A MINI PROJECT REPORT

In partial fulfillment for the requirement for the award of degree of

BACHELOR OF TECHNOLOGY

In

COMPUTER SCIENCE AND ENGINEERING

By

D.MEGHANA	2403A510D9
R.VARSHITH	2403A510D7
CH.HARINI	2403A510E1
G.SUDEEKSHA	2403A510E0

Under the guidance

Of

Brij kishor

Department of Computer Science and Engineering

SR university

H-NO 3-140,Ananthsagar,Hasanparthy,Telangana 506371

Project 5 — Database Migration Assistant

Title

Database Migration Assistant — Migration DDL, Testing & Rollback Plan

Abstract

This project demonstrates how to draft and apply SQL schema migrations, test them on SQLite, and prepare reliable backups and rollback strategies. It includes sample migration DDL, testing steps, backup approaches, and an AI-suggested rollback plan to safely revert schema and data changes.

Introduction

Database schema changes are essential to evolve applications but carry risk: data loss, downtime, and incompatibilities. A disciplined migration process with clear DDL, repeatable tests, backups, and well-defined rollback procedures reduces risk and enables teams to deploy changes confidently. This document walks through a small but realistic migration example, shows how to test it on SQLite, and provides robust rollback guidance.

Objectives of the Project

- Draft clear migration DDL for evolving an existing schema.
- Test migrations locally using SQLite and verify data integrity.
- Create backup and rollback strategies (automated and manual options).
- Document the migration flow, methods used, work completed, and references.

Flowchart

```
flowchart TD
  A[Start]
  B[Prepare migration DDL]
  C[Backup current DB]
  D[Apply migration on test SQLite DB]
```

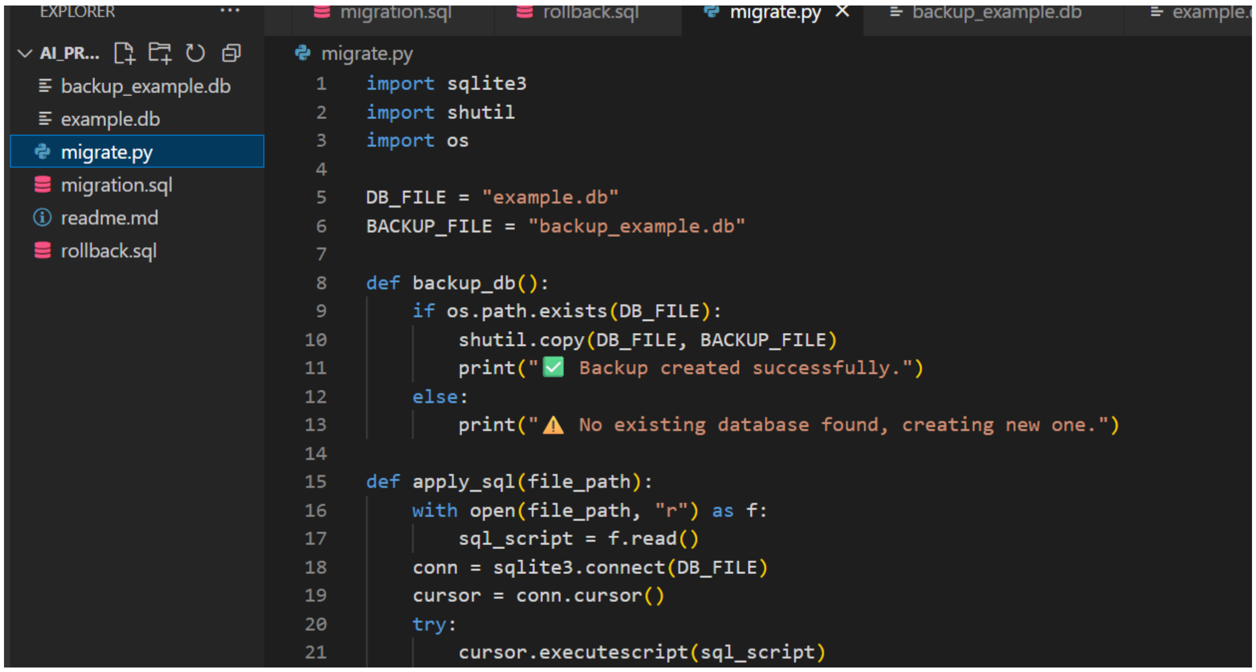
```
E[Run data & application tests]
F[Push to staging]
G[Run staging tests]
H[Deploy to production]
I[Monitor & verify]
J[Rollback if needed]

A-->B-->C-->D-->E
E-- pass -->F-->G-- pass -->H-->I
E-- fail -->J
G-- fail -->J
I-- issue -->J
```

Method Used

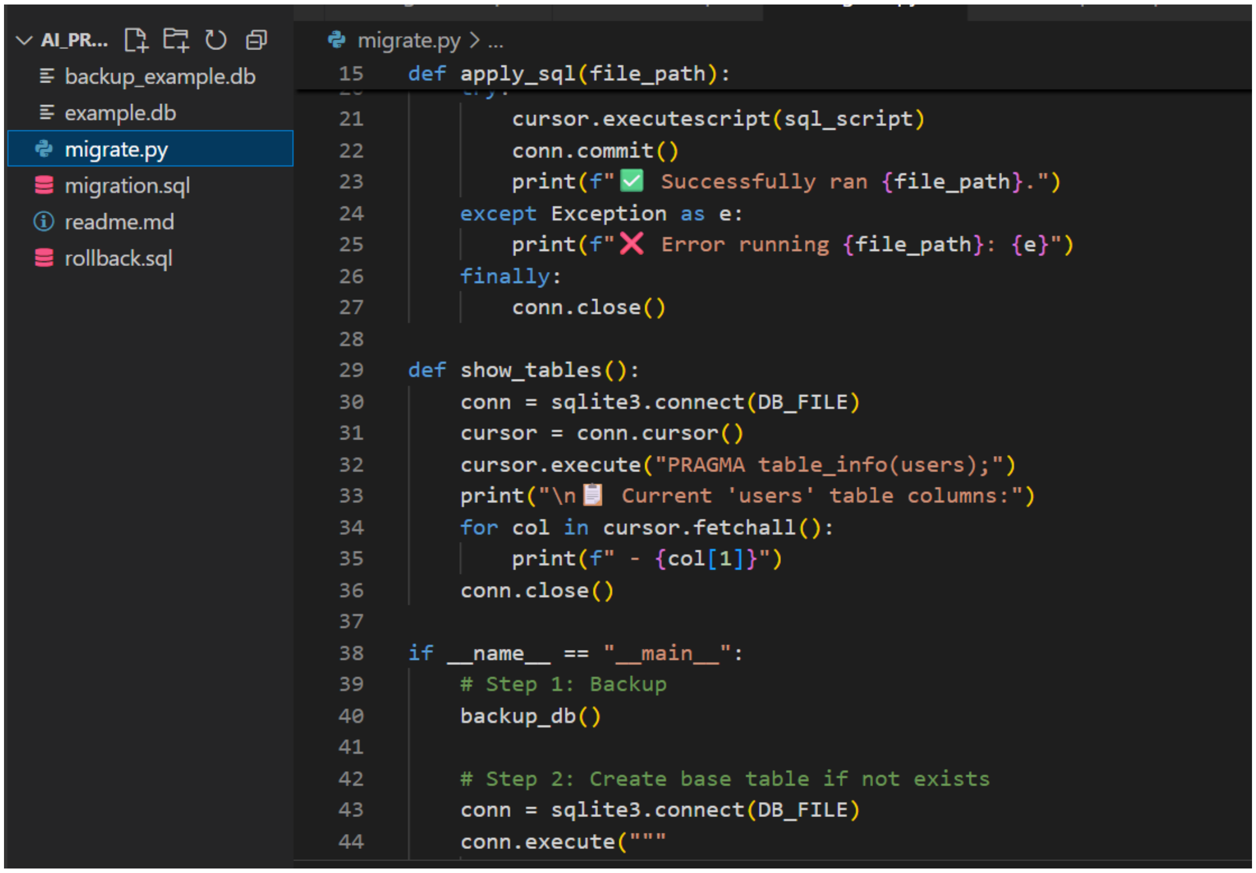
1. **Schema-first migration**: write explicit DDL statements that transform the schema step-by-step.
2. **Safe migration patterns**: add columns with NULL default, backfill data, switch application code to use new columns, then drop old columns in a later migration.
3. **Test-driven**: create unit/integration tests and run them after applying migrations on a local SQLite copy.
4. **Backup before change**: create full SQL dumps and file-level backups where applicable.
5. **Transactional or staged rollout**: where supported, use transactions or toggles (feature flags) to minimize risk.

Code:

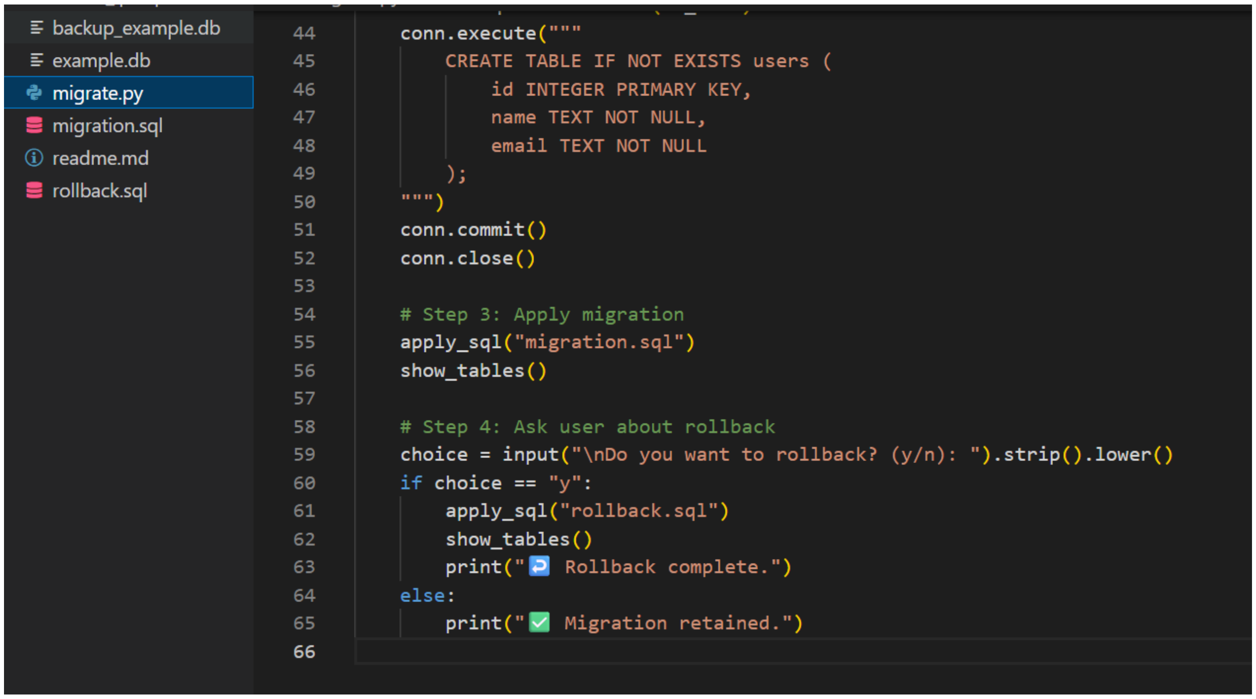
MIGRATE.PY



```python
import sqlite3
import shutil
import os

DB_FILE = "example.db"
BACKUP_FILE = "backup_example.db"

def backup_db():
    if os.path.exists(DB_FILE):
        shutil.copy(DB_FILE, BACKUP_FILE)
        print("✅ Backup created successfully.")
    else:
        print("⚠ No existing database found, creating new one.")

def apply_sql(file_path):
    with open(file_path, "r") as f:
        sql_script = f.read()
    conn = sqlite3.connect(DB_FILE)
    cursor = conn.cursor()
    try:
        cursor.executescript(sql_script)
```

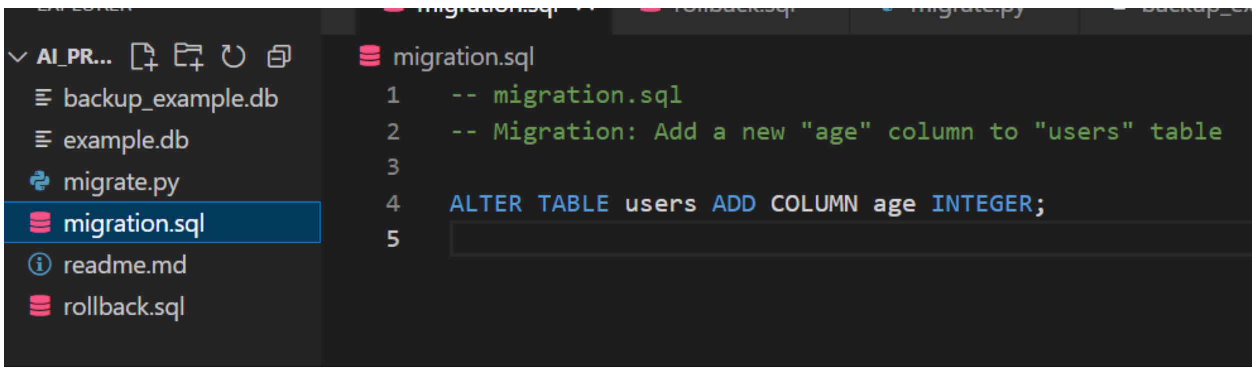


```python
def apply_sql(file_path):
            cursor.executescript(sql_script)
            conn.commit()
            print(f"✅ Successfully ran {file_path}.")
        except Exception as e:
            print(f"❌ Error running {file_path}: {e}")
        finally:
            conn.close()

def show_tables():
    conn = sqlite3.connect(DB_FILE)
    cursor = conn.cursor()
    cursor.execute("PRAGMA table_info(users);")
    print("\n📋 Current 'users' table columns:")
    for col in cursor.fetchall():
        print(f" - {col[1]}")
    conn.close()

if __name__ == "__main__":
    # Step 1: Backup
    backup_db()

    # Step 2: Create base table if not exists
    conn = sqlite3.connect(DB_FILE)
    conn.execute("""
```



```python
    conn.execute("""
        CREATE TABLE IF NOT EXISTS users (
            id INTEGER PRIMARY KEY,
            name TEXT NOT NULL,
            email TEXT NOT NULL
        );
    """)
    conn.commit()
    conn.close()

    # Step 3: Apply migration
    apply_sql("migration.sql")
    show_tables()

    # Step 4: Ask user about rollback
    choice = input("\nDo you want to rollback? (y/n): ").strip().lower()
    if choice == "y":
        apply_sql("rollback.sql")
        show_tables()
        print("↩️ Rollback complete.")
    else:
        print("✅ Migration retained.")

```

Migration.sql



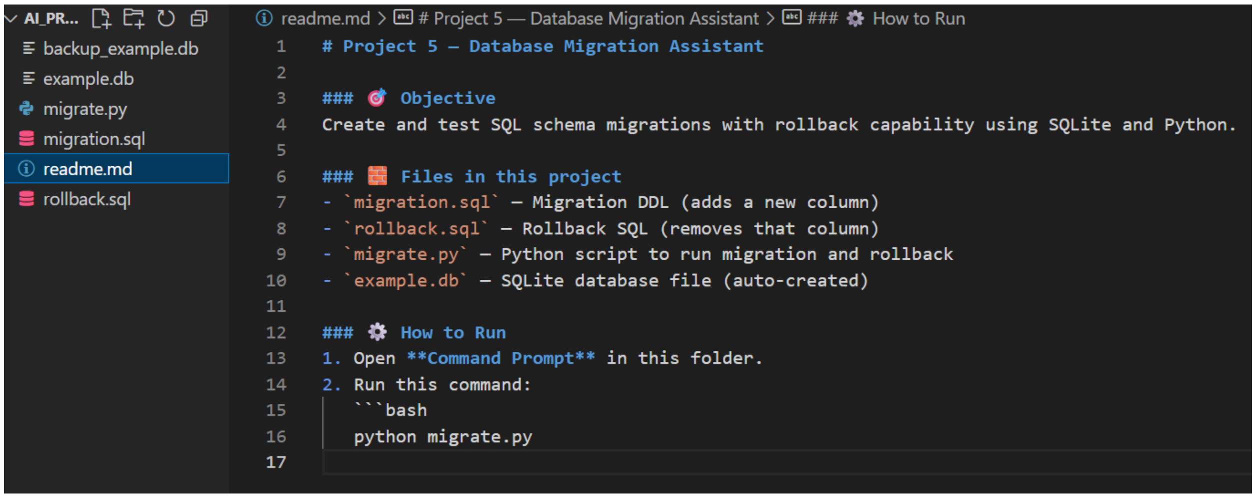
AI_PR...

- backup_example.db
- example.db
- migrate.py
- migration.sql
- readme.md
- rollback.sql

migration.sql

```sql
-- migration.sql
-- Migration: Add a new "age" column to "users" table

ALTER TABLE users ADD COLUMN age INTEGER;

```

README.MD



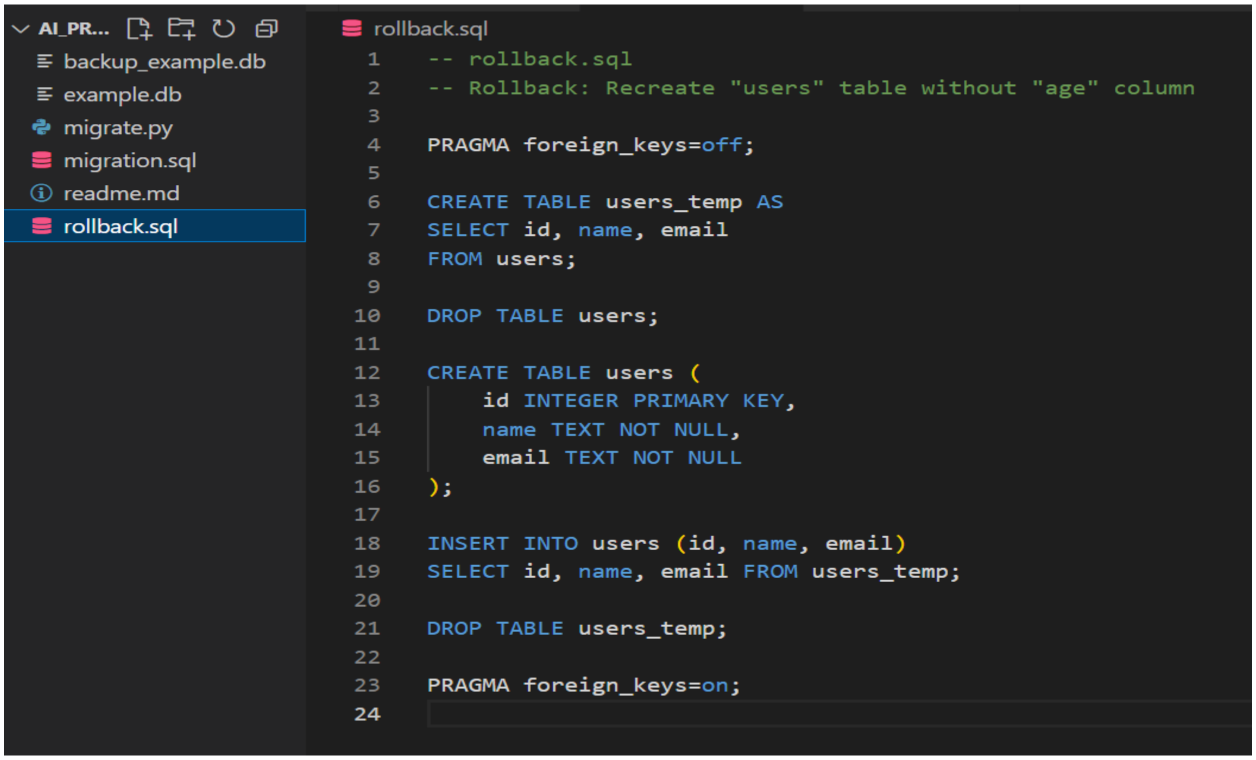
AI_PR...

- backup_example.db
- example.db
- migrate.py
- migration.sql
- readme.md
- rollback.sql

readme.md > # Project 5 — Database Migration Assistant > ### ⚙️ How to Run

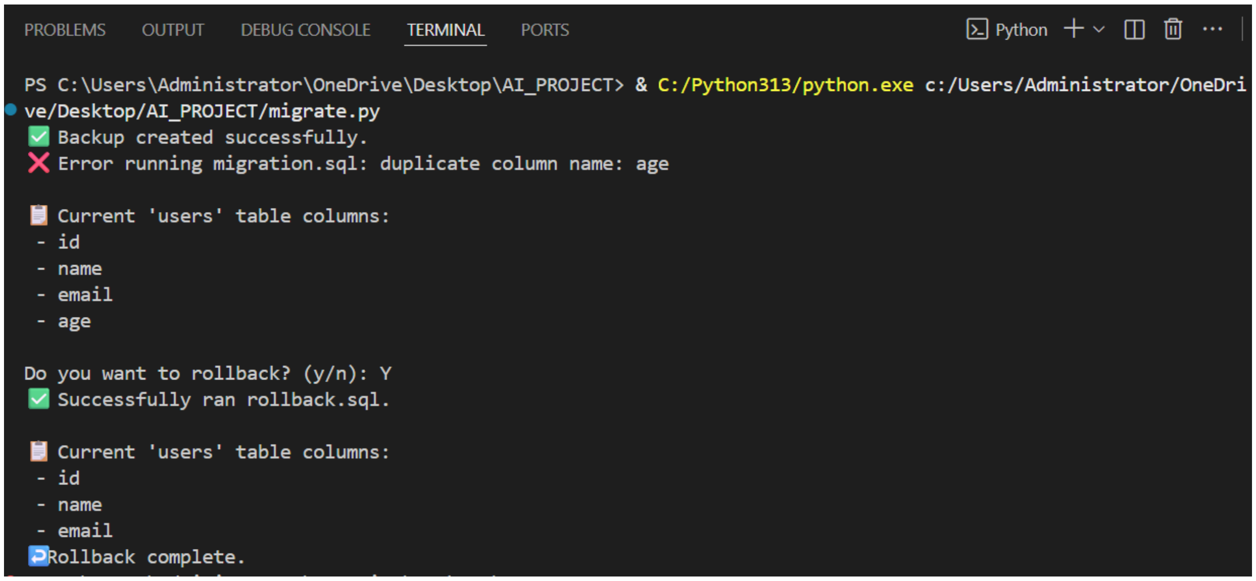
~~~
# Project 5 — Database Migration Assistant

### 🎯 Objective
Create and test SQL schema migrations with rollback capability using SQLite and Python.

### 🧱 Files in this project
- `migration.sql` — Migration DDL (adds a new column)
- `rollback.sql` — Rollback SQL (removes that column)
- `migrate.py` — Python script to run migration and rollback
- `example.db` — SQLite database file (auto-created)

### ⚙️ How to Run
1. Open **Command Prompt** in this folder.
2. Run this command:
   ```bash
   python migrate.py

~~~

ROLLBACK.SQL



```sql
-- rollback.sql
-- Rollback: Recreate "users" table without "age" column

PRAGMA foreign_keys=off;

CREATE TABLE users_temp AS
SELECT id, name, email
FROM users;

DROP TABLE users;

CREATE TABLE users (
    id INTEGER PRIMARY KEY,
    name TEXT NOT NULL,
    email TEXT NOT NULL
);

INSERT INTO users (id, name, email)
SELECT id, name, email FROM users_temp;

DROP TABLE users_temp;

PRAGMA foreign_keys=on;
```

OUTPUT:-



```
PS C:\Users\Administrator\OneDrive\Desktop\AI_PROJECT> & C:/Python313/python.exe c:/Users/Administrator/OneDri
ve/Desktop/AI_PROJECT/migrate.py
✅ Backup created successfully.
❌ Error running migration.sql: duplicate column name: age

📋 Current 'users' table columns:
 - id
 - name
 - email
 - age

Do you want to rollback? (y/n): Y
✅ Successfully ran rollback.sql.

📋 Current 'users' table columns:
 - id
 - name
 - email
↩Rollback complete.
```

Work Done (Step-by-step)

Below is a realistic example showing an initial schema, a set of schema changes, migration DDL, testing steps on SQLite, and an AI-suggested rollback plan.

1) Initial schema (example)

```
-- users table: initial
CREATE TABLE users (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  email TEXT NOT NULL UNIQUE,
  name TEXT NOT NULL,
  created_at DATETIME DEFAULT (CURRENT_TIMESTAMP)
);

-- orders table: initial
CREATE TABLE orders (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  user_id INTEGER NOT NULL,
  amount_cents INTEGER NOT NULL,
  status TEXT NOT NULL DEFAULT 'pending',
  created_at DATETIME DEFAULT (CURRENT_TIMESTAMP),
  FOREIGN KEY (user_id) REFERENCES users(id)
);
```

2) Desired changes

- Add phone column to users (nullable initially).
- Rename `amount_cents` to `amount` and convert integer cents to decimal dollars (store as REAL).
- Add an index on `orders(user_id, status)`.
- Create `order_history` table to archive order status changes.

3) Migration DDL (forward)

Note: SQLite has limited `ALTER TABLE` support. Some operations require creating a new table, copying data, dropping the old table, and renaming the new one.

```
-- 1. Add phone column to users (safe, nullable)
ALTER TABLE users ADD COLUMN phone TEXT;

-- 2. Create new orders table with `amount` as REAL
CREATE TABLE orders_new (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  user_id INTEGER NOT NULL,
  amount REAL NOT NULL,
  status TEXT NOT NULL DEFAULT 'pending',
  created_at DATETIME DEFAULT (CURRENT_TIMESTAMP),
  FOREIGN KEY (user_id) REFERENCES users(id)
);

-- 3. Copy & convert data: cents -> dollars
INSERT INTO orders_new (id, user_id, amount, status, created_at)
SELECT id, user_id, CAST(amount_cents AS REAL)/100.0, status,
created_at
FROM orders;

-- 4. Create order_history table
CREATE TABLE order_history (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  order_id INTEGER NOT NULL,
  old_status TEXT NOT NULL,
  new_status TEXT NOT NULL,
  changed_at DATETIME DEFAULT (CURRENT_TIMESTAMP),
  FOREIGN KEY (order_id) REFERENCES orders(id)
);

-- 5. Drop old orders, rename new
DROP TABLE orders;
ALTER TABLE orders_new RENAME TO orders;

-- 6. Create index on orders(user_id, status)
CREATE INDEX idx_orders_user_status ON orders(user_id, status);
```

For other RDBMS (Postgres/MySQL) some steps would use `ALTER TABLE RENAME COLUMN` and `ALTER TYPE` commands.

4) Test on SQLite (commands)

1. Create a local copy of the database: `cp prod.db test_migration.db` (or export SQL dump).
2. Open sqlite CLI: `sqlite3 test_migration.db`.
3. Run migration SQL (source the file): `.read migration_forward.sql`.
4. Verify schema changes:
   a. `PRAGMA table_info(users);`
   b. `PRAGMA table_info(orders);`
   c. `SELECT count(*) FROM order_history;`
5. Validate data correctness:
   a. Check sample amounts: `SELECT id, amount, amount*100 AS cents FROM orders LIMIT 10;`
   b. Verify no rows lost: `SELECT count(*) FROM orders;` vs original
6. Run application tests pointing to `test_migration.db` and run integration test suite.

5) Backup Strategy (before applying migration)

- **SQLite**: copy the file (`cp prod.db prod.db.bak.YYYYmmddHHMMSS`) and/or use `.backup` from sqlite3:
  - `sqlite3 prod.db ".backup 'prod.db.bak'"`
- **Server RDBMS**: take logical dumps and hot backups:
  - Postgres: `pg_dump -Fc -f prod_YYYYmmdd.dump mydb`

    - MySQL: `mysqldump --single-transaction --routines --triggers -u user -p db > prod.sql`
- **Point-in-time recovery**: enable WAL/archiving if supported and retain enough WAL segments.
- **Export critical tables** to CSV as an extra precaution.

6) AI-suggested Rollback Plan

This plan assumes you have a backup and that the migration is non-destructive for critical data (or you have exported critical tables).

Rollback steps (high-level):

1. **Stop writes** to the production DB (put the app in read-only/maintenance mode) to prevent data drift.
2. **Take a fresh backup** of the current (post-migration) DB for forensic purposes.
3. **Decide rollback strategy**:
    a. **Fast revert (schema-only, minimal data loss risk):** If migration added only nullable columns, indexes, or created new tables, you can revert by running reverse DDL (drop new tables, drop indexes, drop columns where safe).
    b. **Full restore:** If migration mutated or dropped important columns/data (e.g., destructive `DROP TABLE` or type conversions that lost precision), restore from backup and replay accepted writes, or re-apply forward migration after data repair.
4. **Reverse DDL (example)**

```
-- Drop index
DROP INDEX IF EXISTS idx_orders_user_status;

-- If order_history is empty or safe to drop
DROP TABLE IF EXISTS order_history;

-- To revert orders conversion: create old orders table and convert
back
CREATE TABLE orders_old (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  user_id INTEGER NOT NULL,
  amount_cents INTEGER NOT NULL,
  status TEXT NOT NULL DEFAULT 'pending',
  created_at DATETIME DEFAULT (CURRENT_TIMESTAMP)
);

INSERT INTO orders_old (id, user_id, amount_cents, status, created_at)
SELECT id, user_id, CAST(ROUND(amount * 100) AS INTEGER), status,
created_at FROM orders;

DROP TABLE orders;
ALTER TABLE orders_old RENAME TO orders;

-- If desired, drop phone column in users: (SQLite cannot drop column
- would recreate table)
-- Create users_old without phone, copy data, swap tables (careful
with FK constraints)
```

5. **Restore data from backup** (if needed):
    a. For full restore: replace production DB with the backup copy taken before migration (after shutting down writes). If using logical dump, import it.
6. **Post-rollback verification**: run tests, validate counts, and slowly resume writes.
7. **Root cause & remediation**: analyze migration failure, fix DDL or data conversion scripts, test again in staging before re-deploying.

Notes on partial rollback vs full restore

- Partial rollbacks (reverse DDL) are lower downtime but risk data inconsistencies created after the migration—especially if post-migration writes happened to new columns/tables.
- Full restore from pre-migration backup is safer for correctness but may lose post-migration accepted writes unless you replay them.

7) Example rollback scenario (concrete)

- Problem: After migration, `amount` shows rounding issues and some downstream jobs expect `amount_cents` integer.
- Safe rollback:
    - Put app in maintenance mode.
    - Create `orders_old` as above, convert `amount` back to cents rounding carefully.

    - Replace orders table with `orders_old` (swap tables in a transaction where DB supports it). For SQLite: copy & rename as shown.
    - Recreate any dropped constraints/indexes needed by older code.
    - Run integration tests.
    - Bring app back online.

References

- SQLite documentation: https://www.sqlite.org/docs.html
- Postgres `ALTER TABLE` docs
- General migration best practices: online resources and blog posts on zero-downtime migrations

THANKYOU