

# System Architecture Document

## Plan for data structures

### 1. Data Flow Overview

1. **Master Node** reads **Seed URLs** from S3 and enqueues **Task Messages** (**Master** → **Crawler**) via RabbitMQ [Medium](#).
2. **Crawler Nodes** dequeue messages, fetch pages, build **Crawled Page Records**, upload them to S3, extract links, then enqueue new Task Messages and emit **Crawl Metadata** to MongoDB [ZenRows](#).
3. **Indexer Nodes** read Crawled Page Records from S3, process them into **Inverted Index Entries** in Whoosh/Elasticsearch, and serve search queries [GeeksforGeeks](#).

#### 1.1 Master Node Data Flow

The Master Node functions as the central controller of the distributed crawler system, coordinating the activities of all other components. Its primary responsibility is to initiate and manage the crawling process through the following data flow operations:

##### 1.1.1 Input Data Processing

- **Seed URL Loading:** The Master Node retrieves the initial set of URLs (seed URLs) from an S3 bucket. These seeds serve as the starting points for the web crawling process.
- **Configuration Parameters:** The Master Node loads crawling parameters from configuration files, including crawl depth limits, domain restrictions, politeness settings, and rate limiting parameters.

##### 1.1.2 Task Generation and Distribution

- **URL Normalization:** Before distribution, the Master Node normalizes URLs to ensure consistency (handling of trailing slashes, URL encoding, etc.).
- **URL Deduplication:** The Master removes duplicate URLs to prevent redundant crawling.

- **Task Message Creation:** For each URL to be crawled, the Master creates a task message containing:
  - URL to crawl
  - Crawl parameters (depth, domain restrictions)
  - Task ID (unique identifier)
  - Timestamp
  - Priority level
- **Task Queuing:** The Master publishes task messages to RabbitMQ queues using a topic-based exchange system that allows for:
  - Domain-based routing (different queues for different domains to control crawl rates)
  - Priority-based scheduling (higher priority for seed URLs or important domains)

### 1.1.3 System Monitoring and Management

- **Health Check Processing:** The Master continuously monitors heartbeat signals from Crawler and Indexer nodes.
- **Node Failure Detection:** When a node fails to send heartbeats within the configured timeout period, the Master:
  - Marks the node as failed
  - Retrieves its assigned tasks
  - Re-queues those tasks for other crawlers
- **System Metrics Collection:** The Master collects and processes system-wide metrics, including:
  - Crawl progress (URLs crawled, queued, failed)
  - Node performance statistics
  - Queue depths and processing rates

## 1.2 Crawler Node Data Flow

Crawler Nodes represent the distributed workers responsible for fetching and processing web pages. Their data flow encompasses several stages from task reception to result distribution:

### 1.2.1 Task Reception and Preparation

- **Task Message Consumption:** Each Crawler Node consumes task messages from RabbitMQ queues it subscribes to.
- **Task Validation:** The Crawler validates received tasks to ensure they contain all required parameters and are still valid (e.g., not expired).

- **Resource Allocation:** The Crawler allocates resources (memory, network connections) for the upcoming crawl operation.

### 1.2.2 Web Page Fetching

- **Robots.txt Processing:** Before fetching any page, the Crawler checks the domain's robots.txt file to:
  - Verify crawling permissions
  - Determine crawl delay requirements
  - Identify disallowed paths
- **HTTP Request Generation:** The Crawler constructs HTTP requests with:
  - Appropriate user-agent strings
  - Required headers
  - Timeout settings
- **Response Handling:** Upon receiving HTTP responses, the Crawler:
  - Processes status codes (following redirects when appropriate)
  - Handles errors (retrying with exponential backoff for temporary failures)
  - Extracts content and metadata

### 1.2.3 Content Processing and Storage

- **Content Extraction:** The Crawler extracts the following from fetched pages:
  - Raw HTML content
  - Text content (with HTML tags removed)
  - Metadata (page title, description, keywords)
  - HTTP headers
- **Page Record Creation:** The Crawler assembles a Crawled Page Record containing:
  - Original URL
  - Final URL (after redirects)
  - Timestamp
  - HTTP status code
  - Content type
  - Raw HTML
  - Extracted text
  - Metadata
- **S3 Storage:** The Crawler uploads the Crawled Page Record to S3, using:
  - Content-based hashing for object naming
  - Appropriate S3 bucket organization (by domain/date)
  - Compression for efficient storage

#### 1.2.4 Link Extraction and Task Generation

- **URL Discovery:** The Crawler parses the HTML to extract all links using:
  - Anchor tags (`<a href>`)
  - Link tags (`<link>`)
  - Redirects
  - JavaScript parsed URLs (when applicable)
- **URL Filtering and Normalization:** Discovered URLs are:
  - Normalized to absolute form
  - Filtered based on crawl policy (depth, domain restrictions)
  - Deduped against the crawler's local cache
- **Task Message Creation:** For each new URL, the Crawler creates a task message with:
  - URL to crawl
  - Updated depth information
  - Parent URL reference
  - Discovery timestamp
- **Task Publication:** New tasks are published back to RabbitMQ for further crawling.

#### 1.2.5 Metadata Recording

- **Crawl Metadata:** The Crawler records detailed metadata about each crawl operation:
  - Performance metrics (fetch time, processing time)
  - Error conditions
  - Link graph data (source URL → target URLs)
  - Content statistics (page size, word count)
- **MongoDB Storage:** This metadata is stored in MongoDB's ZenRows collection, which provides:
  - Fast querying capabilities
  - Efficient storage of semi-structured data
  - Indexing on frequently queried fields

### 1.3 Indexer Node Data Flow

Indexer Nodes are responsible for processing crawled content into searchable indices and serving search queries. Their data flow involves handling large volumes of text data efficiently:

### 1.3.1 Content Acquisition

- **S3 Monitoring:** Indexer Nodes monitor S3 buckets (via event notifications or polling) for new Crawled Page Records.
- **Content Retrieval:** When new content is available, Indexers:
  - Download Crawled Page Records from S3
  - Verify data integrity
  - Prioritize processing based on content type and importance

### 1.3.2 Text Processing

- **Content Extraction:** Indexers extract text content from the Crawled Page Records.
- **Text Preprocessing:** The extracted text undergoes:
  - Language detection
  - Tokenization (breaking text into words/tokens)
  - Stop word removal
  - Stemming/lemmatization (reducing words to base forms)
  - Named entity recognition
- **Document Analysis:** Additional processing includes:
  - Term frequency calculation
  - Inverse document frequency calculation
  - Keyword extraction
  - Topic modeling (optional)

### 1.3.3 Index Construction

- **Whoosh Index Building:** For smaller deployments or as a backup, Indexers build Whoosh indices:
  - Create document objects with processed text and metadata
  - Update in-memory index structures
  - Periodically flush to disk for persistence
- **Elasticsearch Integration:** For larger deployments, Indexers communicate with Elasticsearch:
  - Create and update index documents
  - Configure index mappings and analyzers
  - Manage index sharding and replication
- **Inverted Index Creation:** Both systems create inverted indices mapping:

- Terms to document IDs
- Term positions within documents
- Term frequencies and weights

#### 1.3.4 Search Service

- **Query Reception:** Indexers receive search queries via:
  - REST API endpoints
  - Internal system calls
- **Query Processing:** Incoming queries undergo:
  - Parsing and normalization
  - Query expansion (synonyms, related terms)
  - Boolean operation resolution
- **Result Generation:** Search results are compiled with:
  - Relevance scoring
  - Result ranking
  - Snippet generation
  - Metadata enrichment

### 1.4 Cross-Component Data Exchange

The system relies on several standardized data structures that flow between different components:

#### 1.4.1 Task Message Structure


Task Messages facilitate communication between the Master Node and Crawler Nodes, containing:



```
1  {
2      "task_id": "uuid-string",
3      "url": "https://example.com/page",
4      "timestamp": "2025-04-26T14:30:00Z",
5      "crawl_depth": 2,
6      "max_depth": 3,
7      "domain_restrictions": ["example.com"],
8      "priority": 5,
9      "retry_count": 0,
10     "parent_url": "https://example.com"
11 }
```

#### 1.4.2 Crawled Page Record Structure

Crawled Page Records represent the output of Crawler Nodes stored in S3:



```
1  {
2    "record_id": "hash-based-identifier",
3    "original_url": "https://example.com/page",
4    "final_url": "https://example.com/final-page",
5    "crawl_timestamp": "2025-04-26T14:35:22Z",
6    "http_status": 200,
7    "content_type": "text/html",
8    "headers": {
9      "Content-Length": "24506",
10     "Server": "nginx/1.18.0",
11     "Last-Modified": "2025-04-25T09:12:00Z"
12   },
13   "html_content": "base64-encoded-compressed-html",
14   "text_content": "extracted plain text",
15   "metadata": {
16     "title": "Example Page Title",
17     "description": "Meta description content",
18     "keywords": ["example", "page", "keywords"]
19   },
20   "links": [
21     {"url": "https://example.com/link1", "anchor_text": "Link 1"},
22     {"url": "https://example.com/link2", "anchor_text": "Link 2"}
23   ]
24 }
```

### 1.4.3 Crawl Metadata Structure

Crawl Metadata stored in MongoDB provides operational insights:





```
1  {
2    "task_id": "uuid-string",
3    "url": "https://example.com/page",
4    "crawler_id": "crawler-node-12",
5    "start_time": "2025-04-26T14:35:10Z",
6    "end_time": "2025-04-26T14:35:22Z",
7    "status": "completed",
8    "http_status": 200,
9    "fetch_time_ms": 450,
10   "processing_time_ms": 320,
11   "content_size_bytes": 24506,
12   "links_discovered": 42,
13   "errors": [],
14   "robots_txt_compliant": true
15 }
```

#### 1.4.4 Inverted Index Entry Structure

Inverted Index Entries represent the core search data structure:



```
1  {
2    "term": "example",
3    "documents": [
4      {
5        "doc_id": "hash-based-identifier",
6        "term_frequency": 12,
7        "positions": [5, 67, 124, 389],
8        "tf_idf_score": 0.875
9      },
10     {
11       "doc_id": "another-document-id",
12       "term_frequency": 3,
13       "positions": [45, 189, 302],
14       "tf_idf_score": 0.43
15     }
16   ],
17   "document_frequency": 2,
18   "inverse_document_frequency": 0.698
19 }
```

## 1.5 Data Storage Systems

The distributed crawler leverages multiple storage systems optimized for different aspects of the data flow:

### 1.5.1 Amazon S3

- **Seed URL Storage:**
  - Bucket: `crawler-seed-urls`
  - Format: Plain text files with one URL per line or JSON arrays
  - Access Pattern: Read-only by Master Node at crawl initialization
- **Crawled Content Storage:**
  - Bucket: `crawler-page-content`
  - Organization: Hierarchical by domain and date (e.g., `example.com/2025/04/26/hash.json.gz`)
  - Format: Compressed JSON (Crawled Page Records)
  - Access Pattern: Write by Crawler Nodes, Read by Indexer Nodes

### 1.5.2 RabbitMQ

- **Crawl Task Queues:**
  - Exchange: `crawl-tasks` (topic exchange)
  - Queue Structure:
    - Domain-specific queues (e.g., `tasks.example.com`)
    - Priority queues (e.g., `tasks.priority.high`)
  - Message Properties: Persistent, TTL (time-to-live) = 24 hours
  - Access Pattern: Publish by Master Node, Consume by Crawler Nodes
- **System Control Queues:**
  - Exchange: `system-control` (direct exchange)
  - Queue Structure:
    - `node.commands` (for start/stop/pause commands)
    - `node.heartbeats` (for health monitoring)
  - Access Pattern: Bidirectional between Master and worker nodes

### 1.5.3 MongoDB (ZenRows Collection)

- **Collection Structure:**
  - `crawl_metadata`: Operational metrics and link graph data
  - `system_metrics`: System-wide performance statistics
  - `url_frontier`: URLs discovered but not yet crawled
  - `domain_policies`: Domain-specific crawling rules
- **Indexes:**

- URL-based indexes for fast lookups
- Timestamp indexes for temporal queries
- Crawler ID indexes for node-specific queries

#### 1.5.4 Search Index Storage

- **Whoosh Index:**
  - Directory Structure: Split by domain or time periods
  - Format: Native Whoosh index files
  - Storage Location: EBS volumes attached to Indexer Nodes
- **Elasticsearch:**
  - Index Naming: `web-crawler-yyyy-mm` (monthly indices)
  - Sharding: By domain (for multi-tenant isolation)
  - Replication: Minimum 2 replicas for fault tolerance
  - Storage: Elasticsearch-managed with snapshot backups to S3

### 1.6 Data Flow Control Mechanisms

To ensure system stability and performance, several control mechanisms regulate the data flow:

#### 1.6.1 Rate Limiting

- **Domain-specific Throttling:**
  - Configurable crawl delays per domain (respecting robots.txt)
  - Maximum concurrent connections per domain
  - Token bucket algorithm implementation for sustained rate control
- **Global Throttling:**
  - System-wide limits on crawling rate
  - Dynamic adjustment based on system load
  - Circuit breakers to prevent system overload

#### 1.6.2 Prioritization System

- **URL Prioritization:**
  - Freshness-based scoring (recently updated pages get higher priority)
  - Depth-based scoring (lower depths get higher priority)
  - Domain importance weighting
  - Content type prioritization (HTML over images, etc.)
- **Resource Allocation:**
  - Higher priority tasks receive more processing resources

- Preemptive scheduling for critical crawl tasks
- Adaptive resource allocation based on task importance

### 1.6.3 Error Handling and Recovery

- **Task Retry Logic:**
  - Exponential backoff for transient failures
  - Dead letter queues for persistent failures
  - Maximum retry limits with escalation to human operators
- **Data Recovery Mechanisms:**
  - Checkpointing of crawl state
  - Transaction logs for critical operations
  - Data reconciliation procedures for consistency

### 1.6.4 Flow Control Feedback Loops

- **Backpressure Mechanisms:**
  - Queue depth monitoring with crawl rate adjustment
  - Storage capacity awareness
  - Indexing latency feedback to crawl rate
- **System Balance Control:**
  - Crawler-to-indexer ratio management
  - Work distribution optimization
  - Resource utilization balancing

## 2. Data Types & Schemas

### 2.1 Seed URLs



```
1 ["http://example.com", "http://foo.org"]
```

### 2.2 Task Message (Master → Crawler)



```
1 {  
2   "task_id": "uuid-1234",  
3   "url": "http://example.com/page1",  
4   "depth": 2,  
5   "max_depth": 4  
6 }
```

### 2.3 Crawled Page Record



```
1  {
2    "url": "http://example.com/page1",
3    "fetched_at": "2025-04-18T18:00:00Z",
4    "status_code": 200,
5    "title": "Example Domain",
6    "html": "<!doctype html><html>...</html>",
7    "text": "This domain is for use in illus
trative examples...",
8    "links": [
9      "http://example.com/page2",
10     "http://example.com/page3"
11   ]
12 }
```

## 2.4 Inverted Index Entry



```
1  {
2    "distributed": ["http://example.com/page1",
3    "http://foo.org/doc2"],
4    "python":      ["http://bar.com/tutorial", "h
http://example.com/page5"]
5  }
```

## 2.5 Crawl Metadata Record

```
1  {
2    "url": "http://example.com/page1",
3    "status": "failed",
4    "error": "Timeout",
5    "retries": 2,
6    "last_attempt": "2025-04-18T18:05:00Z"
7  }
```

## 3. Storage Services & Rationale

Data Type	Format	Service	Why?
Seed URLs	JSON array	S3 (small JSON file)	Simple, durable, versionable
Task Messages	JSON object	RabbitMQ / Celery+Redis	Reliable queue semantics, easy retry
Crawled Page Records	JSON document	S3	Large payloads, low-cost, scalable
Inverted Index Entries	Whoosh/Elasticsearch index	Whoosh on VM or AWS Elasticsearch	Fast term→doc lookups
Crawl Metadata Records	JSON document	MongoDB / DynamoDB	Flexible schema for monitoring & retries

– S3 is the de facto choice for storing large HTML/text blobs securely and cheaply [AWS Documentation](#).



- RabbitMQ (or Celery/Redis) ensures tasks survive restarts and can be re-queued on failure [Informatica Documentation](#).
- Whoosh is lightweight for prototypes; Elasticsearch if you need scale and advanced search [Baeldung](#).

## **4. Sample Code: RabbitMQ JSON Serialization**



```
1  import pika, json
2  from uuid import uuid4
3
4  def send_task(url, depth, max_depth):
5      connection = pika.BlockingConnection(pika.
6      ConnectionParameters('rabbitmq_host'))
7      channel = connection.channel()
8      channel.queue_declare(queue='task_queue',
9      durable=True)
10
11      message = {
12          "task_id": str(uuid4()),
13          "url":      url,
14          "depth":    depth,
15          "max_depth": max_depth
16      }
17      body = json.dumps(message).encode('utf-8')
18      channel.basic_publish(
19          exchange='',
20          routing_key='task_queue',
21          body=body,
22          properties=pika.BasicProperties(delive
23          ry_mode=2) # make message persistent
24      )
25      connection.close()
26
27  # Example usage:
28  send_task("http://example.com/page1", depth=2,
29  max_depth=4)
```

# API interfaces for communication:

## 1. Introduction

To enable modular, scalable, and fault-tolerant communication between the various components of the distributed system, a set of well-defined API interfaces EX:RESTful, has been designed. These APIs standardize how components like the Master Node, Crawler Nodes, Indexer Nodes, Client Interface, and the Distributed Task Queue communicate and exchange data.

**The design prioritizes:**

- Loose coupling between components
- Fault detection and recovery
- Task and data orchestration
- Monitoring and control

## 2. Detailed API Interface Specifications

### 2.1. Client ↔ Master Node

a. Start Crawl

```
1 POST /api/v1/crawl/start
```

Payload

```
1 {  
2   "seed_urls": ["https://example.com", "https://site.org"],  
3   "depth_limit": 2,  
4   "domain_restriction": true  
5 }
```

Response:

```
1
2 { "crawl_id": "c123", "status": "started" }
```

b. Get Crawl Status

```
1
2 GET /api/v1/crawl/status/{crawl_id}
```

Response:

```
1
2 {
3   "crawl_id": "c123",
4   "status": "running",
5   "pages_crawled": 150,
6   "errors": []
7 }
```

## 2.2 Master Node → Crawler Nodes

Mechanism: Via task queue

```
1
2 {
3   "task_id": "task456",
4   "url": "https://example.com/page",
5   "depth": 1,
6   "crawl_id": "c123"
7 }
```

## 2.3. Crawler Nodes → Master Node

a. Status Report

```
1
2 POST /api/v1/crawler/report
```

Payload:

```
1
2 {
3   "task_id": "task456",
4   "crawler_id": "crawler-2",
5   "status": "completed",
6   "extracted_urls": ["https://example.com/page2"],
7   "errors": []
8 }
```

## 2.4 Crawler → Indexer Nodes

Mechanism: REST or Queue

Endpoint (REST):

```
1
2 POST /api/v1/indexer/submit
```

Payload:

```
1
2 {
3   "crawl_id": "c123",
4   "url": "https://example.com/page",
5   "content": "<html>...</html>",
6   "text": "Extracted readable content",
7   "metadata": { "title": "Page Title", "keywords": ["a", "b", "c"] }
8 }
```

## 2.5. Indexer Nodes → Master Node

a. Indexing Status

```
1
2 POST /api/v1/indexer/report
```

Payload:

```
1
2 {
3   "crawl_id": "c123",
4   "indexer_id": "indexer-1",
5   "status": "success",
6   "processed_pages": 10
7 }
```

## 2.6. Client ↔ Indexer Node

### a. Search Query

```
1
2 GET /api/v1/search?q=distributed+systems
```

Response:

```
1
2 {
3   "results": [
4     { "url": "https://site.org/post1", "title": "Intro to Distributed Systems", "
5       snippet": "... " },
6     { "url": "https://example.com/page", "title": "Distributed Architectures", "
7       snippet": "... " }
8   ]
9 }
```

## 3. Fault-Tolerance Communication

### 3.1 Introduction to Fault Tolerance in Distributed Web Crawling

In our distributed web crawling system, fault tolerance is crucial for ensuring system reliability and continuous operation despite potential failures. During Phase 1, we focus primarily on strategies to handle crawler node failures, as these components represent the most numerous and active parts of our system, responsible for fetching and processing web content.

Fault tolerance enables our system to detect, isolate, and recover from failures without significant service disruption or data loss. This report outlines our comprehensive approach to fault tolerance, with a particular emphasis on crawler node failures during this initial phase.

### 3.2 Crawler Node Failure Scenarios and Impact Analysis

We've identified several potential crawler node failure scenarios:

1. **Complete Node Failure:** A crawler VM crashes, becomes unresponsive, or is terminated unexpectedly.
2. **Network Partition:** A crawler node becomes isolated due to network connectivity issues.
3. **Performance Degradation:** A crawler node experiences severe slowdown but remains partially functional.
4. **Erratic Behavior:** A crawler node functions but produces incorrect or inconsistent results.

The impact of these failures includes:

- Incomplete crawling of assigned URLs
- Loss of already crawled but not yet reported data
- Potential system bottlenecks if failures are not addressed promptly
- Reduced overall system throughput

## 3.3 Fault Detection Mechanisms

### 3.3.1 Heartbeat Monitoring System

We will implement a heartbeat monitoring system where:

- Each crawler node sends periodic status updates (heartbeats) to the master node
- The heartbeat contains status information including:
  - Node health metrics (CPU, memory, network utilization)
  - Current crawl task progress
  - Number of URLs processed
  - Timestamp of last successful crawl

### 3.3.2 Task Timeout Implementation

- Each crawling task will have an associated timeout period
- The timeout period will be dynamically calculated based on:
  - URL complexity (depth, domain)
  - Historical performance data for similar URLs
  - Current system load
- If a crawler node doesn't complete a task within the timeout period, the master node will consider it potentially failed

### 3.3.3 Error Reporting Framework

- Crawler nodes will explicitly report errors to the master node
- Error reports will include:
  - Error type and severity classification
  - Task identifier and URL being processed
  - Relevant stack traces or error messages
  - System state at time of error

## 3.4 Failure Response Strategies

### 3.4.1 Task Re-queuing Mechanism

When a crawler node failure is detected:

1. The master node will mark all tasks assigned to the failed node as "pending reassignment"
2. Tasks will be prioritized for reassignment based on:
  - Task importance/priority
  - Time in queue
  - System resource availability
3. Tasks will be redistributed to active crawler nodes using a load-balancing algorithm



### 3.4.2 Checkpointing and Progress Tracking

- Crawler nodes will periodically report their progress to the master node
- Progress reports will include:
  - Current crawl depth
  - URLs discovered but not yet crawled
  - Status of robots.txt compliance checks
- This information will be stored persistently to enable task resumption after failure

### 3.4.3 Duplicate Task Detection

- When tasks are reassigned due to node failure, the system will implement duplicate detection
- If the originally assigned node recovers and attempts to submit results, the system will:
  - Detect duplicate submission
  - Select the most complete or recent result
  - Log the duplicate for analysis

## 3.5 Data Persistence and Recovery

### 3.5.1 Intermediate Result Storage

- Crawler nodes will periodically store intermediate results to durable cloud storage
- Data to be stored includes:
  - Parsed HTML content
  - Extracted URLs
  - Metadata about crawled pages
- Storage will use atomic write operations to prevent data corruption

### 3.5.2 Crawl Queue Persistence

- The master node's task queue will be backed by persistent storage
- Queue operations (enqueue, dequeue, task status updates) will be transactional
- The queue state will be recoverable after system restarts

### 3.5.3 Distributed Task State

- Task state information will be stored in a distributed database
- State transitions will be atomic and consistent
- States will include: pending, assigned, in-progress, completed, failed, reassigned

## 3.6 Scalability Considerations for Fault Tolerance

### 3.6.1 Dynamic Node Replacement

- The system will be designed to allow dynamic addition of new crawler nodes
- When persistent failure of a crawler node is detected, the master can:
  - Request provisioning of a replacement node
  - Incorporate the new node into the crawler pool
  - Begin assigning tasks to the new node

### 3.6.2 Load Balancing During Recovery

- After node failures, the load balancing algorithm will adjust to:
  - Prevent overloading remaining nodes
  - Prioritize critical crawling paths
  - Maintain politeness constraints across all domains

### 3.6.3 Gradual Recovery Process

- To prevent system instability during recovery:
  - Tasks will be reassigned gradually
  - System monitoring will be intensified
  - Thresholds for triggering additional fault responses will be temporarily adjusted

## 3.7 Implementation Plan for Phase 1

For Phase 1, we will focus on implementing:

1. **Basic Heartbeat Mechanism:**
  - Simple periodic status messages from crawler to master
  - Timeout-based failure detection
  - Basic crawler health metrics collection
2. **Initial Task Re-queuing Logic:**
  - URL task tracking in master node
  - Simple FIFO-based reassignment of tasks from failed nodes
  - Task state management (assigned, completed, failed)
3. **Minimal Persistent Storage:**
  - URL queue persistence using a simple storage mechanism
  - Basic crawler task state persistence
  - Crawled data temporary storage for recovery

## 3.8 Testing Strategy for Fault Tolerance

To ensure our fault tolerance mechanisms work as expected, we will implement the following testing approaches:

1. **Simulated Failure Testing:**
  - Programmatically terminate crawler nodes during operation
  - Artificially introduce network partitions
  - Create resource exhaustion situations (memory, CPU)
2. **Recovery Metrics Collection:**
  - Measure time from failure detection to task reassignment
  - Track success rate of reassigned tasks
  - Measure impact on overall system throughput during failure scenarios
3. **Edge Case Testing:**
  - Multiple simultaneous node failures
  - Master node temporary unavailability
  - Partial failures and slow degradation scenarios

## 3.9 Future Expansion of Fault Tolerance (Beyond Phase 1)

While Phase 1 focuses on crawler node failures, subsequent phases will expand fault tolerance to include:

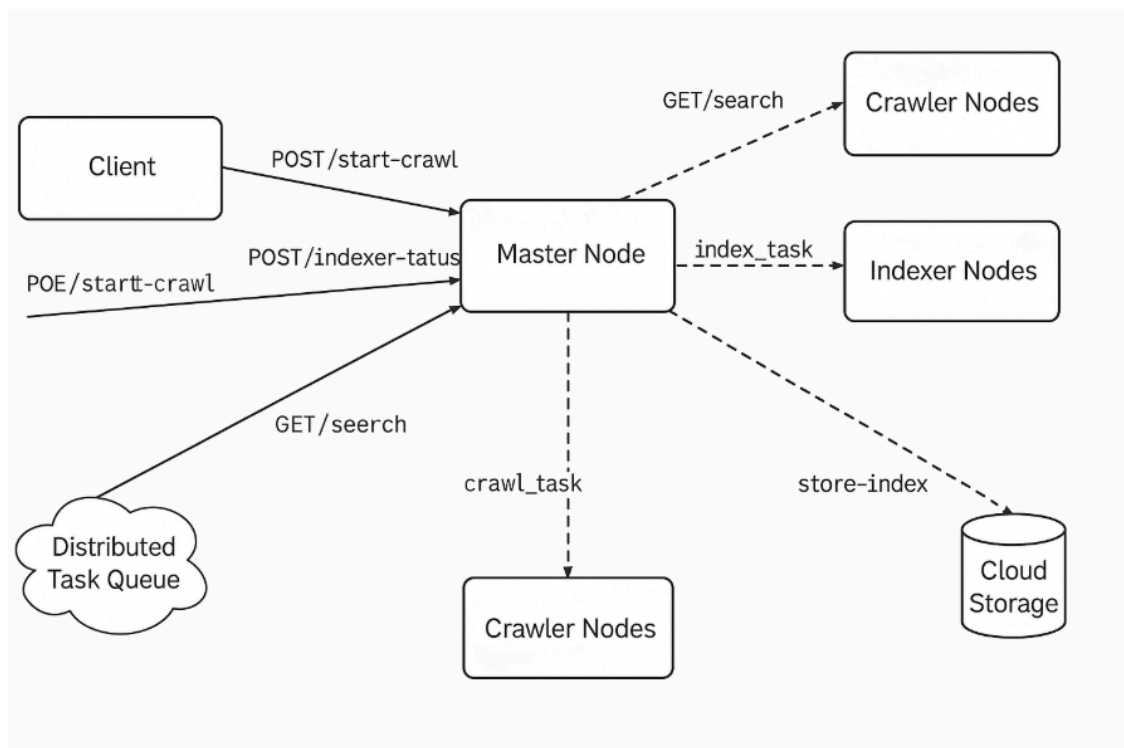
1. **Master Node Fault Tolerance:**
  - Master node redundancy and failover
  - State replication across multiple master instances
  - Leaderless coordination protocols
2. **Indexer Node Fault Tolerance:**
  - Index data replication
  - Distributed indexing with partition tolerance
  - Recovery of partial indices
3. **Network and Infrastructure Resilience:**
  - Multi-region deployment
  - Communication protocol hardening
  - Rate limiting and backoff strategies

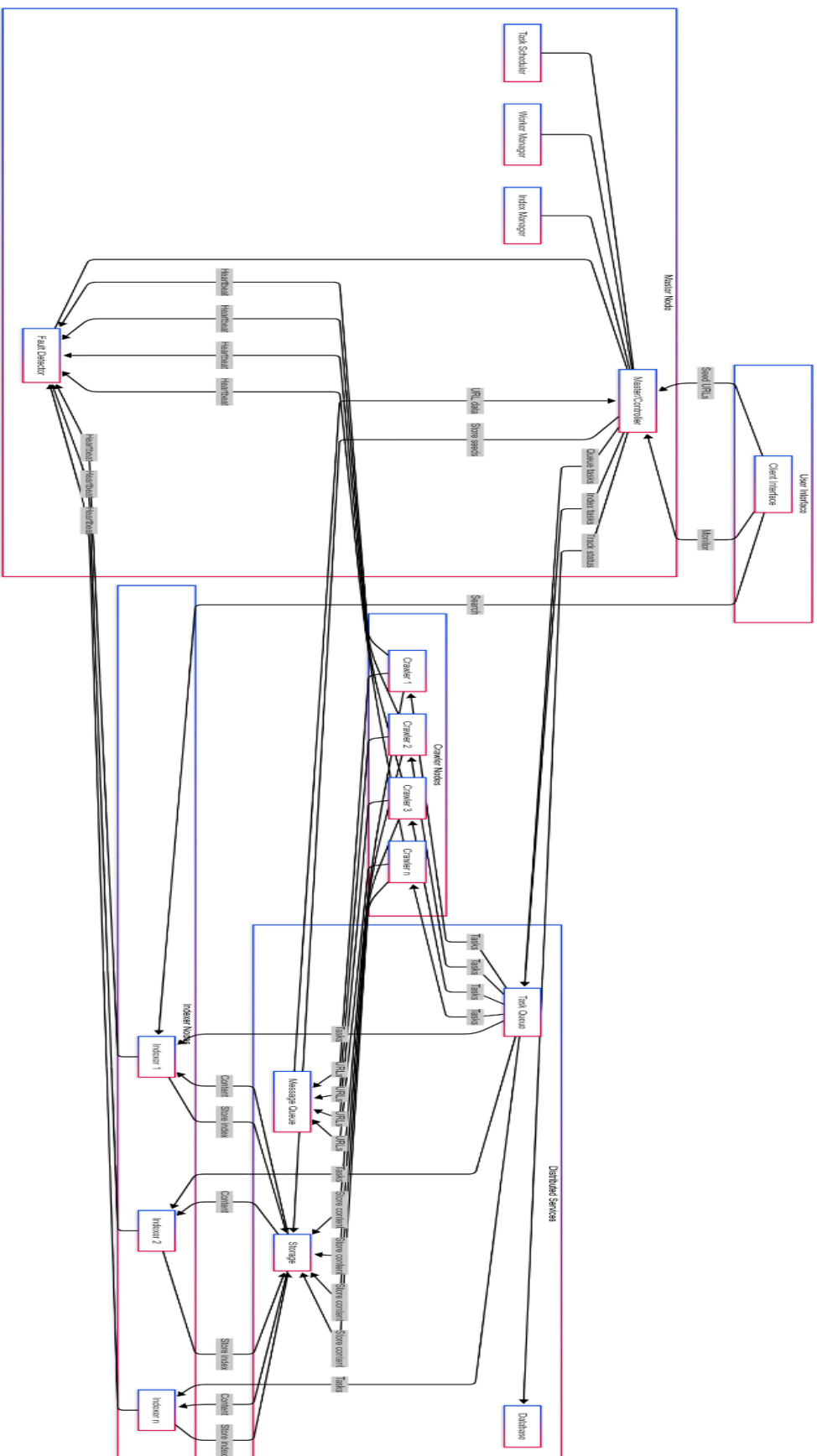
## 3.10 Conclusion

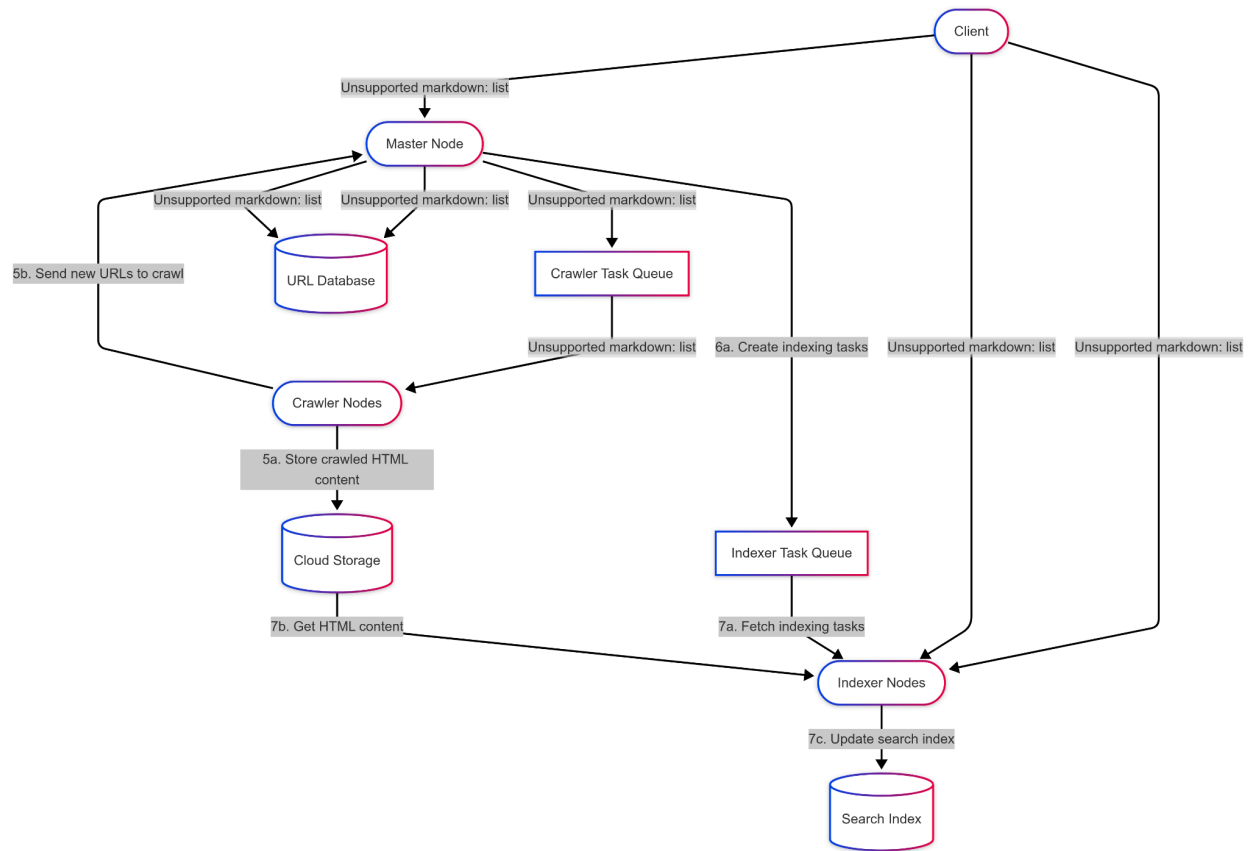
The outlined fault tolerance strategies focus on crawler node failures as our first priority, establishing a foundation for a robust distributed web crawling system. By implementing effective detection, response, and recovery mechanisms, we aim to ensure continuous operation even in the face of individual component failures.

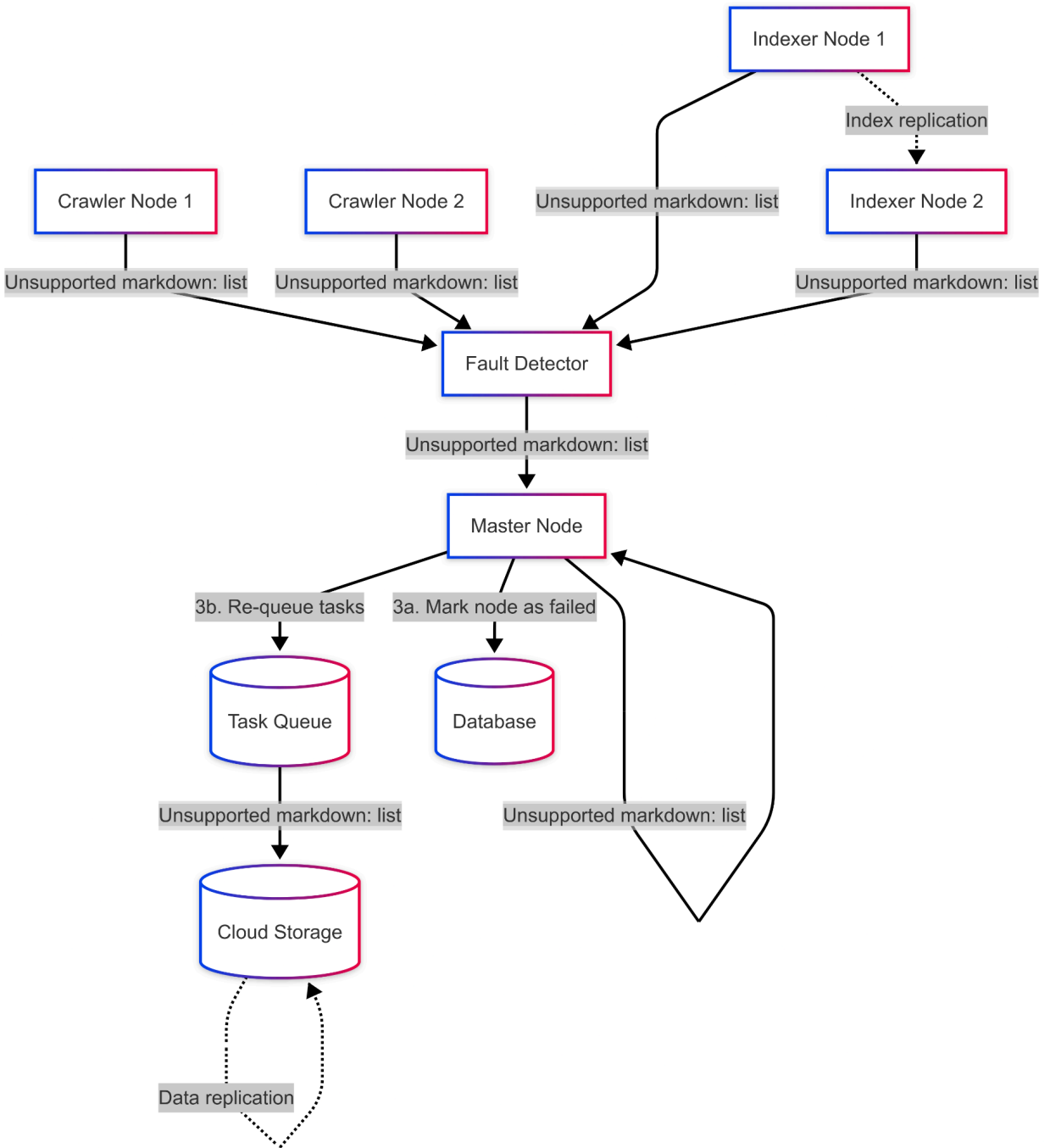
For Phase 1, we will implement the core fault tolerance features while designing the architecture to accommodate future expansions. This approach balances immediate reliability needs with the flexibility required for the system's evolution in subsequent development phases.

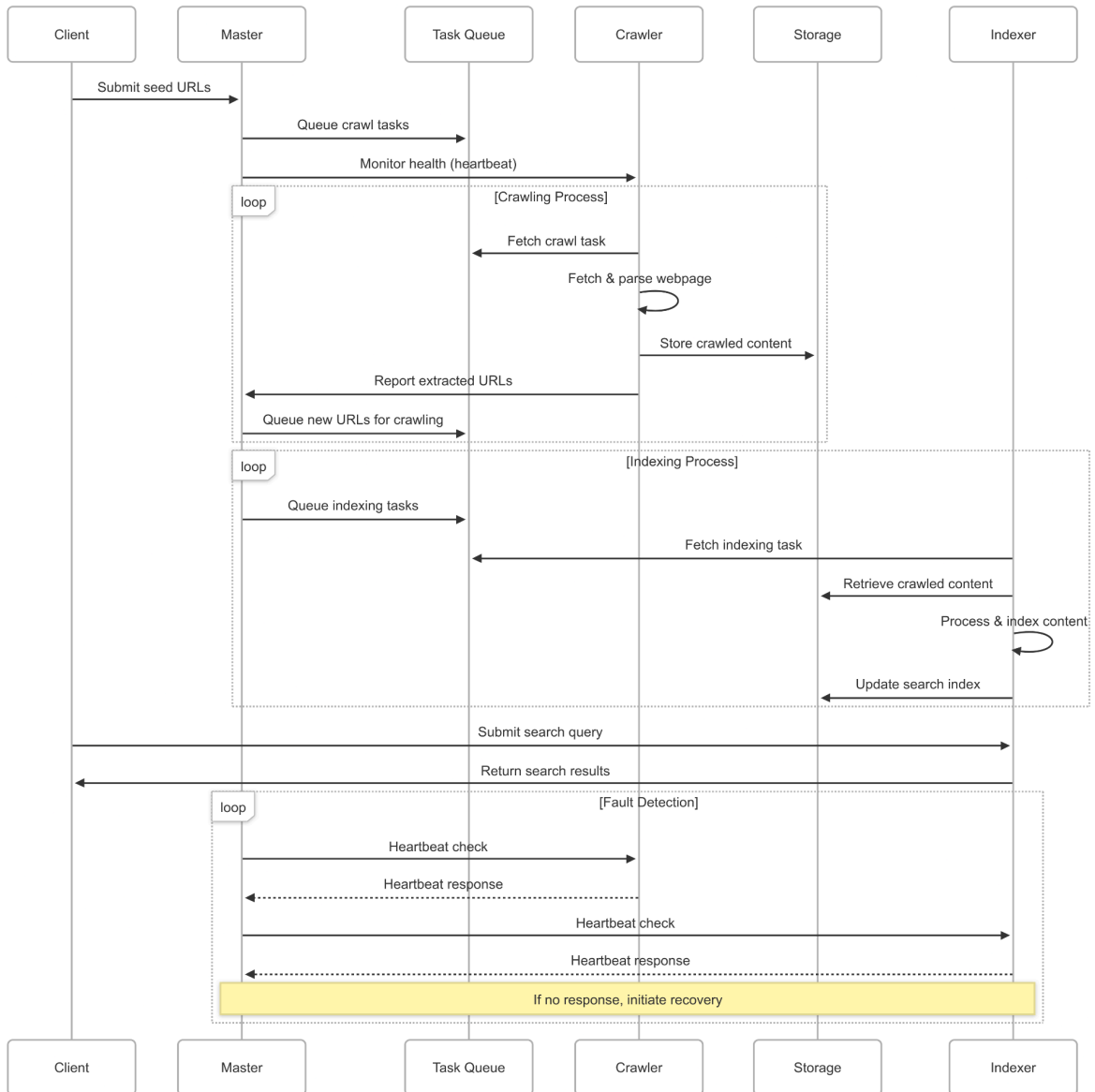
## 4. API diagram













## **5. Conclusion**

These API interfaces form the communication backbone of the distributed web crawling and indexing system. The separation of concerns and use of standardized protocols (REST, message queues) ensures modularity, resilience, and scalability, while also allowing easy testing and monitoring of individual components.