ChatGPT

Project Brief: Web Content Capture Extension with Pyodide and Serverless Backend

Overview

This project (led by Dinis Cruz) is a **Chrome extension with a serverless Python backend** designed to capture and archive web content for authorized sites. It operates by monitoring page loads on whitelisted websites (e.g. news sites) and extracting the page's HTML and JavaScript. The captured content is then sent to a stateless FastAPI backend and stored in **AWS S3**. By leveraging **Pyodide** (Python in WebAssembly) within the browser, the extension runs most logic client-side in Python, including content hashing and even test routines. The architecture's goal is to create a robust pipeline that **preserves raw web content with minimal server-side processing**, enabling future analysis (such as semantic knowledge graphs or provenance checks) while avoiding duplicate storage through content hashing [1] [2]. This API-first approach of separating content capture/storage from presentation or analysis aligns with recommended patterns for content providers, supporting efficient delivery and downstream knowledge graph operations [3].

Goals and Objectives

- **Capture Authorized Content:** Monitor and intercept content from specific allowed websites (e.g. partner news sites) during page loads. Ensure that only authorized domains are captured to respect privacy and permissions.
- **Preserve HTML and Scripts:** Record the full HTML of the page along with any loaded JavaScript files or dynamic content, providing a snapshot of what the user sees and what runs on the page. This supports future fact-checking, analysis, or replay of the content as originally delivered.
- **Minimize Redundancy:** Avoid storing duplicate content. Each captured page is hashed (e.g. using SHA-256) to create a content fingerprint. If the same content has been captured before, the system will detect it via the hash and skip re-uploading, conserving storage and avoiding redundant data.
- **Client-Side Processing:** Perform as much logic as possible in the browser via Pyodide. This includes computing hashes, preparing data payloads, and even running unit/integration tests for the capture logic. Offloading work to the client reduces server load and costs, and ensures the capture logic can be easily tested in an environment identical to runtime (the browser).
- **Stateless Scalable Backend:** Use a serverless FastAPI backend purely as a transient proxy to storage. The backend does not maintain state between requests; it simply accepts content from the extension and writes it to S3. This stateless design makes scaling and deployment easier and more secure (there's no persistent server database to protect) [4].
- **Structured Storage & Indexing:** Organize stored files in S3 by timestamp (year/month/day/hour/minute/second) to provide an inherent versioning and ordering of captures. An index (JSON file) is maintained to map each capture's metadata – such as URL, timestamp, and content hash – to its storage location. This index enables quick lookup and avoids re-capturing or re-storing known content.

System Architecture and Components

The system is composed of three main components working in tandem. The design emphasizes clear separation of concerns: the **browser extension** handles data collection, the **in-browser Python (Pyodide)** handles processing logic, and the **serverless backend with S3** handles storage. This separation mirrors an "API-first" content architecture, where raw content capture/storage is one layer and higher-level analysis or verification can be layered on later [3].

```
graph LR
    subgraph Browser
        A[Chrome Extension<br/>(Content Script + Pyodide)]
        B[Authorized Website<br/>(User visits)]
        A -- "Capture HTML & JS" --> C((Pyodide<br/>Python Logic))
    end
    A -. injects .-> B
    C -->|hashed content & metadata| D[FastAPI Backend<br/>(AWS Lambda)]
    D -->|store files| E[(AWS S3 Bucket)]
    D -->|update index JSON| E
```

Figure: High-level architecture. The browser extension (with an in-browser Python layer) captures content from the visited page and sends it to a FastAPI service, which writes the data to S3 storage. The numbered sections below describe each component in detail and how data flows through the system.

1. Chrome Extension (Content Capture)

The Chrome extension is kept minimal and focused. It uses a content script (and/or background script in Manifest V3 service worker) that activates on authorized domains. When the user navigates to a target site, the extension listens for page load or network events. It can intercept HTTP responses (using Chrome's WebRequest APIs) or read the DOM after load to retrieve the page's HTML. It also intercepts or collects any external JavaScript files loaded by the page. The result is a collection of content: the main HTML document and any relevant script content. This raw data is passed to the in-browser Python layer for processing. The extension's JavaScript essentially acts as a glue between the browser environment and the Python logic, initializing Pyodide and handing off the captured data. By doing the capture at the browser level, we ensure we get the exact content as delivered to the user (including any dynamic HTML populated by JavaScript by the time onload triggers).

2. In-Browser Python via Pyodide

Using **Pyodide**, the extension is capable of running Python code directly within the browser. Pyodide loads a WebAssembly-based Python interpreter, allowing us to leverage Python's rich ecosystem (for tasks like hashing, text processing, etc.) on the client side [5]. The Python code (which can be bundled with the extension or loaded at runtime) receives the raw HTML/JS captured by the content script. Key logic implemented in Python includes:

- **Content Hashing:** The Python code computes a secure hash of the content (e.g. combining the HTML and scripts text) using Python's `hashlib`. This hash will serve as a unique ID for the page content.
- **Duplicate Check:** The Pyodide layer can optionally query the backend or a locally cached index to see if this hash already exists. For example, it might call a lightweight API endpoint to check

the index JSON for the hash, or if the index was previously downloaded, it can check locally. If a duplicate is found, the extension can skip uploading the content again.
- **Packaging Data:** If the content is new, the Python code packages the data for transfer. This could involve structuring a JSON payload containing metadata (URL, timestamp, hash) and the content itself (possibly base64-encoded if binary, though HTML/JS are text). Large data could be compressed or chunked as needed.
- **Client-Side Testing:** One powerful advantage of using Pyodide is that we can run the project's **unit and integration tests directly in the browser**. The same Python logic used to capture and process content can be invoked in a test harness (possibly using frameworks like `unittest` or `pytest` adapted to run in Pyodide). This ensures that the extension's behavior is validated in an environment identical to real usage. For example, a suite of integration tests could simulate loading a test page and verify that the content hashing and network calls work as expected. All of this can run entirely client-side, which speeds up development and continuous integration feedback loops.

By running Python in-browser, we maintain a **"write once, run anywhere"** model for our logic – the code can be reused on the backend if needed and tested in situ. This approach is inspired by patterns in which heavy use of client-side processing and serverless backends leads to simpler, more secure systems (since less server interaction means fewer attack surfaces) [4] [6].

3. FastAPI Serverless Backend

The backend is a **FastAPI** application deployed in a serverless manner (for example, as an AWS Lambda behind API Gateway). Its role is narrowly scoped to receiving content from the extension and storing it in S3. It does not maintain a database or session – each request is handled and forgotten (stateless). The FastAPI endpoint(s) might include:

- An endpoint (e.g. `POST /capture`) that accepts the content payload from the extension's Python code. This endpoint would parse the incoming JSON (which includes the content and metadata). It then saves the content to S3 (see storage below) and updates the index. It might respond with a simple acknowledgment or the storage location.
- A possible endpoint (e.g. `GET /check?hash=<id>`) to allow the extension to query if a given content hash is already stored. This would read the index (from S3 or a cache) and return a yes/no or the existing record. (This is optional – the extension could also just always send content and let the backend deduplicate, but an upfront check can save bandwidth.)

Because it's serverless, this FastAPI service can scale out automatically and we only pay per request. There's no persistent server process running. This design follows Dinis Cruz's recommended approach of using **"serverless functions as an API layer"**, which he has implemented with frameworks like OSBot-FastAPI to simplify deployment [7] [8]. The backend can be easily deployed via CI/CD; for instance, every push to the repo could trigger an AWS Lambda update. Security-wise, keeping the backend stateless and minimal means there's **very little attack surface** or sensitive data to protect on the server [6] – essentially, we delegate data storage security to AWS S3 and minimize custom server logic.

4. AWS S3 Storage and Indexing

All captured content is persistently stored in **AWS S3**, leveraging S3 as a highly durable, scalable datastore. Each capture is saved as one or more objects in an S3 bucket. We use a **timestamp-based key schema** for these objects: each file's key/path is prefixed with the capture timestamp broken down into year, month, day, hour, minute, second. For example, a page captured on 2025/03/26 at 16:45:30 might be stored under:

```
s3://<BucketName>/2025/03/26/16/45/30/<slug>/page.html
```

If there are multiple files (e.g. external JS), they could be stored in a subfolder (`.../45/30/<slug>/script1.js`, `script2.js`, etc.) or bundled. The inclusion of the timestamp in the path ensures **natural versioning** and ordering of content captures [9]. No two captures at the same second will share the exact path, and if the same page is captured at different times (with different content), they'll reside in different timestamped folders. This approach was used successfully in the Cyber Boardroom news feeds system to provide clear versioning through S3 path structure [9]. It effectively treats S3 as an **immutable data ledger** of captured pages – once written, a file is never modified, only new timestamps (versions) are added for updates. This immutability improves security and trust in the data [1] [6].

Alongside storing the raw content files, the backend maintains an **index file** (likely a JSON document) that keeps track of all captured entries. This index could be stored at a fixed location (e.g. `index.json` at the root of the bucket or partitioned by year). Each entry in the index includes metadata such as:

- `url`: The URL of the page that was captured.
- `timestamp`: Capture time (to correlate with the S3 path).
- `hash`: The content hash (for deduplication checks).
- `files`: List of stored file paths (or a standardized naming scheme to derive them from timestamp and perhaps a slug or ID).
- Optionally `title` or other metadata extracted from the page, if useful for search.

When a new capture comes in, the FastAPI backend updates this index – for example, by appending a new JSON object to an array and writing the file back to S3. (In a high-volume scenario, a more sophisticated index like a DynamoDB table might be used, but for the scope of this project a JSON file on S3 is sufficient.) To avoid race conditions with multiple writes, the backend could use optimistic locking or versioning features of S3 (or simply always fetch-update-put the index since writes are infrequent).

The **content hashing** strategy works in concert with the index. Whenever a new page content arrives, its hash is computed (either on the client or recomputed on the server for safety) and compared against entries in the index. If a match is found, the backend can respond that the content is already stored (and possibly return the existing reference). In such cases, it may choose not to store the duplicate again. If no match is found, the content is saved and the index is updated with this new hash. This ensures we never store the exact same content twice. For example, if two news articles are identical or a page hasn't changed between visits, later captures will simply be recognized and skipped. The index provides traceability to the stored content and can also serve as a lookup service for any analysis processes that need to retrieve the captured data later.

Notably, organizing data by timestamp also aligns with the pipelines used in building news archives and knowledge graphs. In Dinis Cruz's MyFeeds.ai project, for instance, content and derived files are stored in S3 with time-based paths, and unique identifiers are generated for each article to track it through different processing stages [10] [11]. Our system takes a similar approach: by storing each page capture in a chronological folder structure, we inherently capture the evolution of that page's content over time (useful for provenance and auditing changes).

Data Flow Sequence

To illustrate how all these pieces work together, below is the typical flow of data when a user visits an authorized website and the extension captures the content:

1. **Page Load Trigger:** A user navigates to an authorized website in Chrome. The extension is listening for this event (either via a content script running at document load or a webRequest in the background). Once the page's HTML begins to load (or after it fully loads), the extension springs into action.

2. **Capture HTML & JS:** The extension collects the page's HTML content. This can be done by grabbing `document.documentElement.outerHTML` in a content script after the page is loaded. In parallel, the extension may capture JavaScript files: using the Chrome devtools protocol or webRequest API, it can listen to responses for `.js` files and record their contents. Alternatively, the content script can fetch the `src` of script tags (if same-domain and permissible) to get the script text. All retrieved content (HTML and JS) is passed into the Pyodide Python environment for processing.

3. **Compute Hash in Browser:** The Pyodide Python code receives the raw content and computes a hash (e.g. SHA-256) of the combined content (or it could compute separate hashes for HTML and each script and then hash those together). This hash represents the page state. The Python code then checks against known content. It might call `GET /check?hash=<hash>` on the backend API, or if the extension recently fetched the index (perhaps cached at startup or periodically), it checks the hash in the local index data. Suppose the hash already exists in the index, meaning this exact content was seen before – the Python code can then decide to skip the upload. (It could still log an event or update a last-seen timestamp, but essentially no new storage is needed.) If the hash is not found (content is new), proceed to the next step.

4. **Send to Backend:** The extension, via the Python code, issues a request to the FastAPI backend to store the content. This is typically an HTTP POST to an endpoint (e.g. `/capture`). The payload includes: the content hash, the page URL, timestamp, and the content itself (HTML and scripts). The content might be sent as raw text or encoded (if binary data were included). Because the extension runs in the user's context, the request must pass CORS checks – the FastAPI service will be configured with the appropriate CORS headers to allow the Chrome extension's origin to post data [12]. The network call from Pyodide uses the browser's fetch under the hood, so it behaves like a normal web request.

5. **Backend Processing:** The FastAPI backend, upon receiving the request, performs a quick validation. It recomputes the hash from the content (for integrity), confirms the user or extension is authorized (the extension could include an API key or use Chrome identity, etc., to ensure only authorized clients send data). Then it creates an S3 key for the content. Using the current timestamp (if none provided) it forms the folder path (year/month/day/hour/minute/second). It saves the HTML file (e.g. as `page.html`). If multiple files are present (scripts), it could save each under the same timestamp folder (with incremented names or using an identifier for the page). Alternatively, the backend might package all content into a single archive file (e.g. a `.zip` or `.mhtml` file) and store that for simplicity. After storing the content, the backend updates the **index JSON**: it adds a new entry with the URL, timestamp, hash, and file pointers. This index file itself is then written back to S3. (If using object tagging or metadata, the backend could also tag the content object with its hash or URL for quick search, though that's an enhancement.)

6. **Acknowledgment:** The FastAPI backend returns a response to the extension. It might include a status ("stored" or "duplicate skipped"), and possibly the S3 path or an ID for the stored content. This response lets the extension know the operation succeeded. If the extension was awaiting this, it could log it or show a badge update to the user (e.g. "1 page archived"). Typically, the user doesn't need to do anything – the process is silent and does not interrupt their browsing.

7. **Future Access & Analysis:** With content safely stored, it can later be accessed for various purposes. The index allows retrieval of a specific page capture by URL or hash. One could build additional tools or APIs to query S3 for a given URL's history or to fetch the latest content snapshot. Because the backend is stateless, direct S3 access can be used for many read scenarios. In fact, the system could expose the stored files via a CloudFront distribution or presigned URLs, enabling other services (like an analysis pipeline or an AI model) to load the content directly from S3 with high speed [4]. This two-interface model – raw data via S3 and value-added queries via FastAPI – follows the pattern from the Cyber Boardroom news feed architecture [1], providing both **performance and flexibility**. The raw content is available in S3 for any integration (and S3's scalability ensures it can handle large volumes), while the FastAPI layer can be extended to provide filtered views, search capabilities, or to initiate further processing (like triggering an NLP analysis on new content).

Throughout this flow, **deduplication** is enforced by the content hash checks, and all heavy content processing is done in the user's browser. The serverless backend remains lightweight, doing minimal CPU work (mostly just moving data to S3). This not only makes the system efficient but also more secure – by **"having less to attack"** on the server side, as Dinis's design philosophy notes [13]. Any complex logic (parsing HTML, running JavaScript emulations, etc.) can be shifted to the client or to specialized offline processors, keeping the capture pipeline simple and robust.

Similar Work and Inspiration

This project's architecture is informed by prior research and projects in content provenance, news feeds, and serverless design. In Dinis Cruz's work on **personalized news feeds and semantic graphs**, he emphasizes an **API-first architecture** where raw content is collected and made accessible in a structured way, separate from the layers that do verification or analysis [3]. Our approach of capturing HTML/JS and storing it immutably aligns with that philosophy – we create a foundational content repository upon which trust and graph-based services can be built (for example, mapping relationships between articles, tracing fact sources, etc.).

The **Cyber Boardroom News Feed** system (first MVP) is a close analog that inspired parts of this design. In that system, RSS feed data from sources like *The Hacker News* is converted into JSON and stored in S3, serving as a feed database [1]. One of the key insights was exposing this data via dual interfaces: a static S3 (file-based) API for fast, direct access, and a dynamic FastAPI for queries and operations [4]. We take a similar approach: our captured content is stored in S3 (which could be made directly accessible as a read-only feed of archived pages), while our FastAPI backend can handle queries (like listing content or searching the index) in addition to ingesting new data. By using S3 as the primary store, we inherit benefits like scalability, low latency (especially if paired with a CDN), and inherent versioning of data by path [9]. This "serverless data pipeline" approach has proven effective in reducing complexity and improving security in related projects [6].

Another influence is the **MyFeeds.ai** architecture for semantic news feeds. In MyFeeds, a series of flows (implemented in FastAPI on AWS Lambda) take content through multiple processing steps, and content files (articles, timelines, graphs) are stored on S3 with timestamped paths [10]. Our project is a smaller-

scale but analogous pipeline: we focus on the initial step of getting content into a store. Down the line, additional **flows** or processes could be added to transform the captured HTML into JSON summaries, populate a knowledge graph, or run verification checks – all using the stored data as a source of truth. The use of Pyodide for in-browser logic is also in line with modern web development trends, and showcases how much can be done on the client side. Similar ideas are seen in projects like *Datasette Lite*, which runs a full Python data tool in the browser via Pyodide [5]. This reinforces the viability of our choice to run Python in-browser for capturing web content.

Future Steps and Modularization

Moving forward, the project will focus on setting up a clean, modular codebase and expanding capabilities. Key next steps include:

- **Repository Structure:** Organize the code into clear modules. For example: a `chrome-extension/` directory for the extension (with subfolders for the content script, background script, and manifest), a `pyodide/` directory for the Python code intended to run in-browser (possibly with a build script to package it for the extension), and a `backend/` directory for the FastAPI code (including its requirements, deployment scripts/Infrastructure-as-Code for AWS). This separation will make it easier for contributors to navigate the project.

- **Modular Code for Reuse:** Ensure that logic is not duplicated between components. For instance, the content hashing and any content parsing functions should be written in Python (so they can run in Pyodide and be reused in the backend if needed). We might factor these into a small Python package (e.g. a `content_capture` module) that can be invoked in Pyodide and in a normal Python environment. Similarly, if any JavaScript helper code is needed for the extension (like to intercept network requests), keep it focused and consider exposing a clear interface to the Python code (perhaps via `pyodide.runPython` calls).

- **Continuous Integration and Testing:** Set up CI pipelines that run automated tests. This could include headless browser tests for the extension – for example, using a tool like Puppeteer or Playwright to simulate Chrome, load a test page, and verify that our extension captures and sends data correctly. We will also leverage Pyodide in a CI setting: it's possible to run Pyodide in Node or in a headless manner to execute the Python unit tests. By automating these tests, we can catch issues early. Every pull request can trigger tests for the Python logic (in a Pyodide environment), tests for the FastAPI endpoints (using pytest on the backend code), and maybe linting for the JS extension code.

- **Deployment Automation:** Configure scripts for packaging and deploying the extension and backend. For the extension, we can use `web-ext` or similar to build the extension and even run it in a headless Chrome for testing. For the backend, use Infrastructure as Code (CloudFormation, Terraform, or AWS CDK) or the OSBot-FastAPI deployment mechanism to update the Lambda. Automating deployment means that when changes are merged, a new version of the backend is deployed (possibly first to a staging environment). This ensures quick iteration and testing in a live environment.

- **Extensibility and Configurability:** Future improvements will make the system more flexible. We plan to add a configuration for which sites or pages to capture (perhaps a UI in the extension popup to toggle which domains are active). We also aim to modularize the capture logic so that new types of content can be added – for example, capturing an API response or an image if

needed. The Python code could be extended with plugins for different content types (HTML, PDF, multimedia). By structuring the code well now, adding such features later will be easier.

- **Integration with Analysis Pipelines:** Although initially the project's scope is just capture and store, we anticipate downstream use of the data. The repository might evolve to include a `analysis/` module or a separate project that reads from the S3 bucket and performs tasks like building a search index, feeding an LLM for summary, or linking content to a graph database. Laying the groundwork now (with the index and clear data format) will facilitate these future projects. For example, one future step is to create a small service that reads new entries from S3 (could be triggered by an S3 event) and runs an **LLM-powered analysis** to extract entities or facts, contributing to a knowledge graph. This ties back into the vision of leveraging **personalised semantic graphs and provenance** in news content [3]. Our captured data will be a foundational layer for such high-level services.

- **Documentation and Examples:** Ensure the GitHub project has thorough documentation. We will include a detailed README with architecture (including the diagram above), setup instructions for developers (how to load the extension in Chrome for development, how to deploy the backend, etc.), and examples of usage. We can provide sample scripts or notebooks that demonstrate how to fetch a stored page from S3 given a URL, or how to extend the Python logic. Given the technical audience, we'll also document the rationales behind using Pyodide, content hashing, and S3 – citing the security and performance benefits evident from prior projects (e.g., immutable storage and low attack surface) [6].

By breaking the project into modular components and automating testing/deployment, we set the stage for a maintainable open-source project. We will continue to iterate on the system, guided by the twin principles shown in similar successful systems: **simplicity** in architecture (favoring static storage and serverless functions over complex always-on servers) and **integrity** of data (capturing content in full fidelity, avoiding alterations, and tracking provenance through content hashes and versioned storage) [13] [2].

Ultimately, this Chrome extension + backend will provide a valuable tool for capturing web content in a trustworthy way – enabling applications in archival, fact-checking, and personalised news delivery that builds on a solid foundation of recorded truth.

Sources: The design draws on concepts and lessons from Dinis Cruz's research and projects, including content API architectures for news providers [3], serverless feed generation pipelines [1], and the MyFeeds.ai semantic news feed implementation [10], as well as broader use of Pyodide for running Python in-browser [5]. All code and documentation will be made available on GitHub for collaboration.

---

[1] [4] [6] [13] Creating a Secure GenAI News Feed: When the Best Defence is Not Having Anything to Attack (with Stride and OWASP Top 10)
https://www.linkedin.com/pulse/creating-secure-genai-news-feed-when-best-defence-having-dinis-cruz-5iqre?trk=article-ssr-frontend-pulse_x-social-details_comments-action_comment-text

[2] [9] Building two API News Feed Architecture with S3, FastAPI, and GenAI
https://www.linkedin.com/pulse/building-two-api-news-feed-architecture-s3-fastapi-genai-dinis-cruz-kimde

[3] Monetising Trust and Knowledge: How News Providers can leverage Personalised Semantic Graphs - Dinis Cruz - Documents and Research
https://docs.diniscruz.ai/2025/02/02/monetising-trust-and-knowledge-for-news-providers.html

5 URL-addressable Pyodide Python environments
https://simonw.substack.com/p/url-addressable-pyodide-python-environments

7 8 How I'm Building Personalised News Feeds with Semantic Graphs - Part 1
https://mvp.myfeeds.ai/publishing-a-new-personalised-set-of-posts-part-1/

10 11 How I'm Building Personalised News Feeds with Semantic Graphs - Part 2
https://mvp.myfeeds.ai/how-im-building-personalised-news-feeds-with-semantic-graphs-part-2/

12 How to configure CORS to allow a Chrome Extension in FastAPI?
https://stackoverflow.com/questions/76766436/how-to-configure-cors-to-allow-a-chrome-extension-in-fastapi