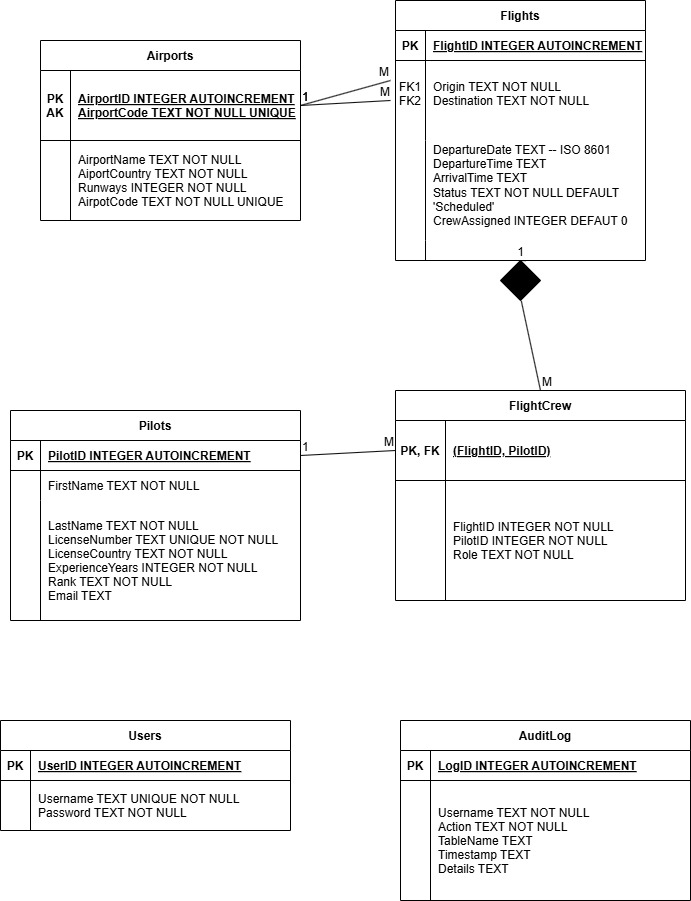
**Entity Relationship Diagram**

Object Based Model, (min, max)



**Table Schema Dictionary:**

Tables = {  
 "Airports": """  
 AirportID INTEGER PRIMARY KEY AUTOINCREMENT,  
 AirportCode TEXT NOT NULL UNIQUE,  
 AirportName TEXT NOT NULL,  
 AirportCountry TEXT NOT NULL,  
 Runways INTEGER NOT NULL  
 """,  
 "Pilots": """  
 PilotID INTEGER PRIMARY KEY AUTOINCREMENT,  
 FirstName TEXT NOT NULL,  
 LastName TEXT NOT NULL,  
 LicenseNumber TEXT UNIQUE NOT NULL,  
 LicenseCountry TEXT NOT NULL,  
 ExperienceYears INTEGER NOT NULL,  
 Rank TEXT NOT NULL,  
 Email TEXT  
 """,  
 "Flights": """  
 FlightID INTEGER PRIMARY KEY AUTOINCREMENT,  
 DepartureDate TEXT, -- ISO 8601 date  
 Origin TEXT NOT NULL,  
 Destination TEXT NOT NULL,  
 DepartureTime TEXT, -- ISO 8601 format: 'HH:MM'  
 ArrivalTime TEXT, -- ISO 8601 format: 'HH:MM'  
 Status TEXT NOT NULL DEFAULT 'Schedule',  
 CrewAssigned INTEGER DEFAULT 0,  
 FOREIGN KEY (Origin) REFERENCES Airports (AirportCode),  
 FOREIGN KEY (Destination) REFERENCES Airports (AirportCode)  
 """,  
 "Users": """  
 UserID INTEGER PRIMARY KEY AUTOINCREMENT,  
 Username TEXT UNIQUE NOT NULL,  
 Password TEXT NOT NULL  
 """,  
 "AuditLog": """  
 LogID INTEGER PRIMARY KEY AUTOINCREMENT,  
 Username TEXT NOT NULL,  
 Action TEXT NOT NULL,  
 TableName TEXT,   
 Timestamp TEXT DEFAULT (datetime('now', 'localtime')),  
 Details TEXT  
 """,  
 'FlightCrew': """   
 FlightID INTEGER NOT NULL,  
 PilotID INTEGER NOT NULL,  
 Role TEXT NOT NULL,  
 PRIMARY KEY(FlightID, PilotID),  
 FOREIGN KEY(FlightID) REFERENCES Flights(FlightID) ON DELETE CASCADE,  
 FOREIGN KEY(PilotID) REFERENCES Pilots(PilotID)  
 """  
}

**Reflective Documentation**

* + - * Explanation of the database structure and the purpose of each table.
      * A description of each SQL query, including how it works and why it’s used.
      * A reflection on the experience, challenges faced, and solutions.
      * Screenshots of the application interacting with the database (e.g., adding a flight, retrieving pilot schedules).

**Structural Overview**

**Airports**

The ‘Airport’s table was created to ensure the database structure adheres to third normal form by isolating airport specific attributes such as Code, Name, Country and Runways. The design decision prevents data redundancy and modification anomalies in the ‘Flights’ table. For example, storing the number of runways in the ‘Flights’ table would introduce the risk of insertion anomalies as inconsistent or incorrect values could be entered during record insertion. While introducing multiple tables and foreign keys increases the complexity of the database design, the reward is that the ‘Airports’ table reduces operational risk and increases the precision of our flight scheduling system.

Selecting an appropriate key for the Airports table was an important design consideration that highlights the natural tension between normalisation and usability. While the table uses an integer-based key (AirportID) as the primary key, it presented challenges in practical system implementation. Users typically think of airports in terms of codes rather than numeric identifiers, and remembering 20 integer/airport combinations became difficult. An alternate key AirportCode was therefore identified for foreign key referencing. Choosing an alternate key requires careful assessment of its volatility, uniqueness, and quality to avoid potential issues such as key changes or duplicate entries. AirportCodes such as ‘JFK’ and ‘LHR’ are semantically meaningful identifiers that users in aviation recognise, but are also unique and stable due to international standardisation (IATA, 2025). This decision to use an appropriate alternate key rather than strictly enforcing the primary key, highlights the importance of balancing database normalisation with practical usability.

**Flights**

The ‘Flights’ table was designed to serve as the central hub of the flight scheduling system, recording critical operational data. Each flight is uniquely identified by an auto-incrementing primary key ‘FlightID’ to facilitate indexing and joins, ensuring the system can retrieve records without relying on complex composite keys.

An important design consideration for the Flights table was the management of crew assignments and maintaining normalisation. Embedding pilot information directly into the ‘Flights’ table via a ‘PilotID’ field would create a one-to-many relationship between pilots and flights when adhering to requirements of atomicity. This relationship, however, contradicts the real world many-to-many relationship of multiple pilots per flight. A separate ‘FlightCrew’ table was implemented referencing ‘FlightID’ as a foreign key, thus enforcing normalisation and providing flexibility in assigning multiple pilots (e.g. Captain, First Officer) to a single flight.

While adding ‘FlightCrew’ enabled a many-to-many relationship, it increased complexity and abstracted away from the ‘Flights’ table the direct association between a flight and its assigned crew. A custom report could be created using a join between tables to show users relevant contextual information of a crew’s attributes per flight. This approach, however, reduces the useability of the Flight table. Simply, if you work for an airline and you check today’s flight schedule via your primary view (the Flights table), you want to know immediately that there is a flight crew on the plane without having to check another source.

From a useability perspective the ‘Flight’s table must overcome two challenges in presenting crew assignments. First, a flight can be scheduled months in advanced with no crew assigned in the real world. The ‘Flights’ table cannot have a CrewID field, or it would violate First Normal Form of atomicity. Second, users of the Flights table are typically more concerned with the number of pilots assigned to a flight rather than with an integer pointer to a separate table of crew assignments.

To balance normalisation with usability, the ‘CrewAssigned’ field was added to the ‘Flights’ table with an integer default value 0 as a de-normalised indicative field derived logically from another table without a join. Its value indicates the count of pilots assigned to each flightID, and is updated from the ‘FlightCrew’ table to the ‘Flights’ table through a database trigger (code 1) on inserts and deletes. This design allows users to continue querying the main flight table without needing a custom join report, but importantly retains an accurate present status of crew assignments.

Code 1

@staticmethod  
def sql\_create\_crew\_trigger():  
 return """CREATE TRIGGER IF NOT EXISTS trg\_update\_crew\_assigned  
 AFTER INSERT ON FlightCrew  
 BEGIN  
 UPDATE Flights  
 SET CrewAssigned = (  
 SELECT COUNT(\*)  
 FROM FlightCrew  
 WHERE FlightID = NEW.FlightID  
 )  
 WHERE FlightID = NEW.FlightID;  
 END;"""

**FlightCrew**

The ‘FlightCrew’ table enables normalisation of the many-to-many relationship between pilots to flights whilst having itself a one-to-many relationship between both these tables. FlightCrew references foreign keys ‘FlightID’ from ‘Flights’ and ‘PilotID’ from ‘Pilots’ with a composite primary key of FlightID, PilotID.

The table has two key controls to maintain operational consistency. First, a pilot cannot fly two planes simultaneously and a control trigger was created on the FlightCrew table that blocks these inserts. The trigger (code 2) assumes only that a pilot cannot be assigned to a flightID on the same departure date as an existing FlightCrew entry. Whilst this solves a serious problem, it is not exhaustive and shows a limitation that database design must overcome. For example, further logic should be created that checks a pilot’s scheduled location to prevent inserts where a pilot is not physically present. To facilitate this control, the Pilots table could be expanded with prescribed airport routes combinations and a trigger added when an origin/destination violates their route and current location.

***Code 2***

@staticmethod  
def sql\_no\_double\_bookings():  
 return """CREATE TRIGGER IF NOT EXISTS trg\_no\_double\_booking  
 AFTER INSERT ON FlightCrew  
 BEGIN  
 -- Check for conflicting flights  
 SELECT  
 CASE  
 WHEN (  
 SELECT COUNT(\*)  
 FROM FlightCrew fc  
 JOIN Flights f1 ON fc.FlightID = f1.FlightID  
 WHERE  
 fc.PilotID = NEW.PilotID  
 AND f1.DepartureDate = (  
 SELECT DepartureDate  
 FROM Flights  
 WHERE FlightID = NEW.FlightID  
 )  
 AND fc.FlightID != NEW.FlightID  
 ) > 0  
 THEN

Second, the reference to ‘FlightID’ has the ON DELETE CASCADE condition from the ‘Flights’ table but not from the ‘Pilots’ table. By ensuring deletion across both Flights and FlightCrew, an individual Pilot returns to a ‘free’ state, allowing them to be assigned to a new flight. This appropriately conforms to the code 2 trigger constraint to enable efficient user interaction.

The ON DELETE CASCADE clause was omitted from the Pilots table to facilitate record keeping, but presents future challenges to the database design. Deletion of a pilot from the Pilots table should not impact the record retention of completed scheduled flights. The Federal Aviation Administration (insert here) requires that flight records are retained for 3 years, during which time a pilot may cease service with the flight management company. The limitation of the Icarus system here is that deleting a pilot from the Pilots table, whilst maintains their service record, removes all relevant contact information other than their PilotID. This identifies a challenge on where to maintain retired pilot information. The Pilots table should be logically focused on *available* pilots for scheduling. Potentially a combination of a former employee’s table, a join report from this table and the past flight records, and replacing the primary key PolitID with an alternate key license registration number could all address this challenge.

**Pilots**

Similarly to the Airports table, the Pilots table ensures the Flights table adheres to normal form. A unique integer primary key PilotID is created with auto increment to facilitate relationship foreign keys. This abstracts pertinent personal pilot information from flights whilst maintaining a record for the business use cases.

A limitation of this design is the table does not protect pilot personal information anonymity. All users of the management system can view pilot information such as their email address and license number. Whilst the Users table discussed later addresses system access security concerns, no protection is provided for unauthorised access. A cipher system ideally should have been applied to protect pilot data from unauthorised retrieval. For example, by using the existing transaction manager function, inserted records could be encrypted into the table, an encryption key created, and a decryption algorithm applied when records are retrieved. Whilst this increases the database management system complexity, the database access object class was created with a transaction manager as a foundation to build in future functionality.

**Users**

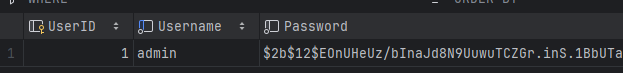
The Users table is intentionally lightweight, containing the primary key UserID, username and a hashed, encoded, and salted password to enforce appropriate system access security controls and support audit logging. Access to the system is restricted through an authentication process that confirms username and password in the command line (Code 5).

No roles are prescribed currently. The admin user can view the User table including their own encrypted password. Both the table and the User class should be expanded to accommodate roles and their relevant access level, and the table should be removed from the CLI menu.

Users have no means to add themselves to the User table. If necessary, the database manager could wrap the add\_user method (Code 3) in the database access object class in a prompt through the CLI class. More likely in a production environment, this table would be integrated with an enterprise authentication system for centralised single sign on management.

For demonstration: dao.add\_user("admin", "Password123")

**Figure 1**



**Code 3**

def add\_user(self, username, password):  
 def operation(conn):  
 hashed\_password = bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt())  
 cursor = conn.cursor()  
 statement = qb.get\_sql\_add\_user() #Code 4 <<  
 cursor.execute(statement, (username, hashed\_password.decode('utf-8')))  
 print(f"Successfully added user {username}.")  
 return self.transaction\_wrapper(operation)

**Code 4**

@staticmethod  
def get\_sql\_add\_user():  
 return f"""  
 INSERT INTO Users (Username, Password) VALUES (?, ?)  
 """

**Code 5**

def check\_credentials(self, username, password):  
 def operation(conn):  
 cursor = conn.cursor()  
 sql\_statement = qb.get\_sql\_user()  
 cursor.execute(sql\_statement, (username,))  
 result = cursor.fetchone()  
 if result:  
 stored\_hash = result[0]  
 if bcrypt.checkpw(password.encode('utf-8'), stored\_hash.encode('utf-8')):  
 return True  
 return False  
 return self.dao.transaction\_wrapper(operation)

**AuditLog**

The AuditLog table provides a security journal of all Data Manipulation Language queries (insert, delete, update), usernames, impacted table names, timestamp, details and a LogID as primary key.

The AuditLog provides useful information for both managing security threats and user accountability. Following NIST SP 800-53 guidance (<https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-53r5.pdf>) event logging is crucial as a management strategy for data governance. Whilst Icarus is a light weight locally stored database, a flight scheduling system for an industry notorious for safety regulation should attempt to adhere to contextual constraints.

The AuditLog records are provided unencrypted to all logged in users of the Icarus flight management system. Should the application become enterprise level, these decisions should be assessed against a documented security policy for the company as to whether appropriate data access level are in force.

**SQL Query Descriptions**

The SQL queries created were designed to prevent SQL injection attacks via parameterisation whilst being reusable to multiple functions. They are dynamic, with the QueryBuilder class constructing SQL queries based on CLI input that has been manipulated by both the logiclayer class and the Database Access Object class. For example, if user inputs ‘1’ on specific menu, they refer to table ‘Airports’, which gets converted, and eventually sent to the QueryBuilder class to conversion to SQL before being executed by the DAO class.

**Create tables**

***Code 6***

@staticmethod  
def get\_sql\_create\_table(table\_name, schema):  
 return f"CREATE TABLE IF NOT EXISTS {table\_name} ({schema}) STRICT"

Contextually free from SQL injection, this function is called during program initialisation and uses the tables dictionary in the Schema class. Tables names are dictionary keys, and fields are contained in the values. Sqlite3 requires the STRICT condition to enforce field type constraints. We run the create tables on every startup hence IF NOT EXISTS.

**Add data**

***Code 7***

@staticmethod  
def get\_sql\_insert\_statement(table\_name, columns, placeholders):  
 return f"""  
 INSERT INTO {table\_name} ({','.join(columns)})  
 VALUES ({placeholders})  
 """

This function adheres to parametrisation relying on code 8 to place “?” placeholders dynamically for len(columns). Table\_name and columns are full string values such that “’x;’ ‘1=1;’” is one statement and would throw an error that it is not a valid column or table name. table\_name is not a dictionary but is a string value obtained from the CLI. Users input an integer value that returns an associate table\_name string.

***Code 8***

@staticmethod  
def get\_placeholders(columns):  
 return ",".join(['?'] \* len(columns))

**Search or Delete**

Called by the DAO object with arguments obtained from the CLI. Users can only custom input value in the criteria tuple which forms the WHERE statement. Criteria is optional to allow “SELECT \* FROM y” . This code calls Code 10 to form the WHERE statement.

***Code 9***

@staticmethod  
def get\_sql\_select\_delete(table\_name, column\_name, criteria=None, action='SELECT'):  
 *"""  
 Constructs a parameterised SQL SELECT or DELETE statement for the specified table.  
  
 Args:  
 table\_name (str): Name of the table to query.  
 column\_name (optional str or list): Column(s) to select, empty ‘’ on delete commands.  
 criteria (list of tuple, optional): A list of 4-tuple specifying WHERE clause conditions.  
 Each tuple should be in the format:  
 (logical\_operator, column, operator, value)  
 where:  
 logical\_operator (str): Optional; can be 'AND' or 'OR' (first condition uses an empty string '').  
 column (str): Column name to filter on.  
 operator (str): Comparison operator (e.g. '=', '<', '>', '<>').  
 value (str): Value to compare against.  
 action (str, optional): Either 'SELECT' or 'DELETE'. Defaults to 'SELECT'.  
  
 Returns:  
 tuple: A tuple containing the SQL statement (str) and a list of parameter values.  
 """* if isinstance(column\_name, list):  
 columns\_str = ', '.join(column\_name)  
 else:  
 columns\_str = column\_name  
  
 sql\_statement = f"{action} {columns\_str} FROM {table\_name}"  
  
 values = []  
 if criteria:  
 sql\_statement, values = QueryBuilder.get\_sql\_where(criteria, sql\_statement)  
 return sql\_statement, values

**Code 10**

Takes statements and adds parameter placeholders ? for each value key from a 4 tuple

#use paramatisation to defend from sql injection  
@staticmethod  
def get\_sql\_where(criteria, sql\_statement):  
 values = []  
 where\_clause = []  
 for row in criteria:  
 where\_clause.append(f"{row[0]} {row[1]} {row[2]} ?")  
 values.append(row[3])  
  
 sql\_statement += " WHERE "  
 sql\_statement += "".join(where\_clause)  
 return sql\_statement, values

**Update**

Code 11 Similar to Add/Delete but broken into separate function for better separation of concerns.

Table\_name string obtained from CLI integer input converted to string value. Change is from the CLI, and converted to string e.g. j'; DROP TABLE X; would throw an error. Again, the parametrisation function code 10 for WHERE statements is called. Note users can update multiple fields in the same statement but only a single table. Users are forced to input WHERE conditions, notice criteria is non optional vs optional in Code 9 to prevent mass unintended changes.

**Code 11**

@staticmethod  
def get\_sql\_update(table\_name, change, criteria):  
 sql\_statement =f"""UPDATE {table\_name} SET {change}"""  
 values = []  
 if criteria:  
 sql\_statement, values = QueryBuilder.get\_sql\_where(criteria, sql\_statement) #<< Code 10  
 return sql\_statement, values

**Add to audit log**

**Code 12**

Tightly coupled with the current AduitLog schema. Whilst easier to interpret, will fail if AuditLog schema is modified. Keeps parametrisation equally static and does not call code 10.

@staticmethod  
def get\_sql\_transaction\_log():  
 return f"""  
 INSERT INTO AuditLog (Username, Action, TableName, Details)  
 VALUES (?, ?, ?, ?)  
 """

For brevity, we skip to ‘boilerplate’ SQL statements for custom reports

**Get pilot schedule**

Business view to return pertinent schedule information by unique PilotID.

Requires 4 joins  
1: Flightcrew schedule information joined to flights table by flightID to allow access to Role   
2: Airport table joined on flights table via Code and Origin to access to AirportName  
3: repeat on different Airport table to Code and Destination on flights for Airport Name  
4: PilotID from flightcrew to Pilots table to access PilotName composite field

**Code 13**

@staticmethod  
def get\_sql\_pilot\_schedule():  
 sql\_statement = f"""  
 SELECT  
 p.FirstName || ' ' || p.LastName AS PilotName,  
 fc.Role,  
 f.FlightID,  
 f.DepartureDate,  
 f.DepartureTime,  
 ao.AirportCode AS OriginCode,  
 ao.AirportName AS OriginName,  
 af.AirportCode AS DestinationCode,  
 af.AirportName AS DestinationName,  
 f.Status  
 FROM FlightCrew fc  
 JOIN Flights f ON fc.FlightID = f.FlightID  
 JOIN Airports ao ON f.Origin = ao.AirportCode  
 JOIN Airports af ON f.Destination = af.AirportCode  
 JOIN Pilots p ON fc.PilotID = p.PilotID  
 WHERE fc.PilotID = ?  
 ORDER BY f.DepartureDate, f.DepartureTime;  
 """  
 return sql\_statement

**Summarise number of flights by Destination or Pilot**

Count the number of flightIDs in the flight table, join flights and airports table through Destination/AirportCode, group the information by destination descending order.

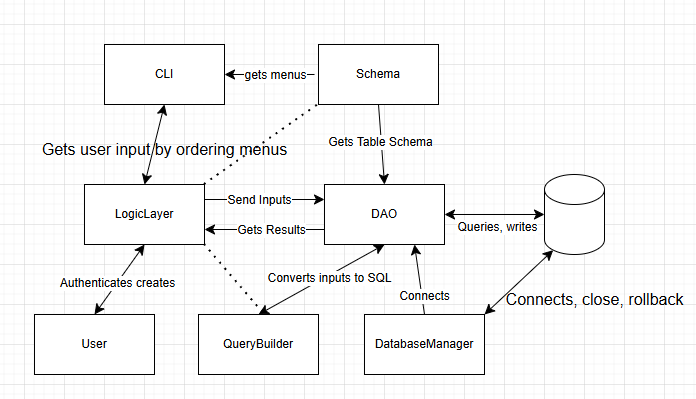
**Code 14**

@staticmethod  
def get\_sql\_number\_of\_flights(GroupBy):  
 if GroupBy == 'Destination':  
 return f"""  
 SELECT   
 a.AirportCode,  
 a.AirportName,  
 COUNT(f.FlightID) AS NumberOfFlights  
 FROM Flights f  
 JOIN Airports a ON f.Destination = a.AirportCode  
 GROUP BY f.Destination  
 ORDER BY NumberOfFlights DESC;  
 """  
 elif GroupBy == 'Pilot':  
 return f"""  
 SELECT   
 p.FirstName || ' ' || p.LastName AS Pilot,  
 p.PilotID,  
 p.LicenseNumber,  
 COUNT(fc.FlightID) AS NumberOfFlights  
 FROM FlightCrew fc  
 JOIN Pilots p ON fc.PilotID = p.PilotID  
 GROUP BY fc.PilotID  
 ORDER BY NumberOfFlights DESC;  
 """

**Reflection**

The overall architectural design of the Icarus system presented additional challenges beyond table schema decisions. The system was encapsulated into several classes to separate concerns, but became dramatically complex.

Figure 2



Further work is required to remove the relationship between the logiclayer and the query builder. The DAO class could likely be extended into a child class that focuses on database initialisation.

Transaction management was addressed through the DAO and DatabaseManager classes which increased complexity. Whilst SQLite3 supports multiple user read concurrency through shared locks on tables, the Icarus system established a transaction manager function that would connect/commit or rollback/close quickly for each DDL operation.

Transaction management in the Icarus system was implemented through the DAO and DatabaseManager classes, which coordinate all database operations. This design centralises control over data access and enforces consistent commit or rollback behaviour across different queries.

While SQLite3 natively supports concurrent read operations using shared locks, write operations (INSERT, UPDATE, DELETE) acquire exclusive locks, blocking other transactions from modifying the data until the transaction completes. This highlights the importance of a well-designed transaction manager to ensures atomicity and consistency, core principles of the ACID model recommended by **xxx** for secure data management (book 520).

The Icarus system uses a transaction manager function that in the DAO that quickly calls the DatabaseManager for connects, commits, or rolls back and closes after each DML (Data Manipulation Language) operation. It has error handling for effective system recovery, and provides messaging to the user through the command line. This ensures that each operation is isolated, that the system maintains data integrity in cases of errors or unexpected failures, and the user is informed what went wrong.

In an effort to reuse code, the transaction wrapper is used even on SELECT queries, and thus commits nothing unnecessarily and provides a commit message on each SELECT. Further work can be made to address this issue to improve atomicity.

**Code 15**

def transaction\_wrapper(self, operation):  
 conn = self.db\_manager.connect()  
 if not conn:  
 print("No database connection.")  
 return  
  
 result = None  
 try:  
 result = operation(conn)  
 conn.commit()  
 print("Operation Complete.")  
 except Exception as e:  
 print(f"Error performing operation: {e}")  
 self.db\_manager.rollback(conn)  
 finally:  
 self.db\_manager.close(conn)  
 return result

**Parametrisation**

**Security**

Architecture, Seperating concerns, modularity, security, paramtisation, transaction management