ChatGPT

Crawler-Guide Bot: Advanced Design & Architecture

Introduction: Modern web crawlers must intelligently decide what to crawl next in order to maximize the value of fetched data under tight resource and politeness constraints. In 2025, naive breadth-first or fixed-interval crawling is no longer viable at scale – instead, **machine learning-driven scheduling** has become essential [1]. The **Crawler-Guide Bot** is an intelligent orchestration layer that sits in front of distributed crawlers, using learned models and policies to predict and prioritize the most valuable URLs for each crawler instance. This design blends advanced ML models (graph neural networks, transformers, bandits) with a robust **Model Context Protocol (MCP)** interface to operate seamlessly across a fleet of crawlers. The result is a system that continuously learns from crawl data and dynamically guides crawling processes toward fresh, relevant, high-value content while respecting budget and politeness constraints.

High-Level Vision

Goal	How the Bot Achieves It
Predict the most promising URLs for each crawler instance.	A lightweight next-URL predictor model trained on crawl history, link graph topology, and external signals (e.g. popularity metrics, freshness indicators). It suggests which URLs are likely to yield high-value content if crawled next.
Evaluate / Score every candidate URL.	Multi-objective scoring that combines **(a) relevance to the crawler's current focus, (b) freshness (time since last fetch), (c) authority/impact metrics**, and (d) cost & politeness factors. This yields a priority rank for each URL.
Filter out disallowed or low-value URLs.	A rule-based engine with guardrails and a risk classifier drops URLs that violate policies (robots.txt, blacklists) or are likely low-value/malicious.
Expose predictions via a standard **MCP interface** so many crawlers can query together with minimal latency.	The bot runs behind an RPC/gRPC or WebSocket API (via a thin context adapter) that returns a uniform JSON/YAML response. This allows any crawler (in any language) to request the next-URLs easily.
Scale to billions of queries across data-centers.	Stateless, horizontally scalable microservices with sharded context storage and caching. The design supports massive parallelism and geo-distributed deployment, ensuring low latency and high throughput.

In essence, the bot serves as a central "brain" predicting the best crawl targets and enforcing policies. This approach aligns with industry trends where ML-driven scheduling maximizes crawl yield and freshness [2], replacing ad-hoc rules with learned predictions.

2️⃣Architecture Overview

The Crawler-Guide Bot is composed of modular services that communicate via the Model Context Protocol. At a high level, crawlers send context to the bot and receive a filtered, sorted list of URLs to crawl next. The architecture balances **prediction** (ML inference) with **policy enforcement** and **scalability**. An overview is illustrated below:

```
+-------------------+       +---------------------------+
|  Crawler Instances| <----> | MCP Gateway / Router      |
+-------------------+       +-------------+-------------+
                                 |             |
                                 v             v
                            +---------------+   +-----------------+
                            | Prediction API |   | Scoring Service |
                            +--------+-------+   +----------+------+
                                     |                     |
                                     v                     v
                          +-----------------------------------------+
                          |   Context Store (sharded key-value store) |
                          +----------------------+--------------------+
                                                 |
                                    +----------------------v---------------------+
                                    |    Unified Prediction & Policy Layer        |
                                    +--------------------------------------------+

Result: filtered top-N URL list → returned to each crawler via its native
crawler-loop integration point (API, message bus, etc).
```

Core Components

Component	Responsibility	**Typical Tech Choices** (examples)
MCP Gateway / Router	Exposes a well-known API endpoint for crawlers. It multiplexes incoming requests, handles authentication, and applies rate limiting. Routes each request to the Prediction/Scoring backend.	Examples: Envoy or NGINX as reverse proxies, with gRPC or HTTP/2; API gateways (Kong, AWS API Gateway) or custom microservice.
Prediction API (Model Server)	Returns the top-K predicted next URLs for a given crawler context. Internally uses a **next-URL predictor model** (which can be a graph neural net, transformer, etc.). Supports high QPS inference (possibly with GPUs or optimized runtimes).	Examples: TorchServe or NVIDIA Triton for serving PyTorch/ TensorFlow models [3] [4] ; or a FastAPI+Uvicorn microservice wrapping a model.

Component	**Responsibility**	**Typical Tech Choices** (examples)
Scoring Service	Takes a raw set of candidate URLs (and the crawler's context like budget, TTL) and computes a composite priority score for each URL. Implements the multi-factor scoring formula (relevance, freshness, authority, cost, politeness). Returns an ordered list.	Examples: For large batches, stream processing with Flink or Spark; for per-request, an in-memory service in Go or Python. Redis Lua scripts or sorted sets can be used for fast scoring calculations.
Filtering / Policy Engine	Enforces rules and risk policies on the candidate list. Removes any URLs disallowed by robots.txt, outside the crawl scope, or exceeding politeness limits. Also uses a trained classifier to drop URLs likely to cause errors or traps.	Examples: Open Policy Agent (OPA) with Rego policies for static rules; a Python or Go plugin module for custom filters; an ML model for risk classification (e.g. scikit-learn or TensorFlow for binary classification of "safe to crawl").
Context Store	Holds state and metadata needed for decisions. This includes per-URL and per-domain data: e.g. last crawl timestamp, fetch success/failure counts, content fingerprints, PageRank/impact scores, etc. A fast key–value or NoSQL store, sharded by URL or domain, ensures quick lookups and updates.	Examples: Redis or Aerospike cluster for low-latency K/V; DynamoDB (with DAX caching) for serverless scale; or a sharded RocksDB via a distributed log (e.g. NATS JetStream). In practice, scalable K/V storage is crucial for large crawl frontiers [5].
Model Trainer & Feeder (offline)	Runs offline to periodically retrain the prediction models (and any authority or risk models) using fresh crawl data. Deploys updated model snapshots to the Prediction API without downtime.	Examples: Airflow or Prefect for orchestration; model versioning with MLflow; Canary deployments via Kubernetes (Argo Rollouts) to test new model versions on a small fraction of traffic before full release.

How it works: A crawler instance sends a context query to the Gateway (e.g., via gRPC). The Gateway forwards it to the Prediction API, which fetches needed context from the store and runs the ML model to propose candidate URLs. The candidate list is then fed to the Scoring Service, which applies the multi-objective formula and queries the policy engine to filter out any disallowed URLs. The final pruned list is returned to the crawler. This modular separation ensures each concern (prediction vs. scoring vs. filtering) can scale and evolve independently. Notably, similar multi-stage architectures have been used in large-scale crawlers – for example, Scrapy's **Frontera** framework separates strategy workers (for scoring/prioritization) from crawling workers using a message queue [6] [7].

3 The Prediction Layer – "What to Crawl Next?"

3.1 Input Context (per crawler request)

When a crawler asks for new URLs, it provides a **context payload** that captures its state and needs. Key fields include:

- **`crawler_id`:** Unique identifier for the crawler instance (used to isolate its crawl budget and scope).
- **`seed_url` or `domain_hash`:** The starting point or domain the crawler is focused on (helps partition the frontier by site or topic).
- **`budget_quota`:** How many URLs the crawler can still fetch in this cycle (e.g. it may ask for up to 50 URLs).
- **`time_to_live_ms`:** How much time remains for this crawler (it might be a ephemeral task that restarts periodically).
- **`historical_yield`:** Stats from past cycles (e.g. ratio of new pages discovered vs. visited, which hints at diminishing returns).
- **`external_features`:** Any external signals (domain PageRank or MozRank, last modified timestamps, social media popularity of URLs, etc.).

Upon receiving this, the service **enriches** the context with global data from the Context Store, for example:

- **Global Link Graph stats:** e.g. outbound link count of each candidate URL, number of other sites linking to it (backlinks), anchor text signals, etc.
- **Domain reputation & history:** e.g. average freshness of the site (how often it updates), last crawl status (HTTP 200 vs errors), average page load time, known traps or login requirements.

This enriched context forms the input to the prediction model, allowing it to consider both the local crawler state and the broader web graph/neighborhood around the seed.

3.2 Predictor Model Families

Different ML model types can be used (and even combined) to predict URL utility. Each has strengths:

- **Graph Neural Networks / Transformers:** These models (e.g. Graph Attention Networks, GraphSAGE, or Transformer-XL variants operating on graph data) learn embeddings for URLs based on the link graph structure. They capture long-range dependencies – for example, a URL deeply nested in a site can still be deemed important if high-authority pages link to it. A Graph Neural Network can output an importance score for each URL given the current context. Frameworks like PyTorch Geometric or DGL make it feasible to run inference on graph-based models [8] [9]. This approach often yields the best global signal but can be computationally heavy.
- **Contextual Ranking Models (Cross-Encoders):** These take a pair of inputs – (candidate URL, crawler context features) – and directly predict a relevance score. For instance, a BERT-based cross-encoder can be fine-tuned to output how relevant a given URL is to the crawler's goal (learned from past crawl outcomes). Such models look at the URL (and maybe page content snippet or metadata) in conjunction with context, yielding a powerful relevance prediction [10]. However, cross-encoders are expensive since they perform full attention over each (URL, context) pair, so they are typically used to re-rank a small set of top candidates [11] rather than applied to thousands of URLs.

- **Reinforcement Learning Policies (Multi-Armed Bandits):** Here, each URL (or site) is treated as an "arm" in a bandit problem. Crawling a URL gives a reward (e.g. discovering new content or a successful fetch) or a penalty (error, low yield, high latency). An RL agent can learn a policy to maximize long-term reward under the crawl budget. For example, a **contextual bandit** may use the crawler's context as state to decide the next URL, balancing exploration vs. exploitation [12]. Multi-armed bandit algorithms (e.g. Thompson Sampling) have been proposed for crawl scheduling where the reward is information gain and the cost (negative reward) is bandwidth/time [13] [14]. Such an approach adapts on the fly, learning which sites are "worth" crawling more frequently.
- **Lightweight Heuristic Layer (Fallback):** In scenarios requiring ultra-low latency (tens of thousands of predictions per second), the system can fall back to simple heuristic scoring – e.g. a weighted sum of known features like PageRank, site update frequency, and remaining budget. This is essentially a hand-tuned formula that, while not as adaptive as ML models, can execute in microseconds. The heuristic can act as a first-stage filter, and then the heavier ML models (above) only refine the top-N results. This mirrors the common **two-stage ranking pipeline** used in search engines, where a fast retrieval step precedes a slower re-ranking step [15] [10].

  Why multiple models? A layered predictor yields the best balance of coverage and efficiency. For example, a GNN might preprocess the entire frontier with an importance estimate, a supervised model then re-ranks the top candidates more precisely, and an RL policy fine-tunes the selection online to account for recent changes. Using a heavy cross-encoder re-ranker on only a small candidate subset dramatically improves precision without slowing down the whole system [11]. This multi-model approach is analogous to modern search systems where a bi-encoder retrieves candidates and a cross-encoder re-ranks them for final quality [16] [17].

3.3 Training Pipeline

Building an effective predictor requires continuous training on fresh data:

1. **Data Collection:** Each crawl produces rich data that is logged to a Crawl Events dataset. For every URL attempted, the system logs whether it was fetched successfully, the HTTP status, content length, fetch latency, and the list of outlinks extracted. The content may be hashed or stored (for change detection). Over time this yields a large table of `(URL, timestamp, crawler_context, outcome)` records. This can be augmented with downstream signals – for example, if a fetched page turned out to have high-value content (perhaps measured by it being indexed or used by some application), that can be noted for training labels. [18] [19]

2. **Label Generation:** We define what constitutes a "good" outcome for a URL. Heuristics for labeling might include: discovered new content (the fetched page had a content hash not seen before), content freshness (the page had changed significantly since last crawl), or topic relevance (if focusing on a topic, was the page on-topic?). These can be turned into binary or continuous reward labels. In some cases, human feedback or business metrics (e.g. "this page yielded a price drop that saved money") are used to label high-value crawls.

3. **Feature Engineering:** For each training example (a URL crawl attempt in context), the system computes a variety of features:

4. **Graph/Link Features:** e.g. number of outlinks from that URL, number of backlinks pointing to it, the PageRank of its domain, anchor text diversity, URL depth from seed.

5. **Content/Metadata Features:** e.g. length of URL, presence of certain keywords in URL or title, MIME type (HTML vs PDF vs image), language or location hints.
6. **Temporal Features:** last fetch time, frequency of content change historically, how recently it was discovered, and how long since it was last updated (if known).
7. **Performance/Cost Features:** average fetch time for this site, past failure rate (4xx/5xx errors) for this URL or site, whether the URL tended to cause CAPTCHA or bans.

In practice, these features can be quite detailed. For example, one can include **path tokens** (splitting the URL path into words), **site-level patterns** (like the site's overall change rate), or **similarity clusters** (does this URL belong to a cluster of pages that often change together?). All these features are computed and normalized to feed into the models [20] [21]. Temporal features like "time since last crawl" and change frequency are especially important for predicting freshness [21], while performance features like past latency or block rate help the model learn the cost or risk associated with a URL [22].

1. **Model Training:** The system may train two sets of models corresponding to the earlier "model families" – e.g. a graph encoder model that produces a base score for each URL, and a re-ranker model that refines the score given context. Training is done on GPUs for speed (using frameworks like PyTorch Lightning or TensorFlow). Techniques like mixed precision and distributed training are used due to the large volume of data. The models are validated on recent crawls (holding out some portion of data to ensure they generalize to new time periods or sites). If using an RL approach, training might involve simulations or replaying crawl sequences to tune a policy network.

2. **Versioning & Deployment:** Each trained model is packaged (exported to ONNX or TorchScript for consistency across languages). A model registry tags a new version (e.g. v2025-11-01) which is then deployed to the Prediction API servers. To avoid disruptions, new versions can be rolled out gradually – for example, deploy the new model on a subset of inference servers and compare its decisions against the previous version (a form of A/B testing or canary release). Only when the new model shows improvement (e.g. better yield or fewer errors) do we switch all traffic to it. The MCP interface can include a version field so that crawlers can report which model version guided their crawl, aiding in monitoring.

4️⃣ Scoring and Multi-Objective Evaluation

After the predictor proposes a set of candidate URLs (often with an initial relevance score), the **Scoring Service** refines these into a final crawl order by incorporating additional objectives. We define a composite score function:

```
score(url) = α · Relevance_model_output
           + β · Freshness_factor
           + γ · Authority_weight
           − δ · Cost_penalty
           + ε · Politeness_score
```

Each term corresponds to a facet of the URL's priority:

- **Relevance:** The predictor's raw output for how valuable this URL is in the current context. This could be the probability of a "good outcome" or a learned relevance rank. (Weight α might be high for focused crawls where topical relevance is crucial.)
- **FreshnessFactor:** A value that decreases the longer it's been since the URL (or site) was last crawled. For example, we can use an exponential decay like `exp(-Δt/τ)`, where Δt is time since last fetch and τ is a half-life parameter. This term ensures recently changed or newly discovered pages get priority over ones that were crawled recently (which likely haven't changed yet).
- **AuthorityWeight:** A measure of the URL's global importance or authority. This could derive from the PageRank of the page or domain, the number of high-quality backlinks, or an "influence score" from social media. Pages that are more authoritative or popular get a higher weight (γ). However, authority alone shouldn't dominate since it can bias toward older/static pages [23], so it's balanced with freshness.
- **CostPenalty:** This subtractive term accounts for the estimated cost of crawling this URL. For instance, if the URL is known to load very slowly or has huge images (large bytes), it gets a penalty. Similarly, if the site tends to trigger bans (necessitating expensive proxy/IP rotation or solving CAPTCHAs), the cost penalty would be higher. This helps the scheduler prefer cheaper URLs when value is otherwise similar.
- **PolitenessScore:** A bonus for URLs that are "safe" to crawl frequently with respect to the target site's rate limits. If a site has a `Crawl-Delay: 10` directive, for example, the scheduler might reduce the score for additional URLs from that site if it's already fetched one recently [24]. Conversely, if the crawler has built up credit (e.g. it hasn't hit a site in a long time, so it's polite to do so now), this can slightly raise the score. This term implements per-site politeness and ensures we distribute crawls across sites fairly [25].

The weights (α, β, γ, δ, ε) are hyperparameters set by the crawl operator or learned through regression. The **Scoring Service** computes this score for each candidate URL and then sorts URLs by this score in descending order (highest priority first).

Finally, if the crawler's budget_quota is N, the service will take the top N URLs (or a bit more to account for filtering, see below) as the recommended list. In effect, scoring transforms the predictor's suggestions into an actionable priority queue that balances relevance with practical constraints. This approach is similar to best-first crawling strategies, but instead of a single heuristic like PageRank or a fixed schedule, it uses a weighted blend of signals, which is much more adaptable and can be tuned to specific goals [26] [23].

5️⃣Filtering & Policy Enforcement Engine

Even after scoring, the system must enforce hard constraints and protect against unwanted URLs. The Filtering/Policy Engine acts as a final gate:

5.1 Static Rules (Enforced Immediately)

Some rules are non-negotiable and checked for every candidate URL: - **Robots.txt Policies:** The engine checks the site's latest robots.txt to ensure the URL is allowed for crawling. If the URL path is disallowed for our user-agent, it is dropped **unless** the crawler is operating in an override mode (e.g. administrative crawl where we have permission despite robots.txt). The system fetches and caches robots.txt for domains and respects standards like `Disallow` and `Noindex` directives [27] [28]. - **Domain Blacklist/**

Whitelist: The engine maintains lists of domains that are completely off-limits (e.g. known malware domains, or per policy such as no crawling government sites) and those that are exclusively allowed. Any URL on the blacklist is removed; those not on a required whitelist (if in use) are removed. This provides an administrative control over the crawl scope. - **Crawl-Delay & Rate Limits:** If a site's robots.txt specifies a `Crawl-delay` (an unofficial but widely used directive to throttle crawl rate), the engine will ensure we do not exceed that rate [24]. Additionally, the Context Store might track how many requests have been made to a domain in the last minute. The engine will defer or drop URLs that would violate politeness (e.g. more than X requests in Y seconds to the same site). Essentially, each host/domain has a token bucket of requests – if empty, further URLs from that host are held back.

These static rules are often implemented with a high-speed policy engine (like OPA with pre-loaded rules) for millisecond decisions. By compiling rules into bytecode or using in-memory checks, even thousands of URL candidates can be filtered in negligible time.

5.2 Dynamic Risk Classification

Beyond static rules, the system uses an ML-based **Risk Classifier** to protect against problematic URLs. This classifier (trained on historical crawl outcomes) flags URLs that are likely to cause issues such as: - **Error-prone URLs:** e.g. those that consistently return HTTP 404/410 or server errors (500s), or time out. If the model predicts a high probability of failure, the URL might be skipped to save bandwidth. - **Spider Traps / Bot Traps:** Certain URLs are designed to trap crawlers in infinite loops or overload them (e.g. calendar pages, infinite scroll APIs, or deliberately autogenerated pages). The classifier uses features like URL patterns (very long numeric sequences, or repeated path segments) to detect likely traps (sometimes called crawler traps [29]) and excludes them. - **Malicious or Heavy Resources:** URLs pointing to huge files (videos, large binaries) or known malware distribution are dropped. Also, if a URL is predicted to lead to a content that triggers excessive computation (like an extremely heavy JavaScript page), it might be filtered if it's not mission-critical.

For any URL that the risk model flags with probability > 0.8 (for example), the engine will **reject** it, log the reason ("high risk: possible trap"), and possibly mark it in the Context Store so future predictions avoid it too. This keeps the crawl efficient and safe.

5.3 Budget- and Diversity-Aware Pruning

Finally, the engine ensures the final list of URLs respects budget and diversity constraints: - **Per-Domain Deduplication:** It avoids giving one crawler 100 URLs from the same domain at once (unless specifically intended). For example, if the top scores are all from `example.com`, the engine might only allow the top K from that domain and push the rest to a later cycle. This prevents over-concentration on a single site and improves overall coverage. - **Duplicate URL Removal:** If by chance the candidate list contained the same URL twice (due to merging lists), it will be unified. - **Budget Cap:** If the crawler asked for 30 URLs, the engine will only return at most 30 (or slightly fewer if many were filtered out). Any extra are dropped. The engine makes sure that the total estimated bytes of those 30 URLs do not wildly exceed the crawler's `max_bytes` budget either (if provided). For instance, if all 30 happen to be video links and would sum to several GBs, it might reduce the number or mark them as over-budget. - **Scheduling Delays:** In some designs, the policy engine could also schedule some URLs for later instead of now. For example, if two high-priority URLs are from the same site and it's not polite to hit them together, one URL might be withheld with a note like "defer until next cycle for politeness."

The outcome of filtering is a **final prioritized list** of URLs that the crawler is permitted and expected to crawl next. This list is then sent back to the crawler through the MCP Gateway.

Stateful Scheduler Note: The system often maintains a scheduler state (e.g. in Redis) per crawler and per site to track these quotas. This ensures that if multiple crawler instances ask for URLs from the same domain, they collectively don't violate the domain's crawl-delay or max concurrent fetches. In our design, the Context Store or a small Scheduler Service can handle this coordination. This approach is in line with how large-scale distributed crawlers enforce politeness by partitioning work per host [25] .

6️⃣Model Context Protocol (MCP) – Uniform Interface

To integrate with diverse crawlers (written in different languages and running in different data centers), the system exposes a **Model Context Protocol** interface. MCP defines a standard message format (JSON or Protobuf) and endpoints for requesting URL predictions. This decouples the crawler logic from the prediction engine.

6.1 Example Request/Response Schema

A simplified MCP **request** (JSON) might look like:

```
{
 "mcp_version": "2025-11",
 "crawler_id": "c-902",
 "request_id": "123e4567-e89b-12d3-a456-426614174000",
 "budget": {
   "max_urls": 30,
   "max_bytes": 268435456,   // 256 MB
   "timeout_ms": 120000      // 2 minutes
 },
 "context": {
  "seed_urls": ["https://example.com/a"],
  "recent_history": [
    {"url": "https://example.com/a", "fetch_time": "2025-11-03T14:15:00Z", "status": 200}
  ],
  "metadata": {
    "crawl_cohort_id": "Q4-2025"
  }
 },
 "request_timestamp": "2025-11-03T14:17:31Z"
}
```

Key fields explained: `crawler_id` identifies the client; `budget` gives the crawler's resource limits; `context` includes current seed URLs, some recent crawl history (could be empty or a summary of what it last fetched), and an extensible metadata map (used for things like cohort tagging, experiments, etc.). The `request_id` is a UUID for idempotency and tracing (useful if the same request is retried).

A corresponding MCP **response** might be:

```
{
 "assigned_urls": [
  {
   "url": "https://example.com/b",
   "score": 0.938,
   "reason": "high_relevance+recency",
   "estimated_fetch_ms": 12456
  },
  {
   "url": "https://other.com/page?id=123",
   "score": 0.912,
   "reason": "new_domain_high_pagerank",
   "estimated_fetch_ms": 342
  }
  // ... up to max_urls entries
 ],
 "metadata": {
  "num_queries_completed": 7245,
  "last_model_version": "v2025-11-01",
  "filters_applied": ["robots_allow", "politeness_ok"]
 }
}
```

Here, `assigned_urls` is the list of URLs recommended (typically sorted by score). For each, the bot can optionally include a `reason` string – a human-readable explanation or tag (useful for logging/debugging, e.g. “high_relevance+recency” might mean this URL scored high due to both topical relevance and being recently updated). `estimated_fetch_ms` is an estimate of how long the fetch might take (perhaps from historical data). The `metadata` in the response can include any stats the bot wants to convey, such as how many queries it has handled or the model version used. It could also list summary info like `"filters_applied": [...]` indicating which filter rules were applied (for transparency, e.g. maybe it filtered 5 URLs due to robots.txt – though not shown in the above JSON for brevity).

This protocol is versioned (here `"mcp_version": "2025-11"`) to allow evolution. By using a consistent MCP, any crawler (whether a central scheduler or individual spider) can query the bot with minimal integration effort.

6.2 Transport Options

Several transport mechanisms can carry MCP messages:

- **gRPC / HTTP/2:** This is a popular choice for its performance and built-in features. The bot can define a protobuf schema for the request/response and host a gRPC service. Benefits include binary serialization (efficient), multiplexing over a single connection, and native streaming support. Latency is low (sub-5 ms overhead typically). The downside is that the client needs gRPC support and code generation from .proto files, which might be a barrier for some legacy crawlers.
- **WebSocket (binary or JSON):** This allows maintaining a persistent connection where crawlers can send a stream of requests and receive streamed responses. It’s useful for real-time or interactive scenarios (e.g. if the crawler wants continuous updates). WebSockets can reduce connection setup overhead and are fairly scalable, though a single node might handle tens of thousands of open sockets so careful load balancing is needed.

- **Message Bus (e.g. Kafka):** In a batch crawling scenario, the crawler might not call an API at all, but instead publish a message (with the context) to a Kafka topic. The Crawler-Guide Bot (or rather a consumer service for it) would consume the message, do its work, and publish the result to another topic which the crawler listens to. This decouples the system and can be useful for extreme scales or when the crawling is asynchronous. The trade-off is added complexity and slightly higher latency (since it's not direct RPC but via queue).

In our design, the **MCP Gateway/Router** abstract these details. For instance, it might expose both a gRPC endpoint and a WebSocket server, both translating to the same internal service calls. The gateway can also perform input validation (e.g. JSON schema checks) to reject malformed requests before they hit the core services.

Pro-tip: Using a standardized MCP means that if you have multiple crawling systems (maybe one focused on news sites, another on e-commerce), they can all tap into the same prediction service simply by conforming to the protocol. This centralizes the "intelligence" and makes it easier to deploy improvements universally.

7️⃣Deployment Considerations – From Single Server to Planet-Scale

Depending on the scale of operations, this system can be deployed on a single server or distributed globally. The design is cloud-native and emphasizes horizontal scalability.

7.1 Centralized Deployment (Single Region)

For moderate loads (say, up to a few thousand predictions per second), a centralized approach simplifies operations: - **All-in-One Cluster:** Deploy the Prediction API, Scoring Service, and Context Store in one region/data center. For example, 2–3 application servers behind a load balancer and a small Redis cluster for state. - **Resource Sizing:** Each service instance might be something like 8 vCPUs and 32 GB RAM. If using GPU for the model, perhaps a single GPU box can handle inference (modern GPUs can serve thousands of transformer inferences per second). Memory is mainly for caching recent context and model data. - **Throughput:** A single modest server can handle thousands of QPS, especially if much of the logic is I/O bound (database lookups) and the model is optimized. - **Advantages:** Low complexity, all data in one place (no cross-region latency), easy to debug. Model updates are straightforward – you replace the model on the central server and you're done.

This scenario is analogous to an academic prototype or a small production system where crawlers and the predictor are all in one cloud region. Latency from crawler to predictor is maybe < 10 ms. The **single source of truth** nature (one context store) avoids inconsistencies.

However, as traffic or the number of crawlers grows, or if crawlers are worldwide, we need to distribute the system.

7.2 Distributed Deployment (Multi-Region, High Scale)

To scale to billions of URL predictions and to keep latency low globally, several strategies are employed:

- **Geographically Distributed Edge Nodes:** We can deploy MCP Gateway proxies at various geographic "edge" locations (for instance, using a CDN worker or cloud functions at edge). These edge nodes handle incoming requests nearby to the crawler, serve any cached predictions, and then forward the request to the central service if needed. This reduces round-trip time for far-flung crawlers and offloads some work (like caching or simple heuristic predictions) to the edge.

- **Sharding by Domain or Key:** The context store and possibly parts of the prediction service are sharded by some key such as domain. For example, all data for `.jp` domains could live on shard A, `.eu` domains on shard B, etc., or even a consistent hash on the URL. The Gateway can quickly route requests to the correct shard based on the crawler's current seed or domain. This ensures that each shard handles a subset of the frontier, reducing contention. It also localizes cache usage (since one shard will repeatedly see the same domains).
- **Horizontal Autoscaling of Models:** The Prediction API can be replicated N times and load-balanced. Since it's stateless (model inference only), scaling it is straightforward. For heavy models, one can use a GPU cluster or even specialized hardware (TPUs). Autoscaling triggers could be QPS or model latency. A model router component could direct specific crawler_id ranges to specific instances to improve cache hits (as mentioned, e.g., `crawler_id % N` or domain prefix mapping).
- **Batch Processing for Bulk Updates:** If facing very high throughputs, instead of scoring each request individually, one can aggregate candidate URLs into mini-batches (using a stream processor). For instance, all candidates from 100 requests might be scored together in a vectorized manner on a GPU, then results split out per crawler. Apache Flink or Spark Streaming jobs could be employed to read a stream of "candidate URL" events and output scored results into a fast datastore (like a Redis sorted set) [30] . The crawlers (or the gateway) then simply read the top results from these pre-computed sorted sets. This trades a bit of freshness (a few seconds delay) for massive throughput.
- **Fault Tolerance:** In a distributed setup, each service is replicated. If one node goes down, others continue serving. A discovery service or consistent hashing ensures that if a shard is down, requests can be rerouted to a replica or a backup shard. The system should degrade gracefully – e.g., if the ML predictor is unreachable, perhaps the gateway can fall back to a simpler rule-based suggestion so that crawling doesn't stop entirely.

Example: Imagine we deploy this system on Kubernetes across 3 regions (US, Europe, Asia). In each region, we run an Envoy proxy (with gRPC support) as the MCP Gateway. We have a global Redis cluster sharded by domain hash, with nodes also in those regions (to minimize read latency). The Prediction service runs as multiple pods with GPU support, scaled as needed, and a separate Scoring service (or it could be combined with Prediction in one process for simplicity) runs with enough instances to handle peak load. We also deploy OPA for policy checks as a sidecar or microservice. A global load balancer (DNS or IP-based) directs each crawler to the nearest MCP Gateway. Once set up, a crawler in Asia will hit the Asia gateway, which will use the Asia cache/shard for context, talk to the model service (possibly globally or in Asia if the shard indicates that), and return results – all in perhaps 50ms instead of 200ms if it had to reach a single US server.

All components are designed to be **stateless or weakly-stateful**, meaning they can be killed and restarted without losing critical data (since state is in the context store or message queues). This enables rolling upgrades and easy scaling. We also integrate **observability**: each service emits Prometheus metrics (QPS, latency, errors) and logs; a centralized Grafana dashboard tracks these. Distributed tracing (e.g. OpenTelemetry) is used to trace a request from the gateway through prediction and scoring to the database, which is invaluable for debugging performance issues.

---

8 End-to-End Walkthrough (Example Data Flow)

Let’s walk through a typical cycle to see how all the pieces interact:

1. **A crawler instance (`c-902`) sends a request** to the MCP Gateway for more URLs. It includes its current context: suppose it has just crawled a page from **example.com** and found some new links. It tells the bot: “I can crawl 30 more URLs, I have 50 MB of budget left, and I’m focusing on example.com (seed).” The `request_id` and timestamp are also included.

2. **Gateway forwards the request** to the Prediction API. The Prediction service looks at `crawler_id = c-902` and the seed `example.com`, and determines which shard (context store) holds the data for example.com. It queries that shard for context: e.g., last fetch times of any URLs on example.com, that domain’s crawl delay or robots rules (if cached), and perhaps global popularity metrics (maybe example.com’s overall PageRank or site change rate). It builds the model input from this plus the provided context fields.

3. **Candidate Generation:** Based on the seed and context, the system generates a list of candidate URLs. In this case, the crawler’s recent fetch might have provided new outlinks. Those outlinks (say URLs on example.com and a few on other.com that were found) are added to the candidate pool. The Context Store is also consulted for any high-priority URLs that were pending (for example, URLs that were deferred from a previous cycle or known important pages that come up periodically). The **Candidate Service** (which could be part of the Prediction API or a separate producer) compiles this list of, say, 100 URLs that are potential fetch targets next [7].

4. **Prediction Model runs:** The predictor model takes each candidate URL with the context features and computes a **relevance score** (or utility value). For instance, it might predict that URLs on example.com/news/ are very likely to have changed (high freshness value) and relevant, giving them a high base score, whereas a link to other.com/about might get a lower score. Suppose it assigns scores to the 100 candidates.

5. **Scoring Service ranks the candidates:** It then applies the multi-objective scoring formula to each of these 100 URLs. It pulls additional data as needed (e.g., for each URL, fetch the last_crawl_time to compute freshness decay, and the domain’s reputation score for authority, etc., from the Context Store). Each URL’s final score is computed. For example, one URL might get +0.9 relevance, +0.5 freshness (not crawled in a month), +0.3 authority (decent PageRank), –0.2 cost (it’s a large page), +0.0 politeness (first request to that domain now) = **1.5** total. Another might get +0.8, +0.0 (was recently crawled), +0.7 (high authority), –0.1, +0.0 = **1.4**. After scoring all, the service sorts them by score [30].

6. **Policy Engine filters the sorted list:** The top candidates are now checked for compliance:

7. It reads the robots.txt rules (likely cached) for each domain among the top URLs. If, say, the second-highest URL is disallowed by robots, it’s removed.
8. It checks that we’re not overloading any domain: if the top 5 URLs are all from **example.com**, and the crawl-delay or politeness budget says we should only fetch 2 from example.com at a time, it will drop or postpone the others, possibly allowing lower-ranked URLs from other domains to move up.
9. It runs the risk classifier on the top URLs – if one of them has a pattern that the model flags as a spam trap (maybe it sees “calendar=2025&month=11&day=*" in the URL, a known infinite calendar pattern), that URL is removed.

10. In our example, assume a couple of URLs were filtered out (one due to robots.txt, one due to being a duplicate already crawled recently).

11. **Final list returned:** The system now has, say, 30 URLs that passed all filters and are within the crawler's budget. These are packaged into the MCP response JSON with their scores and reasons. The Gateway sends this back over the network to **crawler c-902**.

12. **Crawler receives the URLs** and proceeds to fetch them one by one (or in parallel, depending on its internal design). As it crawls them, it will output new discovered links and page metadata, which eventually flow back into the Context Store (often via an asynchronous process or log). For instance, after crawling, it might push an event to a Kafka topic like "crawled URL X, found links {…}". Those events are consumed by an ingester that updates the Context Store (and possibly triggers the Candidate Service to consider those new links for future predictions) [31]. The crawler, once it nears the end of the 30 URLs, will make another MCP request for more, and the cycle continues.

Throughout this process, the **Crawler-Guide Bot** provides a smart decision layer, leveraging both the latest crawl data and learned models to constantly answer "Where should I crawl next?" in an optimal way. This design not only improves the efficiency of web data collection (by focusing crawlers on high-yield pages [32]), but also adapts to the evolving Web – as sites change or new sites emerge, the ML models and context data quickly adjust the crawling strategy. The result is a scalable, **autonomous crawling orchestration** system at an academic level of sophistication, but engineered for real-world deployment.

---

[1] [2] [12] [13] [14] [18] [19] [20] [21] [22] [32] ML-Driven Crawl Scheduling - Predicting High-Value Pages Before You Visit | ScrapingAnt
https://scrapingant.com/blog/ml-driven-crawl-scheduling-predicting-high-value-pages

[3] TorchServe — PyTorch/Serve master documentation
https://docs.pytorch.org/serve/

[4] NVIDIA Triton Inference Server
https://docs.nvidia.com/deeplearning/triton-inference-server/user-guide/docs/index.html

[5] [6] [7] [25] [30] [31] Distributed Frontera: Web Crawling At Scale
https://www.zyte.com/blog/distributed-frontera-web-crawling-at-large-scale/

[8] pyg-team/pytorch_geometric: Graph Neural Network Library for ...
https://github.com/pyg-team/pytorch_geometric

[9] Graph Transformers: Revolutionizing Graph Neural Networks
https://www.linkedin.com/pulse/graph-transformers-revolutionizing-neural-networks-sarvex-jatasra-cxuwc

[10] [11] [15] [16] [17] Retrieve & Re-Rank — Sentence Transformers documentation
https://sbert.net/examples/sentence_transformer/applications/retrieve_rerank/README.html

[23] [26] Neural Prioritisation for Web Crawling
https://arxiv.org/html/2506.16146v2

[24] [27] [28] [29] robots.txt - Wikipedia
https://en.wikipedia.org/wiki/Robots.txt