ChatGPT

Technical Debrief: Evolution from Electron-Based to Python-Native Web Content Capture App

Executive Summary

This document provides a technical debrief on the evolution of a web content capture application from an Electron-based architecture to a new Python-native design. The original solution was a cross-platform desktop app built with **Electron + Node.js + Playwright + Python**, intended to embed a Chromium browser and capture all web content (HTML, JavaScript, CSS, XHR, etc.) in real time [1]. While this approach met some initial goals (full page capture with a familiar Chrome-based UI [2]), it proved **ineffective due to complexity and operational limitations**. The Electron strategy required multiple runtimes (JavaScript and Python) coordinated via inter-process communication, resulting in high overhead and brittle integration. In practice, the multi-layered design introduced performance issues and complicated packaging, outweighing its benefits.

The team has since pivoted to a **unified Python architecture** leveraging **PyQt6 (QtWebEngine)** for the GUI and embedded browser, and **mitmproxy** for network interception, with FastAPI still used for backend storage. This new design simplifies the stack to a single language (Python) and process, offering deeper native desktop integration and more robust content capture. Key advantages include improved maintainability, lower resource consumption, and streamlined deployment. Mitmproxy allows capturing all HTTP/S traffic externally, avoiding the pitfalls of in-browser scripting or extension limitations. A comparison of the old vs. new architectures is provided, highlighting how the Python-native approach mitigates the original strategy's shortcomings. Finally, we outline next steps and a roadmap for implementing the new architecture, ensuring a clear path forward for development and deployment.

Original Electron-Based Architecture and Its Limitations

Overview of the Electron + Playwright Approach

The original application was implemented as an Electron desktop client with an embedded Chromium browser and a headless Python engine (Playwright) running in parallel [2]. The high-level design separated the system into three primary components:

- **Electron Desktop Application (Chromium UI):** The desktop app was built with Electron, bundling a Chromium engine and Node.js runtime. This provided a familiar browser UI that users could interact with normally, identical to a Chrome experience [3]. Users could navigate pages as in a standard browser, while behind the scenes the app would capture everything loaded in each page (DOM, scripts, resources). The cross-platform nature of Electron meant the same codebase could target macOS, Windows, and Linux, and Electron's packaging tools support native menus, auto-updates, and installers [4].

- **Python Capture Engine (Playwright):** A Python process running Playwright would attach to the Electron's Chromium instance to monitor and capture all network activity and page content. Playwright (in headless mode) acted as the automation and interception layer, retrieving each

page's HTML, scripts, AJAX responses, and other resources for archiving [5] [2]. The Electron app's main process would spawn or connect to this Python service on startup [6]. Communication between Node (Electron) and Python was handled via IPC mechanisms – for example, using Electron's `ipcMain`/`ipcRenderer` messages or a local WebSocket to pass events and data [6]. The Node side could forward user actions or settings (like toggling capture modes, allowed domains, etc.) to the Python side, which performed the actual capture logic. Within Electron, it was also possible to intercept web requests using the Chromium DevTools protocol or `session.webRequest` APIs, but the design primarily relied on Playwright's Python logic for network interception [7].

- **FastAPI Backend Service:** Captured content was sent via HTTP from the client to a lightweight FastAPI backend (either running in the cloud or locally) for storage [2]. The backend's sole responsibility was to receive the captured data and store it (e.g. in AWS S3) and update an index for deduplication [8] [9]. This was architected as a stateless, serverless-friendly component to simplify scaling and security. In **Production Mode**, the app would POST captures to a remote FastAPI service and S3 cloud storage, whereas in **Development/Offline Mode**, it could use a local FastAPI server and local S3 emulator (via LocalStack) so that everything runs on the developer's machine [10]. This dual-mode setup enabled testing and iteration without cloud dependencies while using the same codebase for live deployment and offline use.

Diagram – Original Architecture: In the Electron-based design, the **Electron app** (Node/Chromium) handled the user interface and navigation, while a **Python Playwright controller** ran in parallel to intercept network traffic and page content. The Python process captured all assets and then **posted the data to a FastAPI endpoint** for storage in S3. The diagram below illustrates the major components and data flow:

```
[User Browser UI (Electron: Chromium)]  <--user navigates-->
  ↳ [Playwright Python Engine] (attaches to Electron's Chromium, intercepts
content)
    ↳ Captured content (HTML, JS, CSS, XHR, etc.)
              → HTTP POST →  [FastAPI Backend] → S3 Storage (or Local disk)
```

Figure: Electron-based architecture with a Node.js front-end and Python automation back-end. The Electron process provides the Chromium browser UI, while the Python Playwright process captures network traffic and page resources, sending them to a FastAPI service for persistence. [2] [6]

Why the Original Strategy Was Ineffective

Despite achieving a functional prototype, the Electron-based approach revealed several technical and operational limitations that made it unsustainable. The key issues were:

- **Excessive Complexity (Multi-Runtime Overhead):** The architecture introduced significant complexity by coupling two runtimes (Node.js and Python) and an automation layer. Electron's Node process had to **orchestrate with a separate Python process** for core functionality [6]. This IPC bridging added many moving parts (Electron app, Playwright controller, IPC channel, etc.), making the system brittle and harder to debug. As one observer noted in a related context, each attempted solution that adds such complexity risks making the project more brittle [11]. The need to synchronize state between the browser UI and the Python capture engine (e.g. navigation events, data handoff) increased the chance of race conditions and errors.

- **Performance and Memory Footprint:** Electron applications are known to be heavy on resources – even simple Electron apps can consume hundreds of MB of memory due to the Chromium engine and embedded Node runtime [12]. In our case, we effectively had **two heavyweight processes** (Electron/Chromium and Python/Playwright) running concurrently. This led to high memory usage and CPU overhead for what should be a lightweight capture tool. The community often cautions that many Electron apps suffer performance issues [13], and indeed our multi-process design exacerbated resource consumption. The Python Playwright process itself launches or attaches to a browser (the Electron's Chromium) which is less efficient than a single integrated solution. Overall, the app was taxing in terms of RAM and could become sluggish, especially on lower-end hardware.

- **Operational and Packaging Complexity:** Building and distributing the Electron-based app proved complicated. We had to package a Node.js application *and* bundle a Python environment (with Playwright and dependencies). This is not a standard use-case for Electron packaging. While Electron has tools for creating installers, incorporating a Python runtime required extra steps (either embedding Python binaries or running Python scripts externally). The result was a large application bundle and a complex build pipeline. In contrast, a pure Python application can be packaged with tools like PyInstaller or Nuitka more straightforwardly. The original approach's distribution pipeline (Node + Python) was harder to automate and test. Auto-update mechanisms also become tricky – Electron can self-update the JS app, but ensuring the bundled Python components update in lockstep adds complexity.

- **Reliance on DevTools Integration:** The capture of network traffic in the Electron approach primarily relied on **Playwright's automation** hooking into Chromium's devtools protocol [7]. While Playwright is powerful, using it in a live user-driven context introduced some uncertainty. The Playwright engine had to stay in sync with user actions, and any failures in the automation layer could cause missed captures. There were also edge cases (like downloading large files or intercepting WebSocket streams) that would require additional handling in Playwright. In effect, we were using a test automation tool in a production capture scenario, which is not its typical use. This could lead to maintenance issues with upstream updates or unforeseen limitations in interception (for example, Playwright might not easily capture certain low-level request details or binary responses without extra code).

- **High Memory & Bundle Size:** Because Electron includes an entire Chromium browser and Node, and Playwright brings its own overhead, the application was very large in size. The user had to download a hefty installer (including Chromium, Node, Python, Playwright, etc.). Running the app consumed a lot of memory – anecdotal comparisons show an Electron app can use 3× more memory than a Qt-based equivalent for a similar "Hello World" functionality [14]. This footprint was a concern for deploying the tool in environments like journalist desktops or enterprise laptops, where a leaner solution is preferable.

- **Maintenance and Skill Set Mismatch:** The hybrid tech stack required expertise in web front-end/Electron development *and* Python development. This split brain made development slower and debugging more complex. Every feature involved touching code in two ecosystems (JavaScript/Node and Python) and ensuring they interoperate. For a small team, this overhead was significant. In addition, debugging issues through the Electron <-> Python boundary (such as why a certain network event didn't get captured or a message didn't arrive via IPC) was non-trivial. A single-language codebase would be far easier to maintain and extend.

In summary, the original Electron-based strategy, while conceptually sound for capturing rich web content, **introduced too much complexity and overhead relative to its benefits**. The design broke a

fundamental principle: new capabilities should simplify the system, not make it more complicated [11]. The multi-process architecture made the app heavy and prone to brittleness. These limitations motivated a re-architecture to achieve the same goals with a more streamlined, Python-centric solution.

New Python-Native Architecture (PyQt6 + QtWebEngine + mitmproxy + FastAPI)

To address the shortcomings of the Electron approach, we redesigned the application with an all-Python stack. The new architecture uses **PyQt6** for the desktop application (leveraging QtWebEngine for an embedded Chromium browser component) and **mitmproxy** for intercepting web traffic. The FastAPI backend and storage logic remain, but now everything except the browser engine and FastAPI server is consolidated in one Python process. This design preserves the core functionality (capturing all web content a user sees) while eliminating the overhead of Node.js and simplifying the data capture mechanism.

Architecture Overview: The Python-native app consists of a single desktop application process with the following integrated components: - **PyQt6 Application (QtWebEngineView Browser):** The UI is built using PyQt6, which provides a native window and an embedded Chromium browser via QtWebEngine. The user interacts with this browser just like a normal web browser – typing URLs, clicking links, logging in, etc. QtWebEngine renders pages with full fidelity (since it's essentially Chromium under the hood). Because this is a Qt app, we can also incorporate native GUI elements (menus, dialogs, system tray integration) directly via Qt's APIs for a more seamless desktop experience. The QtWebEngineView can be configured to use an HTTP proxy for all network requests, which is central to our capture strategy.

- **Mitmproxy Intercepting Proxy:** Instead of relying on in-browser scripts or devtools protocols to capture content, the new design routes all browser traffic through **mitmproxy**. Mitmproxy is a powerful Python-based "man-in-the-middle" proxy that can intercept and record HTTP/HTTPS requests and responses [15] [16]. By running a mitmproxy instance locally and setting it as the proxy for the QWebEngine, **every network request from the embedded browser (and the corresponding response)** will pass through mitmproxy. This gives us complete visibility into all web content being loaded – HTML, JSON API calls, images, stylesheets, scripts, etc. – without needing to instrument the browser from inside. Mitmproxy effectively decouples the capture from the UI: the browser is unaware of the interception (it sees a proxy as if it were the internet), and our application can listen in on all traffic externally. We configure mitmproxy with our own script or addon to extract the desired data. For example, we can write an addon that triggers on each response and saves the content or sends it to the backend. Mitmproxy's scripting ability means we can filter or transform data if needed during capture, using pure Python code. This approach is robust and future-proof – it doesn't rely on Chrome extension APIs or devtools hacks, and it works for any content loaded in the browser [15].

- **FastAPI Backend (Cloud or Local):** We continue to use a FastAPI-based web service to handle storage of captured content, similar to the original design. The difference is that now the FastAPI service could be invoked either by the mitmproxy script or by the Qt application directly. There are two possible data flows: in **online mode**, mitmproxy can act as a pass-through that simply logs traffic and the Qt app can send captured data via HTTP POST to a remote FastAPI endpoint (e.g., with the same format as before). In a more integrated approach, we might incorporate a small FastAPI server in the application itself (running on localhost) and have mitmproxy forward or push data to it in real time. For the scope of this project, we assume the same cloud backend exists – mitmproxy will dump the intercepted content and our code will package it (e.g., as JSON

with metadata and content hash) and send to the backend API. In offline mode, the app can bundle a local FastAPI and local storage (or simply write to disk) for development testing, just as before [10]. Essentially, the backend component remains stateless and simple, and the capture app is responsible for preparing and transmitting the data.

Diagram – New Python-Native Architecture: The re-architected system collapses into a single application process with an external proxy helper:

```
[User Desktop App – PyQt6]
   └─ QWebEngineView (Chromium) — (All HTTP/S requests) → ■ mitmproxy ■ →
(All responses) —> [FastAPI backend/API] -> Storage (S3)
```

Figure: Python-native architecture. The PyQt6 application contains an embedded browser (QtWebEngine). All network traffic from this browser is funneled through a local mitmproxy instance, which intercepts and logs content. Captured data is then forwarded to a FastAPI service for storage in the cloud or local filesystem. [15] [16]

Key Interactions and Components in the New Design:

- **Unified Python Process:** The entire GUI and capture logic run in one process (or a set of Python threads). This means no more context-switching between Node and Python. The application starts up by launching the Qt application and initializing a mitmproxy proxy (which can be started programmatically as a background task). Because both the browser UI and the proxy are Python-controlled, synchronization is easier. For instance, the app can programmatically instruct the QWebEngine to trust the mitmproxy's certificate authority (to avoid HTTPS warnings), and can start/stop the proxy as needed. Communication between the browser component and our logic is direct via Qt signals/slots or simply by reading the proxy's output, rather than via IPC over a socket. The unified codebase improves stability and simplifies development (one language, one runtime).

- **Network Traffic Capture via Proxy:** By using mitmproxy, we gain a **complete capture of all HTTP(S) traffic** without needing to modify the web page or rely on browser internals. Mitmproxy operates at the network level, so even dynamic requests like XHRs, fetch API calls, and redirects are captured uniformly. This method was chosen after exploring alternatives and realizing that an external proxy is the most reliable way to grab all traffic [16]. Mitmproxy is scriptable, so we can write Python functions to filter or process flows. For example, we can configure it to intercept response bodies and immediately save them to a file or database. This out-of-band capture is more robust than the previous Playwright approach – even if a page uses complex web sockets or loads resources from multiple domains, the proxy sees it all. Another benefit is that our capture logic becomes browser-agnostic: although we use QtWebEngine (Chromium) now, we could switch to another browser or engine in future (even an external browser pointed at our proxy) and still capture content, since the proxy approach is generic.

- **Native Desktop Integration:** With PyQt6, the app can utilize native GUI features and better OS integration than was possible with the browser-extension-like approach. We can create custom menus, system tray icons, native file dialogs, and other widgets easily via Qt. This allows enhancements like a **capture control panel** (to start/stop capturing, view status, etc.) within the application window, or native notifications when a capture is saved. While Electron also supported menus and notifications, doing so in PyQt6 ties into the OS more directly (using Qt's

cross-platform abstractions that map to native controls). The result is a more polished user experience that feels like a true desktop application of the host OS. Additionally, Qt's event loop and signal/slot mechanism provide a robust way to handle asynchronous events (like incoming data from mitmproxy or user navigation events) in a thread-safe manner within one process.

- **Simplified Packaging & Deployment:** The new architecture will be packaged as a **single Python application**. Using tools such as PyInstaller or Nuitka, we can bundle the Python interpreter, our PyQt6 app, and any required libraries (including mitmproxy and its dependencies) into one executable for each target platform. This greatly simplifies the distribution pipeline compared to the Electron solution which required combining Node and Python artifacts. The app's size is expected to be smaller as well: Qt WebEngine will add some weight, but we avoid shipping a full Node.js runtime and Electron's overhead. Memory usage at runtime should be improved as well since we run one Chromium instance (QtWebEngine) instead of two (Electron's Chromium + potentially a separate headless browser in Playwright) and eliminate the Node.js process. Initial research suggests a Qt WebEngine app can be significantly lighter on memory than an equivalent Electron app [12]. Auto-update can be implemented by the app itself (for example, by checking a version endpoint and downloading a new installer) or by leveraging existing Python frameworks for self-update. With a unified codebase, maintaining updates is easier — only one set of dependencies and code to manage. Continuous integration can build the app for all platforms in one workflow using Python tools.

- **Continued Use of FastAPI Backend:** The cloud storage workflow remains largely the same. After mitmproxy intercepts the content, our code will likely assemble the captured data (HTML plus resources or their hashes) and send it to the FastAPI endpoint over HTTP. FastAPI will continue to handle writing to S3 and updating an index (e.g., a JSON index of captured pages, as described in the original plan) [17] [9]. Because FastAPI is also Python, we maintain a homogenous technology stack on both client and server side. We might also consider running the FastAPI component as part of the application in offline mode for an even more self-contained solution. Regardless, the contract between the capture app and backend (an HTTP API for uploads) does not change with the new architecture.

Annotated Code Snippet (PyQt6 + mitmproxy Integration): Below is a simplified example showing how the Python-native app could initialize the browser and proxy:

```
import sys, subprocess
from PyQt6.QtWidgets import QApplication
from PyQt6.QtWebEngineWidgets import QWebEngineView
from PyQt6.QtCore import QUrl
from PyQt6.QtNetwork import QNetworkProxy

# 1. Start mitmproxy (mitmdump) on a local port (e.g., 8080) to intercept
traffic
mitm_process = subprocess.Popen(["mitmdump", "-p", "8080", "-w",
"capture.log"])
# The -w option writes all flows to capture.log for later analysis or
forwarding.

# 2. Initialize the Qt application and browser view
app = QApplication(sys.argv)
browser = QWebEngineView()
```

```
# 3. Configure the global proxy for Qt WebEngine to use mitmproxy
proxy = QNetworkProxy(QNetworkProxy.ProxyType.HttpProxy, "127.0.0.1", 8080)
QNetworkProxy.setApplicationProxy(proxy)
# Now all network requests from QWebEngineView will go through the mitmproxy
at 127.0.0.1:8080.

# 4. Load a web page (for demonstration)
browser.load(QUrl("https://example.com"))
browser.show()

# (The mitmproxy will capture the HTTP GET to example.com and the response
HTML/CSS/JS)
# Additional logic would go here to handle captured data, e.g., reading
capture.log
# or using mitmproxy's Python API to register event hooks that send data to
FastAPI.

# 5. Run the application event loop
sys.exit(app.exec())
```

Code explanation: In this snippet, we launch **mitmdump** (a headless version of mitmproxy) as a background process on port 8080. Then we create a PyQt6 application with a `QWebEngineView`. We set a global network proxy for the application to point to our mitmproxy (`127.0.0.1:8080`). When the browser widget loads a page, all its requests will be funneled through the proxy. Mitmproxy, running in the background, intercepts those requests/responses and logs them (here, to a file `capture.log`). In a full implementation, we would replace or augment the `-w capture.log` with a custom mitmproxy addon script that could, for example, POST each response to the FastAPI server as it comes in, or save them into a structured archive. We also need to handle the mitmproxy certificate: QtWebEngine will block HTTPS traffic by default if the proxy's certificate is untrusted, so the app would install the mitmproxy CA certificate into Qt's certificate store at runtime (mitmproxy provides an easy way to get its cert, and Qt's network settings allow installing a CA). This ensures even TLS traffic is seamlessly captured. After that, the user can browse normally in the `QWebEngineView`, and every piece of content loaded will be captured by mitmproxy in the background.

Overall, this code demonstrates how straightforward the integration becomes: we leverage Qt's built-in ability to use a proxy and mitmproxy's power to intercept traffic. There is no need for an Electron container or a separate automation layer to pull data from the browser – the proxy does all the heavy lifting of capture, and our application remains responsive and focused on the user interface.

Comparison of Old vs. New Architectures

The following table summarizes the key differences between the original Electron-based design and the new Python-native design, highlighting how the new approach improves upon the old:

Aspect	Electron+Node+Playwright (Original)	PyQt6+mitmproxy (New)
Tech Stack	Heterogeneous: *JavaScript/Node.js* for UI + *Python* for logic [6]. Requires expertise in two platforms.	Homogeneous: *Python-only* (PyQt6 for UI, Python for logic). Single language and runtime.

Aspect	Electron+Node+Playwright (Original)	PyQt6+mitmproxy (New)
Browser Engine	Embedded **Chromium** via Electron (Blink engine). UI built with HTML/JS/CSS.	Embedded **Chromium** via QtWebEngine (Blink engine). UI via Qt Widgets/QML (native windows).
Network Interception	Via **Playwright** hooking DevTools protocol and/or Electron's `webRequest` API [7]. Needed custom code to capture responses.	Via **mitmproxy** acting as an external proxy capturing all HTTP/S traffic [15]. No browser-internal hacks; full coverage of requests/responses.
Process Model	Multi-process: Electron (Node.js + Chromium) **and** separate Python process. Requires IPC coordination [6].	Single-process (for app and capture). mitmproxy runs as a thread or sub-process but is managed by the Python app. Minimal inter-process communication (all Python).
Performance Footprint	Heavy: Large memory usage (Chromium + Node + Python). High CPU overhead for two runtimes. Known Electron performance issues [13].	Lighter: One primary GUI process (Chromium via Qt) + proxy. No Node.js overhead. Lower memory and CPU due to unified architecture [12].
Desktop Integration	Good web UI flexibility, but relying on web tech for UI means less native feel (though Electron provides some native API access).	Strong native integration via Qt: native window, OS dialogs, menus, system tray, etc. Feels more like a native app on each platform.
Development Complexity	High: Two codebases (JS and Python) to develop and sync. Complex debugging across the Node/Python boundary.	Lower: Single codebase in Python. Simpler debugging and faster iteration. All components (UI, capture logic) in one environment.
Packaging & Deployment	Complicated: Need to bundle Node.js, Chromium, and Python environment. Large installer size (hundreds of MB). Electron's auto-update must also account for Python code updates.	Simplified: Bundle Python app with PyQt and dependencies. Installer size potentially smaller (no Node). Standard Python bundlers can produce executables. Updates are unified (one codebase to update).
Extensibility	Adding features requires touching frontend (Electron JS) and backend (Python). E.g., new capture rule means updating Python; new UI control means Electron changes. Chrome extension updates could affect Electron if used.	Adding features is straightforward in Python/PyQt. E.g., to support a new resource type, update the mitmproxy script or Qt logic. No context switching between languages. Qt's signals/slots make it easy to handle new events or UI components.

Aspect	Electron+Node+Playwright (Original)	PyQt6+mitmproxy (New)
Content Capture Reliability	Relied on Playwright; possible edge cases if Playwright didn't capture something or if user interactions outpaced the automation. Some network events might be missed without explicit handling.	Very robust: mitmproxy captures **all** traffic at the network level, independent of page scripting. Less prone to missing data. Even edge cases like redirects, binary streams, and WebSocket messages can be handled by mitmproxy's framework.
Future Flexibility	Tied to Chromium via Electron; using any other browser engine would be difficult. Also required Chrome-specific code for extension context previously.	More flexible: While currently using QtWebEngine (Chromium), we could point any browser to the proxy (even external ones) to capture from there. The capture logic is decoupled from the UI browser, enabling potential support for multiple browsers or headless operation in future.

Table: Comparison of the Electron-based architecture vs. the new PyQt6 (QtWebEngine) architecture. The Python-native approach offers a single-language solution with lower overhead and improved simplicity in most areas. Notably, the use of mitmproxy for content interception replaces the need for an embedded automation layer and improves the fidelity of captures.

Advantages of the Unified Python Codebase

Adopting a Python-native architecture yields several clear benefits:

- **Single Language & Runtime:** All application logic is written in Python, which means the development team can focus on one technology stack. This reduces context switching and errors. It also means we can leverage Python's rich ecosystem uniformly across the app – for example, using the same logging, config, and testing frameworks end-to-end. The learning curve is lower for new developers (they only need to know Python/Qt, not Node/Electron) and bugs are easier to trace without crossing a language boundary.

- **Native Desktop Experience:** Using PyQt6 allows the app to behave and feel like a native application on each OS. We can use native file pickers, integrate with OS-specific features (dock icon, taskbar, notifications) directly, and comply with UI guidelines more easily than in a browser-based shell. This improves user trust and comfort, which is important if the tool is used in enterprise or professional workflows. It also avoids some pitfalls of Electron apps, such as high idle resource usage and slow startup times; Qt applications can be optimized and don't need to load an entire browser runtime for simple UI tasks.

- **Improved Content Interception:** Mitmproxy provides a **more powerful and transparent interception** than the previous in-app methods. We no longer depend on what the browser exposes (or allows) – instead we capture raw HTTP traffic. This means even if the web page uses advanced techniques (like dynamically generating requests via JavaScript or using service workers), the proxy sees it. We can also record exact bytes of responses for cryptographic hashing or replay, something that was more complicated under Playwright. As the mitmproxy docs highlight, it can even intercept and modify on the fly and save complete HTTP

conversations for later analysis [18]. We leverage these capabilities to ensure a **faithful archive of web pages** as they were delivered, which is crucial for content verification and provenance.

- **Simplified Build and Deployment Pipeline:** The move to a single executable simplifies CI/CD. We can have one build process to produce the application for all platforms using PyInstaller or a similar tool. The absence of Node modules (which can number in the hundreds in an Electron app) reduces the attack surface and avoids supply-chain concerns in deployment. There are fewer external dependencies to manage (e.g., no need to keep Chromium DevTools protocol in sync with Playwright versions, etc.). We also avoid the Chrome extension packaging/signing requirements entirely – everything is self-contained in our app, which we control fully.

- **Scalability and Maintainability:** The Python codebase can be more easily extended to incorporate new features. For instance, if we want to add on-the-fly analysis of captured content (say, running a machine learning model on the text), we can do so directly in the Python app where the data is present, rather than marshaling data out of a renderer process. The use of Qt signals and slots makes it simple to ensure the UI remains responsive (e.g., the capture can run in a background thread while the UI shows progress). The design is also more **testable** – we can write unit tests for the Python functions (including mitmproxy addons) and use Qt's QTest frameworks for GUI testing. With Electron, end-to-end testing relied on Playwright driving the whole app, which is heavier compared to testing Python functions in isolation.

In essence, the new architecture aligns with the goal of making the system **simpler, more robust, and easier to evolve**, directly addressing the pain points discovered in the Electron approach.

Next Steps and Implementation Roadmap

With the decision to proceed with the PyQt6/mitmproxy architecture, the following steps outline the implementation plan and timeline:

1. **Project Setup and Dependencies:** Begin by setting up the Python project structure. Ensure that PyQt6 and mitmproxy (as a Python library) are added to the project's environment. Mitmproxy provides a pip-installable package and can also be used via its **python API** or by calling the `mitmdump` binary. We will experiment with both approaches to see which integrates better (initially, using `mitmdump` as shown in the snippet is simplest). Verify that a basic PyQt6 window with a QWebEngineView can be launched.

2. **Browser Proxy Integration:** Implement the logic to launch mitmproxy in the background when the app starts. This includes generating or retrieving the mitmproxy CA certificate. Configure QWebEngineProfile (the profile for QWebEngineView) to trust this certificate and route traffic through the proxy. This step will involve using Qt's networking classes to set the application proxy and possibly using QtWebEngine's settings to accept a custom certificate for HTTPS. We will test this by loading both HTTP and HTTPS sites and confirming that mitmproxy is logging the traffic (e.g., by examining `capture.log` or using mitmproxy's interactive interface in development).

3. **Basic Capture and Storage Flow:** Develop a minimal mitmproxy addon script that captures essential data. For example, write a function that listens for the HTTP response event in mitmproxy, and within it, package the URL, response headers, and body into a Python object or JSON. At first, we can simply log this to a file to validate it works. Then integrate the FastAPI client: have the addon (or the main app) perform an HTTP POST to the FastAPI backend (which

could be a test server or a dummy endpoint at first). This will exercise the end-to-end path: user loads a page -> mitmproxy intercepts -> our code sends data to backend. We will likely iterate here to ensure we don't send redundant data (e.g., filter out advertising or irrelevant third-party requests if needed) and that we compute content hashes as planned for deduplication.

4. **User Interface & Controls:** Build out the PyQt UI around the browser component. At minimum, implement an address bar for URL entry, basic navigation buttons (back, forward, refresh), and a status bar to show capture status. Because QWebEngineView behaves much like a normal browser, we can connect to its signals (like `loadStarted`, `loadFinished`) to update the UI or trigger certain actions. For instance, we might only start capturing when a page is fully loaded to get all sub-resources. Additionally, provide UI feedback such as "Capturing content..." indicator, and maybe a menu or preferences dialog to configure settings (e.g., toggle capturing on/off, set which domains to capture or ignore, etc.). This step focuses on ensuring the app is usable as a browser replacement for the user.

5. **Enhance Mitmproxy Addon Logic:** As the UI stabilizes, refine the mitmproxy capture logic. We can add features like content filtering (only store content from certain domains or content types), redacting sensitive information (if required), and handling streaming content or large files gracefully (perhaps by not capturing videos or limiting by size). If performance becomes an issue with many requests, we'll consider using mitmproxy's filtering to only intercept relevant requests (for example, if the use-case only cares about HTML and JSON, we might skip images or other media). Ensure that the addon properly catches any exceptions (we don't want the proxy to crash and leave the browser without connectivity).

6. **Local vs Cloud Mode Implementation:** Implement the dual-mode operation. This involves having a configuration for the FastAPI endpoint – in production mode, use the remote API (with authentication if needed, such as an API key or token the app supplies). In offline mode, either start a local FastAPI server (packaged with the app) or simply write captures to a local directory. Using a local FastAPI can simulate the network upload and allow the same code path with just a different URL. Alternatively, to minimize running two servers, the app could directly save files locally when in offline mode. We will also integrate LocalStack (or a dummy S3) if we want to fully emulate cloud storage locally, as the original plan suggested [10]. This step will likely involve writing documentation or scripts for how to deploy the FastAPI backend (if not already existing) and ensuring the app can switch modes via a setting.

7. **Testing & QA:** Rigorously test the new application across scenarios:

8. **Functional tests:** Does it capture all content of a sample page (verify by comparing page source and captured data)? Test dynamic sites with lots of XHR calls (e.g. news sites, social media feeds) to ensure nothing is missed. Also test on pages with HTTPS, self-signed certs, etc., to ensure the proxy handles them (this tests our certificate installation process).
9. **Performance tests:** Measure memory and CPU usage with the new app and confirm improvement over Electron baseline. Check that the app remains responsive during heavy captures (if not, consider moving capture sending to a worker thread).
10. **Cross-platform tests:** Build and run the app on Windows, macOS, and Linux. Since PyQt6 and mitmproxy are cross-platform, we anticipate minimal code changes. However, on macOS, for example, we might need to package the QtWebEngine resources properly. Verify that the proxy works on all OS (firewall or security settings might prompt the user on first run, e.g., Windows may ask to allow the proxy on localhost).

11. **Security review:** Although the app runs locally, we should ensure that the proxy cannot be misused by other apps on the machine. By default, mitmproxy on localhost:8080 could theoretically be used by other processes to intercept traffic. We might generate a random port or use authentication on the proxy to prevent misuse. Additionally, ensure the captured data is transmitted to backend securely (HTTPS with certificate verification) and that no sensitive data is accidentally logged or stored in plaintext (especially in cloud mode, where we should use TLS to FastAPI).

12. **Packaging & Deployment:** Create build scripts using PyInstaller (or an alternative like cx_Freeze or Nuitka) to package the application. We will produce executables/installers for each target OS. This will involve specifying data files (like the mitmproxy certificate, any default config, etc.) to include. After building, test the packaged app to make sure everything works in an isolated environment (for example, test on a clean virtual machine or container to catch missing dependencies). Prepare distribution through the appropriate channels (could be direct download, or publishing to an internal app store, etc., depending on the context). Also, if auto-update is desired, implement a mechanism such as checking a JSON feed or using an updater service to fetch new versions.

13. **Documentation and Training:** Update project documentation to reflect the new architecture. This includes developer docs (how the system is structured, how to extend the mitmproxy script, etc.), deployment guides for the backend (if needed), and user documentation explaining how to install and use the new app. Given that originally this project was documented through deep research documents, we should ensure the new approach is equally well-documented for future maintainers. If applicable, include a brief user guide on interpreting the captured data or troubleshooting common issues (e.g., "What to do if pages aren't loading – likely the proxy's certificate isn't trusted" and the steps to fix it).

14. **Future Enhancements (Roadmap):** Once the core system is stable, outline future features that can be layered on. Potential enhancements include: a viewer for captured pages (possibly an interface in the app to browse archived pages or verify their hashes), integration with analysis tools (like running fact-checking algorithms on the captured text), support for scheduling automated captures (like a headless mode that periodically archives certain pages), and multi-user or collaboration features if needed (though that might involve turning the backend into a more complex system). These are beyond the immediate scope, but the new architecture should make them easier to implement. For instance, because we are in Python, integrating an NLP library to analyze page text is straightforward – it could even be done on the client side before uploading the data.

By following this roadmap, we anticipate a transition to the Python-native design that can be accomplished in iterative phases. Each step will be validated to ensure we don't regress on functionality (i.e., we must capture everything we did before, now with less hassle). The end result will be a **more efficient, maintainable, and user-friendly web content capture application** that fulfills the original mission without the baggage of the previous architecture. This evolution demonstrates how embracing simplicity and native integration can lead to a more robust solution in software architecture [11], turning the lessons learned from the Electron attempt into a foundation for long-term success.

---

[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [17] files.diniscruz.ai
https://files.diniscruz.ai/github/pdf/2025/05/21/project__electron-based-web-content-capture-app-with-playwright-and-python.pdf

11 When failing to make something work is a great success , and what... | Dinis Cruz
https://www.linkedin.com/posts/diniscruz_when-failing-to-make-something-work-is-a-activity-7330536672520007680-6029/

12 14 HTML, CSS and JS in a Desktop App... Qt WebEngine vs. Electron vs.?
https://www.pythonguis.com/faq/html-css-and-js-in-a-desktop-app-qt-webengine-vs-electron-vs/

13 If you had to create a desktop app what tech stack would you use? - DEV Community
https://dev.to/perigk/if-you-had-to-create-a-desktop-app-what-tech-stack-would-you-use-3h59

15 16 Intercepting JSON HTTP Responses to Web Browser Page Requests Using MITMProxy – OUseful.Info, the blog...
https://blog.ouseful.info/2019/08/28/intercepting-json-http-responses-to-web-browser-page-requests-using-mitmproxy/

18 Introduction - mitmproxy docs
https://docs.mitmproxy.org/stable