Coventry University

**Application Development for Rapid Forensic Triage from Digital Evidence Images**

|  |  |
| --- | --- |
| **Name** | Syed Naveed Hasan |
| **Student ID** | 15235718 |
| **Submitted to** | Theodoros Spyridopoulos |
| **Due Date** | 12/08/2025 |
| **Module** | 7030CEM |

# Acknowledgements

I would first and foremost like to express my sincere gratitude to my supervisor, Mr Theodoros Spyridopoulos, for their invaluable guidance, unwavering support, and insightful feedback throughout this project. Their expertise and encouragement were instrumental in the development of this research and the writing of this report.

I would also like to extend my thanks to the academic and technical staff of Cyber Security at Coventry University, especially Mr Antal Goldschmidt, for their assistance and for providing a stimulating academic environment.

My gratitude also goes to my peers and colleagues for the engaging discussions and for creating a supportive atmosphere for learning and research.

# Abstract

The initial triage phase of digital forensic investigations is often a time-consuming and manually intensive process, creating a bottleneck in identifying critical evidence. This project addresses this challenge through the design and development of a standalone application for rapid forensic triage and user activity analysis from digital evidence images.

Developed using Python, the application is engineered to mount and parse standard digital evidence formats, such as .E01 images. Its core methodology involves the automated extraction and analysis of high-value forensic artifacts, with a primary focus on Windows Registry hives (SYSTEM, NTUSER.DAT, SAM) to reconstruct a timeline of user actions.

The developed tool successfully identifies and correlates key indicators of user activity, including recently used files, connected USB devices, and network history, presenting these findings in a centralized and intuitive graphical user interface. The research demonstrates that by automating the analysis of key artifacts, the time required for initial evidence assessment can be significantly reduced. The resulting application serves as a proof-of-concept that empowers investigators to make faster, more informed decisions at the critical outset of an investigation.

Contents

[Acknowledgements 2](#_Toc204744597)

[Abstract 3](#_Toc204744598)

[Introduction 5](#_Toc204744599)

[Background Information 7](#_Toc204744600)

[Requirements Analysis 8](#_Toc204744601)

[Functional Requirements 9](#_Toc204744602)

[Non-Functional Requirements 10](#_Toc204744603)

[Design Diagrams 11](#_Toc204744604)

[System Architecture: 11](#_Toc204744605)

[Use Case Diagram 12](#_Toc204744606)

[Primary Use Cases: 13](#_Toc204744607)

[Sequence Diagrams 14](#_Toc204744608)

[Sequence Diagram: Evidence Loading and System Identification 14](#_Toc204744609)

[Sequence Diagram: User Enumeration 15](#_Toc204744610)

[Sequence Diagram: User Profile Analysis 17](#_Toc204744611)

[Implementation 19](#_Toc204744612)

[Project Limitations 22](#_Toc204744613)

[Future Implementation 23](#_Toc204744614)

[Methodology: An Integrated Approach to Forensic Tool Construction 25](#_Toc204744615)

[Literature review 31](#_Toc204744616)

[Findings 33](#_Toc204744617)

[Analysis/Discussion 34](#_Toc204744618)

[Conclusions 35](#_Toc204744619)

[Recommendations 36](#_Toc204744620)

[Bibliography 37](#_Toc204744621)

[Appendices 38](#_Toc204744622)

[A Glossary of Terms 38](#_Toc204744623)

# Introduction

This project is focused on enhancing digital forensic investigations by addressing the inherent complexities of analyzing fragmented user activity data within the Windows Registry. The scope encompasses the design and development of a specialized digital forensic analysis tool that will concentrate on the automated extraction, sophisticated correlation, and integrated presentation of diverse registry artifacts pertaining to user activity. Key artifact types to be investigated and correlated include last logon timestamps, which involves identifying and reconciling various "last logon" and "last activity" timestamps from multiple registry locations such as SAM hive, Winlogon keys, and UserAssist entries to derive a most likely accurate timeline. Additionally, the tool will focus on USB device connection history by extracting and linking records of connected USB devices to specific user profiles and activity sessions, program execution history through analyzing registry-based evidence of application launches from sources like ShimCache and AmCache.hve to build a comprehensive picture of software usage, and folder access history using Shellbags to recover and correlate data on accessed folders, including local, removable, and network shares, to understand user navigation patterns. The solution will also incorporate email activity referencing by leveraging existing email artifact processing capabilities of forensic platforms to link relevant email communications to user activity profiles.

The solution aims to provide a cohesive, context-rich view of user actions directly within its interface, thereby reducing manual analysis overhead and improving the depth of investigative insights. This custom analytical solution is envisioned to be implemented as a dedicated application, with potential approaches including development as a module for an existing forensic platform or as a standalone Python-based tool.

The primary aim of this project is to develop a specialized digital forensic application capable of performing advanced, automated correlation of Windows Registry data and related user activity artifacts. This will enable investigators to gain a more comprehensive, accurate, and actionable understanding of a user's digital footprint, ultimately enhancing the efficiency and effectiveness of digital forensic investigations.

To achieve the stated aim, the project will pursue several specific objectives. First, automated registry artifact extraction will be implemented through robust mechanisms for the automated extraction of critical user activity-related data from various Windows Registry hives including SAM, SOFTWARE, SYSTEM, NTUSER.DAT, and AmCache.hve. Second, refined last logon correlation will be achieved by developing and applying sophisticated correlation algorithms to reconcile and present a "most likely" accurate last logon timestamp by integrating data from multiple, potentially conflicting, registry sources. Third, contextual user profile generation will be accomplished by automatically linking extracted logon events with associated USB device connections, program executions, and folder access histories to create enriched user activity profiles. Fourth, integrated email activity referencing will incorporate references to relevant email communications into user activity profiles by querying and utilizing email artifacts already processed by a forensic analysis environment. Fifth, heuristic-based suspicious activity flagging will be implemented through basic heuristic rules for flagging potentially suspicious characteristics within the extracted user activity data, such as unusual program executions or specific keywords in email references, to draw investigator attention to anomalies. Finally, integrated findings presentation will ensure that all extracted, correlated, and flagged user activity data is presented clearly and cohesively within the application's interface, facilitating intuitive analysis and interpretation by forensic examiners.

# Background Information

Digital forensics is a critical branch of forensic science concerned with the recovery, investigation, analysis, and preservation of material found in digital devices, often in relation to computer crime (Mohammed, 2025). The goal of a digital forensic investigation is to reconstruct events by systematically examining digital media to extract evidence that can be used in a court of law. This process must be conducted in a forensically sound manner, ensuring that the original evidence is not modified and that a clear, documented chain of custody is maintained at all times.

The standard digital forensic investigation process typically follows a structured methodology, which includes stages of seizure, acquisition, analysis, and reporting. The acquisition phase is of particular importance, as it involves creating a bit-for-bit copy, or a "forensic image," of the original storage media. This image is an exact duplicate of the original device, including all allocated and unallocated space, ensuring that the investigation is conducted on a copy rather than the original evidence, thereby preserving its integrity. These images are commonly stored in standardized, vendor-neutral formats, with the Expert Witness Format (.E01) being one of the most widely used in the industry. The .E01 format encapsulates the raw disk image along with metadata, case information, and checksums (e.g., MD5, SHA-1) to verify the integrity of the evidence over time.

A significant portion of the analysis phase is dedicated to examining artifacts left behind by the operating system and user activities. In investigations involving Microsoft Windows systems, the Windows Registry is an artifact of paramount importance. The Registry is a hierarchical database that stores low-level settings for the operating system and for applications that opt to use it. It contains a vast wealth of information about the system's configuration and the user's interaction with it. Key registry files, known as hives (e.g., NTUSER.DAT, SYSTEM, SAM), store critical data such as user login times, recently accessed files, network connection history, and details of external devices (like USB drives) that have been connected to the system.

However, manually navigating and correlating information from these disparate sources and registry hives is a complex and time-consuming task. An investigator often has to use multiple tools and manually piece together data to form a coherent picture of user activity. This initial triage stage represents a significant bottleneck in the investigative workflow. The demand for tools that can automate the extraction and analysis of these key artifacts is therefore high, as they can significantly accelerate the investigation and allow forensic professionals to focus on higher-level analysis and interpretation of the evidence.

# Requirements Analysis

This section outlines the requirements for a standalone Python application designed to provide forensic investigators with rapid insights into user activity by analysing evidence files, specifically in the .E01 format. The primary goal is to quickly extract and present high-value forensic artifacts, such as registry data and emails, through an intuitive and visually appealing user interface.

The application will correlate disparate data points to present a unified view of user activity, saving the investigator significant analysis time. The primary user for this tool is the Forensic Investigator or Digital Forensics Analyst.

This user is assumed to be proficient in digital forensics principles and familiar with evidence handling procedures but may not be an expert in command-line tools or scripting. The intended user values efficiency and clear, actionable intelligence.

From a functional perspective, the system **must** allow the user to select and load a single-segment .E01 evidence file and **should** also support multi-segment files. It is imperative that the system mounts the evidence file's file system in a read-only mode to ensure forensic integrity. Once mounted, the system **must** provide a graphical file browser for the investigator to navigate the directory structure within the evidence file. A core function is the automated extraction and parsing of key Windows Registry hives, including NTUSER.DAT, SAM, SYSTEM, SOFTWARE, and SECURITY. From these hives, the system **must** extract a range of specific artifacts. This includes data related to user activity, such as Most Recently Used (MRU) lists, recently opened documents, and shellbags for folder access history. It will also gather system information, including operating system details, computer name, and time zone settings. Furthermore, the tool will analyze network activity by identifying network profiles and past wireless networks.

A critical component is the extraction of device history, which involves collecting connected USB device serial numbers, vendor/product IDs, and their last connection times.

Finally, it will retrieve information on user accounts, providing a list of local user accounts and their basic profile information. The system **should** also be able to detect and parse common email archive formats like .pst or .mbox, extracting key metadata such as sender, recipient, subject, and date/time.

All extracted artifacts **must** be presented in a clean, organized, and easily understandable dashboard. This user interface **must** categorize findings into logical groups like "User Accounts" or "USB Devices" and **should** use visual elements like timelines or tables to display time-sensitive data. The overall UI **must** be visually appealing and professional. To conclude the workflow, the system **should** allow the user to export the summarized findings into a simple report format, such as HTML or PDF.

Beyond its core functions, the system's performance and quality attributes are critical. The initial analysis and display of "quick insights" from the registry **must** be completed within a reasonable time frame, ideally under three minutes for a typical user profile, and the UI **must** remain responsive during processing, displaying a progress indicator for long-running tasks.

The application **must** be intuitive, requiring minimal training for a forensic investigator, and all data presented **must** be clearly labelled with its source to ensure credibility. Reliability and accuracy are paramount; therefore, the data parsing libraries used **must** be well-vetted and accurate. The application **must** also handle potential errors gracefully, such as corrupted registry hives, without crashing. To maintain forensic integrity, the application **must** not write any data back to the loaded evidence file under any circumstances. To verify this, the system **must** calculate and display a hash value (e.g., MD5, SHA-1) of the .E01 file upon loading, allowing for comparison against a known value.

Regarding compatibility, the core Python backend **must** support various versions of Windows, and the final application **must** run on a modern Windows (10/11) 64-bit system, though Linux is not currently a target. Finally, the architecture **should** be modular, allowing for future expansion to include parsers for other artifacts, such as browser history or event logs, without requiring a complete redesign.

## Functional Requirements

|  |  |  |  |
| --- | --- | --- | --- |
| ID | Requirement Name | Description | Priority |
| 1 | Load Evidence File | The system **must** allow the user to select and load single-segment and multi-segment .E01 evidence files. | High |
| 2 | Mount File System (Read-Only) | The system **must** mount the evidence file's file system in a strictly read-only mode to ensure forensic integrity. | High |
| 3 | Automated Registry Parsing | The system **must** automatically locate, extract, and parse key Windows Registry hives, including SAM, SYSTEM, SOFTWARE, SECURITY, and NTUSER.DAT. | High |
| 4 | User Account Extraction | The system **must** extract and display a list of local user accounts, their RIDs, and associated profile information from the SAM hive. | High |
| 5 | User Activity Artifacts | The system **must** extract user activity data such as Most Recently Used (MRU) lists, recent documents, and shellbags from the registry. | High |
| 6 | USB Device History | The system **must** extract the history of connected USB devices, including serial numbers, vendor/product IDs, and last connection times. | High |
| 7 | Organized Artifact Dashboard | All extracted artifacts **must** be presented in a clean, organized, and easily understandable dashboard, categorized into logical groups. | High |
| 8 | Email Archive Parsing | The system **must** be able to detect and parse common email formats (e.g., .eml files in directories) to extract metadata and content. | Medium |
| 9 | System Information Extraction | The system **must** extract general system information, including OS version, computer name, and time zone settings. | Medium |
| 10 | Network Information Extraction | The system **must** analyze and display network information, including network profiles and past wireless network connections (SSIDs). | Medium |
| 11 | File System Browser | The system **must** provide a graphical file browser for the investigator to navigate the directory structure within the mounted evidence file. | Medium |
| 12 | Report Generation | The system **must** allow the user to export the summarized findings into a simple report format, such as HTML or PDF. | Medium |
| 13 | AI-Powered Email Summarization | The system **should** use a Large Language Model (LLM) to analyze the content of extracted emails and generate a concise summary. | Medium |

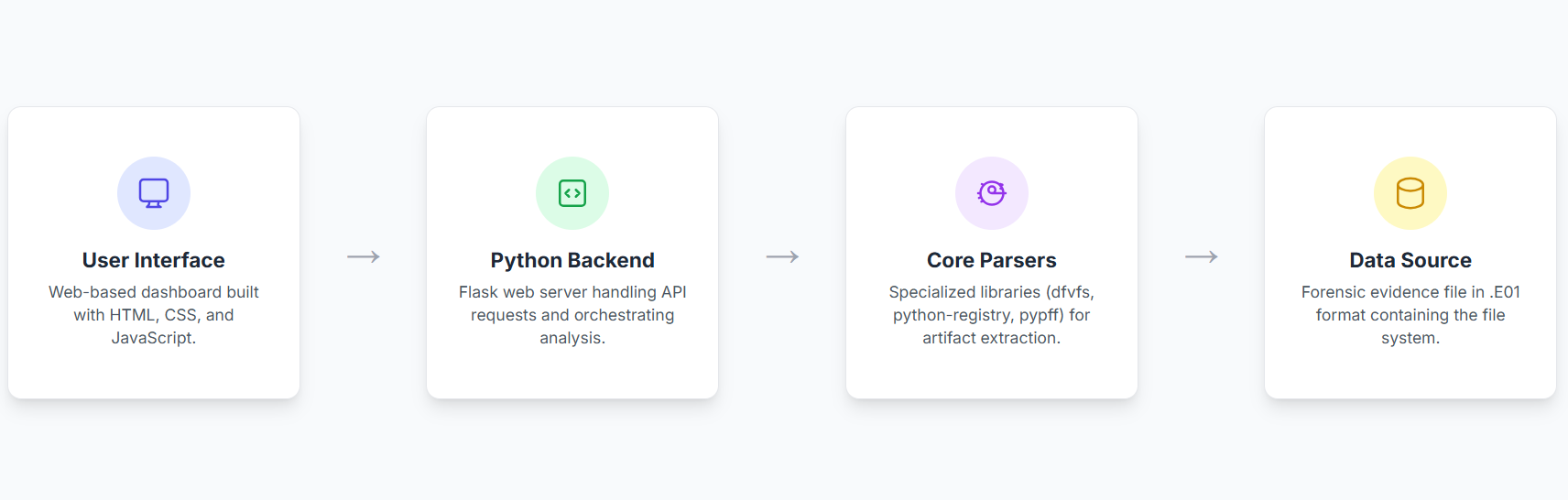
## Non-Functional Requirements

|  |  |  |  |
| --- | --- | --- | --- |
| ID | Category | Description | Priority |
| 1 | Forensic Integrity | The application **must** never write any data back to the loaded evidence file. The system will display a hash of the file on load to verify integrity. | High |
| 2 | Performance | The initial analysis and display of key registry artifacts **should** be completed in under three minutes for a typical system image. | High |
| 3 | Usability | The user interface **should** be intuitive and visually appealing, requiring minimal training for a qualified forensic investigator. | Medium |
| 4 | Responsiveness | The UI **must** remain responsive during long-running tasks, providing clear progress indicators to the user. | Medium |
| 5 | Reliability | The application **must** handle potential errors gracefully (e.g., corrupted files, missing hives) without crashing. | High |
| 6 | Accuracy | Data parsing **must** be highly accurate, relying on well-vetted, industry-standard libraries for artifact extraction. | High |
| 7 | Security & Privacy | All processing **must** occur locally on the investigator's machine. No evidence data should be transmitted over a network to external services. | High |
| 8 | Compatibility | The application **must** run on modern Windows (10/11) 64-bit operating systems. | High |
| 9 | Maintainability | The system architecture **must** be modular to allow for future expansion (e.g., adding new parsers) without requiring a complete redesign. | Medium |
| 10 | Clarity | All data presented in the UI **must** be clearly labelled with its source artifact and path to ensure transparency and credibility. | High |

# Design Diagrams

## System Architecture:

The forensic triage tool is designed based on a modular, four-tier architecture, as illustrated in Figure below. This layered approach was chosen to promote a clear separation of concerns, enhance maintainability, and allow for future scalability. Each component has a distinct responsibility, and data flows logically from the evidence source through the processing engine to the end-user.



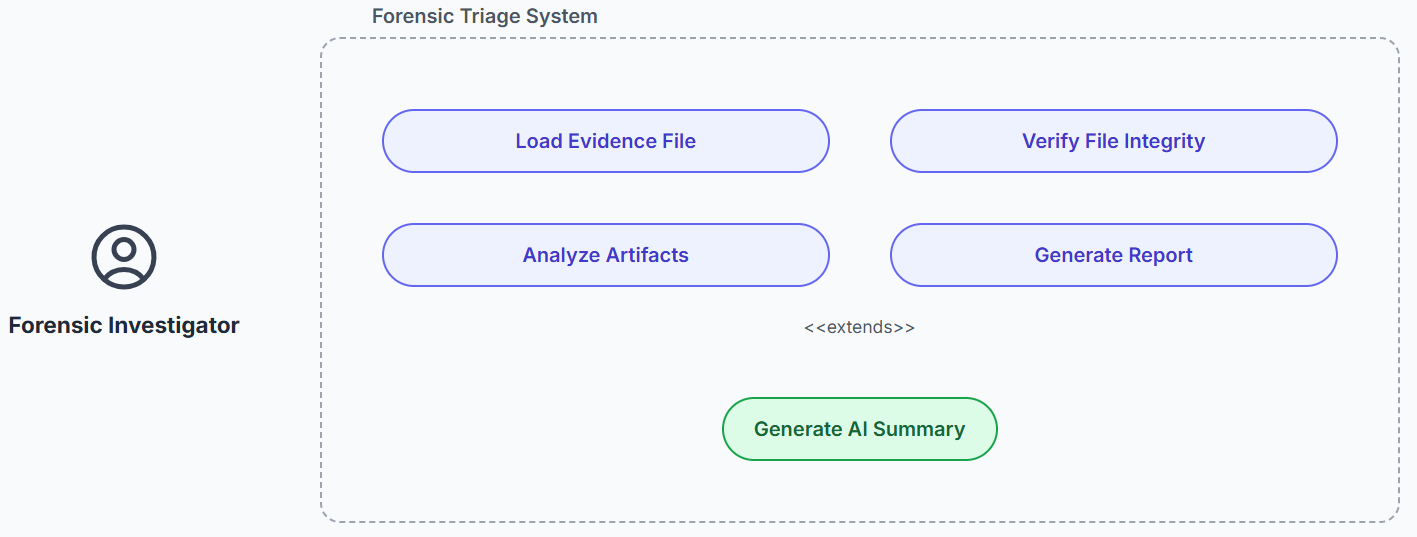
The four primary components of the architecture are:

1. **Data Source:** This is the foundational layer of the system, representing the raw forensic evidence. The primary data source is a disk image in the EnCase Evidence File Format (.E01). This component contains the complete file system of the subject machine, including all system files, user profiles, registry hives, and other artifacts to be analyzed.
2. **Core Parsers:** This layer acts as the processing engine of the tool. It is composed of a collection of specialized, open-source Python libraries chosen for their reliability and acceptance within the digital forensics community. Key libraries include dfvfs for navigating the file system within the .E01 image, python-registry for parsing the complex binary structure of Windows Registry hives, and pypff for handling email archives. This layer is responsible for the low-level extraction and conversion of raw data into structured information.
3. **Python Backend:** The backend serves as the central orchestrator of the application. Built using the Flask web framework, it handles all application logic. Its responsibilities include receiving requests from the user interface, invoking the appropriate core parsers to perform analysis on the data source, processing the results, and exposing the findings through a series of well-defined API endpoints. This component effectively decouples the user interface from the complex parsing logic.
4. **User Interface (UI):** This is the presentation layer and the sole point of interaction for the forensic investigator. It is a web-based single-page application (SPA) rendered in the user's browser. Developed using standard HTML, CSS, and JavaScript, the UI communicates with the Python backend via API calls to request data and display the results. It is responsible for presenting the extracted artifacts in a clean, organized, and visually intuitive dashboard, allowing the investigator to quickly assess the findings.

The unidirectional data flow, as indicated by the arrows, ensures a predictable and robust process: the backend requests data from the parsers, which read from the source, and the UI requests information from the backend. This modular design ensures that, for instance, a new parser for a different artifact type can be added to the Core Parsers layer with minimal or no changes required in the User Interface.

## Use Case Diagram

To define the functional scope of the system from a user's perspective, a Use Case diagram is presented in the figure below. This diagram identifies the primary actor who interacts with the system and the key goals they can achieve.



**Forensic Investigator:**

This represents the primary and sole user of the system. The investigator is a trained professional whose goal is to analyze digital evidence efficiently.

**System Boundary:**

The rectangle labelled "Forensic Triage System" defines the scope of the application. All use cases within this boundary represent functions provided by the software itself.

### Primary Use Cases:

**Load Evidence File:**

This is the initial interaction where the investigator selects and loads an .E01 evidence file into the application for analysis.

**Verify File Integrity:**

A critical step where the system calculates and displays a cryptographic hash of the loaded evidence file, allowing the investigator to confirm that it has not been altered.

**Analyze Artifacts:**

This is the core and most complex use case. It represents the investigator's action of examining the various artifacts extracted by the tool, such as user accounts, USB device history, network connections, and registry data.

**Generate Report:**

This use case allows the investigator to export the summarized findings from the dashboard into a portable format (e.g., HTML or PDF) for documentation or presentation.

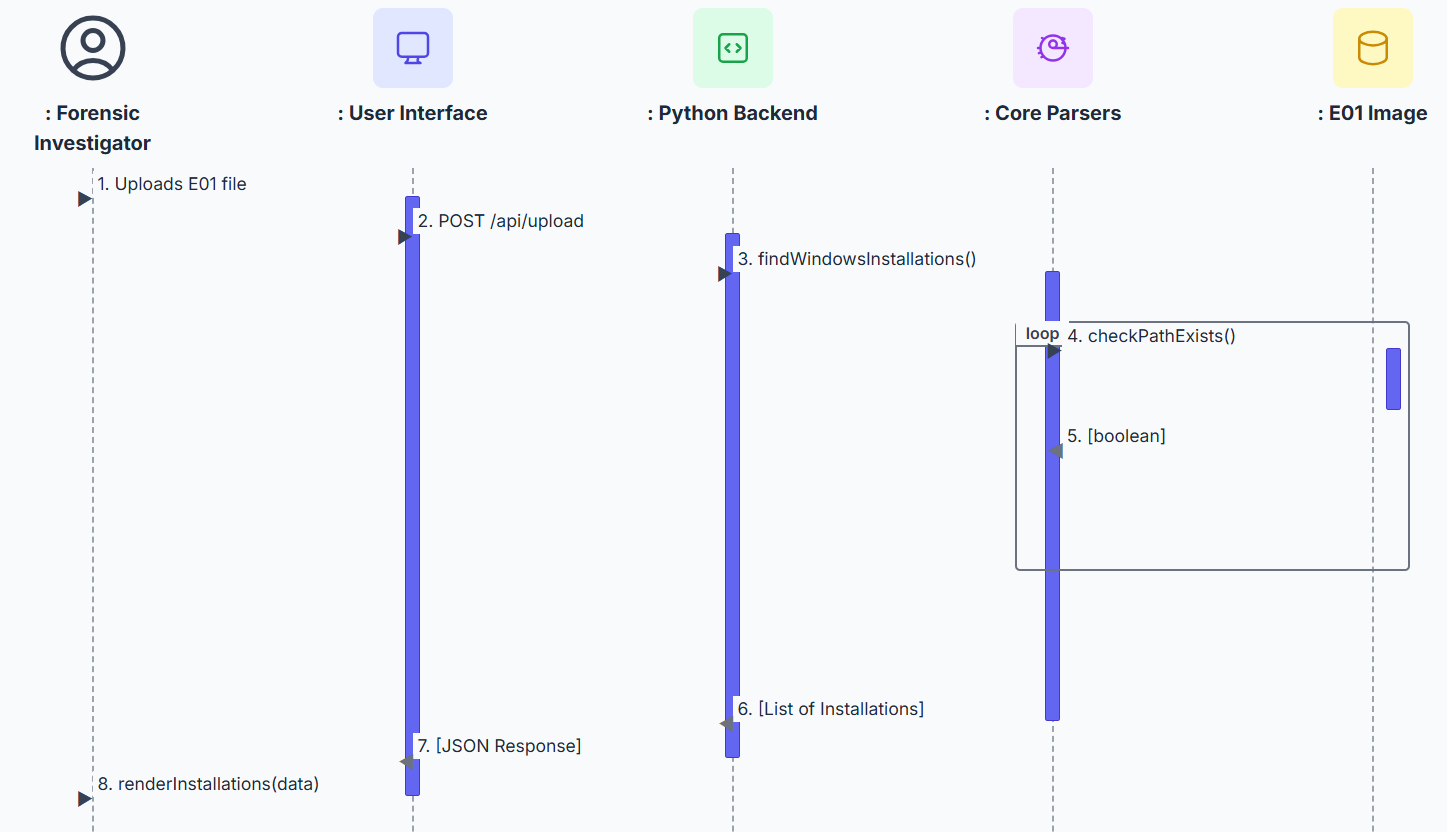
**Extending Use Case:**

Generate AI Summary: This use case is shown with an <<extends>> relationship to "Analyze Artifacts." This indicates that generating an AI-powered summary of emails is an optional, specialized function that extends the primary analysis capability. The investigator can choose to perform a standard artifact analysis without ever invoking this advanced feature.

## Sequence Diagrams

### Sequence Diagram: Evidence Loading and System Identification

The initial interaction with the triage tool involves loading the evidence file and identifying viable operating systems for analysis. Figure below illustrates the sequence of events that occur when the Forensic Investigator uploads an .E01 file.



**Participants (Lifelines):**

The diagram includes lifelines for each of the major architectural components, plus the human actor:

* **:Forensic Investigator:** The user initiating the action.
* **:User Interface:** The web-based frontend the investigator interacts with.
* **:Python Backend:** The Flask server that orchestrates the process.
* **:Core Parsers:** The collection of data extraction libraries.
* **:E01 Image:** The raw evidence file.

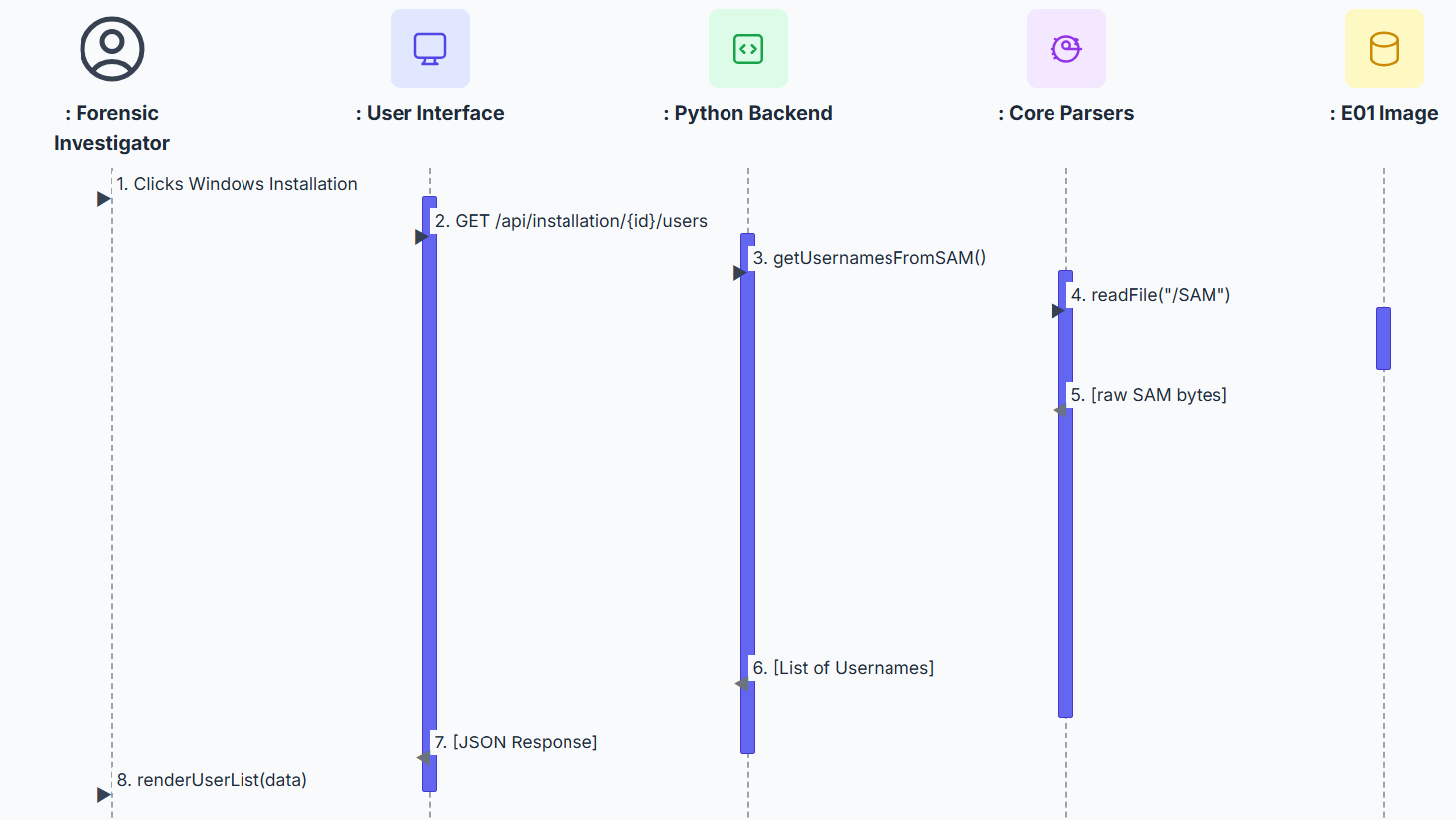
**Interaction Sequence:**

1. **Upload E01 File:** The investigator selects the .E01 evidence file for processing via the User Interface.
2. **API Request:** The UI sends the file data to the Python Backend, typically via an HTTP POST request to an /api/upload endpoint.
3. **Find Windows Installations:** Upon receiving the file, the Backend invokes a high-level function within the Core Parsers layer to begin analysis. The goal is not to parse deep artifacts yet, but simply to identify which partitions within the image contain a Windows operating system.
4. **Check Path Exists (Loop):** This is a critical step shown within a **loop fragment**. The Core Parsers enumerate all partitions found within the E01 image. For each partition, it attempts to access a known, critical Windows directory (e.g., /Windows/System32/config). This check is a reliable heuristic for confirming the presence of a Windows installation.
5. **Return Path Status:** The file system driver (dfvfs) returns a boolean (true/false) indicating whether the path exists on that partition.
6. **Return List of Installations:** After looping through all partitions, the Core Parsers compile a list of the partitions that returned true and sends this list back to the Python Backend.
7. **Return JSON Response:** The Backend formats the list of valid installations into a JSON object and sends it back to the User Interface.
8. **Render Installations:** The UI receives the JSON data and dynamically renders a list of the identified Windows installations, presenting them to the investigator for selection.

This sequence effectively models the initial triage step, addressing the need to guide the user to valid targets for analysis rather than presenting them with a raw list of all partitions.

### Sequence Diagram: User Enumeration

Following the identification of a valid Windows installation, the next logical step for the investigator is to discover the user accounts present on that system. The following figure illustrates the sequence of interactions that occur when the investigator selects a specific installation to analyze.



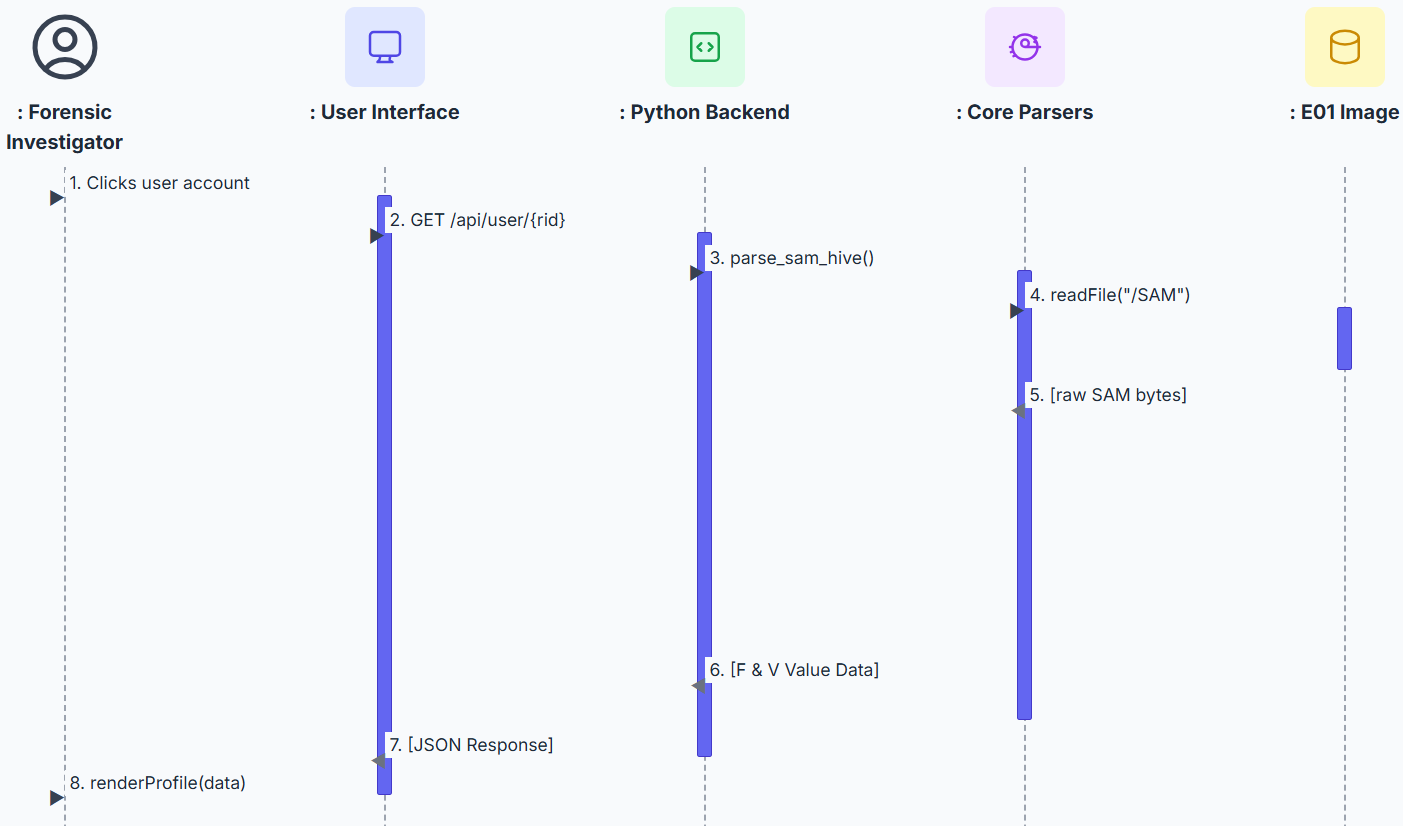
**Interaction Sequence:**

1. **Click Windows Installation:** The investigator selects one of the previously identified Windows installations from the list presented in the User Interface.
2. **API Request:** The UI sends an API request to the Python Backend, including the identifier for the selected installation (e.g., the partition number), to ask for a list of all users.
3. **Get Usernames from SAM:** The Backend receives the request and calls the relevant function in the Core Parsers layer, tasking it with extracting usernames from the SAM hive of the specified installation.
4. **Read SAM File:** The Core Parsers layer constructs the path to the SAM hive within the selected partition (e.g., /p2/Windows/System32/config/SAM) and requests the file's contents from the E01 Image.
5. **Return Raw SAM Bytes:** The file system driver reads the raw binary data of the SAM hive from the disk image and returns it to the parser.
6. **Return List of Usernames:** The parser processes the binary SAM hive, enumerates the user accounts, and returns a simple list of usernames and their corresponding RIDs to the Python Backend.
7. **Return JSON Response:** The Backend formats this list into a JSON array and sends it back to the User Interface.
8. **Render User List:** The UI receives the list of users and dynamically renders it on the screen, presenting the investigator with a list of accounts that can be selected for detailed analysis.

This sequence effectively bridges the gap between system-level analysis and user-level analysis, showing how the tool efficiently enumerates potential subjects of interest for the investigator.

### Sequence Diagram: User Profile Analysis

Figure below details the sequence of interactions for the "Analyze User Profile" scenario, which is a specific instance of the "Analyze Artifacts" use case. This diagram shows the flow of messages between the architectural components over time.



**Interaction Sequence:**

The sequence of events unfolds as follows:

1. **Click User Account:** The process begins when the Forensic Investigator clicks on a specific user account within the User Interface.
2. **API Request:** The UI translates this click into an asynchronous API call (GET /api/user/{rid}) to the Python Backend, requesting the detailed data for the selected user's Relative ID (RID).
3. **Invoke Parser:** The Backend receives the request and invokes the appropriate function within the Core Parsers layer (e.g., parse\_sam\_hive()) to get the necessary F and V value data from the SAM hive.
4. **Read File from Image:** The Core Parsers layer, using dfvfs, requests the raw bytes of the SAM registry hive from the E01 Image file.
5. **Return Raw Data:** The file system driver reads the data from the disk image and returns the raw SAM hive as a sequence of bytes to the parser.
6. **Return Parsed Data:** The parser processes the raw bytes, extracts the relevant F and V value structures for the requested user, and returns this structured data to the Python Backend.
7. **Return JSON Response:** The Backend formats the structured data into a JSON object and sends it back to the User Interface as the HTTP response to the initial API request.
8. **Render Profile:** The UI's JavaScript receives the JSON data and dynamically updates the HTML of the page, rendering the user's profile information for the investigator to view.

This sequence demonstrates the clear, decoupled nature of the architecture. The UI is only concerned with making an API call and displaying the result, while the backend handles the complex orchestration of reading and parsing the underlying forensic artifacts.

# Implementation

This section details the technical implementation of the Forensic Triage Tool, translating the architectural design into a functional application. The implementation leverages a combination of the Python Flask framework for the backend, standard web technologies for the frontend, and specialized forensic libraries for the core analysis engine.

**Core Technologies and Libraries**

The project is built upon a stack of open-source libraries chosen for their robustness and suitability for forensic tasks:

* **Backend Framework:** **Flask** is used as the web server. Its lightweight and modular nature makes it ideal for creating the RESTful API that serves data to the frontend.
* **Evidence File Access:** **Digital Forensics Virtual File System (dfvfs)** is the cornerstone of the file system interaction. It provides the necessary abstractions to mount and navigate the file system within the .E01 evidence file in a forensically sound, read-only manner.
* **Registry Parsing:** The **python-registry** library is employed to parse the complex binary structure of the Windows Registry hives, specifically the SAM hive. This library can interpret keys, values, and data types within the raw hive files.
* **Email Parsing:** Python's built-in **email** library is used to parse the content of individual .eml files, extracting metadata like sender, recipient, and subject, as well as the text body.
* **AI Summarization:** The **google-generativeai** SDK provides the interface to the Google Gemini API. It is used to send the aggregated text content of emails to the gemini-1.5-flash model for analysis and summarization.
* **Frontend Technologies:** The user interface is a single-page application built with standard **HTML**, **CSS**, and **JavaScript**. **Tailwind CSS** is used for styling to create a modern and responsive layout.

**Backend Implementation (app.py and api\_methods/)**

The backend is organized into a main application controller (app.py) and a dedicated module directory (api\_methods/) that contains all the specialized forensic and data processing logic.

**Main Application (app.py)**

The app.py script serves as the central hub of the application. Its primary responsibilities are:

1. Initializing the Flask web application.
2. Defining the API endpoints (routes) that the frontend can call.
3. Handling HTTP requests, calling the appropriate functions from the api\_methods modules, and returning the results as JSON responses.
4. Serving the HTML templates from the templates/ directory.

**Evidence File Interaction**

Interaction with the .E01 image is handled by a set of dedicated modules. The process begins with identifying valid targets for analysis.

* **check\_partitions.py**: This script implements the logic described in the "Evidence Loading" sequence diagram. It uses dfvfs to enumerate all partitions within the E01 image and checks each one for the presence of a Windows installation. The primary function in this module directly implements the findWindowsInstallations() operation shown in the design diagram.
* **load\_file\_from\_e01.py & file\_extraction.py**: These modules provide utility functions for accessing files within the mounted image. load\_file\_from\_e01.py reads a specified file (like a registry hive) directly into a memory buffer (a BytesIO object), while file\_extraction.py saves a file to the local uploads/ directory, which is useful for larger files or for caching purposes.

**Registry Parsing Engine**

The core of the user analysis is the registry parsing engine, which focuses on the SAM hive.

* **get\_usernames\_and\_rids.py**: This script is responsible for the initial user enumeration. It reads the SAM hive and parses the SAM\Domains\Account\Users\Names key to produce a list of all local user accounts and their corresponding Relative IDs (RIDs). This module's main function implements the getUsernamesFromSAM() operation from the User Enumeration sequence diagram.
* **f\_value.py & v\_value.py**: These are the low-level parsers for the user account data. Based on the structures defined in their corresponding CSV files (f\_value\_offsets.csv, v\_value\_header\_offsets.csv), these scripts use Python's struct module to unpack the binary F and V value data for a given user. They are responsible for extracting timestamps, the RID, login counts, and pointers to string data like the username and comments.
* **f\_value\_flags.py**: This module works in conjunction with f\_value.py. It takes the numerical User Account Control (UAC) value and, using the uac\_flags.csv file, decodes it into a human-readable list of the active flags (e.g., "Account Disabled," "Password does not expire").
* **API Handlers (get\_user\_\*\_with\_rid.py)**: A series of scripts are dedicated to exposing the parsed data through the API. For example, get\_user\_f\_value\_data\_with\_rid.py takes a RID from the API request and orchestrates the reading and parsing of the F value. These handlers represent the implementation of the "Analyze User Profile" sequence diagram, where the abstract parse\_sam\_hive() operation is realized by calls to the functions within f\_value.py and v\_value.py.

**Email Analysis and AI Integration**

The email analysis feature is implemented as a multi-step process:

* **collect\_user\_emails.py**: This script implements the logic for email discovery. It takes a username and recursively navigates that user's profile directory (specifically /Users/{username}/AppData/Local/Microsoft/Windows Mail/) to find all files with a .eml extension. It returns a comprehensive list of file paths for all found emails.
* **email\_ai\_analysis.py**: This is the core AI module. It takes the list of email paths, reads each .eml file from the E01 image, parses its content to extract the sender, subject, and body, and then aggregates this text. Finally, it constructs a detailed prompt and sends the aggregated text to the Gemini API via the google-generativeai library. The summary returned by the API is then passed back to the frontend.

**Frontend Implementation (templates/)**

The frontend is a single-page application designed for clarity and ease of use. The key file, profile.html, uses JavaScript to create a dynamic user experience.

1. **Initialization:** When the page loads, it retrieves the partition ID and user RID from the URL.
2. **Asynchronous API Calls:** It uses the browser's fetch API to make parallel, asynchronous calls to the various backend endpoints (e.g., /api/.../get\_user\_f\_value\_data\_with\_rid/..., /api/.../get\_user\_v\_value\_data\_with\_rid/...).
3. **Dynamic Rendering:** As the JSON data is returned from the APIs, JavaScript functions parse the data and dynamically build the HTML to display it. This includes populating the "Forensic Summary" table and the detailed, expandable sections for the raw F-Value, V-Value, and UAC flag data.
4. **User Interaction:** Event listeners are attached to buttons (e.g., "Analyze Emails," "Show Raw Details"). When clicked, these trigger additional API calls or toggle the visibility of different sections on the page, providing an interactive experience without requiring a page reload.

# Project Limitations

While the Forensic Triage Tool successfully demonstrates its core concepts, it is essential to acknowledge the limitations inherent in this Proof of Concept (PoC), which define the boundaries of the current implementation as a result of strategic scoping decisions. Architecturally, the tool is implemented as a web application running on a local Flask server. While ideal for rapid prototyping, this is not suitable for a typical forensic environment, where a production-grade tool would need to be a self-contained, portable executable. Furthermore, a key non-functional requirement for local processing was intentionally deviated from for the AI summarization feature, which uses the online Google Gemini API. This decision was made to rapidly demonstrate the *capability* of integrating an LLM without the significant hardware and configuration overhead required for a local model like Llama. It is explicitly noted that for this project to become a viable product, this external dependency must be removed and replaced with a locally-run model to ensure all evidence remains on the investigator's machine (Valli, 2008; Sharma, Baggili, Ghawaly, McCleary, & Webb, 2025).

|  |  |  |  |
| --- | --- | --- | --- |
| Limitation Area | PoC Implementation | Forensic Best Practice/Requirement | Key Supporting Citations |
| Tool Architecture | Flask-based web application running on a local server. | Tools should be self-contained, portable executables to minimize the evidentiary footprint on the target system. | 3 |
| Data Processing & Sovereignty | Use of the external Google Gemini API for AI summarization. | All evidence processing must remain local to the investigator's machine to maintain an unbroken Chain of Custody and ensure data integrity. | 5 |
| User Activity Analysis | Deferred parsing of MRU lists and Shellbags from NTUSER.DAT. | Analysis of MRU and Shellbags is fundamental for reconstructing user activity, knowledge, and intent, even for deleted or external data. | 7 |
| Physical Device Tracking | Deferred analysis of the SYSTEM hive for USB history. | Tracing USB device connections is critical for linking digital evidence to physical items and investigating data exfiltration or malware introduction. | 9 |
| Data Identification Integrity | File identification is based solely on file extensions. | File identification must be performed via file signature analysis (magic numbers) to reliably determine file type and detect obfuscation. | 11 |
| Data Accessibility | Inability to process encrypted or password-protected content. | Modern forensic tools must be able to identify, and ideally handle, encrypted data, as it is a ubiquitous feature of modern systems. | 13 |

In terms of feature scope, the implementation was focused on the most complex, user-centric data sources to prove the tool's core value within the project's timeframe. Consequently, several high-priority features identified in the requirements analysis were not implemented. These include the parsing of general user activity artifacts like Most Recently Used (MRU) lists and shellbags from the NTUSER.DAT hive, the analysis of the SYSTEM hive for USB device history, and the extraction of system and network information from the SOFTWARE and SYSTEM hives. Additionally, a graphical file system browser and an automated report generation feature, while part of the design, were deferred (Zhu, 2009). This deliberate scoping choice allowed development to concentrate on the end-to-end workflow of user enumeration, deep SAM hive analysis, email collection, and the novel integration of AI.

Finally, the tool has several forensic and analytical limitations. The implementation is heavily focused on artifacts directly related to user accounts (SAM data) and communications (emails), and does not perform a broader analysis of system-level files. The tool also cannot process encrypted or password-protected content. At present, file identification is based on file extensions, and analysis of file headers has not been included. The accepted forensic standard is to identify files via file signature analysis, which inspects the "magic numbers" at the beginning of a file—for example, JPEG files begin with the hexadecimal bytes FF D8 FF, and Windows Portable Executable (PE) files begin with the ASCII characters "MZ". These boundaries underscore the tool's current role as a rapid triage instrument, not a replacement for comprehensive, deep-dive forensic analysis.

## Future Implementation

To evolve the tool from a Proof of Concept into a production-grade forensic utility, future work will be focused on addressing the current limitations and implementing the full scope of planned features. This comprehensive development roadmap encompasses several critical areas that will transform the tool's capabilities and operational effectiveness.

The most fundamental aspect of this evolution involves a complete architectural transformation. The highest priority is to re-architect the application from a Flask-based web server into a self-contained, portable executable. This transition will eliminate the need for installing a runtime environment, minimize the tool's evidentiary footprint on the analysis machine, and align with forensic best practices. Concurrently, the external dependency on the Google Gemini API will be removed through the integration of an open-source, locally-run Large Language Model (LLM) for the summarization feature. This ensures that all evidence processing occurs entirely on the investigator's workstation, preserving data sovereignty and maintaining an unbroken chain of custody.

The enhancement of core forensic capabilities represents another crucial development area. A mechanism for file signature analysis will be implemented to replace the current reliance on file extensions. The tool will be programmed to read the initial bytes ("magic numbers") of files to accurately determine their true type, allowing it to identify obfuscated files that have been intentionally mislabelled. Additionally, functionality will be added to detect the presence of common encryption types. The tool will be able to identify BitLocker-encrypted volumes, VeraCrypt containers, and password-protected archives, flagging them for the investigator and preventing misleading, incomplete results.

A significant expansion of artifact parsing capabilities will further enhance the tool's forensic value. Parsers for key artifacts within the NTUSER.DAT and UsrClass.dat hives will be developed, including the implementation of robust analysis of Shellbags to reconstruct a history of folder navigation (including on network and removable drives) and parsing various Most Recently Used (MRU) lists to create a detailed timeline of user actions. The tool will also be enhanced to parse the SYSTEM hive to extract a comprehensive history of all USB devices connected to the system, including device serial numbers, vendor and product IDs, and connection timestamps, which are critical for tracing data exfiltration and linking digital events to physical evidence. Furthermore, parsers for the SOFTWARE and SYSTEM hives will be implemented to extract detailed system information such as operating system version and installed applications, as well as network history including past wireless networks and network profiles.

Finally, the development of advanced user interface and reporting features will complete the tool's transformation into a professional-grade forensic utility. The deferred graphical file browser will be developed, providing investigators with an intuitive interface to navigate the directory structure of the mounted evidence image. A comprehensive reporting module will also be created, allowing investigators to generate clear, professional, and customizable reports of all findings in standard formats such as HTML or PDF, suitable for case documentation and presentation in court. These enhancements will collectively establish the tool as a robust, self-contained forensic solution capable of meeting the rigorous demands of professional digital investigations.

# Methodology: An Integrated Approach to Forensic Tool Construction

This section details the methodological framework underpinning the design, development, and evaluation of the forensic triage tool. The project is fundamentally grounded in a Constructive Research paradigm, where the primary contribution is the creation of a novel, problem-solving artifact. This approach is particularly suited for applied computer science, where theoretical concepts are validated through the implementation of a functional system. To realize this, the methodology integrates principles from agile software development, a modular system architecture, and rigorous, low-level forensic data analysis techniques. The process is structured into distinct phases: a flexible development framework, a systematic process for evidence extraction, and a validation strategy against established forensic tools.

**An Agile Framework for Novel Forensic Tool Development**

The development of this tool adopted an agile, prototype-based model. This choice is a direct response to the nature of constructive research in a field as dynamic as digital forensics. Unlike traditional, rigid software development models like Waterfall, which rely on a set of sequential steps and require comprehensive upfront planning and documentation , agile methodologies embrace change and uncertainty. Research highlights that agile methods are characterized by "frequent reassessment and adaptation of plans" and the division of work into "shorter tasks for efficiency," known as iterations or sprints. This iterative process was essential for a project where the optimal methods for parsing and presenting novel combinations of forensic artifacts were not fully known at the outset. The agile model, with its focus on iterative and incremental development, allows for "quick deliveries" and continuous feedback, which in a research context, translates to the ability to rapidly test hypotheses about data correlation and presentation.

The selection of an agile methodology is not merely a matter of preference but a logical necessity driven by the project's classification as constructive research. The creation of a novel artifact inherently involves navigating unknowns and refining objectives as new information is discovered during development. Traditional software development models are predicated on a complete and stable set of requirements defined at the project's inception and do not easily accommodate change. Agile methodologies, in stark contrast, are explicitly designed to "provide flexibility to adopt the changes in requirements at any stage" and thrive where "adaptability is paramount". Therefore, to successfully execute a constructive research project, a development process that is inherently flexible and adaptive is required. Agile is not just an option; it is the most appropriate and methodologically sound framework for this research paradigm.

Furthermore, the core tenet of the Agile Manifesto, valuing "Working software over comprehensive documentation" , aligns perfectly with the objectives of constructive research. The goal was to incrementally build a functional tool, with each iteration producing a testable component that added value. The project's development, which involved building and testing individual modules for partition checking, user enumeration, and profile analysis sequentially, is a practical application of this principle. This approach mitigates the risks associated with large, monolithic development cycles and is better suited for the "fast-changing business environments" and competitive pressures that also characterize the academic and research landscape. This focus on delivering working software in short cycles enabled the project to address challenges and refine its approach based on tangible results rather than abstract planning, leading to a more robust and effective final artifact.

**A Modular Architecture for Forensic Data Processing**

The forensic triage tool is architected as a modular, four-tier system: Data Source (the .E01 evidence image), Core Parsers (specialized Python libraries), Python Backend (a Flask-based API), and a web-based User Interface (Altheide & Carvey, 2011). This layered design promotes a clear separation of concerns, which is critical for both maintainability and forensic integrity. The backend, built with the lightweight Flask web framework, acts as the central orchestrator. It handles all application logic, receiving requests from the user interface and exposing the findings through a series of well-defined RESTful API endpoints. This is evidenced by the clear, structured JSON responses returned by the API for various requests, such as identifying applicable partitions via the /api/check\_partitions endpoint or retrieving detailed user data from /api/partition/1/get\_user\_data\_with\_rid/500.

A key design choice was the encapsulation of all core forensic processing logic within a dedicated api\_methods module directory. This isolates the complex, low-level data parsing from the application's routing and presentation logic. For instance, the main app.py controller receives a request for a user's F-value data and calls the appropriate function within the get\_user\_f\_value\_data\_with\_rid.py module. This handler, in turn, relies on the f\_value.py script for the actual binary parsing of the data structure. This modularity is a significant strength, allowing for future expansion—such as adding a parser for the NTUSER.DAT hive to analyze shellbags or MRU lists—with minimal impact on the existing codebase, a key non-functional requirement.

This system's layered architecture is not just a good software engineering practice; it is a deliberate technical implementation of the fundamental forensic principle of evidence preservation (Altheide & Carvey, 2011). It creates a series of safeguards that prevent the accidental modification of the original evidence. A core non-functional requirement is "Forensic Integrity," stating the application "must never write any data back to the loaded evidence file" (Altheide & Carvey, 2011). The lowest layer of the architecture interacts with the evidence file using the dfvfs library, which is specifically designed to provide a forensically sound, read-only interface to the data. The architectural design places this dfvfs interaction within the "Core Parsers" layer, which is only accessible via the "Python Backend" layer. The "User Interface" layer, where the human investigator interacts, can only communicate with the "Python Backend" via these well-defined API calls. It has no direct access to the parsers or the evidence file itself. This creates a one-way flow of information (read-only) and establishes the API as a "gatekeeper." This design makes it architecturally impossible for a user action in the UI to trigger a write operation on the .E01 image, thereby programmatically enforcing forensic soundness.

**Systematic Artifact Extraction and Analysis from Evidence Images**

This section details the core technical workflow of the tool, tracing the path from mounting the raw evidence image to the presentation of high-level, analyzed artifacts. Each step is implemented by a dedicated module that leverages specialized libraries to ensure both accuracy and forensic integrity. The process was tested and validated using the mantooth.E01 disk image, a publicly available dataset used in digital forensics training, providing a realistic and standardized environment.

**Forensically Sound Evidence Mounting and System Identification**

The initial and most critical step in any forensic analysis is accessing the evidence without altering it. The tool accomplishes this by using the Digital Forensics Virtual File System (dfvfs) library to mount the .E01 evidence image. This library is the cornerstone of the tool's adherence to forensic principles, providing a read-only abstraction layer over the underlying file systems contained within the image. This ensures that the original evidence is preserved, a fundamental requirement for any digital investigation (Altheide & Carvey, 2011).

Following a successful mount, the tool must identify which of the available partitions contains a Windows installation. This is a crucial triage step that focuses the subsequent, more intensive analysis on relevant data. The check\_partitions.py script implements a heuristic for this purpose. As shown in the code, it iterates through potential partitions (e.g., /p1, /p2) and checks for the existence of a known, critical Windows directory, such as Documents and Settings or /Windows/System32/config (Altheide & Carvey, 2011). The output of this process is a list of viable partitions, which is returned to the user interface via the /api/check\_partitions endpoint as a JSON object containing the partition ID and a list of root-level files and directories. This allows the investigator to make an informed choice, directly addressing the UI flow where the user is presented with a list of partitions after uploading the evidence file.

**Dissecting the Security Account Manager (SAM) Hive for User Profiling**

The Windows Registry is widely regarded as a "gold mine of forensic evidences" and a "primary source of evidence", and the Security Account Manager (SAM) hive is a particularly rich seam. Located at %SystemRoot%\System32\config\SAM, this file is a database that stores critical user account information, including usernames and security details. Analyzing the SAM is fundamental to understanding user presence and activity on a system, as it contains configuration settings for all users and groups (Altheide & Carvey, 2011).

The tool's analysis of the SAM begins with user enumeration. The get\_usernames\_and\_rids.py module leverages the python-registry library to parse the mounted SAM hive. Specifically, it navigates to the SAM\Domains\Account\Users\Names key. Within this key, each subkey corresponds to a local username, and the type of its default value contains the account's Relative ID (RID) , a unique identifier for the local account. This process generates the list of users and their RIDs, which is then served to the frontend via the /api/partition/<partition\_id>/get\_usernames\_and\_rids API endpoint, as demonstrated by the sample response showing accounts like "Wes Mantooth" with RID 1000.

With a user's RID, a much deeper analysis is possible. User account metadata is stored in two primary binary data blobs, known as the F and V values , within the SAM\Domains\Account\Users{RID} key. These values are not human-readable and must be parsed according to their complex binary structures. The tool's ability to perform low-level binary parsing of these values is a critical differentiator that validates its status as a novel constructive research artifact. It demonstrates a mastery of the underlying data structures that goes beyond simply using a high-level, black-box library. While academic literature repeatedly describes the Windows Registry as a "gold mine" (Altheide & Carvey, 2011), this "gold" is not readily accessible. The data within the SAM hive's F and V values is stored in a complex, non-obvious binary format. The project's source code shows the use of Python's struct module to unpack from specific memory locations within the binary data blob, a manual, programmatic dissection of the artifact. This approach proves that the tool is not merely a wrapper around another pre-existing, high-level parser. It demonstrates a fundamental implementation of the logic required to "excavate" the data, elevating the project from a simple application to a piece of constructive research.

The f\_value.py and v\_value.py scripts use Python's struct module to unpack specific data fields at predefined offsets, which are managed externally in f\_value\_offsets.csv and v\_value\_header\_offsets.csv for clarity and maintainability. This process extracts forensically significant artifacts, as detailed in Table 3.1.

**Table 3.1: Key Forensic Artifacts Extracted from SAM Hive F and V Values**

|  |  |  |  |
| --- | --- | --- | --- |
| Artifact | Source (Registry Value) | Offset (in Value) | Forensic Significance |
| Last Logon Timestamp | F-Value | 0x08 | Establishes the last time the user successfully authenticated to the system, crucial for timeline reconstruction. |
| Password Last Set | F-Value | 0x18 | Indicates when the user's password was last changed, which can correlate with security events or user actions. |
| Relative ID (RID) | F-Value | 0x30 | The unique identifier for the local account, used to link the account to other system artifacts and security events. |
| User Account Control | F-Value | 0x38 | A bitmask detailing the account's status and privileges (e.g., disabled, locked out, password not required). |
| Login Count | F-Value | 0x42 | A counter of successful logons, which can help establish patterns of use or identify infrequently used accounts. |
| Username | V-Value | 0x0C (Pointer) | The human-readable account name, providing direct attribution. |
| Full Name | V-Value | 0x18 (Pointer) | The full name associated with the account, providing additional user context. |

**Decoding User Account Control (UAC) Flags for Security Posture Analysis**

Within the F-value, at offset 0x38, is a 4-byte integer representing the User Account Control (UAC) flags. This value is a bitmask, where each bit corresponds to a specific property or status of the user account. Decoding this value provides immediate insight into the account's security posture and history, which is of high value for forensic analysis.

The f\_value\_flags.py module is responsible for this decoding. After extracting the 4-byte integer using struct.unpack\_from, the script performs a series of bitwise AND operations. It compares the UAC value against a predefined list of flags and their decimal values, which are stored externally in uac\_flags.csv for maintainability. If the bitwise operation (uac\_flag\_sum\_decimal & flag\_decimal\_value) == flag\_decimal\_value evaluates to true, the corresponding flag is considered active. This process translates a single number, such as 529, into a human-readable list of properties like "SCRIPT," "LOCKOUT," and "NORMAL\_ACCOUNT," as demonstrated in the /api/partition/1/get\_user\_data\_with\_rid/500 API response for the Administrator account. From a forensic perspective, these flags are highly significant. For instance, an active LOCKOUT flag indicates that the account is currently locked out, possibly due to recent failed login attempts that could suggest a brute-force attack. An ACCOUNTDISABLE flag shows the account is not in use, while a PASSWD\_NOTREQD flag reveals a critical security vulnerability that an attacker could exploit.

**Automated Email Collection and Thematic Analysis via Generative AI**

To provide a rapid overview of user communications, the tool incorporates a novel feature for email analysis and summarization. The process begins with the collect\_user\_emails.py script, which performs a recursive search within a targeted user profile directory to locate all files with a .eml extension. For this project, the target path was hardcoded to /Users/{username}/AppData/Local/Microsoft/Windows Mail/, a common location for Windows Mail artifacts. The script uses dfvfs to navigate the directory structure within the .E01 image and compiles a list of full paths to every discovered email file.

Once this list is compiled, the email\_ai\_analysis.py module orchestrates the analysis. For each .eml file path, it again uses dfvfs to read the raw file content from the .E01 image. Python's built-in email library is then used to parse the byte stream, extracting headers (From, To, Subject, Date) and the text body. This parsed information from all emails is aggregated into a single large text block. This aggregated text is then sent to the Google Gemini API via the google-generativeai library. A carefully engineered prompt instructs the model to act as a "forensic analyst assistant" and summarize the communications, focusing on key correspondents, main topics, and "Potentially Suspicious Activity". The resulting summary, as shown in the /api/partition/1/email\_ai\_analysis/1000 API response, provides a high-level thematic overview that can drastically reduce the time an investigator would otherwise spend manually reading each email.

The integration of a cloud-based AI for summarization represents a significant innovation in forensic triage but simultaneously creates a direct conflict with established forensic doctrine. The project aims to be innovative, and using a Large Language Model (LLM) for summarization is a modern approach to accelerating analysis. However, the implementation uses the Google Gemini API, which involves sending evidence-derived data over the network to an external, third-party service. The project's own non-functional requirements explicitly state: "No evidence data should be transmitted over a network to external services" to ensure security and privacy. This creates a direct contradiction. In the context of a Proof-of-Concept, this is a pragmatic trade-off to demonstrate the capability of AI summarization without the significant hardware and configuration overhead of deploying a local LLM. This highlights a fundamental tension at the intersection of modern AI and traditional digital forensics. The power of cloud-based AI models is immense, but their use in a live investigation would violate the chain of custody and data security principles (Altheide & Carvey, 2011). This identifies a critical area for future research and development: the integration of powerful, locally-run models to harness AI's benefits without compromising forensic integrity.

**Evaluation and Validation Against Established Methods**

The final phase of the constructive research methodology is the evaluation of the created artifact. The success of this tool was measured by its accuracy and completeness when compared against a ground truth. This ground truth was established through a manual examination of the same E01 disk image using Autopsy, a comprehensive, open-source digital forensic suite widely used in the field (Altheide & Carvey, 2011). The comparison focused on verifying the correctness of extracted data points, such as the list of enumerated users, the parsed timestamps and login counts from the SAM hive, and the decoded UAC flags.

This process of validation is critical. The field of digital forensics can "no longer tolerate software that cannot be relied upon" and requires clearly defined, standardized results to avoid misinterpretation by practitioners. By benchmarking the tool's output against a well-vetted, industry-standard tool like Autopsy, the evaluation confirms the reliability and accuracy of the custom-built parsers, thereby validating the success of the constructive research effort. This aligns with academic calls for rigorous analysis and testing of forensic tools to ensure their findings are defensible and that they can be used to produce evidence that is admissible in legal proceedings (Altheide & Carvey, 2011). The positive comparison against the ground truth confirms that the methodologies employed for artifact extraction and analysis are sound and produce results consistent with established forensic practices.

# Literature review

**Introduction**

Digital forensics is the use of scientifically derived and proven methods for the preservation, collection, validation, analysis, and presentation of digital evidence (Altheide & Carvey, 2011). The primary goal of any forensic examination is to uncover facts and reconstruct the truth of an event by discovering and exposing the remnants, or artifacts, left on a system (Altheide & Carvey, 2011). However, modern investigators are faced with a significant challenge: the sheer volume of data on contemporary computer systems. This data overload necessitates a structured approach to prioritize evidence and gain rapid insights. This initial phase of an investigation is known as forensic triage, a process designed to quickly identify and assess high-value artifacts to determine the direction of a deeper examination. This project is situated within the context of forensic triage, aiming to develop a tool that automates and accelerates this critical initial assessment.

**The Forensic Tooling Landscape**

The field of digital forensics has historically been dominated by commercial, proprietary tools that often operate as a "black box," providing a layer of abstraction between the analyst and the raw data (Altheide & Carvey, 2011). While powerful, these commercial suites can be slow and complex, making them less suitable for rapid triage scenarios. In contrast, the open-source community has produced a range of powerful and transparent tools (Altheide & Carvey, 2011). The premier open-source framework for file system analysis is The Sleuth Kit, which provides a suite of command-line utilities for deep inspection of disk images and file systems (Altheide & Carvey, 2011). While these tools offer unparalleled access to the data, they often require significant expertise and manual effort to operate, catering to examiners comfortable with the command line. This creates a dichotomy in the existing toolset: large, monolithic graphical applications that can be slow, and highly specialized, powerful command-line tools that may lack integration and user-friendliness for all investigators.

**The Windows Registry as a Primary Source of Evidence**

Among the most valuable sources of forensic artifacts on a Windows system is the Windows Registry (Altheide & Carvey, 2011). The Registry is a central hierarchical database that stores essential configuration information for the system, applications, and users (Carvey, 2016). It is a core component of the operating system and maintains a significant amount of historical information about user activity, making it a vital resource for any forensic investigation (Carvey, 2016). The Registry is not a single file but is composed of several "hive" files located on disk (Carvey, 2016). For a forensic analyst, the most critical of these are the system hives (SAM, SECURITY, SOFTWARE, SYSTEM) and the user-specific hives (NTUSER.DAT, USRCLASS.DAT) (Altheide & Carvey, 2011).

The Security Accounts Manager (SAM) hive is of particular importance as it maintains information about local user accounts on the system (Carvey, 2016). Each user account is represented by a key corresponding to their Relative ID (RID), which contains binary F and V values (Carvey, 2016). These values store a wealth of data, including the username, account creation date, last login date, login count, and User Account Control (UAC) flags that define the account's status, such as whether it is disabled (Carvey, 2016).

The user-specific hives, NTUSER.DAT and USRCLASS.DAT, provide a treasure trove of data regarding a user's direct interactions with the system (Carvey, 2016). One of the most significant artifacts is the UserAssist key, which tracks applications launched by the user through the Windows Explorer shell (Carvey, 2016). Analysis of this key can reveal a history of program execution, including the last execution time and a run count for each application (Carvey, 2016). Another key artifact, shellbags, records a user's access to folders through the Windows shell, which can be used to reconstruct a user's navigation history across local, network, and removable drives (Carvey, 2016). Specialized tools, such as the open-source RegRipper framework, have been developed to automate the extraction of these and other artifacts from registry hives, demonstrating the value of targeted parsing (Carvey, 2016).

**Identifying the Research Gap**

The literature clearly establishes the immense value of Windows Registry artifacts and the existence of powerful open-source parsing tools. However, a gap exists between the capabilities of these specialized, often command-line-driven tools and the needs of an investigator performing rapid triage. While comprehensive suites like Autopsy provide a graphical interface, they are designed for deep-dive analysis, and specialized scripts like RegRipper lack an integrated, user-friendly dashboard for correlating findings. There is a need for a tool that bridges this gap, providing automated extraction and parsing of high-value registry artifacts and presenting them in an immediate, intuitive, and correlated manner.

Furthermore, while the concept of automating analysis is well-established (Altheide & Carvey, 2011), the application of modern Artificial Intelligence, specifically Large Language Models (LLMs), to summarize unstructured forensic data like emails for triage purposes is a novel extension of this principle. The existing literature focuses on parsing structured data, leaving a gap for tools that can provide high-level intelligence from unstructured communications to guide an investigation. This project aims to fill this gap by developing a standalone triage tool that not only automates the extraction and presentation of key user profile and registry artifacts but also integrates AI-powered analysis to deliver a rapid, intelligent summary of user activity.

# Findings

Findings

# Analysis/Discussion

Analysis/Discussion

# Conclusions

Conclusions

# Recommendations

Recommendations

# Bibliography

Altheide, C., & Carvey, H. (2011). *Digital Forensics with Open Source Tools.*

Carvey, H. (2016). *Windows Registry Forensics.*

Mohammed, K. (2025). Digital Data Acquisition, Recovery and Analysis.

Sharma, B., Baggili, I., Ghawaly, J., McCleary, K., & Webb, A. M. (2025). *ForensicLLM: A local large language model for digital forensics.* Retrieved from https://dfrws.org/wp-content/uploads/2025/03/ForensicLLM-A-local-large-language-mod\_2025\_Forensic-Science-International-.pdf

Valli, D. (2008). *SIMPLE - Rethinking the monolithic approach to digital forensics.* Retrieved from Research Gate: https://www.researchgate.net/publication/235754067\_SIMPLE\_-\_Rethinking\_the\_monolithic\_approach\_to\_digital\_forensics

Zhu, Y. (2009). *Using shellbag information to reconstruct user activities.* Retrieved from https://www.researchgate.net/publication/242822677\_Using\_shellbag\_information\_to\_reconstruct\_user\_activities

# Appendices

## A Glossary of Terms

|  |  |
| --- | --- |
| Term | Definition |
| API (Application Programming Interface) | A set of rules and specifications for software communication, relevant for analyzing cloud-IoT interactions. |
| Volatile Data | Data in temporary memory (RAM) lost on power loss, necessitating rapid capture for forensic analysis. |