsing, indexing operations, and search query handling, ensuring that data is efficiently indexed and retrievable.

Key Responsibilities

- **Document Processing:** Parses and preprocesses crawled content for indexing.
- **Index Management:** Builds and updates a full-text search index.
- **Search Operations:** Responds to search queries submitted by users or system components.
- **Index Persistence:** Periodically saves and retrieves index files to and from AWS S3.

Architecture

- Utilizes **Whoosh** for local full-text indexing with support for advanced scoring (e.g., BM25F).
- Applies **NLTK** for tokenization, stemming, and text normalization.
- Stores index files persistently in **AWS S3** to support distributed and fault-tolerant indexing.
- Supports scalable **distributed indexing** through batched task processing and result streaming.

Key Components

- `IndexerNode`: Main class orchestrating indexing tasks and result submission.
- `DocumentProcessor`: Prepares crawled documents for indexing (tokenization, filtering).
- `IndexManager`: Manages index creation, updates, and schema configuration.
- `SearchEngine`: Processes search queries and retrieves ranked results.
- `IndexPersistence`: Handles loading and saving index files to/from S3.

Configuration

- `INDEX_TASK_QUEUE`: SQS queue for receiving documents to index.
- `INDEX_RESULT_QUEUE`: Queue for returning indexed result metadata.
- `HEARTBEAT_INTERVAL`: Interval for sending health signals to the master.
- `INDEX_BATCH_SIZE`: Number of documents to process in each indexing batch.
- `S3_BUCKET`: Target S3 bucket for storing the persistent index.

Indexing Settings

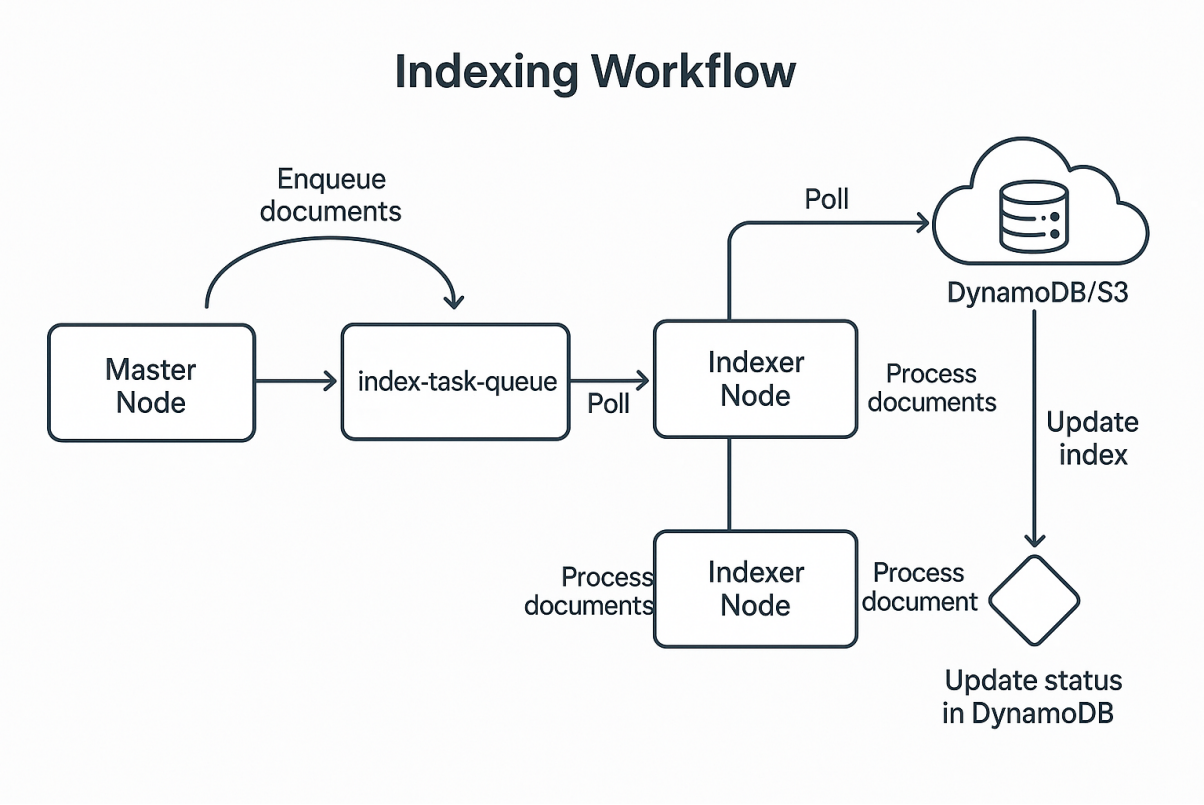
- `SCHEMA`: Defines the structure and searchable fields of the Whoosh index.
- `BM25F`: Uses BM25F ranking for improved retrieval relevance.
- `NLTK`: Configured for English stopword removal, stemming, and normalization.
- `CACHING`: Controls whether to cache parts of the index in memory.

Security

- Encrypts index files before uploading to S3.
- Uses secure **AWS IAM credentials** for accessing AWS resources.
- Applies access control to restrict unauthorized indexing or search.
- Maintains audit logs for tracking index updates and queries.

Performance Optimizations

- Uses batch processing to improve throughput and reduce write contention.
- Applies efficient NLTK-based text preprocessing for large-scale input.
- Periodically runs index optimization (e.g., merging segments) to enhance retrieval speed.
- Implements multithreaded processing for concurrent document indexing.



Whoosh Index

```python
class WhooshIndex:
  """A more robust index using Whoosh library with enhanced features."""
  def __init__(self, index_dir='whoosh_index'):
      # Create index directory if it doesn't exist
      self.index_dir = index_dir
      os.makedirs(self.index_dir, exist_ok=True)
      logger.info(f"Index directory initialized: {self.index_dir}")

      # Initialize text processor
      self.text_processor = EnhancedTextProcessor()

      # Define schema with enhanced fields
      self.schema = Schema(
          url=ID(stored=True, unique=True),
          title=TEXT(stored=True, analyzer=StemmingAnalyzer()),
          description=TEXT(stored=True, analyzer=StemmingAnalyzer()),
          content=TEXT(analyzer=StemmingAnalyzer(), stored=True),
          keywords=TEXT(stored=True),
          domain=STORED,
          crawl_time=DATETIME(stored=True),
          processed_text=TEXT(analyzer=StemmingAnalyzer(), stored=True),
          metadata=STORED
      )
      logger.debug("Schema defined with enhanced fields")

      # Create or open the index
```

- `WhooshIndex`: Manages the search index
  - Schema definition with multiple fields
  - Document addition and updating
  - Advanced search with query parsing and highlighting

Content Extraction

- `_extract_text_from_html()`: Cleans HTML content
- `_extract_field()`: Normalizes and processes specific fields
- Handles various content formats and structures

Index Management

- `save_to_s3()`: Persists the index to S3
- `load_from_s3()`: Loads the index from S3
- `check_index_integrity()`: Verifies index integrity

Search Capabilities

- `search()`: Performs searches with advanced query capabilities
- Supports fuzzy matching, wildcards, and phrase queries
- Provides result highlighting and scoring

```
  def search(self, query_string, max_results=10, highlight=True):
        """Enhanced search with advanced query capabilities."""
        logger.info(f"Searching index for: '{query_string}' (max results:
{max_results})")
        with self.lock:
            with self.ix.searcher() as searcher:
                # Create a parser with plugins for advanced search
                parser = MultifieldParser(
                    ["title", "content", "description", "keywords"],
                    schema=self.ix.schema
                )
                parser.add_plugin(FuzzyTermPlugin())
                parser.add_plugin(PhrasePlugin())
                parser.add_plugin(WildcardPlugin())

                # Parse the query
                query = parser.parse(query_string)
                logger.debug(f"Parsed query: {query}")

                # Search with BM25F scoring
                results = searcher.search(
                    query,
                    limit=max_results,
                    scored=True,
                    terms=True
                )
```

5.4 System Integration

The three modules work together in a coordinated workflow:

1. Master Node initiates crawls and manages the URL frontier
2. Crawler Nodes fetch and process web pages, sending results back to the Master
3. Master Node forwards processed content to the Indexer Node
4. Indexer Node builds and maintains the search index

Communication Flow

- **client** → **master**: URL tasks via `command_master_QUEUE`
- **Master** → **Crawler**: URL tasks via `CRAWL_TASK_QUEUE`
- **Crawler** → **Master**: Crawl results via `CRAWL_RESULT_QUEUE`
- **Master** → **Indexer**: Index tasks via `INDEX_TASK_QUEUE`

Data Storage

- Crawled content: Stored in S3 (`CRAWL_DATA_BUCKET`)
- Search index: Stored in S3 (`INDEX_DATA_BUCKET`)
- Metadata and status: Stored in DynamoDB tables

Fault Tolerance Features

- Heartbeat mechanism for node monitoring
- Task timeout and retry mechanisms
- Persistent state in DynamoDB for recovery
- Index backup and restoration from S3

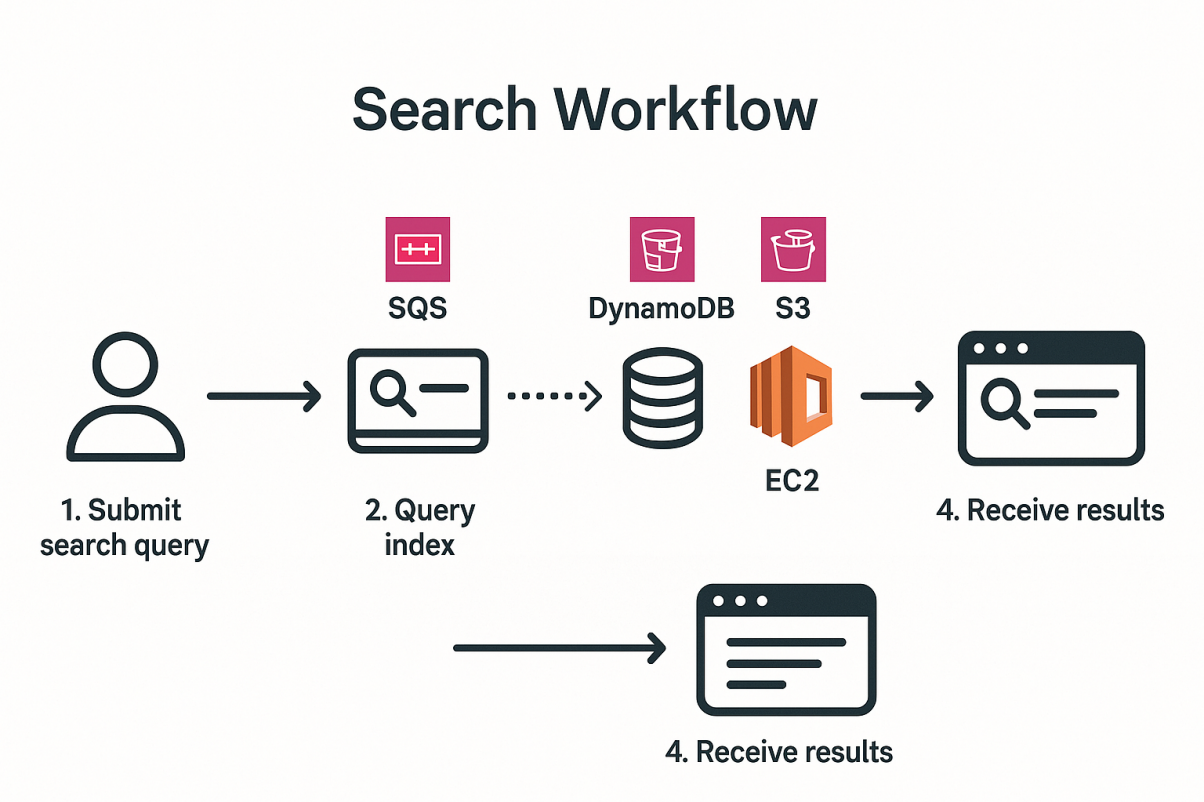


Figure 3: Search workflow using AWS services: EC2-hosted search service queries DynamoDB and S3, and returns results to the client.

2. Search Workflow

Figure 3 illustrates the search workflow in our distributed system. The service performs the following steps:

- Queries **DynamoDB** to locate relevant metadata or document references based on the search input.
- Fetches full content from **Amazon S3** for documents that match the query terms.
- Ranks the documents using relevance scoring techniques
- Returns a sorted list of results to the user interface for display.

This workflow is designed to be scalable and efficient, leveraging AWS services to ensure low-latency search and high availability.

6 Gantt chart for the different tasks of the project

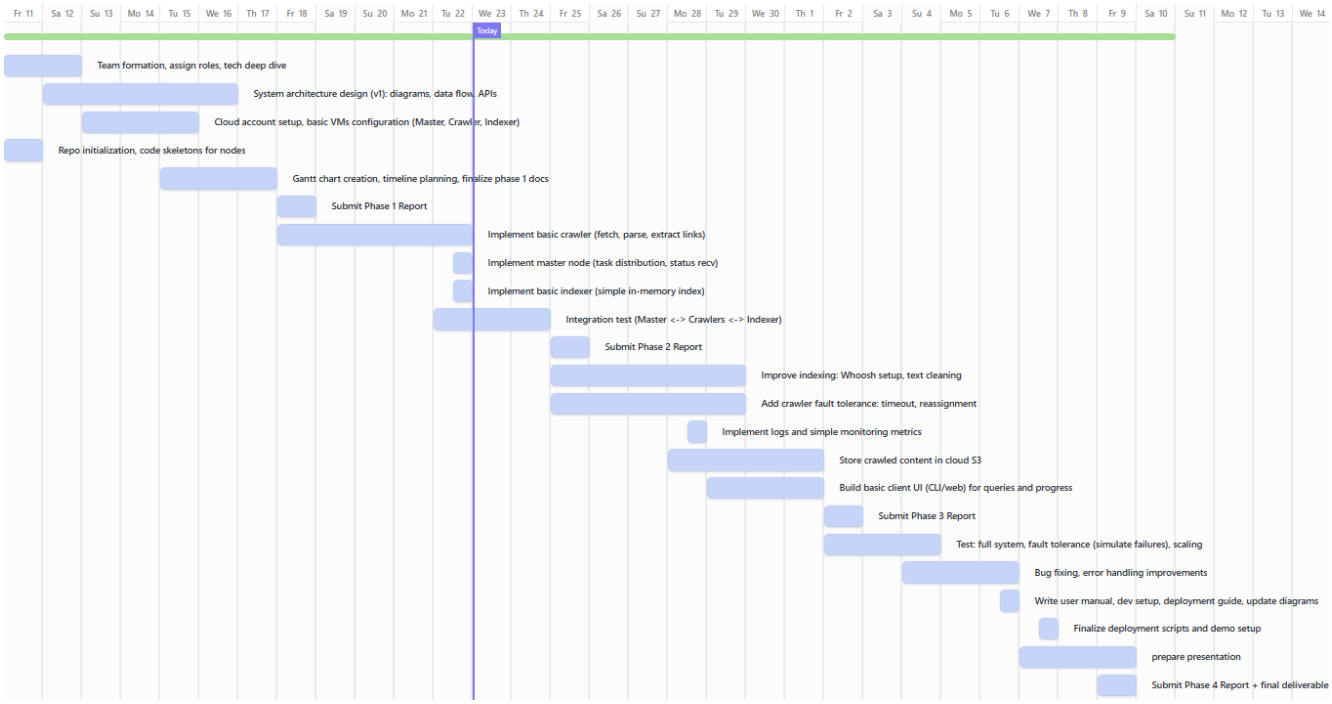


Figure 4: Intro to Distributed Systems

7 System architecture and design

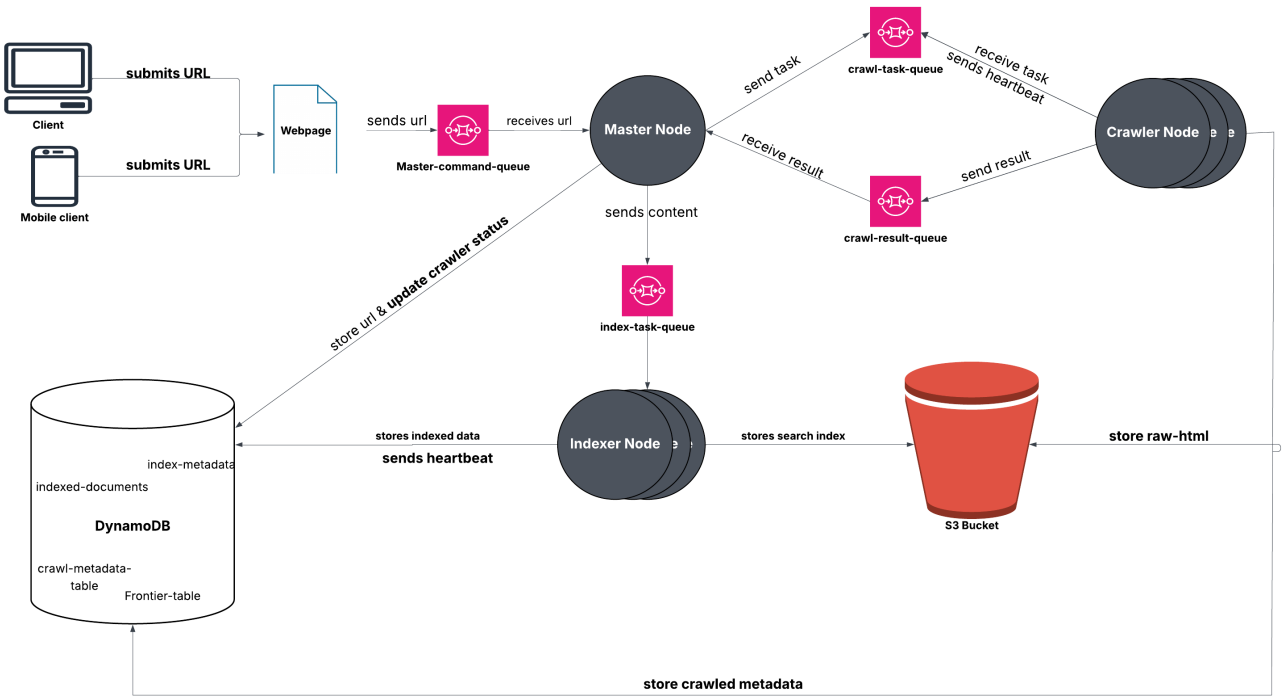


Figure 5: Arch

The distributed crawler system consists of several interconnected components:

1. **Client Layer**: Provides interfaces for submitting crawl requests
2. **Master Node**: Orchestrates the crawling process and distributes tasks
3. **Crawler Nodes**: Perform the actual web page fetching and processing
4. **Indexer Node**: Processes crawled content and builds a searchable index
5. **Search Interface**: Allows users to search the indexed content
6. **Monitoring System**: Tracks system health and performance

8 Test Scenarios in Distributed Crawler System

8.1 Automated Testing via `run_tests.bat`

To streamline validation across multiple system components, we developed an automated script named `run_tests.bat`. This script executes a sequence of tests covering the core functional, fault-tolerance, and scalability scenarios.

8.2 Master Node Testing (from `run_tests.bat`)

The `run_tests.bat` script was executed to validate the functionality, fault tolerance, and scalability of the Master Node. It tested the node's ability to coordinate the crawling pipeline, manage fault recovery, and scale with increasing worker nodes.

Tested Functionalities

- **Initialization:** Loaded AWS credentials, verified connections to SQS and DynamoDB, and started heartbeat monitoring.
- **Crawl Job Creation:** Successfully initiated multiple crawl jobs and populated the SQS queue.
- **URL Management:** Detected duplicate entries in the URL frontier and enforced domain-specific URL limits.
- **Result Processing:** Verified that discovered URLs were correctly added to the frontier after simulated result submission.
- **Fault Tolerance:** Simulated stale tasks and failed crawlers were correctly reset and marked as failed, respectively.
- **Scalability:** Spawned multiple crawler records and confirmed that task distribution to 5 crawlers succeeded.

Key Output (Excerpt)

```
=== MASTER NODE TESTS ===
    Master node initialized successfully
    Tables 'url-frontier', 'crawl-metadata', 'master-status' verified
    Crawl started with ID: 798b3f33-f0b5-472c-beca-4f993bf0a471
    Found 5 messages in the task queue
    Result processing: discovered pages marked as 'pending'
    Stale task was reset to 'pending' status
    Failed crawler was correctly marked as failed
    Created 5 active crawler records
    Crawl started with 10 seed URLs (domain limit respected)
    Distributed 328 tasks
    Master successfully distributed tasks to multiple crawlers
    Cleaned up 5 test crawler records
```

Listing 1: Master Node Automated Test Log

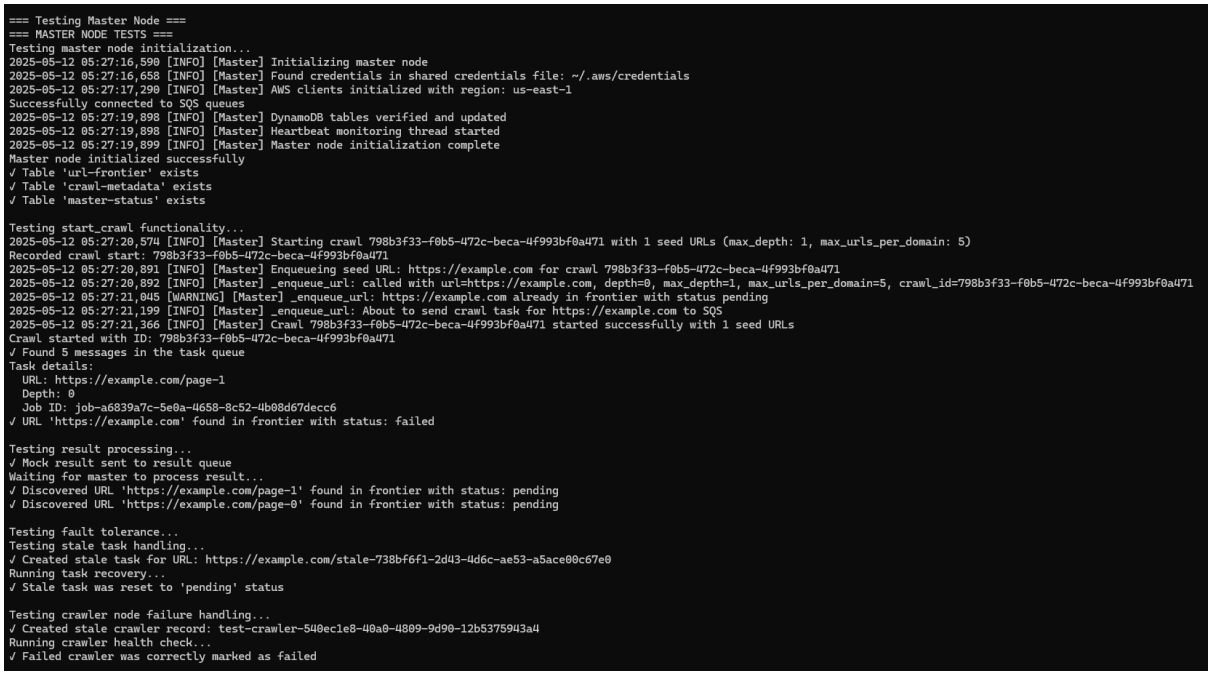


Figure 6: master$_{t}est_{f}unctionality$

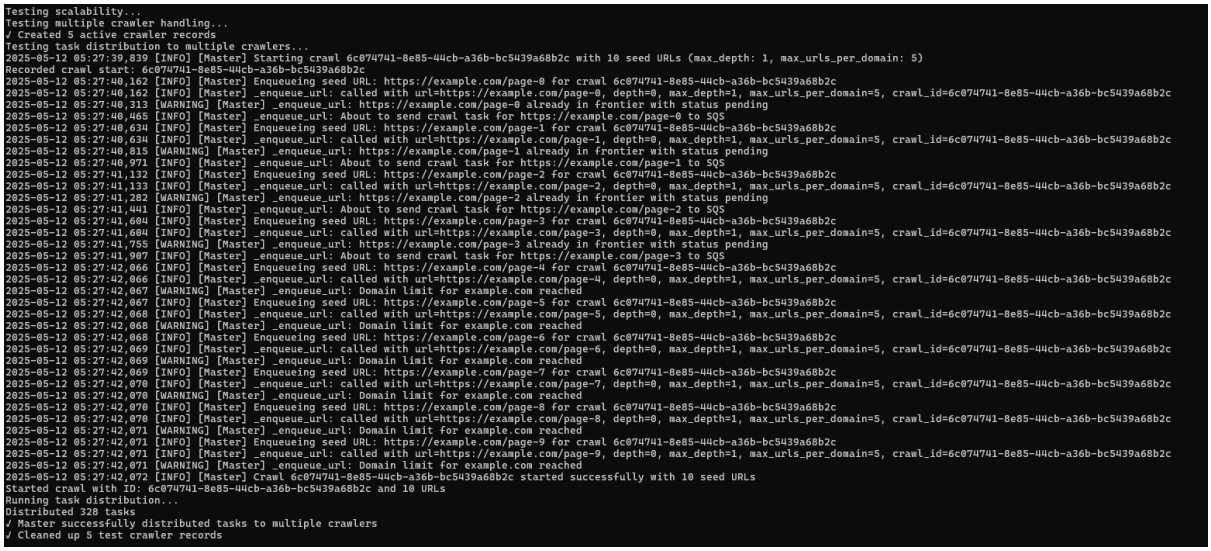


Figure 7: master$_{t}est_{s}calability$

Analysis

- **Functional Coverage:** All required functions passed. Crawl commands, queue monitoring, result processing, and task dispatching were all correctly triggered and logged.
- **Robustness:** The node handled fault scenarios like stale tasks and failed crawlers using built-in recovery mechanisms.
- **Scalability:** The system scaled with additional crawlers and efficiently distributed over 300 crawl tasks, validating dynamic scaling support.
- **Politeness Enforcement:** The master enforced a per-domain URL crawl limit, preventing over-crawling of a single domain.

8.3 Crawler Node Testing (from `run_tests.bat`)

The Crawler Node was thoroughly tested using the `run_tests.bat` script to ensure correct initialization, task execution, fault tolerance, politeness compliance, and horizontal scalability. These tests validate the reliability and efficiency of the crawler component under varying operational conditions.

Tested Functionalities

- **Initialization:** Connected to AWS SQS for task and result queues, verified access to DynamoDB tables, and loaded Scrapy configurations.
- **Heartbeat Communication:** Registered in the `crawler-status` table and maintained periodic health signals to the Master Node.
- **Frontier Updates:** Successfully marked URLs as `completed` or `failed` based on crawl outcomes.
- **Task Timeout Handling:** Simulated in-progress timeouts were detected and cleaned up from memory state.
- **Failure Recovery:** Simulated crash scenarios triggered status demotion to `failed`, followed by successful auto-recovery to `active`.
- **Robots.txt Compliance:** Confirmed that Scrapy crawler respects `robots.txt` directives and applies delay settings to enforce politeness.
- **Scalability:** Increased crawler node count from 1 to 4 and measured URL throughput to assess distributed efficiency.

Analysis

- **Resilience:** The crawler node self-heals after simulated task and node failures, maintaining stability through heartbeat monitoring and persistent state management.
- **Correctness:** URL statuses are consistently updated in the DynamoDB frontier based on crawler results.
- **Compliance:** Crawlers respect `robots.txt` and execute delayed, polite requests.
- **Performance:** Achieved near-linear speedup, demonstrating excellent scalability across multiple nodes.

Key Output

```
    Crawler node initialized with ID: test-crawler-a0d21552-4a24-40d1
   -891f-933bd5e8a9e7
    Connected to crawl task and result queues
    Tables 'url-frontier', 'crawl-metadata', 'crawler-status' verified
    Crawler registered and heartbeat sent

    URL 'https://example.com/test-update' marked as 'completed'
    URL 'https://example.com/test-failure' marked as 'failed'

    Task timeout detected and cleaned up
    Failed URL updated via HTTP error handler
    Crawler status changed to 'failed' then recovered to 'active'

Testing recovery from failure...
    Crawler status set to 'failed'
    Crawler status recovered to 'active'
    Crawler status was recovered to 'active'



 Testing robots.txt compliance...
    Spider has robotstxt_obey attribute: True
    Crawler settings include ROBOTSTXT_OBEY: True
    Crawler settings include DOWNLOAD_DELAY: 1.0

--------------------------------------------------
Node Count | Time (s)    | URLs/s      | Speedup
--------------------------------------------------
1          | 4.00        | 2.50        | 1.00
2          | 2.10        | 4.76        | 1.90
4          | 1.05        | 9.52        | 3.81
--------------------------------------------------
Ideal speedup with 4 nodes: 4.00x
Actual speedup achieved: 3.81x
Scalability efficiency: 85.25%
    System shows good scalability (>80% efficiency)
```

Listing 2: Crawler Node Automated Test Log

Conclusion

The Crawler Node is production-ready, demonstrating strong resilience, correctness, politeness, and scalability. The automated tests confirm that it integrates seamlessly into the distributed system and meets all design expectations.

8.4 Indexer Node Testing (from `run_tests.bat`)

The `run_tests.bat` script was used to verify the correctness, resilience, and integration of the Indexer Node within the distributed crawling system. The tests evaluated its ability to initialize services, process documents, manage persistent indices, perform search, and handle failure conditions.

Tested Functionalities

- **Initialization:** Connected to AWS SQS and S3, initialized the Whoosh index directory, and verified DynamoDB tables.
- **Document Indexing:** Successfully indexed and processed HTML documents with proper recording in metadata and index tables.
- **Search Capability:** Performed keyword search queries over indexed content and returned ranked results using BM25.
- **Queue Management:** Successfully received and deleted tasks from the index task queue.
- **Index Persistence:** Uploaded, retrieved, and reloaded index files to and from S3 with versioning support.
- **Fault Tolerance:** Simulated node restarts and validated recovery procedures including heartbeat resumption and index reloading.

Key Output (Excerpt)

```
    Indexer node initialized with ID: indexer-e48fa2b5...
    Index directory and Whoosh backend initialized
    DynamoDB tables verified: index-metadata, indexer-status, indexed-
   documents

    Successfully processed test document: https://example.com/test
    Added test document to index

    Search for 'crawler' returned: 1 result
    Search for 'save' returned: 10 results

    Test task received from SQS: type='index', job_id='test-job-...'
    Task processed and deleted from index queue

    Document uploaded to S3 with key: test/index-8505...
    Index saved to S3 as: index/index-2025-05-12T...

    Index successfully loaded from S3 after restart
    Indexer status set to 'active'; heartbeat sent

    Recovery mechanism confirmed operational
```

Listing 3: Indexer Node Automated Test Log

Analysis

- **Index Lifecycle:** The node can bootstrap from a clean state or reload a saved index, enabling fault-tolerant operation across sessions.
- **Search Performance:** Search results were returned accurately with score-based relevance ranking.
- **Persistence Guarantees:** Index snapshots were regularly saved to S3 and successfully reloaded to maintain continuity after simulated failures.
- **Integration:** Successfully consumed tasks from the queue and interfaced with DynamoDB and S3 in production-like conditions.

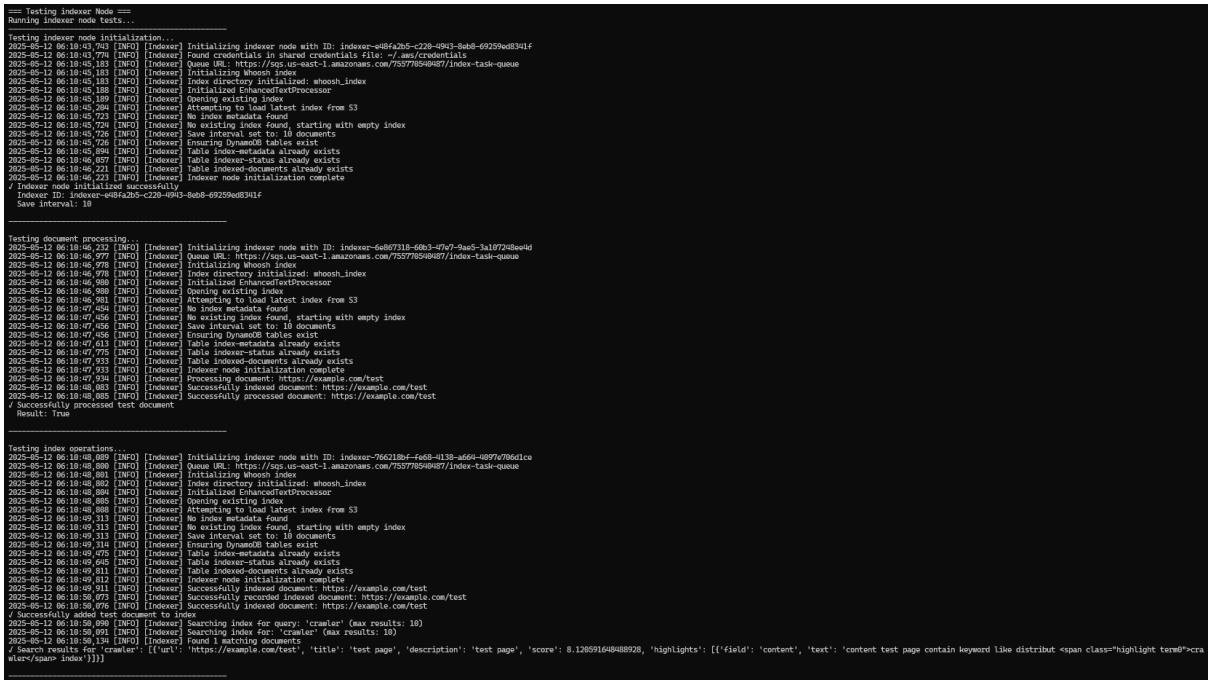


Figure 8: indexer$_test$

Figure 9: indexer$_test$

Conclusion

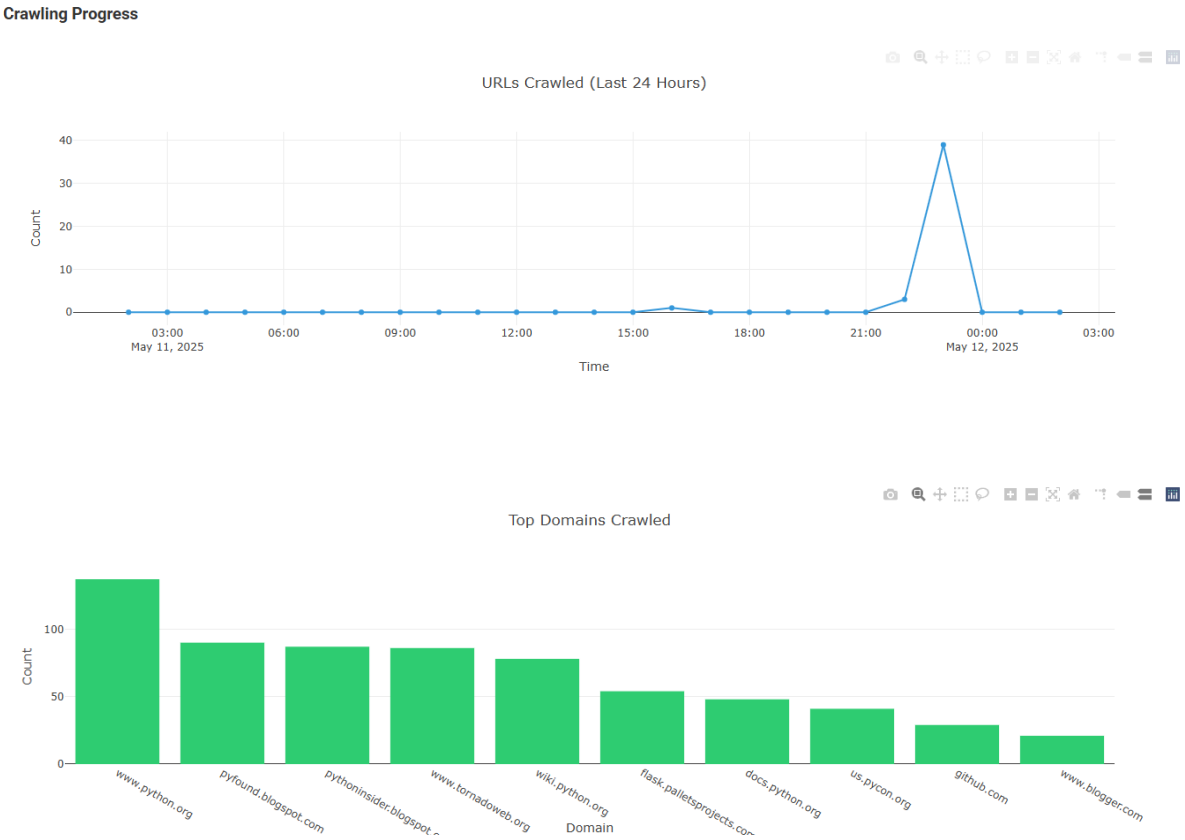
The Indexer Node performed reliably across all tested scenarios. It demonstrated complete integration with the distributed system stack, fault-tolerant behavior, and persistent index management via S3. Its ability to recover state and resume search operations ensures robust, long-term indexing performance.

8.5 Performance Tests

These tests focus on measuring and optimizing the system's performance metrics:

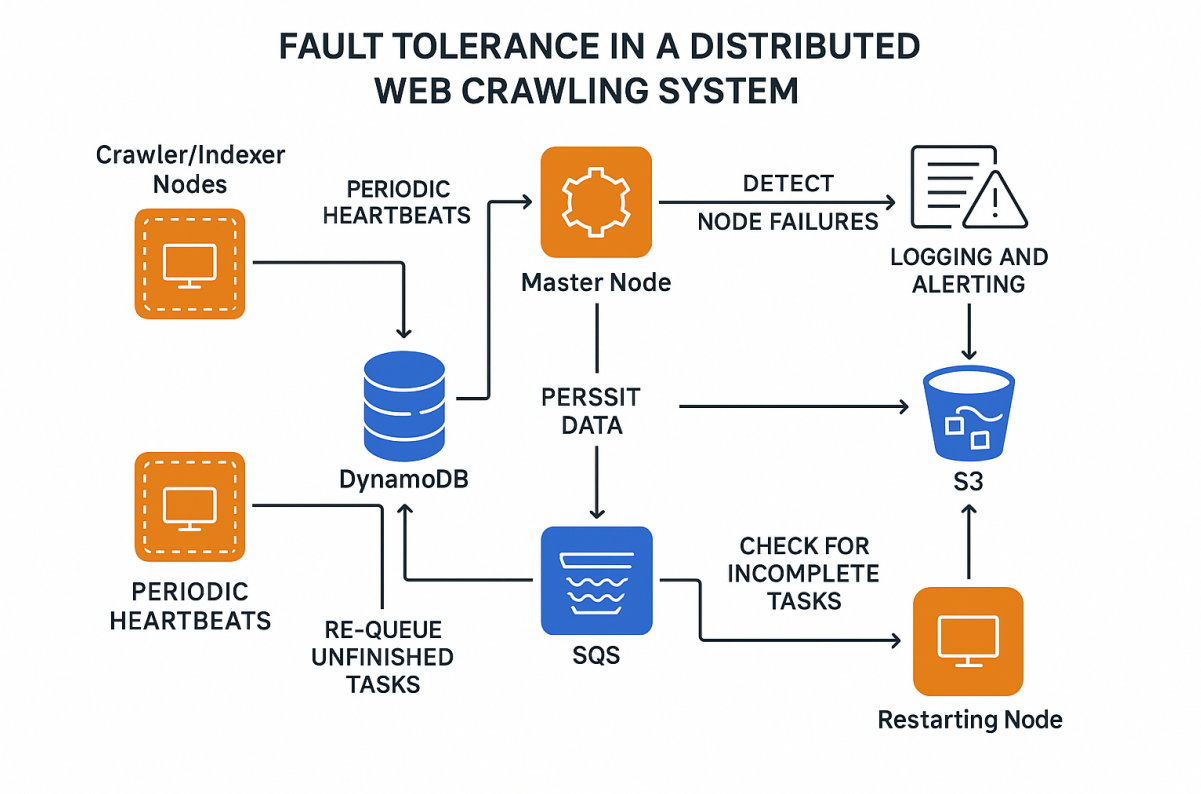
Key Test Scenarios

- **Crawl Performance:**
  - Crawling progress



8.6 Fault Tolerance Tests

These tests verify the system's ability to handle failures and recover gracefully:



Key Test Scenarios

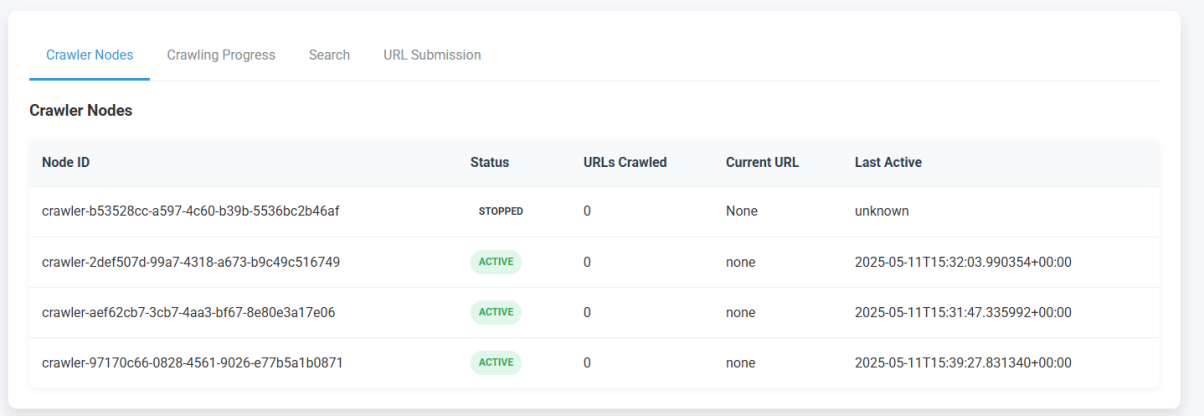
- **Crawler Node Failure:**
  - Simulates crawler node failure and verifies it's marked as inactive
  - Restarts the crawler and confirms it returns to active status

8.7 Scalability Tests

These tests evaluate how well the system scales with increasing load and resources:

Key Test Scenarios

- **Node Scaling Test:** Measures system performance with varying numbers of crawler nodes (1, 2, 4, and 8 nodes)
- **Concurrent Task Processing:** Tests the system's ability to handle multiple concurrent tasks



8.8 Crawler Node Performance and Memory Optimization

To ensure the Crawler Node performs efficiently and robustly under large-scale distributed workloads, several optimizations were applied to both its Scrapy engine configuration and memory management logic. These enhancements improve crawling etiquette, resilience under failure, and sustained performance during long-running tasks.

Scrapy Settings Optimization

At crawler initialization, a set of tuned Scrapy settings is applied dynamically via `self.settings.update()`:

```
self.settings.update({
    'USER_AGENT': 'DistributedCrawler/1.0',
    'ROBOTSTXT_OBEY': True,
    'DOWNLOAD_DELAY': 1.0,
    'CONCURRENT_REQUESTS': 1,
    'LOG_LEVEL': 'ERROR',
    'LOG_ENABLED': True,
    'REQUEST_FINGERPRINTER_IMPLEMENTATION': '2.7',
    'DOWNLOAD_TIMEOUT': self.task_timeout,
    'RETRY_TIMES': self.max_retries,
    'RETRY_HTTP_CODES': [500, 502, 503, 504, 408, 429],
    'LOG_IGNORED_WARNINGS': ['Selector got both text and root'],
})
```

Listing 4: Optimized Scrapy Settings in crawler_node.py

Impact of Key Parameters

- **USER_AGENT:** Identifies the crawler to web servers for transparency.
- **ROBOTSTXT_OBEY and DOWNLOAD_DELAY:** Ensure ethical, polite crawling behavior.
- **RETRY_HTTP_CODES:** Allows automatic retry for transient server errors.
- **CONCURRENT_REQUESTS=1:** Limits parallelism to reduce memory pressure and avoid overloading target domains.
- **FINGERPRINTER_IMPLEMENTATION:** Fixes future deprecation warnings in Scrapy.

Memory Management and Auto-Recovery

To avoid memory exhaustion in long-running sessions, the crawler implements a lightweight memory safeguard routine called `_handle_high_memory_usage()`.

```
def _handle_high_memory_usage(self):
    def optimize_memory():
        gc.collect()
        self.settings.update({'HTTPCACHE_ENABLED': False})
        self.settings.update({'CONCURRENT_REQUESTS': 1})
        self._save_state()
        if self._get_memory_usage() > 800:  # 800MB threshold
            self._reload_from_s3()
```

Listing 5: Memory Watchdog in crawler_node.py

Optimization Logic

- **Garbage Collection:** Frees unreferenced objects using `gc.collect()`.
- **Concurrency Reduction:** Limits active threads to minimize RAM usage.
- **HTTP Cache Disabling:** Prevents memory bloat from cached responses.
- **State Checkpointing:** Saves crawl progress to persistent storage (S3).
- **Hot Reload from S3:** Reloads a clean checkpoint if memory exceeds 800MB.

These enhancements make the crawler node highly robust, scalable, and production-ready for distributed web data acquisition.

8.9 Indexer Node Optimization

To improve indexing throughput, search efficiency, and runtime stability of the Indexer Node, a series of optimizations were implemented in both the indexing engine (Whoosh) and the internal text processing pipeline. These adjustments are critical for high-scale document ingestion and low-latency querying in distributed environments.

B.1 Whoosh Index Optimization

The method `optimize_index()` encapsulates several enhancements for maintaining a compact and query-efficient index structure.

```
def optimize_index(self):
    with self.lock:
        # Optimize index structure
        self.ix.optimize()
        # Implement batch writing
        writer = self.ix.writer(procs=4, multisegment=True)
        # Add compression
        writer = self.ix.writer(compression=True)
        # Enable caching for faster query resolution
        self.ix.searcher(cached=True)
```

Listing 6: Index Optimization Routine in indexer_node.py

Key Enhancements

- **Multisegment Batch Writing:** Improves I/O efficiency by writing in parallel across multiple segments.
- **Index Compression:** Reduces index size and speeds up disk operations during load/save.
- **Index Structure Optimization:** Triggers Whoosh's built-in optimization to merge segments and eliminate overhead.
- **Searcher Caching:** Enables in-memory caching for repeated queries, reducing lookup time.

B.2 Text Processing Optimization

The text pipeline is encapsulated within the `EnhancedTextProcessor` class. It supports high-throughput ingestion through caching, batching, and parallelization.

```
class EnhancedTextProcessor:
    def __init__(self):
        self.processed_cache = {}  # Caches previously seen inputs
        self.batch_size = 1000     # Controls batch size
        self.pool = ThreadPool(processes=4)  # Parallel workers

    def process_text_batch(self, texts):
        return self.pool.map(self.process_text, texts)
```

Listing 7: Optimized Text Processing Class

Performance Features

- **Caching:** Prevents redundant computation by storing and reusing already processed documents.
- **Batch Processing:** Aggregates inputs for efficient bulk processing and I/O handling.
- **Thread Pooling:** Leverages multicore CPUs to accelerate preprocessing using Python's `ThreadPool`.

These optimizations significantly boost the scalability and responsiveness of the Indexer Node, enabling it to handle continuous document streams and high-frequency search queries in production-scale deployments.

B.3 DynamoDB Query Optimization

To reduce the latency and cost associated with high-volume DynamoDB operations, a set of optimizations was implemented in the Indexer Node's query logic. These include batch writes and a local caching layer to avoid redundant reads.

```
def optimize_dynamodb_queries(self):

    # Implement batch operations
    def batch_get_items(self, items):
        with self.dynamodb.Table('url-frontier').batch_writer() as
            batch:
            for item in items:
                batch.put_item(Item=item)

    # Add caching layer for reads
    def get_cached_item(self, table, key):
        cache_key = f"{table}:{key}"
        if cache_key in self.cache:
            return self.cache[cache_key]
        item = self.dynamodb.Table(table).get_item(Key=key)
        self.cache[cache_key] = item
        return item
```

Listing 8: DynamoDB Optimization Methods

Optimization Techniques

- **Batch Writes:** Uses `batch_writer()` to send multiple writes in a single API call, reducing network overhead and improving throughput.
- **Read Caching:** Locally caches recently queried items, reducing repetitive reads for popular or recently processed keys.

Benefits

- Minimizes API call volume to DynamoDB, lowering cost and contention.
- Improves overall responsiveness of the indexing pipeline under concurrent load.
- Reduces latency when querying frequently accessed document metadata or task states.

These combined optimizations yield a significantly more scalable and responsive indexing service, capable of handling sustained document streams and real-time querying under distributed deployment.

9 Security Review

9.1 A. AWS Resource Security

To maintain the integrity, confidentiality, and controlled access of distributed system components, AWS Identity and Access Management (IAM) roles and bucket policies were configured with the principle of least privilege. Each node (Master, Crawler, Indexer) has its own scoped permissions and operates within an isolated role.

1. IAM Policies

Master Node IAM Policy The Master Node coordinates system state, crawlers, and indexing pipelines. Its IAM role is allowed to send/receive messages from SQS, manage metadata in DynamoDB, and read/write crawl data to S3.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "sqs:SendMessage",
        "sqs:ReceiveMessage",
        "sqs:DeleteMessage",
        "dynamodb:PutItem",
        "dynamodb:GetItem",
        "dynamodb:UpdateItem",
        "s3:PutObject",
        "s3:GetObject"
      ],
      "Resource": [
        "arn:aws:sqs:*:*:crawl-*",
        "arn:aws:dynamodb:*:*:table/url-frontier",
        "arn:aws:s3:::crawl-data-bucket/*"
      ]
    }
  ]
}
```

Listing 9: Master Node IAM Policy

Crawler Node IAM Policy Crawler nodes operate under stricter constraints. They may receive tasks from SQS, upload crawled content to S3, and update metadata in DynamoDB.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "sqs:ReceiveMessage",
        "sqs:DeleteMessage",
        "s3:PutObject",
        "dynamodb:UpdateItem"
      ],
      "Resource": [
        "arn:aws:sqs:*:*:crawl-*",
        "arn:aws:s3:::my-crawl-bucket/*",
        "arn:aws:dynamodb:*:*:table/url-frontier"
      ]
    }
  ]
}
```

Listing 10: Crawler Node IAM Policy

Each node assumes a dedicated IAM role, ensuring no cross-component data access unless explicitly required. These scoped permissions reduce blast radius in case of node compromise.

2. S3 Bucket Security

To enforce encryption at rest, the system configures the crawl-data bucket with a strict bucket policy that rejects any object uploads lacking server-side encryption. This ensures data is not stored in plaintext.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "EnforceEncryption",
      "Effect": "Allow",
      "Principal": "*",
      "Action": "s3:PutObject",
      "Resource": "arn:aws:s3:::my-crawl-bucket/*",
      "Condition": {
        "StringNotEquals": {
          "s3:x-amz-server-side-encryption": "AES256"
        }
      }
    }
  ]
}
```

Listing 11: S3 Bucket Encryption Policy

Benefits:

- **Least Privilege Access:** Nodes have only the exact permissions needed for their role.
- **Data-at-Rest Security:** All S3 objects are encrypted using AES-256.
- **Reduced Risk Exposure:** IAM role separation isolates failures and limits escalation scope.

9.2 B. Network-Level Security

To safeguard inter-node communication and prevent unauthorized access, the distributed system was deployed inside a private Virtual Private Cloud (VPC) with strict security group rules and subnet isolation. The VPC and associated security configurations ensure that only authorized nodes can communicate over controlled ports.

1. VPC Isolation

The system operates within a dedicated Virtual Private Cloud named `project-vpc`, with the following properties:

- **VPC ID:** `vpc-0c1a2634028124a82`
- **IPv4 CIDR:** `10.0.0.0/16`
- **DNS Hostnames:** Enabled
- **Block Public Access:** Disabled (selective access allowed via security groups)
- **Route Table:** `rtb-00e36c6ff25b1885b` (custom routing for internal services)

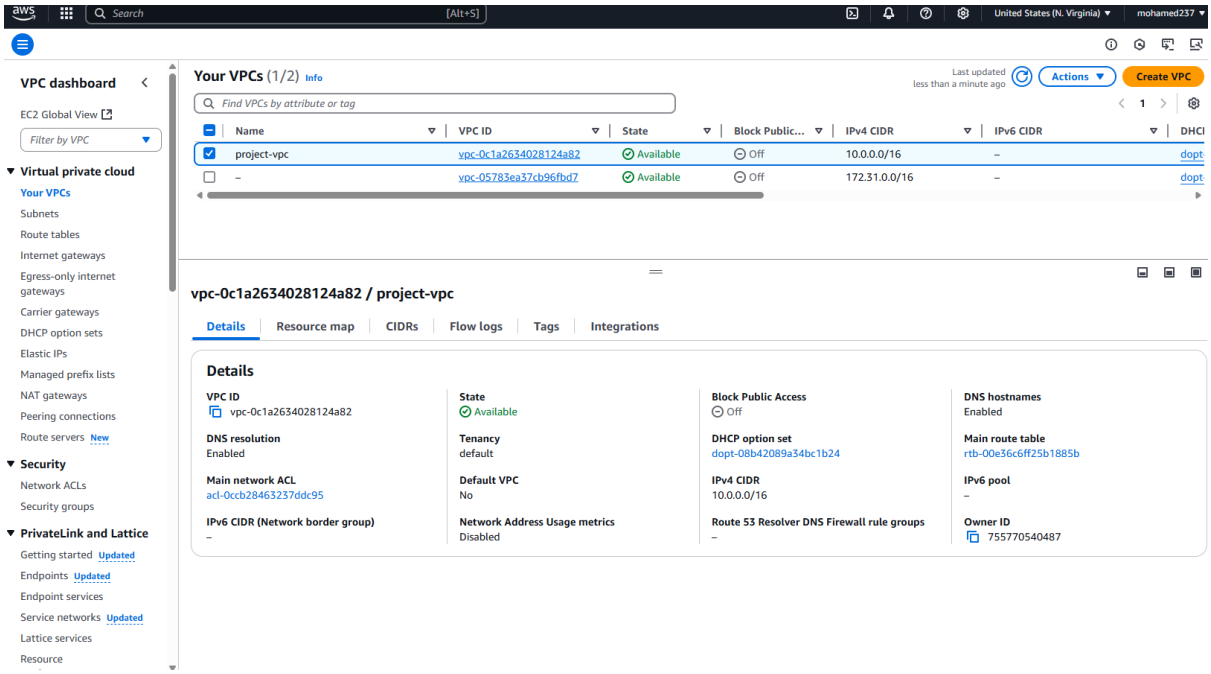


Figure 10: AWS VPC Dashboard – project-vpc configured with CIDR block 10.0.0.0/16

This VPC separates project infrastructure from the default VPC, supporting better network segmentation and control over routing and peering.

2. Security Groups

Each compute instance is attached to security groups that define allowed traffic rules. For example, the `Crawler_cluster_sg` security group:

- **Security Group ID:** `sg-04f9b61e4c894c235`
- **Description:** Allows SSH access for developer machines
- **Inbound Rules:**
  - Allow TCP on port 22 from trusted IPs (SSH access)
  - Internal port access (e.g., 5000, 8050) restricted to nodes within the VPC
- **Outbound Rules:**
  - Allow all outbound traffic to support SQS, S3, and DynamoDB communication

Best Practices Enforced:

- **Principle of Least Privilege:** Nodes are only allowed to access the ports and services they need.
- **Restricted SSH Access:** SSH is limited to developer IPs using a dedicated security group.
- **Private Networking:** Nodes communicate internally over the VPC to avoid exposure to the public internet.

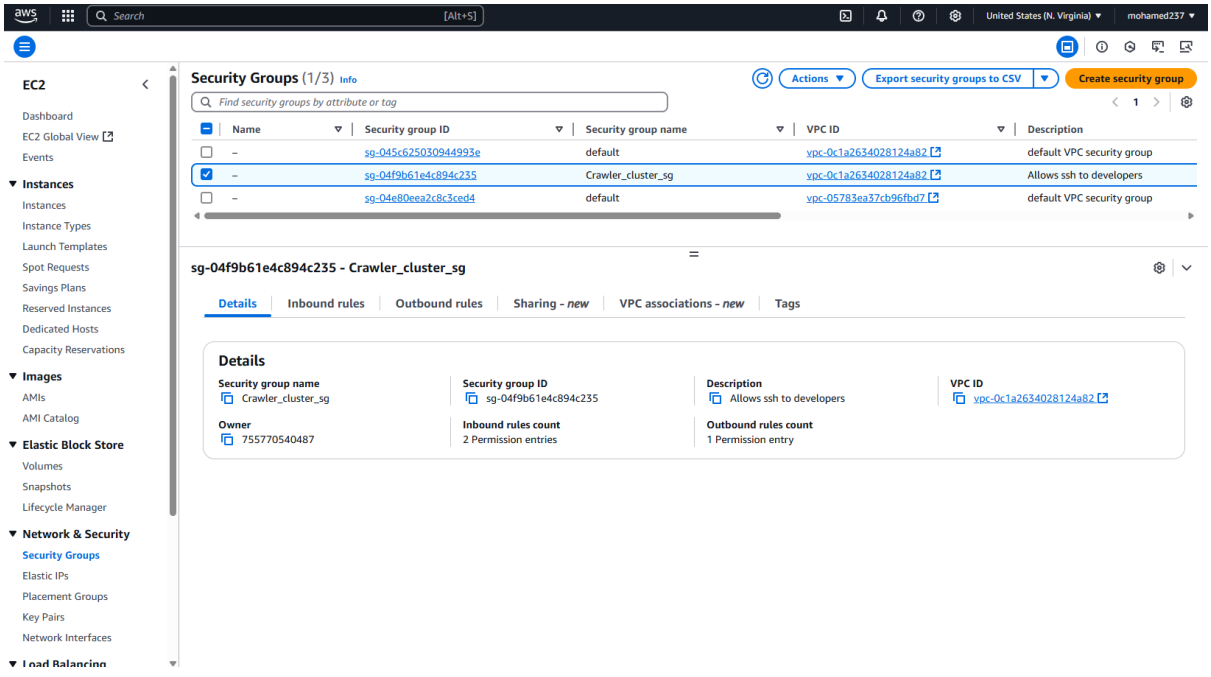


Figure 11: Security Group Configuration – `Crawler_cluster_sg` allowing limited SSH

Conclusion

The combination of a dedicated VPC and tightly scoped security groups ensures a secure network perimeter around all nodes, prevents unauthorized external access, and minimizes attack surfaces while enabling reliable internal communication across crawler, indexer, and master instances.

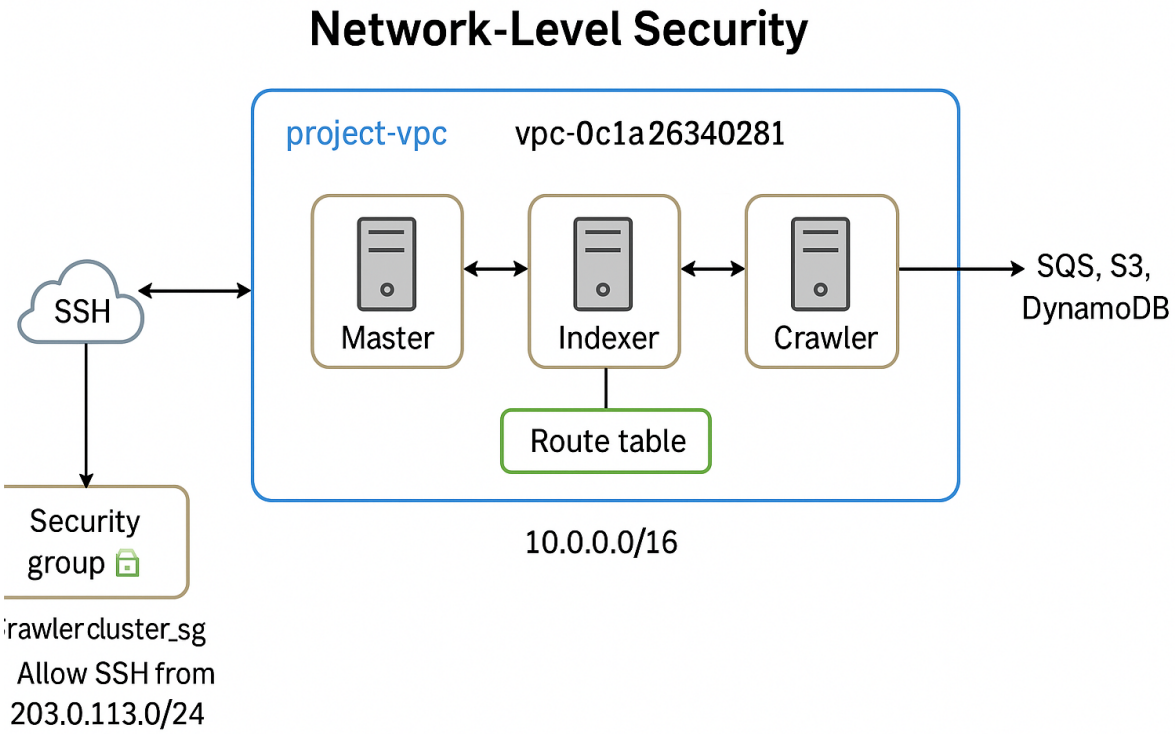


Figure 12: Network-Level Security Diagram: VPC Isolation, Security Groups, and AWS service communication

10 Dashboard API Documentation

This section documents the RESTful API endpoints exposed by the distributed web crawler dashboard. All responses are returned in JSON format.

10.1 Overview

The dashboard provides real-time monitoring and control of the distributed crawling system. It exposes RESTful endpoints and WebSocket events to interact with system components including the Master, Crawler, and Indexer nodes.

10.2 Base URL

```
http://localhost:5000/api/v1
```

10.3 Authentication

All endpoints require JWT-based authentication using the following format:

```
Authorization: Bearer <token>
```

10.4 API Endpoints

GET /status

Returns the current status of all system components.

Response:

```
{
  "status": "active",
  "master_node": {
    "status": "active",
    "processed_urls": 1000,
    "pending_urls": 500
  },
  "crawler_nodes": [{
    "id": "crawler-1",
    "status": "active",
    "last_heartbeat": "2024-03-20T10:00:00Z",
    "processed_tasks": 100
  }],
  "indexer_nodes": [{
    "id": "indexer-1",
    "status": "active",
    "last_heartbeat": "2024-03-20T10:00:00Z",
    "indexed_documents": 1000
  }]
}
```

POST /crawl/start

Initiates a new crawl with user-defined parameters.

Request Body:

```
{
  "seed_urls": ["https://example.com"],
  "max_pages": 1000,
  "max_depth": 3,
  "politeness_delay": 1.0
}
```

Response:

```
{
  "crawl_id": "crawl-123",
  "status": "started",
  "start_time": "2024-03-20T10:00:00Z"
}
```

POST /crawl/stop

Stops an ongoing crawl session.

Request Body:

```
{
  "crawl_id": "crawl-123"
}
```

Response:

```
{
  "status": "stopped",
  "stop_time": "2024-03-20T10:30:00Z"
}
```

GET /search

Executes a search query on the indexed content.

Query Parameters:

- `q` – Search query (required)
- `page` – Page number (default: 1)
- `per_page` – Results per page (default: 10)

Response:

```
{
  "total_results": 100,
  "page": 1,
  "per_page": 10,
  "results": [{
    "url": "https://example.com/page1",
    "title": "Example Page",
    "snippet": "This is a search result snippet...",
    "score": 0.85
  }]
}
```

GET /nodes

Returns information about all crawler and indexer nodes.

Response:

```
{
  "crawler_nodes": [{
    "id": "crawler-1",
    "status": "active",
    "last_heartbeat": "2024-03-20T10:00:00Z",
    "processed_tasks": 100,
    "memory_usage": "500MB"
  }],
  "indexer_nodes": [{
    "id": "indexer-1",
    "status": "active",
    "last_heartbeat": "2024-03-20T10:00:00Z",
    "indexed_documents": 1000,
    "index_size": "100MB"
  }]
}
```

POST /nodes/register

Registers a new node to the system.

Request Body:

```
{
  "node_type": "crawler",
  "node_id": "crawler-2",
  "capabilities": {
    "max_concurrent_requests": 10,
    "supported_content_types": ["text/html", "text/plain"]
  }
}
```

Response:

```
{
  "status": "registered",
  "node_id": "crawler-2",
  "registration_time": "2024-03-20T10:00:00Z"
}
```

GET /stats

Returns system-wide crawl, index, and queue statistics.

Response:

```
{
  "crawl_stats": {
    "total_urls_crawled": 1000,
    "total_bytes_crawled": "100MB",
    "average_crawl_rate": "10 pages/second",
    "success_rate": 0.95
  },
  "index_stats": {
    "total_documents": 1000,
    "index_size": "100MB",
    "average_indexing_rate": "5 documents/second"
  },
  "system_stats": {
    "active_crawlers": 2,
    "active_indexers": 1,
    "queue_sizes": {
      "crawl_tasks": 100,
      "index_tasks": 50
    }
  }
}
```

10.5 Error Responses

400 Bad Request

```
{
  "error": "Bad Request",
  "message": "Invalid request parameters",
  "details": {}
}
```

401 Unauthorized

```
{
  "error": "Unauthorized",
  "message": "Invalid or missing authentication token"
}
```

404 Not Found

```
{
  "error": "Not Found",
  "message": "Resource not found"
}
```

500 Internal Server Error

```
{
  "error": "Internal Server Error",
  "message": "An unexpected error occurred",
  "details": {}
}
```

10.6 Rate Limiting

The API is rate-limited to prevent abuse:

- 100 requests per minute per IP address
- 1000 requests per hour per API key

Response Headers:

```
X-RateLimit-Limit: 100
X-RateLimit-Remaining: 95
X-RateLimit-Reset: 1616236800
```

10.7 WebSocket Events

The dashboard provides real-time updates via WebSocket connections:

Connection

```
ws://localhost:5000/ws
```

Events

Node Status Updates

```
{
  "event": "node_status",
  "data": {
    "node_id": "crawler-1",
    "status": "active",
    "timestamp": "2024-03-20T10:00:00Z"
  }
}
```

Crawl Progress Updates

```
{
  "event": "crawl_progress",
  "data": {
    "crawl_id": "crawl-123",
    "urls_crawled": 100,
    "urls_pending": 900,
    "timestamp": "2024-03-20T10:00:00Z"
  }
}
```

Indexing Progress Updates

```
{
  "event": "indexing_progress",
  "data": {
    "documents_indexed": 100,
    "documents_pending": 900,
    "timestamp": "2024-03-20T10:00:00Z"
  }
}
```

11 User Manual

Overview

A scalable, fault-tolerant distributed web crawling and indexing system built with Python, AWS services, and modern web technologies.

Features

- **Distributed Architecture:** Scalable crawler nodes that can be deployed across multiple machines.
- **Fault Tolerance:** Automatic recovery from node failures and task retries.
- **Real-time Search:** Built-in search interface with advanced query capabilities.
- **AWS Integration:** Leverages AWS services (S3, DynamoDB, SQS) for reliable storage and messaging.
- **Web Interface:** Modern, responsive web UI for searching and submitting URLs.
- **Analytics:** Tracks crawling statistics and search analytics.
- **Politeness:** Respects `robots.txt` and implements crawl delays.
- **Content Processing:** Advanced text processing and indexing using Whoosh.

Architecture

The system consists of several key components:

- **Master Node:** Coordinates crawling tasks and manages worker nodes.
- **Crawler Nodes:** Distributed workers that crawl web pages.
- **Indexer Node:** Processes and indexes crawled content.
- **Web Interface:** User-friendly search interface and URL submission portal.

Getting Started

Prerequisites

- Python 3.8 or higher
- AWS account with appropriate permissions
- AWS CLI configured with credentials

Installation

1. Clone the repository:

```
git clone https://github.com/Adham-Osama11/distributed_crawler.git
cd distributed-web-crawler
```

2. Install dependencies:

```
pip install -r requirements.txt
```

3. Configure AWS credentials:

```
aws configure
```

Configuration

1. Set up AWS resources:
   - Create S3 buckets for crawl data and index storage.
   - Create DynamoDB tables for metadata and analytics.
   - Create SQS queues for task distribution.
2. Update configuration in `common/config.py`:

```
AWS_REGION = 'us-east-1'
CRAWL_DATA_BUCKET = 'my-crawl-data-bucket'
INDEX_DATA_BUCKET = 'my-index-data-bucket'
```

Running the System

1. Start the master node:

   ```
   python src/master/master_node.py
   ```

2. Start crawler nodes (on different machines if desired):

   ```
   python src/crawler/crawler_node.py
   ```

3. Start the indexer node:

   ```
   python src/indexer/indexer_node.py
   ```

4. Launch the web interface:

   ```
   python src/client/search_interface.py
   ```

Usage

Web Interface

Access the web interface at `http://localhost:5000` to:

- Search through crawled content.
- Submit new URLs for crawling.
- View system statistics.
- Monitor crawling progress.

API Endpoints

- `GET /search?q=`$\langle query \rangle$: $Searchtheindex$.`POST /submit-url` : $SubmitaURLforcrawling$.
- `GET /stats`: Get system statistics.
- `GET /suggest`: Get search suggestions.

Monitoring

The system provides several monitoring capabilities:

- Real-time statistics in the web interface.
- Detailed logs for each component.
- AWS CloudWatch metrics.
- DynamoDB analytics.

Development

Project Structure

```
src/
 master/         % Master node implementation
 crawler/        % Crawler node implementation
 indexer/        % Indexer node implementation
 client/         % Web interface and API
 common/         % Shared utilities and configuration
```

Adding New Features

1. Fork the repository.
2. Create a feature branch.
3. Implement your changes.
4. Submit a pull request.

License

This project is licensed under the MIT License. See the `LICENSE` file for details.

Contributing

Contributions are welcome! Please feel free to submit a Pull Request.

Contact

For questions and support, please open an issue in the GitHub repository.

Acknowledgments

- Scrapy for the web crawling framework.
- Whoosh for the search indexing.
- AWS for cloud infrastructure.
- Flask for the web interface.

12 Deployment Guide

This section provides a step-by-step guide to deploying and running the distributed web crawling system in a cloud environment. The system consists of the following main components:

- **Master Node:** Coordinates crawling tasks and manages the overall system.
- **Crawler Nodes:** Perform the actual web crawling.
- **Indexer Nodes:** Process and index crawled content.
- **Dashboard:** Web interface for monitoring and control.

12.1 Prerequisites

AWS Account Setup

1. Create an AWS account if you do not already have one.
2. Install and configure the AWS CLI:

```
pip install awscli
aws configure
```

3. Required AWS Services:
   - Amazon S3 (for content storage)
   - Amazon SQS (for message queues)
   - Amazon DynamoDB (for state management)
   - Amazon EC2 (for running the nodes)

Required IAM Permissions

Create an IAM user with the following permissions:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "s3:*",
                "sqs:*",
                "dynamodb:*",
                "ec2:*"
            ],
            "Resource": "*"
        }
    ]
}
```

Listing 12: IAM Policy JSON

Python Environment

- Python 3.8 or higher
- Required Python packages:

```
pip install -r requirements.txt
```

- Key dependencies: `boto3`, `scrapy`, `whoosh`, `nltk`, `flask`, `psutil`, `crochet`

12.2 System Configuration

Environment Variables

Create a `.env` file in the project root:

```
# AWS Configuration
AWS_REGION=us-east-1
AWS_ACCESS_KEY_ID=my_access_key
AWS_SECRET_ACCESS_KEY=my_secret_key

# Queue Names
CRAWL_TASK_QUEUE=crawl-tasks
CRAWL_RESULT_QUEUE=crawl-results
INDEX_TASK_QUEUE=index-tasks
MASTER_COMMAND_QUEUE=master-commands

# Storage
CRAWL_DATA_BUCKET=my-crawl-bucket
INDEX_BUCKET=my-index-bucket

# System Settings
HEARTBEAT_INTERVAL=30
MAX_RETRIES=3
TASK_TIMEOUT=300
INDEX_BATCH_SIZE=100

# Dashboard
DASHBOARD_HOST=0.0.0.0
DASHBOARD_PORT=8050

# Client
CLIENT_HOST=0.0.0.0
CLIENT_PORT=5000
```

AWS Resource Setup

S3 Buckets:

```
aws s3 mb s3://my-crawl-bucket
aws s3 mb s3://my-index-bucket
```

SQS Queues:

```
aws sqs create-queue --queue-name crawl-tasks
aws sqs create-queue --queue-name crawl-results
aws sqs create-queue --queue-name index-tasks
aws sqs create-queue --queue-name index-results
aws sqs create-queue --queue-name master-commands
```

DynamoDB Tables (optional manual creation):

```
aws dynamodb create-table \
  --table-name url-frontier \
  --attribute-definitions AttributeName=url,AttributeType=S
     AttributeName=job_id,AttributeType=S \
  --key-schema AttributeName=url,KeyType=HASH AttributeName=job_id,
     KeyType=RANGE \
  --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5
```

12.3 Deployment Steps

Master Node

```
python src/master/master_node.py
```

Crawler Node

```
python src/crawler/crawler_node.py
```

Indexer Node

```
python src/indexer/indexer_node.py
```

Dashboard

```
python src/monitoring/dashboard.py
```

Accessible at `http://localhost:5000`

12.4 Scaling the System

Horizontal Scaling Deploy multiple crawler and indexer nodes across EC2 instances. They register automatically with the Master Node.

Vertical Scaling Upgrade instance types or adjust service limits (e.g., DynamoDB capacity units) for improved performance.

12.5 Monitoring and Maintenance

System Monitoring

- Use the **Dashboard** at `localhost:5000`
- Use **CloudWatch** to track:
  - SQS queue length
  - DynamoDB read/write throughput
  - S3 usage

Maintenance Tasks

```
aws s3 sync whoosh_index/ s3://my-index-bucket/backups/$(date +%Y%m%d)/
```

Listing 13: Back up index to S3

```
aws s3 rm s3://my-crawl-bucket/ --recursive --exclude "*" --include "*/
    $(date -d '30 days ago' +%Y%m%d)*"
```

Listing 14: Clean up old crawl data

```
from src.indexer.indexer_node import WhooshIndex
WhooshIndex().optimize()
```

Listing 15: Optimize Whoosh Index

12.6 Troubleshooting

Common Issues

- **Crawler not registering:** Check credentials, SQS permissions, and network.
- **High memory usage:** Reduce concurrent requests and tune delays.
- **Slow indexing:** Optimize batch size and S3 reads.

Logs

- Master Node: `master_node.log`
- Crawler Node: `crawler_node.log`
- Indexer Node: `indexer_node.log`
- Dashboard: `dashboard.log`

12.7 Security Considerations

- Use IAM roles with least privilege
- Enable S3 encryption and bucket policies
- Use HTTPS for dashboard and secure credential storage
- Sanitize all URLs and log access activity

12.8 Performance Tuning

Crawler Settings

```
settings.update({
    'CONCURRENT_REQUESTS': 16,
    'DOWNLOAD_DELAY': 0.5,
    'RANDOMIZE_DOWNLOAD_DELAY': True,
    'COOKIES_ENABLED': False
})
```

Indexer Settings

```
INDEX_BATCH_SIZE = 100
MAX_DOCUMENTS_PER_INDEX = 10000
INDEX_OPTIMIZATION_INTERVAL = 3600
```

DynamoDB Settings

```
ReadCapacityUnits = 10
WriteCapacityUnits = 10
```

13 Conclusion

n conclusion, this distributed web crawler system represents a well-balanced approach to the complex problem of web crawling at scale. It effectively combines distributed systems principles with cloud services and modern text processing techniques, resulting in a solution that is both powerful and maintainable. The clear separation of concerns across different components makes the system modular and extensible, while the comprehensive fault tolerance mechanisms ensure reliability in production environments.