COSC 499 – Edited Project Proposal

Mining Digital Work Artifacts

Team Number: 06
Team Members: Aakash Tirathdas 19000413; Mithish Ravisankar Geetha 54389754; Ansh Rastogi 17164336; Harjot Sahota 97986475; Mohamed Sakr 14163851; Mandira Samarasekara 34542282
Edited on: October 12, 2025

This edited version aligns the proposal with Milestone #1 requirements (Oct–Dec 07) and clarifies scope, architecture, testing, consent/privacy, and optional LLM/RAG usage. All outputs for this milestone are text-first (JSON/CSV/plain text) with a local SQLite store; the system is built in Python.

1. Project Overview & Goal

This project mines individual work artifacts from a user's computer to identify the digital outputs created in the course of professional or creative activities. Artifacts include programming code and repositories, written documents and notes, or design sketches and media files. By analyzing artifacts and their metadata, the system uncovers meaningful insights into a user's work contributions, creative processes, and project evolution. The outputs support portfolio creation, résumé enhancement, and self-reflection, while explicitly addressing consent, data privacy, and ethical analysis of personal work history.

2. Target Users

A graduating student or an early professional who works on the computer regular and is wanting to gather information about the projects they worked on over the years. They hope to use this information to showcase as part of a web portfolio, or a dashaboard of some kind that lets them easily see the metrics and highlights of their hard work, or descriptions and metrics that could be useful for improving their résumeé.

3. Milestone #1 Scope & Deliverables (October – December 07)

The milestone focuses on parsing and correct output of information, strong system design, and rigorous testing. All outputs are text-first (CSV, JSON, or plain text) with a local SQLite database. The implementation language is Python. A command-line interface (CLI) is sufficient for this milestone; API and front-end will follow in Term 2.

4. Functional Requirements – Design & Verification

Requirement	Design/Verification
Consent gating	Require explicit, logged consent before reading any data. Store consent decisions with timestamps and scope.
ZIP parsing	Accept a specified zipped folder; recursively traverse nested folders/files; stream-safe unzip; validate structure.
Wrong format errors	Reject non-ZIP inputs with precise error codes and messages; add negative tests.
External services permission	Before any external call (e.g., LLM), present purpose, data categories, and privacy implications; proceed only on allow.
No-external alternative	Provide full local analysis path producing equivalent textual outputs without network calls.
Config persistence	Persist user configurations (e.g., exclusions, ranking weights) in a local config and DB table.
Individual vs collaborative detection	If Git history exists, infer collaborators from commits; otherwise mark as 'unknown/assisted' with confidence.
Language & framework ID	Detect languages via extensions and heuristics; frameworks via manifest/build files (e.g., requirements.txt, package.json).
Individual contribution extrapolation	With VCS, compute per-author LOC added/removed, files owned, commit density; otherwise approximate via file ownership and metadata.

Contribution metrics & duration	Compute project start/end from timestamps; classify activity mix (code/test/docs/design); include size and complexity proxies.
Skill extraction	Deterministic keyword/taxonomy mapping from code/doc artifacts; optional LLM normalization if consented.
Project key info output	Emit per-project JSON summary plus CSV tables for artifacts, contributors, skills, metrics, evidence pointers.
Database storage	Store normalized entities in SQLite with foreign keys and indexes for retrieval.
Retrieve portfolio info	Query top facts/skills/summaries for portfolio use; export as JSON/CSV text blocks.
Retrieve résumé item	Generate or retrieve succinct résumé-ready bullets per project; store variants.
Project ranking	Transparent score combining recency, duration, contribution share, impact signals, and skill breadth (weights configurable).
Summarize top projects	Template-based summaries by default; optional RAG-backed LLM summaries with consent.
Safe deletion	Logical delete insights and decrement file reference counts so shared files are preserved; emit a deletion receipt.
Chronological projects list	Sort projects by start/end; export as CSV/JSON.
Chronological skills list	Track first/last observed usage per skill; export a skill timeline.
Python implementation	Python 3.11+, CLI-driven for Milestone #1; testing with pytest; DB is SQLite.

5. System Architecture (Local-First, Modular)

1. CLI Shell: subcommands for ingest/analyze/rank/export/retrieve/purge; structured JSON logs; graceful errors.
2. Consent Manager: explicit consent per scope (ZIP read, external calls); timestamped ledger.
3. Ingestion (ZIP): validate archive, stream unzip to workspace, compute SHA-256, capture mtime/ctime, MIME.
4. Project Grouper: identify repo roots (e.g., .git/), manifests (pyproject.toml, package.json), cluster artifacts into projects.
5. Analyzers (plugins): Code/Docs/Media analyzers return normalized facts; handle OS permissions; evidence pointers.
6. Metrics & Skills: compute duration, activity mix, complexity proxies; map keywords to skill taxonomy; store confidence.
7. Ranker & Summarizer: transparent scoring; template-based summaries by default; optional LLM enhancement.
8. Persistence: SQLite schema with foreign keys (projects, files, artifacts, contributors, skills, insights, configs, consent, refcounts).
9. Exporter/Retriever: CSV/JSON outputs; retrieval by project/date/skill for portfolio and résumé uses.
10. Deletion: logical deletes with reference counting; rebuild indexes; emit a deletion receipt.

6. Data Model (SQLite, Simplified)

- projects(project_id PK, name, type, start_ts, end_ts, is_collab, rank_score, created_at)
- files(file_id PK, sha256, path, mime, size, mtime, first_seen_project_id)
- artifacts(artifact_id PK, project_id FK, file_id FK, activity_type, language, framework, complexity, evidence)
- contributors(project_id FK, person, email, share, commits, added_loc)
- skills(project_id FK, term, source, confidence, first_seen, last_seen)
- insights(project_id FK, kind, text, created_at, source)
- portfolio_items(project_id FK, variant, text, created_at)
- resume_bullets(project_id FK, text, created_at)
- consent(scope, decision, text, timestamp)
- configs(profile, key, value)
- refcounts(file_id FK, count)

7. Key Use Cases (Edited)

Based on the usage scenario, describe all the use cases in detail. Do this by providing a UML use case diagram. Then for each use case in your diagram, identify: the name of the use case, the primary actor, a general description, the precondition, the postcondition, the main scenario, and possible extensions to consider.

Use Case 1: Dashboard Generator

- **Primary actor:** Student/ Professor
- **Description:** The program will generate a dashboard which will contain information of the projects present in a user's laptop.
- **Precondition:** The user must have given access to the application to go through a folder or their laptop.
- **Postcondition:** The dashboard is displayed on the desktop application such that the user can interact with it.
- **Main Scenario:**

    1. The user clicks on the “Generate dashboard” button.
    2. System validates that the requirements are met
    3. If requirements met, system begins dashboard generation
    4. System displays the dashboard on the application
- **Extensions:**
    1. Invalid data is entered in Step 2. System notifies the user that the requirements were not met and guides the user to submit the correct requirements

Use Case 2: Upload folder

- **Primary actor:** Student/ Professor
- **Description:** The system accepts a zipped folder such that the scanning process can take place. (folder name if no database)
- **Precondition:** The user must have selected the folder upload mode (can be called folder scan mode if no database)
- **Postcondition:** The zipped folder is stored in database (folder name is stored in memory)
- **Main Scenario:**
    1. User selected the upload folder option
    2. User drags and drops a zipped folder
    3. System validates if the folder meets the requirements
    4. If folder passes validation the system gives a confirmation message
    5. Folder is uploaded to data base

Extensions:

- Invalid folder format in step 3 lead to an error thrown and the user having to reupload or upload a different folder to continue.

Use Case 3: Scanning and Analyzing Files

- **Primary actor:** User (final year student / professional)

- **Description:**

  Through the desktop app, the user selects the required folders and drivers for a local scan, and the app extracts the data in the files, generates summaries and releases results in a local database. The system also groups related files into one, and respects OS permissions. The scan is logged and provides any prompts if needed to the user.

- **Precondition:**
  a. The desktop app is installed locally.
  b. The user is logged in to the system (Discuss whether we have authentication or not as it's a local app).
  c. The user selects folders to scan.
  d. The app has the required permissions to create files and to write logs.
  e. There must be enough memory available to run the scan.
  f. If a scan is already running, the system must either wait for the first scan to finish or should prevent a new one to start.
- **Postcondition:**
  a. There are extracted content summaries and logs saved in the required folder/drive of the system.
  b. All stored data remains local and encrypted.
  c. The metadata for scanned documents and artifacts are stored in the local database.
  d. A scan session entry is recorded as metadata in the database.
- **Main Scenario:**
  1. The user opens the desktop app and clicks the Scan button.
  2. The system prompts for a dialog where the user can select folders and drives and potentially skip a few files/folders.
  3. Prior to scanning the files, the system must check if any of the file paths the user provided requires any elevated permissions. If so, the system prompts for a consent dialog which tells the user what might be read, and asks the user for elevated access grant, skip the folder or cancel the scan.
  4. If the user grants elevated access, the scan proceeds for those paths.
  5. The system creates a new scan session and stores it in the local database, with attributes being sessionID, userID, time, paths.
  6. The app checks if the system can handle resource constraints.
  7. The system reads through each file, checks permissions, and extracts metadata and content.
     a. Code repositories: Detect programming language based on the extension, count lines of codes, detect functions and classes, map each line to the user contribution, generate required summary. Use python libraries accordingly.

    b. Document files: Detect the author, author, and section headings. Extract the raw text from the pages and store the content snippets accordingly.
8. The system will store extracted summaries and content snippets in the database.
9. The system builds summary statistics by counting the number of files per file, timeline of projects etc.
10. After the scan is complete, the initial results are sent into the analytics function, and the system notifies that the scan is completed and provides the links to the dashboard.

- **Extensions:**
    1. If the system cannot access a file due to denied permission, it skips the file.
    2. If the system memory use exceeds thresholds, the scan pauses or terminates, saving partial results.
    3. If the user cancels mid scan, the system immediately halts scanning, stores partial results and logs it as incomplete.
    4. If the user initiates another scan, the system saves the metadata only and marks the extraction as partial.
    5. If results cannot be written to the database, the system logs the failure and notifies the user.

Use Case 4: Laptop scanner mode

- **Primary actor:** Student/ Professor
- **Description:** The program will go through your laptop and scan for relevant projects. Will avoid certain folders that the user wants to keep private
- **Precondition:**
    a. The user must have selected the laptop scanner mode
    b. The laptop must be either Windows or Mac (NO LINUX)
- **Postcondition:** The application has a layout of the laptop to scan and scanning begins. This layout does not include files that the application should avoid
- **Main Scenario:**

    1. The user selects scan laptop
    2. User fills in form on folders and files the scan should avoid
    3. Application validates that the form is filled correctly and the laptop can be scanned (check if operating system is compatible)
    4. If validation passes, the application creates a layout of folders and files that should be scanned. Folders that should not be scanned and operating system files will be removed from this layout.
       Confirmation page on the scan is presented with information on what will be scanned

- **Extensions:**

    1. Invalid folders or files (names of folders and files) is entered in 2, the system throws an error and asks users to double check the folders that should not be scanned
    2. A system with an invalid operating system runs the application. Throw an error warning the user that this application was not made for this system and only the upload folder system is compatible with the OS.

Use Case 5: Mode selection

- **Primary actor:** Student/ Professor
- **Description:** The program will prompt the user to pick either folder upload mode or laptop scanner mode. There will be a description on how each mode will work and what to expect
  Precondition:
    1. The user must have downloaded the application successfully.
    2. There should not be another scan that is currently underway
- **Postcondition:** The application moves to the next step where the user is required to upload or give permission for folder(s) to be scanned
  Main Scenario:
    1. Open application.
    2. System validates that the application has been downloaded correctly.
    3. The system brings the user to a page with the option to select one of 2 modes.
    4. Upon selection the system validates if the system is not currently running a scan.
    5. If validation passes, the system moves the user onto the next stage of the process so that the details of the scan can be filled

- **Extensions:**

    1. If a scan is already underway, the system throws a message and waits till the ongoing scan is complete before proceeding with the system flow.

Use Case 6: Secure Data Deletion

Primary actor: Student / Professor (with optional Admin controls)

Description:
The user permanently removes previously collected scan results (artifacts, summaries, metadata, and logs) from the local machine. The operation enforces privacy by securely erasing encrypted data at rest, clearing indexes, and removing cached analytics so nothing remains recoverable through the app.

Precondition:

- The desktop app is installed and opens successfully
- At least one prior scan has produced stored artifacts/metadata locally (encrypted)
- No scan or analytics job is currently running.
- User is authenticated to the local profile; Admin role required for bulk/multi-user deletions.

Postcondition:

- Selected datasets (e.g., a scan session, project group, or all data) are irreversibly deleted from the local database, cache, and export directories.
- Application indexes and dashboards are recompiled or marked empty.
- An auditable deletion receipt (timestamp, scope, actor) is written to a minimal local log that contains no user content.

Main Scenario (brief):

1. Open **Privacy & Data Controls → Delete Data**.
2. Choose deletion scope (session/project group / all).
3. System verifies no active jobs and required permissions.
4. Confirm via dialog (e.g., type “DELETE”).
5. System deletes targeted data and derived assets; rebuilds indexes.
6. Show **Deletion Receipt** with scope, timestamp, and space reclaimed.

Extensions (brief):

- **Active job:** Block and prompt to cancel or retry later.
- **Insufficient permissions:** Deny and explain Admin requirement.
- **Partial failure:** Roll back/mark inconsistent items; offer retry/report.
- **External exports detected:** List locations; advise manual removal.
- **File lock/low space:** Identify blocker and provide fix steps; retry option.
- **Compliance mode:** Dual confirmation (Admin + User) required for “All Data”; log acknowledgments.

8. Testing & Verification Plan

11. Unit tests per analyzer using fixed fixtures (expected JSON/CSV).
12. Property-based tests for ZIP traversal and ID stability; same inputs → same outputs.
13. Golden tests: sample datasets with frozen expected outputs in repo.
14. Negative tests: wrong file type; denied permissions; out-of-memory simulations.
15. Privacy tests: external calls are blocked unless consent flag is true; PII redaction before any optional prompts.
16. Performance smoke: medium ZIP completes within target time budget on reference machine.

9. Privacy, Consent, and Ethical Considerations

The system is local-first. External services (e.g., LLMs) are disabled by default and require explicit, scoped consent explaining data categories and risks. Analysis follows data minimization (only the least necessary content is processed), and evidence pointers are stored for auditability. Sensitive personal content encountered during scans is not exported by default. A secure deletion flow removes derived data and indexes and provides an auditable receipt.

10. Platforms and Technology (Milestone #1)

- Operating Systems: Windows and macOS (Linux optional post-M1).
- Language: Python 3.11+.
- Storage: SQLite.
- Interfaces: CLI for ingest/analyze/rank/export/retrieve/purge.
- Libraries (indicative): zipfile/pyzipper, python-magic, chardet, GitPython/PyDriller, radon, PyPDF2, python-docx, pandas, scikit-learn (for TF-IDF), pytest, hypothesis.

11. Optional LLM/RAG Integration (Consent-Gated)

If consent is granted, a Retrieval-Augmented Generation (RAG) path can summarize projects and normalize skills without uploading full files. The system builds a local vector index (e.g., FAISS) over READMEs, commit messages, and extracted text snippets. The summarizer LLM—preferably local (e.g., via llama.cpp/Ollama) or a privacy-vetted API—receives only retrieved snippets and structured metrics. This improves narrative quality while grounding outputs in real artifacts. The default mode remains deterministic and offline to ensure reproducibility and privacy.