Technology Selection Document

Distributed Web Crawling and Indexing System

Focus: Core Crawling and Basic Indexing Functionality

1 Programming Language: Python

Justification:
Python offers powerful libraries for web crawling, task distribution, and indexing. Its asynchronous capabilities and wide community support make it ideal for rapid development. It is also compatible with AWS services through SDKs like `Boto3`.

2 Web Crawling Libraries: Scrapy, BeautifulSoup, requests

Justification:

- **requests**: Simple and efficient HTTP library for fetching web pages. Ideal for initial crawling tasks.
- **BeautifulSoup**: Excellent for parsing HTML and extracting text and links quickly.
- **Scrapy** (optional for later stages): A full-fledged framework for scalable crawling; not necessary initially, but useful later.

3 Indexing Libraries/Search Engine: Whoosh

Justification:
Whoosh is a lightweight, pure-Python search library, perfect for early stages. It enables quick keyword indexing and search without heavy server overhead. Future scalability can shift toward **Elasticsearch** (AWS OpenSearch Service), but Whoosh allows us to focus on basic functionality first.

4 Distributed Task Queue: Celery with AWS SQS

Justification:

- **Celery** is a well-established distributed task queue system in Python.
- **AWS SQS** (Simple Queue Service) integrates seamlessly with Celery, offering a fully managed, scalable message queue.
- Using AWS SQS removes the need to self-host brokers like RabbitMQ or Redis, reducing operational overhead.

5 Cloud Platform: AWS

Justification:

- AWS provides all required services with granular pricing.
- Familiarity with AWS reduces setup time.

- Services like EC2, SQS, and RDS align perfectly with the project architecture.
- AWS allows easy future scalability.

6 Database: AWS RDS (PostgreSQL)

Justification:
AWS RDS with PostgreSQL offers a fully managed and reliable relational database. It is ideal for storing crawl metadata, seed URLs, and status reports. PostgreSQL supports complex queries efficiently, and relational databases are sufficient for Phase 2 requirements.

7 Architecture Overview

Component	**Technology**	**AWS Service Used**
Crawler Nodes	Python + requests + BeautifulSoup	EC2 Instances
Master Node	Python + Celery Task Producer	EC2 Instance + SQS
Task Queue	Celery Worker/Consumer	AWS SQS
Indexer Node	Python + Whoosh	EC2 Instance
Database (Crawl Metadata)	PostgreSQL	AWS RDS

8 Conclusion

By selecting **Python**, **requests + BeautifulSoup**, **Celery with AWS SQS**, **Whoosh**, and **AWS EC2/RDS**, the project achieves:

- Rapid development
- Easy debugging
- Cloud-native scaling potential
- Low initial complexity while keeping scalability in mind for future phases

This setup ensures that project objectives of the basic distributed crawling and simple indexing can be achieved effectively and reliably.