

**GigSphere:  
 A Cohesive Sphere to Bridge Freelancers and Clients**

**21CSC205P – DATABASE MANAGEMENT SYSTEM**

**PROJECT REPORT**

**B.Tech. CSE - CYBERSECURITY**

**SCHOOL OF COMPUTING**

**COLLEGE OF ENGINEERING AND TECHNOLOGY**

**SRM INSTITUTE OF SCIENCE AND TECHNOLOGY**

**(Under Section 3 of UGC Act, 1956)**

S.R.M. NAGAR, KATTANKULATHUR – 603 203 CHENGALPATTU DISTRICT

# **May 2025**

KIRONMOY MUKHERJEE (RA2311030010129)  
ADITYA KHANDELWAL (RA2311030010130)

**TABLE OF CONTENTS**

|  |  |  |
| --- | --- | --- |
| **S No.** | **Title** | **Page No.** |
| 1 | Introduction | 3 |
| 2 | Problem Statement | 4 |
| 3 | Abstract | 5 |
| 4 | Tech Stack | 6 |
| 5 | Existing Challenges | 7 |
| 6 | Listing of Entities | 8 |
| 7 | Code | 11 |
| 8 | Complex Queries | 14 |
| 9 | Pitfall Analysis | 19 |
| 10 | Functional Dependencies | 22 |
| 11 | Normalization | 24 |
| 12 | Transactions | 27 |
| 13 | User Interface | 30 |
| 14 | ER Diagram | 31 |
| 15 | Relational Schema | 32 |

**CHAPTER -1  
INTRODUCTION**

GigSphere is a complete freelance management system that helps optimize hiring and managing freelance experts. It offers a safe interface for freelancers and clients to work together, monitor progress, administer contracts, process payments, and settle conflicts. The system has features such as proposal submission, contract tracking, invoice creation, milestone payment, and user endorsements. Clients can list projects, view freelancer proposals, and hire experts on the basis of skills, while freelancers can search for work, send proposals, and track contracts. A formal system of endorsement improves reliability, enabling clients to identify freelancer ability through reviews and ratings.

The site comprises 16 basic entities, which are Users, Clients, Freelancers, Projects, Proposals, Contracts, Invoices, Payments, and Support Tickets. A Withdrawal process provides contract termination when required. GigSphere relies on a PostgreSQL database for scalability and data integrity, with API requests, payments, and authentication managed by a Python-Flask back-end. The front-end, created with HTML, CSS, and JavaScript, provides an intuitive user experience. With its strong architecture and secure financial management, GigSphere promotes trust, efficiency, and productivity in the gig economy.

**CHAPTER - 2  
PROBLEM STATEMENT**

The freelance market is expanding, but efficient project management is still a problem. Conventional platforms lack organized contracts, milestone payments, and open dispute resolution, resulting in delays, mismanagement, and distrust. Clients find it difficult to identify trustworthy freelancers and monitor progress, while freelancers experience unjust contract modifications, payment disputes, and non-recognition.

GigSphere addresses these problems through proposal tracking, milestone payments, flexible contracts, and an endorsement-based credibility framework. Secure transactions, dispute resolution, and transparent work tracking create a fair and efficient freelancing experience for every user.

**CHAPTER - 3  
ABSTRACT**

The independent industry is developing at a quick pace but handling projects continues to be tricky as a result of unorganized agreements, payment-related problems, and trust factors. Most sites center on advertisement posts and auctions but do not possess adequate solutions for tracking contracts, milestone payments, and solving disagreements. GigSphere addresses such challenges by presenting a full system of freelance administration to improve coordination, provide secure transactions, and ease project workflow processes.

GigSphere incorporates essential features such as proposal tracking, contract adaptability, milestone payments, and freelancer endorsements. Clients can publish projects, evaluate proposals, and hire freelancers, whereas freelancers can post bids, track contracts, and get paid securely. Dynamic contract management enables open-ended changes, avoiding disputes on project scope changes. Work submissions, endorsements, invoices, and support tickets further facilitate seamless operations. Developed using PostgreSQL, Flask, and a contemporary web interface, GigSphere provides an efficient and secure freelancing website that fosters efficiency and trust for everyone.

**CHAPTER - 4  
TECH-STACK**(TENTATIVE)

|  |  |
| --- | --- |
| **CATEGORY** | **TECHNOLOGY** |
| FRONT END | STREAMLIT (PYTHON) |
| BACK END | PYTHON (FLASK) |
| DATABASE | POSTGRE SQL |
| HOSTING | STREAMLIT CLOUD SERVER |
| VERSION CONTROL | GITHUB |

**CHAPTER - 5  
EXISTING CHALLENGES**

Several freelancing websites do not have an effective contract tracking system, which creates conflict regarding project scope, deadline, and compensation. In the absence of formal agreements, both clients and freelancers end up with misunderstandings and conflicts.

Payment delays are also a regular occurrence, since freelancers don't always receive payments on time or securely. The lack of milestone payments and escrow facilities means that clients can easily withhold or delay payments. Clients also often change project requirements, requiring freelancers to work overtime without adequate compensation or extension of deadlines.

Trust and credibility are also big issues. Without an endorsement system, freelancers can't demonstrate their expertise, and clients can't find reliable professionals. Most of these platforms do not have a formal work submission and approval procedure, leading to delays in completing projects and receiving payments.

Dispute resolution is also a problem. There are no defined mechanisms to manage contract breaches, scope creep, or payment defaults, leaving clients and freelancers with no adequate assistance. Financial monitoring is also limited, as most platforms lack transparent invoicing systems to monitor payments, taxes, and outstanding balances.

Finally, contract cancellation is usually rigid. When a project is no longer viable, neither clients nor freelancers have a formal method of terminating contracts without fines, which makes it hard to handle unforeseen circumstances smoothly.

**CHAPTER - 6  
LIST OF ENTITIES IN PROJECT**

users{  
user\_id PK  
username  
email  
user\_type  
created\_at  
password\_hash  
}  
  
clients{  
client\_id PK FK references users.user\_id  
company\_name  
business\_type  
}  
  
projects{  
project\_id PK  
client\_id FK references clients.client\_id  
title  
description  
budget  
deadline  
posted\_at  
status  
}  
  
freelancers{  
freelancer\_id PK FK references users.user\_id  
bio  
portfolio  
experience  
hourly\_rate  
}  
  
skills{  
skill\_id PK  
skill\_name  
}  
  
freelancer\_skills{  
skill\_id PK FK references skills.skill\_id  
freelancer\_id FK references freelancers.freelancer\_id  
}  
  
proposals{  
proposal\_id PK  
freelancer\_id FK references freelancers.freelancer\_id  
project\_id references projects.project\_id  
proposal\_text  
bid\_amount  
status  
submitted\_at  
}  
  
  
reviews{  
review\_id PK  
reviewer\_id FK references user.user\_id  
reviewee\_id FK references user.user\_id  
project\_id FK references projects.project\_id  
review\_text  
rating  
review\_date  
}  
  
endorsements{  
endorsement\_id PK  
client\_id FK references clients.client\_id  
freelancer FK references freelancers.freelancer\_id  
endorsement\_text  
}  
  
contracts{  
contract\_id PK  
client\_id FK references clients.client\_id  
freelancer\_id FK references freelancers.freelancer\_id  
project\_id FK references projects.project\_id  
proposal\_id FK references proposals.proposal\_id  
agreed\_price  
start\_date  
end\_date  
status  
}  
  
contract\_modification{  
modification\_id PK  
contract\_id FK references contracts.contract\_id  
modified\_by FK references users.user\_id  
modified\_at  
old\_price  
new\_price  
old\_deadline  
new\_deadline  
status  
}  
  
payments{  
payment\_id PK  
contract\_id FK references contracts.contract\_id  
amount  
payment\_method  
status  
}  
  
invoice\_id{  
invoice\_id PK  
contract\_id FK references contracts.contract\_id  
net\_amount  
issued\_at  
due\_date  
status  
paid\_at  
}  
  
submissions{  
submission\_id PK  
submitted\_by FK references freelancers.freelancer\_id  
contract\_id FK references contracts.contract\_id  
description  
submitted\_file  
submitted\_at  
approved  
}  
  
withdrawals{  
withdrawal\_id PK  
requested\_by FK references users.user\_id  
approved\_by FK references users.user\_id  
contract\_id FK references contracts.contract\_id  
reason  
status  
withdrawn\_at  
}  
  
support\_tickets{  
ticket\_id PK  
raised\_by FK references users.user\_id  
created\_at  
subject  
description  
status  
resolved\_at  
}

**CHAPTER - 7  
CODE**

A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.**CREATING THE TABLES**

**A screenshot of a computer program

AI-generated content may be incorrect.**

A screenshot of a computer

AI-generated content may be incorrect.**INSERTING DATA INTO TABLES**

A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.

**A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.OUTPUT**

A screenshot of a computer

AI-generated content may be incorrect.

**CHAPTER - 8  
COMPLEX QUERIES**

SETS

A screenshot of a computer program

AI-generated content may be incorrect.

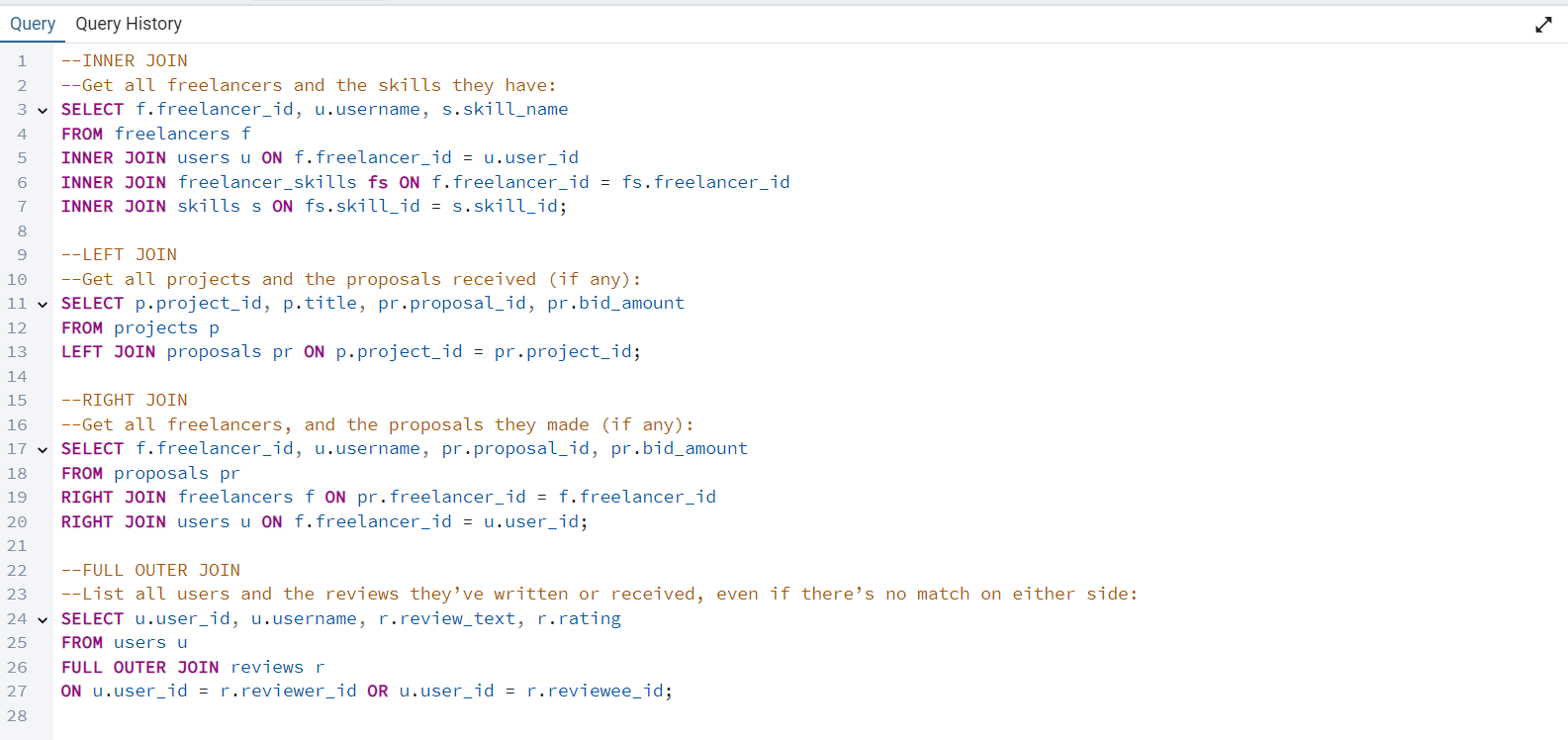
A screenshot of a computer

AI-generated content may be incorrect.

Sets in SQL refer to collections of rows returned by queries. SQL provides set operations like UNION, INTERSECT, and EXCEPT to compare and combine results from multiple queries, similar to mathematical set theory.

* The first query uses UNION to combine two sets: users who are freelancers and users who are clients, labeling each with their role.
* The second query uses INTERSECT to find users who are both freelancers and have received endorsements—only those present in both sets.
* The third query uses EXCEPT to find freelancers who have listed skills but have never submitted a proposal—returning only those in the first set but not the second.
* The fourth query also uses EXCEPT, identifying projects that received proposals but haven't resulted in signed contracts.

JOINS



Joins are used to combine rows from two or more tables based on related columns.

* INNER JOIN: Returns only freelancers who have matching user accounts and skills listed.
  + Shows freelancers along with their usernames and skill names.
* LEFT JOIN: Returns all projects, along with any proposals they received (if any).
  + Projects without proposals still appear with NULL in proposal fields.
* RIGHT JOIN: Returns all freelancers (and their usernames), plus any proposals they've made.
  + Freelancers without proposals still appear.
* FULL OUTER JOIN: Returns all users and any reviews they’ve written or received.
  + Ensures inclusion of users with no reviews and reviews not linked to active users.

VIEWS

A screenshot of a computer code

AI-generated content may be incorrect.

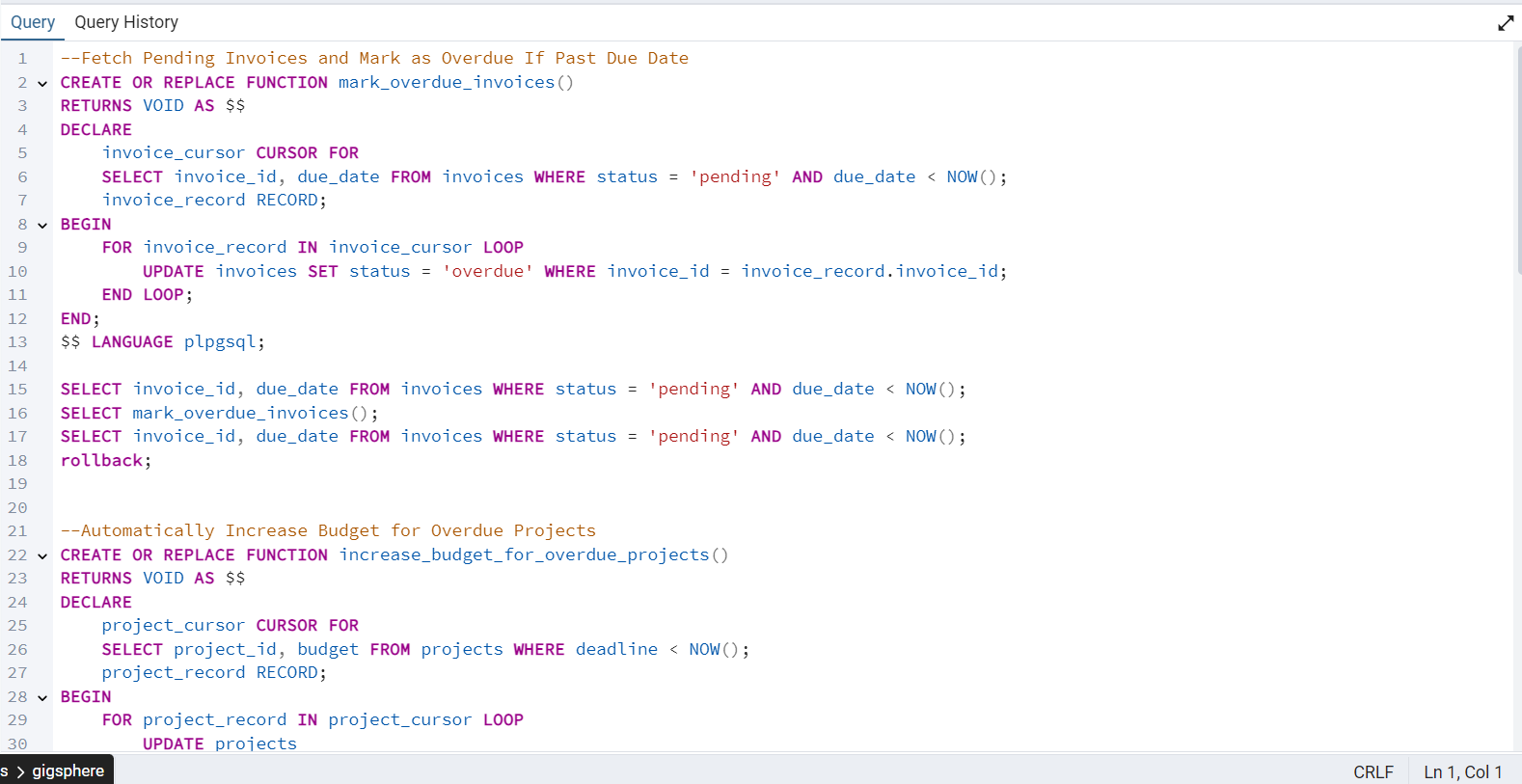
A screenshot of a computer

AI-generated content may be incorrect.

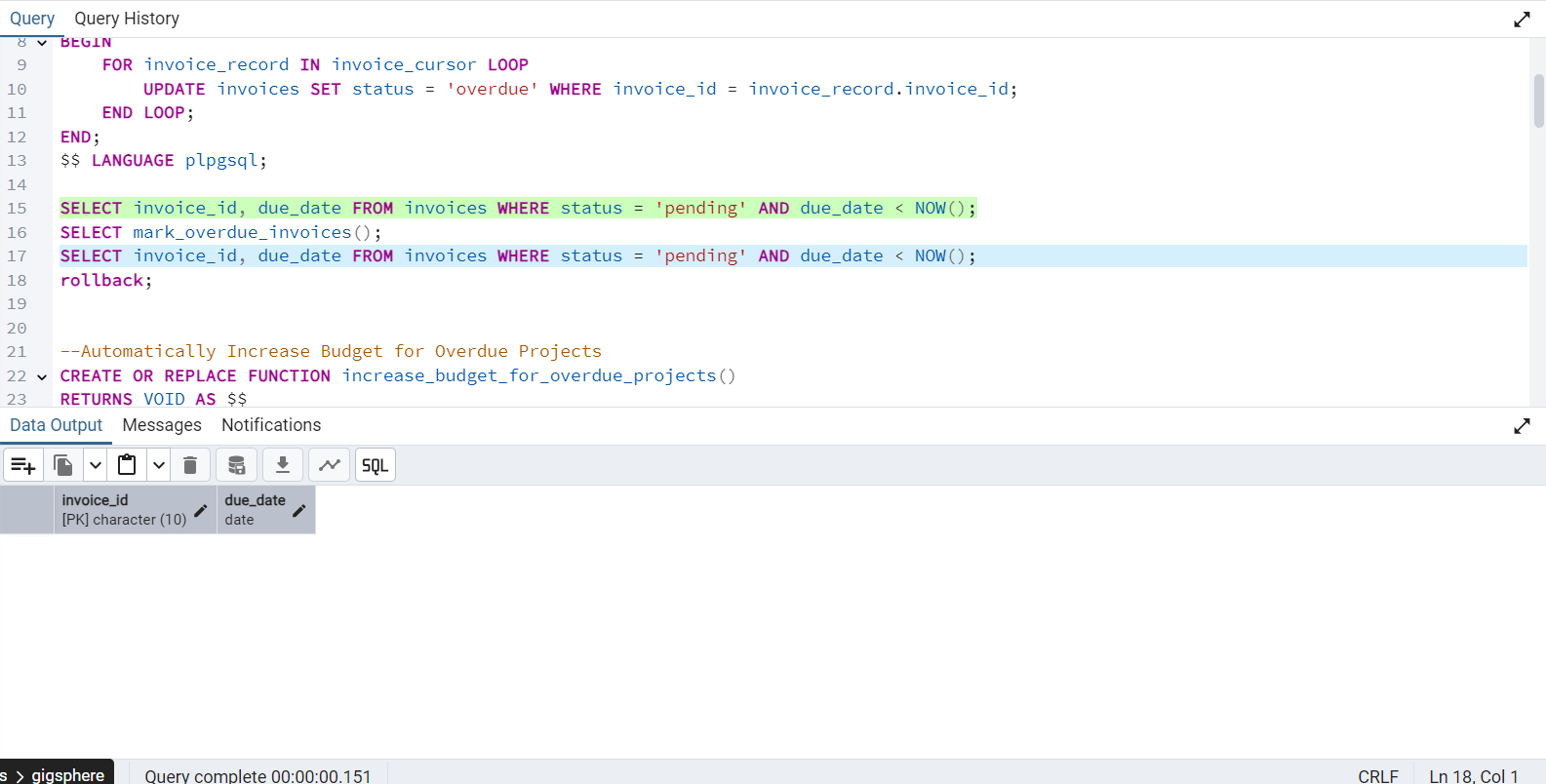
In SQL, views are virtual tables created by saving a query under a name. They allow you to simplify complex joins and aggregations by encapsulating logic into reusable, queryable components.

* The project\_proposal\_count view shows each project along with how many proposals it has received. It joins projects with proposals and uses COUNT to tally proposals per project.
* The freelancer\_skill\_summary view lists each freelancer's username and a comma-separated list of their skills. It joins the freelancers, users, freelancer\_skills, and skills tables, using STRING\_AGG to format skills into a single string per user.
* The projects\_without\_submissions view identifies projects that have not received any submissions after being contracted and whose deadlines have already passed. It joins projects, contracts, and submissions using LEFT JOIN and filters for missing submissions and expired deadlines.
* The withdrawn\_contracts\_refund view calculates the total amount paid for each withdrawn contract (i.e., potentially refundable). It joins contracts with payments, sums up payments per contract using COALESCE to handle cases with no payments, and filters for contracts with the status 'withdrawn'.

CURSORS

****

**A screenshot of a computer

AI-generated content may be incorrect.**

These PL/pgSQL functions automate database updates using cursors and loops to process multiple rows conditionally.

* mark\_overdue\_invoices(): Loops through all pending invoices past their due date and updates their status to 'overdue'.
* increase\_budget\_for\_overdue\_projects(): Increases the budget of all projects whose deadlines have passed by 10%.
* update\_missing\_project\_status(): Sets the status of any project missing a status (NULL) to 'open'.
* close\_completed\_projects(): Updates projects that have passed their end date and are still marked 'open', changing their status to 'closed'.

TRIGGERS



**A screenshot of a computer

AI-generated content may be incorrect.**

Triggers are special procedures in SQL that automatically execute in response to specific events (like INSERT, UPDATE, or DELETE) on a table.

* set\_default\_project\_status(): Automatically sets the status of a new project to 'open' when inserted.
* prevent\_user\_id\_change(): Prevents users from changing their own user\_id by raising an exception during an update.
* update\_invoice\_on\_payment(): Updates an invoice to 'paid' and sets the timestamp when a completed payment is inserted.
* enforce\_minimum\_bid(): Blocks any proposal where the bid\_amount is less than the freelancer's hourly\_rate.

**CHAPTER - 9  
PITFALL ANALYSIS**

**1. Repeating Groups and Non-Atomic Values (AVOIDED)**

Pitfall: Storing multiple values in a single field (e.g., skills as a comma-separated string) leads to unstructured data and complicates queries.

How it's avoided: The use of the freelancer\_skills junction table enforces atomic values, enabling a clean many-to-many mapping between freelancers and skills.

A screenshot of a computer

AI-generated content may be incorrect.

**2. Update, Insert, and Delete Anomalies (AVOIDED)**

Pitfall: Storing redundant information across tables may cause anomalies when updating or deleting records.

A screenshot of a computer

AI-generated content may be incorrect.How it's avoided: Normalization eliminates redundant storage. For instance, freelancer data is stored once in freelancers, and linked via foreign keys in proposals, contracts, and submissions.

**3. Lack of Referential Integrity (AVOIDED)**

Pitfall: Missing foreign keys or inadequate ON DELETE behavior can lead to orphaned records.

**A screenshot of a computer code

AI-generated content may be incorrect.**How it's avoided: Every relationship is clearly defined with foreign keys and ON DELETE CASCADE, ensuring that dependent records are automatically cleaned up (e.g., when a user is deleted, their freelancer or client record and linked contracts are also removed).

**4. Poor Separation of Concerns (AVOIDED)**

Pitfall: Mixing unrelated concerns into a single table (e.g., storing user credentials alongside business logic or project data).

How it's avoided:

users handles credentials and common data,

clients and freelancers extend user roles with role-specific fields,

projects, proposals, and contracts manage domain-specific interactions.

**A diagram of a flowchart

AI-generated content may be incorrect.**

**CHAPTER - 10  
FUNCTIONAL DEPENDENCIES**

**1) Functional Dependencies**

These exist when a unique identifier (usually a primary key) determines the rest of the attributes in the table.

Examples:

user\_id → username, email, user\_type, created\_at, password\_hash

project\_id → client\_id, title, description, budget, deadline, posted\_at, status

**2) Partial Dependencies**

These occur only in tables with composite primary keys — when a non-key attribute is dependent on only part of the key.

Using surrogate keys (e.g., proposal\_id instead of (freelancer\_id, project\_id))

Using junction tables (like freelancer\_skills) where all attributes are part of the composite key.

**3) Transitive Dependencies**

A transitive dependency occurs when: A → B and B → C, but C is stored in the same table as

Example (potential issue):

In invoices, contract\_id → freelancer\_id (through the contracts table), and freelancer\_id → email (via users), so storing freelancer\_email directly in invoices would be a transitive dependency.

**A screenshot of a computer screen

AI-generated content may be incorrect.**

A screen shot of a computer

AI-generated content may be incorrect.

**CHAPTER - 11  
NORMALIZATION**

1. **1NF**

* **Issue**: The skills column contains a list, violating 1NF's rule that each field must contain only one value per record.
* **Result**: Difficult to search, filter, or manage individual skills.
* **Fix**: A separate skills table lists all skills uniquely. freelancer\_skills maps freelancers to skills via a many-to-many relationship.
* **Benefit**: Allows efficient queries like "find all freelancers who know Python."

A screenshot of a computer code

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a web page

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

1. **2NF**

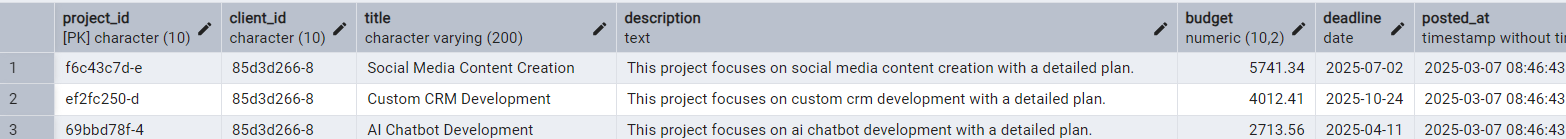
* **Issue**: freelancer\_name is dependent only on part of the composite key (freelancer\_id), violating 2NF.
* **Result**: Redundant storage of freelancer name across every project row.
* **Fix**: Freelancer details go in their own table, and references are made via foreign keys in proposals.
* **Benefit**: No redundancy. Freelancer info is stored once and reused via relationships.

A screenshot of a computer code

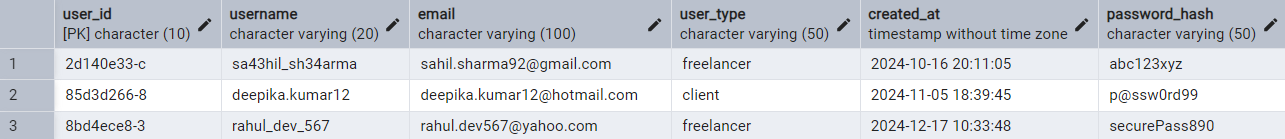
AI-generated content may be incorrect.

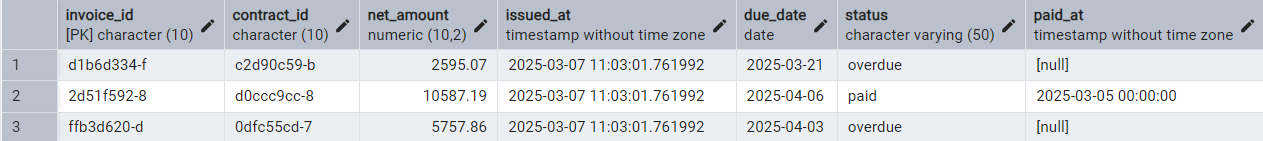
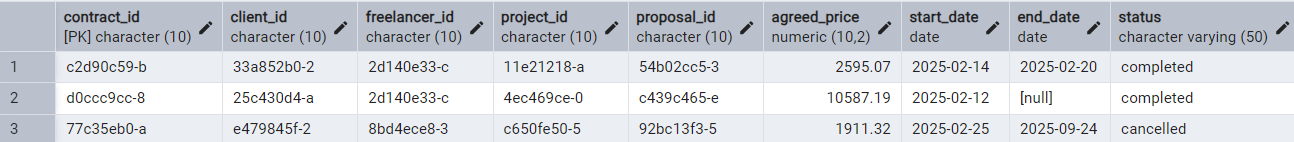
A screenshot of a web page

AI-generated content may be incorrect.

****

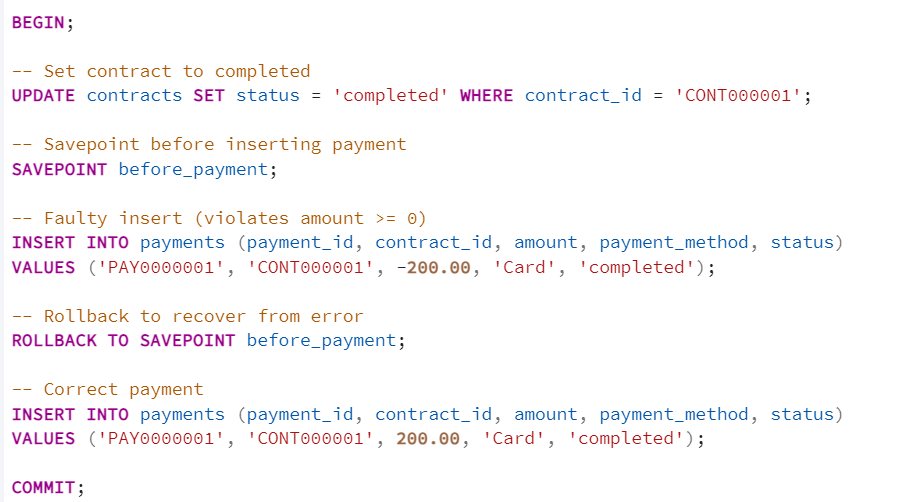
1. **3NF**
   * **Issue**: freelancer\_email is not directly dependent on invoice\_id. It depends on contract\_id, which then links to the freelancer.
   * **Result**: Data redundancy and potential inconsistency if the freelancer's email changes but isn't updated in invoices.
   * **Fix**: Instead of duplicating freelancer\_email in invoices, the design allows access to it via joins:
   * invoices → contracts → freelancers → users.
   * **Benefit**: Keeps invoices focused only on invoice-specific data. Reduces redundancy, ensures accuracy, and simplifies updates.

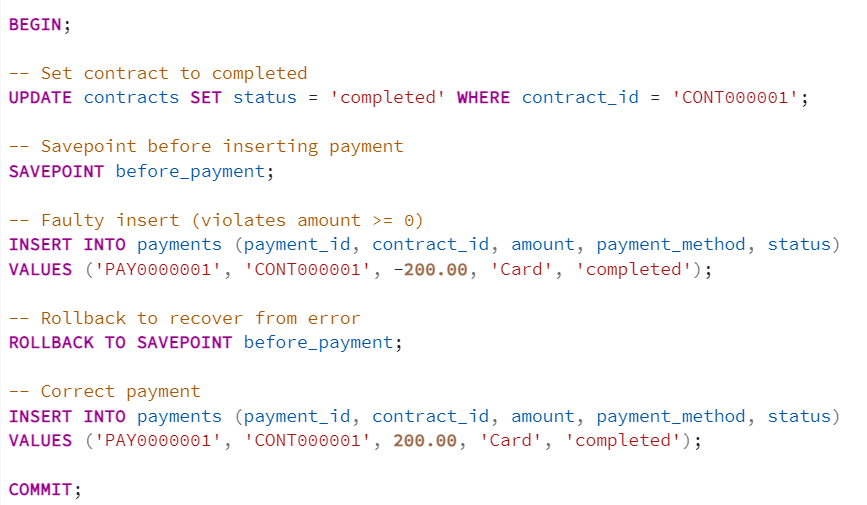
****A screenshot of a web page

AI-generated content may be incorrect.**A computer code with text

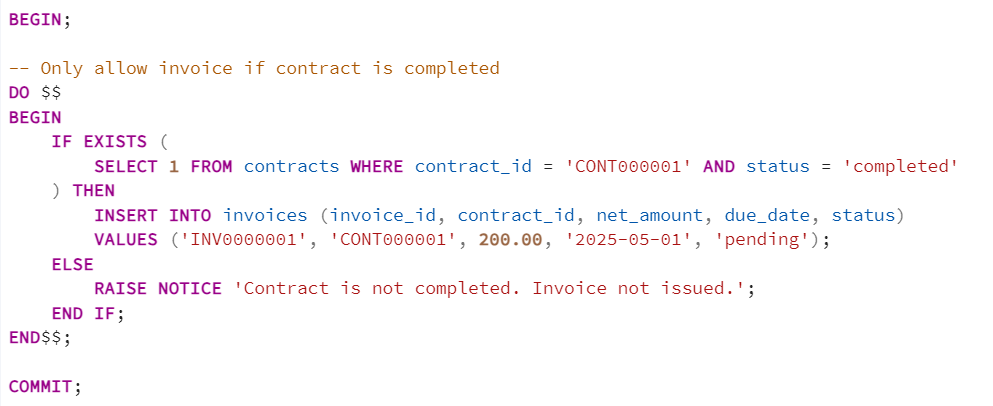
AI-generated content may be incorrect.**

**CHAPTER - 12  
TRANSACTIONS**

1. Simple atomic Transaction:
   1. Updates project and contract status together to ensure consistency.
2. Transaction with savepoint and rollback (recovery):
   1. Recovers from a failed payment insert by rolling back to a savepoint and retrying.



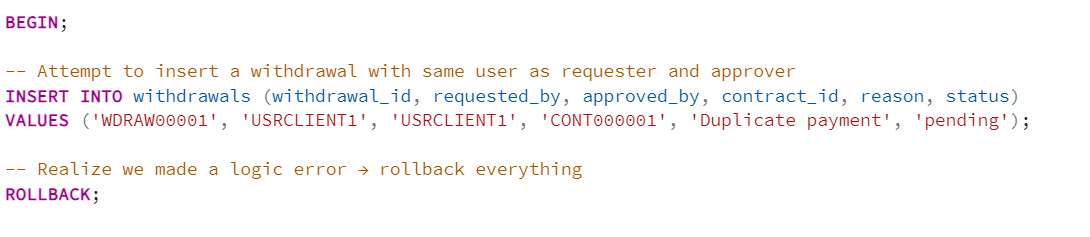
1. Transaction with conditional logic:
   1. Inserts an invoice only if the associated contract is marked as completed.



1. Transaction with concurrency control:
   1. Locks a contract row to safely update its price and status without interference.



1. Aborted transaction with full rollback:
   1. Cancels the entire withdrawal insert due to a logic error in user roles.



**CHAPTER - 13  
USER INTERFACE**

**A screenshot of a video game

AI-generated content may be incorrect.**

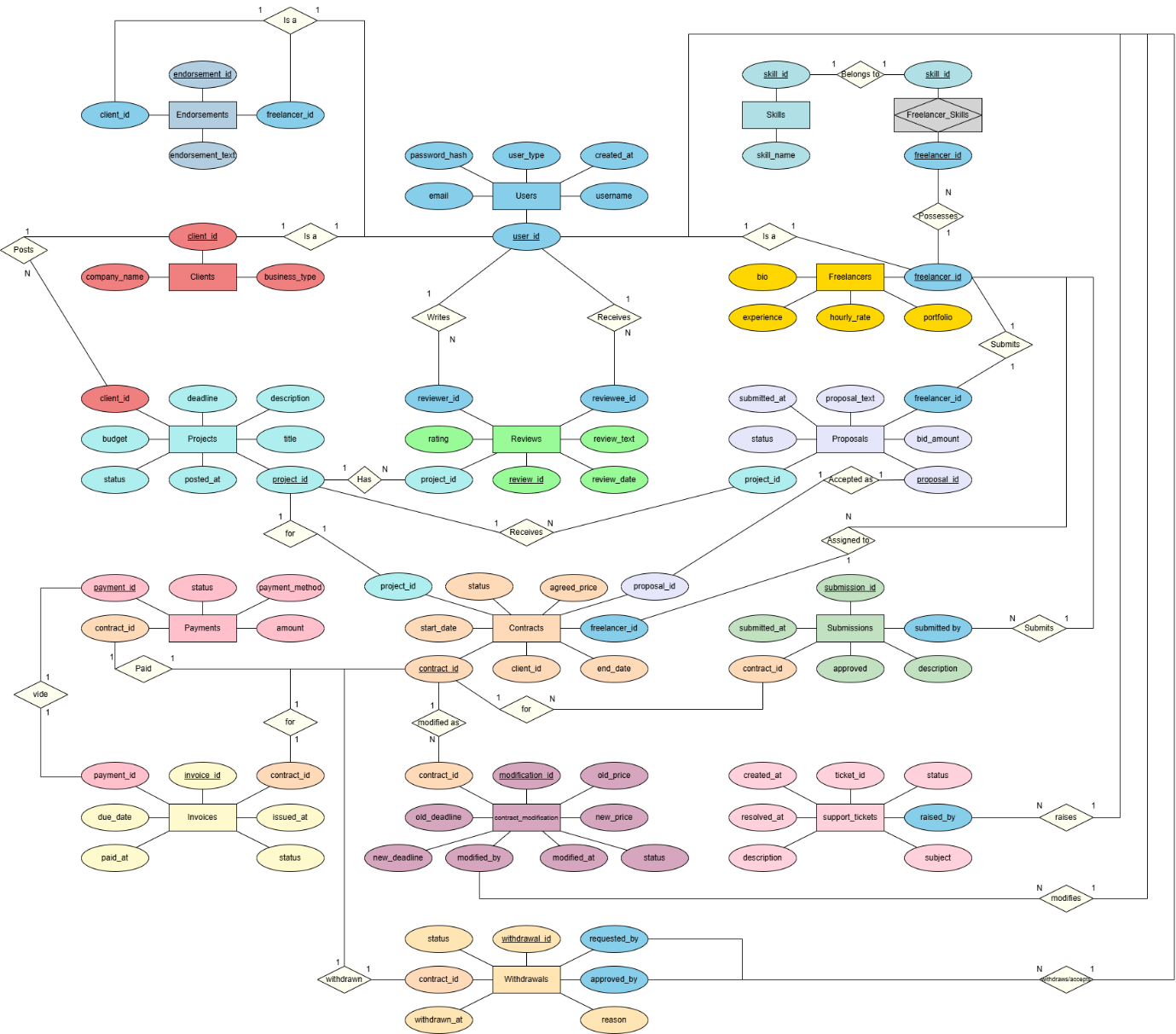
**A screenshot of a phone

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.**

**CHAPTER - 14  
ER DIAGRAM**



**CHAPTER - 15  
RELATIONAL SCHEMA**

A screenshot of a computer program

AI-generated content may be incorrect.