Solution: Crawler for Discovering Product URLs on E-commerce Websites

URL Pattern Analysis

Approach 1: Heuristic-Based Pattern Matching

Implementation:

- **URL Pattern Matching**:
  - Use regex patterns for common product URL structures (e.g., `/product/`, `/p/`, `/dp/`).
  - Example regex: `re.compile(r'/product/¥d+')` for numeric product IDs.
- **Content-Based Analysis**:
  - **Keyword Scoring**: Assign scores for product-specific terms like "price", "SKU", "add to cart", and "specifications".
  - **Structural Checks**:
    - Detect `<meta property="og:type" content="product">`.
    - Check for schema.org `Product` markup.
    - Look for multiple product indicators (price + add-to-cart + product images).
  - **Threshold Logic**: Classify as product page if total score exceeds threshold (e.g., ≥3/5 indicators).

Limitations:

- False positives from marketing pages or category pages with "Add to Cart".
- Static rules may fail for novel URL structures.

Improvements:

- Combine URL patterns with DOM structure analysis (e.g., presence of `price` class or `data-product-id` attributes).
- Use negative keywords (e.g., "blog", "category") to reduce false positives.

Approach 2: Dynamic Rule Database

Implementation:

- **Rule Storage**:

    - Use Redis/PostgreSQL to store `{domain: regex_pattern}` pairs.
    - Example entry: `{”` [`example.com`](http://example.com) `”: ”/product/[a-zA-Z0-9]+”}`.
- **Rule Discovery**:
    - Seed with sitemaps or high-traffic pages.
    - Cluster URLs by path segments to auto-generate regex patterns (e.g., `/product/<id>` → `r’/product/¥w+’`).

Advantages:

- 95-99% accuracy for known domains.
- Minimal computation during crawling.

Challenges:

- Cold-start problem for new domains.
- Requires monitoring for URL pattern changes.

Improvements:

- Hybrid approach: Use dynamic rules as primary, fall back to heuristics/ML for new domains.
- Add versioning to rules to track changes over time.

Approach 3: Machine Learning

Implementation:

- **Features**:
    - URL structure (e.g., path depth, numeric segments).
    - HTML tags (`<h1>`, `<meta name=”product_id”>`).
    - Text embeddings of page content (TF-IDF or BERT).
- **Model**:
    - Binary classifier (product vs. non-product) using a lightweight neural network.
    - Train on labeled dataset from diverse e-commerce sites.

Advantages:

- Adapts to unseen URL patterns.
- Handles edge cases (e.g., parameterized URLs like `?product_id=123`).

Challenges:

- Requires ongoing retraining.

- Computational overhead for real-time inference.

Improvements:

- Use semi-supervised learning to auto-label new URLs.
- Deploy model via TensorFlow Serving for scalability.

Scalability

We need to address **concurrency, distributed processing, and resource optimization** to make this crawler scalable for handling hundreds of domains.

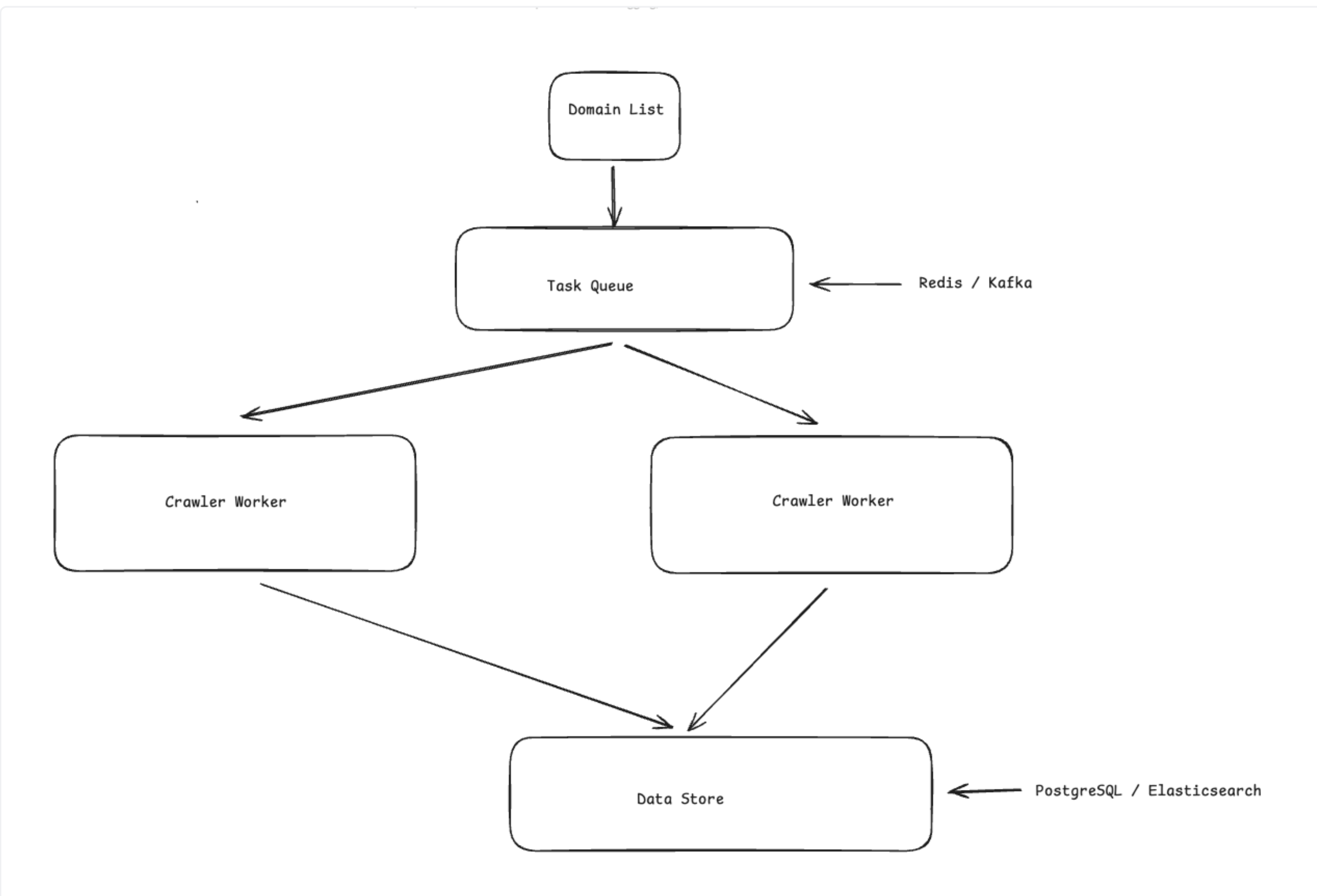

Fig: Distributed Architecture

Distributed Processing:

Tools:

- **Scrapy Cluster** (Redis/Kafka): For URL prioritization and distributed task queues.
- **Kubernetes**: Orchestrate crawler workers across nodes.

Concurrency:

Implementation:

- **Async I/O**:
  - Use `asyncio` + `aiohttp` for 1,000+ concurrent requests.
  - Limit connections per domain using semaphores.
- **Rate Limiting**:
  - Respect `robots.txt` crawl-delay.
  - Auto-throttle based on response times (e.g., 200ms/request).

Resouce Optimization:

Strategies:

1. **Priority Queues**:
   - Prioritize `/product/*` > `/category/*` > others.
   - Use Redis's `ZSET` for weighted priorities.
2. **Proxy Rotation**:
   - Integrate with services like BrightData/ScrapingBee.
   - Rotate user agents and IPs using middleware.
3. **Headless Browsers**:
   - Deploy Playwright in Docker for JavaScript-heavy sites.
   - Reuse browser instances to reduce overhead.