iExtract iOS Album to Folder Conversion Tool

Basic Team Information:

- **(Names: Roles):**
  - Sam Daughtry: Developer
  - Noah Gregie: UI/UX
  - Kevin Gustafson: Director
  - Brendon Wong: QA Tester
- **GitHub Repo Link:** https://github.com/Gustakev/cs362-class-project
- **GitHub AI Image Description Feature Repo:** https://github.com/BrendxnW/image-captioning-cnn-lstm
- **Communication Rules:**
  - We will communicate primarily through a Microsoft Teams group chat and our OSU associated emails. We may establish other means of communication amongst ourselves, but that is optional.
  - In terms of timeliness, as long as someone responds to a message sent via Teams or email within a few hours, it will be considered acceptable. However, there may be some times when quicker communication is expected and/or necessary to reach deadlines on time. During those times, it is preferable that team members respond to messages from other members within at least an hour, so that we may accomplish our goals on time.

Project Milestone-3: Project Architecture and Design:

- **Software Architecture:** Layered Architecture
- **Major Components:**
  - **CLI:** The CLI, as part of the Presentation Layer, will accept user input, send that input to the Application Layer in a predictable manner, and the Application Layer will call (or import data from) one of the lower layers, the Domain Layer or the Data Layer, to make a request given the data it received from the CLI.
  - **Backup Locator & Validator:** This will work by having a certain backup path handed down from the Presentation Layer to the Application Layer, which will first determine whether it could even

be a real file path to begin with, but if so, will then proceed to sending it to the Data Layer, along with the correct SQL to execute, and then the Data Layer will open the correct database and execute the correct SQL using SQLite, and then it will return the SQLite response to the Application Layer, which will validate the response and create a persistent data structure representing the information from the databases, and then that will be returned to the Presentation Layer to render.
    - **SQLite Query Component (Manifest.db + Photos.sqlite):** This is the part of the Data Layer that executes the SQL query specified by the Application Layer on the specified file and returns the response data back to the Application Layer.
    - **File Extraction Engine:** This component is responsible for reconstructing an album/collection into a folder of media items. It simply receives an altered version of the object representing the backup, from the Application Layer, based on user choices in the Presentation Layer, along with another object from the Application Layer specifying what to create folders for, and then it creates the correct extraction based on the states of those objects.
    - **Conversion Engine (HEIC to PNG, MOV to MP4, Live Photo to MP4):** This component is responsible for interrupting the file extraction process whenever the file extraction engine attempts to copy a file that is marked as needing to be converted first, taking control of the process for a moment, converting the file, and then allowing the process to continue.
- **Interface(s) Between Components:**
    - **CLI to Backup Locator & Validator:**
        - The CLI passes a parsed Command object containing a file path (if relevant, like a backup path), requested action, and user choices. The Backup Locator & Validator returns a CommandResult with status and messages, which are then interpreted by the CLI and shown to the user.
    - **Backup Locator & Validator to SQLite Query Component:**
        - The Backup Locator & Validator calls the SQLite Query Component with a database path, SQL query string, and parameters. The SQLite Query Component returns a list of raw

SQL rows to be interpreted and used to construct the JSON object representing the backup info.

- **Backup Locator & Validator to File Extraction Engine:**
  - After constructing the Backup Model, the Backup Locator & Validator passes a BackupModel object and an ExtractionPlan to the File Extraction Engine.
- **File Extraction Engine to Conversion Engine:**
  - When a media item requires conversion, the File Extraction Engine calls the Conversion Engine with an input path, output path, and target format. The Conversion Engine returns the converted file path.
- **File Extraction Engine to CLI:**
  - The File Extraction Engine returns a Result object summarizing the extraction outcome. It also returns live information about the progress that has been made on the extraction operation, which the CLI will use to update a loading bar that works based on boolean checkpoints throughout the execution of the program, as well as Error objects for the CLI to display under the loading bar in a readable format, if relevant.

- **Data Storage Explanation (High Level Schema or Organization):**
  - Our system will not use a database. Instead, it will extract info. from multiple databases and use it to construct a single master JSON object that represents the backup's structure. This object will not be turned into a file. It will stay in memory throughout the duration of the program's runtime.
- **Assumptions Underpinning Architecture Choice:**
  - **Stability of iOS Backup Format:** We make the assumption that Photos.sqlite and Manifest.db adhere to a stable schema that can be consistently queried during development.
  - **Cross-Platform SQLite Support:** We assume that SQLite operates uniformly on Linux, macOS, and Windows, allowing for a single Data Layer.
  - **CLI-Only Presentation:** Since we anticipate that the system will only be used via a command-line interface, a straightforward Presentation Layer free from GUI issues is possible.

    - **In-Memory Backup Model:** We are making the assumption that the user's photo library databases are sufficiently small to be fully rebuilt in the Application Layer's memory.
    - **Availability of the File System Tools:** We assume that the host system has the required file operations necessary to run the program.
    - **User Permissions:** We assume users have read access to the backup directory and write access to the destination directory.
    - **I/O-Heavy Workflow:** We assume the system's complexity lies primarily in I/O and workflow orchestration, making layered architecture an appropriate choice.
- **First Decision Pertaining to Architecture Choice (Layered Architecture Choice):**
    - **Identify And Describe an Alternative:**
        - **Microkernel Design:** A single microkernel that facilitates interactions between program components, which would all be plugins, and implements only the most basic operations.
    - **Identify Pros Relative to Our Actual Choice:**
        - It would make it easier for us to integrate plugins later on without needing to create a more complex plugin pattern that aligns with the Layered Architecture.
        - It would make it easier for the team to work on things simultaneously.
    - **Identify Cons Relative to Our Actual Choice:**
        - It would be more difficult to design a microkernel that respects multiple ways of communication with different plugins than it would be to make a Presentation Layer that only supports communication with an Application Layer that does all the interpretation from the lower layers for it.
- **Second Decision Pertaining to Architecture Choice (In-memory Backup Data Structure):**
    - **Identify And Describe an Alternative:**
        - **Backup Data File:** A single file that represents the backup's contents, likely in JSON format.
    - **Identify Pros Relative to Our Actual Choice:**
        - It would make it so that we could recover from program crashes more easily, as long as the backup data file could be

found, validated as correct for the selected iPhone backup, and reloaded.
        - It would not use up RAM.
    - **Identify Cons Relative to Our Actual Choice:**
        - It would take up disk space.
        - It would be slower to create and read from.
        - It would be possible for another program (or the user) to ruin the backup data file during the runtime of the program.

Software Design (What packages, classes, or other units of abstraction form these components, and what are each component's responsibilities?):

- **Components:**
    - **CLI:**
        - **Units of Abstraction:**
            - **CommandParser:** Reads raw user input and extracts flags, paths, and requested actions.
            - **CommandValidator:** Ensures the parsed command is complete, well-formed, and refers to valid options.
            - **CommandDispatcher:** Sends validated commands to the Application Layer for execution.
    - **Backup Locator & Validator:**
        - **Units of Abstraction:**
            - **PathValidator:** Confirms the provided backup directory exists and contains required iOS backup files.
            - **BackupLocator:** Identifies Manifest.db and Photos.sqlite within the backup structure.
            - **SQLQueryPlanner:** Determines which SQL queries must be executed to retrieve album/collection metadata.
            - **BackupModelBuilder:** Constructs the in-memory BackupModel object from raw SQL results.
    - **SQLite Query Component:**
        - **Units of Abstraction:**
            - **SQLiteConnectionManager:** Opens and closes SQLite database files safely.
            - **SQLExecutor:** Executes SQL queries provided by the Application Layer and handles errors.

            - **RowMapper:** Converts raw SQLite rows into structured Python data for further processing.
    - **File Extraction Engine:**
        - **Units of Abstraction:**
            - **ExtractionPlanner:** Determines which albums, collections, and items must be extracted based on user choices.
            - **FolderBuilder:** Creates the destination folder structure that mirrors the album/collection hierarchy.
            - **FileCopier:** Copies media files into the correct folders or staging areas.
            - **SymlinkManager:** Creates symbolic links for items that appear in multiple albums to avoid duplication.
    - **Conversion Engine:**
        - **Units of Abstraction:**
            - **ImageConverter:** Converts HEIC images into JPEG or PNG at the user-specified quality.
            - **VideoConverter:** Converts MOV videos into MP4 format.
            - **LivePhotoConverter:** Merges the still image and video components of a Live Photo into a single MP4.

Coding Guidelines (Briefly state why you chose these guidelines and how you plan to enforce them.):

- **Standard Python PEP 8 (https://peps.python.org/pep-0008/):**
    - We will be using the standard PEP 8 Python Style Guide for our project. To ensure that it is enforced, we will combine the use of a linter and the Black auto formatter to reduce time spent on correcting formatting issues.
- **SQL: GitLab Style Guide (SQL Style Guide):**
    - We will be using the GitLab Handbook SQL Style Guide for our project, along with the SQLFluff linter/auto formatter to fix any inconsistencies and easily enforce conventions.

Product Description:

- **Abstract:**
    - iExtract is a tool that helps people intelligently extract their photos and videos from their iPhones via reconstructing a folder structure

and converting proprietary formats to common formats. This utility is dedicated to making iOS Photos app albums and collections easily convertible into folders en masse, which is not currently possible via any other tool than iMazing and a few other niche, monolithic tools. Additionally, iMazing has an expensive licensing model, as it provides many more features than just photo/video album to folder conversion, yet in a way that is not as focused.

- **Goal:**
    - The goal of this application is to solve the user experience issue regarding the either tedious, expensive, or iCloud-centric method of photo album and collection to local storage transfer that is available to iOS users at the moment. This app will facilitate a completely local transfer of albums and collections to local storage (converted into a folder-based format) on a user's PC, given they have created an unencrypted backup of their device, including their camera roll items.
- **Current Practice (Local Transfer Options):**
    - **Windows Transfer Tools:** Use the Windows Photos app (or older importer via File Explorer) to import your photos and videos without album/collection structure. Live photos get flattened via the Photos app. The older importer tool has the ability to organize photos and videos based on time, and it does preserve live photos, but the key photo and video of each live photo will not necessarily be kept together.
    - **iMazing:** The camera roll related features of iMazing closest resemble what iExtract is aiming to accomplish, but the issue with iMazing is that since it includes so many more features in addition to those ones, it has a high price tag and is subscription-based. This business model is not ideal for users that simply wish to convert their albums and collections into folders without having to manually reconstruct them from a raw dump, which is the kind of import method that the Windows Photos app provides for free.
    - **iTunes (Manually Copying User Created Folders of Photos/Videos from the iOS Files App):** This is a free method that anyone can use. However, the issues with this method are that it is entirely manual, tedious, wastes storage space on your phone during the process, and is prone to conversion failures, especially regarding live photos,

which get converted into still images if not manually converted to videos before exporting to the Files App. Additionally, the album/collection structure must be entirely manually recreated by the user in order to preserve it, which is extremely tedious and infeasible for large numbers of albums, especially since in order to copy an album to your Windows computer, for example, you must first create a folder of the same name in the iOS Files App, then you must ensure you convert all live photos to videos and re-add them to the original album, and then you must copy all of the items from said album into the folder of the same name in the Files App. After that, you must use iTunes to access the folder in which you stored these items and copy its contents to your computer.

- **Novelty:**
    - This application provides a cheaper and more focused method to users who want to avoid iCloud and backup their important albums and collections en masse, locally, in an automated manner. This application, unlike the alternatives, will not require an account, will be entirely free (or very cheap compared to the $29.99 per device price of iMazing), and will have power-user features, such as album blacklisting and whitelisting, ensuring users only back up the photos they want. It will be functional cross-platform, as it will use Python, which will allow it to be used on Windows, Linux, and macOS, or anywhere that has a Python interpreter for the version we decide to use. Additionally, unlike some tools (iTunes) which only allow users to back up their items via a tedious process if their phone is still functional, this application will work on unencrypted/decrypted iOS backups, not active phones, which allows for much more flexibility in terms of use-cases.
- **Effects:**
    - If we are successful in creating this app in the given timeframe, there will finally be a free (or very inexpensive) tool that may be used by anyone to locally backup their iOS devices' photos and videos without the friction that currently exists, such as cost and tedium.
- **Use Cases (Functional Requirements):**
    - User exports a single album (or collection; it will be a similar process) (*Part of **Feature 3***.):
        - Actors: Any iphone user

- Triggers: The user wants to export an album to a folder of the same name.
- Preconditions:
    1. The user must have an album already saved in their iPhone backup that they want to convert into a folder.
    2. The user's computer must have enough storage to store the extracted items in.
- Postconditions (success scenario): The selected album has been successfully exported to the folder of the same name.
- List of steps (success scenario):
    1. The user opens the CLI app.
    2. The user enters the proper command necessary to initialize the album export from the backup by using the whitelist feature to select it and export it, as a folder of the same name, to a selected parent folder.
    3. The files in the album are successfully located (converted if needed or selected) and exported into the selected folder.
    4. The program displays a success message indicating that the same number of files as the user selected have been exported.
- Extensions/variations of the success scenario
    1. User enters the full command following the name of the application executable, e.g., "./iExtract --backup_location = ~/myBackup/, --whitelist = dogs, --dest = ~/extractedAlbums/, --convert = heic/jpg, mov/mp4, live_photo/mp4, --cq_img = 0.8, --cq_vid = 0.75, cq_live = 0.9".
        - cq is short for "conversion quality".
        - Note: Real flag system may work differently.
    2. The files in the album are successfully located (converted in the specified qualities, since selected) and exported into the selected folder.
    3. The program displays a success message indicating that the same number of files as the user selected have been exported.
- Exceptions: failure conditions and scenarios

1. When whitelisting an album to export, the user enters the name of a non-existent folder:
    - The program discovers that no such album exists within the specified backup. An error message is printed stating that there is no such album and the program returns to the main menu.
2. When exporting the album, the parent folder for the new folder to be stored within, the destination folder, does not exist:
    - The program creates the non-existent folder and tells the user that the folder did not exist, so the program created it.
3. When exporting the album, the user enters an invalid backup folder location:
    - As the program cannot extract anything, given it doesn't know where the backup folder is, it simply fails immediately, tells the user that they have offered an invalid path to a backup, and returns to the main menu.

- User exports all albums (they may contain overlapping items) (*Part of **Feature 1***.):
    - Actors: A user with a highly organized album structure where multiple photos exist in multiple albums.
    - Triggers: The user wants to preserve their albums but does not want multiple copies taking up data.
    - Preconditions:
        1. The user must have an album already saved in their iPhone backup that they want to convert into a folder.
        2. The user's computer must have enough storage to store the extracted items in.
        3. Either the file system of the OS supports shortcuts, or the computer has enough storage space to store the duplicated items across multiple album folders.
    - Postconditions: There is a folder named "nonExclusiveItems" that is created within the main folder of the overall extraction. This folder contains the actual items that are not exclusive to one specific album, while the program provides shortcuts, or

symbolic links, to every file in each folder representing an album that that item is supposed to be in, which prevents wasting storage space. If the OS doesn't support this, the program simply extracts a given file into each folder it belongs in, wasting storage space, as there is no alternative in that situation.

- List of Steps:
    1. User initializes a full export of all albums.
    2. The system identifies that a certain photo is linked to both album A and album B.
    3. The album exports the real file into the "nonExclusiveAlbums" directory and adds a shortcut to that file to the folders representing both album A and album B.
- Exceptions: If the OS does not support shortcuts, then the system copies the full files everywhere they belong, and then it alerts the user that the shortcuts were not created successfully and, instead, the files were duplicated.

- User exports every collection (literally all of them) within their backup (*Part of **Feature 2.***):
    - Actors: An iPhone user
    - Triggers: The user wants to back up all their photos and videos.
    - Preconditions: The user must have enough free storage on the destination drive partition that they're trying to save to.
    - Postconditions: Files are stored without losing quality, and album/collection structure is preserved in the created folders of the same names within the destination folder.
    - List of Steps:
        1. The user opens the app.
        2. The user selects "Back up the entire camera roll".
        3. The app scans the photo library within the source backup.
        4. The app prompts the user to choose a backup destination.
            - Connected drive, network drive, cloud location.
        5. The user selects a destination.

6. App shows a summary screen:
    - Number of photos/albums to back up.
    - Chosen destination path.
7. User selects “Start Backup”.
8. The app shows a completion screen when finished.

- Convert proprietary formats to common formats within an extraction (*Part of **Feature 4***.):
    - Actors: User
    - Triggers: The user needs to view or share proprietary file types on a platform that only supports common formats
    - Preconditions: The media files have already been identified and iExtract has access to the local storage
    - Postconditions: All selected proprietary files are converted to standard formats
    - List of Steps
        1. User selects a folder of extracted files within the iExtract interface
        2. System scans for proprietary formats
        3. User selects the “convert to common format” option
        4. System converts all selected files into a standard format
        5. System notifies the user that the conversion was successful
    - Extensions/Variations: System either archives or replaces files based on the users preference.
    - Exceptions: If the file is unreadable, the system will mark it with an error tag before moving on to the next file.

- **Non-functional Requirements:**
    - **Portability:**
        - iExtract must work without a separate codebase for each platform on any machine that:
            1. Meets the system requirements, in terms of hardware.
            2. Has the ability to run programs of the Python version we choose to use.
    - **Usability:**
        - The error messages produced by the CLI must be clear and understandable to non-technical users.

        - The CLI must display a summary of the exported contents once an operation is complete.
    - **Maintainability:**
        - The architecture of the app will be modular such that different features are part of distinctly separate components, enabling features to be developed and/or fixed quickly without the risk of breaking other features.
- **External Requirements:**
    - Understanding the iPhone backup format:
        - This application can only function successfully if it is able to successfully parse an iPhone backup, which is external and unalterable to the developers of this app. Thus, the requirement that we study the backup structure of iPhone backups is tantamount to the success of this project.
    - Understanding the user's filesystem:
        - Operating system rules across Windows, Linux, and macOS must be followed in order to enable backups to be found and parsed into extracted folders.
    - Users expectations:
        - Users expect instructions and error messages to be clearly written and resolvable without our direct support.
    - The project must be open-source and buildable:
        - The source code for this application must be publicly available for others to download, enabling them to run it directly, without having to download a package or installer.
    - Consistent offline approach:
        - The tool must work offline, as it is made to perform pure offline iPhone backup extractions.
- **Technical Approach:**
    - We are planning to use Python and libraries that work across Windows, Linux, and macOS in order to create this app in such a way that it is functional across all the major desktop platforms. The main user experience will be a command line interface, but in order to reduce friction for less technical users, we will include a detailed list of commands that will be shown on-screen once the user enters the "Help" section of the menu. Additionally, we will be utilizing SQLite

in our program to query the databases iOS uses to track which albums and collections media belongs to.

- **Timeline:**
    - **Week 3:**
        - Ensure the architecture of the app, including its design, in terms of how the directories and modularity will be implemented, is finished, and the CLI minimally works without errors (no major features must be implemented at this point).
    - **Week 4:**
        - Ensure that the CLI design is complete, in terms of the basic mockup, and ensure that the architecture has been decided so development can truly begin in full.
    - **Week 5:**
        - Ensure that the user can navigate a mockup of the final CLI and choose certain actions, with the first functional one being to “load a backup” successfully.
        - Begin training the AI model that we will be using to fulfil the “stretch goal” feature: Add AI Descriptions of Content to Photo Metadata.
    - **Week 6:**
        - Ensure that the BackupModel exists in its *final form* so that development can move forward on the components that rely upon it.
        - Ensure the Domain Layer components of the Backup Locator and Validator, Conversion Engine, and the SQL Command Facilitator have been prototyped.
        - Start work on the GUI prototype (using Textual) that utilizes textual text user interface (TUI) technology, which works upon the CLI, actually works to a minimum degree.
        - Train the AI model that we will be using to fulfil the “stretch goal” feature: Add AI Descriptions of Content to Photo Metadata.
    - **Week 7:**
        - Ensure that the BackupModel can be populated with device metadata, including basic details (no asset or album

information yet) like the encryption status, phone model and submodel, backup GUID, etc.
    - Continue working on training the AI model that we will be using to fulfil the “stretch goal” feature: Add AI Descriptions of Content to Photo Metadata.
    - Continue working on the Textual GUI prototype.
    - Work on creating tests for the testing suite to increase robustness.
- **Week 8:**
    - Ensure that the Conversion Engine passes tests to convert a given Asset from one format to another, covering MOV and HEIC (including live photo pairings) to common formats such as MP4 and JPEG at a user-specified quality level.
    - Ensure that, within the CLI, the blacklist and whitelist work to specify which albums and collections to include or exclude from extractions, and that non-exclusive assets shared across multiple collections are handled correctly via shortcuts or symlinks rather than duplication.
    - Continue working on the Textual GUI system.
    - Continue working on training the AI model that we will be using to fulfil the “stretch goal” feature: Add AI Descriptions of Content to Photo Metadata.
    - Continue to work on creating tests for the testing suite to increase robustness.
- **Week 9:**
    - Continue training the AI descriptions feature for the beta showcase.
    - Continue with Textual GUI development.
    - Continue developing and testing iExtract’s mass export feature (integrate blacklist and whitelist functionality).
    - Create the beta release, including the mass export feature as the first major feature for people to test.
    - Finish up the blacklist/whitelist functionality.
    - Finish up the individual collection export functionality.
    - Ensure that each and every known failure mode so far has a detailed error message associated with it that explains to a common user what they must do to mitigate the error in the

future. Test the major features for bugs and usability issues and prepare for Week 10 (***1.0 Release Due Monday of Week 10***).

- **Week 10:**
  - Implement some extra feature(s) (like the AI descriptions) if we have time before the full release is due.

Team Process Description:

- **Software Toolset:**
  - **Python:** We are planning on using Python to develop our app quickly and incrementally. Additionally, using this language will allow us to deploy the app on multiple operating systems using one codebase if we don't utilize platform specific libraries (or if we implement multiple platform specific libraries to facilitate the same actions on the 3 OSes we are targeting: Windows 10+, Linux, and macOS.
  - **SQLite:** SQLite will need to be used for this application to function properly, as iPhone backups store a lot of data and metadata within SQLite databases such as Manifest.db and Photos.sqlite, which need to be parsed in order to reconstruct album and collection structure accurately in terms of folders. Moreover, Photos.sqlite contains metadata about images and videos, which we plan on preserving.
- **Risk Assessment:**
  - **Scope Creep (Medium Likelihood, High Impact):**
    - **Evidence for Estimates:** We have already come up with some advanced features, like SQL filtering or addons, and those could make it infeasible to complete our project on time if we were to integrate them into the design from the beginning.
    - **Steps to Reduce Likelihood & Impact & Permit Better Estimates:** To avoid this issue, we will develop incrementally, adding basic features first, which will compose a minimum viable product before we move on to adding things like advanced filtering and detailed metadata preservation.
    - **Plan for Detecting Problems:** Checking in on the timeliness of the integration of one of the advanced features and ensuring that it does not introduce bugs to the core functionality of the program by running automated tests on the core components.

        - **Mitigation Plan:** If this is to occur, we will need to scale back the scope of our project to something achievable before the deadline that provides value to a user and is stable.
    - **Team Members Not Communicating (Medium Likelihood, High Impact):**
        - **Evidence for Estimates:** We have all been in teams in the past that have failed to communicate and caused some members to have to crunch to make up for the lack of communication and retain their own grades.
        - **Steps to Reduce Likelihood & Impact & Permit Better Estimates:** We will hold each other accountable if any of us are not responding, even to follow-up messages, making the completion of the project strained.
        - **Plan for Detecting Problems:** We will try to follow-up to any ignored messages, and if a teammate still isn't responding, we will ask another team member to do so as well, and if that doesn't work, we will talk to the unresponsive team member in person and ask about what's going on.
        - **Mitigation Plan:** Once we contact the unresponsive person in person, or if we can't, we will try to more thoroughly establish communication protocols amongst the team so we don't fall behind.
    - **Difficulty of Understanding the iPhone Backups (Low Likelihood, High Impact):**
        - **Evidence for Estimates:** Based on the manual exploration that we have done, we have been able to find albums, some collections, and the items belonging to albums, as well as those items' metadata, partially, all in a repeatable pattern of steps. This makes it only slightly likely that it would be hard to understand the rest of what we need to understand and reconstruct a logical mapping of items to metadata.
        - **Steps to Reduce Likelihood & Impact & Permit Better Estimates:** We will ensure that we already understand how to perform the necessary operations on a backup that we are set out to automate manually before we begin attempting to automate those features.

        - **Plan for Detecting Problems:** To detect issues, we will test across backups of different iPhones and ensure that constructing a logical representation of the backup in memory works consistently.
        - **Mitigation Plan:** To counteract this risk, should it become reality, we will do research into the existing documentation around how iOS stores camera roll files and their metadata, which is publicly available.
    - **Complexity of Keeping Architecture Pure (Medium Likelihood, Medium Impact):**
        - **Evidence for Estimates:** It has already been a bit difficult to determine which architecture pattern we would like to use based on what our software will need to be doing, as it is not something that really fits nicely into one specific box.
        - **Steps to Reduce Likelihood & Impact & Permit Better Estimates:** We will attempt to refine the architecture of the app, which is the Layered Pattern, and ensure that it is consistent enough to where we don't have any major contradictions in our code that cause problems down the line.
        - **Plan for Detecting Problems:** To detect problems, we will go through the implementation of the layers of the pattern and see if anything within each layer violates a rule or not.
        - **Mitigation Plan:** To mitigate this risk, should it occur, we will move the incorrectly implemented functionality to the correct layer, so that we don't build up technical debt.
    - **Inability to Follow Established Conventions (Low Likelihood, Low Impact):**
        - **Evidence for Estimates:** We all have different coding styles, which we have already noticed based on the code that we have written in in-class exercises, and based on the reactions to the code that one team member wrote for the CLI mockup.
        - **Steps to Reduce Likelihood & Impact & Permit Better Estimates:** We will each read the style guides for the Python and SQL that we have put in this document, limiting the time it will take to fix any issues.
        - **Plan for Detecting Problems:** We will use a linter on each file to detect code inconsistencies.

        - **Mitigation Plan:** We will use linters and code auto-formatters to fix any issues that remain after manual fixing, and fix any unresolved issues after that manually.
- **How Risk Assessment Has Changed:**
    - Our risk assessment has become more precise and based on the project's technical specifications established since the Requirements document was made. In the beginning, our risks were wide-ranging and based on presumptions regarding the iOS backup format and the intricacy of the extraction process. Now, our risk estimates are more detailed and, the list itself, more complete. We have more extensive plans for problem prevention, along with plans to correct issues, may they still occur.

INSTRUCTIONS TO COMPLETE PROJECT SCHEDULE:

- Identify milestones (external and internal), define tasks along with effort estimates (at granularity no coarser than 1-person-week units), and identify dependencies among them. (What has to be complete before you can begin implementing component X? What has to be complete before you can start testing component X? What has to be complete before you can run an entire (small) use case?) This should reflect your actual plan of work, possibly including items your team has already completed.
- To build a schedule, start with your major milestones (tend to be noun-like) and fill in the tasks (tend to start with a verb) that will allow you to achieve them. A simple table is sufficient for this size of a project.

- **Project Schedule:**

ID	Milestone / Task	Role	Effort Estimate	Dependencies
M1	**Internal: System Architecture & Skeleton [Week 3]**	--	--	--
T1.1	Create skeleton code for all components with modular layered design	Developer	1 week	None

T1.2	Create mock data for initial feature testing	Director	1 day	T1.1
T1.3	Host weekly meeting & confirm skeleton functionality	Director	1 day	T1.1
M2	**External: UI/UX Design Approval [Week 4]**	--	--	--
T2.1	Identify user flow paths for core features	UI/UX & Director	3 days	M1
T2.2	Create initial CLI design and mockup	UI/UX & Director	1 week	T2.1
M3	**Internal: BackupModel Fully Populated [Week 6–7]**	--	--	--
T3.1	Define and finalize BackupModel domain layer (all data structures)	Developer & Director	3 days	T1.1
T3.2	Implement Backup Locator & Validator (device metadata from Info.plist / Manifest.plist)	Developer & Director	3 days	T3.1
T3.3	Implement SQL Command Facilitator (Photos.sqlite &	Developer & Director	1 week	T3.2

	Manifest.db queries, asset/album population)			
T3.4	Write unit tests for BackupModel builder and SQL component	QA & Developer	3 days	T3.2, T3.3
M4	**Internal: Conversion Engine Complete [Week 8]**	--	--	--
T4.1	Implement ImageConverter (HEIC → JPEG/PNG at user-specified quality)	Developer	3 days	T3.1
T4.2	Implement VideoConverter (MOV → MP4 at user-specified quality)	Developer	3 days	T3.1
T4.3	Implement LivePhotoConverter (Reuse the Other Conversion Components) (HEIC + MOV → MP4)	QA & Director	3 days	T4.1, T4.2
T4.4	Write unit tests for all conversion paths	QA	3 days	T4.1, T4.2, T4.3
M5	**Internal: Core Functional Readiness (Features 1 & 2 — Mass Export) [Week 8–9]**	--	--	--

T5.1	Implement ExtractionPlanner (determine what to extract based on BackupModel)	Developer & Director	3 days	T3.3
T5.2	Implement FolderBuilder (create destination folder structure mirroring album hierarchy)	Developer & Director	2 days	T5.1
T5.3	Implement FileCopier (copy media files, invoke Conversion Engine when needed)	Developer & Director	3 days	T5.2, T4.1, T4.2
T5.4	Implement SymlinkManager (handle non-exclusive assets across multiple albums)	Developer & Director	2 days	T5.3
T5.5	Integrate File Extraction Engine backend with CLI	UI/UX & Developer	3 days	T2.2, T5.4

- **Team Structure (Member Roles & Justifications):**
    - **Kevin (Director):** This project needs a director so that the direction of the project stays on track and within scope. This role will prevent us from wasting time or not knowing what to do while waiting for others to finish their work. Kevin is being chosen for this role because the project was his idea, so it makes sense that the general direction of it should stay in line with that idea.
    - **Noah (UI/UX):** A good program requires intuitive controls and a navigable menu. We want our program to be user-friendly and easy to use. Noah is being chosen for this role because he has prior experience in creating UI and designing websites.

- **Brendon (QA Tester):** Having a QA tester ensures that we catch functional failures and edge-cases. The person taking on this role will need to create test cases and give us criteria for correctness. He will also spend time to validate that the product is usable and reliable for users.
- **Sam (Developer):** This program requires developers to ensure that the project is functional and meets the product description. This role will work on the back end of the program to build up all the required features and will be continuously reviewing the code to ensure that the quality is upheld throughout the codebase. Sam is being chosen for this role because he has prior experience in working as a back end developer in previous projects.

INSTRUCTIONS TO COMPLETE TEST PLAN: Describe what aspects of your system you plan to test and why they are sufficient, as well as how specifically you plan to test those aspects in a disciplined way. Describe a strategy for each of unit testing, system (integration) testing, and usability testing, along with any specific test suites identified to capture the requirements.
NOTE: We require that you use GitHub Issues to track bugs that occur during use and testing.

- **Test Plan:**
  - We will test our CLI menu to make sure that each option works accordingly. We will also independently test each feature in our app and also have a final test of the whole app when we complete the beta version. We plan to conduct manual tests at the end of each week so we can catch and fix any bugs that we find in the following week. We will be tracking all of our tests from our regression testing sheet and make any updates accordingly. We will also have acceptance criterias for each feature to make sure that we meet all of our functional and non-functional requirements. We will also track each individual issue as it appears using GitHub issues.
  - Unit Testing
    - We will utilize unit testing for the Application and Data Layers. More specifically, targeting the logic functionalities like validation logic, data transformation, and decision making algorithms. We will utilize Python's built-in 'unittest' framework alongside unittest.mock for mocking, to isolate the

Application and Data Layers. We chose unittest over alternatives because it is included in Python's standard library, requiring no additional dependencies, and integrates directly with our CI pipeline via a single 'python -m unittest discover tests' command.

- We considered but decided against the following alternatives: 'Pytest', which offers more expressive syntax and better output formatting but introduces an external dependency and was more than we needed for this project; 'Nose2', which extends 'unittest' but is largely in maintenance mode and has a smaller community; and 'Hypothesis', which provides property-based testing and would be useful for fuzzing our SQL query logic, but was too complex to justify for our current test coverage goals.

- Test Automation Infrastructure and Continuous Integration
  - Our CI service is GitHub Actions, linked directly to our repository at https://github.com/Gustakev/cs362-class-project. GitHub Actions is configured via a YAML workflow file in '.github/workflows/' that triggers automatically on every push and pull request to the main branch, running 'python -m unittest discover tests' against the commit. We chose GitHub Actions because it is natively integrated into GitHub with no external account or service required, is free for public repositories, and requires minimal configuration to get going for a Python project. The following table compares the CI services we considered and what we decided on:

CI Service Option	Pros	Cons
GitHub Actions	Native GitHub integration, no external setup, free for public repositories, large marketplace of reusable actions, YAML config file lives in the repo	Less flexible than dedicated CI tools for complex pipelines, limited free minutes for private repos (but ours is public)
CircleCI	Highly configurable, fast build times, good	Requires external account and service linkage, more

	parallelism support, free tier available	complex setup than GitHub Actions for a simple Python project
Azure Pipelines	Enterprise-grade, great Windows support, free for open-source usage	Overkill for a small academic project, more complex UI and configuration, primarily designed for Microsoft ecosystem workflows, which we aren't really using
Travis CI	Simple YAML configuration file, long-established community, good Python support	Free tier was significantly reduced in 2021 and is now very limited for open-source projects, making it less practical for teams if the team isn't willing to pay for a paid plan

    - Usability Testing
        - We will conduct observational tests, where participants will attempt to perform core functionalities. This will help validate that the CLI is usable and conveys understandable messages.
    - This test strategy is sufficient because it covers the entire system at three different levels. Unit testing ensures that the (Data Layer) complex logic is bug free in isolation. Integration testing makes sure that file system operations (Application Layer) work on the host machine. Usability testing makes sure that the CLI (Presentation Layer) is navigable and intuitive.
- **Documentation Plan:**
    - **Readme:** Include instructions on how to use the app within the file 'README.md' hosted on the GitHub repository. We will update these instructions, starting in Week 6, and until the release of the project.
    - **Help Section:** The main menu of our app will include a help section which, when entered, will display all the different sorts of clarifications and FAQ answers to common questions and confusions regarding the functionality of the app.
- **External Feedback:**
    - **Export Everything Feature Complete:**

        - At this point, we will need to ensure that the export actually exports everything, not including thumbnails and other files that are meant to be temporary and aren't shown in the Photos app on iOS. To do so, we may recruit some people we know in order to test this tool on backups of their phones, as well as to test this feature in front of TAs and/or the professor to ensure that the code is not only functional, but that it is also efficient and complies with common standards.
    - **Export Only Certain Albums/Collections Feature:**
        - At this point, we will need to ensure that when a user uses the application with the intention of exporting only certain albums and collections, only those are included in the export, and that all the real items are included in each exported album and collection, without the temporary files and thumbnails. Once again, to do this, we will need to have multiple people, some that we know in our personal lives, and perhaps some TAs, attempt to use the tool for this purpose and demonstrate its usage on backups of their devices. We will also need to have TAs and other people with expertise look over our code and help us catch bad practices early on, so that we don't build up technical debt.
    - **Conversion Quality Testing:**
        - When a user chooses that they would like to convert proprietary file types like live photos into common file types, those conversions need to be done properly in every case. In order to ensure that this is true, we need to test the program on multiple backups from different devices, ensuring consistent behavior. We will likely need external sources to volunteer backups for us to perform operations on, or we will need to create more backups to examine ourselves. Alternatively, we could have people try out the tool on their own time, so that we don't need to invade their privacy, and they could tell us if the conversion process was successful or not, allowing us to get real world "customer" feedback.

Major Features (4+):

1. Mass Album -> Folder Conversion

2. Mass Collection -> Folder Conversion
3. Folder & Collection Blacklisting (And Whitelisting) System
    a. This feature will enable users to choose which albums and collections they want to convert into folders.
4. Proprietary File Type -> Common File Type Conversion Options

Stretch Goals (2+):

1. Extract Folders of Files from iOS Files App & Incorporate Blacklist (And Whitelist) Features
2. Custom Folder Creation Implemented Via SQL Filtering
    - The user would use a UI to apply the filters, not needing to enter the commands themselves.
3. Implement a GUI Alternative to the CLI
4. Add AI Descriptions of Content to Photo Metadata

Notes:

- The major features should constitute a minimal viable product (MVP).

TO-DO:

1. Refer to the GitHub Issues page: [Issues · Gustakev/cs362-class-project]