AES-Based Local Password Manager with Dynamic Encryption Techniques

Capstone-I project report submitted

by the student of

Hybrid UG program in Computer Science & Data Analytics

Abhijeet Kumar

Roll No. **24A12RES802**

Group No. **20**

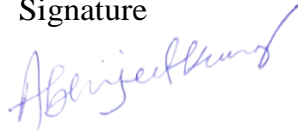INDIAN INSTITUTE OF TECHNOLOGY PATNA
BIHTA - 801106, INDIA
Date: 27/04/25

Declaration

I hereby declare that t h i s submission is my own work and that, t o the best of our knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of the university or other institute of higher learning, except where due acknowledgment has been made in the text.

Date: 27/04/25

Name: **Abhijeet Kumar**, Roll: **24A12RES802**
Group No: 20

Signature

Summary of the Project

Summary

This project focuses on developing a secure local credential manager using AES encryption to protect sensitive data such as usernames and passwords. It is designed so that the user only needs to remember a single master password. This master password is used to derive the encryption key, which securely encrypts and decrypts all of the stored credentials. Users can store multiple service names, usernames, and passwords under the protection of just one master password.

The primary objective is to ensure that user credentials are stored securely, and that operations such as adding, updating, and deleting credentials are properly supported. The backend is built using SQLite, providing a lightweight and serverless database solution. It consists of two main tables: one for storing credentials (service name, username, and encrypted password) and another for configuration settings such as the encryption salt.

For security, the system employs AES encryption combined with PBKDF2 for key derivation. The master password itself is never stored in the database. Instead, it is used at runtime to derive the encryption key dynamically, which is then applied to encrypt or decrypt the credentials.

Notably, even if an incorrect master password is provided, the system proceeds with decryption and outputs data based on the stored encrypted content, without revealing whether the input was correct. There are no hints, errors, or failure messages that could leak information, maintaining a true Zero-Knowledge behavior. Correctness can only be determined by actually using the decrypted credentials on the intended service or by the user itself.

This follows a Zero-Persistent Key philosophy, meaning the encryption key is never stored at any point. It is derived dynamically every time the user accesses their credentials, ensuring that the encryption key remains secure even if the database itself is compromised.

The user interface (UI) is being developed using the Tkinter and CustomTkinter libraries, offering flexibility and modern design for a responsive desktop application.

Future work will include features such as CSV export/import for data portability and automatic backups for enhanced reliability. These improvements are planned for future versions.

Mention the work done by each member

1. Abhijeet Kumar (**24A12RES802**):

   Proposed the foundational idea of the project, laid out the development plan, and worked on backend and core algorithm development using Python and SQLite. By the time of this mid-term report submission, approximately 90% of the backend work has been completed, contributing to around 60% of the overall project progress.

2. Abhijeet Kumar (**24A12RES801**):
3. Abhijeet Kumar (**24A12RES803**):
4. Abhishek Kumar (**24A12RES115**):
5. Abhishek Kumar (**24A12RES807**):

Contents

1. Introduction

1.1 Background

In a digital-first landscape, the usage of online services by both individuals and organizations has seen a rapid increase, each relying on credentials like usernames and passwords for authentication. With this expanding list of credentials, the challenge of securely managing them also grows. Often people cut corners by reusing passwords, writing them down, or keeping them in insecure digital notes, creating vulnerabilities that can be easily exploited. The more convenient these shortcuts are, the more dangerous they become from a security standpoint.

The motivation for this project emerged from a clear gap: a need for a simple, **local-first** solution that could offer robust protection for sensitive login information without depending on external servers or internet connectivity. Unlike popular cloud-based managers that sync across devices, this project is deliberately **offline, self-contained, and dependency-free**, giving users complete control over their data.

The core idea was to ensure that no sensitive data—including the master password—is ever stored in plain form. If the master password is lost, there should be no recovery option, reinforcing a privacy-first, zero-trust design philosophy. This approach ensures that even if the storage device falls into the wrong hands, the encrypted credentials remain inaccessible without the user's key knowledge.

The project also serves as a practical application of advanced cryptographic techniques—specifically, the **Advanced Encryption Standard (AES)**—implemented through Python and supported by an SQLite database. This balance of security and simplicity formed the foundation of the development effort.

1.2 Credential Management Landscape

Credential managers today fall broadly into two categories: **cloud-based** and **local/offline-based**. Cloud-based managers (such as LastPass, 1Password, or Bitwarden) offer convenience by synchronizing data across devices (Sommerville, 2021). However, this convenience comes at a cost—cloud infrastructure, even when encrypted, introduces risks such as centralized data breaches, unauthorized access, and metadata leaks. On the other hand, offline or local credential managers store data directly on the user's device and typically offer more control but less flexibility in terms of syncing and recovery. Many such tools require manual setup and lack user-friendly interfaces, making them inaccessible to average users.

Moreover, some local tools still rely on external storage of keys or use weak forms of encryption. These design compromises often stem from trying to maintain simplicity, but they ultimately weaken the security of the system.

This project sits at a hybrid point in that spectrum: **a purely local credential manager** that integrates **strong AES encryption, secure key derivation with PBKDF2**, and an intuitive storage format using SQLite. Its focus is not to compete with commercial products but to provide a secure, transparent, and educational alternative that demonstrates how encryption and software design can intersect meaningfully.

1.3 Aim & Objectives

The primary goal of this project is to develop a standalone, locally executable password and credential manager that:

- Stores usernames, passwords, and associated metadata (like URLs or app names)
- Encrypts and decrypts credentials on the fly using a dynamic, user-provided master password
- Ensures that no sensitive credential is stored unencrypted at any point
- Enables basic credential management operations such as add, update, delete, and search
- Offers a backup mechanism in the form of CSV file.

2. Problem Definition & Proposed Solution

2.1 Why Local Credential Managers?

As the number of services, we access daily grows—ranging from personal banking to developer tools—the need to securely store and retrieve passwords has become a persistent concern. Despite the widespread availability of password managers, many users are hesitant to trust third-party solutions with their most sensitive data. This skepticism is not unfounded, especially considering the growing number of data leaks and security incidents involving high-profile services.

A **local credential manager**, by contrast, avoids the internet altogether. By storing and managing credentials entirely on the user's device, it removes the risk of network-based attacks or cloud service breaches. It ensures that users are not dependent on an external service to access their data.

Local solutions also allow full transparency. The user knows where the data is stored, how it is encrypted, and when it is accessed. There are no hidden processes, no telemetry, and no background sync jobs.

2.2 Cloud Risks & Security Trade-offs

While cloud-based password managers offer significant convenience—syncing data across multiple devices and providing recovery options—they introduce a range of security trade-offs:

- **Centralized Targets**: A cloud vault becomes a high-value target for attackers. Even with encryption, attackers can attempt brute force or social engineering attacks if they gain access.
- **Metadata Leakage**: Even if passwords are encrypted, information about when and how often you use credentials may be exposed.
- **Key Recovery Policies**: Some cloud services store master password hashes or recovery keys, potentially weakening the "zero-knowledge" principle they claim to follow.
- **Availability Dependency**: If the cloud service is offline, discontinued, or blocked (e.g., in a restricted environment), access to your credentials can be affected.

This project aims to completely sidestep these risks by enforcing **offline-only** storage, **no recovery** design, and **fully local** encryption.

2.3 Scope and Limitations

The system is intentionally designed with a focused scope. It stores and retrieves credentials securely on a single device and does not provide multi-device syncing or online access. This limitation is a conscious decision to enhance security and reduce complexity.

Key features supported include:

- Storing multiple credentials, each consisting of a service name (or URL), username, and password.
- Encrypting all sensitive fields using AES encryption tied to a dynamic master password.
- Simple text-based search within stored credentials.
- Optional export/import feature via encrypted CSV for portability.

However, it does not support:

- Password auto-fill or browser integration.
- Biometric authentication.
- Recovery mechanisms in case of a forgotten master password.
- Cross-platform sync or cloud backup.

These exclusions are by design and aligned with the **privacy-first**, **user-controlled** philosophy of the application.

2.4 Design Philosophy & Core Features

 The system is built around the principle of "Zero Persistent Key," meaning the master password or any derived encryption key is never stored anywhere, not even in encrypted form. Users must enter their master password every time to access or modify their data, and all encryption and decryption happen entirely at runtime. The design also ensures that there is **no error, no rejection, and no failure**, even if the master password entered is wrong. Decryption will always produce some output, whether correct or not, without any hint from the system. Only the user (or the real service) can verify if the decrypted credentials are valid by actually trying them.

Core features include:
- Local-only credential storage
- AES encryption with runtime key derivation
- Search, add, update, and delete operations
- Portable architecture with no installation dependencies
- Encrypted backup and restore functionality

This minimalist yet secure approach ensures that **only the user has access to their data**, and the application remains lightweight, auditable, and easy to understand. **Beyond just being a tool**, the project is also an effort to **empower individuals** by encouraging them to take ownership of their digital privacy. In an age where convenience often overrides caution, this application advocates for **personal responsibility in safeguarding sensitive information**. Relying on third-party cloud services to manage highly private data—such as passwords—goes against the core principle of personal data ownership. Through this system, users are not only offered a secure solution but are also introduced to the foundational ideas of **encryption, local storage, and data sovereignty** in a transparent, educational way.

3. System Overview & Requirements

3.1 Functional & Non-Functional Requirements

The system is built to deliver secure, offline-first credential management with simplicity and transparency. Its requirements are divided into **functional** and **non-functional** categories.

Functional Requirements:
- **Master Password Prompt on Startup**
  The system must request a master password to unlock access to stored credentials.
- **Add, View, Update, and Delete Credentials**
  Users must be able to create, read, update, and delete entries containing:
  - Service/Application Name
  - Username or Email
  - Encrypted Password
  - Optional metadata (e.g., Notes)

- **Encryption & Decryption of Data**
  All credentials must be encrypted using AES before storage, and decrypted only in-memory after user authentication.
- **Search Functionality**
  Ability to perform basic keyword search over service names or usernames.

- **Encrypted Backup and CSV Import/Export**
Allow users to create and restore backups in encrypted format, and optionally use a CSV format for portability.

Non-Functional Requirements:

- **Security**
    - No sensitive data should be stored in plaintext at any time.
    - The encryption key must be derived dynamically using the master password and a salt, without persisting either.
- **Portability**
The system should run on any windows machine (windows 10 or higher) as a standalone **.exe**.
- **Performance**
Database operations (CRUD) should execute efficiently even with a growing number of records.
- **Maintainability**
The codebase should be modular, with logical separation between encryption logic, database interactions, and user interface.
- **Usability**
The interface is GUI based, should be minimal and intuitive.

3.2 System Architecture & Data Flow

System Architecture

At a high level, the system consists of three main components:
1. **User Interface Layer**
Handles user input/output, using Graphical User Interface.
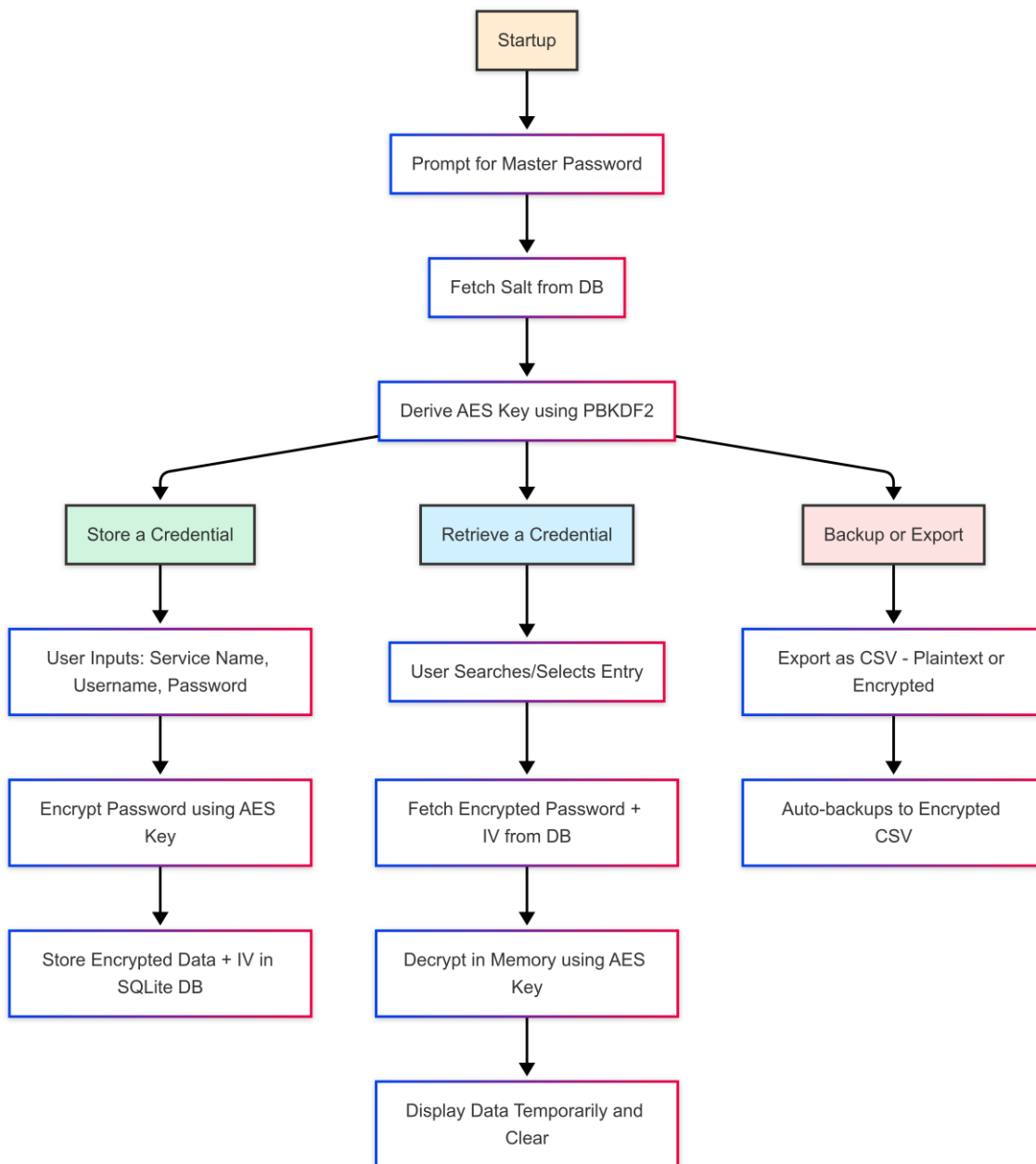2. **Application Logic Layer**
Coordinates between modules: handles user authentication, routes encryption/decryption calls, performs validation.
3. **Data Persistence Layer**
Uses SQLite to store encrypted credential records.

Data Flow Overview

The data flow follows a strict pattern to maintain security.

```mermaid
flowchart TD
    Startup --> Prompt[Prompt for Master Password]
    Prompt --> Fetch[Fetch Salt from DB]
    Fetch --> Derive[Derive AES Key using PBKDF2]
    Derive --> Store[Store a Credential]
    Derive --> Retrieve[Retrieve a Credential]
    Derive --> Backup[Backup or Export]

    Store --> UserInput[User Inputs: Service Name, Username, Password]
    UserInput --> Encrypt[Encrypt Password using AES Key]
    Encrypt --> StoreDB[Store Encrypted Data + IV in SQLite DB]

    Retrieve --> Search[User Searches/Selects Entry]
    Search --> FetchEnc[Fetch Encrypted Password + IV from DB]
    FetchEnc --> Decrypt[Decrypt in Memory using AES Key]
    Decrypt --> Display[Display Data Temporarily and Clear]

    Backup --> Export[Export as CSV - Plaintext or Encrypted]
    Export --> Auto[Auto-backups to Encrypted CSV]
```

Security Notes

- The master password is never stored.
- The key derivation process includes a random salt and a high iteration count for protection against brute force.
- Initialization vectors (IVs) are generated per encryption session and stored alongside encrypted data.
- Memory is cleared immediately after decryption to avoid leaks.

4. Development Stack

4.1 Languages, Libraries, and Tools

The backend logic was written in **Python 3.11**, chosen for its enhanced performance and modern language features. Core cryptographic operations use the cryptography library, ensuring secure implementation of AES (Advanced Encryption Standard) and PBKDF2 key derivation.

SQLite serves as the embedded database for credential storage, offering reliable local persistence with minimal setup. Additional standard libraries, such as csv, support data portability through backup and export features.

4.2 Development Environment & Version Control

Development was conducted using **Visual Studio Code** across Linux and Windows platforms. The project repository is managed on **GitHub**, with Git used for version control. Code was organized in branches to separate features and maintain a clean main development line.
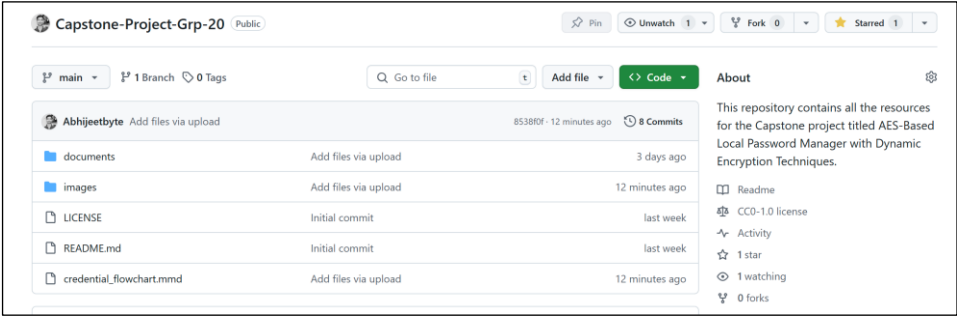

Fig: Repository Overview

Summary of Key Tools and Libraries

Component	Tool / Library	Purpose
Programming Language	Python 3.11	Core backend development
Database	SQLite	Encrypted local credential storage
Cryptography Library	cryptography	AES encryption, PBKDF2, key derivation
CSV Utility	Python csv module	Backup, import/export functionality

Development Stack Summary

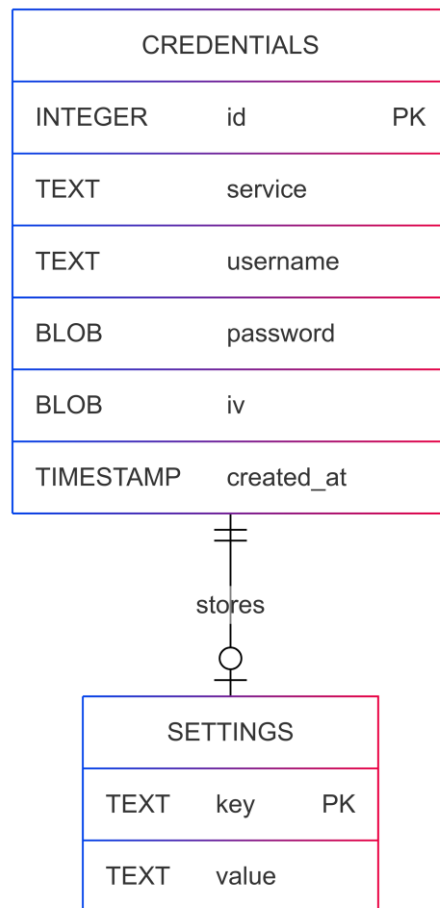| Category	Tool
IDE	Visual Studio Code
OS	Windows 11
Version Control	Git
Hosting Platform	GitHub.com

5. Database & Backend Design

5.1 Schema & Table Structure

The project uses SQLite, a lightweight and serverless database engine, suitable for local storage needs without introducing additional configuration complexity. Only one table is required for storing user credentials, as the scope is limited to single-user storage on a single device. Below is a description of the schema used for this project.

Main Table: credentials
The credentials table stores all the necessary user information related to the credentials being managed. The table has the following columns:

- **id (INTEGER PRIMARY KEY AUTOINCREMENT)**: This is the **Primary Key (PK)** of the credentials table. It uniquely identifies each record. The AUTOINCREMENT ensures that a unique ID is automatically generated for each new credential, which prevents duplication and ensures efficient data retrieval.
- **service (TEXT)**: This column stores the name of the service or application (e.g., Gmail, Facebook) associated with the credential.
- **username (TEXT)**: This stores the user's login name or email address associated with the service.
- **password (BLOB)**: The encrypted password is stored as binary data (BLOB). This column never contains plaintext passwords due to the encryption mechanism applied before storage.
- **iv (BLOB)**: The **Initialization Vector (IV)** used in AES encryption is stored as binary data. The IV ensures that the encryption process is secure and different each time the password is encrypted, even for identical passwords.
- **created_at (TIMESTAMP)**: This timestamp records when the credential was created, helping with tracking or sorting of stored data.

CREDENTIALS

INTEGER	id	PK
TEXT	service	
TEXT	username	
BLOB	password	
BLOB	iv	
TIMESTAMP	created_at	

stores

SETTINGS

| TEXT | key | PK |
| TEXT | value | |

Secondary Table: settings

The settings table is used to store configuration data, such as encryption salt and backup metadata. It consists of the following columns:

- **key (TEXT)**: This column acts as the **Primary Key (PK)** of the settings table. Each setting (e.g., "salt", "last_backup") is uniquely identified by its key. This column ensures that no two configuration entries are identical and supports efficient data retrieval.
- **value (TEXT or BLOB)**: The associated value for each setting. For instance, the salt for encryption might be stored as a BLOB, and the last backup timestamp might be stored as TEXT.

Design Rationale

This schema was chosen for its simplicity, clarity, and efficiency. By using **Primary Keys (PK)** in both tables, data integrity is ensured, and no duplicate entries can occur. The use of **AUTOINCREMENT** in the credentials table guarantees that each record is assigned a unique and incremental ID, simplifying data management and retrieval.

Moreover, encryption is applied to the password before it is stored in the database, meaning the database never stores plaintext passwords, enhancing security. The separation of configuration data into the settings table also makes the design scalable and adaptable for future enhancements.

5.2 Backend Module Layout

The Python backend is structured into clearly defined modules. Each file handles a specific responsibility, following the principles of separation of concerns and modular design. The primary modules are:

- **encryption.py**: Handles AES encryption and decryption using the cryptography library. Implements key derivation using PBKDF2, and wraps data securely with random IVs.
- **db_handler.py**: Manages all interactions with the SQLite database—initializing tables, inserting records, querying data, and handling updates or deletions.
- **main.py**: Acts as the entry point for the application. It handles the program flow, prompts the user, collects input, and routes actions to the appropriate modules.
- **utils.py**: Contains helper functions for formatting, logging, and secure memory handling.
- **backup.py**: Handles export/import of credentials in encrypted CSV format.
-

This modular structure allows independent development and testing of encryption logic and database operations. Additionally, it improves maintainability by reducing coupling between components.

5.3 CRUD, Logging, and Error Handling

CRUD Operations
The four CRUD operations are implemented as follows:

- **Create**: The plaintext password is encrypted before insertion into the database.
- **Read**: Encrypted entries are fetched, decrypted in memory, and displayed briefly.
- **Update**: Passwords are decrypted, displayed securely, then re-encrypted upon update.
- **Delete**: Deletion is permanent, with an optional backup prompt before removal.

```python
def create_credential(service, username, password):

def read_credential(service, username):

def update_credential(service, username, new_password):

def delete_credential(service, username):
```

Logging

Logging is intentionally minimal to avoid leaking sensitive data . Where necessary (e.g., during a failed decryption), logs only relevant messages such as error types or success information on CLI terminal during the development.

```python
# Configure the logger
logging.basicConfig(level=logging.DEBUG, format='%(asctime)s - %(levelname)s - %(message)s')

def log_error(message):
    logging.error(message)

def log_info(message):
    logging.info(message)
```

Error Handling

Error handling is implemented at critical points, such as database connections, user inputs, and cryptographic operations. The system uses **try-except** blocks to catch exceptions and provide clear error messages, ensuring smooth user experience without crashes.

```python
def add_credential(service, username, password):
    try:
        # Encrypt password before storing
        encrypted_password = aes_encrypt(password, key)
        # Store encrypted data in the database
        store_to_db(service, username, encrypted_password)
    except ValueError as e:
        log_error(f"Invalid input: {e}")
    except Exception as e:
        log_error(f"Failed to add credential: {e}")
```

Fig: Example of error handling

6. Encryption Mechanism (AES)

6.1 AES Overview & Key Derivation

Encryption mechanisms such as AES (Advanced Encryption Standard) are widely used to ensure secure data storage. AES, a symmetric key encryption algorithm, is used to encrypt user credentials before storing them in the database (National Institute of Standards and Technology, 2001). AES encryption operates by using the same key for both encryption and decryption, which is derived from the user's master password using PBKDF2 (Eastlake & Jones, 2001).

In this project, AES with a 256-bit key length is utilized to ensure strong security for the stored credentials. It is used in conjunction with careful key derivation, random initialization vectors (IVs), and a specialized decryption philosophy designed to eliminate side-channel leakage. The entire encryption mechanism ensures that no system behavior, error message, or internal rejection can ever hint to an attacker whether a decryption attempt was correct or not.

6.1 AES Overview & Key Derivation

For this system, AES-256 in CBC (Cipher Block Chaining) mode is employed to encrypt user credentials, providing robust protection against both passive and active attacks.
To prevent direct dependence on the user's master password, the system uses a **Key Derivation Function (KDF)** — specifically, **PBKDF2** — to generate the encryption key.

The derivation process is as follows:
- The user inputs their master password.
- A random **salt** is fetched from the database.
- The master password and salt are processed through PBKDF2 to produce a strong 256-bit AES key.

```python
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.primitives import hashes
import os
    ...

def derive_key(master_password: str, salt: bytes) -> bytes:
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=32,
        salt=salt,
        iterations=100000
    )
    return kdf.derive(master_password.encode())
```

Fig: AES Key Derivation using PBKDF2

The **salt** ensures that even if two users select identical master passwords, the resulting AES keys differ, making precomputed attacks like rainbow tables ineffective.
Importantly, the master password itself is **never stored** anywhere in the system. Only the salt (necessary for key derivation) and the encrypted

credentials are preserved. Thus, even in the event of a database breach, no sensitive key material is exposed.

```python
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
import os

def encrypt_data(key: bytes, data: str) -> bytes:
    iv = os.urandom(16)
    cipher = Cipher(algorithms.AES(key), modes.CBC(iv))
    encryptor = cipher.encryptor()
    ciphertext = encryptor.update(data.encode()) + encryptor.finalize()
    return iv + ciphertext

def decrypt_data(key: bytes, encrypted_data: bytes) -> str:
    iv, ciphertext = encrypted_data[:16], encrypted_data[16:]
    cipher = Cipher(algorithms.AES(key), modes.CBC(iv))
    decryptor = cipher.decryptor()
    plaintext = decryptor.update(ciphertext) + decryptor.finalize()
    return plaintext.decode()
```
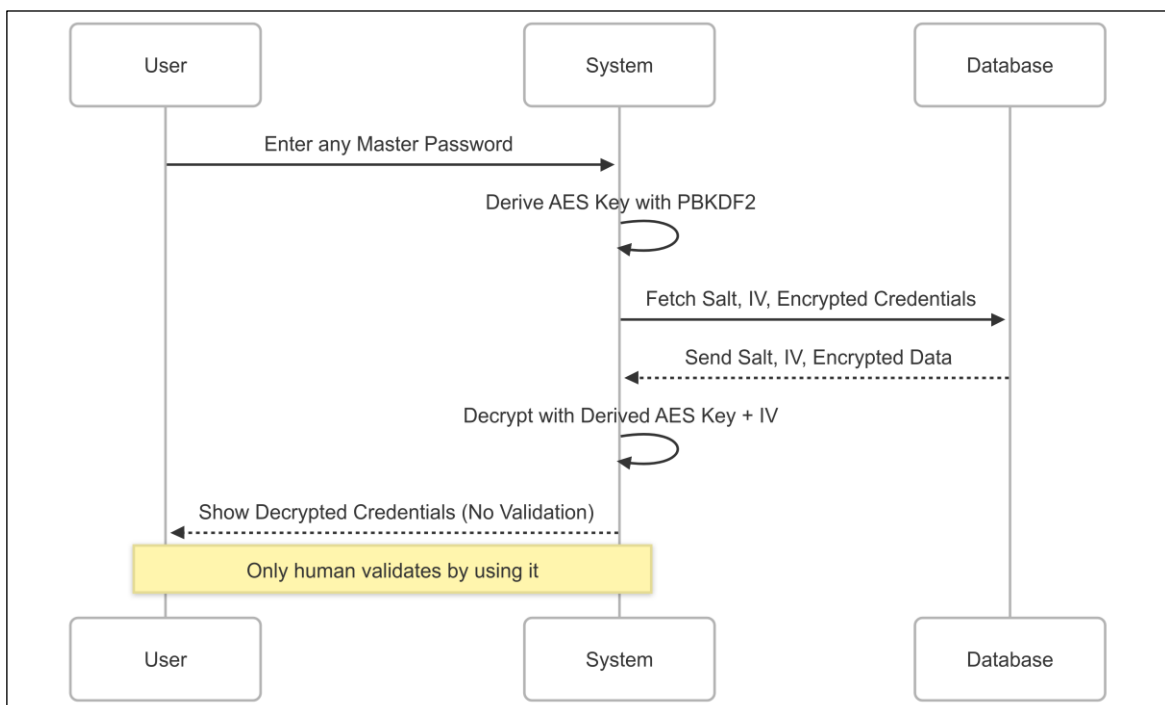
Fig: AES Encryption and Decryption Example

6.2 IV Management & Secure Workflow

Each time a new credential is encrypted, the system generates a new random **Initialization Vector (IV)** to maintain semantic security. This prevents attackers from detecting patterns across multiple encrypted entries.

The **secure workflow** can be summarized as:
1. User enters their master password.
2. Salt is retrieved from the database.
3. The AES key is derived using PBKDF2 with the salt and master password.
4. A random IV is generated for the current encryption operation.

5. Credentials (service name, username, password) are encrypted using AES-256-CBC.
6. The encrypted credentials, salt, and IV are stored in the database.

During **decryption**:

1. The user inputs a master password.
2. The system retrieves the salt and IV from the database.
3. An AES key is derived using PBKDF2 with the provided master password and the stored salt.
4. Decryption is attempted using the derived key and IV.
5. The resulting plaintext is output without any internal validation or error signaling.

Critically, this system never checks the validity of the decrypted output. Even if the user enters the wrong master password, the system silently decrypts and displays whatever data results — whether meaningful or not. Only the user can determine, by attempting to use the credentials externally, whether they are valid.

6.3 Sequence Diagram & Security Analysis

Sequence Diagram (Logical Flow)
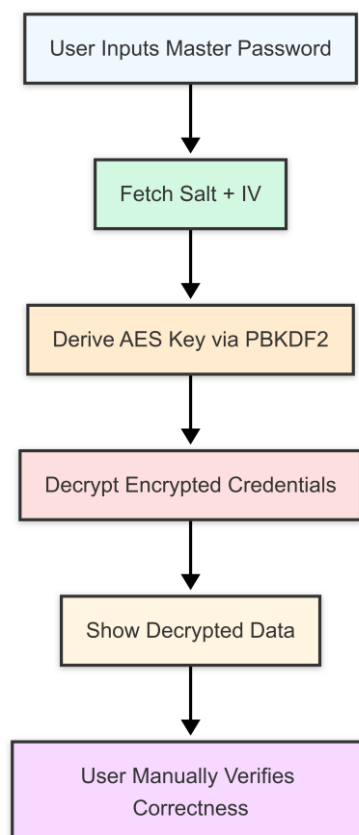
Decryption Behavior: No Oracle, No Error

Key Differentiator

> **In traditional systems:**
> Entering an incorrect master password typically causes decryption to fail with a visible error (e.g., padding error).
> Such behavior can serve as an **oracle leak** — revealing hints about whether an attacker is "closer" to the correct password.
> **In this system:**
> Regardless of the master password's correctness:
> - Decryption **always** succeeds technically (outputs something).
> - No system error or rejection is raised.
> - No hint is given whether the decryption was successful.
>
> Only by practically attempting to use the decrypted credentials, can the user determine if the decrypted credentials are valid.

Core Principal Summary

Principle	Meaning
No Error on Wrong Password	No Oracle, no leakage of correctness information
Always Produce Decryption Output	No observable distinction between right and wrong
No Stored Master Password	Safe even if full database is stolen
Random Passwords Preferred	Hide semantic leakage, prevent human guessability

This design prevents attackers from using system behavior to learn anything about the master password. Without errors, rejections, or visible hints, every decryption attempt appears identical.

If user passwords and usernames are **random-looking** (not human-readable), then even an attacker observing decrypted data cannot distinguish between correct and incorrect decryption.

This achieves a **Zero-Knowledge-like** (ZKProof, 2023) security posture:
- No side-channel leakage.
- No clues from system output.
- Only external verification (actual login attempt) reveals whether credentials are correct.
-

Thus, even under extensive brute-force attacks, attackers are blind — unable to determine progress without explicitly trying each set of decrypted credentials in real-world services.

- **Encouraging user to use random, machine-generated passwords** further improves secrecy.
- **Normalizing decrypted output** can help prevent garbage strings from disrupting GUI applications.

7. User Interface & Usability

The user interface (UI) and usability aspects of the AES-Based Local Credential Manager are still in the early stages of development. However, several key ideas and features have been outlined, with a primary focus on simplicity, clarity.

Key UI features include:
- **Login Screen**: A secure screen where users will enter their master password to access the credential manager.
- **Dashboard**: Displays the stored credentials, offering options to add, edit, or delete entries as needed.
- **Backup/Export Options**: Users can back up their credentials or export them in an encrypted format for safekeeping or transport.
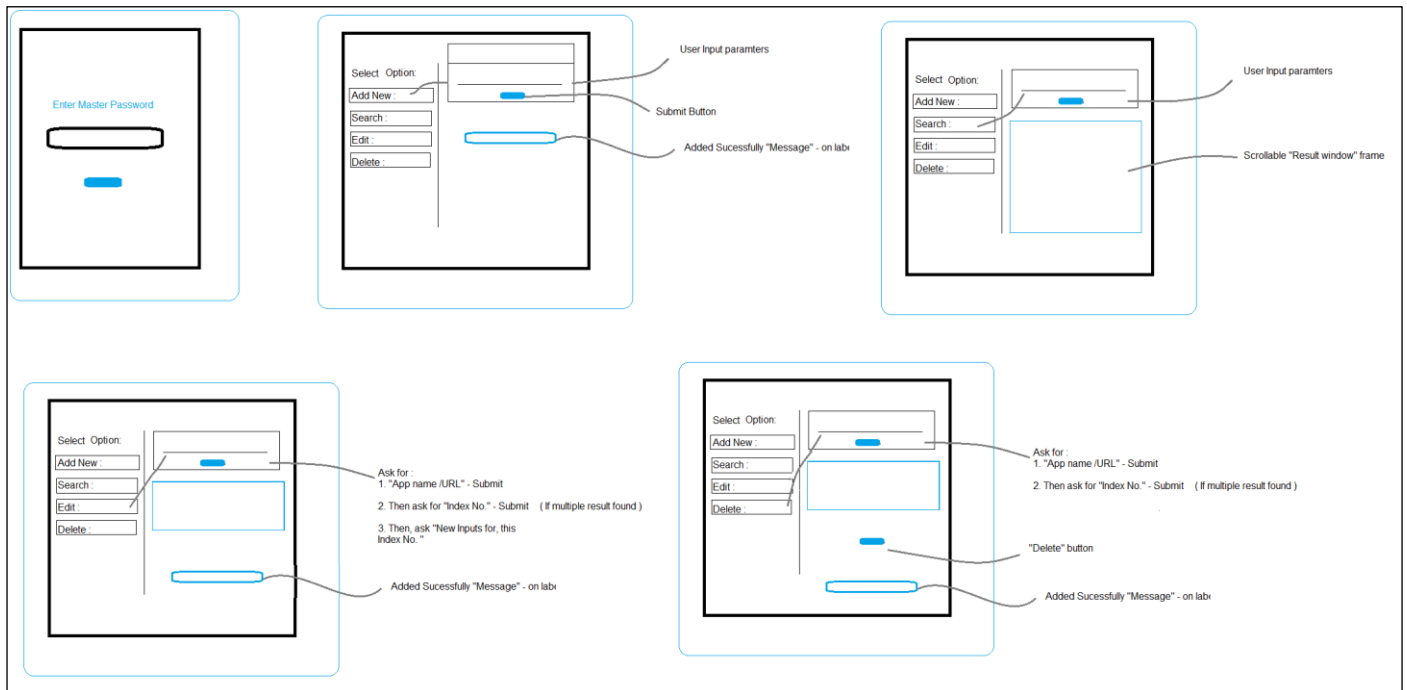
Fig: Proposed User Interface

The interface will be designed to be responsive, ensuring a seamless experience across different screen sizes and devices. This responsiveness is essential for users accessing the credential manager on various platforms (e.g., desktops, laptops).

Development Framework

The User Interface (UI) of the AES-Based Local Credential Manager will be developed using the **CustomTkinter** and **Tkinter** module, a standard Python interface to the Tk GUI toolkit (Tkinter, 2023). Tkinter allows for quick and easy development of window-based applications with essential widgets such as buttons, labels, and entry fields.

8. Data Portability & Backup

Data portability and backup are essential aspects of the AES-Based Local Credential Manager to ensure users can securely manage and safeguard their credentials. While full implementation is yet to be done, several ideas and features have been considered for future development in this area.

8.1 CSV Export/Import
One of the core features under consideration is the ability to **export** and **import** user credentials in **CSV** format. This functionality will allow users to back up their stored credentials to an external file or migrate data across systems in a standardized format. The CSV file will store data in a way that maintains encryption, ensuring sensitive information such as usernames and passwords are not exposed in plaintext during the export process. The import feature will allow users to bring in credentials from another system, ensuring ease of transition while keeping security intact. The design will focus on user convenience without compromising on encryption and safety.

8.2 Auto Backup & Recovery
To enhance the security and reliability of the system, **auto backup and recovery** options will be considered. This feature would enable users to have periodic backups of their credential data, either in encrypted format or as encrypted CSV files.

While these features are still in the planning phase, the goal is to offer users a seamless, secure, and convenient way to manage their credentials with minimal risk of data loss.

10. Challenges, Conclusion & Future Work

As the AES-Based Local Credential Manager project progresses, several challenges have emerged, and the development process has provided valuable insights into the intricacies of secure credential management. The following sections outline the key challenges faced, lessons learned, and plans for future work.

10.1 Implementation Issues & Learnings

During the development of the project, various challenges were encountered, particularly concerning encryption mechanisms, data security, and user interface design. One of the primary difficulties was ensuring robust encryption and decryption functionality that would work consistently with various password complexities. Handling encryption keys securely without compromising the system's integrity proved to be a crucial concern.

Additionally, integrating a **secure backup and import/export** mechanism required careful consideration of data formats, while encoding and decoding text and how to ensure the integrity of encrypted data during these processes.
A major insight was the need for a **zero-knowledge** approach to the system, ensuring that the system does not leak any information about whether the entered master password is correct or incorrect. Implementing this was more complex than initially anticipated.

10.2 Final Thoughts & Impact

The AES-Based Local Credential Manager project has the potential to significantly enhance the security of personal credential management. By combining AES encryption with a zero-knowledge design and a local, user-controlled database, it offers a robust solution to the growing concerns over password security. This system ensures that users have full control over their credentials, with encryption happening entirely on their local device, mitigating the risks of online breaches.
The impact of this project is twofold: it provides a practical solution to the problem of secure password storage, while also fostering a deeper understanding of encryption practices, database design using SQlite, and user interface development in python using Tkinter modules. The system's design philosophy emphasizes simplicity, security, and user empowerment, making it accessible for everyday use while maintaining high security standards.

10.3 Planned Features

Several features and enhancements are planned for the next phases of the project:

- **User Interface Enhancements**: The development of a responsive, user-friendly interface will be a priority. This will involve the use of frameworks like CustomTkinter and Tkinter to ensure a smooth user experience across different screen sizes and devices.
- **Backup and Recovery**: Implementing robust backup and recovery mechanisms will provide users with more control over their data, ensuring they are not at risk of losing their credentials due to hardware failure or accidental deletion.

- **Password Strength Meter**: The inclusion of a password strength meter will assist users in evaluating the strength of their passwords, encouraging them to create more robust passwords. This feature will provide feedback on password complexity, helping users make more informed decisions when setting their passwords.

In conclusion, the AES-Based Local Credential Manager aims to provide a secure, user-centric solution for credential storage, with an emphasis on privacy and usability. The lessons learned and challenges faced during the development will serve as a foundation for improving the system and exploring additional features to meet the evolving needs of users in the realm of secure password management.
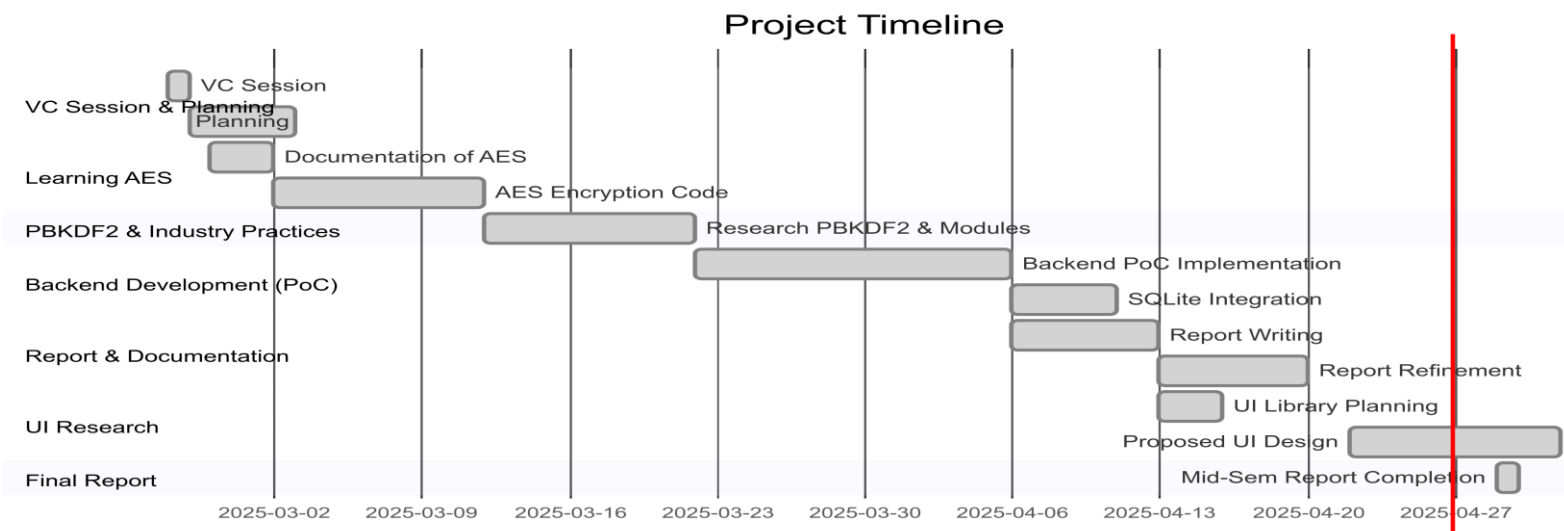
Fig: Gantt Chart - Project Timeline and Progress

11. References

1. **Sommerville, I. (2021)**. *Software Engineering: A Practitioner's Approach* (10th ed.). McGraw-Hill Education.
2. **National Institute of Standards and Technology (NIST)**. (2001). *FIPS PUB 197: Advanced Encryption Standard (AES)*. https://doi.org/10.6028/NIST.FIPS.197.
3. **Python Documentation**
   Python Software Foundation. (2025). *Python 3.11 Documentation*. Retrieved from https://docs.python.org/3/
4. **SQLite Documentation**
   The SQLite Consortium. (2025). *SQLite Documentation*. Retrieved from https://www.sqlite.org/docs.html
5. **PyCryptodome Library**
   Legrange, L. (2025). *PyCryptodome Documentation*. Retrieved from https://www.pycryptodome.org/
6. **CustomTkinter Documentation**
   Tkinter User Group. (2025). *CustomTkinter Library Documentation*. Retrieved

from https://github.com/TomSchimansky/CustomTkinter

7. **PBKDF2**
Eastlake, D., & Jones, P. (2001). *RFC 2898 - PKCS #5: Password-Based Cryptography Specification Version 2.0*. IETF. Retrieved from https://tools.ietf.org/html/rfc2898

8. **AES Encryption**
National Institute of Standards and Technology (NIST). (2001). *FIPS PUB 197 - Advanced Encryption Standard (AES)*. Retrieved from https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf

9. **ZKProof. (2023).** *Zero-Knowledge Proofs*. https://zkproof.org.

10. **Tkinter.** (2023). *Tkinter Documentation*. https://docs.python.org/3/library/tkinter.html.

11. **Software Engineering Practices**
Sommerville, I. (2021). *Software Engineering (10th Edition)*. Pearson Education.

12. **CustomTkinter GitHub Repository**
Schimansky, T. (2025). *CustomTkinter - GitHub Repository*. Retrieved from https://github.com/TomSchimansky/CustomTkinter

13. **Python Cryptography Toolkit (PyCryptodome)**
Legrange, L. (2025). *PyCryptodome Documentation*. Retrieved from https://www.pycryptodome.org/