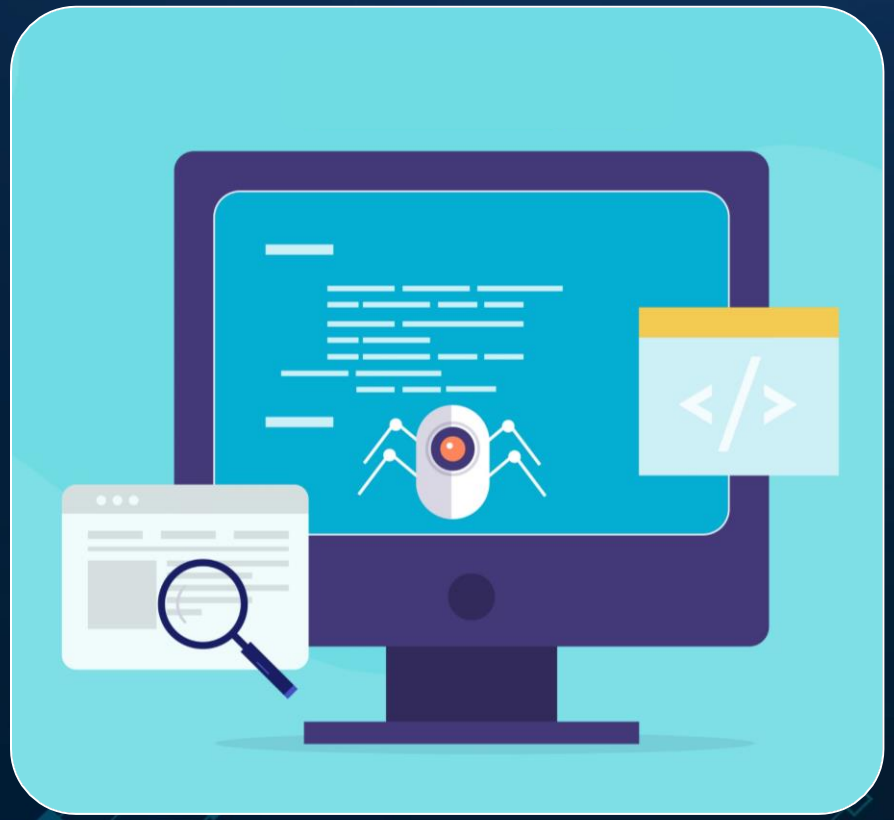
Learn to Design Cloud Architecture

Web Crawler

What is a Web Crawler?

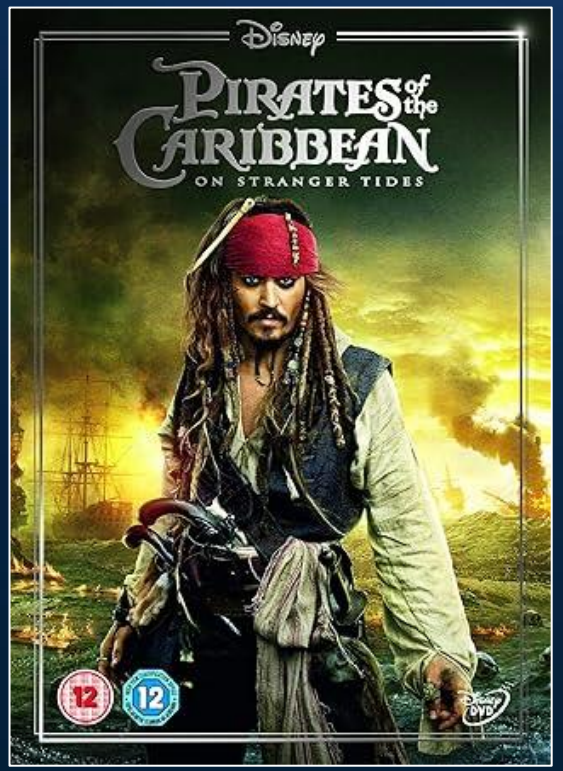
- A web crawler (also known as a spider or bot) is a program designed to systematically browse and retrieve content from websites.

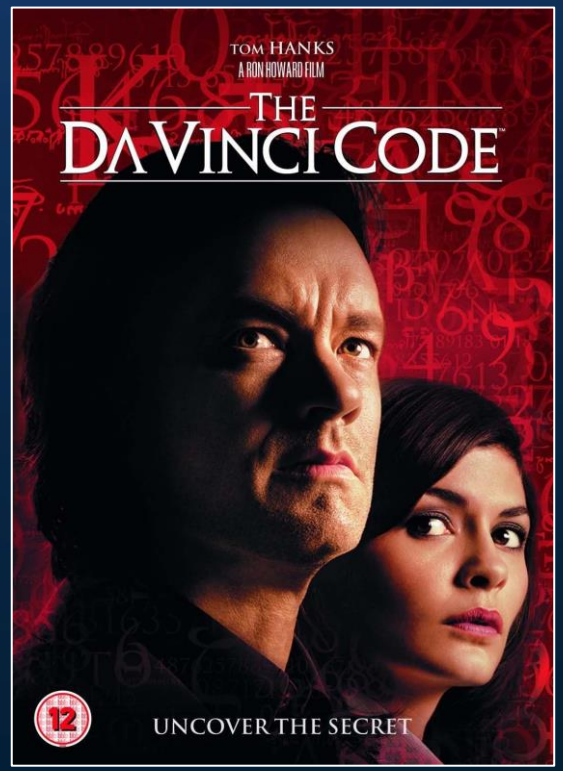


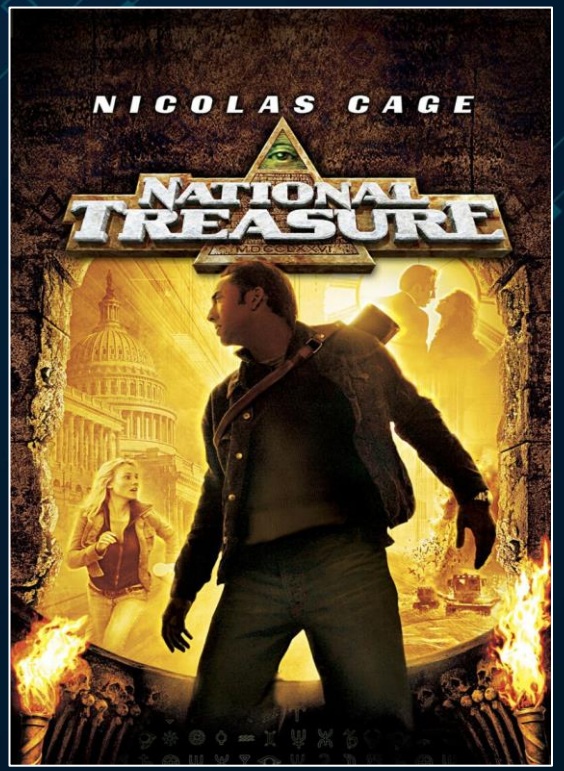
What a Web Crawler does?

- Starts from a set of seed URLs.
- Fetches content from web pages.
- Extracts links from each page.
- Recursively follows those links to fetch additional pages.
- Stores or processes the content based on the goal (e.g., indexing, monitoring, analysis).

Have you watched any treasure hunt movies?







Treasure Hunter vs. Web Crawler





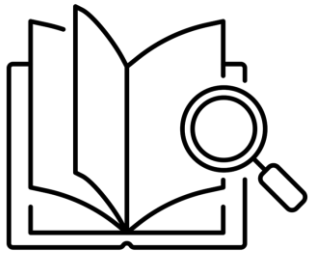

Treasure Hunter	Gets	Map	Finds	Clues	Explores	Hidden Spots	Finds	More Clues	Gets	Treasure






Web Crawler	Gets	Seed URLs	Finds	Linked URLs	Explores	New Pages	Finds	Linked Pages	Gets	Content

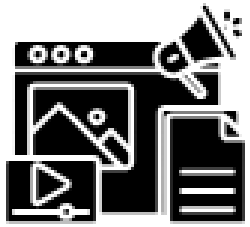
Use Cases of Web Crawlers



Search Indexes

SEO Tools

Data Mining



Content Aggregation

Commonly used Web Crawlers

Popular web crawler:

Googlebot

Bingbot

DuckDuckbot

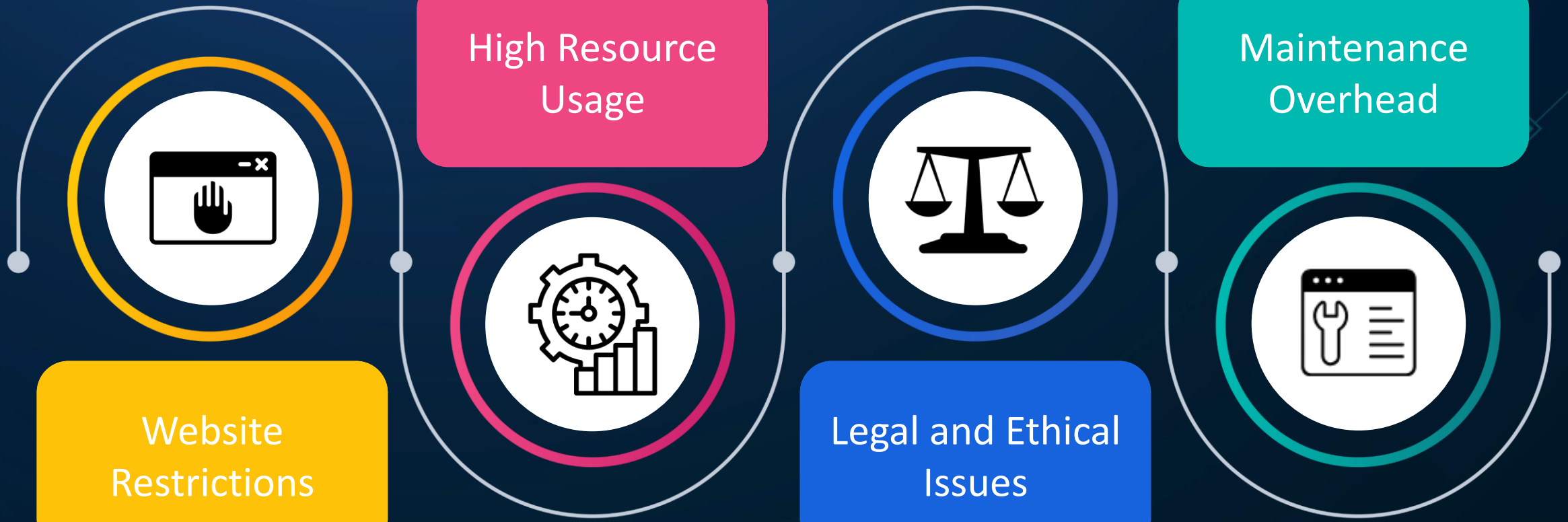
Baiduspider

Some open source crawler:

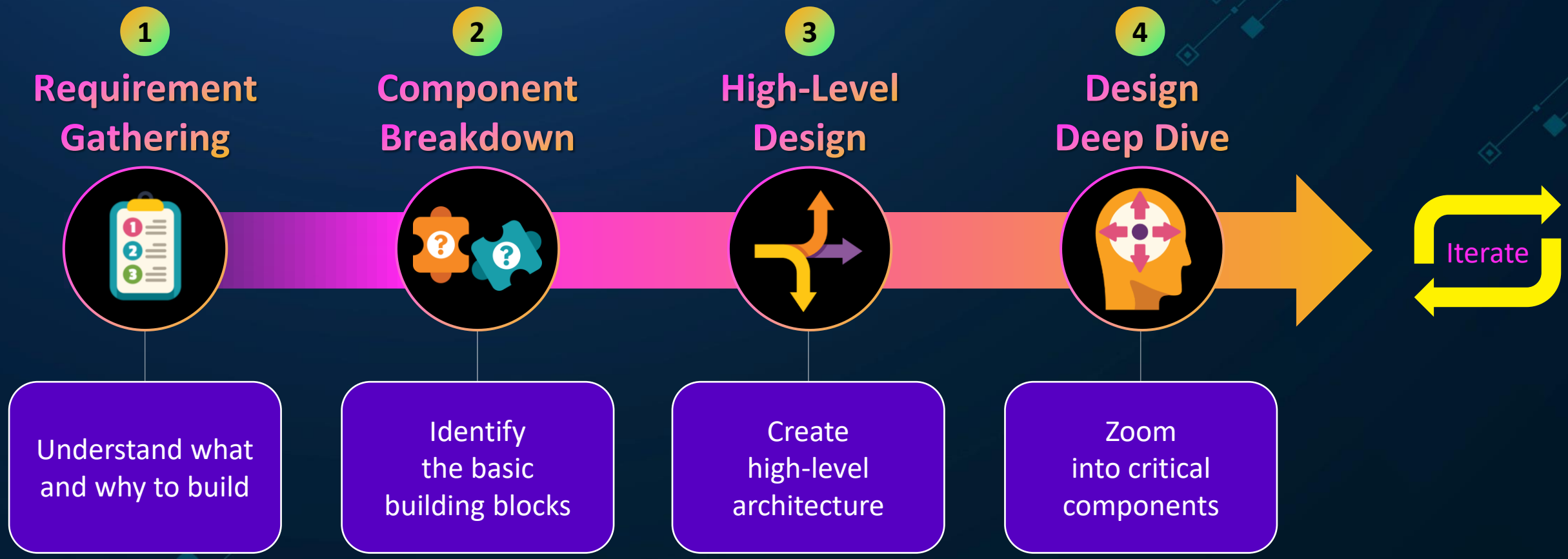
Scrapy

Crawler4J

Challenges of Web Crawlers



Designing a System – A simple framework



Requirement Gathering

Expectations from a Web Crawler

- Functional Requirement (What the systems should do?)
  - Accept one or more seed URLs to start crawling.
  - Extract and follow links recursively with configurable depth.
  - Store text data in JSON, CSV, database, or cloud storage.
  - Respect website rules via **robots.txt** and meta directives.

- Nice to have
  - Detect and skip duplicate URLs or content.
  - Implement rate limiting/throttling to avoid overloading sites.
  - Handle errors (e.g., retries, logging, broken pages).
  - Support scheduling or recurring crawls for freshness.

Non-functional requirements

- Performance
  - The crawler shall fetch and process a minimum of 100 web pages per minute
- Scalability
  - The crawler must be able to handle up to 1,000,000 URLs per crawl session.
- Security
  - The crawler must respect robots.txt directives on all domains.
- Reliability
  - The system shall have at least 99.9% uptime over a 30-day period.
- Monitoring & Logging
  - The system shall log 100% of crawl jobs with metadata (e.g., start time, end time, pages crawled, error count).

Out of scope

- Authentication/Authorization handling
- Dynamic content rendering
- Form submission & interaction
- Media files
- Compliance & legal enforcement
- Non-English text
- Use of blacklist of domains and keywords
- Domain classification APIs
- Other ….

Foundational Knowledge

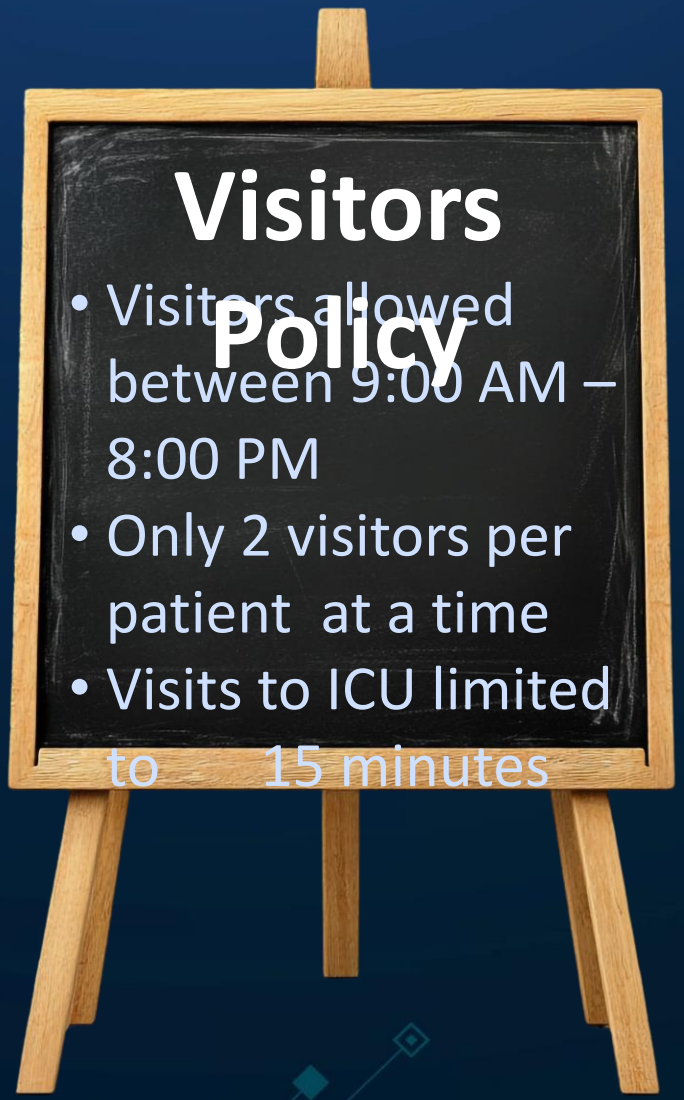
Politeness for web crawlers

- Politeness in web crawlers refers to how respectfully a crawler behaves toward the websites it visits, ensuring it doesn't overload or harm servers.
- Key Aspects of Crawler Politeness:

Aspect	Description
Crawl Delay	Wait time between requests to the same server (e.g., 1 request every 10 sec)
robots.txt Rules	Respecting allowed/disallowed paths specified by the website
Rate Limiting	Limiting the number of concurrent requests to avoid server overload
Retry Strategy	Backing off after failures rather than hammering the server repeatedly
User-Agent Header	Clearly identifying the crawler in requests

An analogy

Hospital



ROBOTS.TXT

Set of instructions to web crawlers (robots)

Website

Robots.txt

- A robots.txt file is a plain text document located in a website's root directory, serving as a set of instructions to web crawlers.
- It's not an official standard set by any standards organization, although all major search engines adhere to it.

Sample Robots.txt

```
User-agent: *
Disallow: /private/
Disallow: /admin/
Allow: /public/

Crawl-delay: 10

Sitemap: https://example.com/sitemap.xml
```

https://www.google.com/robots.txt

Meta Directives

- Meta directives are special HTML <meta> tags placed inside the <head> section of a web page that give instructions to web crawlers on how to index or interact with the page.
- Common Meta Directives for Crawlers:

Directive	What It Tells the Crawler?
noindex	Don’t show this page in search engine results.
nofollow	Don’t follow links on this page.
index	Allow indexing of this page (default behavior).
follow	Follow links on this page (default behavior).
noarchive	Don’t store a cached copy of this page.
noimageindex	Don’t index images on this page.

Example Usage in HTML

```
<head>
  <meta name="robots" content="noindex, nofollow">
</head>
```

- This tells all crawlers not to index the page and not to follow any links on it.

Component Breakdown

Component Breakdown

Seed URLs

Initial websites to start crawling

URL Queue

Holds pending URLs to visit.

Worker Nodes

Fetch and process web pages

Coordinator

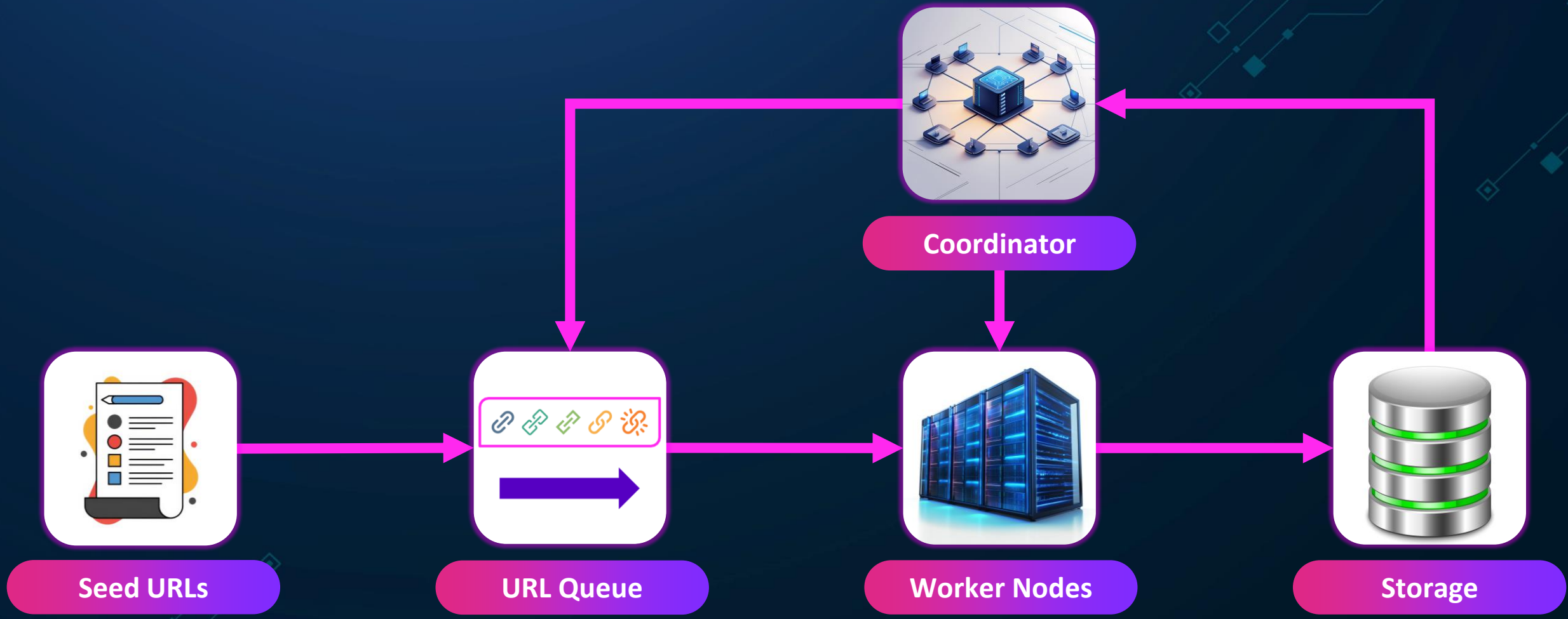
Controls task assignment and scheduling

Storage

Saves fetched data and metadata

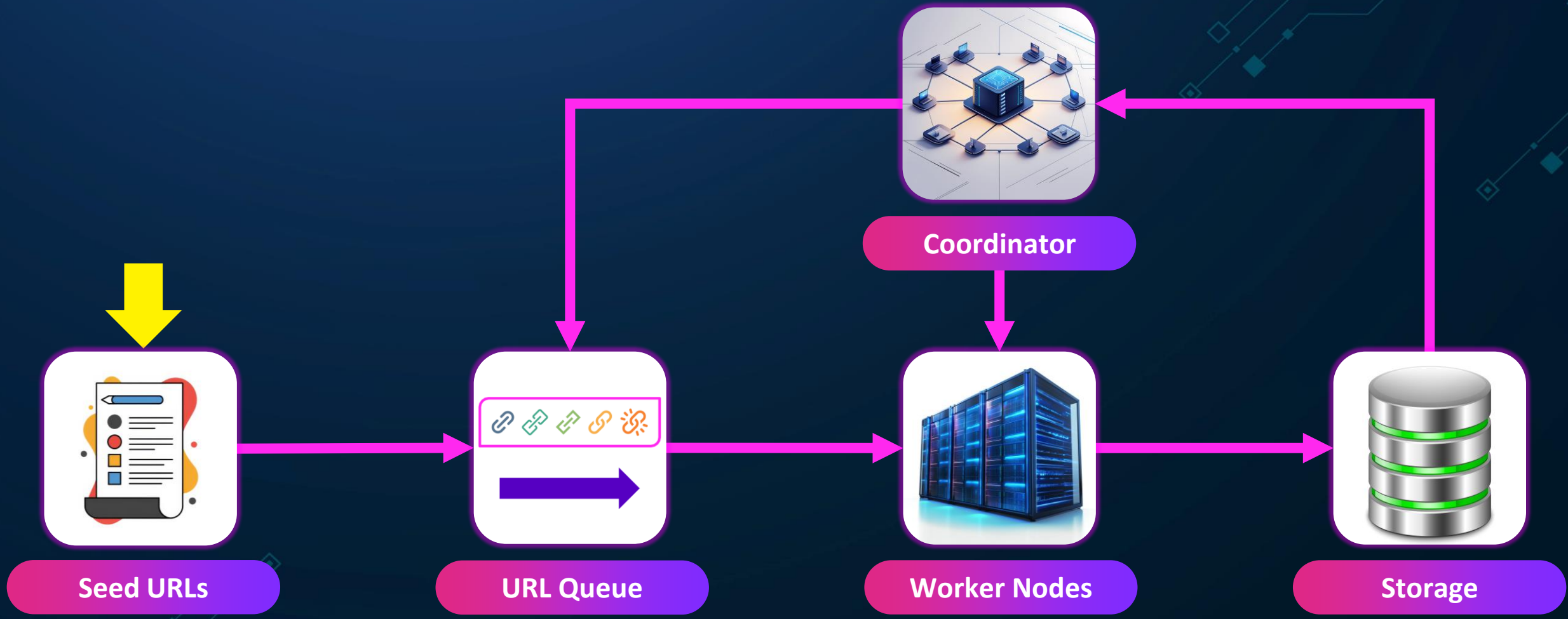
High-Level Design

High-level architecture



Design Deep Dive

Deep Dive – Seed URLs



Seed URLs

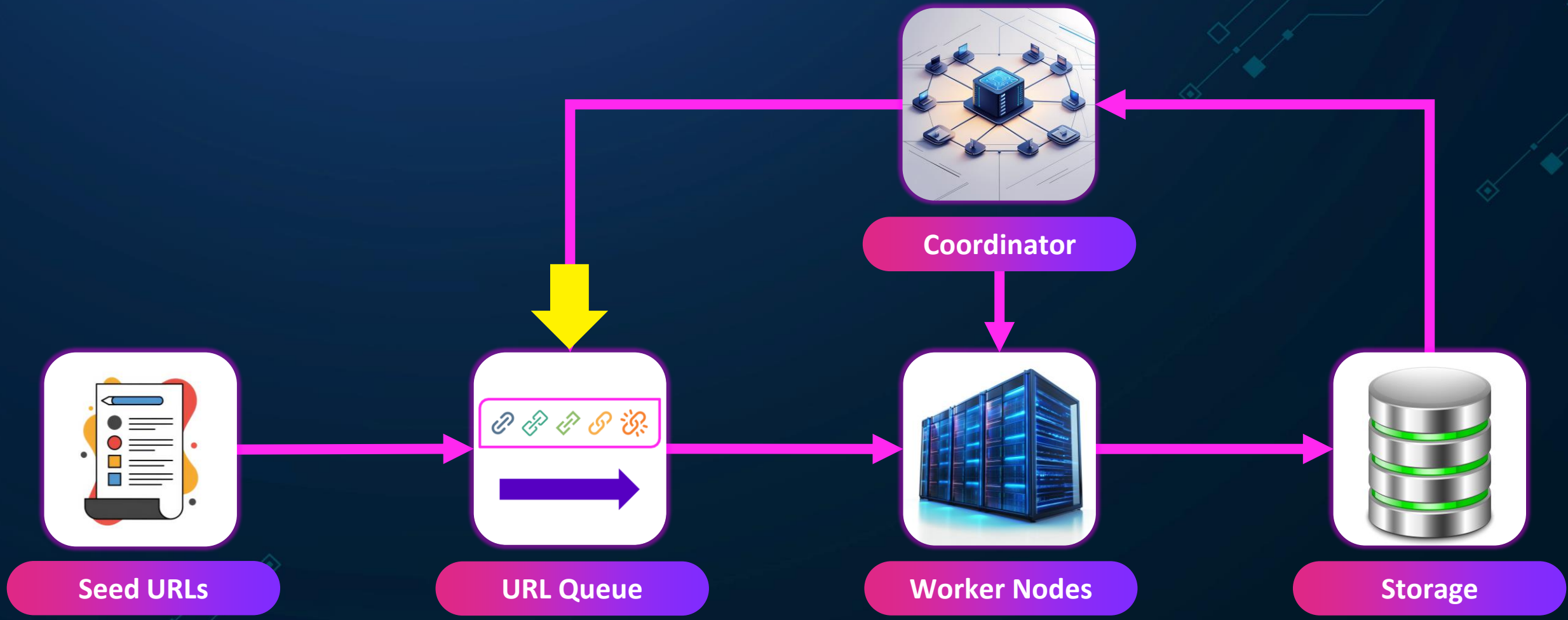
Factor	Manual Input	Top Sites Lists (e.g., Tranco, Similarweb)	Search Engine Queries (via APIs)
What is it?	Manually curated list of URLs	Public rankings of popular domains	Programmatic querying of search engines (e.g., Bing, Google)
Control	High	Low	High
Scalability	Low	High	Limited by Quota
Freshness	Depends on upkeep	Updated Regularly	Very fresh
Complexity	Simple	Easy to use	Medium (API Integration)
Best for	Small-scale, domain-specific crawlers, or compliance-focused systems.	Large-scale general-purpose crawlers or academic/research crawling.	Topic-based crawling (e.g., “climate change research sites”)

Example - Top Sites Lists

- https://tranco-list.eu/
  - Free and downloadable

- High Level Steps
  1. Download or Access the List
  2. Pre-process the Domains into Valid URLs - https://{domain}
     - Optional - Remove .gov, .edu, or non-English domains, Include only .com, .org, etc.
  3. Store into a file / Feed into your crawling pipeline
  4. Crawl With Respect to Robots.txt

Deep Dive – URL Queue (aka – URL Frontier)

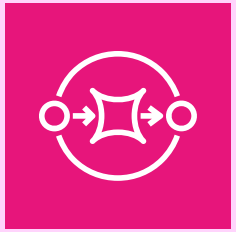
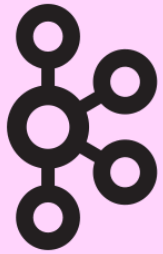


Why we need a Queue?

- Decoupling
  - Separate URL discovery from processing
- Scalability
  - Supports parallel crawling at scale
- Reliability
  - Retry on failure, avoid duplicates
- Politeness
  - Enables delay handling and domain fairness
- Control
  - Prioritize and throttle crawl behaviour

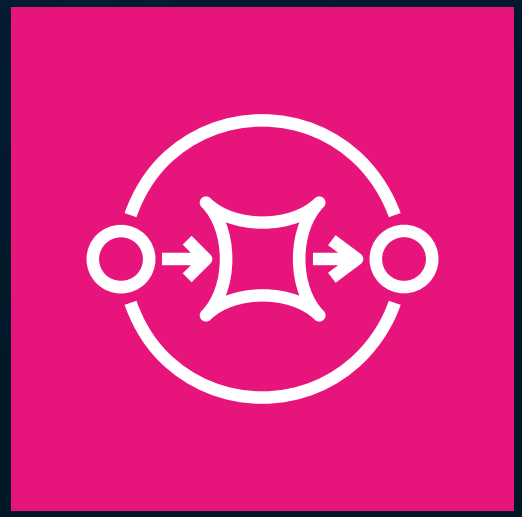
URL Queue

How to implement a Queue?

Feature	 Amazon SQS	 Apache Kafka	 Redis
Speed	Moderate	Fast	Very fast
Scalability	High	High	Medium
Persistence	Strong	Strong	Optional
Priority Support	Limited (FIFO only)	No	Yes
Retry Support	Native	Native via offsets	Manual
Ease of Use	Very Easy	Complex	Easy
Best For	Simple managed queues	High-volume pipelines	Fast, Flexible queuing

Amazon SQS – Best Practices

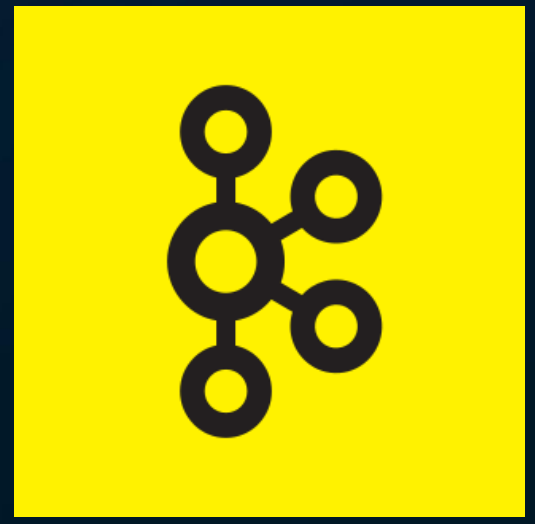
- Prefer Standard Queue, use FIFO Queues only if necessary
- Use Multiple Queues for Control (Split by domain, content type, or priority)
- Crawl Politeness & Delay Handling
- Use Visibility Timeout to Manage Concurrency
- Use a De-Duplication layer



Apache Kafka – Best Practices

- Use Separate Topics by Function or Priority
- Use Partitions to Scale Consumers
- Implement custom delay logic (use a scheduling layer)
- Use a De-Duplication layer

Amazon Managed Streaming for Apache Kafka

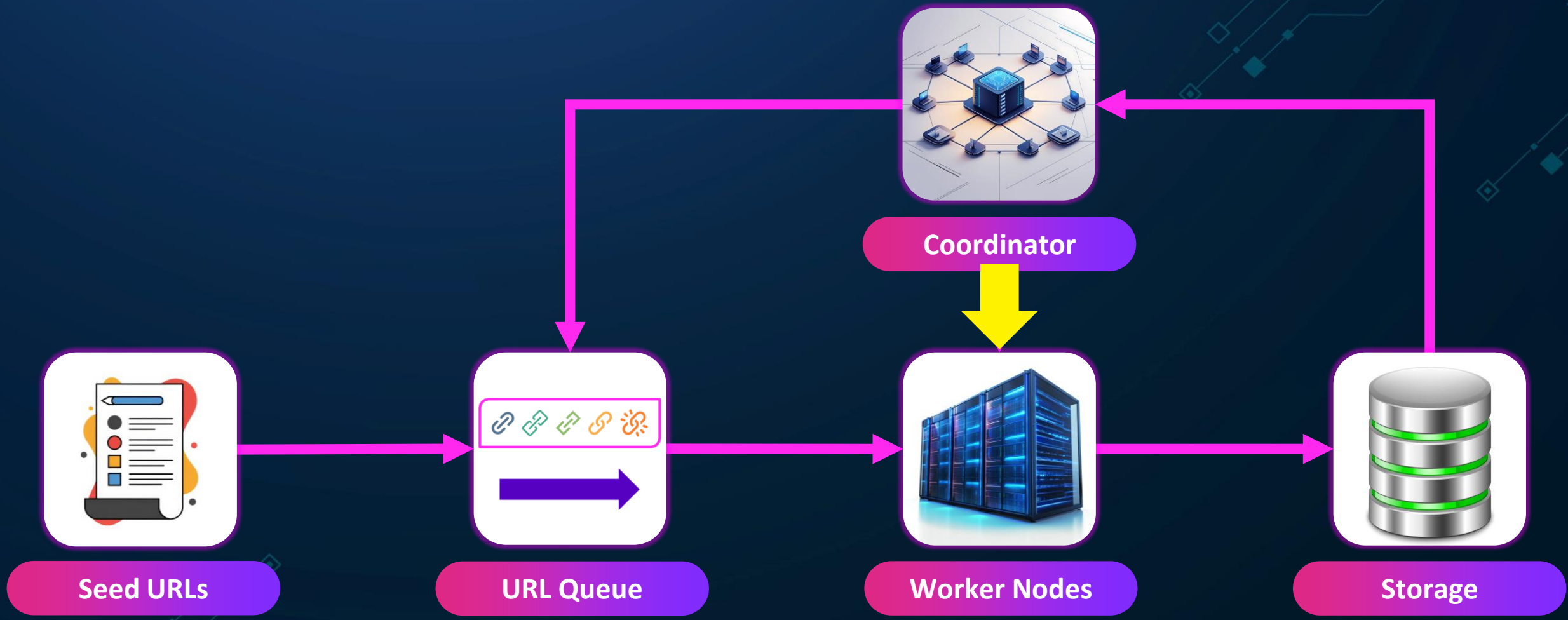


Redis – Best Practices

- Use Lists
- Use Sorted Sets for delay scheduling
- Implement a Per-domain rate limiting
- Use a De-Duplication layer
- Use Redis Replication for fault tolerance

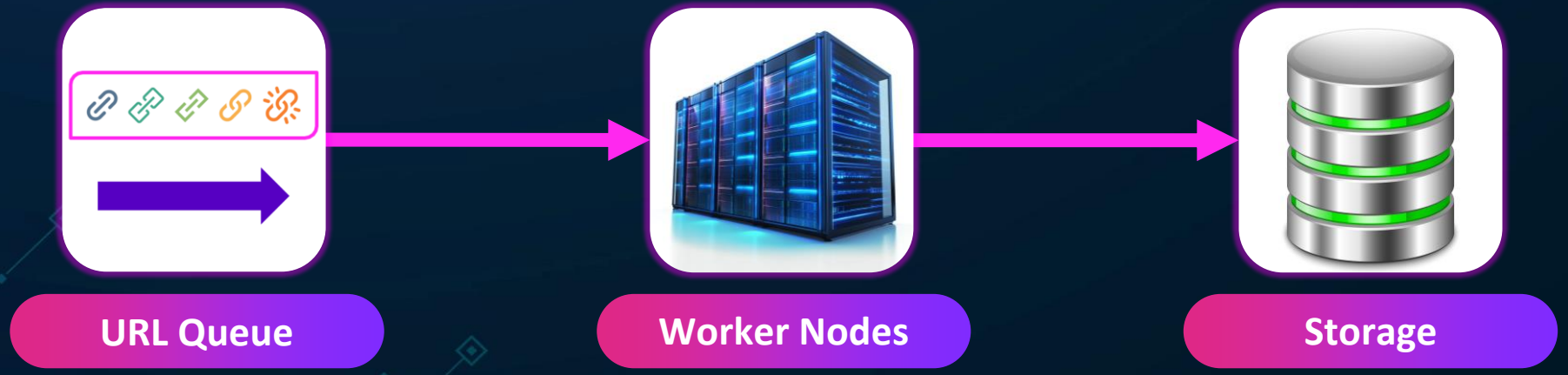
Amazon ElastiCache For Redis

Deep Dive – Worker Nodes



Core responsibility of a worker node

1. Poll a message from the queue (e.g., next URL to crawl)
2. Crawl the URL (e.g., GET request) while being polite (robotos.txt)
3. Parse and extract relevant data or links
4. Send results to storage (e.g., metadata, data)
5. Log status, mark URL as processed, and delete from queue
6. Repeat



Some considerations for worker nodes

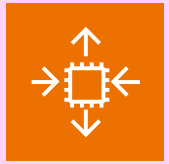
Politeness Handling

Fault tolerance

Scaling concurrency

Cost Control

Implementing Worker Nodes

	 **Amazon EC2 Auto Scaling Groups**	 **AWS Fargate (with ECS or EKS)**	 **AWS Lambda**
Pros	• Full control over OS, libraries, and tooling • Can run stateful or resource-heavy crawlers • Auto Scaling Group can scale based on SQS depth • Cost-optimized with Spot Instances	• Serverless compute for containers • Scales automatically based on demand or queue depth • Native integration with EventBridge, SQS, CloudWatch • Supports resource limits per task (CPU/memory)	• Serverless and fully managed • Instant scale-out for high concurrency • Great for URL pre-checks, status fetching, or metadata scraping • Integrated with SQS, DynamoDB, EventBridge

Cons	• Higher operational overhead. You must manage:
	  • OS patches
	  • Scaling logic
	  • Failures & reboots
	  • Longer boot time
		• Cold start time for task launch (~30–60 sec)
		• Limited by runtime duration (up to 120 hours per task)
		• Slightly higher cost than EC2 if running 24/7
			• Short execution time (max 15 minutes)
			• Not suitable for full-page crawls or large content downloads
			• Limited CPU/memory options (max 10 GB RAM, 6 vCPUs)

Best for	• Long-running or resource-heavy crawlers	• Scalable, stateless containerized workers	• Lightweight URL checking / bursty workloads

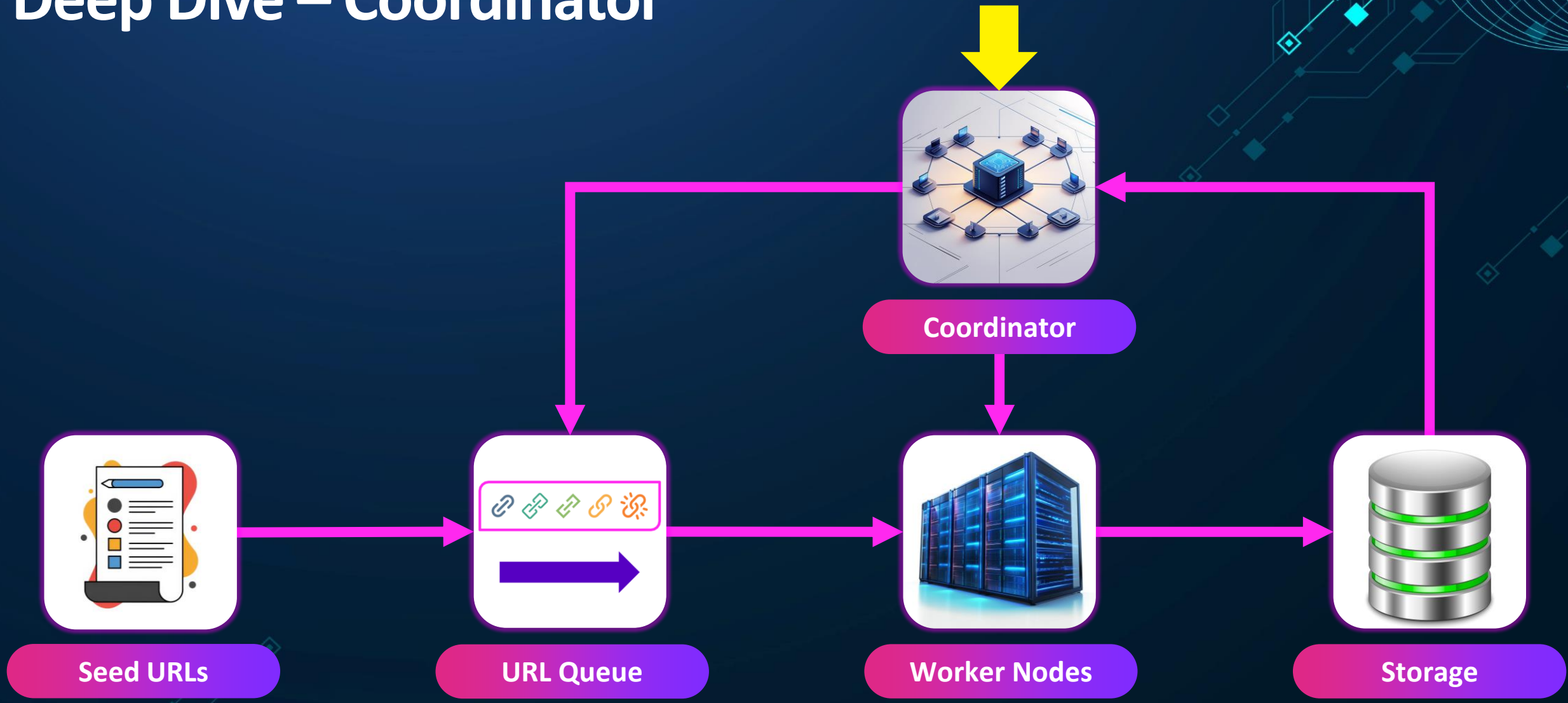
Which crawler program to use?

Tool	Language	Scale	Strength	Best for
Scrapy	Python	Small–Medium	• Built-in support for data pipelines, retries, throttling, etc. • Great community and plugins	Data extraction and scraping workflows, especially if you need to write custom logic quickly
Apache Nutch	Java	Very Large	• Extremely scalable • Pluggable architecture (e.g., custom parsers, indexers) • Integrates well with Elasticsearch, Solr	Enterprise-level crawling with big data workflows
Crawler4j	Java	Medium	• Simple API, easy to set up • Multi-threaded crawling out of the box • Good for academic or smaller-scale industrial crawlers	Mid-sized custom crawlers without the overhead of Hadoop
StormCrawler	Java	Large	• Streaming architecture: no batch processing delays • Good for continuous crawling/ indexing • Scalable and fast with low-latency	Real-time, event-driven web indexing or news aggregation

Choose based on your goals

- For ease of use and rapid development: go with Scrapy.
- For scalable, Hadoop-based crawling: go with Apache Nutch.
- For lightweight Java crawling: use Crawler4j.
- For real-time data ingestion: choose StormCrawler.

Deep Dive – Coordinator



Key Responsibilities of the Coordinator

- **Populating the URL Queue**
  - Adds newly discovered, valid URLs from crawled pages to the queue.
- **Task Assignment to Worker Nodes**
  - Distributes URLs to available workers for parallel processing.
- **Respecting Crawl Rules**
  - Ensures compliance with robots.txt and domain-specific crawl delays.
- **Duplicate URL Handling**
  - Avoids re-crawling by checking and tracking previously visited URLs.
- **Failure Handling and Retries**
  - Detects worker failures and reassigns unfinished tasks to other nodes.
- **Storage Management**
  - Directs crawled content and metadata to the appropriate storage system.

Coordinator

Core characteristics of the coordinator system

How to implement coordinator system?

- Build it from scratch
  - Teams with strong engineering resources and unique, tightly controlled requirements

- Use existing solutions
  - When you need to move fast, reduce risk, and rely on battle-tested solutions

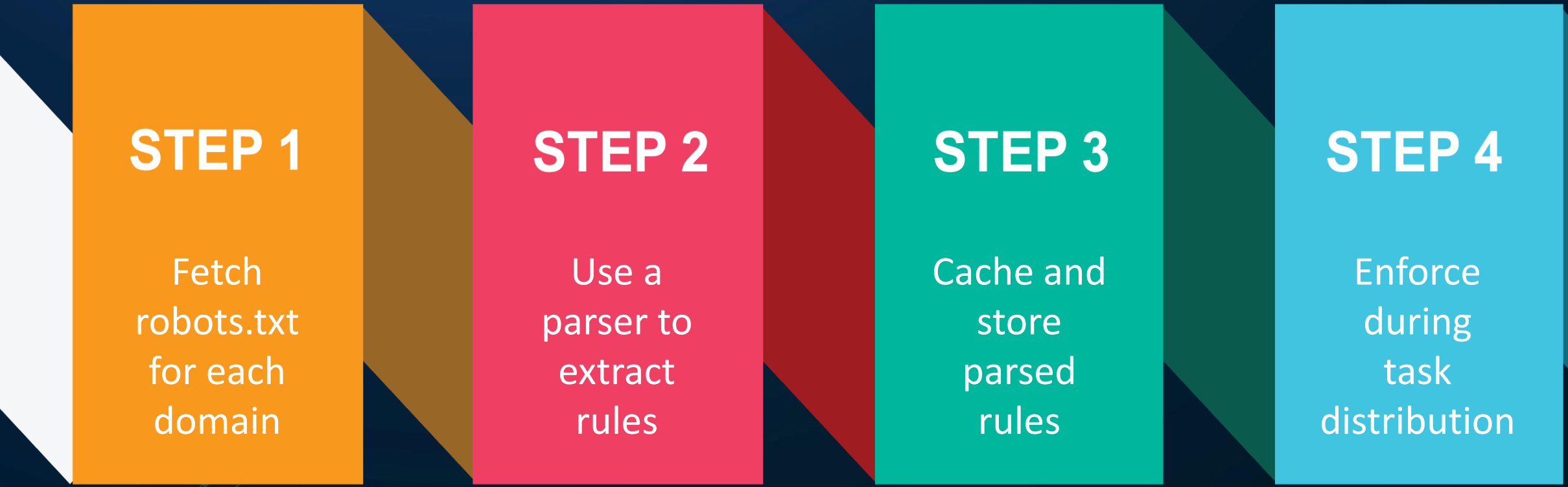
Aspect	Apache Airflow	AWS Step Function
Scalability	Horizontally scalable via Celery or KubernetesExecutor; good for task parallelism.	Automatically scales with AWS Lambda and integrated services; handles thousands of concurrent executions.
Reliability	Task retries, alerting, and backfills available; depends on external components (e.g., DB, workers).	High reliability with built-in retries, timeouts, and state tracking; managed by AWS.
Extensibility	Highly extensible via Python operators, custom plugins, and DAG logic.	Extensible through integration with custom Lambda functions and service connectors.
Pros	- Open-source and flexible - Python-based - Strong community	- Fully managed - Serverless - Native integration with AWS services
Cons	- Requires infra setup and scaling knowledge - UI can lag at scale	- Limited to AWS ecosystem - Complex logic can become hard to visualize/debug

URL Deduplication Techniques

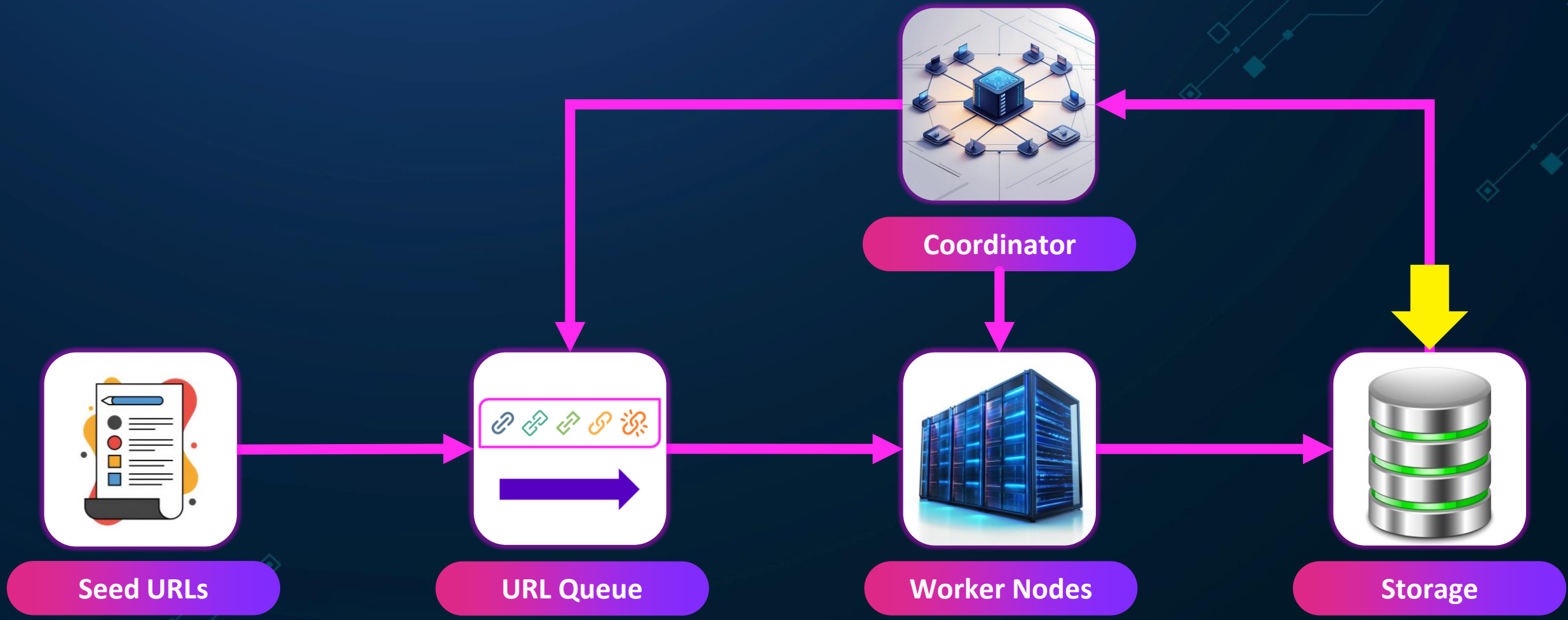
- URL Fingerprinting (Hashing)
  - Generate a hash (e.g., SHA-256) of each normalized URL and store it in a hash set or DB.
- Bloom Filter
  - Use a memory-efficient probabilistic structure for large-scale URL deduplication (may allow false positives).
- Persistent Store (e.g., DB)
  - Maintain a durable record of crawled URLs in a SQL/NoSQL database for long-term tracking.
- Distributed Cache (e.g., Redis)
  - Store visited URLs or hashes in a fast-access store with TTL if temporary crawling is allowed.

Respecting Crawl Rules

- Here's how it can be implemented in a coordinator:



Deep Dive



What we should store?

What can be stored as part of Data?

Data Field	Description	Purpose
Raw HTML Content	Full source HTML of the crawled page	Enables re-parsing, auditing, or extracting new fields later
Parsed Text Content	Clean plain text from the HTML	Search indexing, NLP tasks like summarization or classification
Page Title	Text inside the <title> tag	Useful for summarization, indexing, and page labeling
Meta Tags	<meta> tag content like description, keywords	Helps with SEO analysis or topic identification
Main Content	Extracted central body (e.g., article text)	Focused analysis, sentiment detection, or summarization
Outbound Links	All URLs the page links to	Supports link graph building, crawl expansion, or relevance scoring

And many more …

What can be stored as part of Metadata?

Metadata Field	Description	Purpose
URL	The unique address of the crawled page	Identifies the page and ensures uniqueness
Canonical URL	Preferred URL declared via <link rel="canonical">	Prevents duplication and consolidates link signals
Crawl Status	Result of the crawl (e.g., success, timeout, 404)	Used for monitoring, error handling, and retries
HTTP Status Code	Response code (e.g., 200, 301, 404)	Indicates page availability and redirect behavior
Crawl Depth	Number of hops from the seed URL	Controls crawl scope and prioritization
Last Modified	Value from HTTP header (if available)	Helps decide whether re-crawling is necessary

And many more …

Why Separate Storage?

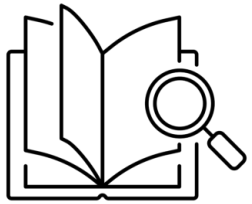
- Performance
  - Keeps fast-access metadata operations separate from large-volume content storage.
- Scalability
  - Metadata can be distributed in low-latency databases, while data can be archived in scalable object storage.
- Cost Efficiency
  - Metadata often needs frequent access (costlier), while data can be stored in cheaper long-term storage.
- Query Optimization
  - Metadata storage supports structured querying, while raw data is usually accessed less frequently.

Where we should store it?

- Object Storage
  - (e.g., AWS S3, Azure Blob Storage)
- Document Store
  - (e.g., MongoDB, Elasticsearch)
- Distributed File Systems
  - (e.g., HDFS)

- Relational Database
  - (e.g., PostgreSQL, MySQL)
- Key-Value Store
  - (e.g., Redis, DynamoDB)
- Search Engine
  - (e.g., Elasticsearch)

Choose storage based on your use case



Search Indexes

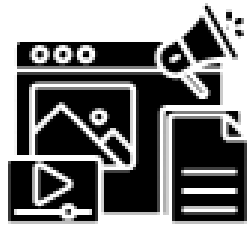
- Object Storage (e.g., AWS S3)
- Document Store (e.g., Elasticsearch)

SEO Tools

- Relational DB (e.g., PostgreSQL/MySQL)
- Key-Value Store (e.g., Redis)

Data Mining

- Object Storage (e.g., AWS S3)
- Data Lake (e.g., AWS Lake Formation, Delta Lake)



Content Aggregation

- NoSQL DB (e.g., MongoDB, DynamoDB)
- Object Storage (e.g., AWS S3)

How much storage capacity we may need for data?

- Approximation:
  - Total website ≈ 1.2 billion
  - Active websites ≈ 200 million
  - Average pages per active website ≈ 250
  - Average text size per page ≈ 7 KB

- Total text data:
  - Total pages=200,000,000 websites×250 pages/website=50,000,000,000 pages
  - Total text size=50,000,000,000 pages×7 KB=350,000,000,000 KB
  - Convert to Terabytes ≈326 TB

Putting it all together

- Amazon Managed Workflows for Apache Airflow (MWAA)
- AWS Step Function

Coordinator

Seed URLs

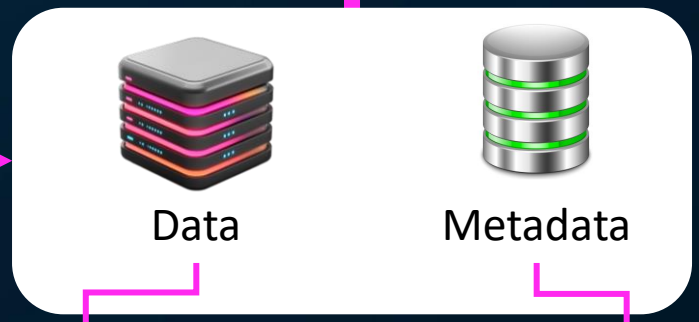
- Manual Input
- Top Sites Lists
- Search Engine Queries

URL Queue

- Amazon SQS
- Apache Kafka
- Redis

Worker Nodes

- Amazon EC2
- AWS Fargate
- AWS Lambda

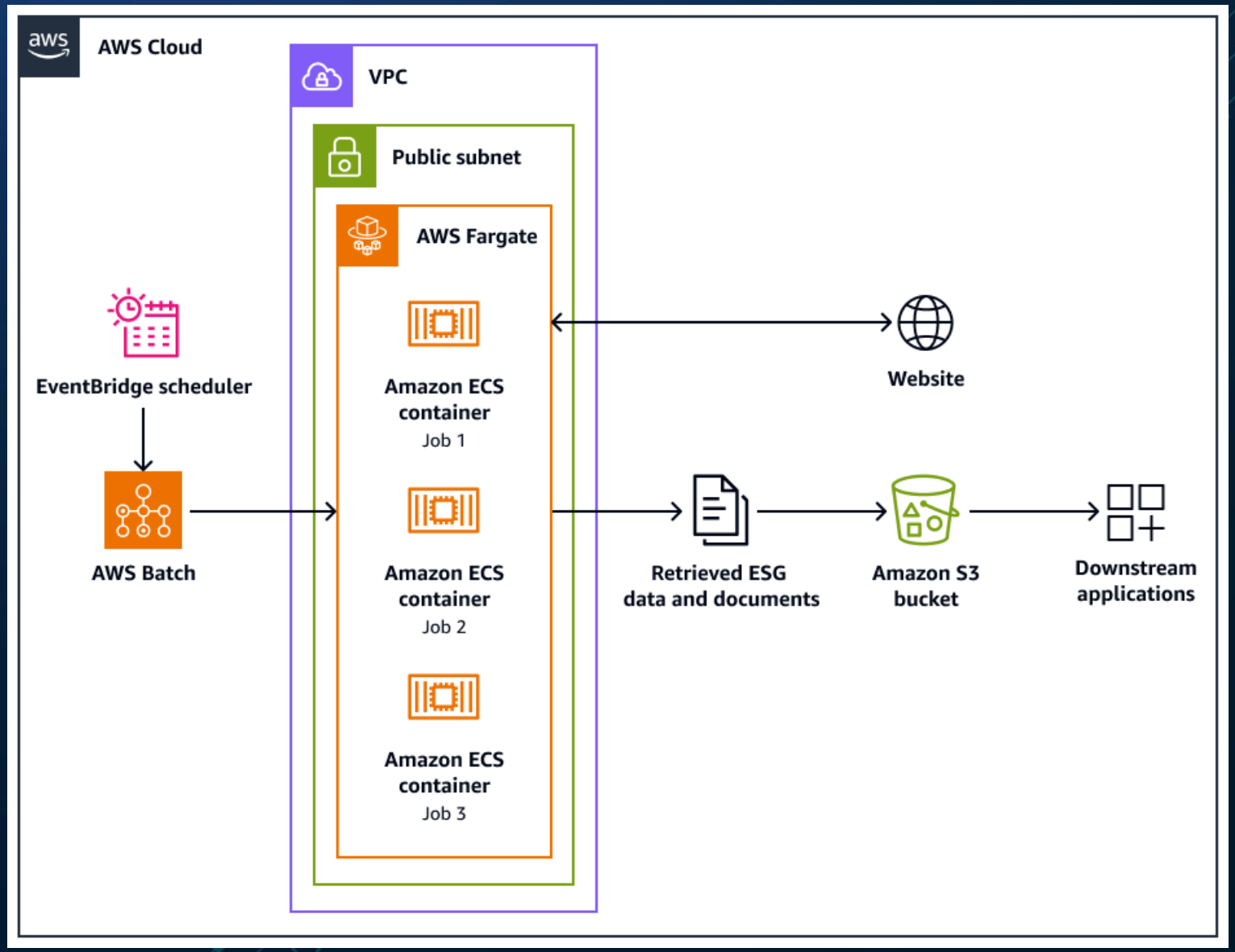


Storage

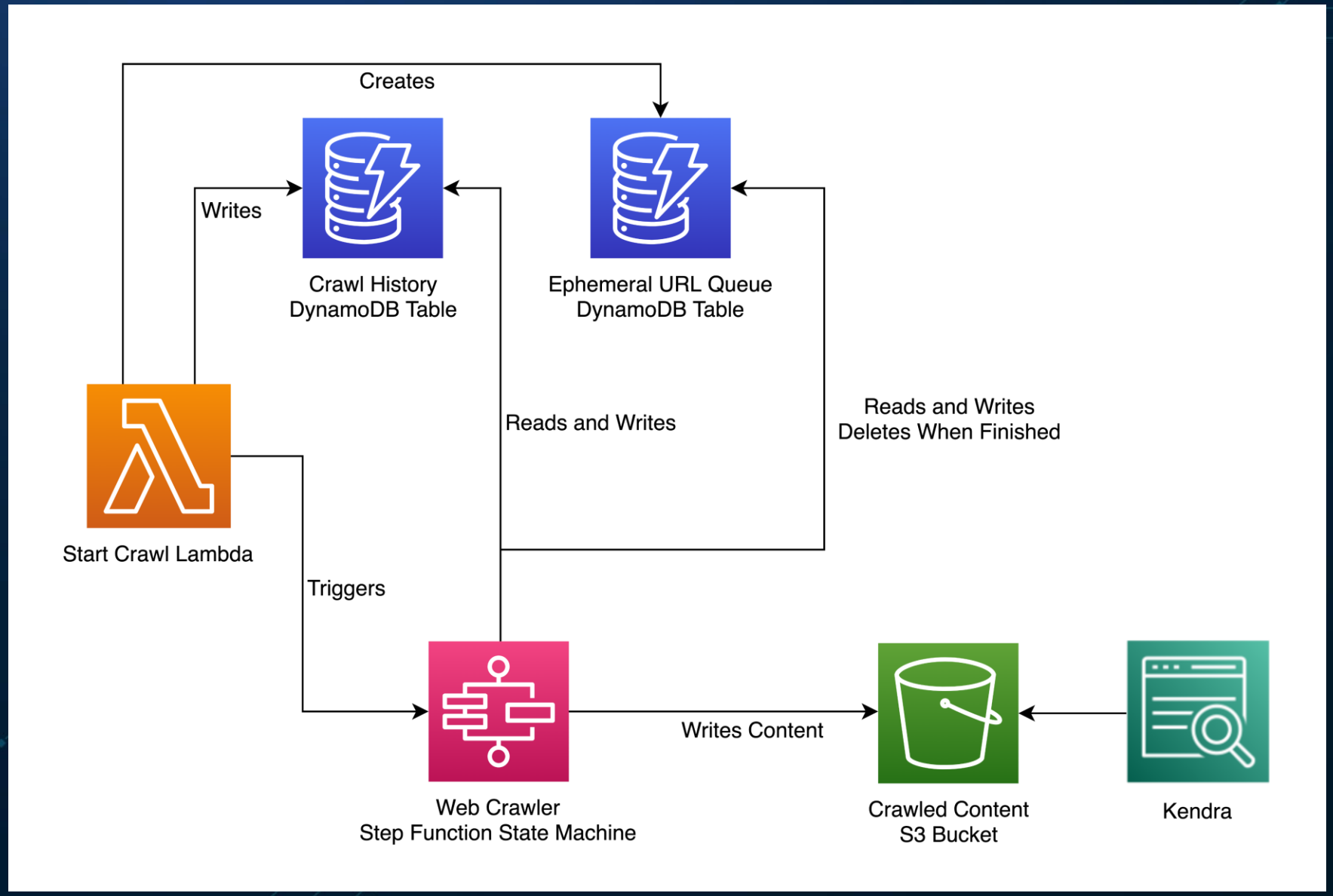
- Amazon S3
- Amazon EMR
- Amazon ElasticSearch

- Amazon RDS
- Amazon DynamoDB
- Amazon ElasticSearch

Web Crawler Architecture on AWS (1)

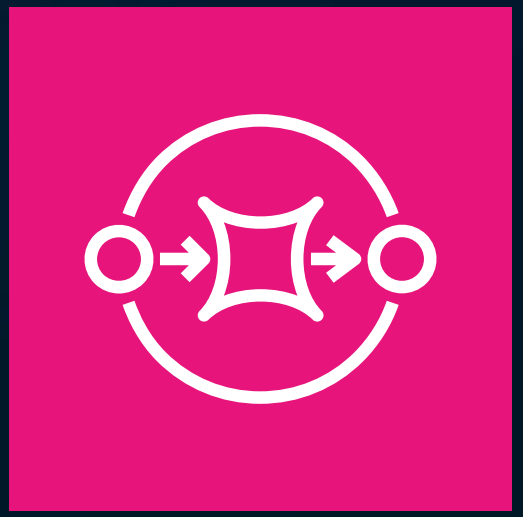


Web Crawler Architecture on AWS (2)



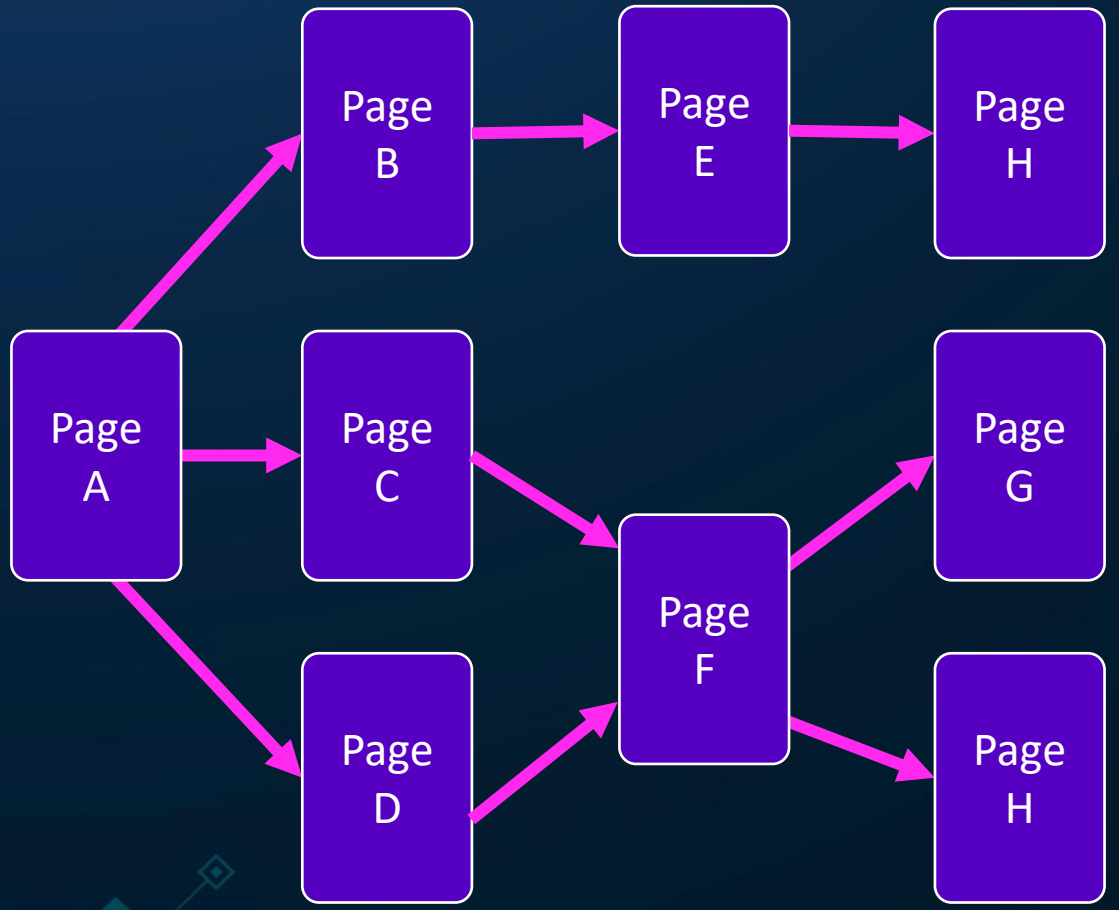
Further deep dive – Amazon SQS

- Use Standard Queue (not FIFO)
- Batch Operations
- Long Polling
- Visibility Timeout Configuration
- Use Dead-Letter Queues (DLQs)
- Handle Duplicates Gracefully
- Scale Consumers Dynamically
- Monitor and Alert



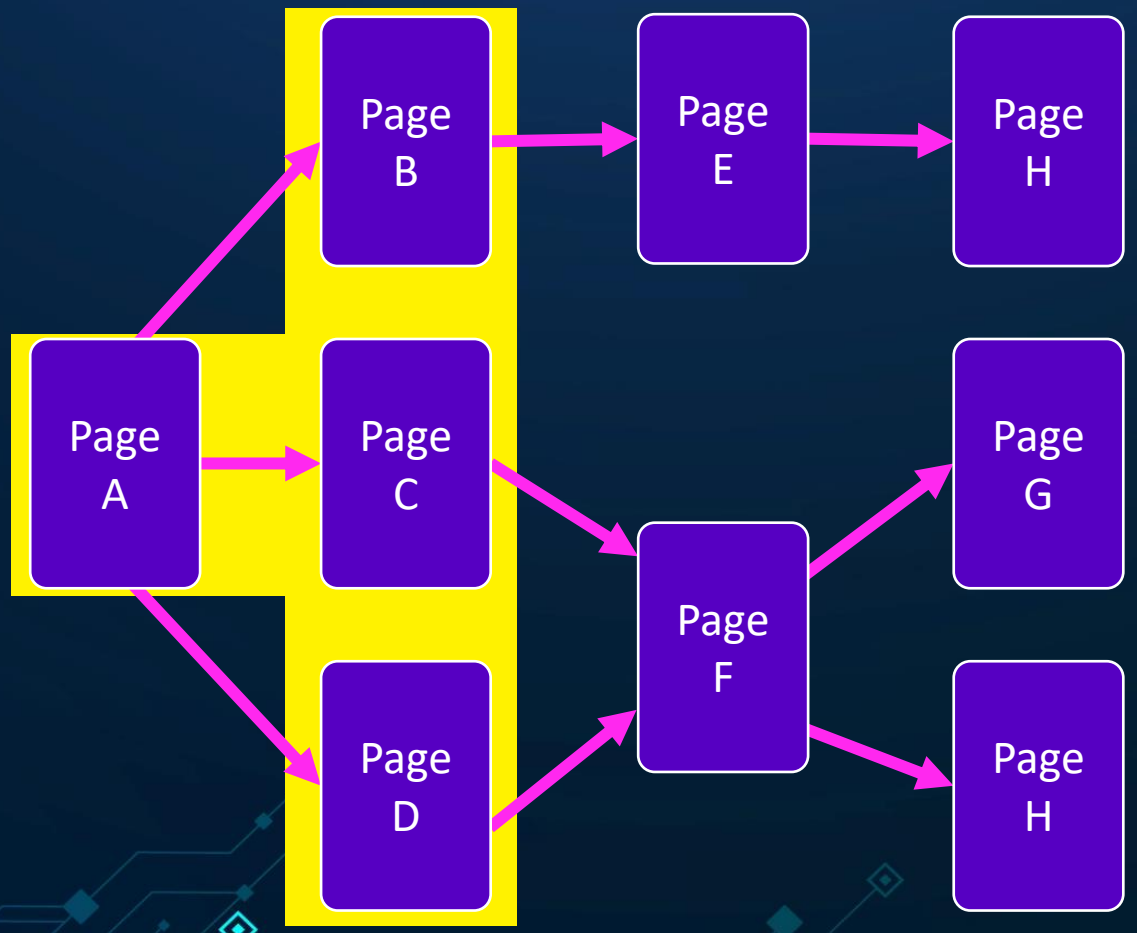
Further deep dive – Algorithm

- Web traversal = Graph traversal
  - Webpage = Nodes
  - Links = Edges

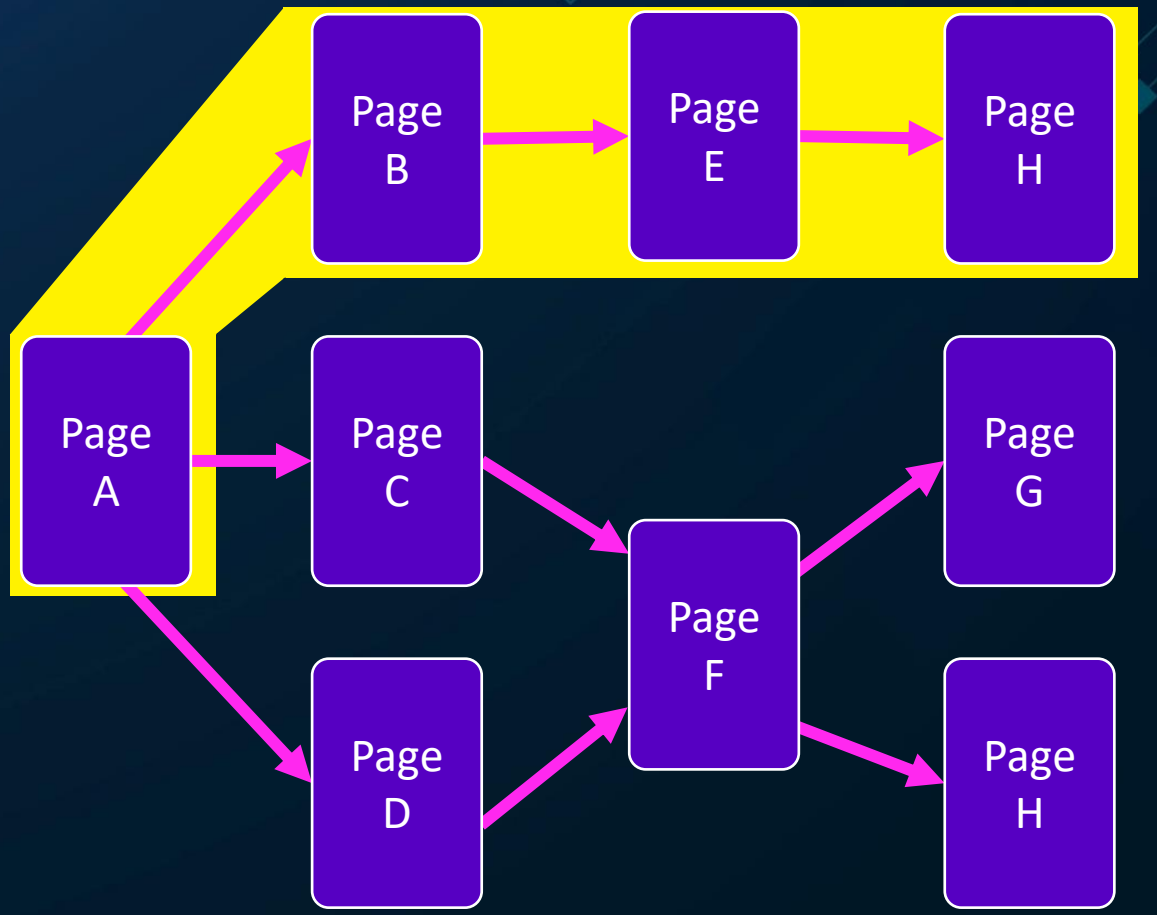


Deep dive – Web Crawler Algorithm

Breadth-First Search (BFS)



Depth-First Search (DFS)

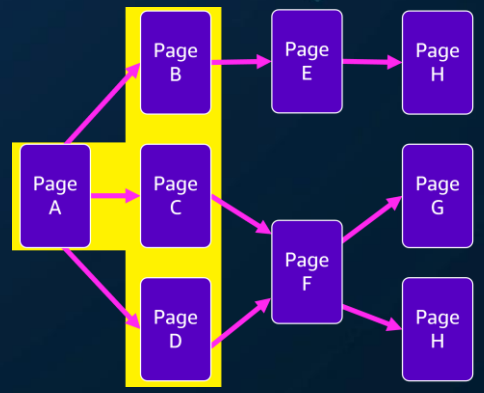


Breadth-First vs. Depth-First Search

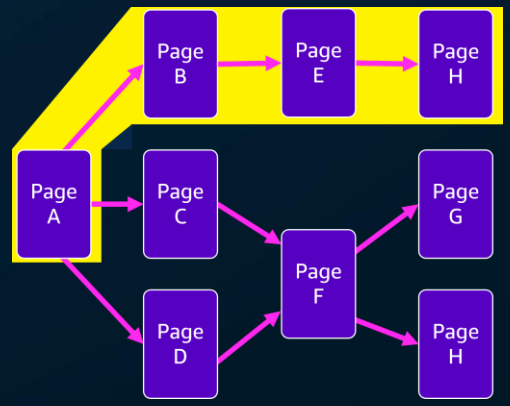
Feature	Breadth-First Search (BFS)	Depth-First Search (DFS)
Concept	Explores all neighbours (links) of the current page before going deeper.	Explores one path fully down before backtracking.
Page Discovery	Shallow pages first	Deep pages first
Risk of Trap	Lower	Higher (deep recursion/loops)
Pros	• Good for discovering pages near the root. • Useful if you want broad coverage or priority to high-level links (e.g., home → categories → articles).	• Lower memory usage (fewer URLs held at once). • Can reach deeply nested pages faster.
Cons	• Memory-intensive (stores many URLs in the queue). • Can be slower for deep content (buried links).	• Risk of going too deep into one domain/folder. • May miss breadth or take long to reach lateral links.

Which Is Better for a Web Crawler?

- Breadth-First Search is generally better for building web crawlers, especially for:
  - Search engines like Google
  - Indexing content hierarchically
  - Avoiding deep, irrelevant traps
  - Ensuring better URL diversity



- When DFS Might Be Useful:
  - Crawling specific deep resources (e.g., archives)
  - Controlled, domain-specific crawling
  - Crawling APIs or sites with a known structure

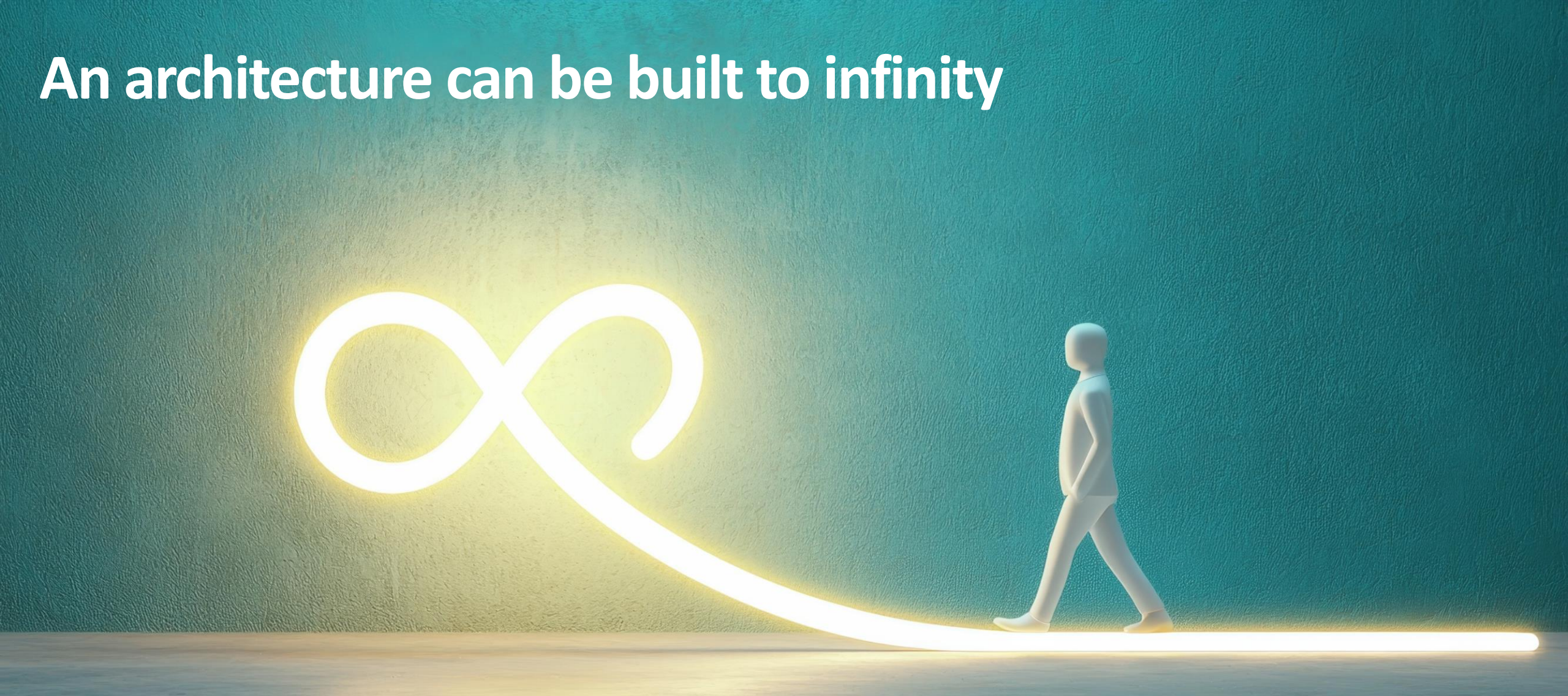


What else?

- Database schema design
- Prefix design for storage
- Monitoring and analytics
- Scheduling and content refresh
- Domain filter lists
- Content deduplication
- Trap Detection & Loop Avoidance
- Content other than text
- Multi-Tenant Architecture
- Well architected review

And many more…

An architecture can be built to infinity



No Perfect End-State	Scale Drives Complexity
Infinite Trade-offs	Technology Evolves

In Practice: The Real Takeaway

- Aim for evolvable, modular, and observable systems — not infinite ones.

- Design knowing that:
  - You won’t get everything right the first time.
  - Your architecture should support change, not resist it.
  - You must know when to stop and ship.