ChatGPT

ConfRadar MVP – Product Requirements Document

Problem Statement

Researchers and academics often struggle to keep track of rapidly approaching conference submission deadlines and details. Conference information (e.g. Call for Papers dates, venues, submission deadlines) is scattered across many websites, mailing lists, and portals, making it time-consuming to manually monitor. Existing community-driven trackers (such as WikiCFP or field-specific deadline aggregators) cover only a small portion of events and are not always up-to-date [1]. Moreover, conference data is highly **dynamic** – dates and deadlines frequently change (e.g. deadline extensions, venue updates) [1]. Missing a deadline can mean lost opportunities for submissions, so an automated solution is needed to ensure researchers never miss important updates.

ConfRadar is an AI-powered conference tracking system designed to address this need. It will leverage web data mining and NLP to continuously gather conference information, extract key dates, detect changes over time, and present the latest data in one convenient place. By automating conference tracking, ConfRadar aims to reduce manual effort and ensure researchers have the freshest, most accurate conference deadlines at their fingertips.

Objectives and Goals

The overarching goal of ConfRadar's MVP is to **simplify and automate the monitoring of academic conference deadlines and information**. The MVP (Minimum Viable Product) will focus on the data mining backbone of the system. Key objectives and success goals include:

- **Comprehensive Coverage (High Recall):** Track a broad set of conferences in the chosen domain (e.g. all major computer science conferences), minimizing the chance of missing an important event. The system should go beyond static lists to discover new calls autonomously [1].
- **Accurate & Timely Information:** Provide reliable, up-to-date information that users can trust. The extracted dates and details must match official sources, and the system should update promptly when changes occur. ConfRadar should serve as a single source of truth for conference deadlines, notifying users of any extensions or updates as soon as they are detected.
- **Core Data Mining Capabilities:** Demonstrate the ability of an AI agent to perform web-based tasks that traditionally required manual effort. The MVP should showcase automated web retrieval, NLP-driven information extraction, change detection, and alias resolution working together. This will validate the technical approach (e.g. use of an LLM agent) for gathering and organizing information.
- **Maintainability & Extensibility:** Ensure the system is built with a modular, developer-friendly design for easy maintenance. Adding a new conference series or adapting to another research field (e.g. tracking deadlines in another domain) should require minimal effort. Clear separation of components and configuration (for known conference sources, aliases, etc.) will support future extension.

Note: While user-facing integration (e.g. a Google Doc or Notion page of deadlines) is part of the overall vision, the **MVP** will prioritize backend data mining components first. A simple output format (such as a generated document or JSON API) will be provided, but advanced UI/UX features are not primary goals for the MVP.

MVP Scope and Core Features

The ConfRadar MVP will implement the essential data mining pipeline needed for automated conference tracking. This section outlines the core features and capabilities included in scope, as well as what is considered out-of-scope for the initial version.

In-Scope (Core Features for MVP)

- **Automated Web Retrieval of CFPs:** The system will automatically fetch conference calls-for-papers and important dates from the web. This includes crawling known conference websites and using web search to discover new CFP announcements. The MVP will maintain a seed list of top conference series with their usual websites and will periodically check these sites for updates [2]. It will also perform targeted searches (via an API or scripted browser) for conferences not in the seed list, using keywords (e.g. “<field> conference 2025 deadline”) to ensure broad coverage [3]. The retrieval component will fetch page content (HTML or PDF) once a relevant link is found [4], supplying the raw text for extraction.

- **NLP-Based Information Extraction:** ConfRadar will parse unstructured conference announcements (webpages or documents) and extract structured data fields. The MVP will focus on extracting key attributes such as conference name, year, location, website URL, and **important dates** (submission deadline, abstract deadline, notification date, conference dates) [5] [6]. The extraction module will combine rule-based parsing (e.g. finding an "Important Dates" section in HTML) with NLP techniques to handle variations in format. Where necessary, an LLM (Large Language Model) will be employed to interpret content and output a structured JSON record [7]. This allows flexibility in parsing different page layouts (the LLM can locate and interpret dates even if the HTML structure varies [8]). The structured output will conform to a predefined schema (see **Data Schema** in Assumptions) to ensure consistency.

- **Conference Knowledge Base (Data Storage):** The system will store extracted information in a structured **knowledge base** (database). Each conference event will be represented as a record with all the extracted fields and metadata (e.g. timestamp of last update) [9] [10]. The MVP will implement this storage using a suitable database (for example, a relational database with tables for conferences and series, or a graph database to naturally represent series-event relationships). The knowledge base will enable querying and will serve as the memory of the agent for detecting updates. It will also maintain **version history** of data points to support change tracking (e.g. keeping both old and new deadlines when a change occurs for transparency [11]).

- **Alias Resolution & Event Clustering:** ConfRadar will include basic **alias resolution** logic to handle different names referring to the same conference series. For example, it should recognize that "NeurIPS" and "NIPS" or "IJCAI 2025" and "International Joint Conference on AI 2025" are the same event or series [12]. The MVP will group conference entries by series and link related events (such as workshops to their parent conference). This entails clustering records based on known acronyms or title similarity and assigning each event a series identifier. By clustering related events, the system avoids duplicate entries and provides a clean, consolidated view of each conference series.

- **Update Detection (Change Monitoring):** A critical feature of the MVP is the ability to detect changes in previously gathered data. The system will periodically re-run the retrieval and extraction for conferences in the database and compare new information to the stored record. If a deadline date is extended or any detail is updated on the official site, ConfRadar will detect the discrepancy and update the knowledge base accordingly [13] [11]. The system will record what changed (e.g. “submission deadline moved from Jan 10 to Jan 17”) and when. In the MVP, update detection might simply log changes or mark updated fields, and it can notify the user by updating the output data. This ensures data freshness and that users are alerted to schedule changes in a timely manner.

- **Basic User-Facing Output:** While the emphasis is on backend functionality, the MVP will include a minimal user-facing component to demonstrate the end-to-end system. This could be as simple as generating a static webpage, document, or console output listing upcoming conferences and their deadlines, updated continuously. For example, the system might populate a Google Sheet or a Markdown/HTML page with a table of conferences. The integration with a platform like Google Docs or Notion (as envisioned for the full product) will be rudimentary in the MVP – likely updating a single shared document with the latest data. This is mainly to validate that the data pipeline works; a more polished UI can follow in later iterations.

Out-of-Scope (MVP)

- **Full User Interface or App:** Building a dedicated web or mobile application, complex user dashboards, account management, or notification system (email alerts, etc.) is out of scope for the MVP. The focus is on the data pipeline; user interaction will be limited to viewing the aggregated results in a simple format (as described above). Any sophisticated UI/UX improvements will be considered in future versions after the core system is proven.

- **Advanced Recommendation or Analytics:** The MVP will not include features like recommending conferences to users, trend analysis of deadlines, or other analytical insights. Its purpose is to gather and present raw conference information. Features such as personalized recommendations or calendar syncing might be valuable, but they are beyond the initial scope.

- **Multi-Language and Multi-Domain Support:** Initially, ConfRadar MVP will target conferences in a specific domain (e.g. Computer Science/AI) and primarily in English. Handling conferences in other research fields or multiple languages (translations of CFPs) is not addressed in the MVP. It’s assumed that focusing on one domain will provide a controlled environment to develop the core capabilities, after which expanding to others can be evaluated.

- **Robust PDF Parsing or Non-web Sources:** Some conferences provide CFPs as PDF documents or solely via email/mailing lists. The MVP will primarily handle HTML web content (and possibly simple text PDFs if easily supported). Comprehensive PDF parsing (beyond basic text extraction) or integration with email list archives is out-of-scope for now, although the system’s design will keep the door open to add these data sources later.

By clearly defining the scope, the team can concentrate on delivering the key features above without distraction. The MVP will deliver a working end-to-end conference tracking backend, laying the groundwork for future enhancements in user interface and expanded functionality.

System Architecture Overview

ConfRadar's MVP will be built with a modular, pipeline-oriented architecture. At a high level, an **LLM-powered agent** will orchestrate several components that correspond to the core features described above [7]. The architecture is designed for continuous, autonomous operation: it will regularly gather new data, update the knowledge base, and output changes. The main components are:

- **Agent Controller (Orchestrator):** At the heart of the system is an agent (inspired by frameworks like LangChain) that coordinates the workflow [14]. This controller is responsible for deciding when to perform a web search, which site to crawl next, or when to trigger the extraction on a fetched page. It can incorporate a scheduling mechanism to run these tasks periodically (e.g. daily) or in response to triggers (like a known deadline approaching or passing). The agent may use an LLM to enhance decision-making – for example, formulating search queries, parsing content dynamically, or deciding if a piece of text likely contains conference information. The agent orchestrator ensures all components work together in sequence (retrieve -> extract -> store -> detect changes -> output).

- **Web Retrieval Module:** This component handles the discovery and fetching of conference information from the internet. It consists of two sub-parts: a **seeded crawler** and a **search-driven fetcher**. The seeded crawler periodically visits known URLs (official conference websites or annual pages for known series) to check for updates [2]. The search component uses a search engine API or browser automation to find new conferences or workshops not in the seed list [3]. For each discovered conference page or CFP link, the retrieval module fetches the content (making HTTP requests or using a headless browser if needed for dynamic content). Retrieved content (HTML, or text from PDFs) is then passed to the extraction module. The retrieval module must handle heterogeneous sources: some conferences list deadlines on a dedicated "Important Dates" section, others might post a PDF CFP, and some announcements appear on aggregator sites like WikiCFP [15]. This means the retriever will have fallback strategies (e.g. if the official site is not found or lacks info, try known aggregators or society pages). It also manages **rate-limiting** and politeness (respecting `robots.txt`, avoiding overloading any single site).

- **NLP Extraction Module:** The extraction module takes raw page content and pulls out the structured conference data. It will use a mix of parsing techniques:

- *Rule-based Parsing:* for known patterns like an HTML table of dates or an "Important Dates" list. For example, a simple scraper might find date strings in the vicinity of keywords like "Deadline" or "Conference Date".
- *LLM-Assisted Extraction:* for unstructured or varied formats, the module can employ a Large Language Model prompt to extract fields. The agent can provide the LLM with a predefined JSON schema and ask it to fill in values based on the page text [7]. Modern LLMs can return JSON following a schema if prompted correctly, enabling extraction of fields like dates, locations, etc., even when the page format is unfamiliar [16].

The combination of these methods will make the extractor robust to different page layouts. The output is a structured record for each conference event (see Data Schema under Assumptions for the list of fields). The extraction module also normalizes data (for example, converting dates to a standard format and time zone, ensuring conference names are consistent, etc.).

- **Knowledge Base / Database:** The extracted records are saved to a persistent store. The knowledge base acts as a central repository of all tracked conference events. For the MVP, a relational database (SQL) is a straightforward choice, with tables for conference series and conference events, including relationships (foreign keys) to link workshops to their parent conference. Alternatively, a graph database could naturally model series-event relationships (nodes and edges), but a SQL database will suffice initially. Key functionalities of the knowledge base component:
- Ensure each conference event is stored uniquely (avoid duplicates by checking if the same series/year already exists).
- Support queries by the agent (e.g. to check if a newly found conference is already known, or to find the last stored deadline for comparison).
- Store metadata like `last_updated` timestamps and possibly the raw content snippet for provenance.
- Maintain history of changes: the database design will allow storing previous values of a field when an update occurs (e.g. keep an archived record of the old deadline alongside the new one, or a separate table for changes).

By structuring the data, the system can later present it in user-friendly formats (such as tables or calendar feeds) and quickly answer questions (e.g. "find all conferences in April 2025").

- **Alias Resolution & Clustering:** This logic can be seen as part of the knowledge base management or a separate module. Its role is to group related entries and resolve naming variations. The MVP will implement a simple clustering algorithm that runs whenever a new event is added or during periodic maintenance:
- Maintain a dictionary of known conference series (with official names and common aliases). For example, the series "International Conference on Machine Learning" may be known by the alias "ICML".
- When a new conference is extracted, compare its title and acronym to known series. If it matches or is a close variant (e.g. acronym matches or title contains a known series name), link it to that series. If it's not recognized, create a new series entry.
- If a conference is a workshop (detected by its title or context, e.g. title contains "Workshop" and possibly a main conference name), then mark its `parent_event` field to link it to the main conference [17] [18].

The alias resolution ensures consistency (e.g., all editions of the same conference series are connected) and prevents duplication in the output. As a result, the system can present a clean list of unique conferences rather than scattered entries.

- **Change Detection Monitor:** Built into the agent's routine is the ability to detect updates. Each time the retrieval+extraction pipeline runs for a conference that is already in the database, the newly extracted data is compared with the existing record. If differences are found (e.g. the submission deadline field differs from what was stored), the change detector will flag this. The knowledge base can then update the record's fields to the new values and log the change (e.g. in a `changelog` table or by versioning the record with a new timestamp). The change detection component may also trigger a notification or mark the output (for instance, highlighting the conference in the output document as “Updated” or appending a note about the change). This component ensures the system not only accumulates data but also keeps it **fresh** and correct over time [13].

- **Output & Integration Module:** In the MVP architecture, this is a simple module that takes the curated data from the knowledge base and publishes it in the chosen output format. It could be a script that generates a Markdown or HTML page, or an integration with an external service (e.g. using Google Docs API to update a document, or Notion API to update a page). Each time new data is added or existing data changes, this module refreshes the output so the user can always access the latest information. While minimal in the MVP, this layer is important to close the loop of the system, demonstrating how the data can be consumed in practice. It will be designed such that swapping out or expanding to other output formats (like a web dashboard or an email alert system) is possible in the future.

Overall, the system architecture is **pipeline-driven** with clear separation of concerns between components. The LLM agent ties everything together, enabling an intelligent crawl-extract-update cycle. This design allows ConfRadar to run continuously and autonomously, scaling from initial seed conferences to broad discovery, while maintaining accurate and updated knowledge of conference deadlines. It also ensures that future improvements or components (e.g. adding a PDF parser, more advanced alias resolution, or a UI) can be integrated with minimal changes to the core system.

Functional Requirements

This section enumerates the functional requirements of the ConfRadar MVP. These requirements detail the specific capabilities and behaviors the system must exhibit:

- **FR1: Conference Discovery and Retrieval** – The system shall retrieve conference information from the web automatically. It **shall fetch Calls for Papers (CFPs) and important dates** from:
- *Known sources:* official conference websites and trusted CFP aggregator pages (e.g. WikiCFP, IEEE/ACM listings) for a provided list of conference series [19].
- *Web search:* use search queries (via API or an automated agent) to discover new conferences or workshops not in the known list. The system shall be able to conduct searches by keywords (conference name, acronyms, year, etc.) and follow result links to find relevant pages [3].

- The retrieval process shall handle different content formats, including HTML pages and simple PDF text, at minimum. Retrieved content should be stored or passed to the extraction component for processing.

- **FR2: Information Extraction and Parsing** – The system shall extract structured information from conference announcements. It must parse the retrieved content to identify at least the following fields for each conference event: **conference name, acronym, year, website URL, location, submission deadline (and any other key deadlines like abstract deadline and notification date), conference dates** [20] [21].

- The extractor should recognize date formats and convert them to a standard ISO date representation (including time zone if provided).

- It shall handle common patterns (e.g. an "Important Dates" list) via deterministic parsing, and use NLP or ML techniques for more complex text where needed (for example, using an LLM prompt to fill a JSON schema with extracted fields).

- If certain expected fields are missing or not explicitly stated (e.g. no separate abstract deadline), the system should handle gracefully by leaving them blank or null, and capturing at least the main submission deadline.

- **FR3: Data Storage and Management** – The system shall store all extracted conference data in a persistent data store (knowledge base). Each conference **event** must have a unique entry identified by series name and year (or a generated ID). The data store shall maintain relationships between data, including:

- Linking each event to its conference **series** (so that all annual editions of a conference are connected).
- Linking **workshops** to their parent conference event (if applicable).
- Storing a timestamp of last update for each event record [9] [22].

- The storage mechanism should allow querying by various fields (e.g. find events by year, by series, by upcoming deadline before a certain date, etc.). This implies using a structured database (SQL or similar) with indexed fields for efficient lookup.

- **FR4: Alias Resolution and Duplicate Handling** – The system shall recognize when different names refer to the same conference and handle them as one entity. For example, if one source uses an acronym and another uses the full name, they should be merged or linked to the same series. Similarly, if a workshop name contains the main conference name (or an abbreviation of it), the system should infer the relationship. This requirement ensures the output does not list duplicates. Concretely:

- The system shall maintain an internal mapping of known aliases (acronyms <-> full names) for major conferences, which can be configured or learned.
- When adding a new event, the system shall check against existing series names and acronyms. If a match or alias is found, the event is tagged as that series; if not, a new series is created.
- If an event is identified as a **workshop** (for example, title contains “Workshop” and the name of a known conference), the system shall record its `parent_event` or association to the main conference.

- The system shall avoid entering the same conference event twice; if a fetched page is found to be an update of an existing event, it should update that entry rather than create a duplicate.

- **FR5: Update Detection and Change Logging** – The system shall detect updates to conference information and handle them appropriately:

- It shall periodically re-fetch data for known conferences (e.g. on a schedule or when prompted by the user/agent) and compare the newly extracted data to the existing stored data.
- If any field has changed (e.g. deadline date, venue, etc.), the system shall update the stored record with the new value. The previous value should be recorded (either in a history log or by versioning the record) to maintain a change history [11].

- The system shall flag updated records so that the output or user can be alerted. For example, if a deadline is extended, the change could be highlighted or an annotation added (e.g. "Deadline updated on Feb 10").
- If new events are discovered (e.g. a new conference CFP is found), the system shall add them to the knowledge base and include them in the output.

- **FR6: Output Generation** – The system shall provide an aggregated view of the tracked conferences and their key dates as an output. For the MVP, this could be:

- A dynamically updated document (Google Doc, Notion page, or simple webpage) listing all upcoming conferences with their deadlines and details.
- The output should be structured in a readable format (e.g. a table sorted by deadline or grouped by field) so that users can easily scan upcoming deadlines.
- The system shall update this output whenever the knowledge base changes (new conference added or existing one updated). Consistency between the database and the output is required.
- The output does not need to support complex interactivity in MVP, but it must reflect the latest data and be accessible to the target users (e.g. the development team or a test user group).

- **FR7: Scheduling and Autonomy** – The system shall operate with minimal manual intervention. This means:

- A scheduler or automated trigger shall initiate the retrieval process on a regular interval (for example, a daily job that checks for updates or new CFPs).
- The agent/controller should be able to run through the entire pipeline autonomously: starting from a list of tasks (like "update all known conferences" or "search for new CFPs"), performing retrieval and extraction for each, updating the database, and finally refreshing the output.
- In case of any errors (e.g. a site is unreachable, or parsing fails for a page), the system should log the issue and continue with other tasks, ensuring one failure does not halt the whole update cycle. It should be possible to rerun or reattempt failed tasks in the next cycle.

Each of the above functional requirements contributes to the core promise of ConfRadar: automatically keeping a comprehensive list of conferences and deadlines accurate and up-to-date. The MVP will be considered successful if it can autonomously gather and refresh conference deadline data over a period of time (e.g. daily updates), with minimal errors or omissions in the tracked information.

Non-Functional Requirements

In addition to functional features, the ConfRadar MVP must meet certain non-functional criteria. These requirements ensure the system is performant, reliable, and maintainable in a real-world setting:

- **Scalability:** The architecture should handle an increasing number of conferences and data sources without significant degradation. In the MVP phase, the system might track on the order of tens to hundreds of conferences, but it should be designed with scaling in mind (e.g. adding more conferences or even other domains in the future). This affects choices like using efficient crawling (avoiding brute-force fetching), database indexing for quick queries, and possibly enabling parallel processing (fetching multiple sites concurrently).

- **Performance and Latency:** While real-time operation is not required, the system should update information in a **timely** manner. For example, once a conference update is posted online, ConfRadar should detect and reflect that change within at most 24 hours (with the possibility of more frequent checks for critical sources). The end-to-end cycle (crawl -> extract -> update DB -> publish output) should be optimized to complete quickly for a given run. Users viewing the output should not experience noticeable lag – if the output is on a platform like Google Docs, updates can happen asynchronously, but ideally the data refresh is completed within minutes for a moderate list of conferences.

- **Accuracy and Robustness:** The extraction pipeline's accuracy is crucial – dates, names, and other details must be correctly identified. The system should be robust to common variations or noise in web content. It must handle cases like missing fields or unexpected date formats gracefully (e.g. using context or defaults, rather than crashing or storing incorrect data). A target goal can be high precision and recall in extraction (e.g. >90% of important dates correctly extracted for known test pages) [23] [24]. Additionally, the system should avoid false updates (e.g. detecting a change where there is none, due to minor HTML differences) by using stable identifiers or text matching for change detection.

- **Reliability and Error Handling:** The system should run reliably over time:

- **Fault Tolerance:** If a particular site is down or a network request fails, the system should catch the error and retry later, without stopping the entire update process. Logging of failures and exceptions is required for debugging.
- **Idempotency:** Running the update process multiple times on the same data should not cause duplication or corruption. The design of the pipeline (especially database updates) should ensure that repeated crawls of the same conference either make no change (if no new info) or update cleanly (if there is a change).

- **Consistency:** Data in the knowledge base and the output must remain consistent. Partially updated information (e.g. updated DB but not output, or vice versa) should be avoided through transaction-like updates or by designing the output refresh to happen after all DB changes are done.

- **Maintainability:** Developer-friendliness is a priority for the MVP. The codebase should be organized into modules corresponding to components (crawler, extractor, DB interface, etc.), with clear interfaces between them. Configuration (like the list of seed conferences, user API keys for external services, or extraction rules) should be isolated from code, possibly in JSON/YAML config files or environment variables. This allows adjusting the system (e.g. adding a new seed site or changing a parsing rule) without modifying core logic. Unit tests or at least manual test cases for critical functions (date parsing, alias matching) should be provided to facilitate future changes. Documentation in code and a README describing setup and usage are expected.

- **Extensibility:** The MVP should be built such that new features can be added with minimal refactoring. For example, if later we want to integrate a new data source (say, a conference calendar ICS feed or a new aggregator website), the retrieval module should be able to accommodate that (perhaps by adding a new fetcher class or plugin). Similarly, if a decision is made to incorporate a different NLP model or an upgraded LLM, the architecture should allow swapping that component without a complete rewrite. Use of standard interfaces or abstract classes for key parts (like a generic `ContentExtractor` interface that could have implementations using regex/spaCy or using LLM) will facilitate this.

- **Security and Compliance:** Although the data involved is public conference information (not sensitive user data), the system should still adhere to good security practices. API keys (for search APIs, Google Docs/Notion integration, etc.) must be stored securely (not hard-coded in code, but in environment configs). The crawler should obey website rules (honor `robots.txt` where applicable, and not scrape pages that disallow scraping). If many requests are made, they should be rate-limited to avoid overwhelming sources or getting IP-blocked. Additionally, any open-source licenses of dependencies (e.g. libraries for web scraping or NLP) should be respected, and the project's own license should be clarified if it's to be shared publicly.

- **Latency in Change Propagation:** When a change is detected in a conference's info, the propagation of that change through to the user-facing output should be fast. The requirement here is not interactive real-time, but if a deadline is extended, a user checking ConfRadar's output later that day should see the new date. This implies the scheduling frequency (e.g. daily jobs) and the performance of the pipeline are aligned with user expectations for "freshness" (no more than 1-day lag for critical changes, as an initial benchmark).

- **Resource Efficiency:** As an AI-powered system, some components (like LLM calls or browser automation) can be resource-intensive. The MVP should be mindful of costs and resource usage. For instance, if using an external LLM API, minimize the number of calls by extracting multiple fields in one prompt if possible, and only invoke the LLM for pages that truly need it (maybe try regex/rule parsing first, and resort to LLM if that fails). Similarly, the system should avoid re-processing unchanged pages unnecessarily – implementing checksums or `last-modified` checks on pages can save work. Efficient operation will be important if the system scales up to more conferences or is run continuously.

In summary, these non-functional requirements ensure that ConfRadar MVP is not only feature-rich, but also **stable, efficient, and easy to work with**. Meeting these criteria will lay a solid foundation for evolving the prototype into a production-ready tool.

Technical Stack Recommendations

To implement the above requirements, we recommend the following technologies and tools for the ConfRadar MVP. This stack is chosen for developer productivity, alignment with the AI/NLP needs, and ease of future scaling:

- **Programming Language: Python** is recommended as the primary language. Python has rich libraries for web scraping (e.g. `requests`, `BeautifulSoup`, `Scrapy`), NLP (spaCy, NLTK, transformers), and integrates well with machine learning and AI tools. Its ecosystem will speed up development of the data mining components. Additionally, Python is well-suited for scripting the periodic tasks and orchestrating different modules.

- **Web Crawling & Retrieval:** Use Python HTTP clients like `requests` for simple page fetches and **BeautifulSoup** for parsing HTML content. For more complex crawling needs, a framework like

Scrapy could be used to manage queues of URLs and parallel requests. If some conference pages require running JavaScript (e.g. a site that loads content dynamically), consider using **Playwright** or **Selenium** for headless browser automation on those specific sites. The system can combine approaches: try a lightweight fetch first, and fall back to a headless browser only if necessary (to keep things efficient). For the search capability, one option is to use a search API (like Google Custom Search API or Bing Web Search API) to get relevant pages. This avoids scraping Google results HTML directly (which is against terms of service). Python libraries or SDKs for these APIs can be used. If using an LLM agent for search (LangChain-based), it might simulate searching by iteratively querying an API and following links.

- **Natural Language Processing:** Leverage existing NLP libraries for parsing dates and names from text. For example, use **spaCy** with pre-trained models to detect date phrases, locations, etc., or the **dateparser** library to normalize date strings. However, a highlight of the tech stack is integrating a **Large Language Model (LLM)** for extraction. We suggest using the OpenAI API (e.g. GPT-4 or GPT-3.5) or a locally hosted model via HuggingFace Transformers for the LLM-assisted parsing. The LLM can be prompted with the page text and asked to extract fields in JSON format (taking advantage of techniques like few-shot examples or OpenAI's function calling interface). The **LangChain** framework is a strong candidate to implement this logic [14], as it provides abstractions for chaining prompts, tools, and managing the agent's memory. LangChain can help structure the extraction as a conversation where the agent can ask clarification or re-search if something is missing (though a simpler single-shot prompt might suffice for MVP). If using LangChain, the retrieval module can be one tool and the extraction can be another in the agent's toolkit.

- **Database / Storage:** For simplicity and reliability, a **PostgreSQL** or **MySQL** database is recommended to store the conference data. An object-relational mapper (ORM) like SQLAlchemy (if using Python) can speed up development and make it easy to switch DB backends if needed. PostgreSQL is preferred for its JSON support (could store raw JSON of CFP if needed) and robust indexing options (useful for text search or similarity if doing alias matching in SQL). If the data model is straightforward (a few tables), using a lightweight SQLite database could be acceptable for the MVP (especially for a single-user prototype environment), but moving to Postgres is better for multi-user or cloud deployment scenarios. In the future, if representing complex relationships, a **graph database** like Neo4j or an RDF store could be explored, but that may not be necessary at MVP scale. The schema design (see Data Schema in Assumptions) will guide the database table structure.

- **Alias Resolution & Clustering:** Implement initial alias resolution in Python using simple approaches: e.g., maintain a JSON or YAML file of known conference series names and aliases. Load this into a dictionary at runtime for quick lookup. For more dynamic matching, use string similarity measures (Python's `difflib` or `Levenshtein` distance) to catch cases where names are very similar. If needed, use spaCy or other NLP to compare name tokens (for instance, to see that "International Joint Conference on Artificial Intelligence" and "IJCAI" share capital letters). Another trick is to use embeddings: a sentence-transformer model could encode conference titles and cluster them by cosine similarity. This might be overkill for MVP, but it's an option if manual alias lists prove insufficient. The **scikit-learn** library can be used for clustering algorithms if needed (like DBSCAN or hierarchical clustering on name embeddings). However, given the manageable scope, static rules combined with careful data entry (ensuring each series is identified by a unique key) might suffice initially.

- **Change Detection:** This can be implemented at the database layer or application layer. A simple approach: whenever new data is extracted for an event, compare it with the old data loaded from the DB (in Python). Alternatively, use the database to do comparisons (e.g. keep an `old_value` and `new_value` column during updates). Since this is a custom logic, embedding it in the Python code that updates the DB is fine. For logging changes, consider having a separate table for changes with fields: event_id, field_name, old_value, new_value, timestamp. This is straightforward to implement with SQL. No specialized library is needed for the detection itself; it's mostly comparing Python dicts or ORM objects.

- **Deployment & Scheduling:** During development, running the system on a local machine or server manually is fine. But for continuous operation, deploying on a cloud server (like an AWS EC2, Heroku, or a Python-friendly platform) is recommended. Use a scheduler like **cron** (simple cron jobs) or a lightweight task scheduler library such as `APScheduler` within the app to trigger periodic updates. If using cloud, AWS Lambda or Google Cloud Functions could even schedule a daily run of the agent (if the task can complete within their execution time limits), though a persistent service is easier for continuous checks. Containerizing the application with **Docker** can help with reproducibility – developers can run the same container locally or on server. Given developer audience, including a `Dockerfile` in the project might be useful for environment setup.

- **APIs for Integration:** For output, if choosing Google Docs or Sheets as the presentation, use the official Google APIs (`google-api-python-client` library) to programmatically write to a doc/spreadsheet. This requires OAuth credentials setup, which the developer can handle and store securely. For Notion, use Notion's API and an SDK like `notion-client`. These integrations will need API keys/tokens, so ensure they are configurable. If the output is just a static HTML page, the tech stack could simply be generating the HTML from a Jinja2 template and hosting it via GitHub Pages or an S3 static site, etc. The choice of output integration will determine which API to use, but these are relatively self-contained and can be swapped.

- **Logging and Monitoring:** Use Python's `logging` library to record the system's activity – for example, logging when a conference page is fetched, when a new conference is found, or when a change is detected. This will be crucial for debugging. In a production scenario, you might integrate with monitoring tools or at least output logs to a file or console that can be reviewed. For MVP, console logging is sufficient. It's also recommended to use a tool like **Git** (with a GitHub repository) for version control of the code [25], which is already planned. This allows tracking issues and collaborative development if needed.

- **Testing Tools:** During development, using Jupyter notebooks or interactive Python sessions can help refine the extraction logic (especially for LLM prompts). Once solidified, the code can be moved into script/module form. For testing, utilize Python's `unittest` or `pytest` to write a few tests for, say, date parsing and alias resolution functions. While full test coverage is not required in MVP, having some automated tests ensures that future changes (like upgrading the LLM model or changing a library) won't silently break core functionality.

In summary, the technical stack centers on Python for flexibility and rapid development, leveraging modern AI/ML frameworks for the "smart" parts of the system and proven tools for web crawling and data management. This stack is chosen to minimize reinventing the wheel – using existing libraries and services where possible – and to allow a single developer or small team to build the MVP efficiently.

Assumptions and Constraints

To clarify the context and limits of the MVP, the following assumptions and constraints have been identified. These represent conditions we take to be true for the project and factors that may restrict the solution space:

- **Domain Focus Assumption:** The MVP will focus on **computer science conferences** (especially in AI/ML, since those have frequent deadlines and existing lists like AI Deadlines [26]). We assume most target conferences will publish their CFPs in English on public websites. Other domains (medicine, physics, etc.) or non-English CFPs are out-of-scope initially. This focus allows us to leverage common sources like WikiCFP, AIdeadlin.es, and conference portals, and to fine-tune extraction rules for the styles commonly used in CS/AI conference announcements.

- **Data Availability:** It is assumed that the necessary conference information is publicly available on the web without login or paywall. All information we need (dates, locations, etc.) should be obtainable from either official conference sites, CFP announcements, or community lists. If some conferences only disseminate info via email or private channels, those may not be covered in MVP. We also assume that the volume of data (number of conferences and frequency of changes) is manageable with the chosen approach (i.e., a daily crawl will not miss critical updates and the system can process all target pages in a reasonable time frame).

- **Seed List and Knowledge Base Initialization:** The project assumes an initial seed list of known conference series will be provided or gathered during development (e.g. a list of top 50 CS conferences with their names and URLs). This will bootstrap the crawling process. Completely "discovering from scratch" is not expected in MVP – there is at least some curated knowledge to start with. The quality of this seed list will affect coverage; we assume it will be reasonably comprehensive for major conferences. Also, the knowledge base may be pre-filled with some known data to test extraction (for instance, using past conference deadlines from AIdeadlin.es as a baseline), although that's optional.

- **Time & Resource Constraints:** The MVP is being developed within a limited time frame (on the order of a few months, see Timeline). This imposes constraints on the depth of certain features:

- The NLP extraction might not cover every edge case or minority formatting scenario in CFPs, focusing instead on the most common patterns.
- The use of LLMs is constrained by API availability and cost – we assume we have access to an OpenAI API or similar for development/testing, but heavy reliance on LLM for every page could be costly. We'll constrain usage to where it adds clear value.

- Computational resources: The MVP will run on a standard development machine or a single cloud VM. We are not assuming a distributed system or multiple servers. This means the solution must be efficient enough to run on one machine (possibly multi-threaded for crawling, but no cluster or big data infrastructure).

- **Accuracy vs. Effort Trade-off:** We assume that a **good-enough** level of accuracy in extraction is acceptable for MVP. That is, if occasionally a deadline is parsed incorrectly or a lesser-known conference is missed, it's not catastrophic. The goal is to automate the majority of the work. We won't implement extremely complex ML just to gain a few percentage points in accuracy if a simpler rule covers 90% of cases. This trade-off is acceptable in the short term, given MVP status, and can be revisited with more time or data.

- **User Interaction:** We assume users (initially likely the developers themselves or a small test audience) are fine with a static output (doc or page) and do not need interactive features in MVP. The expectation is that users will consult the output to plan submissions, but they won't be feeding input into the system (no user queries, custom filters, etc. in this version). This simplifies the design – for instance, we don't need a query interface or authentication in the MVP.

- **Ethical and Legal Constraints:** We must consider that crawling some websites might have legal/ ethical constraints. We assume that by targeting conference info, which is publicly intended for dissemination, we are not violating any usage policies as long as we crawl politely. We will adhere to `robots.txt` where applicable and include appropriate User-Agent identifiers. Also, storing and republishing the data is assumed to be permissible because it's factual public information (dates, locations) and we will cite sources if needed in the output (or at least provide links to conference websites). The MVP will not include any data that is not publicly available or that violates privacy.

- **Integration Constraints:** If integrating with Google Docs or Notion, we assume the availability of API access and credentials. For Google's API, daily update quotas should be sufficient (the amount of data is small, just text of deadlines, so well under typical API limits). But we need to be mindful of not hitting rate limits if we do frequent updates. Similarly, Notion's free tier has limits on updates which should be fine for MVP scale. We also assume the team has or can create the necessary accounts and credentials for these services. Setting up these integrations might require some upfront time, so the plan is to use whichever is easier for MVP (Google Sheets is often a quick win for tabular data, for example).

- **External Dependencies:** The project depends on external services (search API, OpenAI API, etc.). We assume these services remain accessible and stable during development. If the search API has a limit, we'll work within it (for instance, limit to a certain number of queries per day). If needed, we can cache search results to avoid repeated calls. The reliance on an external LLM means internet access and API keys are required for the system to function; the MVP is not designed to run completely offline.

Understanding these assumptions and constraints is important for setting correct expectations. They highlight why certain design decisions are made (e.g. focusing on one domain, not building a full UI) and what factors might impact the project's success (e.g. quality of seed data, availability of APIs). During development, if any assumption proves false or a constraint becomes too limiting, we will document it and adjust scope or design as needed.

Open Questions / Dependencies

Despite careful planning, there are a few open questions and external dependencies that need resolution as we develop the ConfRadar MVP. These are areas requiring further clarification or decisions to be made:

- **Optimal Source for Initial Conference List:** We need to determine the best way to obtain a comprehensive list of conferences to seed the system. Options include manually curating from sources like WikiCFP or AI Deadlines, using an official list from an academic society, or scraping an existing aggregator. This is a dependency because a good seed list will vastly improve coverage. Open question: *Should we integrate directly with WikiCFP's data via scraping or API (if any) as a starting point, or rely purely on our own web searches?* There is a risk that purely starting from scratch via search may miss some events or spend more resources.

- **Web Search Mechanism:** We must decide on the search strategy for discovering new conferences. The options are:

- Use a **paid API** (like Google Custom Search) vs. a **free approach** (like scraping Google results or using Bing's free tier). Using an official API has stability and legality benefits but incurs cost and sometimes lower query limits. On the other hand, scraping search results is against terms of service and can be blocked. This is an open question to resolve early, as it affects implementation. A possible approach is to use an API during development (for reliability) and consider more sophisticated agent search later if needed.

- If using an LLM agent to interpret search results (LangChain approach), we need to validate how effective that is in practice. Dependency: having API access to an LLM with browsing capability (OpenAI's GPT-4 with browsing or tools, for example) or implementing our own tool loop with LangChain.

- **LLM Choice and Parameters:** It's yet to be decided which LLM to use (GPT-4 vs GPT-3.5, or a local model like LLaMA 2). This depends on factors like cost, performance, and ease of integration. GPT-4 offers better understanding (likely higher extraction accuracy) but costs more and has rate limits; GPT-3.5 is cheaper and faster but might make more errors in extraction. If using OpenAI, we need to manage API keys and consider error handling for API failures. Alternatively, using a local model (with something like HuggingFace pipeline) avoids external calls but might require hosting a large model which is resource-heavy. This trade-off remains an open question. For MVP, leaning towards OpenAI API for convenience, but we must keep an eye on usage. We also need to define the **prompting strategy** for extraction (how to format the prompt and whether to use one prompt per page or multiple). Some experimentation will be needed to get this right.

- **PDF Handling:** As noted, some CFPs or important date announcements might be PDFs. The open question is how much we should support PDFs in MVP. If many target conferences post a PDF, ignoring them would reduce coverage. We might use a PDF-to-text tool (like PDFPlumber or PyMuPDF) to extract raw text and then feed that to the same NLP pipeline. The dependency here is having a reliable PDF parsing library. We need to evaluate how well these tools work on typical CFP PDFs (which might have columns or weird formatting). If it's too much work to reliably parse PDFs (which can be a project on its own), we might decide to skip or handle only simple cases. This decision will be made based on how many PDFs we encounter in our seed list during early testing.

- **Frequency of Updates:** How often should the agent run? While daily seems a reasonable default, some conferences might change more rapidly or new CFPs could appear at any time. Should we run the crawl twice a day? Hourly is likely unnecessary and would waste resources. This is an adjustable parameter, but the open question is if we allow user configuration (probably not in MVP) or just hardcode a frequency. It also ties to resource usage if using APIs (we have to ensure we don't exceed quotas). As a dependency, if using a free tier of an API, the number of calls per day might cap how often we can run the full pipeline.

- **Output Format and Platform:** We have a couple of choices for the user-facing output and need to decide which to implement for MVP:

- **Google Sheets/Docs:** A Google Sheet could nicely tabulate deadlines and is easy to share. A Google Doc could be more free-form but still fine. Using Google's API requires OAuth setup but is well-documented.
- **Notion page:** Notion has a nice interface and the API would allow structured updates, but fewer academics use Notion widely compared to Google, perhaps. Also, Notion's API and data model might be slightly more complex for tabular data.
- **Static Web Page:** Simply generate an HTML page and host it (like via GitHub Pages). This avoids external dependencies but is not real-time collaborative.
- **Console/Command-line output:** just for development testing.

The decision depends on ease of implementation and what the target users prefer. For MVP demonstration, a Google Sheet might be simplest (everyone can open it, and the data can be pushed via API easily). This is an open question to confirm with stakeholders (e.g. if the professor or test users have a preference).

- **Maintaining Data Quality:** We have an open question around how to ensure data quality over time. For example, if a conference's dates change multiple times (deadline extended twice), do we keep all history? How to avoid cluttering the output with too much info vs. keeping it minimal? Perhaps the MVP will just show the current dates and maybe a flag "extended" if applicable, but not all previous deadlines. We need to decide on a policy for handling multiple changes. It's a design choice that might be refined after seeing some real examples. This touches on how we present changes to users (maybe an open question: do we need a separate change log for users, or just internally track it?).

- **Collaboration and Open Sourcing:** There is mention of possibly open-sourcing the tool eventually [27]. While not directly affecting the MVP functionality, if the intent is to open source, we need to be mindful of things like removing any hard-coded secrets and writing clear documentation. The open question is if the MVP is solely for internal use or if it's intended to be released to the community upon completion. If the latter, additional tasks like writing a proper README, usage instructions, and selecting an open source license will be needed. This decision can be made towards the end of MVP development.

- **Future Integration Hooks:** We should consider any dependencies that might not be needed now but could affect design if we anticipate them. For example, integration with a calendar (iCal feed) or email notifications is out-of-scope, but if we think it's likely later, perhaps we ensure our system can easily produce an iCal feed from the data. This isn't a pressing question for MVP, but worth keeping in mind so we don't paint ourselves into a corner.

These open questions will be addressed as development progresses. Some (like output format and search API choice) need early decisions, while others (like handling multiple changes or open-sourcing) will be resolved later in the project. We will track these issues and update the requirements or design accordingly once decisions are made or more information is obtained.

Milestones / Timeline

The development of ConfRadar MVP will be organized into a series of milestones with an estimated timeline of around **6-7 months** (typical for an academic project, though in a lean startup setting an MVP could be iterated faster). Each phase will produce specific deliverables, reducing risk by tackling the project in manageable chunks. Below is the proposed timeline:

1. **Month 1-2: Requirements Refinement & Design** – In the initial phase, we will finalize requirements (based on this PRD) and conduct background research. Tasks include:
2. Performing a literature review of related tools and prior work (e.g. existing deadline aggregator sites, any academic papers on information extraction from similar domains).
3. Validating the feasibility of using an LLM agent (maybe prototyping a quick LangChain example of extracting data from a known conference page).
4. Designing the high-level system architecture and data schema [28]. Decide on the tech stack specifics (confirm language, DB choice, which APIs to use).
5. Outcome: A detailed design document or architecture diagram, and possibly some sample input/output examples to ensure understanding (for instance, take one conference page and manually create the JSON output – this will act as a target for the extraction module).
6. **Month 2-3: Data Source Integration & Web Retrieval Module** – Focus on building the web crawling and retrieval capabilities:
7. Compile the initial list of conferences to track. Implement scripts or use APIs to fetch content from their websites [29].
8. If using a crawling framework, set it up and write spiders for a few representative sites. Also set up the search mechanism (e.g. code for calling the search API and parsing results).
9. Stand up the database and create tables for the knowledge base. Write a simple data access layer (using an ORM or raw SQL) to be able to insert and query conference records.
10. By the end of this phase, we should be able to run a job that goes out to a couple of known conference sites, downloads the pages, and stores some placeholder info in the DB. This might not yet extract all fields (that's next phase), but we validate that connectivity and data flow from web to storage works.
11. **Month 3-4: NLP Extraction Pipeline Implementation** – Develop the core extraction logic:
12. Start with rule-based extraction for structured content. For example, implement a function to parse an "Important Dates" HTML section (perhaps using BeautifulSoup to find lists or table rows containing dates).
13. Integrate an NLP library or write regex patterns for common date formats. Test these on sample pages.

14. Introduce the LLM component: define prompts and use the OpenAI API (or alternative) on a few saved pages to see if it can output the needed fields [30]. Iterate on prompt design to improve accuracy.
15. Implement a combined extraction function that tries rule-based parsing first, and falls back to LLM for unstructured or tricky cases.
16. Validate extraction on a small set of pages: compare the output JSON to manually curated correct data (this serves as an evaluation of accuracy; metrics like precision/recall can be computed on this sample [23]).

17. By end of month 4, the system should be able to process raw HTML from several conferences and produce structured records with most fields correctly populated.

18. **Month 4-5: Alias Resolution and Clustering** – Build the logic to unify and link conference entries:

19. Create the data structures for series and alias mapping. This might involve making a separate table for series in the DB and a script to populate it with known aliases (from the seed list).
20. Implement the code that, after extraction of a new event, checks against existing series names/aliases to assign it appropriately [31].
21. Develop clustering for workshops: define heuristics to detect workshops (e.g. if title contains known conference acronym + "Workshop"). Update the data model to link workshops to main events (e.g. set the parent_event field) and ensure the DB can store such links.
22. Test the alias resolution by feeding sample inputs like different spellings of the same conference and seeing if the system recognizes them as one. Adjust the rules or add aliases as needed.

23. Outcome: The knowledge base now maintains clean, non-duplicate entries. For example, if we input "NeurIPS 2025" and "NIPS 2025" from different sources, the system would end up with one series (NeurIPS) and one event for 2025, rather than two separate entries.

24. **Month 5-6: Integration & Agent Automation** – Bring all components together and enable the agent to run the whole pipeline:

25. Integrate the retrieval, extraction, and DB update flows into a single agent loop or script. Essentially, develop the main routine that the scheduler will call: it goes through the list of sources (or series), fetches pages, extracts info, updates the DB, resolves aliases, and then generates the output.
26. Add the scheduling trigger (for development, this could just be a loop with a sleep or a cron job).
27. Implement the output generation: choose the platform (e.g. Google Sheets) and write the code that queries the DB and formats the data into the output. Test the output manually to ensure it's correctly formatted and contains all expected info.
28. Conduct a dry-run: simulate the agent running as intended. This might involve running it on a test set of 5-10 conferences repeatedly (maybe daily) to see how it behaves. This is where we might catch issues like memory leaks, incomplete updates, or mis-identification of changes.

29. By end of month 6, the ConfRadar MVP should be essentially **feature-complete**: all major functionalities implemented and working in unison. We should have a demo where the system automatically updates a document with conference deadlines over a period of days, proving that it can track changes.

30. **Month 6 (overlap with 7): Testing, Evaluation, and Refinement** – In this phase, we thoroughly test and improve the system:

31. Evaluate the system on a larger set of conferences (if earlier phases used 10-20, now try with say 50). Measure how many it successfully tracked, and check the data for correctness.
32. Use the metrics defined (precision/recall of extraction, timeliness of updates) to quantify performance [23]. Identify where errors are coming from – e.g. are certain pages failing to parse? Are certain fields often wrong? Use this analysis to tweak the extraction rules or add aliases.
33. Collect any user feedback if the output has been shared with colleagues (perhaps see if they notice missing conferences or confusing output).
34. Improve the system based on feedback: e.g. add any high-priority alias that was missing, adjust the schedule if needed, or refine the formatting of output for clarity.

35. The goal is to enter the final month with a robust MVP that meets the requirements. By now, most critical bugs should be fixed and the system should handle the common scenarios reliably.

36. **Month 7: Documentation and Deployment** – Finalize the project for handoff or launch:

37. Prepare documentation: a user guide (how to view and interpret the output, how to add a new conference to track), and a developer guide (how the system is designed, how to set up and run the code, etc.). This PRD can be updated to reflect any changes.
38. Ensure the code repository is clean, well-organized, and possibly made public (if open-sourcing). Remove any hardcoded secrets, ensure configs are in place.
39. Deploy the MVP in a persistent environment if not already. For example, host it on a cloud VM that runs continuously, or set up a Docker on a server with a cron job to run the update daily. The deployment should also consider monitoring – maybe set up simple alerts if the agent fails (this might be as simple as an email if the script crashes, for MVP).
40. Conduct a final review meeting/presentation (if this is an academic project milestone) demonstrating the working MVP: show how a new conference added gets picked up, how a changed deadline is detected, and the final output document that users can see.
41. Mark the completion of MVP scope. Any features or improvements not done are backlogged for future work.

Throughout these milestones, the approach will be iterative. For instance, some extraction improvements might happen earlier or in parallel with alias resolution. The timeline is a guideline; overlaps are expected (e.g. while waiting for API access, one can work on DB schema, etc.). Regular check-ins (weekly or bi-weekly sprints) will ensure we stay on track with the timeline.

By following this timeline, we plan to deliver a functioning ConfRadar MVP by the end of Month 7, at which point it can either be used by the target users (researchers tracking deadlines) and/or serve as a strong prototype for further development into a full product.

---

[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28] [29] [30] [31] ConfRadar.md
file://file_000000006ccc61f7b8d9c9c00b7ef7bb