ChatGPT

UConn Scraper: Enhanced Future Development Plan

Last Updated: 2025-10-01 (Revision 2)
Status: Focused on next-phase improvements post immediate priorities
Completed Items: See IMMEDIATE_PRIORITIES_COMPLETED.md

Table of Contents

Overview & Strategic Goals

Context: With the immediate high-priority fixes completed (as of October 2025) [1], the UConn web scraping pipeline is stable and functional. The next phase focuses on **expanding capabilities** and **refining quality** to transform this pipeline into a **comprehensive, cutting-edge data collection system**. The ultimate goal is to build a **universal knowledge base** of UConn content that can power advanced applications such as student Q&A chatbots and domain-specific language model training.

Key Objectives:

- **Maximize Coverage:** Ensure Stage 1 finds *every possible URL* within `uconn.edu`, including dynamic content and hidden endpoints. This means handling JavaScript-heavy pages, APIs, and external data sources beyond basic HTML links.
- **Accurate Filtering:** Enable Stage 2 to not only check availability but also **classify URLs by type and content**, filtering out dead links or irrelevant resources. This stage will gatekeep Stage 3 so that only valuable pages (HTML content, PDFs, media, etc.) proceed for deep processing.
- **Deep Content Enrichment:** Elevate Stage 3 to extract meaningful information from each page. This includes textual content, metadata, entities, categories, and possibly converting non-HTML content (PDFs, images, videos) into text. The enriched output should be **model-ready**, meaning it's suitable for training or feeding into an index for search/Q&A.
- **Data Quality & Relevance:** Introduce measures of content importance, recency, and quality. Heavily information-rich pages (e.g. course descriptions, policies) should be flagged or scored higher than sparse pages. Outdated content should be noted or potentially down-weighted.
- **Scalability & Maintainability:** Refactor legacy or UConn-specific code into more generalized, modular components. Remove hard-coded assumptions (like fixed domain names, paths) [2] [3] to prepare for potential multi-domain use or open-sourcing, all without deviating from our current focus on UConn.
- **Observability & Robustness:** Implement thorough monitoring, logging, and testing so that the pipeline can run autonomously for long periods (e.g. entire site crawl) with minimal intervention, and any issues are caught and alerted promptly.

These goals align with building a **state-of-the-art scraping pipeline** that not only collects data but understands and organizes it – essentially laying the groundwork for a UConn knowledge graph or fine-tuned LLM.

Priority Focus Areas

◉ High Priority – Address Immediately

- **Dynamic Content Crawling:** Integration of a headless browser for JavaScript-rendered content and AJAX endpoints. High impact on coverage.
- **Content-Type Handling:** Expand Stage 2/3 to handle PDFs and media (images, videos) – ensure valuable data isn't missed just because it's not HTML.
- **Configuration & Legacy Cleanup:** Replace remaining hard-coded values (domains, file paths) with config entries [3] ; validate configurations thoroughly to prevent mis-runs.
- **Test Coverage & Reliability:** Boost automated test coverage to ≥80% and implement new tests for recently added features (e.g. persistent dedup, checkpoint resume).
- **Performance Tuning:** Optimize I/O and concurrency (async improvements, memory usage) to comfortably crawl millions of pages without crashing or slowing dramatically.

◉ Medium Priority – Plan & Implement Next

- **Adaptive Throttling & Heuristics:** Fine-tune Stage 1 discovery heuristics with feedback loops from Stage 2 (reduce "noisy" URL generation) [4] .

- **Advanced NLP Integration:** Incorporate summarization, advanced NER (e.g. using Transformers), and content classification into Stage 3 enrichment.
- **Monitoring Dashboard:** Develop a simple web dashboard to visualize crawl progress, success rates, error rates, and content stats in real time.
- **Database Integration:** Prototype migrating from JSONL/SQLite to a relational DB (PostgreSQL) for storing results and metadata for easier querying and updates.
- **Plugin System:** Begin designing a plugin architecture to allow custom extensions (e.g. plug in a different university profile, or custom content parsers) without modifying core code.

Low Priority / Future – Long Term Ideas

- **Distributed Crawling:** Explore splitting the crawl across multiple machines or processes for load balancing (though UConn alone might not require it, this prepares for multi-domain or very large scale).
- **Multi-Domain (Multi-University) Support:** Abstract domain-specific rules so that the pipeline could crawl other universities with minimal changes (e.g. configurable allowed domains, seed lists, and content tag keywords).
- **Machine Learning Agents:** Investigate using LLMs or autonomous agents to guide the crawler (for example, an LLM that decides which links are most promising or interprets JavaScript code to find hidden URLs).
- **Visual Data Extraction:** Use vision-language models to interpret pages where important info is locked in charts or images (beyond simple OCR).
- **Knowledge Base Integration:** Build a layer on top of Stage 3 output to link related content (e.g. link faculty names to faculty profiles) and potentially feed into a Q&A system (for instance, using retrieval-augmented generation with the scraped data).

The following sections break down these focus areas into specific improvements, with reasoning and step-by-step solution approaches for each.

Testing & Validation Enhancements

High code quality and reliability are foundational. While basic tests exist, we need to broaden coverage and ensure the pipeline is **correct under all scenarios**. This includes both **unit tests** for individual components and **integration tests** for end-to-end behavior.

1.1 Boost Code Coverage & Test Depth (Priority: High, Impact: Medium)

- **Current State:** The project has a test suite (under `tests/`) covering core functionality (URL normalization, storage, pipeline orchestration, etc.) [5] . However, coverage can be improved especially as we add new features (target 80%+ coverage).
- **Improvements:**
- Integrate `pytest-cov` to measure coverage and fail CI if coverage drops below a threshold (initially 80%) [6] .
- Add tests for edge cases in URL handling. For example, implement **property-based tests** for the `normalize_url` function (to ensure idempotence and correctness on random URL inputs) [7] .

- Expand tests for the **metrics collector** and any **global state** to ensure tests remain isolated. E.g., test that running two pipeline instances does not bleed metrics between them (the current global metrics collector is not thread-safe [8], so at least ensure sequential runs don't interfere).
- **Test naming & organization:** Standardize naming (e.g. test function names include what they test) and use fixtures to avoid repetitive setup code. This was noted as an internal improvement (e.g. ORG-057 in original plan) and will make maintenance easier.
- **Step-by-Step:**
- Set up coverage in `pytest.ini` and update CI configuration to record and report coverage trends [6].
- Write new tests targeting untested modules (`alerts.py`, `config.py`, spider logic for dynamic discovery, etc.). For example, simulate a small crawl with a dummy HTML containing dynamic scripts and verify the spider discovers expected URLs.
- Use **monkeypatch or dependency injection** in tests to isolate and simulate certain conditions – e.g., monkeypatch `URLCache.add_url_if_new` to throw an exception and ensure the spider handles storage failures gracefully.
- Ensure tests do not leave behind artifacts. Update pytest configuration to use a temporary directory for any files (or ensure teardown removes them) so that running tests doesn't pollute `data/` in the repo (this was identified as ORG-055 issue).
- Track coverage after adding tests; identify any critical logic still untested and address it.

Outcome: Improved confidence in each release – tests will catch regressions early. The pipeline's reliability increases, which is crucial as we add complexity.

1.2 Configuration Validation System (Priority: High, Impact: Medium)

- **Current State:** The config loader (`Config` in orchestrator) loads YAML and applies some validations (required fields and simple ranges) [9] [10]. There is no formal schema or deep validation (e.g., if a key is misspelled or of wrong type, it might slip by).
- **Problems:** Misconfiguration could lead to subtle bugs (e.g., if someone sets `stages.discovery.max_depth` as a string `"5"` by mistake, does it get caught?). Also, new config options might not be validated at all.
- **Improvements:**
- Implement a **schema-driven validation** using a library like `voluptuous` or `pydantic` or simpler approach with custom checks. Define the expected types and allowed ranges for each config key (building on the partial checks we have) [11].
- Validate that no **unknown keys** are present – to catch typos that could silently be ignored. (For example, if user writes `maxDepth` instead of `max_depth`, we want to warn or error.)
- Integrate this validation in the startup sequence (fail fast if config is invalid). The `ConfigValidationError` mechanism exists [12] [13]; extend it with more rules.
- **Step-by-Step:**
    1. Write a formal schema dictionary or use `pydantic.BaseModel` for config. Include complex structures (like lists of channels in alerts, or lists of heuristics strings for Stage 1).
    2. Enhance `_validate_config()` in `Config` to check types of nested fields. For example, verify that `stages.validation.max_workers` is an int and within a sensible range (already partially done) [10].
    3. Add unit tests for `Config` using sample YAML files: one valid, and several invalid (missing fields, wrong types) to ensure validation triggers.

4. Document the configuration schema in `docs/` so users/contributors know the options (link this documentation in error messages for ease of debugging).
- **Reasoning:** A robust config system prevents runtime errors caused by simple mistakes. It's easier to fix a config error on startup than to debug a failing crawl that ran for an hour with a wrong setting. As we add more config options (for new features, toggles, etc.), this becomes even more important.

1.3 Data Validation Between Stages (Priority: Medium, Impact: Medium)

- **Goal:** Ensure that the *output of each stage meets the expectations of the next*. Given our pipeline writes JSONL at each stage, we should verify consistency (e.g., Stage 1 outputs required fields for Stage 2, Stage 2 marks validity correctly for Stage 3).
- **Approach:**
- Define formal **schemas for Stage outputs** using a schema library or dataclasses with validation. We already have dataclass definitions (`DiscoveryItem`, `ValidationResult`, `EnrichmentItem` in `schemas.py`) [14] [15] [16] . We should enforce these on the JSONL output to catch any missing or malformed entries.
- Use libraries like `pandera` or `jsonschema` to validate a sample of the output periodically. For instance, after Stage 1 finishes, run a quick schema check on `discovery_output.jsonl` to ensure each record has `source_url`, `discovered_url`, etc., and no unexpected fields.
- Insert validation in the pipeline orchestrator: after writing Stage 1 data (or before feeding to Stage 2), perform an assertion or log a warning if validation fails. Similarly for Stage 2 -> Stage 3 handoff.
- Add **integrity assertions**: e.g., every `url_hash` in Stage 2 should have come from a Stage 1 `url_hash` (we can track or spot-check a few), and every Stage 3 item's `url` should match a Stage 2 `url` that was marked valid.
- **Failure Handling:** If a validation fails (e.g. Stage 2 output is missing a field due to a bug), decide on behavior: maybe log an error and skip those records, or halt the pipeline if it indicates a serious logic error. In a development environment we'd likely throw, in production perhaps skip with alert.
- **Step-by-Step Example:** Implement a function `validate_stage_output(stage: int, filepath: str)` that reads the JSONL and checks schema. Use it in orchestrator:
- After Stage 1: ensure each record has `discovered_url` and that field is a valid URL string belonging to UConn (we can reuse `is_valid_uconn_url` for quick domain check [2]).
- After Stage 2: ensure all `status_code` are integers, `is_valid` is boolean, etc., and maybe that `url` fields are unique (they should be, given dedup).
- After Stage 3: ensure each entry has either text content or flags indicating why not (e.g., if in future we include PDFs, an enriched item might have text_content empty but perhaps a flag that it was a PDF — in which case we should have processed it differently).
- Write tests with intentionally malformed JSONL inputs to verify the validation catches issues.
- **Benefit:** Early detection of inconsistencies. This prevents, for example, Stage 3 from silently skipping items because a key it expects is missing. It also guards against future refactoring mistakes that could break the contract between stages.

1.4 Resilience Testing & Fault Injection (Priority: Medium, Impact: High)

To truly trust the pipeline for long runs, we must test how it behaves under failure conditions: - Simulate network failures in Stage 2 (using pytest to monkeypatch `aiohttp.ClientSession` to throw timeouts or errors) and ensure our retry with backoff works and eventually flags the URL as failed after max retries [17] [18] . - Simulate partial pipeline restarts: run Stage 1 for a short time, stop it, then run again and see if it

correctly resumes using the checkpoint/URL cache (no duplicate URLs, continues crawling new ones). Similarly test Stage 2 resume using its checkpoint file. - Force a **graceful shutdown** signal (SIGINT) during each stage and verify that: - Stage 1 spider closes and writes all pending URLs (the Spider's `closed` handler logs a summary [19] [20] – ensure this triggers). - Stage 2 stops after finishing the current batch and writes a checkpoint. - Stage 3 stops the Scrapy process gracefully (this one might be trickier to simulate, but we can at least test the pipeline orchestrator's ability to terminate the subprocess if needed). - These scenarios ensure our pipeline can handle real-world interruptions (manual stop or crash) without data loss or corruption. It ties into improvements in [Orchestration & Pipeline Resilience](#).

By intensively testing and validating at each step, we lay a reliable foundation so that subsequent sections' enhancements (like crawling more content types or adding ML) will be built on **solid ground**.

Stage 1 – URL Discovery Improvements

Stage 1 is the **entry point** of the pipeline, responsible for breadth-first crawling and finding as many URLs as possible on the UConn domain. Currently, Stage 1 uses Scrapy and custom heuristics to find dynamic links. We aim to make Stage 1 even more powerful and flexible, while reducing noise (unnecessary or duplicate URLs).

2.1 Dynamic Tuning & Throttling Heuristics (Priority: High, Impact: High)

- **Current Mechanism:** Stage 1's `DiscoverySpider` scrapes pages and extracts URLs via:
- Scrapy's built-in LinkExtractor for HTML anchor links (with a deny list for certain file types) [21] .
- Custom parsing of inline scripts for AJAX endpoints, JSON blobs, and potential pagination links [22] [23] [24] . It uses confidence scores (e.g., 0.9 for certain AJAX, 0.6 for generic patterns) and a `discovery_source` label (like `"ajax_endpoint"`, `"json_blob"`) [25] .
- A simple breadth-first strategy with a max depth (currently 3 by default) [26] [27] .
- **Problems:**
- **Overgeneration:** The dynamic regex can sometimes capture URLs that are not actual endpoints (e.g., template strings or dummy links in scripts). These are low-confidence and could clutter Stage 2 with many 404s or irrelevant URLs.
- **Lack of Adaptation:** The spider doesn't currently adjust if a particular site section is "noisy." For example, if one page produces hundreds of low-confidence URLs that mostly 404, we should throttle or skip similar ones.
- **No Feature Flags:** All heuristics run unconditionally. If we know some are misbehaving, we cannot turn them off without altering code.
- **Improvements:**
- **Feature Flags for Heuristics:** Implement settings to toggle each class of dynamic discovery. For instance, `enable_json_discovery`, `enable_ajax_regex`, `enable_pagination_guess`. These can default to True, but having them allows quick experimentation and disabling if needed [28] .
- **Throttling & Limits:** Introduce counters to detect if a heuristic is yielding too many results that turn out invalid. For example, track how many "ajax_endpoint" URLs found at confidence 0.6 later get 404 or filtered out by Stage 2. If that ratio is high, the spider could temporarily pause using that heuristic on deeper pages. This **feedback loop** from Stage 2 can be implemented in subsequent runs: Stage 2

could log patterns of URLs that were invalid (like certain repeating query parameters), and Stage 1 can read that and adjust.

- **TTL Cache for Pagination:** Many page URL patterns include numeric pagination or offsets. We can generate pagination URLs (as done in `_generate_pagination_urls`) but should ensure we don't generate beyond a reasonable range. Implement a TTL or limit on how far these guesses go (maybe stop after 10 pages unless some are confirmed valid). Also cache the fact that "page=5" was last valid, so we don't try page=100 repeatedly across runs.
- **Adaptive Depth per Section:** Depth is global now (e.g., 3). If certain subdomains are very content-rich, we may want to go deeper there, versus shallower elsewhere. We could refine this by domain or path (not trivial, but something to consider if needed).
- **Step-by-Step:**
- **Feature Flags:** Add config entries under `stages.discovery.heuristics` (in YAML) for each heuristic block (script regex, JSON search, etc.). Check these flags in the spider before applying that logic. E.g., wrap the JSON script parsing loop in `if self.settings.getbool('ENABLE_JSON_DISCOVERY', True): ...` [28] .
- **Logging & Metrics:** Enhance logging when a dynamic URL is found. Include `discovery_source` and confidence in the log line. We already attach those fields to items [29] ; also update metrics (maybe increment counters for each source type, which is partly done in `_dynamic_discovery_stats`) [30] .
- **Tune Confidence Usage:** Currently if `confidence < 0.6` they increment a `low_quality_count` [31] . We can decide that if a page yields too many low confidence URLs, we slow down or stop processing low-conf links from that page. For example, after processing a batch of dynamic candidates, if `low_quality_count` is a significant portion, set a short delay or skip deeper crawling from that page. This requires careful design to not miss rare good links, but prevents explosion of junk URLs.
- **Feedback from Stage 2:** Implement Stage 2 to gather simple stats – e.g. count of URLs by `discovery_source` that resulted in 404 or other errors. This could be added to the `ValidationResult` as a field or just done in memory. After Stage 2, output a small report (or update URLCache with a flag) marking which source types had high failure rates. Next crawl, Stage 1 can reduce confidence or outright disable those sources if they appear again. (This is a more advanced adaptive loop, perhaps a medium-term goal).
- **Test the adjustments:** Use a controlled environment (maybe a subset of UConn site or a test site) to generate scenarios – e.g. an HTML page with a fake script containing many URLs like `"test/api/ endpoint/xyz123"` that will 404. See if the spider can identify and throttle these.
- **Expected Outcome:** A more intelligent crawler that finds *more real content* and less noise. By completing the throttling implementation [28] and adding adaptivity, Stage 1 remains comprehensive but efficient. This directly improves Stage 2 throughput (less time wasted on invalid URLs) and Stage 3 relevance.

2.2 Browser-Backed Discovery (JavaScript Rendering) (Priority: Medium, Impact: High)

- **Rationale:** Some UConn pages may rely on client-side rendering (e.g., React applications, or content that appears only after user interaction like clicking "Load more"). The current spider will miss such URLs. The plan is to integrate a **headless browser** (likely Playwright or Selenium) to handle these cases [32] .
- **Scope:**

- Use **Playwright** (preferred for its Python async support) to render pages where static scraping might miss content. Potential triggers to use Playwright: presence of significant `<script>` tags, known URLs (like `/events` pages that load items via XHR), or a config list of paths that require dynamic loading.
- While rendering, **capture network requests**. This will surface any AJAX calls (their URLs and maybe responses) that a page makes. For instance, if visiting a page triggers requests to `/api/events?page=2`, we capture those URLs.
- **Intercept user interface elements:** If a page requires clicking "Load More", use Playwright to click it programmatically until no more content loads, gathering all new URLs that appear.
- **SPA support:** In Single Page Apps, navigation may not cause full page loads. We might need to read the DOM after certain delays or after simulating scroll.
- **Design:**
- Introduce a **Stage 1.5 or extended Stage 1 mode**: The discovery spider could spawn a browser for specific pages. Possibly implement as a separate spider class or a utility function. For example, when `DiscoverySpider` sees a URL that matches certain criteria, instead of normal requests, it could hand it off to a Playwright routine.
- Alternatively, have a *post-processing step* after Stage 1: a small module that reads the Stage 1 output and for any URLs flagged as requiring JS (perhaps marked in discovery metadata or via a naming pattern like `#dynamic`), it launches a browser to extract further links.
- **Resource Management:** Browser automation is expensive, so we must use it sparingly:
    - Limit the number of pages rendered (maybe only those that we strongly suspect have hidden links).
    - Limit time per page (e.g., 10 seconds max to render and interact).
    - Ideally run browsers concurrently up to a small number (like 2-3 at a time) to avoid CPU overload.
- **Technical Steps:**
    1. Integrate Playwright: add optional dependency and initialization. On first use, launch a browser instance (maybe in headless mode) with context.
    2. For a target page, use `await page.goto(url)` and possibly `await page.wait_for_load_state('networkidle')` to ensure initial AJAX have loaded.
    3. Execute `page.content()` and feed it to something like Scrapy's `LinkExtractor` to get any anchors that weren't present in original HTML (though in a fully dynamic page, initial HTML is minimal).
    4. Use `page.on("request")` event to log outgoing requests URLs. Filter those to same-domain and plausible endpoints (e.g. containing `.uconn.edu`).
    5. Optionally, use `page.evaluate()` to scroll or click as needed. We might need a library of typical actions (scroll to bottom, click any button with text "More" or icons).
    6. Collect discovered URLs from these actions, and add them back into the Stage 1 URL list (either yield them in the Scrapy spider or write them to Stage 1 output as new DiscoveryItems with a special source like `browser_render`).
- **Testing:** Identify a known dynamic page on UConn (if any exist, e.g., some event calendars or map applications). Test manually with Playwright to ensure we can capture hidden URLs. Write integration tests with a dummy page (e.g., an HTML that loads content via JS after 3 seconds).
- **Expected Gains:** This will allow crawling sections that were previously unreachable. For example, if UConn had an infinite-scrolling news feed or a Vue.js app for course listings, we will now get those URLs. It's high effort, but the impact on completeness is high. We should see an increase in unique

URLs discovered (especially if currently our Stage 1 stats show some sections with zero links due to heavy JS). This aligns with the priority of **maximizing coverage**.

2.3 External Intelligence & Site Feeds (Priority: Low, Impact: Medium)

- **Description:** Beyond crawling the site itself, we can incorporate external sources of URLs:
- **XML Sitemaps:** Check if `uconn.edu/sitemap.xml` or similar exists and parse it for URLs (often very comprehensive). This could be integrated as a pre-crawl step.
- **RSS/Atom Feeds:** Universities often have news feeds or event feeds. Auto-discover `<link type="application/rss+xml">` in pages or guess common feed URLs. Parse those feeds to get article URLs.
- **Site Search results:** Use UConn's own search (if available via an API or by scraping search results pages for queries like "site:uconn.edu"). This can reveal pages that are not easily reached by navigation. Caution: this can be expansive, but maybe querying for specific keywords (like academic terms, department names) could surface hidden pages.
- **Historical archives:** Consider using the Internet Archive or Common Crawl if we suspect some URLs exist historically but not linked currently. For example, if something was removed from navigation but still live, a search on Common Crawl might find it. This is advanced and potentially out-of-scope for near term, but worth noting for completeness.
- **Implementation Ideas:**
- Add a new component to Stage 1 that attempts these external sources. Possibly run it *before* the main crawl: e.g., fetch known sitemaps and seed those URLs into the crawler (so dedup logic will skip them if discovered via navigation anyway, but we ensure they're covered).
- For RSS feeds, maintain a small list of likely feed URLs or discover from the homepage HTML `<head>`.
- For site search, if an API is available (some sites have a JSON search endpoint), use it responsibly. If only HTML search, this might be more trouble (also can appear as bot behavior – use carefully).
- Ensure to mark any such URLs with a distinct `discovery_source` ("sitemap", "rss_feed", "external_search") so we can track their origin.
- All these would also respect domain rules (only bring in uconn.edu links).
- **Benefit:** This can capture pages that are otherwise isolated (orphan pages). Impact is medium because UConn likely maintains good navigation, but it could help find things like PDFs or old pages not linked anymore. Given it's lower priority, this can be scheduled after core improvements.

2.4 Enhanced Static Analysis & URL Parsing (Priority: Medium, Impact: Medium)

- **Goal:** Catch more URLs from the HTML content itself (without executing JS):
- Some web pages include URLs in places like data attributes or even comments which our current approach might not catch.
- **Improvements:**
- Extend the LinkExtractor or supplement it with custom parsing for:
  - **Data attributes:** We have a list (`data-url`, `data-src`, etc.) in config [33], but currently the spider doesn't explicitly scrape those. Implement a scan of the HTML for any attributes matching our list and treat their values as URLs to normalize. This could be done via Scrapy's `response.xpath` looking for `//@data-url` etc., then feeding through `_normalize_candidate` [34].
  - **HTML Comments:** Search for `<!-- ... -->` content where URLs might be present (rare, but possible in embedded scripts or templates).

- ○ **Regex for URL-like strings:** Already, `SCRIPT_URL_PATTERN` regex looks for `://` or `"/path"` in scripts [35]. We might add a similar regex for styles or other text.
- ○ **Form actions and meta refresh:** Extract URLs from `<form action="...">` and `<meta http-equiv="refresh" content="URL=...">` if any.
- **Improve** `_looks_like_api_endpoint`: It currently checks for certain substrings in URL path or query (like `/api/` or `.json`) [36]. Expand this to catch more patterns (maybe `/v1/`, `/services/`, etc.) and perhaps integrate it with the confidence scoring (maybe API endpoints get a slightly different handling).
- **Adaptive Regex Patterns:** As we crawl, we might encounter common patterns like `page=1` or `index=2`. Perhaps automatically generate some variations (like increment numbers) with caution. This is partially what `_generate_pagination_urls` does using `_looks_like_api_endpoint` as a trigger [37]. We should verify this works well and maybe refine:
  - ○ If an endpoint looks like `page=1`, generate page 2,3 etc. up to some max or until Stage 2 stops finding them valid.
  - ○ If an endpoint has a `offset=0` param, attempt offset=10,20... to mimic pagination.
  - ○ This could be done by a small function analyzing query strings of discovered URLs.
- **Why it matters:** These enhancements ensure no stone is left unturned in Stage 1. Even without the heavy browser approach, we squeeze as much as possible from static HTML. For instance, if a page has a `<div data-feed="/events/list">`, our crawler should pick up `/events/list` as a URL to try.

By implementing the above improvements in Stage 1, we expect: - An increase in **unique URLs discovered** (the metric `Unique URLs found` in the spider summary [38] should go up). - A controlled or reduced number of **junk URLs** (tracked by `Duplicates skipped` and the dynamic discovery stats for low_quality_count [31]). - Better **coverage of dynamic content** due to the Playwright integration (reflected by new URLs that were previously missing, possibly identified by comparing crawl results from before and after). - Stage 1 will remain efficient thanks to throttling and feature flags: we can tune it in config if it's doing too much or too little.

Stage 2 – URL Validation & Classification

Stage 2 currently takes the list of discovered URLs and performs HTTP HEAD/GET to verify reachability, get content type, etc., marking `is_valid` true for HTML pages (status < 400 and content-type HTML) [39]. We want to expand Stage 2 into a smarter filter that not only checks "is it alive and HTML" but also **classifies content types and enriches metadata** for Stage 3's decision making.

3.1 Robust Retry & Error Handling (Priority: High, Impact: High)

- **Current State:** The validator uses up to 3 retries with exponential backoff for timeouts or certain errors [40] [18]. It doesn't differentiate error types beyond that. All failures ultimately produce a `ValidationResult` with `is_valid=False` and an `error_message` string [41] [42].
- **Issues:**
- Certain errors might be transient (network hiccup) and worth more aggressive retry, whereas others (e.g., 404 or DNS not found) are permanent for that run.
- There is no **jitter** in backoff or circuit breaker for repeated failures hitting the same domain.

- If one domain is very slow or down, we don't have a way to back off that domain specifically (could tie up workers).
- **Improvements:**
- Implement **jittered exponential backoff**: instead of fixed 1,2,4 second delays, add randomness (e.g., ±10%) to avoid synchronized retries if many requests time out at once.
- Introduce a basic **circuit breaker**: e.g., if 10 consecutive requests to the same host fail quickly (timeouts or connection refused), then perhaps skip further attempts to that host for a while (mark them as failed with a generalized error like "Circuit breaker tripped"). This prevents wasting time on a host that is down.
- **Classify errors**: In the `except aiohttp.ClientError` clause [43], inspect `exc`. If it's say DNS failure, mark error_message accordingly (we might already get an exception name in message). For HTTP errors (though aiohttp would not throw for HTTP status, we handle those differently), if a HEAD gets 405 (method not allowed), we fall back to GET – which we do implicitly by our logic.
- Extend the `ValidationResult` to optionally include the **redirect chain** (we have a placeholder in schema [44]). aiohttp's `response.history` can give redirects. This could help Stage 3 understand if a URL was redirected to a different one (which might indicate an alias or moved page).
- **Step-by-Step:**
    1. Modify `_validate_with_session` loop [17] to apply jitter: e.g., `await asyncio.sleep(retry_delay * (2 ** attempt) + random.uniform(0, 0.5))`.
    2. Track failure counts per domain: maintain a dict in URLValidator (not persisted, just runtime) for consecutive failures. If a threshold is reached, skip further attempts for that domain for a certain time. (Could integrate with URLCache or separate static memory.)
    3. Implement logging at WARNING level when circuit breaker triggers for a domain, so we know it happened.
    4. Ensure that when marking is_valid False, we preserve as much info as useful. For example, if a URL consistently times out, maybe note `"Timeout after 3 retries"` in error_message (currently it just says "Request timeout" on final attempt) [18] [41].
    5. For known patterns (maybe specific to UConn infrastructure): if a request returns a login page or some forbidden message instead of content, we might treat it as special (though likely not applicable since public content is accessible).
    6. Add unit tests for the retry logic, possibly by monkeypatching `session.head` to raise `asyncio.TimeoutError` a couple times then succeed, to ensure it does retry.
- **Impact:** More robust validation means Stage 2 won't prematurely drop URLs that might succeed on second try, and also won't waste too much time on hopeless cases. The error classification also means Stage 3 (or any analysis) can differentiate *why* something failed – which could guide future enhancements (for example, if many failures are SSL certificate issues, maybe we need to adjust SSL settings or trust store).

3.2 Content-Type Handling & Non-HTML URLs (Priority: High, Impact: High)

One major enhancement: **don't ignore non-HTML content**. Currently, Stage 2 only marks `is_valid=True` for HTML pages [39], effectively filtering out PDFs, images, etc., from Stage 3. While it's sensible to skip images for text analysis, PDFs and possibly certain document formats likely contain rich text we want.

- **Plan:**

- Expand the definition of "valid" beyond just HTML. Introduce categories like:
  - **HTML page (valid for Stage 3 HTML pipeline)** – what we already have.
  - **PDF or Document (valid for Stage 3 document pipeline)** – e.g., content types `application/pdf`, `application/msword`, etc.
  - **Media (audio/video)** – probably skip or handle differently (we might not "enrich" an MP4 file, but could note it).
  - **Other** – if not recognized, still mark as valid? Perhaps not, to avoid binary junk.
- To implement this, we can:
  - Add a field in ValidationResult like `content_category` (or reuse `content_type` but parse it). For example, if `content_type` starts with `application/pdf`, mark `content_category="pdf"` and set `is_valid=True` *if* we intend to handle PDFs in Stage 3.
  - Alternatively, overload `is_valid`: keep it boolean but allow True for other content we plan to handle. Then ensure Stage 3 knows what to do based on content_type.
- **Stage 3 adjustments:** We will need to modify Stage 3 to accept these new "valid" URLs. Possibly, feed Stage 3 all URLs with `is_valid=True`, whether HTML or PDF. In the EnrichmentSpider, detect the content type (we can pass it via `validation_data` meta [45]) and take different actions:
  - If HTML: fetch and parse as usual.
  - If PDF: fetch (or we may have to fetch content if Stage 2 only HEADed it? Actually Stage 2 does GET for non-HTML because HEAD would have returned None result; our logic falls back to GET if HEAD wasn't sufficient [46], so Stage 2 likely downloaded a snippet or whole body for content length).
  - We might not want to re-download in Stage 3. Maybe Stage 2 should *not* fully GET large PDFs, just HEAD. It currently will do GET if HEAD says not HTML (since `_evaluate_head_response` returns None, then `_perform_get` runs) [17] [47]. That means Stage 2 might actually download the PDF bytes just to measure length. This is inefficient. Perhaps optimize: for known large content types, we can rely on Content-Length header from HEAD (if present) rather than GET. But aiohttp HEAD might not return Content-Length due to no body. Actually, the code tries HEAD first and only does GET if HEAD wasn't "sufficient" (not HTML content triggers None result, which then does GET) [46]. We might want to change that logic: If content_type is PDF and status 200, we *could* decide we have enough info (we know it's there, status 200, length via header) and avoid GET. This may require adjusting `_evaluate_head_response` to accept certain content types as valid. Or a simpler path: set `is_valid=True` in `_evaluate_head_response` if status 200 and content-type is in {html,pdf,doc,etc.}. Then Stage 2 will mark PDF as is_valid without doing GET. **This is a key change** for efficiency.
  - Stage 3 handling of PDFs: Use an external library (like PyMuPDF or PDFMiner) to extract text. Possibly integrate an OCR step if PDF is scanned (using an OCR like Tesseract or AWS Textract – but that might be overkill initially). For now, focus on text-based PDFs which many university PDFs are (policies, forms, etc.).
  - This could be implemented as a separate pipeline or within `EnrichmentSpider.parse`: detect PDF by `response.headers['Content-Type']` or by checking file extension in URL. If PDF, instead of `response.xpath`, we would get `response.body` (the PDF bytes). We might not want Scrapy to try to parse PDF as HTML; it would give gibberish. We can either:
    - Use Scrapy's `HttpResponse` body and feed it to a PDF parser on the fly, yielding an EnrichmentItem with the extracted text.

- - Or have Stage 3 skip downloading and directly call a PDF processing function that reads from disk if Stage 2 already saved it (Stage 2 doesn't save content bytes, only length).
    - Likely simplest: in Stage 3, make a second request for the PDF via something like `response.follow(..., self.parse_pdf)` or even using requests outside Scrapy. However, this doubles download. Perhaps accept it for now unless we engineer Stage 2 to save PDF content (not currently done).
  - Because PDF processing is potentially slow, consider doing it synchronously in the spider parse (which is already synchronous per item). If it's too slow, we might need to offload to an asynchronous task or process (could be future improvement if needed).
  - Extracted text from PDF can be put into `text_content` of EnrichmentItem, or maybe a separate field like `pdf_text_content` (but likely reuse text_content for consistency).
  - We should also set `content_tags` or entity extraction on that text as we do for HTML.
  - Images: likely ignore or mark as such (we can skip enriching images beyond maybe noting they exist – Stage 2 could just mark them invalid to filter out, or we keep them out of Stage 3 entirely for now).
  - Videos: probably skip for now. If we want to handle them, Stage 3 could possibly use an external call to a speech-to-text (like Whisper) if it's a video with spoken content. This is advanced and may be tackled later (see Advanced Features).
- **Outcome for Stage 2:** After changes, Stage 2 `validation_output.jsonl` will include entries for PDFs with `is_valid=true` and content_type=application/pdf, content_length etc. Stage 3 will then process them. We'll gain coverage of content previously missed.
- **Step-by-Step Changes in Code:**
- Modify `_evaluate_head_response` in `URLValidator` [48] to return a ValidationResult for certain non-HTML types too. For example, if status 200 and content-type is `application/pdf`, we return a result with `is_valid=True` but maybe mark differently (could set `validation_method="HEAD"` to note we only did HEAD).
- Adjust `_validate_with_session` loop so that if HEAD says content-type not HTML but still returns None (we changed it to return not None for PDFs as above), then we might decide not to GET. If we still do GET, consider limiting download size (maybe set a smaller timeout or a max bytes to read).
- In `EnrichmentSpider.parse`, add logic: `if 'pdf' in response.headers.get('Content-Type','').lower(): ...` (or check file extension via `urlparse`). If PDF:
  - Use a PDF parsing library to extract text. (Add it to requirements if not already, e.g., `PyMuPDF` for performance.)
  - Populate `title` (maybe from PDF metadata if available) or use the file name as title.
  - Populate `text_content` with extracted text (cap at some length perhaps).
  - We might also set a flag in `content_tags` like `"pdf"` or set `has_pdf_links=False` but maybe add a new field `source_is_pdf=True` for clarity (or rely on content_type field which is already stored [49]).
  - Ensure `entities` and `keywords` extraction is run on this text as well (our `extract_entities_and_keywords` likely can handle it).
  - Possibly skip `response.xpath('//body//text()')` for PDF, obviously.
- Update `EnrichmentItem` schema if needed to accommodate anything special (perhaps not needed, we can reuse fields).
- Test with a known PDF URL. UConn likely has PDFs (e.g., academic calendars, forms). Manually run Stage 2 on a PDF URL to ensure it's marked valid and Stage 3 prints out extracted text.

- Monitor performance: parsing many PDFs could slow Stage 3; consider adding a configuration to limit PDF processing (e.g., max size of PDF to parse or an option to turn it off if unwanted).
- **Note:** We should explicitly document that **images and binaries are still skipped**. Stage 2 deny list (in LinkExtractor) already prevented many image links from Stage 1 [50] . But if any slip through, Stage 2 should probably mark them invalid to avoid pointless Stage 3 attempts. We might extend the deny list or handle by content-type: e.g., if `image/png` , do not mark valid (which is current behavior since not HTML).
- **Benefit:** By not dropping PDFs and other text-bearing documents, the pipeline's final dataset becomes much richer. For instance, PDFs of research reports or policy documents could answer many student questions but were previously missing. This aligns with the goal of an **in-depth comprehensive analysis of every aspect of the university** – including content locked in documents.

3.3 Importance & Freshness Metrics (Priority: Medium, Impact: Medium)

Stage 2 can do more than pass/fail URLs; it can start assessing them for importance and content size, feeding hints to Stage 3 on how to prioritize or handle them:

- **Content Length & Word Count:** We already capture `content_length` (bytes) in ValidationResult [51] . We can estimate if a page is content-heavy by content_length. Stage 2 could set an `importance_score` based on size (larger HTML pages might have more info, though not always true). Alternatively, Stage 3 will compute `word_count` after extracting text [52] , which is more accurate. Perhaps Stage 2 can simply flag very small pages (content_length < some threshold) as likely low content.
- **Freshness:** If possible, capture `Last-Modified` header in Stage 2. Many pages include this in HTTP headers. We could store it in ValidationResult (maybe in `server_headers` or a new field). Then Stage 3 (or post-processing) can use it to determine how up-to-date content is. If Last-Modified is too old (or missing), maybe treat the content with caution for current relevance. This is not a direct filter but useful metadata.
- **Dead-end detection:** Stage 2 could mark if a URL appears to be a *dead-end* navigation-wise (though that's more a Stage 1 concept). However, Stage 2 could detect patterns like a page that redirects to a login page (content might have certain keywords like "Login required") and mark those as not useful for Stage 3. This might require looking at response bodies, which Stage 2 currently doesn't do for HTML (only HEAD). If HEAD indicated some login, not likely though (maybe a 302 to a login page).
- **Implementations:**
- Extend `ValidationResult` schema to include optional fields for `last_modified` and perhaps a simple `page_size` or keep using `content_length` .
- In Stage 2 code, after getting response headers, if `'Last-Modified' in response.headers` , capture it.
- For content_length, it's already there. Could do: if content_length <, say, 500 bytes and status 200, that page might be essentially empty (maybe just a nav page). We might still mark it valid, but Stage 3 could use this as a signal to possibly skip enrichment or mark it differently. Alternatively, Stage 3 can itself decide after extracting text if word_count is extremely low, then maybe skip heavy NLP to save time (but still output an item).
- Possibly compute a quick heuristic in Stage 2: e.g., `is_valid but low_content = True` if length < threshold and content_type HTML. Then Stage 3 could check that flag to decide to do minimal processing. This is an optimization for performance more than necessity.

- **Why in Stage 2?** It's our last chance before fetching full content in Stage 3 to decide how to handle the page. Stage 2 is lightweight and can quickly note these things from headers or small GET. This sets the stage for Stage 3 to allocate resources wisely (for example, maybe Stage 3 processes large pages first or uses different NLP for small vs large pages).
- **Steps:**
- Add `last_modified` (string) to ValidationResult dataclass and JSONL output.
- Capture it in `_perform_get` and `_evaluate_head_response` when available.
- Add logic for low content: e.g., `if is_valid and content_length < 500: validation_result['low_content'] = True` (this could go into `network_metadata` or a new field).
- In Stage 3, when reading the validation JSONL, we get `validation_data`. The EnrichmentSpider currently passes the whole validation record in `response.meta['validation_data']` [45]. So inside parse, we can easily access content_length or low_content flag. We could then:
  - If `low_content` is True, maybe skip `extract_entities` to save time (or still do it but it likely finds nothing).
  - If we had an `importance_score` (not calculated yet, but could be something like content_length or maybe Stage 3 will calculate from word_count), Stage 3 might use it for logging or decisions.
  - If `last_modified` exists, we might include it in EnrichmentItem (maybe in `processing_metadata` or add a field for it) so that the final dataset knows the page currency.
- Test on known pages: find a trivial page vs a large page and see that Stage 2 appropriately flags them.
- **Result:** Stage 2 evolves from a blunt filter to a stage that also **annotates URLs** with useful metadata for downstream. This will especially aid when building a search index or deciding which pages to prioritize if resources are limited (e.g., if we ever implement partial crawling or scheduling, we might choose to re-crawl recently updated pages more often, etc.).

Overall, these enhancements turn Stage 2 into both a **bouncer** (keeping bad stuff out) and a **triage nurse** (labeling the good stuff with critical info). By the time URLs reach Stage 3, we'll know for each: "this is an HTML page last updated last month, fairly large content" versus "this is a PDF file, 2MB, from 2018" – information that can guide how we handle it in Stage 3 enrichment.

Stage 3 – Content Enrichment & Analysis

Stage 3 is where the heavy lifting of data understanding happens. It currently uses Scrapy to fetch each valid URL and extracts basic text and NLP entities/keywords via SpaCy [53]. We plan to **significantly enhance Stage 3** to enrich content more deeply and handle different content types.

4.1 NLP Model Upgrades & Entity Extraction (Priority: Medium, Impact: Medium)

- **Current Approach:** Using SpaCy (en_core_web_sm) to extract named entities and keywords (likely via a simple approach, perhaps TF-IDF or similar) [53]. This is fine for a baseline but can be improved:
- The small SpaCy model might not capture domain-specific entities (e.g., "HuskyCT" or building names might not be recognized).
- No summarization or classification is done.

- Keywords extraction might just be splitting or a basic algorithm.
- **Upgrades:**
- Integrate **HuggingFace Transformers** for NLP tasks. For example:
    - Use a pre-trained model like `all-MiniLM-L6-v2` from SentenceTransformers (which we already load for link scoring) [54] to generate embeddings for pages or to improve keyword extraction (via semantic similarity).
    - Fine-tune or zero-shot classify pages into categories (academics, sports, news, etc.). HuggingFace has models for zero-shot classification; we could feed page text and labels like ["Academic", "Sports", ...] to get probabilities. This might be heavy to do for every page though.
    - Named Entity Recognition: Consider using a model like SciSpacy or custom models if we need academic-specific entities (like course codes, faculty names).
- **Summarization:** Use a model (e.g., T5 or BART) to generate a brief summary of each page. This could be stored in `content_summary` field of EnrichmentItem [55]. Summaries help quickly understand pages and are useful for Q&A contexts. Due to performance, we might restrict summarization to certain pages (like very large pages or those identified as important).
- **Topic Modeling or Clustering:** Possibly in post-processing rather than live in Stage 3, but we could assign each page some topics. Alternatively, just rely on the keywords/entities which we already plan to improve.
- **Relation Extraction:** If we want, identify relationships (like person X is in department Y) — this might be beyond immediate scope, but if using an LLM, we could prompt it to extract certain info. Likely too advanced for now, unless some specific need arises (like building a faculty directory mapping).
- **Implementation:**
- These enhancements can be controlled by config flags to enable/disable advanced NLP due to their computational cost (`nlp.transformers_enabled`, etc.). The config has `nlp_enabled` and model names already [56] [57].
- We might incorporate a pipeline using HuggingFace's pipeline API for summarization and classification. Alternatively, use specific libraries (like for summarization, perhaps BART large model).
- Because these models can be slow, consider running them outside Scrapy's parsing loop if possible. Maybe accumulate text and process in batches. However, memory might be an issue if many pages – maybe better to process one by one in parse.
- Perhaps integrate an **async queue for NLP**: e.g., as items are scraped, pass content to a thread/ processing pool running heavy models so the spider isn't blocked. But this adds complexity. Simpler: do it synchronously for now and optimize later if needed.
- **Step-by-Step Example:** For summarization,
    1. Load a summarizer model at spider init (if enabled). e.g., `self.summarizer = pipeline("summarization", model="facebook/bart-large-cnn")`.
    2. In parse, after extracting text_content (and perhaps if `len(text_content) > 500 words` to make summary meaningful), call `summary = self.summarizer(text_content[: 1000], max_length=100, min_length=30)` (maybe summarize first 1000 tokens or so to save time).
    3. Store summary in EnrichmentItem `content_summary`.
    4. Likewise for classification: use a zero-shot model or a small finetuned model to assign categories. Or simpler, use our existing category embeddings approach in a different way (the link scoring with academic categories was a rough proxy [58], but we could output the highest-matching category as a label, and possibly a confidence).

5. Entities: to improve beyond SpaCy small, either use spaCy medium/large model, or use a transformer NER like `dslim/bert-base-NER` for general, plus perhaps add our own logic for university-specific terms. We can also compile a list of key terms (like all academic programs, building names) and do a lookup.
6. Replace or augment `extract_entities_and_keywords` function to incorporate these improvements. Perhaps run spaCy for general entities and also do a lookup for any term in a custom glossary.

- **Validation:** Evaluate these NLP additions on a sample of known content (e.g., does summarizer produce a sensible summary of a course description? Does the classifier correctly flag an athletics news page as "Sports"?). Since this is not easily quantitatively testable without labels, it will be iterative.
- We should be mindful of performance: Summarizing hundreds of pages with BART might be slow. We could restrict summarization to, say, pages above a certain size or those that are likely important (perhaps determined by content tags or depth).
- **Outcome:** Enrichment output will have **richer fields**:
- `content_summary` : a human-readable brief of the page.
- Possibly `academic_relevance_score` if we implement a better measure (right now link-based approach exists; maybe refine using content embedding similarity to academic category embeddings we have) [58] [59] .
- More accurate `entities` list (with proper names relevant to UConn).
- Possibly a new field `categories` or we pack it into `content_tags` / `keywords` . (We have `content_tags` from URL path and predefined tags, which is basic; we might add `predicted_tags` from content).
- **Why it Matters:** For the end goal of **students querying this data**, having semantic annotations and summaries is incredibly useful. For instance, if a student asks "What is the policy on academic misconduct?", the system could use the summary of the "Academic Misconduct Policy" PDF rather than the full text to answer. Entities allow linking different pieces (like linking a professor's name to their profile). Upgrading to modern NLP techniques ensures the pipeline's data is not just raw text, but *structured, meaningful information*.

4.2 Content Quality & Relevance Filtering (Priority: Medium, Impact: Medium)

- **Background:** Not all pages are equally useful. Some pages might be trivial (e.g., navigation pages with just links), duplicates, or outdated. We want Stage 3 to flag or filter such cases so the final dataset is focused and high-quality.
- **Quality metrics:**
- We already considered using content length/word_count as a proxy. Stage 3 now calculates `word_count` [52] for each page. We can use that:
  - If `word_count` is extremely low (like < 50), the page likely has little informational content. Perhaps tag it as `low_content_page` . We might still keep it in output (for completeness), but maybe easier to exclude from training data if needed.
- **Readability scores:** We can compute something like Flesch-Kincaid or similar on the text to gauge readability. University pages might not need that, but it could highlight overly complex pages (maybe less relevant for students? Not sure if needed).
- **Duplicate detection:** After extracting text, Stage 3 could hash or fingerprint the text to catch near duplicates. UConn might have the same content under different URLs (for example, PDFs that have an HTML equivalent, or old archived pages identical to current ones). We could generate a short

hash of the text_content (say, take 100 most significant words or a MinHash). If we see the same hash again, mark the page as duplicate of another.

- This might be better done after crawling all pages (global knowledge), but Stage 3 could do a simple check via the URLCache if we store content hashes there.
- The `URLCache` table has columns for title, word_count, etc. We could consider adding a content_hash after enrichment and query it to see if another URL had the same hash [60] [61]. This is advanced and maybe not urgent, but worth noting.

- **Date relevance:** If we extract dates from content (like news articles dates or policy effective dates), we could mark how current it is. But that's complex to do generically. At least `last_modified` header from Stage 2 is a hint.
- **Actions:**
- If `word_count == 0` (page had no text), consider *not outputting it at all* in Stage 3 or output with a flag indicating empty content. Currently, EnrichmentSpider yields an item even if text_content is empty (with word_count 0). That's fine, but maybe for final training data we'd filter those out.
- If duplicate content is detected, perhaps add a field in EnrichmentItem `is_duplicate_of: <url>` to indicate it.
- Possibly rank pages by some importance and store a score. For now, a simple importance score could be `log(word_count) * (some factor if page is an official policy or main page)`. We don't have straightforward markers of importance beyond size and maybe URL depth.
- Actually, an idea: Use Stage 1 depth as a proxy – pages at depth 0 (seed) or 1 (linked from main pages) might be more important core pages than depth 5. We have `discovery_depth` from Stage 1 in the items [62]. If we propagate that info to Stage 3 (via URLCache or through Stage 2 if we join records), we could include `depth` in EnrichmentItem (the schema `URLRecord` is meant to combine such info [63]). Then we could say anything depth > 3 is likely less central.
- So, incorporate `discovery_depth` into Stage 3's context (the pipeline currently does not merge, but we can query URLCache in Stage 3 to get depth for a given url_hash).
- The `URLCache` already stores discovered_at, validated_at, etc. We might extend it to store `discovery_depth` and maybe a flag if content was enriched.
- **Implementation:**
- Ensure Stage 3 knows `discovery_depth`. We might do a join after the fact to combine Stage1,2,3 data, but possibly simpler: as Stage 3 reads Stage 2 JSONL, it doesn't have depth. If we had used the URLCache to supply Stage 3, it could join internally. Alternatively, since we pass `validation_data` to the spider, we could augment that by looking up the URL in URLCache for depth. Perhaps overkill at runtime – maybe easier to do merging in post-processing.
- Within `parse`, after assembling the item, implement some filters:
  - If `word_count == 0`: log a warning and possibly return without yielding (i.e., skip it). But skipping might complicate if we expect an output for each URL. Instead, yield it but with maybe a tag `content_tags` including "empty-page".
  - If the title or text indicates certain "junk" patterns (like a 404 page content — sometimes a 404 page returns 200 but has "Page not found" text). We could detect keywords "not found" or "error" in a page with low content and treat it as broken. In such case, maybe mark it as not useful or even retroactively we could mark Stage 2 is_valid false for it (but Stage 2 didn't catch because it returned 200). We could simply note it in Stage 3 output as `content_tags: ["error-page"]`.
  - If duplicates: possibly compute a hash for text_content (maybe MD5 of normalized text). Keep a set in memory; if hash seen before, add "duplicate" tag. For safety, also compare lengths or a portion of text to avoid false collisions.

- Add a configuration for **min_content_words** and such, so users can adjust if needed rather than hardcoding thresholds.
- After crawl, we can also do an offline analysis to see how many empty or dupes were flagged and if further action needed.
- **Goal:** By the time Stage 3 finishes, each EnrichmentItem should carry indicators of content quality. This means the final JSONL is ready to be fed into a model training pipeline with minimal cleaning needed: you can filter out low_content or duplicate entries easily. It elevates the overall quality of the dataset and ensures storage isn't wasted on useless data.

4.3 Media Processing (PDFs, Images, Audio) (Priority: Low, Impact: Medium)

(This is partially covered under Stage 2 and 3 adjustments above for PDFs, but here we consolidate the broader media handling plan.)

- **PDFs:** Implemented as above: text extraction via PDF parser, included in enrichment. Additionally, consider **OCR for scanned PDFs** if we encounter some (this might require Tesseract and is expensive; perhaps a future enhancement if needed).
- **Images:** If some pages heavily rely on images that contain text (like infographics), a truly comprehensive approach would perform OCR on those images. However, this is likely out-of-scope unless UConn frequently posts information only in image form. We can note this as a future extension: using an OCR library on `<img>` files whose filenames or surrounding text suggest they might contain meaningful content (e.g., an image of a flyer).
- Implementation would involve downloading the image and running pytesseract or an OCR API. We would then store the extracted text maybe as a separate field or appended to page text (with a marker).
- Due to complexity and probably limited need, we mark this low priority.
- **Audio/Video:** If Stage 1 finds links to videos (maybe YouTube links or .mp4 files on the site), our current pipeline does nothing with them. For completeness:
- We could use OpenAI Whisper or similar ASR to transcribe audio from videos. This is very computationally heavy (transcribing even a 5-minute video is not trivial). Likely we wouldn't do this for all videos, but maybe for important ones (e.g., the President's welcome speech video).
- If doing so, Stage 3 could detect a video URL (maybe .mp4 or YouTube embed), download or stream it, then run a speech-to-text model. The resulting transcript could be stored as text_content. This essentially treats video as another content source.
- Possibly use a lightweight model for shorter clips if available.
- This is a far-future item unless specifically needed; we'll keep it noted.
- **Modularity:** If we foresee more of these, we might break Stage 3 into multiple pipelines: e.g., a specialized pipeline for PDFs (Stage 3A), images (3B), etc. Or incorporate a plugin system where handling of different MIME types can be plugged in. For now, we handle PDFs inline and skip others, which is fine.

In summary, Stage 3 improvements make the enrichment phase **much more powerful**. We'll extract more data (text from PDFs, transcripts maybe later), and extract more insight (summaries, categories, better entities) from each page. We'll also be smarter about which pages to focus on or disregard in this stage. The enriched output JSONL will be a goldmine of information ready for building an intelligent search or Q&A system about UConn.

These enhancements also set the stage for possibly training a custom language model on UConn data. With entity-rich, categorized, and summarized data, a model could learn much more effectively than from raw text alone. This fulfills the vision of enabling "any and every student to answer questions they have" by consulting this comprehensive dataset.

Orchestration & Pipeline Resilience

The Orchestrator (main async pipeline) coordinates stages and ensures data flows correctly. After adding many features, we must ensure the orchestration remains **robust, deadlock-free, and can gracefully handle restarts or failures**.

5.1 Pipeline Backpressure & Flow Control (Priority: High, Impact: High)

- **Recap:** The orchestrator uses `BatchQueue` with a max size and batch_size to feed URLs between stages [64] [65]. This prevents an explosion of memory usage by queuing millions of URLs at once. We should verify and tune these parameters:
- Ensure that Stage 1 production doesn't overwhelm Stage 2. Currently max_queue_size is 10000 in config [66] and batch_size 1000, which seems reasonable.
- With new features, Stage 2 might slow down (if many PDFs to HEAD, etc.), so backpressure might kick in more often.
- **Improvements:**
- **Adaptive batch sizing:** Possibly adjust batch_size based on performance. E.g., if Stage 2 is processing very quickly, increase batch_size for efficiency; if it's slow, smaller batches might help latency. This might be micro-optimization and likely not needed if values are okay.
- **Stage-specific tuning:** Perhaps allow separate batch_size for Stage1->2 vs Stage2->3 (currently both are 1000 by default). Stage 3 might handle larger batches fine if it's mostly I/O bound (Scrapy fetching pages), but if heavy NLP, maybe don't inundate it. We can control it in config (`ENRICHMENT_BATCH_SIZE` exists [67]).
- **Monitoring backpressure:** Add logging when queue usage crosses thresholds (we already log warnings at 80% full [65]). Ensure we act if we see that often (like maybe increase consumer concurrency or simply know that's expected).
- The good news: our pipeline design is mostly sequential (Stage 1 then Stage 2 then Stage 3, unless `--stage all` streams them). Actually, orchestrator's `--stage all` *does* run Stage 1 and Stage 2 concurrently (Stage 1 feeding Stage 2 queue) [68] [69], and then Stage 3 after (it collects Stage 2 results then launches Scrapy) [70] [71].
- We should ensure that if Stage 2 finishes and Stage 3 hasn't started (because it waits for Stage 2 complete collection), we handle that smoothly. The code does this by collecting all Stage 2 results first in a list before launching Stage 3 Scrapy [72] [73].
- **Graceful Shutdown Enhancements:**
- Already, there is a function `_cleanup_temp_directory` to remove old files on startup [74]. We should also handle interruptions:
- E.g., if user hits Ctrl+C during Stage 2, we want to save the partial validation_output and checkpoint.
- Implement signal handlers in orchestrator: trap SIGINT and call `PipelineOrchestrator.stage1_to_stage2_queue.mark_producer_done()` if Stage 1 was running, etc., then exit. Possibly Scrapy handles signals internally.

- On Stage 3's separate process, Scrapy will stop on SIGINT and our main process should detect that and not consider it an error if user requested it.
- We may test sending SIGINT in a controlled way to see if our logs show all needed info (like Stage 1's spider `closed` summary is printed).
- **State Persistence:**
- The checkpoint system is already in place (BatchCheckpoint) for Stage 2 and Stage 3 queue to resume if re-run [75] [76].
- We should ensure after all these changes that checkpoints are updated correctly:
    - For Stage 2, it marks as completed after finishing each batch [77] [78] and uses `CheckpointManager` which likely writes to a file per batch. We should test resuming after a crash: it should skip batches already done.
    - Possibly extend checkpoint to Stage 3 if we ever stream Stage 3 (currently Stage 3 doesn't stream – it runs after Stage 2 fully done).
    - But maybe in future we could pipeline Stage 3 too (though with heavy NLP, maybe keep sequential).
- **Scaling Concurrency:**
- If we wanted to increase Stage 2 concurrency (max_workers) beyond 16 (like to 50) for faster validation, ensure the queue and memory can handle it. The connector limit is min(max_workers*2,100) [79], which is fine up to 50->100. We might consider DNS caching etc which is already on (ttl_dns_cache=300) [80].
- Stage 3 concurrency: Scrapy by default we set 16 concurrency for enrichment [81]. If NLP is CPU heavy, raising concurrency might not help much, but good if many I/O bound tasks (like downloading PDFs).
- **Testing:** Simulate a large crawl (maybe by duplicating seeds or crawling a smaller site with many pages) to see that memory remains stable and throughput is as expected. Monitor log for any warnings about queue full.
- **Conclusion:** After fine-tuning, the orchestrator should handle flows smoothly. We expect that even if Stage 3 or Stage 2 slows down, Stage 1 will pause (backpressure) rather than OOM. And if we stop and restart, the pipeline uses URLCache and checkpoints to pick up where left off with minimal duplication [82] [83].

5.2 Advanced Configuration & Runtime Controls (Priority: High, Impact: Medium)

- **Configuration management:** We want to be able to tweak things without code changes, especially given the new flags and thresholds introduced:
- Already added features like feature flags for heuristics, NLP enable, etc., all should be in the YAML and respect environment overrides.
- We might move Scrapy settings out of code into config (the plan mentioned moving `settings.py` to YAML which we effectively did via `get_scrapy_settings` usage [84]). Ensure any newly needed Scrapy or pipeline setting is exposed similarly.
- Possibly implement config versioning – since we have a complex config, adding a section to track version might be useful for upgrades (the SchemaRegistry in code hints at schema version for data, not config).
- **CLI enhancements:** The `main.py` accepts `--stage` and `--env`. We could add:
- A flag to disable certain features quickly. For instance, `--no-browser` to run without Playwright even if config has it enabled (for debugging).
- A flag `--max-urls N` to limit how many URLs to process (useful for test runs).
- These are conveniences for development/testing to avoid needing separate config files.

- **Logging improvements:** Already in config we have `structured: false` setting. We could allow JSON logging for easier analysis (set structured: true to log JSON lines). This is already accounted for, just ensure to implement it (maybe via Python logging Formatter).
- **Reasoning:** These orchestrator and config improvements ensure that as the system grows in complexity, it remains *controllable*. The team can easily adjust performance knobs or disable parts if issues arise, without redeploying code.

By focusing on orchestration resilience and configurability now, we prevent the scenario where the pipeline is feature-rich but fragile or a "black box". Instead, it will be robust and transparent, giving confidence to run extremely large crawls (e.g., all of `uconn.edu` which could be millions of pages) reliably.

Data Storage & Persistence

Data from the pipeline is currently stored in JSONL files for each stage, and a SQLite `URLCache` for deduplication and tracking. Improvements here revolve around making data access and updates easier for downstream use, and preparing for potential scaling to a database if needed.

6.1 SQLite URLCache Enhancements (Priority: Medium, Impact: Medium)

- **Current URLCache usage:** We use it primarily in Stage 1 for dedup (inserting new URLs) [85] and Stage 2/3 to update status_code, title, etc., as they process [86] [87] . It acts as a mini database of all URLs seen.
- **Improvements:**
- Ensure we store all the new metadata fields: e.g., add columns for `last_modified`, `discovery_depth`, maybe `content_hash`. We already have `title` and `word_count` columns for enrichment results [88] , which is great.
- Use URLCache for **final data merging**: We could at the end query `SELECT * FROM urls` and get a combined view of each URL's discovery, validation, and enrichment info. This is essentially the `URLRecord` dataclass goal [89] .
    - Right now, JSONL outputs are separate per stage. A combined view would be helpful to, say, answer: how many URLs were discovered but failed validation, etc.
    - We can create a script or command to produce a merged CSV/JSON from the URLCache. (Perhaps add a CLI flag for that, or just treat the SQLite as the source).
- **Performance:** For large crawls, ensure the SQLite is efficient:
    - We already set WAL mode and indexes [90] [91] which is good.
    - We might use `executemany` or transactions for bulk updates (like Stage 3 might do many updates – but we call update_enrichment per URL which opens a connection each time [87] . This is not super efficient. Possibly we can batch update using a single connection or add a method to do multiple at once.)
    - However, Stage 3 enrichment likely is network-bound, so the overhead of DB per item is minor.
- Consider moving JSONL persistence of Stage outputs to rely on URLCache entirely:
    - For example, instead of writing `validation_output.jsonl`, we could just rely on URLCache entries updated for each URL. But having the JSONL is convenient for Stage 3 input and for human debugging. We can keep both.

- ◦ Perhaps as an optional mode, allow using only DB to store intermediate results (less file I/O, but then we need to query DB for Stage 3 input).
- **Backup & Integrity:** Provide an easy way to backup the SQLite (though it's a file that can be copied). Could incorporate a step to dump it to SQL or to compress it after run.
- **Step-by-Step:**
- Update `URLCache._init_db` to add new columns:
  - ◦ e.g., `discovery_depth INTEGER` (and set in add_discovery).
  - ◦ `last_modified TEXT` (for validation to update).
  - ◦ Possibly `content_hash TEXT` (for enrichment to update if we implement duplicate check).
  - ◦ Run a migration or if DB exists from before, handle missing column (since it's SQLite, one approach is to detect and ALTER table as needed).
- Use `URLCache.update_validation` to also set last_modified and content_type (we already set content_type in update_validation) [92] . We can parse and pass last_modified from Stage 2 when calling this (need to modify orchestrator or wherever we call update_validation).
- Use `URLCache.update_enrichment` to also set word_count (currently it does) and maybe content_hash (if applicable) [61] .
- After crawl, test that a random sample URL in DB has all fields populated correctly (e.g., pick one that was enriched, ensure its status_code, title, etc. all present).
- Write a small utility (maybe in `data/` or `docs/`) to query the DB for stats, e.g., count of total URLs, how many valid, how many enriched. (We already have `URLCache.get_stats()` that returns counts of total, validated, enriched, valid_urls [93] .)
- **Benefit:** A richer URLCache makes the pipeline's state more persistent. If we crawl again later, we can even decide to only crawl new or updated URLs by checking this DB. (For example, if we store last crawl time per URL, we could avoid re-enriching unchanged pages in incremental crawls – a possible future feature for maintenance). It also simplifies data analysis since one can run SQL queries to answer questions about the whole dataset.

6.2 Transition to Relational Database (Priority: Medium, Impact: High)

- **Vision:** Eventually, if this project expands (multi-university or simply large scale), using a full-fledged database (PostgreSQL or MySQL) can be beneficial for performance and multi-user access. This would involve:
- Designing a normalized schema similar to URLCache but possibly split: e.g., a URLs table, an Entities table (if we want to store extracted entities in a searchable way), etc.
- Using an ORM or direct SQL in the pipeline to insert and update records instead of SQLite calls.
- Handling connections (connection pooling, etc.) – more complexity but solvable.
- **Plan:** This is lower immediate priority because SQLite is handling the scale currently. However, we can prepare by:
- Ensuring our code that interacts with storage (like the `storage.py` module) is abstract enough. Notice we have `ConfigurableStorage` class that can choose JSONL or SQLite based on a setting [94] . We could extend that to a `'postgres'` option in the future. The idea is to funnel all storage operations through a common interface so switching backend is easier.
- Perhaps implement a basic version of Postgres integration as a demonstration:
  - ◦ Use SQLAlchemy or psycopg to connect, create table schema equivalent to URLCache.
  - ◦ Implement add_url_if_new, update_validation, update_enrichment in that context.
  - ◦ Provide config to toggle `storage_type: postgres` and connection details.
  - ◦ Test on a small scale.

- Noting challenges: Multi-threading or async with DB might need careful handling. Since our pipeline is async (Stage 2) but uses sync DB calls now (SQLite), switching to async DB library (like asyncpg) might be needed for full performance. Alternatively, do DB ops in thread executor if needed.
- **Impact:** A well-structured Postgres database could allow writing more complex queries for analysis, join with external data (maybe linking faculty names to HR data, etc.), and scaling to millions of records with better concurrency. But until we hit the limits of SQLite (which might be fine for a single site), it's an optional enhancement.

6.3 Data Export & Backup (Priority: Low, Impact: Low)

- Though covered later in [Data Export & Integration](#), from a storage perspective:
- We may add automated compression of JSONL outputs (to save space, especially if enriched_data.jsonl becomes huge). A quick win: compress old data when starting a new crawl or have an archive process.
- Data retention: if crawling repeatedly, decide how to version outputs (maybe by date).
- Possibly add in config a setting for how long to keep logs or old outputs, and `_cleanup_temp_directory` can be extended to clean processed data older than X days if desired.

In summary, fortifying the storage layer ensures the pipeline's results are durable and accessible. It paves the way for treating the scraped content as a **database of knowledge** that can be queried or updated, rather than just static files. This will be especially important when integrating with other systems (like a web interface or an API that serves answers from this data).

Monitoring, Observability & Alerting

To operate this pipeline at scale (especially if deployed as a continuous service), we need strong observability: metrics, logs, and alerts.

7.1 Metrics Collection & Dashboard (Priority: Medium, Impact: Medium)

- **Current Metrics:** We have a simple `MetricsCollector` class that tracks items processed, succeeded, failed per stage [95] [96], and computes throughput and success rate [97] [98]. It logs a summary at the end of pipeline run [99] [100].
- **Improvements:**
- Expand metrics tracked:
  - For Stage 1: number of URLs discovered per domain or per discovery_source. We have some of that in logs and `_dynamic_discovery_stats` structure inside spider, but we can aggregate it and output in metrics summary.
  - Stage 2: distribution of status codes (how many 200s, 404s, etc.), average response time, maybe 95th percentile latency.
  - Stage 3: average page size (words), count of pages by content type (how many PDFs processed, etc.), maybe count of entities extracted (total).
- Possibly integrate with a monitoring system like **Prometheus**. We could expose metrics via a simple HTTP server (maybe not trivial inside this app, but we could output to a file or pushgateway). The TODO in code suggests exporting to time-series DB [101] which is a good idea for a continuous running scenario.

- At minimum, produce a machine-readable metrics file at end (e.g., JSON of summary stats) in `data/logs/metrics.json`.
- If building a **web dashboard** (perhaps a small Flask or a static HTML that loads JSON), we can use the data from metrics and maybe live log tailing to show:
    - Progress: e.g., "Stage 1: 5000 pages crawled, 12000 URLs found, depth 3 complete."
    - Graphs: items/second over time (maybe from logs we can derive).
    - Could use Chart.js or similar to visualize after the run.
    - A simpler approach: output intermediate metrics at intervals to a file, and a separate process visualizes it. This might be overkill for now. Possibly a "monitor mode" CLI that prints updates every X seconds by checking the queue sizes (could be an idea: since we have queue lengths).
- **Implementation:**
- Add counters for specific things during the run. We might incorporate into `MetricsCollector` or separate. For instance, increment `metrics.stage1_dynamic_urls += 1` whenever spider yields a dynamic URL, etc.
- As Stage 2 runs, it could maintain counters for each status code (maybe in the URLValidator, update a dict).
- After each stage or at certain intervals, call a function to dump metrics to a JSON file (overwriting).
- The UI part: if we have time, create a simple HTML in docs that reads the metrics JSON (which could be served by a lightweight server, or just viewed after run).
- Possibly utilize existing tools: Scrapy has built-in stats collection and even a Telnet or web service extension (Telnet is disabled currently [102] but there's Scrapy's StatsCollector). Since we roll our own orchestrator, maybe not use Scrapy's stats.
- **Value:** With metrics, if something goes wrong (e.g., crawl slows down unexpectedly), we can quickly identify it (maybe Stage 2 throughput dropped due to many timeouts). It also helps in optimizing – we can see the effect of concurrency or other tuning on throughput. And for long crawls, having a live progress view is reassuring.

7.2 Advanced Logging & Alerting (Priority: Medium, Impact: Medium)

- **Logging enhancements:**
- Ensure **sensitive information** is not logged. The config suggests to prevent logging sensitive info (ORG-052). Likely not a big issue here since we aren't dealing with user data, but be mindful if any secrets (like DB passwords or API keys for OCR etc.) were to be used.
- Introduce **structured logs** when useful: e.g., an error log could include a JSON object with URL and error message, making it easier to parse programmatically.
- Use log levels more granularly. Possibly set Stage 1 dynamic discovery debug logs to DEBUG so they can be turned off during normal runs (to avoid log flooding).
- Log summary per stage completion clearly.
- **Alerting system:** We have an `AlertManager` implemented with support for email, webhook, file channels [103] [104] .
- It can send alerts for certain severity and we configured threshold in config (default 'error') [105] . We need to actually use it:
    - Identify events to trigger alerts. For example:
    - Stage failure (exception not caught) – we should catch top-level exceptions in each stage and call `alert_manager.stage_failed(stage, reason)` [106] .
    - Pipeline complete success – could send an info alert (maybe to Slack or email, if configured) using `alert_manager.pipeline_complete(stats)` [107] .
    - If certain critical metrics threshold exceeded (like error rate > 5%), send warning.

- If dynamic discovery yields an extremely high number of URLs (maybe a sign of a potential spider trap), send warning.
- Integrate alert calls in code where appropriate. Possibly wrap Stage 2 validate_batch in try/except to catch if a batch fails entirely and alert.
- Use the channels as per config. We might test file channel first (writes alerts to a file).
- Document how to configure alerts (the YAML examples are given in config).
- **On-call rotation etc.:** Not needed for a one-person or small team project, but the idea is to ensure if this pipeline runs overnight and something breaks, someone gets notified.
- **Testing Alerts:** Simulate a failure (maybe force an exception in Stage 3 parse) to see if alert triggers and what it contains. Fine-tune severity thresholds to avoid spamming (maybe only critical issues like complete stage failures email, but lesser issues just log).
- **Outcome:** We will have a pipeline that essentially can **take care of itself** or at least call for help when needed. Operators can trust that if something goes awry (too many failures, crash, etc.), they will know about it quickly. Meanwhile, the monitoring will allow them to watch progress or performance.

Combining metrics, logging, and alerting completes the observability trifecta. This level of insight is usually found in production systems – bringing it here means we can confidently run very large crawls and know exactly what's happening internally. It also greatly speeds up debugging and optimization, as we'll have data to back decisions (for example, where the bottleneck is).

Performance & Scalability

The pipeline should scale in two dimensions: **volume of data** (potentially millions of pages) and **speed** (time to complete a crawl). We address performance bottlenecks and consider steps towards distributing load if needed.

8.1 Memory & Resource Optimization (Priority: High, Impact: High)

- **Memory usage concerns:**
- Stage 1 in Scrapy can potentially hold a lot in memory (the crawler and the sets of seen URLs). We mitigated some by streaming items to file and using SQLite for dedup to avoid huge in-memory sets [85] . But the spider still keeps `seen_urls` and `url_hashes` sets in memory for quick check [108] [109] . For extremely large crawls, those could be millions of entries. Potential improvement: rely solely on SQLite after initial load (perhaps we could drop the Python set after spider starts, and always call URLCache for dedup – but that might slow down check too much per URL). Alternatively, use a Bloom filter or something probabilistic for memory efficiency (with slight chance of false positive duplication).
- Stage 2: holds processed_hashes set to avoid reprocessing on resume [110] – typically smaller than total URLs. Should be fine unless millions (still, a few million in a Python set might be borderline).
- Stage 3: not much memory except Scrapy's own buffers and any loaded models.
- The orchestrator queues hold up to 10k items by default; each item minimal memory (URL string and a few fields).
- If embedding models (like the SentenceTransformer), that takes some memory (~80MB for MiniLM).

- If we load big transformers for summarization or NER, those can be hundreds of MB (Bart-large ~ 1.6GB model). We should be cautious: maybe use smaller models or allow using GPU if available for those tasks to speed up (GPU usage is another dimension).
- **Optimizations:**
- For dedup sets: possibly periodically clear some memory if not needed. For instance, Stage 1 could discard hashes of URLs after depth passes? But then if same URL appears later unexpectedly it might recrawl… not good. Better keep dedup persistent.
- Use Python's `gc` and memory profiling to find any leaks or high usage spots.
- If memory becomes a problem, we might have to page some data structures to disk (but SQLite is already doing that).
- Consider using more efficient data structures: e.g., instead of Python `set` for seen_urls, maybe use a `set` of hashes only (smaller than full URLs). We do use url_hashes set and also seen_urls. Possibly we could drop storing full URLs in memory and only store hashes, which are fixed 64 bytes hex. Then, when a new candidate emerges, we hash it and check the set of hashes + maybe check DB for collision (very rare that different URL yields same SHA256, basically impossible for our use).
- If hitting memory issues with Python, consider a different language for the crawling core or breaking the crawl (but likely not needed).
- **File I/O:**
- Ensure we open files in append mode and flush appropriately but not too often (Stage 1 pipeline flushes on every write currently [62], which is okay but maybe bufferring 100 items before flush could improve disk performance slightly. But flush ensures data is written in case of crash, so it's safer).
- Stage 2 writing JSONL after each validation item might not flush each time, it likely writes in batches since we accumulate in memory then write after each batch presumably.
- If needed, increase batch_size (like writing 10000 at once) to reduce overhead.
- **Profiling:** We should do a run with a profiler to see CPU usage distribution:
- Suspect Stage 2 might spend time waiting on network mostly (so CPU fine).
- Stage 3 might become CPU-heavy with NLP. If that's an issue, consider enabling multi-threading for NLP or using GPU if present (maybe let huggingface use GPU if available).
- Memory profiling to ensure no major leaks (like if we accidentally keep references to large objects beyond their use).
- **Garbage collection tuning:** Possibly call `gc.collect()` at certain points (e.g., after finishing a stage) to promptly free objects. Python might eventually do it anyway, but explicitly after Stage 1 might free all the response objects etc.
- **Outcome:** With these optimizations, we aim to run the pipeline on commodity hardware (say a standard laptop or VM) for the entire uconn.edu without crashing or slowing to a crawl. It's hard to predict exact usage, but ideally we keep memory under a few GB at peak.

8.2 Throughput Improvements (Priority: High, Impact: High)

- **Network throughput:**
- Stage 2 concurrency can be increased if needed – we currently use 16 workers. If the machine and network can handle it, raising to 50 or 100 could speed validation (especially if many URLs are slow to respond, more parallelism helps). Need to watch not to overwhelm UConn's servers though (ethical scraping practices!). We should obey `robots.txt` (currently disabled) – maybe consider enabling it or manually ensure our request rate is polite.
- Possibly implement **rate limiting** per domain (Scrapy has per-domain concurrency and delay which we set modestly).

- Consider using HTTP/2 if aiohttp supports it and target servers allow (could speed up multiple requests to same domain).
- **Parsing throughput:**
- Stage 3 Scrapy is single-threaded for parsing but concurrency in requests. If we see Stage 3 becoming the slowest stage due to heavy NLP, we might parallelize it:
  - e.g., run multiple instances of EnrichmentSpider each on a subset of URLs. This could be done by splitting the Stage 2 output and running two processes. However, coordinating that in orchestrator is complex. Perhaps easier outside of it (like run `--stage3` twice with half the input each, if needed).
  - Alternatively, incorporate multi-thread in parse for NLP – not straightforward with Scrapy's reactor. But we could push heavy operations to a thread pool using Twisted's deferToThread or asyncio's run_in_executor.
- If content extraction (like PDF parsing) is slow, maybe do it in parallel threads too since GIL release likely in those C libraries.
- **Disk I/O:**
- If writing huge JSONL, consider writing gzipped directly (python can write gzip line by line).
- Check if logging too verbose could slow things; adjust log level in production runs to WARNING or INFO only.
- **Parallelizing Stages:** Currently Stage 3 waits for Stage 2 to finish. Perhaps for large sites, we might stream Stage 3 as well (i.e., start enriching as soon as some validated URLs available). This is complex due to our architecture launching Stage 3 after Stage 2 ends. We could theoretically have Stage 3 spider running concurrently, reading from stage2_to_stage3_queue similarly to how Stage 2 reads stage1 queue. The orchestrator code structure even hints at this possibility (we gather then run subprocess, but one could integrate differently).
- Not implementing now due to complexity, but if speed is crucial, we could attempt it: run Scrapy in a thread or subprocess that reads a file or IPC for URLs continuously. Perhaps not worth it unless absolutely needed.
- **Testing throughput changes:** We can measure how many URLs per second Stage 2 and Stage 3 handle on average and then tune:
- If Stage 2 is bottleneck, raise max_workers until maybe network or CPU saturates. Use metrics to confirm.
- If Stage 3 is bottleneck, see if it's network (pages slow to download) or CPU (NLP). If network-bound, increasing concurrency in Scrapy might help; if CPU-bound (NLP), then either get more CPU cores and parallelize or accept slower speed or reduce NLP tasks (maybe skip summarization if not needed for all pages).
- **Goal:** Ideally the entire crawl of UConn (which might be, say, 100k URLs as a guess) should finish in reasonable time (maybe a few hours). If current design would take 10+ hours, we should see what the slow part is and improve it.

8.3 Distributed & Multi-Process Scalability (Priority: Low, Impact: Low for now)

- If one machine is not enough, could we distribute work? Perhaps in far future:
- Partition by subdomain: e.g., one instance crawls `today.uconn.edu` (news site) and another `catalog.uconn.edu` etc., then combine results.
- Or use a message queue where discovered URLs are pushed to a central queue that many workers pop from to validate and enrich.
- This essentially becomes a big architecture (like a mini search engine). We likely don't need this unless adding multi-university crawling concurrently.

- Nonetheless, designing with modularity (each stage can run independently) means we can run multiple Stage 1 in parallel on different seed sets, etc.
- If implementing, use something like RabbitMQ or Redis queues to coordinate, and perhaps separate processes for each stage.
- This is acknowledged but not to be done now.

By focusing on performance now, we ensure that as UConn's website grows or if we apply this to other data, the pipeline won't crumble. It's better to identify and fix bottlenecks while the project is fresh than to be surprised later. We'll iterate: measure, optimize, measure again.

Security & Compliance

While this is an internal data collection tool, we should still adhere to security best practices and consider compliance, especially as we might handle personal data (faculty info, student info) and possibly use this pipeline in different contexts.

9.1 Ethical Crawling and Compliance

- **Robots.txt & Rate Limits:** The pipeline currently does not obey robots.txt (we set `ROBOTSTXT_OBEY=false` in config [111] intentionally perhaps to not miss content). We should reconsider this:
- At least read the `robots.txt` and highlight if we are disallowed for certain paths. Because UConn might not want certain parts scraped (though being internal project maybe it's fine).
- Perhaps make obey robots an option (`obey_robots: true/false` in config). And/or always obey for public releases.
- Ensure our user-agent is identifiable (currently "UConn-Discovery-Crawler/1.0 (development)" [112]). That's good; we might update version in UA as we change.
- We might implement a polite delay by default (we have 0.1s now, which is quite fast, maybe okay for a domain like uconn.edu which is large and on presumably robust servers. But if not, we could raise it to 0.5 or 1 for safety).
- **User-Agent rotation:** If we worried about being blocked, we could rotate user agents (different browser strings) or use proxies. But since this is for UConn specifically, probably not needed unless their security flags our scraper.
- Still, ORG-051 mentioned rotating UA, so it's considered. Could implement a small list of UAs and pick randomly each request (Scrapy can do that via middleware).
- **SSL Verification:** We currently disable SSL verification (`ssl_context.check_hostname=False, verify_mode=CERT_NONE` in Stage 2) [113] for development convenience. This is a security risk if we care about data integrity (we could be MITM'd). For production, we might enable proper SSL verification, unless UConn has self-signed certs somewhere. At least make it configurable.
- **Pickle usage:** It was mentioned to replace pickle caches with safer alternative (ORG-050). We indeed replaced in dedup with SQLite. Check that we're not using pickle anywhere now. If not, great.
- **Dependency security:** Keep dependencies updated to get security patches (e.g., Scrapy, aiohttp updates). Possibly integrate a `safety` or `pip-audit` check in CI to catch known vulnerabilities.
- **Data Privacy:**
- Even though scraping public data, it could include personal info (names, emails). If we distribute the data, consider privacy. For internal use likely fine.

- But we might implement a **PII detection** in Stage 3 (as hinted in Test Plan section 4) [114] . For example, if a page contains something like SSNs or sensitive data inadvertently, flag it. Probably not applicable to content on public site, but maybe personal addresses or phone numbers are there.
- If we plan to comply with GDPR or similar for storing personal data (maybe not necessary since no user-provided data, but if we considered multi-university, EU ones etc., they'd have rules).
- **Auth & Access Control:**
- If in future pipeline offers an API to data, ensure it requires authentication for sensitive endpoints.
- The scraping itself might need auth if parts of site behind login – not doing that now, but maybe a future concept (like scraping a student-only portal if allowed).
- That would involve storing credentials securely and not logging them, etc.
- **Compliance with site terms:** It's wise to double-check UConn's terms of use for their website to ensure our scraping is allowed for our intended purpose (most likely fine for research).
- **Team Security:** If open-sourcing this repo, ensure no secrets or internal email addresses are in config defaults.
- **Conclusion:** We'll implement robot obedience as optional, tighten SSL if needed, and maintain general good practices to not cause harm or raise alarms with our crawling.

9.2 Hardening & Dependency Isolation (Priority: Medium, Impact: Low)

- Run security scanners (like Bandit for Python code to catch issues).
- Sandbox execution: Stage 3 runs some external libs (like PDF parser). If paranoid, run them in a sub-process to avoid any potential malicious PDF doing bad things (PDF parsing can have exploits). Since our environment is controlled, risk is low, but consider updating the libraries to latest which usually fix such.
- If using LLM APIs, be cautious not to send sensitive data to external services unless terms allow (for now, all local).
- Possibly limit resource usage per process to avoid runaway (like use ulimit or similar if deployment environment supports).
- Rate limiting: implement in our HTTP client to not hit any single server too fast to avoid triggering security blocks.

Given the context, security issues are relatively minor but it's good to have an eye on them so that the pipeline can be run responsibly and safely. Implementing these also future-proofs if the pipeline goes beyond just UConn (some sites have stricter controls).

Data Export & Integration

After scraping and enriching, the next step is making the data usable. Different consumers might want it in different formats or via APIs.

10.1 Flexible Export Formats (Priority: Medium, Impact: Medium)

- **Current Output:** JSONL files for each stage, primarily the enriched_data.jsonl as final output. JSONL is good for machine processing but some may prefer CSV, Excel, or database dumps for analysis.
- **Improvements:**

- Provide scripts or options to export enriched data to:
    - **CSV:** Flattening text_content might be tricky (could keep it or drop for CSV), but useful for a quick view of structured fields like URL, title, word_count, etc.
    - **XLSX (Excel):** Possibly limit content size if exporting to Excel (it has cell size limits).
    - **Parquet:** For large data, Parquet columnar format could be great (especially if feeding to data analytics frameworks or training pipelines).
- Possibly integrate these into pipeline run: e.g., an argument `--export-format csv` to automatically generate after crawl. Or just provide a separate utility script for conversion.
- Ensure encoding issues are handled (UTF-8 output).
- **Step-by-Step:**
- Write a small converter that reads the JSONL and writes CSV. Python's csv library can be used. We have to be careful to quote fields like text_content if writing to one cell.
- For Parquet, use pandas or pyarrow to convert. This is quick if we have those libraries available.
- If adding as part of main tool, perhaps implement as part of a finalization step in orchestrator (but better to keep separate to not slow pipeline).
- Document how to use these.
- **Value:** Some team members or users might not be comfortable with JSONL and want to use Excel or run SQL queries on the data. By offering multiple formats, we increase the accessibility of the data.

10.2 REST API or GraphQL Integration (Priority: Medium, Impact: Medium)

- **Idea:** After having data, one might want to query it programmatically (e.g., a chatbot front-end or search engine UI). Setting up a simple API around the data could be useful.
- Perhaps implement a **Flask or FastAPI** server that reads from the URLCache (or a loaded dataset) and can return answers. This might be out of scope of the scraping pipeline itself, but we can ease integration.
- At minimum, ensure the data format is conducive to indexing. For example, maybe output a single JSON file with all enriched records for feeding into an ElasticSearch or Whoosh index (for building a search engine).
- If GraphQL is desired, we could use Graphene or Ariadne to define a schema where queries can filter data (like get all pages that have keyword X).
- This again might be a separate component rather than the pipeline script itself.
- **Steps (conceptual):**
- Possibly write a reference implementation: a small FastAPI app that on request queries SQLite for a URL or does a search through text_content (maybe using an FTS index if we add one to SQLite for content).
- Alternatively, push data to an external search service (if we had one configured).
- GraphQL could allow queries like `{ page(url: "https://uconn.edu/abc") { title, text_content, entities } }`.
- Provide this as an optional tool.
- **Goal:** Make it easy to integrate the data into user-facing applications. This is where the scraped data truly becomes useful (e.g., powering a Q&A chatbot that queries this knowledge base).

10.3 Automated Report Generation (Priority: Low, Impact: Low)

- The pipeline can generate some summary reports for internal use:
- E.g., an HTML report listing key stats and maybe highlighting potential issues (like "50 pages had no content", "10 pages seem to be error pages").

- Or a report of all PDFs found and their titles (could be useful if someone manually wants to check them).
- Trend reports if run periodically: e.g., compare with last crawl, what's new or removed (requires historical data).
- This is nice-to-have for maintenance. Could incorporate into the future if running regularly.

In essence, once data is collected, these integration steps ensure it doesn't just sit in a file – it can be consumed by others, whether through direct files or via an API. While the core of our project is the scraping itself, facilitating data usage increases its impact and value.

Documentation, Code Quality & Legacy Cleanup

Ensuring the project is well-documented and the codebase is clean will make it easier to maintain as we implement all these features.

11.1 Documentation Overhaul (Priority: Medium, Impact: Medium)

- **User Documentation:**
- Update and expand `README.md` and docs to reflect new features and usage. The README currently gives a solid overview [115] [116], but new things like PDF handling, config flags, etc., should be mentioned.
- Write specific guides:
  - *Configuration Guide:* Explain every config option in `development.yml` (like what does each flag do, e.g., heuristics toggles, alert settings).
  - *Running Guide:* How to run for a different domain (if that's possible by changing allowed_domains & seeds), how to resume after a crash, how to interpret the logs.
  - *Monitoring Guide:* If we have a dashboard or metrics, how to use it.
  - *Data Usage Guide:* How to query the SQLite or convert to CSV, etc.
- Possibly an **architecture diagram** to visually show Stage 1-2-3 workflow. This could be in docs or as a chart.
- Ensure the Future Plan (this document) remains updated as items are completed – maybe mark what was done and add new ideas.
- **Developer Documentation:**
- Clean up inline comments and ensure each public function has a docstring explaining its purpose (some places might lack docstrings).
- Where code is complex, e.g., `_validate_with_session` logic, have comments explaining the flow.
- Ensure consistent style (maybe run a linter/formatter like black).
- Possibly add type hints to all functions for clarity (some are present, but ensure complete).
- Write an **Architecture document** (docs/architecture.md exists) – update it if outdated, to reflect current architecture including orchestrator, queue, etc.
- Consider writing **ADR (Architecture Decision Record)** for major decisions (just to log why we chose a certain approach like using Playwright or using SQLite).
- **Legacy Code Removal:**

- Identify code that was for earlier versions and is now superseded:
  - For example, if we fully trust SQLite dedup, do we still need to write the .hashes file in Stage 1 pipeline? [117] [118] . Maybe we can remove that or at least make it optional (to reduce redundancy). If .hashes was a backup, but SQLite covers it, lean on one method to avoid confusion.
  - The `start_requests` vs `async start` in DiscoverySpider – it uses sync and just delegates to async one. Possibly scrap `start_requests` with newer Scrapy version and use `async def start` fully if supported.
  - Hardcoded allowed_domains in spiders [119] [3] – if we plan multi-domain, allow setting via config and remove those lines or override them via spider arguments.
  - Remove any commented-out code or print statements used in debugging.
- Check for any "TODO" notes that were actually done and remove or update them. Add new TODOs if something is too detailed to do now but noted.
- **Quality checks:**
- Use tools like flake8, mypy (if we add type hints, running mypy can catch some issues).
- Add them in CI for future merges.
- Possibly add pre-commit hooks to auto-format and lint (as noted ORG-064 and ORG-066 about enforcing code style and missing docstrings).
- **Outcome:** A clearly documented codebase where new contributors (or future us) can quickly understand how things work and why. This reduces onboarding time and helps in debugging if an issue arises months later (we can read our docs and recall the design decisions).

11.2 Naming Consistency & Refactoring (Priority: Medium, Impact: Medium)

- Ensure naming of functions, classes, variables is consistent and meaningful:
- e.g., sometimes we use URL vs url, make sure casing is consistent (mostly lowercase with underscores for variables).
- If certain terms are outdated (like using "Stage1Pipeline" – which is fine – but if something was named when it did slightly different job, update it).
- Refactor large functions if any:
- For instance, if `DiscoverySpider.parse` grows too large with new logic, break some parts into helper methods for readability.
- The orchestrator's `run_concurrent_stage3_enrichment` is a bit complex [120] [71] , maybe split into smaller parts or at least comment each section.
- Remove any duplicate code: e.g., if some logic to handle allowed_domains is in two places, unify it.
- Validate that error messages are clear and actionable. If an exception arises, do we log enough context to debug? E.g., if Stage 3 parsing fails on a particular page, log the URL in error message (the code does that in except [121]).
- These code quality improvements, while not directly adding features, prevent bugs and make maintenance of all these new features easier.

By thoroughly documenting and cleaning the project now, we ensure its **longevity**. The project has indeed "gone through a lot of changes and evolutions," and this step is about tidying up after that growth spurt – removing legacy remnants, clarifying design, and smoothing the path for future improvements or for others to understand the system.

Extension & Modularity

Making the pipeline more extensible means others (or we, in the future) can plug in new functionality without modifying the core. Given the goal of possibly a **universal data collection framework**, designing for extensibility is wise.

12.1 Plugin Architecture (Priority: Medium, Impact: Medium)

- **Vision:** Define clear extension points where custom code can run, e.g.:
- **Custom spiders or heuristics:** Perhaps allow adding new heuristics in Stage 1 via entry points. For example, if someone wants to add a special parser for calendar links, they could write a plugin class that DiscoverySpider calls.
- **Custom pipelines or exporters:** Maybe someone wants to directly export data to an API instead of JSONL – a plugin could handle that.
- **NLP plugins:** Allow easily adding a new NLP step in Stage 3 (like a plugin that tags sentiment or translates content).
- **Implementation idea:**
- Use Python's entry points or a plugin manager (like `pluggy` library or just import modules from a `plugins/` directory).
- For instance, define a interface class for a Stage 1 HeuristicPlugin with a method `process(response) -> list of (url, source, confidence)`. At runtime, DiscoverySpider could loop through all registered HeuristicPlugins and aggregate their found URLs.
- For Stage 3, maybe define a plugin interface that processes an EnrichmentItem after basic fields are filled. E.g., a plugin could add a field or modify content_tags.
- Maintain a registry of plugins and call them appropriately.
- Ensure plugin errors are caught so they don't crash pipeline; maybe skip on failure with a log.
- Provide examples of a simple plugin (maybe one that counts images on a page and adds that as a field, just as demonstration).
- **Effort:** This is non-trivial and would require carefully designing how much of internal data to expose to plugins, and documenting it. But doing it gradually is possible (e.g., start with Stage 1 plugins, then Stage 3).
- **Why:** This makes the project more flexible and possibly community-friendly if open-sourced. It also allows adapting to site-specific needs (like multi-university support could be implemented partly as plugins for each university's quirks).
- Given time constraints, we might plan this but not fully implement until core features solid. It's an important architecture direction though.

12.2 Multi-Domain (Multi-University) Support (Priority: Low, Impact: Very High)

- This is essentially making the pipeline domain-agnostic:
- Move UConn-specific things (like `allowed_domains=["uconn.edu"]` in spiders [119] and heuristics tailored to UConn structure) to config.
- The seeds file already is just a list of URLs, so that's fine.
- The content tags list in config [122] is UConn oriented (specific terms like majors, departments). For a different site, one would change these. Perhaps allow multiple profiles.
- Possibly the plugin system above could handle domain-specific logic by enabling certain plugins for certain domains.

- We could conceive of an input file that defines the profile of a university (list of keywords, any custom crawl rules, etc.).
- We should ensure nothing in code is hardcoded for UConn beyond what's easily configurable:
- The `is_valid_uconn_url` function specifically is for uconn.edu [123] . That would need to generalize to any domain (maybe rename to is_valid_domain_url and provide the target domain(s) as parameter).
- allowed_domains similarly.
- If multi-support, might need to run in series or parallel for multiple domains – which enters distributed territory.
- **Conclusion:** Multi-university support is a far-reaching extension. It's an ultimate test of modularity. For now, ensure to abstract domain-specific things behind config so at least someone could manually repoint it to another domain with minimal code changes.

12.3 Modular Component Design

- Break out major components like **URL extraction**, **content parsing**, **NLP tasks** into modules that could be replaced/upgraded:
- e.g., if someone wants to use a completely different crawling library (like Scrapy replaced by something else), could we swap Stage 1 out? That's tough because orchestrator tied to Scrapy.
- But maybe make orchestrator and stage implementations loosely coupled via interfaces.
- At least, keep boundaries clear: Stage 1 yields a JSONL of found URLs. One could generate that by another tool and still use Stage 2 and 3 on it.
- Stage 2 and Stage 3 currently depend on orchestrator linking them, but they can also run standalone from file (like we can run stage2 given stage1 file, etc., which is good).
- Possibly use dependency injection for certain services (like we have `ConfigurableStorage` which is a basic DI for storage backend).
- Ensuring backward compatibility: if we change schema of output, consider versioning (which we did by schema_version fields in items [124] [125]).
- Because of time, these thoughts likely remain more theoretical now, but we keep the code as clean and segmented as possible so that adding or replacing parts is not a nightmare.

By planning for extension, we ensure the pipeline can adapt to future requirements beyond UConn or incorporate new tech easily. It ties in with the earlier documentation and plugin ideas – making the pipeline a general tool rather than a one-off script.

Deployment & Operations

Finally, consider how this pipeline runs in practice in an environment outside a dev laptop – e.g., on a server or in cloud.

13.1 Containerization & Environment Setup (Priority: Medium, Impact: Medium)

- Provide a Dockerfile to encapsulate all dependencies (Scrapy, Playwright browsers, etc.).
- Possibly use multi-stage: one to install system deps (like Playwright needs some webkit libs), then pip install python deps.
- This allows running the pipeline anywhere reliably.

- Add Docker Compose if needed for perhaps running alongside a database or monitoring tool.
- Ensure that in container, data directories are properly mounted or accessible.
- **Signal handling:** When container stops, our graceful shutdown should kick in (so handle SIGTERM).
- This simplifies deployment if someone wants to run this pipeline on a schedule in a cloud service or as a one-off job in Docker.

13.2 Cloud Deployment (K8s, AWS, etc.) (Priority: Low, Impact: Low)

- Perhaps outline how to run in Kubernetes: define a CronJob to run pipeline daily or something.
- Use persistent volume for data or push results to cloud storage (S3, etc.) after run.
- If distributed, maybe multiple pods for different stages – but let's not go too far here.

13.3 Operational Tools & Automation (Priority: Medium, Impact: Medium)

- **Health checks:** If pipeline is meant to run continuously (or in a service mode), implement a simple health check endpoint or status file update so an orchestrator knows it's alive.
- **Automation:** Could set up GitHub Actions to run tests on each commit, maybe even to run a small crawl (to ensure nothing broke at runtime).
- If planning regular crawls, write a script or use Airflow or similar to schedule runs and manage results (like archiving old results).
- **Recovery procedures:** Document how to resume after failure (basically rerun same command; thanks to dedup and checkpoint it should continue). Make sure team knows that.
- **Backup:** If data is critical, schedule backup of the `data/` directory or at least the SQLite DB to some backup location.

The theme is to treat the pipeline as a production system that might run regularly, so we adopt practices from DevOps to manage it.

Advanced Features & Long-Term Ideas

This section speculates on cutting-edge additions that could make the pipeline truly state-of-the-art, beyond the immediate needs.

14.1 LLM-Powered Adaptive Agents (Priority: Low, Impact: Low in near term, Very High potential)

- **Autonomous Scraping Agent:** Imagine using an LLM (like GPT-4 or a smaller fine-tuned model) to *decide how to crawl*. For example:
- The agent reads a page, and based on content might suggest what links to follow or even directly search the site for certain info.
- It could dynamically generate new requests (like "I see a faculty list, let's crawl each faculty page to find research areas").
- This is similar to how a human might plan the crawl.
- Implementation would involve prompt engineering and hooking the LLM's outputs into the crawler – a complex loop of "think and act".
- There is research in this area (LLM as a browser or as a scraper).

- **Semantic Extraction:** Use LLM to extract structured data from a page. E.g., feed the page text to GPT with a prompt "Extract all course names and credits from this catalog page in JSON." It could produce structured results more accurately than regex.
- This could be integrated as an optional Stage 3 step for specific known page types (like if URL contains `/courses/`, call a specialized LLM prompt to parse course details).
- The downside is cost if using an API, and speed. Perhaps use smaller local models for this (maybe fine-tuned ones for specific tasks).
- **Vision + Language (VLM):** If some content is only in images (like an organizational chart), using a model like OFA or PaLI that can do image understanding might extract info. This is quite advanced and likely not necessary for our domain, but the tech is evolving.
- Example: use a model to interpret a screenshot of a campus map and annotate it, though not sure how that helps Q&A.
- **Predictive Crawling:** Use ML to predict which URLs are likely to yield high-value content and prioritize them. We touched on this with confidence scores. Could be extended with a model that learned from previous crawl which link text or URL patterns tend to be fruitful (like a classifier that given a URL or link context, outputs probability of being content-rich).
- **Anomaly Detection:** As we crawl periodically, ML can detect if a page changed significantly (which might be interesting events), or if the site structure changed (maybe triggers an update to crawling strategy).
- These ideas are beyond initial scope but show a path to making the crawler more "intelligent" rather than rule-based.

14.2 Multi-University / Multi-Domain at Scale (Priority: Low, Impact: Very High if realized)

- To truly support multi-university, we might redesign parts of the system:
- A central scheduler feeding multiple crawling workers (maybe one per university).
- Unified storage (so you can query across universities).
- Generic heuristics plus site-specific overrides (some unis might have different structures).
- Possibly incorporate knowledge graphs – linking entities across domains (e.g., collaborations or comparisons).
- This essentially becomes a search engine for universities. It's a massive but intriguing project (something like Google Custom Search but our own improved version).
- If we ever go that route, careful architecture planning is needed (maybe splitting into microservices: one service for discovery, one for validation, etc., managed via message queues).
- However, one can approach by incrementally adding another domain and seeing what breaks or what needs abstraction (likely the allowed_domains and content tag logic first).

14.3 Other Advanced Crawling Strategies (Priority: Low, Impact: Medium)

- **Focused Crawling:** If we only care about specific topics (like "admissions info"), one could use ML to identify relevant pages and ignore others. Might not apply if we want everything.
- **Budgeted Crawling:** If limited time, allocate depth and breadth wisely (like ensure at least some content from each section by not spending all on one deep section).
- **Sitemap-guided crawling:** Already noted, but generating a virtual sitemap from discovered links could also help ensure we didn't miss any section (maybe post-run, see if any obvious URL patterns were not covered).

- **Continuous crawling:** Turn pipeline into a persistent service that watches the site (via RSS or change detection) and only crawls updates. This requires storing last seen state and doing incremental updates. Could be efficient long-term.

These advanced strategies highlight that the field of web scraping can be augmented with a lot of AI and algorithmic techniques. Implementing them would push the project from a straightforward pipeline to more of a research-grade system.

Roadmap & Milestones

Finally, to organize the implementation of all these improvements, we outline a possible roadmap:

Q4 2025 (Immediate – Next 1-2 months):

Focus: Core improvements and backlog fixes (the "Should Address Soon" items). - Implement dynamic throttling for Stage 1 heuristics and feature flags to toggle discovery modules [28] . - Add configuration validation and document all config options clearly. - Integrate PDF handling in Stage 2 & 3 (basic text extraction). - Boost test coverage to >80% and ensure CI passes with new tests. - Deploy monitoring: metrics logging and possibly a simple live log monitor. - **Milestone:** Be able to run a full UConn crawl end-to-end and produce enriched data including PDFs, with no crashes, and observe metrics/alerts for any issues.

Q1 2026:

Focus: Performance & Observability, plus moderate new features. - Optimize memory usage (perhaps remove Stage 1 .hashes file, tweak in-memory dedup sets). - Increase Stage 2 concurrency safely and test crawl speed improvements. - Build a basic web dashboard to view crawl progress in real-time (even if just reading a log or metrics JSON). - Improve NLP: switch to medium or large SpaCy model, and introduce summarization on a subset of pages (experimentally). - Implement and test alerting channels (maybe set up a Slack webhook for alerts in our dev environment). - Possibly containerize the app for easier deployment. - **Milestone:** Achieve ~30% faster crawl and enrichment throughput compared to baseline; have a monitoring system where the team can watch a crawl and get alerted if something fails.

Q2 2026:

Focus: Advanced capabilities and refactoring for extensibility. - Integrate Playwright for a subset of dynamic pages and measure increase in URL discovery. - Introduce plugin system skeleton: allow at least Stage 3 NLP plugin (so one can easily add a custom tagger). - Clean up any remaining UConn-hardcoded logic behind config or plugin (progress towards multi-domain support). - Possibly test on a second domain (maybe a smaller site) to see how adaptable the pipeline is. - Enhance content classification (maybe a simple model to tag categories, feeding into content_tags). - **Milestone:** The pipeline can handle a dynamic-heavy section that was previously impossible (demonstrating Playwright success), and one example plugin is working (e.g., a plugin that adds sentiment analysis to each page).

Q3 2026:

Focus: Scale and Intelligence. - If needed, move storage to PostgreSQL and test pipeline with it (especially if multiple users need to query data simultaneously). - Experiment with LLM integration in a small way (maybe a prompt-based data extraction for one tricky page type) to evaluate feasibility. - Implement continuous or scheduled crawling: ability to pick up new content without redoing everything (maybe via last_modified checks or incremental mode). - Work on documentation for open-sourcing (if planned), ensuring no sensitive info and that usage is clear. - **Milestone:** Pipeline is robust enough to run as a service (maybe running weekly updates on UConn site automatically), and advanced features (like partial LLM usage) are prototyped.

Beyond:

- Multi-university network crawling with cross-analysis (this would be a separate project or major extension).
- Possibly user interface for querying the data (could coincide with releasing a Q&A chatbot trained on the data).
- Continual improvement of ML/NLP as models evolve (maybe by 2026 there are even better open models to use).
- Ensure the project stays up-to-date with Python and dependency updates (Scrapy 3.x if it comes, etc.).

This roadmap is ambitious but achievable with focused effort, and it ensures we deliver incremental value at each step (not just big bang at end).

Success Criteria

To conclude, we define how we measure the success of all these improvements:

Technical Metrics

- **Coverage & Reliability:** 80%+ test coverage; no critical bugs escaping into production runs (as indicated by alerts or failures).
- **Performance:** The pipeline can crawl and enrich at least **100 URLs per minute** on average (including HTML and PDF content) on a single decent machine. This implies ~6000 URLs/hour, so even 100k URLs would finish in <1 day – which is decent. If using concurrency and optimizations fully, maybe even faster.
- **Scalability:** Able to handle a crawl of **1 million pages** if needed, by scaling horizontally or adjusting config, without running out of memory or crashing. (This might be theoretical unless we test on such scale or multiple domains.)
- **Data Quality:** 99% of pages with meaningful content are captured and enriched (meaning minimal misses). And irrelevant pages (like error pages, duplicates) are largely filtered out or flagged.
- **Extensibility:** To test this, try adding a simple plugin or switching domain – measure that minimal code changes are needed (ideally just config changes for a new domain crawl).

Operational Metrics

- **Runtime Stability:** 0 unhandled exceptions in a full crawl; pipeline can run to completion in an unattended manner.
- **Graceful Shutdown/Resume:** If stopped, can resume from last checkpoint with at most 5% reprocessing overlap (due to maybe one batch repeat).
- **Monitoring Coverage:** Key events and stats are visible on the dashboard; any failure triggers an alert within 1 minute.
- **Deployment:** Pipeline can be set up on a new machine or container within an hour (documentation and automation make it straightforward).
- **Maintainability:** New developers can understand the system within a day or two (based on documentation and code clarity), and adding a new feature doesn't require touching too many parts (indicative of good modularity).

User/Project Success

- **Utility:** The final data proves useful for answering student queries. For example, using the enriched data to answer 10 sample FAQs about UConn shows accurate responses (this indirectly measures our content completeness and correctness).
- **Adaptability:** When UConn's site changes (layout or new sections), the pipeline can adapt quickly (either automatically via robust scraping or with minor config updates, not a full rewrite).
- **Community & Contribution:** If open-sourced, the project garners interest – perhaps other universities want to use it (indicating we built something generalizable).
- **Longevity:** The pipeline remains in use over multiple semesters, feeding perhaps a chatbot or search tool that students actually use, with positive feedback.

By these criteria, we ensure we not only implement a lot of fancy features, but actually achieve the original goal: an **in-depth, comprehensive, up-to-date** data collection of UConn's web content that can empower knowledge applications.

We will continuously evaluate against these criteria as we progress, adjusting priorities if needed (this Future Plan is a living document, after all, open to revision as new challenges and opportunities emerge).

Document Version: 2.0 (Extensive Update after completing immediate priorities)
Last Review: 2025-10-01
Next Planned Review: 2025-12-01 (to assess progress on Q4 2025 goals)
Maintained By: Pipeline Development Team

---

1 6 7 28 32 future_plan.md

https://github.com/BenjaminSRussell/Scrapy/blob/3e4ad6f2b63e49d26f70181753b640d1496d41b3/Scraping_project/docs/future_plan.md

2 123 urls.py

https://github.com/BenjaminSRussell/Scrapy/blob/3e4ad6f2b63e49d26f70181753b640d1496d41b3/Scraping_project/src/common/urls.py

3   45   49   52   53   54   58   59   121   enrichment_spider.py

https://github.com/BenjaminSRussell/Scrapy/blob/3e4ad6f2b63e49d26f70181753b640d1496d41b3/Scraping_project/src/stage3/
enrichment_spider.py

4   19   20   21   22   23   24   25   26   27   29   30   31   34   35   36   37   38   50   85   108   109   119   discovery_spider.py

https://github.com/BenjaminSRussell/Scrapy/blob/3e4ad6f2b63e49d26f70181753b640d1496d41b3/Scraping_project/src/stage1/
discovery_spider.py

5   115   116   README.md

https://github.com/BenjaminSRussell/Scrapy/blob/3e4ad6f2b63e49d26f70181753b640d1496d41b3/Scraping_project/
README.md

8   95   96   97   98   99   100   101   metrics.py

https://github.com/BenjaminSRussell/Scrapy/blob/3e4ad6f2b63e49d26f70181753b640d1496d41b3/Scraping_project/src/
common/metrics.py

9   10   11   12   13   84   config.py

https://github.com/BenjaminSRussell/Scrapy/blob/3e4ad6f2b63e49d26f70181753b640d1496d41b3/Scraping_project/src/
orchestrator/config.py

14   15   16   44   55   63   89   124   125   schemas.py

https://github.com/BenjaminSRussell/Scrapy/blob/3e4ad6f2b63e49d26f70181753b640d1496d41b3/Scraping_project/src/
common/schemas.py

17   18   39   40   41   42   43   46   47   48   51   79   80   110   113   validator.py

https://github.com/BenjaminSRussell/Scrapy/blob/3e4ad6f2b63e49d26f70181753b640d1496d41b3/Scraping_project/src/stage2/
validator.py

33   56   57   66   111   112   122   development.yml

https://github.com/BenjaminSRussell/Scrapy/blob/3e4ad6f2b63e49d26f70181753b640d1496d41b3/Scraping_project/config/
development.yml

60   61   86   87   88   90   91   92   93   94   storage.py

https://github.com/BenjaminSRussell/Scrapy/blob/3e4ad6f2b63e49d26f70181753b640d1496d41b3/Scraping_project/src/
common/storage.py

62   117   118   discovery_pipeline.py

https://github.com/BenjaminSRussell/Scrapy/blob/3e4ad6f2b63e49d26f70181753b640d1496d41b3/Scraping_project/src/stage1/
discovery_pipeline.py

64   65   67   68   69   70   71   72   73   77   78   120   pipeline.py

https://github.com/BenjaminSRussell/Scrapy/blob/3e4ad6f2b63e49d26f70181753b640d1496d41b3/Scraping_project/src/
orchestrator/pipeline.py

74   81   102   main.py

https://github.com/BenjaminSRussell/Scrapy/blob/3e4ad6f2b63e49d26f70181753b640d1496d41b3/Scraping_project/src/
orchestrator/main.py

75   76   82   83   IMMEDIATE_PRIORITIES_COMPLETED.md

https://github.com/BenjaminSRussell/Scrapy/blob/3e4ad6f2b63e49d26f70181753b640d1496d41b3/Scraping_project/docs/
IMMEDIATE_PRIORITIES_COMPLETED.md

103   104   105   106   107   alerts.py

https://github.com/BenjaminSRussell/Scrapy/blob/3e4ad6f2b63e49d26f70181753b640d1496d41b3/Scraping_project/src/
common/alerts.py

[114] test_plan.md

https://github.com/BenjaminSRussell/Scrapy/blob/3e4ad6f2b63e49d26f70181753b640d1496d41b3/Scraping_project/docs/test_plan.md