1. Introduction & Requirements

This document presents the final project for the Coding Factory bootcamp: the design and implementation of a proof-of-concept Customer Relationship Management (CRM) system specialized for short-term rental property operations.

The project addresses a real gap in existing platforms (such as Airbnb and Vrbo), which manage customer-facing activities but provide limited tools for vendor, maintenance, and property management. To bridge this gap, the project delivers a lightweight yet extensible CRM prototype that demonstrates both technical proficiency and practical business application.

Purpose & Context

Short-stay rental platforms centralize guest interactions but leave property owners without structured tools for business-to-business (B2B) coordination.

This CRM targets vendor management, pipelines, and property readiness workflows — critical back-office operations for hosts and property managers.

The deliverable showcases how core software engineering principles can be applied to solve this niche problem, while meeting the academic requirements of the assignment brief.

Scope & Deliverables

The submission includes:

- Backend service: Python + Bottle framework, implementing a REST API with a modular CRUD layer.
- Database schema: SQLite relational model with entities for vendors, contracts, properties, pipelines, and tasks (documented with ERD and SQL scripts).
- Validation system: Schema-based JSON payload validators ensuring data integrity.
- Authentication & Authorization: Token-based (JWT), role-controlled access for users.
- Frontend interface: python/tk, HTML + Bootstrap + JavaScript pages providing user workflows (login, create/read/update/delete, pipeline mapping).
- API specification: OpenAPI (YAML) file describing endpoints.
- Testing: Unit, integration of both backend and frontend workflows.
- Documentation: This report, including system design, architecture diagrams, testing strategy, and appendices with literature review.

Technology Stack

- Backend: Python, Bottle micro-framework, SQLite (prototype DB).
- Frontend: HTML5, Bootstrap 5, Vanilla JavaScript.
- Auth: JWT-based login with role separation (admin/user).
- Supporting Tools: LLM-assisted development workflows.

This stack was intentionally chosen for rapid development and extensibility: SQLite can be swapped for MySQL/PostgreSQL, and the frontend can evolve into React or mobile apps without altering backend logic.

Key Features

- REST API supporting CRUD operations across all entities.
- Role-based authentication & authorization (JWT lifecycle, login/logout, token revocation).
- Structured database schema and ERD, supporting properties, owners, vendors, pipelines, and contracts.
- Dynamic frontend pages for data entry, record management, and pipeline creation.
- Validator layer enforcing data types, formats, and required fields.
- Test coverage for validators, CRUD classes, and integration endpoints.

Future Work & Limitations

While the project delivers a minimal viable product (MVP), several enhancements are identified for future iterations:

- Refactoring for better support of DAO/Repository/DTO layers with CRUD abstractions.
- Security hardening: HTTPS, secure token storage, refresh tokens.
- Database migration: Move from SQLite to MySQL/MariaDB for scalability.
- Feature backlog: Reporting, dashboards, analytics, and vendor performance scorecards.
- UI modernization: SPA frameworks (React/Angular) or native mobile clients.

Recent work by Kalai et al. (2025) [15] argues that language models hallucinate not only due to data imperfections but because training and evaluation practices reward “guessing” instead of admitting uncertainty. They suggest that hallucinations can be reduced by modifying evaluation frameworks to avoid penalizing abstentions. This finding supports the approach taken in this project: LLM outputs were always subject to human verification. For future iterations, such research highlights how LLM-assisted workflows could be improved by explicitly rewarding uncertainty signals (e.g., “IDK”) over fabricated content, thereby streamlining reliability in both software development and CRM extensions.

1.1 Problem statement & objectives

While a lot of the operations of running a short stay hosting service is centralised through a short stay portal like airbnb and vrbo, these operations focus on business to client operations (payment processing, customer relational operations etc). A lot of the back end operations dealing with properties is not included within these portals as nothing more than recommendations or as policies, leaving each rental property owner / host to figure out the optimal solution.

Within this context, the author of this report recommends the implementation of a crm meant for business to business activities involving the property owners on one side and a set of other businesses involved with the sole purpose of maintaining properties. The net result of such a crm is

the creation of a management system for a set of activity pipelines whose sole goal is to bring rental properties to a reasonable state to be fit for rental.

Resolving these issues a minimal CRM prototype is implemented as a proof of concept that tracks the information a property owner needs for these activities. This is not a complete solution; rather a starting point one can explore their options, oportunities and risks while their business endeavours grow.

This sort of project fills a market share the same way a program like excel could be used for data management versus a more robust database management system like MySQL or a web service with a relational database persistence layer; the data are handled as they should be; inconsistencies are dealt with on time and reports can be generated orderly allowing for businesses to swiftly adapt to an ever changing landscape.

1.2 Scope of deliverables (docs, code, tests)

The deliverable we recommend in the previous section deviates from a classic business to consumer crm; it focuses to a specific niche market and has a more limited scope than a traditional crm. While comparisons can be made with run of the mill crms, this is for educational and practical reasons -nothing is born in a vacuum, and reference material has been studied before reaching the conclusion that specialised software would better fit the client needs. This is the reason of this document.

In this section there will be a brief overview of what will be produced for the bootcamp assignment as a client deliverable. More detailed analysis will follow in the next sections.

- A common form of crm implementation is in the form of web based services. A REST service has been implemented to accommodate the client needs.
- In order to maintain a reasonable codebase, the REST service has been split in 2 major parts: a backend, dealing with the database and the REST api, and a frontend, dealing with the user interface. The reasons for these choices will be analysed later in the document.
- A validation system has been put in place, to check for data type checks during data entry.
- The user interface has been built with simplicity in mind with bootstrap and html as documents. This allows for flexibility and rapid application development. It is also a more lightweight choice over SPA frameworks.

As per the assignment brief, a web application was built. There is a clear distinction between the front end and back end: a python/TK user interface and a web based user interface and a REST api backend using an embedded SQL database to hold persistent data. The frontend is built as single HTML web pages using bootstrap. The backend is written in Python and as a database engine, SQLite is used. Unit and black box tests were written to validate the codebase functionality.

This approach allows for a minimal viable product to be delivered on schedule for the assignment, with future work being secured: one can replace the front end with a more modern html approach like react, or build an android client and the backend being oblivious of the change. One can scale things using a more robust database solution like MySQL or MariaDB and the frontend be equally capable to deliver to the user as it would with an SQLite solution. Some developers choose java or node.js over python. The codebase can accommodate to those people accordingly.

To assist in the development of the codebase, LLMs were used. This choice was intentional as the project scope is ideal to stress test such models and coding assistants to the limits of their capabilities. For this project, chatGPT 4o, o3, 04-mini-high and 5 were the main core of tools used in this front with experiments done with local models used (with varying levels of success) .It also provided allowed the awthor to use training in tools not familiar with in the past (web development and database management with python, unit testing etc). This is not meant to be the code was utilised intact; in fact the contrary. A special chapter will follow as to the methodology used to produce the final production codebase, what changed, why, what problems did come up, how they were resolved.

1.3 Alignment with assignment brief

The project directly aligns with the Coding Factory assignment brief:

- Domain model and relational database schema
- Service/Controller layers exposed as REST API
- Frontend (HTML/Bootstrap; extensible to SPA frameworks)
- Authentication & authorization system
- Documentation and testing strategy
- OpenAPI spec for API documentation

2 System Design

2.1 Architecture overview (backend, DB, frontend)

The crm follows a traditional MVC architecture; a database (sqlite in this case) offers persistency layer to the service (implemented with the bottle REST library for python) and a view consisting of a set of html files for each database operation.

2.2 Database schema (ERD + pipeline mapping)

This section describes the structure of the relational database used by the CRM backend. It includes definitions of tables, fields, data types, and relationships.

1. vendors

Stores information about service providers.

Column	Type	Description
vendor_id	INTEGER	Primary Key (auto-increment)
name	TEXT	Name of the vendor
vendor_type	TEXT	Type/category of vendor
status	TEXT	Current status
phone	TEXT	Contact number
email	TEXT	Email address
address	TEXT	Street address
created_at	TEXT	Date of creation

updated_at	TEXT	Last update timestamp

2. vendor_contacts

Contacts associated with each vendor.

Column	**Type**	**Description**
contact_id	INTEGER	Primary Key (auto-increment)
vendor_id	INTEGER	FK → vendors.vendor_id
first_name	TEXT	First name
last_name	TEXT	Last name
job_title	TEXT	Role/title
phone	TEXT	Phone number (optional)
email	TEXT	Email (optional)
created_at	TEXT	Created timestamp
updated_at	TEXT	Last updated timestamp

3. vendor_scorecard

Performance data for vendors over a period.

Column	**Type**	**Description**
scorecard_id	INTEGER	Primary Key (auto-increment)
vendor_id	INTEGER	FK → vendors.vendor_id
period_start	TEXT	Start of evaluation period
period_end	TEXT	End of evaluation period TEXT
quality_score	NUMERIC	Numeric score (default: 0)
on_time_delivery	NUMERIC	Timeliness score (default: 0)
defect_rate	NUMERIC	Defect rate percentage (default: 0)
comments	TEXT	Optional reviewer comments
created_at	TEXT	Created timestamp
updated_at	TEXT	Last updated timestamp

4. contracts

Binding agreements with vendors.

Column	Type	Description
contract_id	INTEGER	Primary Key (auto-increment)
vendor_id	INTEGER	FK → vendors.vendor_id
contract_name	TEXT	Name/title of the contract
start_date	TEXT	Contract start date
end_date	TEXT	Contract end date
renewal_terms	TEXT	Optional renewal conditions
created_at	TEXT	Created timestamp
updated_at	TEXT	Last updated timestamp
status	TEXT	Current contract status

5. documents

Files related to contracts and vendors.

Column	Type	Description
document_id	INTEGER	Primary Key (auto-increment)
vendor_id	INTEGER	FK → vendors.vendor_id
contract_id	INTEGER	FK → contracts.contract_id
file_name	TEXT	Name of the uploaded file
file_path	TEXT	Path to stored file
uploaded_at	TEXT	Upload timestamp
updated_at	TEXT	Last modified timestamp

6. communications

Messages between users and vendors.

Column	Type	Description
communication_id	INTEGER	Primary Key (auto-increment)
vendor_id	INTEGER	FK → vendors.vendor_id
date_sent	TEXT	When message was sent
comm_type	TEXT	Type (e.g., email, phone)
Subject	TEXT	Message subject line
content	TEXT	Message body

user_id	TEXT	ID of user who sent it
followup_needed	TEXT	'1' or '0' if follow-up required
created_at	TEXT	Record creation timestamp
updated_at	TEXT	Record update timestamp

7. pipelines

Sales or vendor engagement pipelines.

Column	Type	Description
pipeline_id	INTEGER	Primary Key (auto-increment)
pipeline_name	TEXT	Name of the pipeline
description	TEXT	Description of purpose
created_at	TEXT	Creation timestamp
updated_at	TEXT	Last updated

8. pipeline_stages

Stages within a pipeline.

Column	**Type**	**Description**
stage_id	INTEGER	Primary Key (auto-increment)
pipeline_id	INTEGER	FK → pipelines.pipeline_id
stage_name	TEXT	Name of the stage
stage_order	INTEGER	Order in the pipeline
created_at	TEXT	Created timestamp
updated_at	TEXT	Last updated

9. opportunities_task_tickets

Tracks tasks or deals in pipeline stages.

Column	**Type**	**Description**
opportunity_id	INTEGER	Primary Key (auto-increment)
pipeline_id	INTEGER	FK → pipelines.pipeline_id
stage_id	INTEGER	FK → pipeline_stages.stage_id
vendor_id	INTEGER	FK → vendors.vendor_id
title	TEXT	Title of the task
description	TEXT	Detailed description

property_id	INTEGER	FK → properties.property_id
status	TEXT	Current status
priority	TEXT	Priority (optional)
due_date	TEXT	Deadline
created_at	TEXT	Created timestamp
updated_at	TEXT	Last modified

10. properties

Properties being managed.

Column	Type	Description
property_id	INTEGER	Primary Key (auto-increment)
property_name	TEXT	Name
address	TEXT	Location
status	TEXT	Active/inactive
created_at	TEXT	Created timestamp
updated_at	TEXT	Last modified

11. owner

Property owners.

Column	Type	Description
owner_id	INTEGER	Primary Key (auto-increment)
first_name	TEXT	Owner first name
last_name	TEXT	Owner last name
phone	TEXT	Contact phone
email	TEXT	Contact email
address	TEXT	Mailing address
legal_id	TEXT	Legal ID (e.g., Tax ID)
created_at	TEXT	Created timestamp
updated_at	TEXT	Last modified

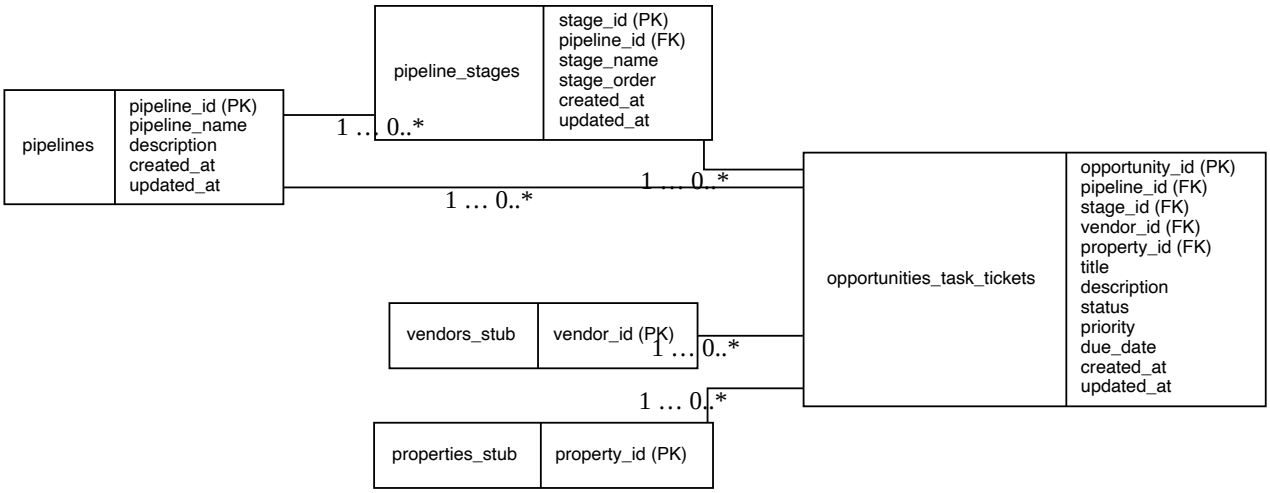
12. owner_property_map

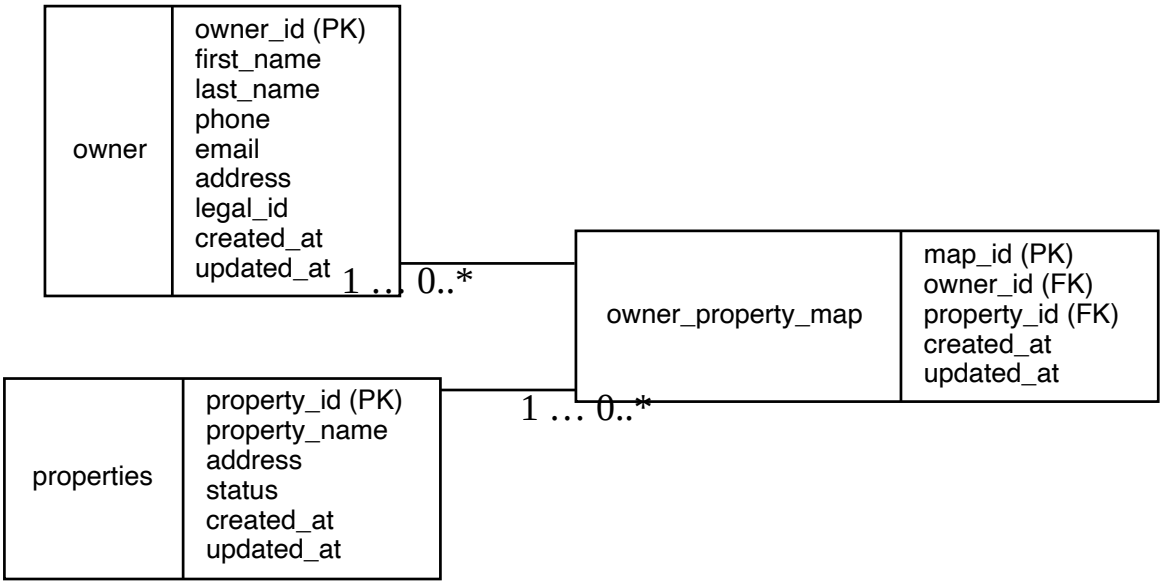
Many-to-many mapping of owners to properties.

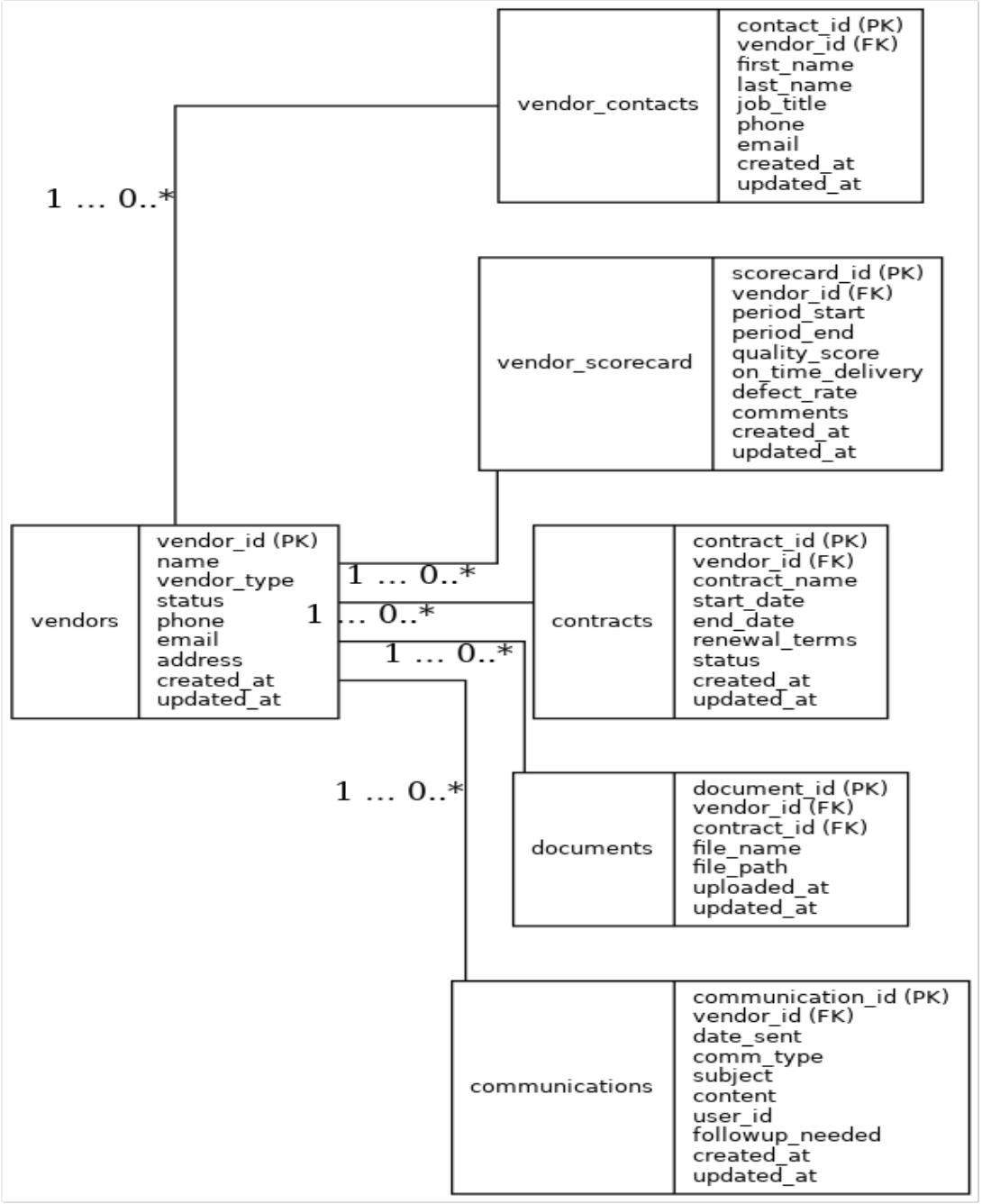
Column	Type	Description
map_id	INTEGER	Primary Key (auto-increment)
owner_id	INTEGER	FK → owner.owner_id
property_id	INTEGER	FK → properties.property_id
created_at	TEXT	Defaults to '00.00.0000'
updated_at	TEXT	Defaults to '00.00.0000'

For manual creation of the database schema for the sqlite database, and for complete transparency of the development process, a set of sql queries were developed as per the appendix at the end of this document.

ER diagram




owner
owner_id (PK)
first_name
last_name
phone
email
address
legal_id
created_at
updated_at
1 … 0..*
owner_property_map
map_id (PK)
owner_id (FK)
property_id (FK)
created_at
updated_at
properties
property_id (PK)
property_name
address
status
created_at
updated_at
1 … 0..*


vendor_contacts
contact_id (PK)
vendor_id (FK)
first_name
last_name
job_title
phone
email
created_at
updated_at
1 ... 0..*
vendor_scorecard
scorecard_id (PK)
vendor_id (FK)
period_start
period_end
quality_score
on_time_delivery
defect_rate
comments
created_at
updated_at
vendors
vendor_id (PK)
name
vendor_type
status
phone
email
address
created_at
updated_at
1 ... 0..*
1 ... 0..*
1 ... 0..*
1 ... 0..*
contracts
contract_id (PK)
vendor_id (FK)
contract_name
start_date
end_date
renewal_terms
status
created_at
updated_at
documents
document_id (PK)
vendor_id (FK)
contract_id (FK)
file_name
file_path
uploaded_at
updated_at
communications
communication_id (PK)
vendor_id (FK)
date_sent
comm_type
subject
content
user_id
followup_needed
created_at
updated_at

2.3 Key invariants (validators, triggers, business rules)

This module provides a structured way to validate incoming JSON payloads against the expected schema of each database table before insertion or update. It helps ensure data integrity and proper formatting.

Key Concepts

- ValidationError: A custom exception raised when payload data violates the schema definition.
- BaseValidator: A generic class that performs validation based on a `SCHEMA` dictionary provided by subclasses.

SCHEMA Format

Each subclass defines a SCHEMA attribute with:

SCHEMA: Dict[str, Tuple[field_type: str, required: bool]]

Where field_type is one of:

INT	Integer
FLOAT	Float/Decimal
STR	String
DATE	Date string in format `DD.MM.YYYY
Required	is a boolean indicating whether the field must be present.

BaseValidator Methods

validate_full(data: Dict[str, Any])	Validates a **create** payload. All required fields must be present and valid.
validate_partial(data: Dict[str, Any])	Validates an **update** payload. Only supplied fields are checked (for type/format only).
_run(data: Dict[str, Any], partial: bool)	Internal method that performs the checks: • Missing required fields (if not partial) • Unknown fields • Type checks • Date format validation

Field Type Constants

INT = "int"
FLOAT = "float"
STR = "str"
DATE = "date"

ValidationError

class ValidationError(ValueError):   """Raised when input data violates a table schema."""
Thrown if: * Required fields are missing (in full mode) * Unknown fields are included * Field types don’t match * Dates are not in `DD.MM.YYYY` format

Example: VendorsValidator

class VendorsValidator(BaseValidator):
  SCHEMA = {
  "name": (STR, True),
  "vendor_type": (STR, True),
  "status": (STR, True),
  "phone": (STR, True),
  "email": (STR, True),
  "address": (STR, True),
  "created_at": (DATE, True),
  "updated_at": (DATE, True),
  }

This validator ensures that any payload for the vendors table: * Includes all required fields * Matches expected types * Uses the correct date format for created_at and updated_at

List of Validators (One per Table)

Table	Validator Class Name
vendors	VendorsValidator
vendor_contacts	VendorContactsValidator
vendor_scorecard	VendorScorecardValidator
contracts	ContractsValidator
pipelines	PipelinesValidator
pipeline_stages	PipelineStagesValidator
opportunities_task_ticket	OpportunitiesTaskTicketsValidator
properties	PropertiesValidator
documents	DocumentsValidator
communications	CommunicationsValidator
owner	OwnerValidator
owner_property_map	OwnerPropertyMapValidator

Each validator reflects its corresponding table’s schema.

3 Implementation

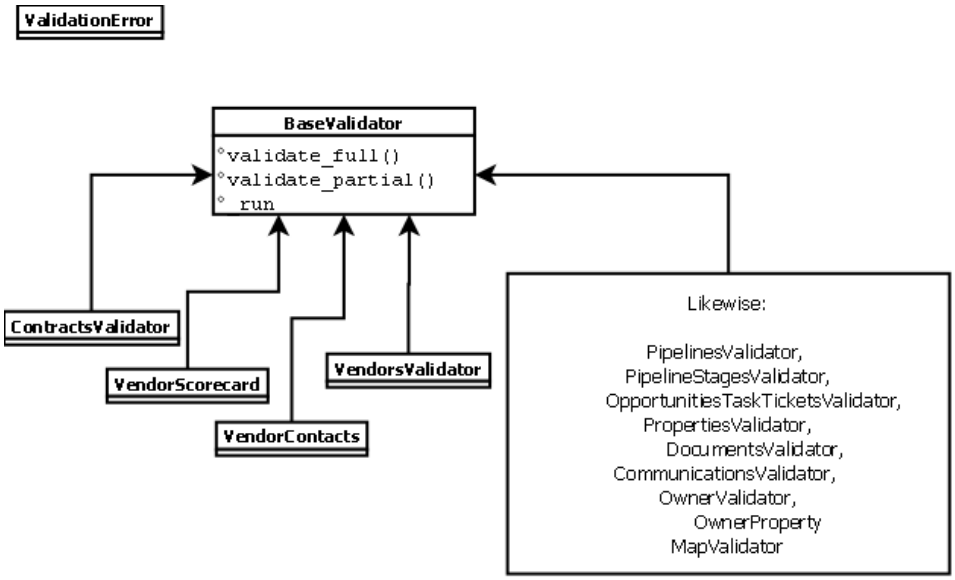
3.1 Backend

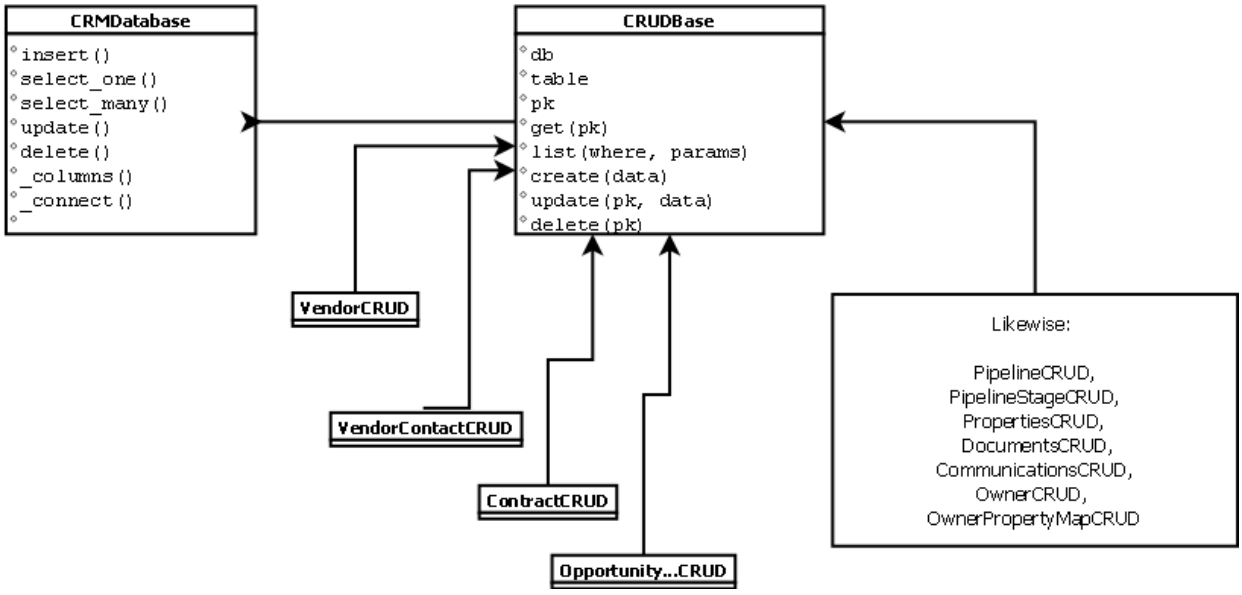
This module defines the web API and application logic for the CRM system using the Bottle micro web framework. It includes authentication, validation, CRUD routing, and DB access.

Core Components Overview

Component	Purpose
Bottle app	Defines HTTP endpoints
JWT auth	Handles login/logout with token-based access control
Validators	Enforce payload conformity to schema
CRUD classes	Interface for reading/writing to each table
Route factory	Auto-generates endpoints for all entities
SQLite database	Lightweight DB backend (via `sqlite3`)

Class diagrams for the backend codebase (Class-heavy OOP service)





Notes

- Routes are generated per entity and call the appropriate validator before invoking CRUD methods.
- JWT/auth decorators and the route factory are functions (not shown since this is a class diagram).

3.1.1 Authentication

Authentication is implemented using JSON Web Tokens (JWT).

On successful login, the backend issues two tokens: a short-lived access token and a long-lived refresh token. Unlike earlier versions of the system that stored tokens in `localStorage`, the current design stores them in **HttpOnly cookies** with `SameSite=Lax` and `Secure` flags. This prevents JavaScript from reading tokens and reduces the risk of XSS token theft.

The typical lifecycle is as follows:

1. User submits username/password via the login form.
2. If credentials are valid, the server issues and sets cookies:
   - `access_token` (30 minutes lifetime)
   - `refresh_token` (14 days lifetime)
3. On each API request, the browser automatically includes cookies. The `@auth_required` decorator verifies the `access_token`.
4. If expired, the client may call `/auth/refresh` using the `refresh_token` to obtain a new access token.
5. Role claims (e.g. `admin`, `user`) are encoded in the token and checked by `@admin_only`.

```
def auth_required(fn):
    def wrapper(*a, **kw):
        tok = request.get_cookie("access_token")
        if not tok: abort(401, "missing token")
        try:
            payload = jwt.decode(tok, SECRET, algorithms=["HS256"])
            request.user = payload
        except jwt.ExpiredSignatureError:
            abort(401, "expired")
        return fn(*a, **kw)
    return wrapper
```

This model balances security and usability while keeping the implementation lightweight.

3.2 API Layer

The API layer is built with Bottle. It exposes **CRUD endpoints** for each entity and handles static file serving for the HTML frontend.

Routing:	Entity routes are auto-registered using a route factory, ensuring consistency across all CRUD operations (`GET /entity`, `POST /entity`, etc.).
Static files:	Static frontend assets (HTML, JS, CSS) are served via Bottle's `static_file` helper:

```
@app.get("/<filename:path>")
def serve_static(filename):
    return static_file(filename, root="./static")
```

Error handling:
All endpoints return JSON error responses with a consistent schema:

```json
{
  "error": "validation_error",
  "fields": { "email": "invalid format" }
}
```

DTOs:	Validators act as Data Transfer Objects: each table has a matching validator class that enforces required fields, types, and enumerations before data reaches the repository layer.

A.4. Frontend – HTML Pages

The frontend consists of multiple HTML files that form a **simple admin interface** for CRUD operations. It uses:

- HTML5
- Bootstrap 5 (CDN)
- Vanilla JavaScript
- Token-based authentication via JWT $Cookies + CSRF model.
- A shared `app.js` module

Shared Features Across Pages

- Common API base: `http://localhost:8000`
- Bootstrap-based responsive UI
- Error messages shown via alerts or `<div>`s

A.4.1 index.html – Login and Navigation

Purpose	Landing page with login form. After login,redirects to create/update page, or allows for read and delete in the apropriate tabs.
Features	• Username/password login

	• Post-login options: Create, Read, Update, Delete
JS Logic Highlights	• Sends `POST /login` • Shows/hides UI sections based on login state • Logout button calls `POST /logout` and clears token

A.4.2 create-update.html – Dynamic Record Creation

Purpose	Data and update entry page .
Features	• Dropdown to select entity • Form fields rendered dynamically based on entity definition • Handles input types (text, email, date, textarea, checkbox) • Adds audit fields: `created_at`, `updated_at` (auto-stamped) • browser cookies + X-CSRF-Token
JS Logic	• `entityFields` defines required fields per entity • Handles date formatting (`DD.MM.YYYY`) • Shows response box with result of API call

A.4.6 pipeline-map.html

Purpose	This page provides a dedicated interface for creating and managing CRM pipeline structures. It supports both manual entry of pipelines with their respective stages, and bulk insertion of pre-defined reference pipelines commonly used in property and vendor management.
Features	• Entity selection dropdown (choose pipeline). • Fetch tasks by pipeline ID via `fetcher()` and `fetchRange()`. • Render as a nested Bootstrap list (pipeline → stages → tasks).
JS Logic Highlights	JavaScript logic: uses the same `fetcher()` function as other CRUD pages but builds nested `<ul>` structures instead of flat tables.
UI Components	• Manual Entry Form • Reference Pipelines Selection Panel
Security & Access	(TODO)

A.4.7 app.js

Purpose	Centralized frontend utility library for CRUD pages.
Features	Shared fetch wrapper with CSRF, input sanitization (esc), unified alert rendering, date formatting helper.
JS Logic Highlights	• fetcher(entity, start, end)` → wrapper around `fetchRange`

	that updates the relevant `<div>`. • fetchRange(entity, start, end)` → performs `fetch` with `credentials: "include"`, adds `X-CSRF-Token` header, parses JSON. • `esc(str)` → HTML escape utility to prevent XSS. • `showAlert(type, msg)` → renders a Bootstrap alert in a target container.

3.1.1 Authentication & authorization (roles, JWT lifecycle, secrets)

JWT Authentication

The system uses **JSON Web Tokens (JWT)** to secure the API endpoints and manage user sessions. This implementation is custom-built using the `jwt` Python package and Bottle middleware.

Overview

The system uses a dual-token JWT model: short-lived access token (30m) and long-lived refresh token (14d). Both are stored in HttpOnly cookies with Secure and SameSite=Lax flags, preventing client-side JS from reading them.

Token Generation

On successful login:

Server issues access + refresh tokens, signed with HS256.

Access token includes claims: username, role, exp.

Refresh token allows obtaining new access tokens at /auth/refresh.

Token Revocation (Logout)

POST /logout clears both cookies and invalidates refresh tokens. Future requests without valid tokens receive 401 Unauthorized.

Authentication Middleware

Custom Bottle decorators enforce security:

@auth_required → requires valid access token.

@admin_only → requires token + role=admin.

Frontend Integration

The browser automatically includes cookies in each request. For write requests, the frontend attaches X-CSRF-Token header obtained during login.

Security Notes

- Cookies use HttpOnly, Secure, and SameSite=Lax.
- Passwords are hashed with bcrypt.
- Tokens are signed with a secret loaded from environment variable SECRET_KEY.
- Users and secret key are stored over a JSON file. This ofters a neutral point of security disjoint from the database and the service code; it is not the best solution; it is a simple neutral for the task

JWT Payload Example

```
{
  "username": "admin",
  "role": "admin",
  "exp": "2025-07-24T18:42:00Z"
}
```

3.1.2 Services & repositories (layer mapping to brief)

The system uses an object-oriented **CRUD abstraction layer** to interact with the SQLite database. It minimizes repetition and enforces consistent access patterns for all tables.

Base Class: CRUDBase

This is the parent class from which all table-specific classes inherit.

Constructor:

```
def __init__(self, db: CRMDatabase, table: str):
    self.db    = db             # instance of CRMDatabase
    self.table = table          # table name as string
    self.pk    = VALIDATORS.get(table, (...))[1]  # primary key name
```

Methods:

Method	**Description**
get(pk)	Returns one record by primary key
list(...)	Returns all records (optionally with filters)
create(data)	Inserts a new row, returns its primary key
update(pk, data)	Updates fields in row identified by PK
delete(pk)	Deletes record with given primary key

Table-Specific Classes

Each table has its own subclass, which extends `CRUDBase` and specifies the table name.

Example:

```
class VendorCRUD(CRUDBase):
    def __init__(self, db):
        super().__init__(db, "vendors")
```

Some classes **override** `get`, `update`, and `delete` to explicitly specify their unique primary key name, e.g.:

```
class VendorContactCRUD(CRUDBase):
    def get(self, pk):
        return self.db._select_one(self.table, "contact_id", pk)
```

Registration

All instances are created and stored in the `entities` dictionary:

```
entities = {
    "vendors": VendorCRUD(db),
    ...
}
```

Related Component: `CRMDatabase

This class encapsulates the raw database I/O logic.

Methods:

`_insert`, `_select_one`, `_select_many`
`_update`, `_delete`
`_columns` – returns set of column names for a table

Used internally by `CRUDBase` to ensure safe, consistent transactions.

Validation Tie-In

Each CRUD route in the backend invokes a corresponding validator class before calling the CRUD methods.

```
err = _run_validation(validator_cls, data, partial=False)
```

This guarantees that the CRUD layer only operates on validated data.

Example Workflow

To **create** a new contract:

1. `POST /contracts/` is called
2. Authenticated and authorized
3. `ContractsValidator.validate_full()` runs
4. If valid, `ContractCRUD.create(data)` calls `_insert()`
5. Response includes new `contract_id`

CRUD API Endpoints (Auto-Generated)

For each entity (e.g. `vendors`, `contracts`), the following endpoints are created:

Method	**Path**	**Auth**	**Description**
POST	/[entity]/	Admin	Create a new record
GET	/[entity]/<id_start>/<id_end>/	Any user	Fetch records in ID range
PUT	/[entity]/	Admin	Update record (by primary key)
DELETE	/[entity]/	Admin	Delete record (by primary key)

Handled via `_register_entity_routes()` using:

```
VALIDATORS: Dict[str, tuple[ValidatorClass, primary_key]]
entities: Dict[str, CRUDInstance]
```

Route Factory Logic

The `_register_entity_routes()` function dynamically builds all 4 CRUD routes for each entity by:

- Retrieving the appropriate validator & primary key name
- Binding `POST`, `GET`, `PUT`, and `DELETE` handlers to the Bottle app
- Applying auth decorators where needed

CRUD Class Structure

```
class CRUDBase:
    def get(self, pk)
    def list(where="", params=())
    def create(data)
    def update(pk, data)
    def delete(pk)
```

Each table has its own subclass, e.g.:

```
class VendorCRUD(CRUDBase):
    def __init__(self, db):
        super().__init__(db, "vendors")
```

Some classes override `get`, `update`, `delete` to use specific PKs (e.g. `contact_id` instead of `id`).

3.1.3 Validators & triggers (with code snippets)

The frontend consists of multiple HTML files that form a **simple admin interface** for CRUD operations. It uses:

- HTML5
- Bootstrap 5 (CDN)
- Vanilla JavaScript
- Token-based authentication
- A shared `app.js` module

Shared Features Across Pages

- Common API base: `http://localhost:8000`
- Bootstrap-based responsive UI
- Error messages shown via alerts or `<div>`s

3.2 API Layer

Static Files

3.2.1 index.html – Login and CRUD operations

Purpose	Landing page with login form. After login, shows navigation to data create and update. Redirects to pipeline page.
Features	• Username/password login • Post-login options: Create, Read, Update, Delete • Redirection to pipelines page.
JS Logic Highlights	• Handles each operation with the apropriate HTTP verb • Tabbed UI sections based on operation • Logout button calls `POST /logout` and clears token

3.2.1 Routes & DTOs

Besides a login/logout api, an /openapi and a /docs endpoint, the api deals with database operations. For each entity (e.g. `vendors`, `contracts`), the following endpoints are created:

Method	**Path**	**Auth**	**Description**
POST	/[entity]/	Admin	Create a new record

GET	/[entity]/<id_start>/<id_end>/	Any user	Fetch records in ID range
PUT	/[entity]/	Admin	Update record (by primary key)
DELETE	/[entity]/	Admin	Delete record (by primary key)

Handled via `_register_entity_routes()` using:

```
VALIDATORS: Dict[str, tuple[ValidatorClass, primary_key]]
entities: Dict[str, CRUDInstance]
```

Route Factory Logic

The `_register_entity_routes()` function dynamically builds all 4 CRUD routes for each entity by:

- Retrieving the appropriate validator & primary key name
- Binding `POST`, `GET`, `PUT`, and `DELETE` handlers to the Bottle app
- Applying auth decorators where needed

3.2.2 OpenAPI spec (aligned with DTOs & error envelopes)

The deliverable has a /openapi and /docs api endpoints for a yaml file description of the api and the servide documentation. Please refer to the appendix for the yaml file for further details

3.2.3 Error handling & pagination/filtering

All API endpoints return a consistent JSON envelope on errors:

```
{ "error": "validation_error", "fields": { "email": "invalid format" } }
```

- 401 Unauthorized: missing/invalid token.
- 403 Forbidden: insufficient role.
- 404 Not Found: resource not found.
- 409 Conflict: FK or uniqueness constraint violation.

Pagination & Filtering
Range-based GET endpoints follow the format:

```
GET /vendors/1/50/
```

This returns IDs 1 through 50 inclusive.

- Reversed or oversized ranges are rejected with 400 Bad Request.
- Future work: support query filtering (e.g., /vendors?status=active) and cursor-based pagination.

3.4 DTOs vs Validators

Context

During the initial MVP, per-table validators doubled as Data Transfer Objects (DTOs). This satisfied the requirement to accept and validate JSON payloads, but led to drift between documentation, OpenAPI schemas, and runtime behavior. As the system evolved, it became clear that separating DTOs from validators was necessary to clarify responsibilities and eliminate inconsistencies.

Options Considered

1.Validators as DTOs

- One artifact enforces both payload shape and business rules.
- Smaller surface, faster to implement.
- Higher risk of spec drift and unclear error semantics.

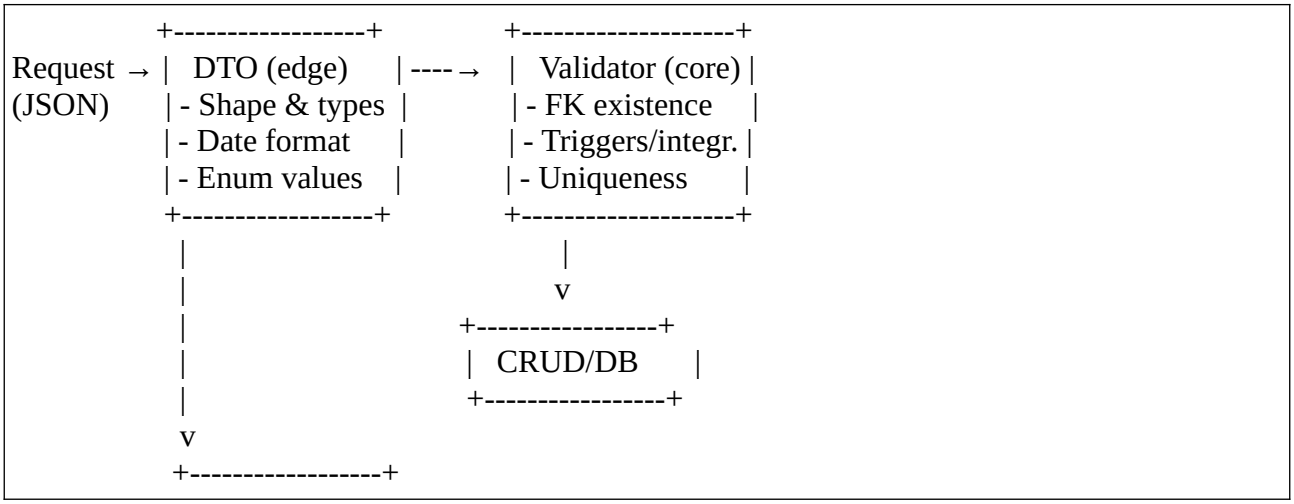
2. Split DTOs (edge) vs Validators (core)

- DTOs (e.g., Pydantic models) enforce types, formats, enums, and provide JSON Schema for OpenAPI generation.
- Validators remain for DB-level rules (FK existence, trigger invariants, uniqueness).
- Clearer layering, better documentation, and less drift.

Decision

We adopt split responsibilities:

1. DTOs define request/response payloads at the API boundary.
   - Shape, types, required fields, date format (DD.MM.YYYY), enumerations.
   - Auto-generate OpenAPI schemas.
   - Standard error envelope for format/type violations.
   - Validators remain the internal guardrail.
   - Check FK relationships, trigger expectations, uniqueness constraints.
   - Catch DB-level inconsistencies beyond JSON shape.



3.3 Frontend

3.3.1 UI pages & workflows

The frontend is a set of static HTML/Bootstrap pages with JS helpers:

- index.html: login page, sets cookies, navigation.
- create-update.html: dynamic form builder for all entities.
- pipeline-map.html: renders nested pipelines → stages → tasks.

Workflows:

- User logs in via index.html.
- Upon login, navigation enables CRUD and pipeline features.
- CRUD pages render forms dynamically from entity definitions.

3.3.2 Client-side validation & security posture (XSS/CSRF)

- JS enforces input sanitization: esc(str) escapes HTML to prevent XSS.
- Dates are validated client-side before submission.
- All API writes attach X-CSRF-Token.
- Cookies are flagged HttpOnly and not accessible via JS.
- Bootstrap alerts provide clear error messages.

Future extension: add stronger client-side validation (regex for emails, phone formats).

4 Testing

4.1 Testing strategy (pyramid)

We follow a pragmatic pyramid: validators & schema at the base, CRUD/database integration in the middle, API/Auth and UI smoke at the top. Focus areas: (a) schema invariants & referential integrity, (b) validator semantics (required/type/date), (c) auth matrix and error envelopes, (d) critical CRUD paths and range reads.

4.2 Schema-level tests (FKs, triggers)

Files: pytest-complete/test_schema_integrity.py, db.pytest/schema.sql.

We seed canonical rows (e.g., vendors, properties) and assert FK/NOT NULL behavior. Orphaned inserts are rejected; guarded deletes fail as expected. Transactions are wrapped with rollback to keep tests isolated.

4.3 Validator/unit tests

Files: pytest-complete/test_validators_positive.py, pytest-complete/test_validators_negative.py.

Full create requires all mandatory fields; partial update only validates supplied keys. We enforce types (INT, FLOAT, STR, DATE) and strictly accept dates formatted as DD.MM.YYYY. Negative cases cover missing fields, unknown keys, and wrong formats.

4.4 API/integration tests (auth matrix, CRUD)

Files: pytest-complete/bb_suite.5.py (smoke), test_delete_guards_matrix.py.

We assert standard CRUD behavior through the service and confirm delete guard rails where FK relations exist.

Planned additions (next iteration):

Role matrix: unauthenticated → 401; user vs admin → 403 vs 200 for POST/PUT/DELETE; GET accessible with valid token.

Error envelopes: malformed payloads and FK violations return a stable {error, details?} structure.

4.4.1 Black-box testing (scope, method, and instances)

Definition & rationale.

We use black-box tests to validate externally observable behavior of the service without relying on internal implementation details. This complements schema/validator unit tests and CRUD integration tests by exercising the system exactly as a client would (HTTP requests against published endpoints, auth cookies/CSRF, and error envelopes). These tests live alongside the API/integration suite and are executed with the same runner. We reference them here as three focused instances: login/refresh/logout, CRUD happy-paths with guards, and range GET/error envelopes. (Runner: pytest-complete/bb_suite.5.py.)

Method.

Each scenario is expressed as request/response pairs against the running Bottle app, asserting:

- status codes (200/401/403/404/409),
- stable error envelope shape ({error, fields?}),
- contract conformance to the OpenAPI spec,
- cookie/CSRF behavior on write operations.

Black-box instances covered

- Auth lifecycle — login → access token cookie → protected GET → refresh → logout
  Asserts cookie issuance, @auth_required/@admin_only gates, and refresh flow semantics. (See 3.1.1 for the cookie+CSRF model.)
- CRUD happy-path + delete guards — for a representative entity (e.g., vendors). POST valid payload → GET by ID range shows record → PUT partial update → DELETE guarded by

FKs where applicable; verifies 409 on constraint violations and consistent error envelope. (Related file: test_delete_guards_matrix.py.)

- Range GET + error envelopes — robustness checks on the read surface. Inclusive ranges (e.g., /vendors/1/50/), empty ranges, reversed bounds (→ 400), and oversized ranges; all negative paths must return the standard error shape documented in 3.2.3.

Traceability (requirements → tests).

Requirement (brief / design)	Observable contract (doc section)	Black-box scenario
Auth with roles & refresh	3.1.1 Authentication (cookies + CSRF)	(1) Auth lifecycle
CRUD over entities	3.2 API layer, routes per entity	(2) CRUD happy-path
Consistent error envelopes	3.2.3 Error handling	(2) and (3)
Range-based reads	3.2.3 Pagination & Filtering	(3)

4.5 End-to-end smoke tests (UI)

Scope: minimal black-box flows covering login, create/list/update/delete, and pipeline bulk insert logging.

Planned additions: fetch-mock tests for common.js (Authorization header injection, serializeForm type coercion, Ts() date stamping).

4.6 CI/coverage setup

4.6 CI & coverage

Runner: pytest (8.4.1) with pytest-cov.

Latest run (DB & CRUD focus): 9 tests, 100% pass (db.pytest/result.txt).

Planned target: ≥ 80% line coverage overall, with validators ≥90%, routes ≥80%.

Reports: line/branch breakdown exported to Appendix 8.5 (HTML or PDF). CI fails below thresholds.

5 Responsible LLM Usage

Given the complex nature of the subject topic, this, as much as topics as

- Where LLMs were applied (with provenance table)
- Human verification steps
- Reflection on effectiveness & limitations

are covered in the appendices, along with a small literature review, theory covering the subject matter and other topics.

6 Future Work & Limitations

As mentioned through this document, there are three main categories of improvement:

- The use of DTOs and DAO oop architecture; which can be resolved through code refactoring and moving to a bigger framework like Django.
- Use of RDBMs like Postgres or MySQL
- An extension of the previous point, the use of 3rd normal form for the database schema (which is covered in the appendix)
- Security (which is covered in the appendix section)

6.1 Security hardening (production-ready gaps)

While the MVP provides working authentication and validation, several hardening measures are planned:

- **Content Security Policy (CSP):** Limit script/style origins to `self` and trusted CDNs, block inline scripts.

```
response.set_header("Content-Security-Policy",
    "default-src 'self'; script-src 'self' https://cdn.jsdelivr.net; style-src 'self' 'unsafe-inline' https://cdn.jsdelivr.net")
```

- **Escape untrusted output:** Use `esc()` helper for all dynamic HTML to prevent XSS.
- **Cookie flags:** All cookies set with `HttpOnly`, `Secure`, `SameSite=Lax`. In production behind TLS, force `Secure`.
- **Token rotation:** Access tokens rotated via `/auth/refresh`; refresh tokens stored in cookies with limited lifetime.
- **Monitoring:**Add logging of failed login attempts and anomaly detection.

Database hardening:

- Principle of least privilege (separate DB users).
- Enforce FK constraints (`PRAGMA foreign_keys = ON`).
- Add audit triggers for sensitive tables.

6.2 Scalability & migration path

- Database: Migrate SQLite → Postgres (preferred) or MySQL. Introduce an ORM (SQLAlchemy) and Alembic for migrations. Add connection pooling (e.g., pgbouncer).
- Service packaging: Containerize (Docker). Create compose profiles (dev/test/prod) and a Kubernetes deployment manifest when scale requires.
- API surface: Stabilize DTOs; add pagination, filtering, and consistent error envelopes. Version the API (/v1).

- Caching: Read-heavy endpoints can utilize in-memory caching (Redis) with cache-key invalidation on writes.
- Search & reporting: Offload heavy queries to a reporting replica; for text search consider Postgres GIN indexes.
- Observability: Health checks, Prometheus metrics, and trace IDs in logs.
- Static frontend: Keep current HTML/Bootstrap for MVP. If/when moving to SPA (React/Angular), keep backend contract stable and deploy as separate artifact via CDN.
- CI/CD: Add tests to pipeline; gate deployments on unit + integration + security checks (bandit, trivy).

6.3 Feature backlog (reporting, dashboards, etc.)

Prioritized backlog tied to the domain model:

- Dashboards & KPIs: Vendor score trends, on-time delivery, defect rates, open tickets by stage, SLA breach alerts.
- Reporting: Periodic PDF/CSV exports (contracts expiring, tickets overdue, vendor comparison).
- Workflow automation: Stage transitions → auto-create tasks/notifications; renewal reminders; escalation policies.
- Attachments & doc mgmt: S3-style object storage; signed URLs; metadata tagging; retention policies.
- Advanced auth: OAuth2/OIDC integration; SSO for admins; refresh token rotation.
- Bulk ops & imports: CSV/Excel imports for vendors/properties with validation preview.
- Integrations: PMS/Channel managers, accounting (e.g., Xero/QuickBooks), Slack/Email webhooks.
- Data quality: Duplicate detection, referential checks, background jobs for cleanup.
- Role matrix & audit UI: Human-readable audit feed; per-field change history for key tables.

6.4 DTOs vs Validators (Current & Future Plan).

In the MVP we intentionally let validators double as DTOs: the same class that checks incoming payloads also defines each entity’s shape (fields, types, required flags). This is a pragmatic choice that keeps the codebase small and the boundary contract explicit. In the next iteration we will formalize DTOs with Pydantic models. Pydantic gives us typed objects, aliasing (to separate internal snake_case from API camelCase), first-class JSON Schema generation for OpenAPI, and richer per-field/cross-field validation. Making OpenAPI derive from DTOs eliminates drift, improves testability, and—critically—increases portability. If we migrate the backend to Ruby on Rails, we will keep the OpenAPI contract (same field names, enums, error envelope, and auth semantics) and re-implement the validations with ActiveModel/custom validators or dry-validation. Because the contract is language-neutral and machine-readable, Rails can match the same API precisely while leveraging its own idioms.

Validators ↔ DTOs crosswalk (scope of this project)

Why this table matters: in this MVP, validators are the DTOs at the HTTP boundary—each validator’s SCHEMA defines the wire contract (fields, types, required flags) and is invoked by the

route factory before any CRUD call. That’s exactly the “boundary-only model” approach you described in the document, and it’s how one can satisfy the brief’s DTO requirement without adding heavy mapping layers.

Notes on behavior & usage (tie-in to docs/spec)

- **Validation modes:** validate_full (create) enforces all required fields; validate_partial (update) checks only supplied keys. Both reject unknown fields, wrong types, and bad dates (DD.MM.YYYY). This is the request DTO behavior.
- Error envelope: on violations, routes should return a stable JSON error with details; OpenAPI responses should reflect this and be tested test for it.
- Route factory wiring: the map VALIDATORS: Dict[str, (ValidatorClass, primary_key)] drives POST/GET-range/PUT/DELETE registration per entity—so the validator defines both the DTO and the PK contract used by routes. code & docs should hold them as single source of truth.

API entity (path)	Vendors
DB table	Vendors
Validator class	VendorsValidator
Primary key (as used by routes)	vendor_id
Required fields (create)	name, vendor_type, status, phone, email, address, created_at, updated_at
Optional / notes	-
Dates (format)	`DD.MM.YYYY` for both
Where used as DTO	Request (POST/PUT) & Response shape
Drift / quick fixes	Spec lists same fields; keep aligned with OpenAPI.

API entity (path)	vendor_contacts
DB table	vendor_contacts
Validator class	VendorContactsValidator
Primary key (as used by routes)	`contact_id` (note: not the default `id`)
Required fields (create)	`vendor_id, first_name, last_name, job_title, phone, email, created_at, updated_at`
Optional / notes	-
Dates (format)	`DD.MM.YYYY`
Where used as DTO	Request & Response
Drift / quick fixes	Ensure OpenAPI uses `contact_id` and field names that match the validator.

API entity (path)	vendor_scorecard
DB table	vendor_scorecard

Validator class	VendorScorecardValidator
Primary key (as used by routes)	scorecard_id
Required fields (create)	vendor_id, period_start, period_end, quality_score, on_time_delivery, defect_rate, created_at, updated_at
Optional / notes	`comments` optional
Dates (format)	`DD.MM.YYYY`
Where used as DTO	Request & Response
Drift / quick fixes	Keep YAML schema in sync with actual numeric field names (score/metrics).

API entity (path)	contracts
DB table	contracts
Validator class	ContractsValidator
Primary key (as used by routes)	contract_id
Required fields (create)	(as per schema section—ids, parties, dates)
Optional / notes	-
Dates (format)	DD.MM.YYYY
Where used as DTO	Request & Response
Drift / quick fixes	Verify required set matches DB columns in appendix.

API entity (path)	pipelines
DB table	pipelines
Validator class	PipelinesValidator
Primary key (as used by routes)	pipeline_id
Required fields (create)	pipeline_name, description, created_at, updated_at
Optional / notes	-
Dates (format)	DD.MM.YYYY
Where used as DTO	Request & Response
Drift / quick fixes	-

API entity (path)	pipeline_stages
DB table	pipeline_stages
Validator class	PipelineStagesValidator
Primary key (as used by routes)	stage_id
Required fields (create)	pipeline_id, stage_name, stage_order, created_at, updated_at
Optional / notes	-

Dates (format)	DD.MM.YYYY
Where used as DTO	Request & Response
Drift / quick fixes	Stage must belong to pipeline (enforced downstream by triggers/CRUD).

API entity (path)	opportunities_task_ticket
DB table	opportunities_task_tickets
Validator class	OpportunitiesTaskTicketsValidator
Primary key (as used by routes)	opportunity_id
Required fields (create)	pipeline_id, stage_id, vendor_id, property_id, title, description, status, created_at, updated_at
Optional / notes	priority, due_date
Dates (format)	DD.MM.YYYY
Where used as DTO	Request & Response
Drift / quick fixes	**Path singular vs table plural**: unify in OpenAPI/JS/tests. Add enum checks for `status/priority` later.

API entity (path)	properties
DB table	properties
Validator class	PropertiesValidator
Primary key (as used by routes)	property_id
Required fields (create)	property_name, address, status, created_at, updated_at
Optional / notes	-
Dates (format)	`DD.MM.YYYY
Where used as DTO	Request & Response
Drift / quick fixes	-

API entity (path)	documents
DB table	documents
Validator class	DocumentsValidator
Primary key (as used by routes)	document_id
Required fields (create)	vendor_id, contract_id, file_name, created_at, updated_at
Optional / notes	Optional: `mime_type`, `file_path`
Dates (format)	DD.MM.YYYY
Where used as DTO	Request & Response
Drift / quick fixes	**Schema drift:** some seed code uses `file_path` & `uploaded_at`; align validator/DB/OpenAPI + JS forms.

API entity (path)	communications
DB table	communications
Validator class	CommunicationsValidator
Primary key (as used by routes)	communication_id
Required fields (create)	vendor_id, date_sent, comm_type, subject, content, followup_needed, created_at, updated_at
Optional / notes	`user_id` optional
Dates (format)	`DD.MM.YYYY` (date_sent as date)
Where used as DTO	Request & Response
Drift / quick fixes	Ensure OpenAPI reflects boolean/int for `followup_needed` consistently.

API entity (path)	owner
DB table	owner
Validator class	OwnerValidator
Primary key (as used by routes)	owner_id
Required fields (create)	first_name, last_name, phone, email, address, legal_id, created_at, updated_at
Optional / notes	-
Dates (format)	DD.MM.YYYY
Where used as DTO	Request & Response
Drift / quick fixes	-

API entity (path)	owner_property_map
DB table	owner_property_map
Validator class	OwnerPropertyMapValidator
Primary key (as used by routes)	map_id (current validator)
Required fields (create)	owner_id, property_id, created_at, updated_at
Optional / notes	-
Dates (format)	DD.MM.YYYY
Where used as DTO	Request & Response
Drift / quick fixes	PK mismatch: schema shows composite `(owner_id, property_id)` in some places; validator/routes assume `map_id`. Pick one and propagate.

6.4 Design Decision: Entity Naming, REST Resource Conventions, Range Retrieval, and Secret Management

Context

The assignment requires a CRUD-centric CRM with a documented REST API, OpenAPI description, and a relational schema. Early iterations revealed drift between validators, OpenAPI, and UI, mostly around **names** (singular/plural, field aliases), **collection semantics**, and h**ow we slice records**. We also had to choose an authoritative source for **service secrets** used by authentication.

Options Considered

1. **DB tables: singular (`Owner`, `Vendor`) vs plural*(`owners`, `vendors`).
2. **REST paths: singular (`/vendor/123`) vs **plural (`/vendors/123`).
3. **Range access: path segments (`/vendors/{start}/{end}/`) vs query params (`/vendors?from=… &to=…`).
4. Secrets: environment variables / OS keyring vs JSON configuration file (repo-local) vs hybrid.

Decision

- Schema (DB) uses singular table names (e.g., `Owner`, `Vendor`, `PipelineStage`).
- REST endpoints use plural collection names (e.g., `GET /vendors`, `GET /vendors/{vendor_id}`).
- Ranges are exposed via path-style slices (`/{start}/{end}/`) to satisfy assignment ergonomics; a query variant (`?from=&to=`) is specified for future compatibility.
- Secrets(service key, bootstrap users/hashes) are loaded from a JSON configuration file at runtime and must override any placeholder defaults; the JSON is the authoritative source of truth.

Rationale

A) Conceptual clarity (schema)

- The schema models entity types; singular names (“`Vendor`”) align with ER/OO modeling and keep migrations and DTOs intuitive.
- Singular tables reduce pluralization edge-cases (e.g., `companies`, `categories`) and keep class⇄table mental mapping 1:1.

B) REST semantics (API)

- REST treats `/vendors` as a collection resource and `/vendors/{id}` as a member resource. Plural is the de facto convention across major ecosystems, aiding API discoverability and client expectations.
- Error messages, pagination, hypermedia (“next/prev”) and list filters read naturally in the plural.

C) Testability & contract checks

- A consistent split—singular in DB, plural in HTTP—lets us write contract-drift tests that compare validator/DTO fields and OpenAPI schemas without arguing about plurality at each layer.

- Tooling (linters, OpenAPI generators, client SDKs) assumes plural collections and benefits from predictable paths.
-

D) Range retrieval

- The assignment’s teaching goal emphasizes URL path literacy; `/{start}/{end}/` is explicit and easy to demonstrate in slides and curl.
- We preserve an alternate query model (`?from=&to=`) for interoperability with code-gen clients and future pagination mechanisms; both represent the same collection slice.

E) Secret management

- A repo-local JSON makes the system reproducible for graders and teammates (no hidden CI secrets required).
- It centralizes bootstrap auth data (service key, initial users), simplifying local HTTPS runs and deterministic tests.
- Placeholders in code are **non-authoritative** and exist only to prevent hard crashes during misconfiguration; they never override JSON at runtime.

Trade-offs

- Plural tables would align strings between DB and API but introduce OO/ER friction and pluralization edge-cases.
- Singular API paths can look tidy but violate common REST expectations, surprising external integrators.
- Path-based ranges don’t compose as flexibly as queries (e.g., adding filters); conversely, query params are less demonstrative for students learning route design.
- JSON secrets are easier to run locally but require discipline (git-ignore, example files) to avoid accidental leaks; env-only setups are safer by default but harder for graders to reproduce.

Mitigations

- Contract Map (single page in docs): for each entity, list DB table (singular), primary key, REST base path (plural), required fields, and date format.
- Contract-drift tests: CI job that parses OpenAPI and compares field sets/required keys to validators/DTOs.
- Dual range support: implement both `/{start}/{end}/` and `?from=&to=`; document them as equivalents.

Secrets hygiene:

- Store `config.secret.json` (real) outside VCS; commit `config.secret.example.json` with structure only.
- On boot: fail hard if JSON is missing; ignore any inline placeholders.
- Optionally allow env vars to override JSON in production.

Evidence in the Codebase

- Schema files and migration notes use singular table names, aligning with entity classes and validators.
- Controllers/Routes expose plural collections (`/vendors`, `/owners`, `/opportunities_task_tickets`).
- Range endpoints implemented as `/{start}/{end}/` with a backlog ticket to add the query form.
- Auth bootstrap loads from the JSON file; the inline secret is a development fallback scheduled for removal before final submission.

Future Work

1. Single-source schema: move validators/DTOs to Pydantic (or equivalent), generate OpenAPI from DTOs to eliminate drift entirely.
2. Pagination & filtering: formalize `limit`, `offset`, `from`, `to`, and common filters under the query model while keeping path ranges for backward compatibility.
3. Secret provider abstraction: plug-in loader that prioritizes environment/secret-manager in production, JSON in dev/test, with a checksum to detect stale configs.
4. Automated docs checks: CI gate that fails on any mismatch among DB columns, DTOs, and OpenAPI schemas (names, types, required, enums).

7 Conclusion

This work delivers a CRM prototype for property operations, with a relational model, validator-guarded REST backend, and lightweight admin UI. It meets the assignment brief (domain model, DB, service/controller layer, auth, basic frontend, testing, documentation) and demonstrates a clean migration path to a production-grade stack (ORM + Postgres, containers, CI/CD, observability). Limitations are intentional (SQLite, minimal RBAC, simple JWT handling, basic UI). The hardening and scalability steps provide a roadmap to evolve this MVP into a maintainable, auditable system suitable for real deployments.
The project deepened my understanding of integrating backend validation with a relational schema and reinforced the importance of security hardening for production systems.

8 Appendices

8.1 Literature review & background theory

What is CRM and VSRM?

When we talk about CRM, we often imagine a system to track prospects and customers. However, the same frameworks and tools can be used in reverse to manage upstream relationships—namely vendors, suppliers, and service providers. Here’s a closer look at how to adapt a CRM for Vendor & Supplier Relationship Management (VSRM), including key use cases, specific features, benefits, and best practices.

According to wikipedia, a definition of a CRM [1]:

> Customer relationship management (CRM) is a strategic process that organizations use to manage, analyze, and improve their interactions with customers. By leveraging data-driven insights, CRM helps businesses optimize communication, enhance customer satisfaction, and drive sustainable growth.[1]
>
> CRM systems compile data from a range of different communication channels, including a company's website, telephone (which many services come with a softphone), email, live chat, marketing materials and more recently, social media.[2] They allow businesses to learn more about their target audiences and how to better cater to their needs, thus retaining customers and driving sales growth.[3] CRM may be used with past, present or potential customers. The concepts, procedures, and rules that a corporation follows when communicating with its consumers are referred to as CRM. This complete connection covers direct contact with customers, such as sales and service-related operations, forecasting, and the analysis of consumer patterns and behaviours, from the perspective of the company.[4]

Vertical integration, as described by wikipedia again [2]:

> In microeconomics, management and international political economy, vertical integration, also referred to as vertical consolidation, is an arrangement in which the supply chain of a company is integrated and owned by that company. Usually each member of the supply chain produces a different product or (market-specific) service, and the products combine to satisfy a common need.[1] It contrasts with horizontal integration, wherein a company produces several items that are related to one another. Vertical integration has also described management styles that bring large portions of the supply chain not only under a common ownership but also into one corporation (as in the 1920s when the Ford River Rouge complex began making much of its own steel rather than buying it from suppliers).
>
> Vertical integration can be desirable because it secures supplies needed by the firm to produce its product and the market needed to sell the product, but it can become undesirable when a firm's actions become anti-competitive and impede free competition in an open marketplace. Vertical integration is one method of avoiding the hold-up problem. A monopoly produced through vertical integration is called a vertical monopoly: vertical in a supply chain measures a firm's distance from the final consumers; for example, a firm that sells directly to the consumers has a vertical position of 0, a firm that supplies to this firm has a vertical position of 1, and so on.[2]

A CRM pipeline, a concept useful within the scope of the project can be defined in the literature in a fashion similar to the definition found in [3]:

> A sales pipeline is a visual illustration depicting where prospective customers are in the sales process. It includes representations of every stage from lead prospecting to the final sale. The visualization generally consists of a bar that resembles a pipe. A sales CRM enables teams to track

and manage each stage of the pipeline.

Sales reps move prospects through the various stages in the pipeline based on their responses and interactions.

Eventually each prospect makes their way to the bottom of the sales funnel—resulting in a deal won or lost status.

This concept can be extended in b2b activities such as the ones implemented by the project CRM.\

Why Use a CRM for Vendor & Supplier Relationship Management?

- **Centralized Database:** CRMs are built to store comprehensive information about all your contacts. Instead of customers, these contacts would be supplier accounts, vendor reps, and procurement/finance team members.
- **Pipeline & Stages:** Just like a sales funnel, you can create a "procurement funnel" or
- "vendor performance funnel" with stages for negotiation, contract finalization, evaluation, and renewal.
- **Automation & Reminders:** CRMs excel at automating communications and tasks. For
- vendors, this can mean automatic reminders for contract renewals, performance evaluations, or scheduled product shipments.
- Reporting & Analytics: CRMs provide robust reporting tools. By customizing dashboards,
- you can quickly review supplier performance, contract spend, or upcoming renewals—no more uggling multiple spreadsheets or siloed systems.

Benefits of Using a CRM for Vendor & Supplier Management

- **Streamlined Vendor Onboarding:** Using pipeline-based stages to onboard new suppliers ensures that all compliance, documentation, and negotiation processes are followed consistently.
- Reduced Risk & Better Compliance: Centralized storage of SLAs, certifications (ISO, FDA approvals, etc.), and compliance forms means audits are easier, reducing regulatory risk.
- **Enhanced Visibility & Accountability:** With all vendor communication and performance history in one place, teams can make data-driven decisions, hold vendors accountable to SLAs, and reduce the chance of missed renewals or duplicative contracts.
- **Better Negotiation Power:** Historical performance data (quality, delivery speed, price changes) gives you leverage in renegotiations and helps identify opportunities for volume discounts or alternative supply sources.
- **Centralized Collaboration:** Cross-functional teams—procurement, finance, and operations — collaborate within the CRM rather than piecemeal spreadsheets or email threads.

Key CRM Features for Vendor & Supplier Management

1. Vendor Profiles & Contract Repository

- **Contact Records:** Each vendor gets a dedicated record with their profile, contract start and end dates, negotiated pricing details, contact info, etc.

- Associated Documents: Upload and store SLAs (Service-Level Agreements), NDAs, contracts, legal forms, or compliance documents for quick reference.
- Custom Fields: Add fields for preferred payment terms, lead times, quality scores, compliance requirements, or certifications.

2. Pipeline & Workflow Stages

- **Custom Pipelines:** Rename standard CRM stages (e.g., “Lead,” “Qualification,” “Negotiation”) to reflect vendor processes (e.g., “Sourcing,” “Price Negotiation,” “Contract Signed,” “Active Vendor,” “Review & Renewal”).
- **Automated Triggers:** Move a vendor record through pipeline stages; automatically generate tasks (like contract review) when a record enters a new stage.
-

3. Performance Tracking & Scorecards

- **Quality & Timeliness:** Log metrics such as on-time delivery percentage, defect rate, or quality ratings after each shipment.
- **SLA Monitoring:** Record whether the vendor meets SLA targets for response times, product quality, or service benchmarks.
- **Vendor Scorecards:** Regularly update with performance data and generate a composite score used for re-evaluations or vendor consolidation strategies.

4. Communication & Collaboration

- **Email & Call Logging:** Keep a record of all vendor communications in one place. This includes negotiations, service incidents, invoicing queries, or credit requests.
- **Collaboration Tools:** Let multiple departments (procurement, operations, finance) share updates, upload relevant docs, and discuss vendor issues within the CRM to keep everything transparent and consistent.

5. Contract & Renewal Management

- **Renewal Alerts**: Set up automated alerts well in advance of contract expiration. This gives you time to renegotiate pricing or reevaluate vendor performance.
- **Negotiation Pipelines:** Track the negotiation process for an upcoming renewal. Engage stakeholders directly in the CRM with next steps, documents, or updated quotes.
- **Milestone Tracking:** Map out specific contract milestones (e.g., new product lines, volume discount thresholds) that might trigger renegotiations or expansions.

6. Integration with Other Systems

- **Accounting & ERP:** Sync vendor data with your accounting or ERP system to ensure consistent invoice management, payment status updates, and cost analysis.
- **Inventory/SCM Systems:** A CRM used for VSRM can share real-time data with supply chain management or inventory platforms for tracking restock levels and lead times.
- Workflow Automation Tools: Use APIs or built-in connectors to trigger approval workflows

(procurement requests, purchase orders) directly from the CRM.

7. Reporting & Analytics

- **Spend Analysis:** Visualize total spend per vendor over time, broken down by product categories or service lines.
- **Comparative Performance:** Run reports comparing multiple vendors or analyzing historical performance of the same vendor, to inform strategic sourcing decisions.
- **Predictive Insights (Advanced Use):** Some CRMs support AI-driven forecasting to highlight potential risks (like vendor churn or quality deterioration) or opportunities to renegotiate pricing based on previous patterns.

How VSRM Principles Translate to Airbnb Rental Management

1. Vendor Profiles & Contract Repository

- **Cleaning & Maintenance Providers:** Each vendor or individual contractor can have a dedicated record that stores service contracts, rates, and agreements.
- **Associated Documents:** Securely upload contracts, proof of insurance, NDAs, or any certifications.
- **Custom Fields:** Add fields for types of service (cleaning frequency, deep cleaning vs. turnover service), geographic areas covered, or average response time.

2. Pipeline & Workflow Stages

- **Service Pipeline:** You could set up custom pipeline stages like “Sourcing New Vendor,” “Onboarding & Contract,” “Active Vendor,” and “Review & Renewal.”
- **Maintenance Requests:** For each property, you might track a service request through stages such as “Ticket Created,” “Assigned to Vendor,” “In Progress,” and “Completed” within the CRM.

3. Performance Tracking & Scorecards

- **Quality & Timeliness:** Keep a record of each vendor’s on-time completion rate for turnovers, average cleaning score based on guest feedback, and cost per turnover.
- **Vendor Scorecards:** Assign a performance rating after each job or monthly/quarterly review. This helps you quickly identify top-performing vendors versus those you may want to replace or renegotiate terms with.

4. Communication & Collaboration

- Conversation History: Log every phone call, email, or text message (if supported by the CRM) with cleaning crews or repair companies.
- Multi-Department Collaboration: If you have separate teams for guest communication,

property maintenance, and finances, a CRM provides a single collaboration hub. The finance team can reference invoices, the property management team can track repairs, and the guest services team can ensure the unit is guest-ready on time.

5. Contract & Renewal Management

- **Auto-Reminders:** Set reminders for when vendor agreements expire or need renewal, ensuring you have adequate time to negotiate rates or revisit performance metrics.
- **Negotiation Pipeline:** Keep track of discussions about new service packages or expanded
- coverage with cleaning/maintenance providers.

6. Integration with Other Systems

- **Property Management Software (PMS):** Many PMS platforms (e.g., Guesty, Hostfully, or Hostaway) either have open APIs or direct integrations. A CRM can pull in property and reservation data to trigger vendor tasks automatically when bookings end.
- **Accounting & Payment:** Sync with your accounting software (e.g., QuickBooks or Xero) to streamline invoices and payments for vendors.
- **Task Management:** Integrate with task automation tools like Zapier or Slack notifications to trigger cleaning tasks as soon as a booking is confirmed or canceled.

7. Reporting & Analytics

- **Cost per Turnover:** By logging vendor invoices and frequency of jobs, you can see which properties are costing the most in cleaning or repairs.
- **Vendor ROI:** Analyze patterns such as “Does vendor A provide consistently higher guest ratings than vendor B?” or “Which cleaning crew has the fastest turnaround with the fewest guest complaints?”
- **Performance Over Time:** Some CRMs can automatically generate charts and dashboards to highlight vendor performance trends, helping you spot potential issues early (like an increase in complaints at a certain property).

Benefits for Airbnb/Short-Term Rental Operators

1. **Consistent Guest Experience:** By tracking vendor performance (speed, cleanliness, reliability), you can address issues proactively, leading to better guest reviews.
2. **Streamlined Procurement & Costs:** Grouping multiple units under one cleaning contract or maintenance vendor can lower costs. A CRM helps you identify spending trends and negotiate better rates.
3. **Reduced Errors & Confusion:** No more overlooked tasks or double-booked cleaners; the CRM’s automated reminders and pipelines keep everyone in sync.
4. **Scalability:** As you add more properties to your portfolio, a centralized system scales with you. You just add new vendor records or property tasks rather than juggling more spreadsheets.

High-Level Overview

1. **Properties**: You have X rental units.
2. **Vendors**: Cleaning crew, handyman/maintenance service, linen supplier, etc.
3. **CRM Objects**: - “Contacts” for each vendor. - “Opportunities” or “Pipeline Stages” to manage turnovers, maintenance requests, or reordering supplies.

CRM Pipeline(s)

Option A: Single Combined Pipeline

Pipeline Stages might look like this:

1. **New Task** – A new turnover or maintenance request is created.
2. **Assigned** – Assigned to a specific vendor.
3. **In Progress** – Vendor is on-site or scheduled.
4. **Completed** – Work is finished.
5. **Closed & Billed** – Invoice generated, payment processed.

All short-term rental tasks (cleaning, repairs, restocking, etc.) move through these stages. You’ll use the CRM pipeline to manage each job as if it were an “opportunity.”

Option B: Multiple Pipelines (e.g., separate pipelines for cleaning vs. maintenance)

- **Cleaning Pipeline**: Tracks daily or weekly turnovers, deep cleans, linen changes.
- **Maintenance Pipeline**: Tracks repairs, emergency fixes, or routine maintenance tasks.
- **Supply Pipeline**: Tracks reorders of cleaning supplies, linens, and other consumables.

This approach can give you a clearer separation of tasks but requires more setup.

common operations in a airbnb property

1. Property Acquisition & Setup

Market Research & Acquisition - Identify target markets (urban, suburban, vacation).

- Analyze local regulations, short-term rental laws, HOA or condo board restrictions.
- Purchase or lease the property, ensuring compliance with zoning and STR (Short-Term Rental) rules.

Renovation & Furnishing - Conduct renovations or minor repairs to meet Airbnb’s quality standards.

- Furnish the space: furniture, décor, linens, kitchenware.
- Install safety features (smoke alarms, CO detectors, fire extinguishers).

Utilities & Services - Set up or transfer utilities (electricity, water, internet).

- Arrange recurring services: trash collection, lawn care, pest control if necessary.

Interior Design & Staging -Optimize layout for maximum comfort and aesthetic appeal.

- Choose a design style (modern, rustic, minimalist) that resonates with your guest demographic.

Photography & Listing Preparations

- Hire professional photographers or create high-quality photos.
- Write a compelling property description and set house rules.
- Decide on nightly rates, cleaning fees, and security deposit (if applicable).

2. Listing & Marketing

Listing Creation - Create Airbnb listing(s), add property photos, descriptions, and pricing.

- Configure advanced settings: booking window, instant booking, cancellation policy.

Channel Management - If listing on multiple platforms (Airbnb, Vrbo, Booking.com), set up a channel manager or use property management software (PMS) to sync calendars and avoid double-bookings.

Marketing & Advertising

- Use social media and local tourism boards for promotion.
- Offer promotions during off-peak seasons (weekly or monthly discounts).
- Gather guest reviews to improve listing rank.

Dynamic Pricing - Implement pricing tools or regularly adjust rates based on demand, seasonality, and local events.

- Monitor competitor listings to remain competitive.

3. Booking Management

Guest Inquiries & Pre-Booking Communication

- Respond promptly to guest questions about availability, property features, or local area.
- Pre-approve or decline inquiries based on your screening criteria (e.g., verified IDs, age, reviews).

Reservation Confirmation - Send booking confirmations, check-in instructions, and house rules.

- Collect or verify any additional documentation if needed (e.g.,
- ID, rental agreement).

Calendar Sync - Use Airbnb’s calendar or a dedicated PMS to prevent double-bookings.

- Block off owner stays or maintenance windows if you use the property personally or need downtime.

4. Guest Experience & Operations

Pre-Arrival Communication - Send automated or manual welcome messages with directions, parking info, and check-in instructions.

- Provide local recommendations or digital guidebooks (restaurants, attractions, etc.)

Self-Check-In / Meet & Greet - Set up smart locks or key safes for self-check-in to streamline guest arrivals.

- If hosting in-person, prepare a quick orientation (house rules, thermostat settings, Wi-Fi).

During-Stay Support - Stay accessible for guest questions or issues (messaging app, phone, email).

- Troubleshoot problems quickly (internet outages, appliance malfunctions, lockouts).
- Offer mid-stay cleaning or linen changes for extended bookings if advertised.

Upsells & Guest Experience
Extras (optional)

- Provide welcome baskets, local tours, or concierge services for an enhanced guest experience.
- Partner with local businesses for discounts (e.g., bike rentals, wine tastings).

5. Cleaning & Turnover Management

Scheduling Cleanings - Assign a cleaning team or contractor as soon as a booking closes.

- Create a standardized cleaning checklist covering bathrooms, kitchen, linens, and high-touch surfaces.

Supply Inventory - Track inventory of toiletries, cleaning supplies, coffee/tea, and consumables.

- Automate re-ordering or maintain a schedule for restocking.

Inspection & Maintenance - Conduct a post-cleaning inspection to ensure no damages, lost items, or missing linens.

- Log any maintenance issues (leaky faucet, broken furniture, HVAC checks) for quick resolution.

6. Maintenance & Vendor Management

Preventive Maintenance - Schedule routine checks (HVAC, plumbing, electrical) to prevent breakdowns.

- Inspect smoke detectors, CO alarms, and fire extinguishers regularly.

Repair Tickets - Log any guest-reported issues (faulty thermostat, clogged sink, broken appliance).

- Assign tasks to vendors or in-house maintenance staff via a ticketing system or CRM pipeline.

Vendor Relationship Management - Keep updated contracts, rates, and contact info for cleaning crews, handymen, electricians, etc.

- Evaluate vendor performance (response time, cost, quality of work).

7. Financial Management

Revenue Tracking - Monitor payout statements from Airbnb or other platforms, verifying income. Commission.

- Keep a ledger of nightly rates, cleaning fees, taxes, or channel

Expense Management - Track all property-related costs: utilities, insurance, HOA fees, vendor invoices, and supplies.

- Maintain documentation for accounting and potential tax deductions.

Profit & Loss Analysis - Generate monthly or quarterly reports to see net operating income (NOI) adjustments.

- Identify trends or off-season dips requiring dynamic pricing

Taxes & Compliance - Remit local occupancy taxes (lodging tax, TOT) if required in your municipality.

- File and pay annual business taxes.
- Ensure compliance with local STR regulations (permits, licensing, etc.).

8. Reviews, Feedback & Guest Relations

Post-Checkout Procedures - Request guest reviews via automated messages.

- Provide your own review of the guest (on Airbnb) which can help future hosts.

Review Monitoring & Reputation Management

- Respond politely to negative reviews and address issues (cleaning oversights, unmet expectations).
- Incorporate feedback into property improvements (e.g., better mattresses, upgraded Wi-Fi).

Guest Follow-Ups - For repeat business or loyalty, offer discounts for returning guests or referrals.

- Build an email list (if permissible) for direct booking outreach.

9. Expansion & Growth (If Managing Multiple Properties)

Scalability - Add more units or take on additional property owners for co-hosting or property management.

- Streamline daily operations with property management software (PMS), channel managers, or CRMs.

Team Building - Hire or outsource guest communications, cleaners, maintenance staff.

- Implement standard operating procedures (SOPs) and training for consistent quality across properties.

Marketing & Branding - Create a distinct brand for your Airbnb portfolio.

- Use a unified branding approach (consistent check-in info, welcome packages, design themes).

10. Offboarding & Property Exit

Plan for Offboarding - If selling or discontinuing the short-term rental, remove active listings.

- Notify upcoming guests and cancel future reservations or transfer them to another host if possible.

Inventory Removal or Transfer - Sell or move furniture, décor, and equipment if you no longer need them.

- Provide final payouts to staff or vendors, settle any open invoices.

Financial & Tax Wrap-Up

- Gather final revenue records, close out books, and update your accountant on the property sale or closure.
- Handle any prorated property taxes or transfer fees if you sold the real estate asset.

Property Sale

- List the property for sale. Some buyers see short-term rental track records as added value—consider marketing it as a turnkey STR investment.
- Provide rental history, occupancy rates, and financial statements to prospective buyers.

a set of vendor types that should log to the crm

Primary Function	Why Log in CRM
Cleaning Services / Turnover Crews	• Routine cleanings between guest stays, deep cleans, linen changes. • Track schedules, assign tasks, maintain performance reviews (on-time completion, guest feedback).
Maintenance & Handyman Services	• General property repairs (e.g., fixing doors, minor plumbing, painting). • Create “maintenance tickets” or tasks for quick assignment, track response times, and store invoices.
Laundry & Linen Providers	• Supply and clean linens, towels, and bedding. • Automate reorder or pick-up requests based on booking calendars; track costs and quality of service.
Plumbers, Electricians & HVAC Technicians	• Specialized contractors for more complex repairs or installations (water heater, electrical wiring, AC units). • Store emergency contact details, track service call history, maintain logs of completed work and parts

	used.
Pest Control / Exterminators	• Prevent or address infestations (insects, rodents). • Set recurring service intervals, log treatment history, and note warranties or special instructions.
Landscaping & Yard Maintenance	• Lawn care, garden upkeep, snow removal (for properties with outdoor spaces). • Automate seasonal tasks, track vendor costs, and manage multiple properties' outdoor schedules from a single dashboard.
Photographers & Media Professionals	• Professional listing photos, virtual tours, drone footage for marketing. • Keep a record of photo shoots, ownership rights, cost, and scheduling for refreshes or new properties.
Interior Designers & Stagers	• Furnishing, decorating, or upgrading property interiors to attract more bookings at higher rates. • Track project proposals, timelines, design budgets, and final results for each property.
Security & Smart Home Providers	• Installation and maintenance of smart locks, security cameras, alarm systems, keyless entry. • Centralize vendor contracts, log updates or software changes, track hardware warranties, and • schedule routine checks.
Insurance Agents & Legal Consultants	• Handle property insurance policies, liability coverage, legal compliance, rental permits. • Save policy documents, renewal dates, legal forms, and keep records of communication regarding claims or regulatory changes.
General Contractors & Renovation Specialists	• Larger-scale projects like kitchen/bath remodels or Document contracts, track milestones, and coordinate payment schedules. • Useful for periodic upgrades or "fix property expansions and flip" scenarios.
Real Estate Agents / Property Scouts	• Sourcing new properties for acquisition or leasing, handling property sales when offboarding. • Manage lead pipelines for new properties, log showings, and store sale/lease contract details.

map operations to vendors

Property Acquisition & Setup - Real Estate Agents / Property Scouts	- General Contractors & Renovation Specialists - Interior Designers & Stagers - Photographers & Media Professionals
Listing & Marketing	- Photographers & Media Professionals

	- Marketing & Advertising Agencies
Booking Management	- Property Management Software (PMS) Providers - Channel Managers
Guest Experience & Operations	- Cleaning Services / Turnover Crews - Maintenance & Handyman Services - Laundry & Linen Providers - Local Service Providers (for concierge services)
Cleaning & Turnover Management	- Cleaning Services / Turnover Crews - Laundry & Linen Providers
Maintenance & Vendor Management	- Maintenance & Handyman Services - Plumbers, Electricians & HVAC Technicians - Pest Control / Exterminators - Landscaping & Yard Maintenance - Security & Smart Home Providers
Financial Management	- Accounting & Tax Services - Insurance Agents & Legal Consultants
Reviews, Feedback & Guest Relations	- Reputation Management Firms - Marketing Consultants
Expansion & Growth - Real Estate Agents / Property Scouts	- Property Management Software (PMS) Providers - Marketing & Branding Consultants
Offboarding & Property Exit Real Estate Agents / Property Scouts	- General Contractors & Renovation Specialists (for final upgrades)

pipelines

General Task Management Pipeline (Combined)	- New Task - Assigned - In Progress - Completed - Closed & Billed
Cleaning Pipeline (Turnover Management)	- Booking Confirmed - Cleaning Scheduled - Cleaning in Progress - Inspection Completed - Ready for Guest
Maintenance Pipeline - Maintenance Ticket Created	- Assigned to Vendor - In Progress - Awaiting Parts (if needed) - Completed - Closed & Billed
Supply & Inventory Management Pipeline	- Reorder Needed - Order Placed - Order Shipped - Order Received - Inventory Updated
Vendor Onboarding & Contract Management Pipeline	- Sourcing New Vendor - Vendor Evaluation - Contract Negotiation - Contract Signed - Active Vendor - Review & Renewal
Property Acquisition Pipeline (Expansion & Growth)	- Property Identified - Under Contract - Due Diligence & Inspections - Purchase Finalized - Ready for Operations
Guest Booking & Reservation Management Pipeline	-Inquiry Received - Reservation Pending - Reservation Confirmed - Pre-Check-In Communication Sent - Check-In Completed - Stay Completed - Post-Stay Follow-Up Sent
Marketing & Advertising Pipeline	- Campaign Idea Generated - Campaign Plan Approved - Content Creation in Progress - Campaign Launched - Campaign Performance Analyzed - Campaign Closed

Contract & Legal Compliance Pipeline Financial Management & Expense Tracking Pipeline	- Compliance Review Needed - Legal Review in Progress - Documents Submitted - Approval Pending - Compliance Approved - Invoice Received - Payment Scheduled - Payment Processed - Expense Reconciled - Report Generated
Property Maintenance & Renovation Projects Pipeline	- Project Idea Created - Budget Approved - Vendor Assigned - Work In Progress - Inspection Completed - Project Closed
Vendor Evaluation & Performance Review Pipeline	???- Evaluation Scheduled - Vendor Data Collected - Review Conducted - Performance Scored - Contract Renewed/Terminated
Emergency Response & Incident Management Pipeline	- Incident Reported - Emergency Vendor Assigned - Issue Being Resolved - Follow-Up Inspection - Incident Closed
Long-Term Expansion & Portfolio Growth Pipeline	???- Market Research Completed - Potential Property Identified - Investment Approved - Property Acquired - Property Onboarded
Crisis Management & Disaster Recovery Pipeline	- Incident Detected - Initial Assessment Completed - Emergency Response Activated - Vendors Assigned - Issue Resolved - Recovery & Post-Mortem Analysis
Guest Complaint & Dispute Resolution Pipeline Legal & Compliance Escalation Pipeline	- Complaint Received - Issue Investigated - Vendor/Team Assigned - Resolution Proposed - Resolution Accepted or Disputed - Case Closed - Issue Identified - Legal Review Initiated - Documents & Evidence Collected - Legal Action Taken

	- Case Settled or Closed
Insurance Claim & Damage Assessment Pipeline	- Incident Reported - Insurance Company Notified - Assessment Scheduled - Claim Filed - Claim Approved or Denied - Claim Closed
Security & Data Breach Response Pipeline	- Breach Detected - Investigation Launched - Security Vendor Assigned - Vulnerability Resolved - System Secured - Breach Closed
Pandemic Response & Health Safety Pipeline	- Health Advisory Issued - Enhanced Cleaning Protocols Activated - Vendor Training Conducted - Health Inspection Scheduled - Compliance Verified - Operations Normalized
Host Exit & Property Liquidation Pipeline	- Host Decision Received - Property Valuation Completed - Property Listed for Sale - Buyer Identified - Sale Finalized - Property Offboarded

the difference between an operation and a pipeline

The difference between an **operation** and a **pipeline** in a CRM context lies in their definitions, roles, and how they function within the system:

1. Operation

Definition	An operation refers to a specific business process or task performed within the CRM. It represents **what needs to be done** and **who performs it.**
Key Characteristics	• **Action-Oriented:** Operations involve tasks like booking management, cleaning assignments, or vendor contract negotiations. • **Defined by Scope:** Each operation has a clear beginning and endpoint. • **Task-Level Details:** Often broken into tasks or activities (e.g., sending booking confirmation emails, assigning a maintenance request).
Examples in Airbnb Management:	• Scheduling a cleaning service. • Processing guest reservations.

	• Conducting maintenance repairs.

2. Pipeline

Definition:	A pipeline is a **structured workflow** that visualizes the sequence of stages through which an operation progresses until completion. It represents **how the process flows.**
Key Characteristics:	• **Process-Oriented:**A pipeline focuses on managing the stages of an operation. • **Stage-Based Progression:** Operations move through defined stages within the pipeline. • **Automation & Tracking:** Pipelines help track progress, automate tasks, and ensure consistency.
Examples in Airbnb Management:	• A **Cleaning Pipeline** with stages like "Scheduled," "In Progress," and "Completed." • A **Vendor Onboarding Pipeline** with stages like "Vendor Evaluated," "Contract Signed," and • "Active Vendor."

Summary: Difference Between Operation & Pipeline

Aspect	**Operation**	**Pipeline**
Definition	Specific task or activity	Workflow process structure
Focus	Task execution	Process management
Detail Level	Granular tasks or actions	High-level process stages
Progress Tracking	Tracks completion of tasks	Tracks stage transitions
Automation	Can be automated or manual	Automation-driven
Examples	Booking a stay, scheduling repair	Booking pipeline, vendor onboarding pipeline

Vendor-pipeline map

Pipeline	Vendor
Vendor Pipelines Mapping	Cleaning Services/Turnover Crews, Laundry & Linen Providers
Maintenance Pipeline	Maintenance & Handyman Services, Plumbers, Electricians & HVAC Technicians, Pest Control/Exterminators, Landscaping & Yard Maintenance, Security & Smart Home Providers
Supply & Inventory Management Pipeline	Laundry & Linen Providers, Cleaning Supply Providers, Inventory Suppliers
Vendor Onboarding & Contract Management Pipeline	All service providers including cleaning crews, maintenance vendors, and specialty contractors
Property Acquisition Pipeline	Real Estate Agents/Property Scouts, General Contractors & Renovation Specialists, Interior Designers & Stagers, Photographers & Media Professionals
Guest Booking & Reservation Management Pipeline	Property Management Software (PMS) Providers, Channel Managers
Marketing & Advertising Pipeline	Photographers & Media Professionals, Marketing & Advertising Agencies
Contract & Legal Compliance Pipeline	Insurance Agents & Legal Consultants
Financial Management & Expense Tracking Pipeline	Accounting & Tax Services, Insurance Agents & Legal Consultants
Property Maintenance & Renovation Projects Pipeline	General Contractors & Renovation Specialists, Maintenance & Handyman Services
Vendor Evaluation & Performance Review Pipeline	All operational vendors (cleaning, maintenance, suppliers)

CRM performance metrics

1. Vendor Performance Metrics

What	Track delivery timelines, defect rates, and quality scores.
Who	Procurement team, Odoo developer, and data analyst.
Where	CRM (Purchase and Inventory modules).
When	Initial setup in the first implementation phase,

	ongoing updates during procurement cycles.
Why	To assess and improve vendor reliability and performance.
How	- Add custom fields in Purchase Orders. - Use pandas for aggregating and analyzing delivery data. - Develop dashboards for real-time monitoring.

2. Cost Analysis

What	Analyze expenses and vendor-specific costs.
Who	Finance team, consultant, and data analyst.
Where	Accounting module.
When	During setup and monthly reviews.
Why	To manage and optimize vendor-related expenses.
How	- Create analytic accounts for vendors. - Use pandas and numpy for expense analysis. - Generate reports using Odoo's Reporting module.

3. Operational Efficiency

What	Optimize task pipelines and track completion times.
Who	Operations team, project managers, and developers.
Where	Project module.
When	Initial pipeline setup, reviewed quarterly.
Why	To streamline workflows and improve productivity.
How	• Set up workflows and stages in the Project module. • Use Python to calculate and visualize efficiency metrics. • Automate time-tracking with Odoo Timesheets.

4. Comparative Vendor Analysis

What	Compare vendors based on performance metrics.
Who	Procurement and analytics teams.
Where	CRM and Purchase modules.
When	Monthly or quarterly performance reviews.
Why	To identify top-performing vendors and areas for improvement.
How	

	• Integrate pandas and matplotlib for data visualization. • Automate ranking updates.

5. Predictive Insights

What	Predict late deliveries and defect risks.
Who	Data science team, developers.
Where	-
When	After collecting at least 3-6 months of historical data.
Why	To proactively address potential risks.
How	• Train predictive models using TensorFlow.

6. Guest Feedback Integration

What	Link guest feedback to vendors for quality monitoring.
Who	Customer service team.
Where	Helpdesk and CRM modules.
When	As feedback is received, reviewed monthly.
Why	To ensure vendor performance aligns with guest expectations.
How	• Create custom fields for feedback scores. • Use sentiment analysis with TensorFlow. • Build feedback dashboards.

7. Contract and Renewal Analytics

What	Manage contract details and automate renewal alerts.
Who	Procurement team, developers.
Where	Purchase and Calendar modules.
When	vendor onboarding and contract renewals.
Why	To avoid missed renewals and strengthen vendor relationships.
How	• Log contracts and set reminders in Calendar. • Automate task creation for renewal discussions.

8. Custom Dashboards

What	Centralize key metrics for stakeholders.
Who	developer, data analyst.
Where	-
When	After all modules are customized.
Why	To provide actionable insights.
How	Develop Python scripts for real-time updates.

9. Integrated Insights

What	Combine vendor, booking, and expense data.
Who	IT and data teams.
Where	APIs, CRM, Accounting modules.
When	Once initial data collection is completed.
Why	To provide unified reporting.
How	• Fetch data via APIs. • Merge and cleanse data using pandas.

10. Advanced Analytics

What	Implement machine learning for churn prediction and value estimation.
Who	Data science team, developers.
Where	AI tools.
When	After building a robust historical dataset.
Why	To make data-driven decisions.
How	• Train machine learning models externally. • Visualize insights in custom dashboards.

SQLite table foreign key concistency.

The main strategy for foreign key consistency used in sqlite literature is either through PRAGMA declerations in the evironment, or the use of triggers before or after database operations. This is stressed here because of the uniqueness of this feature in Sqlite and because this can be a source of errors during and after the development stages of the service. The list of triggers developed is listed in the appendix section.

4.Design Rationale

How pipelines map to vendor operations.

Each pipeline of the deliverable is focuced on rental properties. From this standpoint, these pipelines have specific parties involved, contancted and coordinated by the property owner/manager. Within this context, it is communicated what outcomes are expected by the parties (hence the need for communication tracking in the database)

Custom schema elements and why they were added.

Each table, rest api endpoint and user interface menu in the deliverable represents the management of the pipelines needed for the properties management. Simplicity and efficiency are key to convey meaning, use, and results.

Choice of Bottle vs FastAPI, SQLite vs Postgres, etc.

SQLite was selected because it has been used and proven to be stable as a good choice for small-medium projects in a very broad spectrum of environments, from embedded devices, browsers or web services such as the deliverable of this assignment.

For this MVP the project uses Bottle rather than FastAPI. Bottle's minimal design (single-file, zero dependencies) enabled rapid prototyping and made the internal mechanics (routing, request/response cycle) transparent for educational purposes. This simplicity allowed us to focus on designing validators, repositories, and authentication from first principles. However, this choice comes with trade-offs: unlike FastAPI, Bottle does not provide built-in request validation, OpenAPI generation, or async support. These had to be implemented manually, which increased boilerplate but also improved understanding of the system. In a production environment, a migration to FastAPI (or a similar async framework) would be advisable, as it offers Pydantic-powered DTOs, automatic schema generation, and better scalability. Thus, Bottle is appropriate for an educational MVP, while FastAPI would be a natural choice for an industrial-grade continuation.

Choices of design details

- The use of 00.00.0000 as a default date record is a done as a concious, database engine neutral design choice as the date format as text with the form DD.MM.YYYY; this date sintax as text record is the absolutely minimal and portable way to represent dates across databases and time representation standards and allow for ordering with minimal effort when properly handled (YYY.MM.DD reformat allows for ascending by date regardless of environment.)
- The use of 00.00.0000 as a default date record is also a choice for neutral sentinel value in the event of database migration versus the use of nulls in the schema.
- The users table is implemented at a minimal level, as a table but nothing more. This is also a concious choice; It can allow for better record tracking but also adds up a technical dept in implementing features and tests that utilise it. It is left as an implementation feature stub for development at a later time.
- Beyond a recommendation of how scorecards are represented, there is no other notion of HOW scores are generated by the user entering the data. This is a nuanced issue; not all companies follow the same scoring standards and tests nor should anyone, the developer or the client should leave the decicions of assestment to an AI without prior options taken into

consideration. So, as a design decicion, a table for how a scorecard should look like is used. Users can use this as is with data entry and a manager do the assestment; expand the service to include scoring as a coded algorithm or implement something completely bespoke.

Choices of language, web framework, user interface and database details

- Bottle is minimal REST framework designed to allow for little to no effort in setting up and minimal training for a developer to begin writting production grade code. It has a good standing in the python community and is good at delivering what it promises.
- Sqlite is a industry tested and hardened embedded SQL database. It resolves the issue of data persistence by allowing data to be stored on premice with the rest of the codebase. One can trace the root cause of failure on one source; the python interpreter dealing with the business logic or the database library. Data migration is a breeze; one caries the dataset in a single file from one location to the other.
- Use of html FILES and bootrap was chosen over a SPA approach because bootstrap and HTML are a more modular approach to web development. Each component and service is a standalone single point of failure; the service lives on regardless of features being developed and succeed or fail during production. Users do not have to and should not have to deal with the impact of new features implemented to their daily production schedule.

Argunemts on the need of a query and a api url path generator

In the surface level bottle and a lot of REST frameworks imply each path is it's own island. This implies a function like structure over the web; a REST service api endpoint is called. It parses its path arguments or POST data request package from the user. Then it handles its tasks on the premise of the user data it received and once the api endpoint code was complete the user would be served with an output.

Within the scope of this project, there are two fronts to deal with:

- The service can and will be bound to SQLite and any migration will be hindered by all of the boilerplate SQL codebase designed to work with SQLite
- The service will be unreadable; no developer out of the author will be (in a reasonable timeframe) able to make sense of the codebase and the way it works.

For these reasons a query gennerator codebase was placed early on; this allows for a compromise between verbose codebase and a full ORM system allowingn for seperation between REST service and database and code simplicity.

For a better understanding of the simplification level this type of abstraction ofters to the table, this is a test sniped shipped with the codebase dealing with inserts to the database:

file: db.pytest/test_db_schema_crud.py

```
def insert_baseline(con: sqlite3.Connection):
    cur = con.cursor()
    now = "2025-01-01"

    # vendors
    cur.execute(
        """
```

```
        INSERT INTO vendors(name, vendor_type, status, phone, email, address, created_at,
updated_at)
        VALUES (?,?,?,?,?,?,?,?)
        """,
        ("Acme Cleaners", "cleaning", "active", "+30-210-0000000", "acme@example.com", "Main
St 1", now, now),
    )
    vendor_id = cur.lastrowid

    # properties
    cur.execute(
        """
        INSERT INTO properties(property_name, address, status, created_at, updated_at)
        VALUES (?,?,?,?,?)
        """,
        ("Apt 101", "Athens", "active", now, now),
    )
    property_id = cur.lastrowid

    # owner
    cur.execute(
        """
        INSERT INTO owner(first_name, last_name, phone, email, address, legal_id, created_at,
updated_at)
        VALUES (?,?,?,?,?,?,?,?)
        """,
        ("Maria", "Papadopoulou", "+30-210-1111111", "maria@example.com", "Athens",
"GR123456", now, now),
    )
    owner_id = cur.lastrowid

    # owner_property_map
    cur.execute(
        "INSERT INTO owner_property_map(owner_id, property_id) VALUES (?,?)",
        (owner_id, property_id),
    )
    map_id = cur.lastrowid

    # pipelines
    cur.execute(
        """
        INSERT INTO pipelines(pipeline_name, description, created_at, updated_at)
        VALUES (?,?,?,?)
        """,
        ("Cleaning", "Turnover cleaning pipeline", now, now),
    )
    pipeline_id = cur.lastrowid

    # pipeline_stages
    cur.execute(
```

```
        """
        INSERT INTO pipeline_stages(pipeline_id, stage_name, stage_order, created_at, updated_at)
        VALUES (?,?,?,?,?)
        """,
        (pipeline_id, "Scheduled", 1, now, now),
    )
    stage_id = cur.lastrowid

    # contracts
    cur.execute(
        """
        INSERT INTO contracts(vendor_id, contract_name, start_date, end_date, renewal_terms,
created_at, updated_at, status)
        VALUES (?,?,?,?,?,?,?,?)
        """,
        (vendor_id, "2025 Cleaning", now, None, "auto-renew", now, now, "signed"),
    )
    contract_id = cur.lastrowid

    # documents
    cur.execute(
        """
        INSERT INTO documents(vendor_id, contract_id, file_name, file_path, uploaded_at,
updated_at)
        VALUES (?,?,?,?,?,?)
        """,
        (vendor_id, contract_id, "contract.pdf", "/docs/contract.pdf", now, now),
    )
    document_id = cur.lastrowid

    # vendor_contacts
    cur.execute(
        """
        INSERT INTO vendor_contacts(vendor_id, first_name, last_name, job_title, phone, email,
created_at, updated_at)
        VALUES (?,?,?,?,?,?,?,?)
        """,
        (vendor_id, "Nikos", "K.", "Manager", "+30-210-2222222", "nikos@example.com", now,
now),
    )
    contact_id = cur.lastrowid

    # vendor_scorecard
    cur.execute(
        """
        INSERT INTO vendor_scorecard(vendor_id, period_start, period_end, quality_score,
on_time_delivery, defect_rate, comments, created_at, updated_at)
        VALUES (?,?,?,?,?,?,?,?,?)
        """,
        (vendor_id, "2025-01-01", "2025-03-31", 95, 98, 1.2, "Great start", now, now),
```

```
    )
    scorecard_id = cur.lastrowid

    # communications
    cur.execute(
        """
        INSERT INTO communications(vendor_id, date_sent, comm_type, subject, content, user_id,
followup_needed, created_at, updated_at)
        VALUES (?,?,?,?,?,?,?,?,?)
        """,
        (vendor_id, now, "email", "Kickoff", "Welcome aboard", None, "0", now, now),
    )
    communication_id = cur.lastrowid

    # opportunities_task_tickets
    cur.execute(
        """
        INSERT INTO opportunities_task_tickets(pipeline_id, stage_id, vendor_id, title, description,
property_id, status, priority, due_date, created_at, updated_at)
        VALUES (?,?,?,?,?,?,?,?,?,?,?)
        """,
        (pipeline_id, stage_id, vendor_id, "Turnover #1", "Clean after checkout", property_id,
"open", "high", "2025-01-02", now, now),
    )
    ticket_id = cur.lastrowid
```

The sql abstraction class collection delivered with this project is at it’s base a sql statement generator; a thin layer between the service and the database. Not a full ORM system. Yet, this is the amount of complexity demonstrated that is simplified.

This sql example in itself does cover only half the story: the requirements involved for this project enforce an endpoint for every table. This in turn double the amount of boilerplate code which create a cascade effect that can only be described as detrimental for the project quality; t**he sort of abstraction used should and can be used to improve developer productivity once the system behaviour is understood.**

In a mirroring fashion, the urls being handled dynamically ensures similar boilerplate avoidance on the REST api front.

Arguments on the use or not of DTOs

or a Bottle/Flask-scale app where you already pass JSON dicts end-to-end and validate them, full DTO classes everywhere would be overkill. You’d pay in indirection and churn with little real gain. But adding thin, typed schemas only at the boundaries (request/response) gives you the safety you want without mangling the codebase—and it also ticks the “DTOs” box from the brief.

When DTOs help vs. hurt in Python

DTOs help when:

- You need explicit contracts at system boundaries (incoming HTTP requests, outgoing responses, integrations).
- Multiple teams/clients consume the API and compatibility must be enforced with versioning.
- You want type-checked refactors (rename a field, know exactly what breaks).
- You generate or validate OpenAPI mechanically against the same source of truth.

DTOs hurt when:

- You already have a small framework + simple validators and most logic is CRUD.
- The app mostly shuttles dicts from HTTP → validator → CRUD with little domain behavior.
- You'd end up writing mappings that are just 1:1 field mirrors (dict ↔ DTO ↔ dict) for dozens of entities—extra code, more places to drift.
- Python's dynamic typing makes rigid DTO hierarchies feel noisy compared to Java's normal.

So: don't introduce a wall of DTO classes across the codebase. Keep dicts internally where they're clear and fast. Use lightweight schemas only where they buy you something (API edges).

A minimal, high-leverage pattern (no code mangle)

Boundary-only models (thin DTOs).	Use pydantic (or dataclasses + validators) to define request/response models per endpoint. Keep them flat and narrow—just what crosses the wire. Benefit: strong validation + clear docs. Internally still use plain dicts/records; convert at the edge.
Typed dicts for internals.	Where you want editor/type-checker help without ceremony, use typing.TypedDict for common shapes (e.g., Vendor, Contract). No runtime overhead, zero mapping cost.
One translation hop, only at the edge.	Route handler: HTTP JSON → Pydantic Model (validate) → dict for the CRUD layer. On responses: dict → Pydantic Model → JSON. That's it. No "DTO all the way down."
Keep validators, just shift them.	Move field-level checks into the Pydantic models (regex/date/enum). Keep cross-record/DB-level checks in your existing validators/CRUD (uniqueness, FK existence, trigger expectations).

Align names with OpenAPI.	Define model fields to match the spec's id names (e.g., vendor_contact_id). That solves the spec/code drift and gives you self-documenting responses.
Skip XML unless required.	If you must support XML, serialize at the very edge (model → dict → XML). Don't propagate XML concerns into your domain types.

What this ofters is clarity at boundaries: exact shapes, helpful errors to clients, better docs.

Low intrusion: internal code stays compact; no ping-pong mapping all over.

Assessment win: one can honestly say "DTOs implemented where appropriate," satisfying the brief without bloating Bottle code.

Future-proofing: if Bottle is replaced for FastAPI, these models drop right in.

"DTOs encapsulate but also obfuscate; in Java they're natural, in Python they can be overkill."

In Python, selective DTOs (edges only) strike the right balance: you gain contracts where they matter and keep the small-framework ergonomics you chose (Bottle) everywhere else.

Issues with the owner_property_map table

From your table definitions so far:

- Table: `owner_property_map(map_id INTEGER PRIMARY KEY AUTOINCREMENT, owner_id, property_id, …)` (no `UNIQUE(owner_id,property_id)`).
- Validator: requires `owner_id` and `property_id` for create; `map_id` is *not* part of the validator.
- Routes: registered with a single PK (`map_id`) for PUT/DELETE.

That combination leads to these concrete problems:

1. Duplicate relationships are possible

Because there's no `UNIQUE(owner_id, property_id)`, one can insert the same owner ↔ property pair multiple times. Symptoms:

- Range/list endpoints will display duplicates.
- "Update by pair" in the UI is ambiguous — which row does the user edits?
- Business logic like "assign this owner to that property once" isn't enforced.

2. Updates can create accidental duplicates

An update allows changing `owner_id` or `property_id`. A user can edit row A to a pair that already exists in row B, silently creating two identical relationships. Later deletes by pair in the UI will seem to "miss" one record unless you always act by `map_id`.

3. Mismatched identifiers across layers

- DB column is `map_id`; your OpenAPI and/or UI may refer to `owner_property_map_id`. If those names aren't consistently mapped, devs (and tests) use the wrong field, causing 400/404s on PUT/DELETE.
- Validators don't include `map_id`, so if you ever validate full objects on update, they'll reject bodies that include the id (or let them through but ignore them), which is confusing.

4. Delete UX friction

The safest delete is by `map_id` (since it's the true PK). But the UI usually knows only the pair (owner, property). Without a uniqueness guarantee, "delete by pair" can remove one duplicate and leave ghost entries.

Quick, low-risk fix (keep `map_id`)

This preserves the route factory (single-PK assumption) and keeps your current code simple.

DB migration

```
-- Add uniqueness so each owner↔property pair can appear only once
CREATE UNIQUE INDEX IF NOT EXISTS uq_owner_property
ON owner_property_map(owner_id, property_id);
```

Validator

Keep as-is for create (`owner_id`, `property_id` required).

For update, either:

forbid mutating the pair to avoid semantics-changing updates:**

- Reject bodies that include `owner_id` or `property_id`, or
- Allow it but rely on the unique index to raise a 409 if a duplicate would be created.

Routes/OpenAPI/UI

Standardize the id name everywhere (choose `map_id` for both DB and DTO), e.g.:

- `PUT /owner_property_map/{map_id}`
- `DELETE /owner_property_map/{map_id}`
- Show `map_id` in the UI list/table so users can target the correct row.
- Document 409 conflict on duplicate create/update.

Tests to add (fast)

- Create the same pair twice ⇒ 409.
- Update row A's pair to match row B ⇒ 409.
- Delete by `map_id` ⇒ **200**; subsequent GET shows it's gone.
- PUT/DELETE with non-existent `map_id` ⇒ **404** (or **400**) — pick and document.

Alternative (composite PK)

The pair can be the identity:

Schema

```
CREATE TABLE owner_property_map (
  owner_id    INTEGER NOT NULL,
  property_id INTEGER NOT NULL,
  created_at  TEXT NOT NULL,
  updated_at  TEXT NOT NULL,
  PRIMARY KEY (owner_id, property_id),
  FOREIGN KEY(owner_id)    REFERENCES owner(owner_id),
  FOREIGN KEY(property_id) REFERENCES properties(property_id)
);
```

Stack changes

- Routes: `PUT /owner_property_map/{owner_id}/{property_id}`, `DELETE /owner_property_map/{owner_id}/{property_id}`.
- Validator: treat the PK as a tuple; for updates, you probably **forbid changing** either key (or implement delete+recreate semantics).
- UI: forms carry two ids.
- Tests: target records with the pair.

This is clean relationally, but it’s a larger refactor for the current route-factory pattern.

5.LLM Integration

To assist in the development of the codebase, AI tools were used to assist with editing, diagram rendering, and phrasing improvements. The design, code, and analysis remain the author’s own work.. This choice was intentional as the project scope is ideal to stress test such models and coding assistants to the limits of their capabilities. For this project, chatGPT 4o, o3 and 04-mini-high were the main core of tools used in this front with experiments done with local models used (with varying levels of success) .It also provided allowed the author to use training in tools not familiar with in the past (web development and database management with python, unit testing etc). This is not ment to be the code was utilised intact; in fact the contrary. What will follow as to the methodology used to produce the final production codebase, what changed, why, what problems did come up, how they were resolved.

Given the rise in the usage of LLMs during the software development stages the author decided given the oportunity of the present bootcamp to stress test the coding assistant capabilities of chatGPT by using it during all of the production stages of this assignment. This section deals with the framework used, why this framework was used, the outcomes, what issues came up and how were they overcomed.

The framework used

Step 1: LLM used methodology overview

- An early on decicion was the avoidance of complex llm techniques used like chain of thought [5]. This was done to avoid the overall planing and complexity of the discussions.
- An exception to the previous rule was the use of RAG (retrieval augmented generation) [7], a process which allows for llm models to reason with the context of documents provided by the user. This allowed for a more robust set of discussion patterns that will be discussed later.
- An early decision to avoid any risk of halucinations and generational errors, the discussions would be limited within the scope of states within the waterfall model [6]. It also allowed for more nuanced discussions on each topic involved for each steps of Software Requirement Analysis, Preliminary Design, Detailed Design, Coding and Unit Testing, Integration, and Testing. Finally, this approach allowed for incremental changes of previous stages outcomes on a simple choreographed approach that would not allow for unecessary discussion branching or extended time periods spent in dealing with llm halucinations.
- LLM usage in this project had two fronts to tackle: 1) code feedback when requested 2) text feedback when requested.
- Point 1 of the previous point involves actual feedback; chatGPT was my coding assistant. Another coding functionality was code refactoring. This second functionality is a high risc high yield operation done because it involves the creation of code by an LLM both riddled with hallutinations and errors. In the project scope this risc yielded auctionable results.
- Point 2 of the previous point involved LLMs working as proofreaders and writing assistants; reducing the workload of providing documentation deliverables as much as insights as to improve the quality of this document.

In the next steps there will follow a report on the chatGPT’s main contributions over the aspect of the project. This will not be detailed in a point for point way; discussions analysis covering the span of 6 months dealing with all of the stages of software development goes beyond the scope of this document. The volume of said discussions demands a seperate analysis of what the contributions were and what shortcomings were present. Finaly, a more formal strategy of prompt engineering emerging from these discussions is another case study that is both useful, and beneficial but goes also well beyond the scope of this document.

Step 2: Software Requirement Analysis

There was no actual involvment of LLMs during this step beyond generic discussions over the CRM landscape. Notes were taken in the event they prove useful later on.

Step 3: Preliminary design

Refinement of the concepts involved with the theory was done during this stage. More notes taking, and a sketch of things to come.

Step 4: Detailed design

database design, coding languages / environments and framworks selection was done here. A lot of the discussions with chatGPT was mostly on table schemas. A lot of experimentation with java,

javascript (node.js) and noSQL was done here; mostly for weight optioning than selection for production.

Within the scope of the project, python was selected as the main backend development language, with a REST micro-framework, bottle dealing with the REST API development. As a database, sqlite was selected for the simplicity and portability it provides as much as the testing and scrutiny it has been subjected to proving to be a viable solution for small-medium projects.

Step 5: coding and unit testing

- Development was done by the user with the frameworks/libraries user manuals and examples.
- JWT integration is an exception as a crash course on its use was required. ChatGPT was an excelent teacher and assistant, providing examples and guidance through the process.
- Priority was given to the backend over the frontend. This choice was intentional; black box testing can be done on the backend code with simple tools like curl to check code functionality (the core deliverable operations are data operations. Check for functionality is check for successfull database operations).
- Once the backend black box check was completed the frontend development began. This effort was spearheaded with the use of chatGPT for time constraints reasons. The use of bootstrap over the use of tools like SPA frameworks like react or angular was driven by two reasons :1) complexity. The project scope is too small for the use of SPA frameworks. Individual pages for each functionality can be more easily be tested 2) rapid LLM code development. Each page was developed iteratively, tested for functionality with the backend and integrated to the project.

A special note on refactoring should be adressed. There were refactory sprints early on; bottle framework applications output repetative code; Database code often suffers from boilerplate code. During one of the sessions, chatGPT refactored the code in its current OOP form. The resulting code had to be refined as to eliminate errors, but the net results were far better than the original (verbose) codebase. The codebase is also more modular; there is a distinct and clean class hierarchy with specific roles. Examples focusing on the database side of this are discussed in other parts of the document.

Step 6: integration

The project deliverable has discrete sections:

- The database
- The backend
- The frontend
- The tests

Each section builds on deliverables from the previous step. In this stage, the use of RAG (retrieval Augmented Genneration) in llms allows for the creation of context for the user prompts; the user can craft documents descibing the project and then use his prompts over the document data.

Example:

- Create the database schema, SQL create prompts, selects etc
- Use the database info to generate a REST api to manipulate the database

- Use the REST api to create specs for the frontend, then use the specs to create frontend in the desired environment (HTML, android etc)

Step 7: Testing

The workflow from the previous steps hides a beneficial side effect: manual black box testing by the developer. LLMs can provide further feedback on ways to implement or improve testing strategies.

Unit tests were employed where and when needed for proof of proper functionality with pytest library. Testing further using tools like headless browsers can harden further tests cases on more complex scenarios.

why this framework was used

This approach enforces a man in the loop approach; an analogy can be drawn with the medical field where doctors perform a diagnosis with ai tools; not the other way around. This has specific implications during production:

- Position LLMs are tools, Not Authors: LLMs were used to speed up repetitive tasks (SQL boilerplate, initial validator scaffolds, route factory patterns); the human developer acted as chief architect and QA lead—selecting, adapting, testing, and integrating only what passed review; the LLM did not “own” the design decisions; the developer did. This is closer to how a doctor uses an MRI machine: the AI surfaces candidates, the human confirms and prescribes.
- LLMs have known limits; some practical, other more theoretical. A special section of this was mentioned briefely before in this document; another more extensive will follow. The approach used allowed for minimisation of errors like type inference errors, errors in sql trigger logic or hallucination during any stage of software development.

Theoretical reasoning behind this framework

Empirical studies have been made ([8]) with regards to llm hallucination classification. Discussions on the subject ([9]) also can yield theoretical limits, with Rice-Shapiro and KLST theorems being notable cases of possible limiting factors in coding and LLMs. Within the scope of what can go wrong and possible reasons why, the choice of using some form of structure or guidelines as to **how** llms can effectively utilised in code develompent is important.

Why practical LLM errors have value and should be taken into account before using LLMs during production: Because we already have a body of knowlege as to what can go wrong using LLMs, and more specifically what can go wrong during code production.

Why theoretical LLM errors have value and should be taken into account before using LLMs during production

According to wikipedia ([10]):

> In computability theory, the Rice–Shapiro theorem is a generalization of Rice's theorem, named after Henry Gordon Rice and Norman Shapiro. It states that when a semi-decidable property of partial computable functions is true on a certain partial function, one can extract a finite subfunction such that the property is still true.
>
> The informal idea of the theorem is that the "only general way" to obtain information on the behavior of a program is to run the program, and because a computation is finite, one can only try the program on a finite number of inputs.
>
> A closely related theorem is the Kreisel-Lacombe-Shoenfield-Tseitin theorem (or KLST theorem), which was obtained independently by Georg Kreisel, Daniel Lacombe and Joseph R. Shoenfield, and by Grigori Tseitin.

Within the context of LLMs and code development, we try to move from text (prompts) to code. And we do this with a model that will try to reason on properties of what we ask of it, why and how; which belongs to the Rice-Shapiro theorem domain. This is before we actually know about the llm code reply; which **even when we manage to navigate away from hallucinations and syntax errors will still fall under the risc of logical error and will still be tested with more rigour by compilers, interpreters and code analysis tools or be the subject of unit testing or formal methods of software analysis before we reach to conclution that it works.**

The necessity of a human man-in-the-middle when incorporating LLM-generated code can also be defended from the perspective of computability theory. Rice’s theorem establishes that any non-trivial semantic property of programs is undecidable in the general case. Put differently, there exists no algorithm that can determine, for every possible program, whether it is “secure,” “correct,” or “free of defects.” The Rice–Shapiro theorem refines this picture by showing that some properties are only semi-decidable: it is possible to detect certain violations (e.g., a type error or a missing field), but impossible to prove universal correctness.

Applied to the context of machine-generated code, this result implies that no fully automated pipeline can ever guarantee the trustworthiness of LLM output. Static analyzers, linters, type checkers, and contract tests can (and should) catch specific classes of errors, but by theoretical necessity they cannot provide complete assurance. What remains undecidable must be resolved at the socio-technical level — by human judgment, review, and accountability.

Therefore, the argument for a human intermediary is not merely pragmatic, based on anecdotal drift or inconsistency observed in practice. It is also theoretically mandated: computability limits rule out the possibility of a completely automated vetting process. The role of the man-in-the-middle thus represents an engineering adaptation to a deeper truth about the nature of programs. In combination with empirical evidence from the literature on LLM risks, this theoretical backdrop secures the position that structured human oversight is indispensable.

Provenance & Verification Table

This table is an categories and concrete example list of things gone wrong, how/if they were fixed and can be a reference list of concrete examples within this project.

Artifact	**Origin (Human / LLM / Hybrid)**	**Risks from LLM use**	**Verification Method**	**Evidence in Repo/Docs**
Database schema (DDL)	Initial LLM draft (GPT-4o), then human-edited	Type drift (e.g. `stage_name` → INTEGER), inconsistent defaults	Unit & black-box tests; manual ERD vs schema check	Schema definitions, validator tests, test logs
Triggers / Constraints	LLM suggested, human curated	Invalid SQL, missing FK checks	Only passing triggers kept after test suite	`tests/ test_triggers.sql`, black-box test plan
Validators (Python)	LLM scaffold, human refinement	Wrong types, missing optional handling	Unit tests per entity, date-format enforcement	`validators.py`, `tests/test_validators.py`
REST routes (Bottle)	Human mapping + LLM boilerplate	Naming drift (`opportunities_task_ticket` singular vs plural)	Black-box curl runner; manual endpoint registry review	`routes.py`, `tests/blackbox.sh`
Documentation (ERD, tables)	Human-authored, LLM for prose polish	Copy drift vs schema	Manual cross-check against canonical DDL	This documentation
Unit & black-box tests	Llm-human brainstorming, Human-authored, LLM refactoring, some LLM stubs	Incomplete coverage	Coverage reports, green run logs	`pytest-cov` output, `blackbox_test.log`
documentation narrative	Human first draft, LLM for style, peer review	Over-smoothing technical nuance	Supervisor review, manual corrections	This document

Responsible AI Use

Purpose & Scope

Large Language Models (LLMs) such as GPT-4o and o3 were used in this project only as drafting and support tools. They assisted in generating boilerplate code, refactoring, and narrative polish. All authoritative outputs (database schema, routes, tests, and documentation) were verified and finalized by the project team.

Principles

Human-in-the-loop at every stage	• No LLM output was accepted without human review. • Unit and black-box tests served as the final authority for correctness.
Transparency & Provenance	• Each artifact is tagged in a provenance table (see Appendix X) with its origin and verification method. • Failures of LLM outputs (e.g., type drift, trigger errors, naming inconsistencies) are documented in the LLM Deltas section.
Security & Privacy	• No real secrets, credentials, or PII were ever provided to an LLM. • Example data was synthetic; sensitive identifiers were redacted. • The repository will not store secret keys in code; environment variables are used instead.
Reproducibility	• Models used: GPT-5, GPT-4o and o3, between January–September 2025.
Bias & Fairness	• LLM-suggested scoring logic (e.g., opportunity scorecards) was treated as provisional. • Human review ensured no biased or unfair criteria were adopted. • A bias checklist was consulted for any user-facing evaluation logic.
Ethical & Professional Standards	The project aligns with the ACM Code of Ethics and general responsible AI practices: • Avoid harm through inaccurate or unchecked outputs. • Ensure clarity on where AI contributed. • Retain human accountability for all final deliverables.
Disclosure	• A short “AI assistance disclosure” is included in the README and thesis introduction. • It states: “LLMs were used to assist with drafting code and documentation. All outputs were verified through testing and human review. No LLM output was adopted without validation.”

Checklist for Responsible AI in This Project

- LLM outputs logged and reviewed.
- Unit and black-box tests as verification gate.
- No sensitive data sent to LLMs.
- Licensing of generated code checked against permissive licenses.
- Disclosure in thesis and README.
- Provenance & verification table maintained for each artifact.

Summary

LLMs accelerated development, but never replaced human judgment. All critical artifacts were tested, reviewed, and documented. By treating AI as a junior collaborator — helpful but fallible — the project balances productivity gains with integrity, reproducibility, and ethics.

* What, why, where, when, how Detail: * What parts were generated with LLM help. * What failed (e.g., incorrect SQL schema types, weak validation logic). * How output was refined: e.g., fixing schema bugs, renaming fields, adding type checks. Example: > *"The `stage_name` was mistakenly inferred as an INTEGER by the LLM. Manual correction changed it to TEXT."*

Software Requirement Analysis

What	Basic feedback over the main premise of the reasons behind the creation of the assignment deliverables.
why	Double checking over omitions of useful functionality or any sort of errors during the soft. Requirements analysis
Where	Project documents.
When	During the authoring of the document
How	RAG input of project documents
What parts were generated with LLM help.	Document text.
What failed.	Occasional halucinations.
How output was refined.	User text edits.

Preliminary Design

What	Feedback over the way the project should be structured.
why	Double checking over omitions of useful functionality or any sort of errors during this phase.
Where	Project documents.
When	During the authoring of the documents,
How	RAG input of the document.
What parts were generated with LLM help.	Document text.

What failed.	LLM halucinations.
How output was refined.	User text edits.

Detailed Design

What	Feedback over the way the project should be structured. Relational tables, REST endpoints, sql queries, testing with curl, UI design. Code refactoring.
why	Rapid prototyping on both back and front end. Database design. Testing.
Where	The codebase
When	During the development stages.
How	Code input and discussion.
What parts were generated with LLM help.	SQL tables, python code, html code (snipets of code, nothing concrete)
What failed.	Halucinations
How output was refined.	User edits, debugging, pair programming with the llm for debugging during database design

Coding and Unit Testing

What	Overall testing strategies.
why	The creation of a testing approach to the project deliverables.
Where	The codebase
When	After the prototype
How	RAG input of this document
What parts were generated with LLM help.	SQL tables, python code, html code
What failed.	Halucinations
How output was refined.	User edits, debugging, pair programming with the llm for debugging during database design

Integrity statement

All LLM-generated output was treated as a draft. Production code and schema were validated through testing and manual review. Any errors (type mismatches, odd defaults) are retained in the text only as evidence of LLM limitations, hunman omition and as bugs within a software development cycle to be dealt with in their apropriate time when noticed; not as part of the final deliverables that should be held as features.

Integration, and Testing

What	Code generation
why	Boilerplate code
Where	Code
When	Testing stage
How	Prompts
What parts were generated with LLM help.	Code
What failed.	Complex code not working or working and not performing as expected
How output was refined.	Code refactoring and human quality assurance

Literature

[1]: https://en.wikipedia.org/wiki/Customer_relationship_management
[2]: https://en.wikipedia.org/wiki/Vertical_integration
[3]: https://www.zendesk.com/it/blog/guide-to-the-sales-pipeline/#
[4]: https://en.wikipedia.org/wiki/Kanban_board
[5]: https://arxiv.org/abs/2201.11903
[6]: https://en.wikipedia.org/wiki/Waterfall_model
[7]: https://en.wikipedia.org/wiki/Retrieval-augmented_generation
[8]: https://arxiv.org/abs/2508.01781
[9]: https://www.quora.com/If-AI-is-so-good-at-coding-where-are-the-open-source-projects-fully-generated-by-LLMs
[10]: https://en.wikipedia.org/wiki/Rice%E2%80%93Shapiro_theorem
[11]: https://en.wikipedia.org/wiki/Entity_component_system
[12]:https://en.wikipedia.org/wiki/OAuth
[13]: https://en.wikipedia.org/wiki/Security_Assertion_Markup_Language
[14]: https://www.microsoft.com/el-gr/security/business/security-101/what-is-openid-connect-oidc
[15]: https://openai.com/index/why-language-models-hallucinate/

8.2 Extended schema.sql

SQL table creation queries of the service database

```
CREATE TABLE "communications" (
"communication_id" INTEGER,
"vendor_id" INTEGER NOT NULL,
"date_sent" TEXT NOT NULL,
"comm_type" TEXT NOT NULL,
"subject" TEXT NOT NULL,
"content" TEXT NOT NULL,
"user_id" TEXT,
"followup_needed" TEXT,
"created_at" TEXT NOT NULL,
"updated_at" TEXT NOT NULL,
PRIMARY KEY("communication_id" AUTOINCREMENT),
FOREIGN KEY("vendor_id") REFERENCES "vendors"("vendor_id")
)
```

```
CREATE TABLE "contracts" (
	"contract_id"	INTEGER,
	"vendor_id"	INTEGER NOT NULL,
	"contract_name"	TEXT NOT NULL,
	"start_date"	TEXT NOT NULL,
	"end_date"	TEXT,
	"renewal_terms"	TEXT NOT NULL,
	"created_at"	TEXT NOT NULL,
	"updated_at"	TEXT NOT NULL,
	"status"	TEXT NOT NULL,
	PRIMARY KEY("contract_id" AUTOINCREMENT),
	FOREIGN KEY("vendor_id") REFERENCES "vendors"("vendor_id")
)
```

```
CREATE TABLE "documents" (
"document_id" INTEGER,
"vendor_id" INTEGER NOT NULL,
"contract_id" INTEGER NOT NULL,
"file_name" TEXT NOT NULL,
"file_path" TEXT NOT NULL,
"uploaded_at"TEXT NOT NULL,
"updated_at" TEXT NOT NULL,
PRIMARY KEY("document_id" AUTOINCREMENT),
FOREIGN KEY("contract_id") REFERENCES "contracts"("contract_id"),
FOREIGN KEY("vendor_id") REFERENCES "vendors"("vendor_id")
)
```

```
CREATE TABLE "opportunities_task_tickets" (
	"opportunity_id"	INTEGER,
	"pipeline_id"	INTEGER NOT NULL,
	"stage_id"	INTEGER NOT NULL,
	"vendor_id"	INTEGER NOT NULL,
```

```
        "title"  TEXT NOT NULL,
        "description"  TEXT NOT NULL,
        "property_id"  INTEGER NOT NULL,
        "status"       TEXT NOT NULL,
        "priority"     TEXT,
        "due_date"     TEXT NOT NULL,
        "created_at"   TEXT NOT NULL,
        "updated_at"   TEXT NOT NULL,
        PRIMARY KEY("opportunity_id" AUTOINCREMENT),
        FOREIGN KEY("pipeline_id") REFERENCES "pipelines"("pipeline_id"),
        FOREIGN KEY("property_id") REFERENCES "properties"("property_id"),
        FOREIGN KEY("stage_id") REFERENCES "pipeline_stages"("stage_id"),
        FOREIGN KEY("vendor_id") REFERENCES "vendors"("vendor_id")
)
```

```
CREATE TABLE "owner" (
"owner_id" INTEGER,
"first_name" TEXT NOT NULL,
"last_name" TEXT NOT NULL,
"phone" TEXT NOT NULL,
"email" TEXT NOT NULL,
"address" TEXT NOT NULL,
"legal_id" TEXT NOT NULL,
"created_at" TEXT NOT NULL,
"updated_at" TEXT NOT NULL,
PRIMARY KEY("owner_id" AUTOINCREMENT)
)
```

```
CREATE TABLE "owner_property_map" (
"map_id" INTEGER,
"owner_id" INTEGER NOT NULL,
"property_id" INTEGER NOT NULL,
"created_at" TEXT NOT NULL DEFAULT '00.00.0000',
"updated_at" TEXT NOT NULL DEFAULT '00.00.0000',
PRIMARY KEY("map_id" AUTOINCREMENT),
FOREIGN KEY("owner_id") REFERENCES "owner"("owner_id"),
FOREIGN KEY("property_id") REFERENCES "properties"("property_id")
)
```

```
CREATE TABLE "pipeline_stages" (
        "stage_id"      INTEGER,
        "pipeline_id"  INTEGER NOT NULL,
        "stage_name"  TEXT NOT NULL,
        "stage_order"  INTEGER NOT NULL DEFAULT 1,
        "created_at"   TEXT NOT NULL,
        "updated_at"   TEXT NOT NULL,
        PRIMARY KEY("stage_id" AUTOINCREMENT),
        FOREIGN KEY("pipeline_id") REFERENCES "pipelines"("pipeline_id")
)
```

```
CREATE TABLE "pipelines" (
"pipeline_id" INTEGER,
```

```
"pipeline_name" TEXT NOT NULL,
"description" TEXT NOT NULL,
"created_at" TEXT NOT NULL,"updated_at" TEXT NOT NULL,
PRIMARY KEY("pipeline_id" AUTOINCREMENT)
)
```

```
CREATE TABLE "properties" (
"property_id" INTEGER,
"property_name" TEXT NOT NULL,
"address" TEXT NOT NULL,
"status" TEXT NOT NULL,
"created_at" TEXT NOT NULL,
"updated_at" TEXT NOT NULL,
PRIMARY KEY("property_id" AUTOINCREMENT)
)
```

```
CREATE TABLE "vendor_contacts" (
"contact_id" INTEGER,
"vendor_id" INTEGER NOT NULL,
"first_name" TEXT NOT NULL,
"last_name" TEXT NOT NULL,
"job_title" TEXT NOT NULL,
"phone" TEXT,
"email" TEXT,
"created_at" TEXT NOT NULL,
"updated_at" TEXT NOT NULL,
PRIMARY KEY("contact_id" AUTOINCREMENT),
FOREIGN KEY("vendor_id") REFERENCES "vendors"("vendor_id")
)
```

```
CREATE TABLE "vendor_scorecard" (
"scorecard_id" INTEGER,
"vendor_id" INTEGER NOT NULL,
"period_start" TEXT NOT NULL,
"period_end" TEXT NOT NULL,
"quality_score" NUMERIC DEFAULT 0,
"on_time_delivery" NUMERIC DEFAULT 0,
"defect_rate" NUMERIC DEFAULT 0,
"comments" TEXT,
"created_at" TEXT NOT NULL,
"updated_at" TEXT NOT NULL,
PRIMARY KEY("scorecard_id" AUTOINCREMENT),
FOREIGN KEY("vendor_id") REFERENCES "vendors"("vendor_id")
)
```

```
CREATE TABLE "vendors" (
"vendor_id" INTEGER,
"name" TEXT NOT NULL,
"vendor_type"TEXT NOT NULL,
"status" TEXT NOT NULL,
"phone" TEXT NOT NULL,
"email" TEXT NOT NULL,
```

```
"address" TEXT NOT NULL,
"created_at" TEXT NOT NULL,"updated_at" TEXT NOT NULL,
PRIMARY KEY("vendor_id" AUTOINCREMENT)
)
```

```
CREATE TABLE "users" (
        "userId"        INTEGER,
        "userName"      TEXT UNIQUE,
        "userRole"      TEXT NOT NULL,
        PRIMARY KEY("userId" AUTOINCREMENT)
)
```

SQL trigger creation queries

```
CREATE TRIGGER fk_communications_vendor_ins
BEFORE INSERT ON communications
FOR EACH ROW
WHEN NOT EXISTS (SELECT 1 FROM vendors v WHERE v.vendor_id = NEW.vendor_id)
BEGIN
  SELECT RAISE(ABORT,'Foreign-key violation: communications.vendor_id not found in
vendors');
END
```

```
CREATE TRIGGER fk_communications_vendor_upd
BEFORE UPDATE OF vendor_id ON communications
FOR EACH ROW
WHEN NOT EXISTS (SELECT 1 FROM vendors v WHERE v.vendor_id = NEW.vendor_id)
BEGIN
  SELECT RAISE(ABORT,'Foreign-key violation: communications.vendor_id not found in
vendors');
END
```

```
CREATE TRIGGER fk_contracts_has_docs_bd
BEFORE DELETE ON contracts
FOR EACH ROW
WHEN EXISTS (SELECT 1 FROM documents d WHERE d.contract_id = OLD.contract_id)
BEGIN
  SELECT RAISE(ABORT,'Delete violates FK: contract still referenced by documents');
END
```

```
CREATE TRIGGER fk_contracts_vendor_ins
BEFORE INSERT ON contracts
FOR EACH ROW
WHEN NOT EXISTS (SELECT 1 FROM vendors v WHERE v.vendor_id = NEW.vendor_id)
BEGIN
  SELECT RAISE(ABORT,'Foreign-key violation: contracts.vendor_id not found in vendors');
END
```

```
CREATE TRIGGER fk_contracts_vendor_upd
BEFORE UPDATE OF vendor_id ON contracts
FOR EACH ROW
WHEN NOT EXISTS (SELECT 1 FROM vendors v WHERE v.vendor_id = NEW.vendor_id)
```

```
BEGIN
  SELECT RAISE(ABORT,'Foreign-key violation: contracts.vendor_id not found in vendors');
END
```

```
CREATE TRIGGER fk_document_vendor_matches_contract_ins
BEFORE INSERT ON documents
FOR EACH ROW
WHEN (SELECT vendor_id FROM contracts WHERE contract_id = NEW.contract_id) <>
NEW.vendor_id
BEGIN
  SELECT RAISE(ABORT, 'document vendor mismatch');
END
```

```
CREATE TRIGGER fk_document_vendor_matches_contract_upd
BEFORE UPDATE OF vendor_id, contract_id ON documents
FOR EACH ROW
WHEN (SELECT vendor_id FROM contracts WHERE contract_id = NEW.contract_id) <>
NEW.vendor_id
BEGIN
  SELECT RAISE(ABORT, 'document vendor mismatch');
END
```

```
CREATE TRIGGER fk_documents_contract_ins
BEFORE INSERT ON documents
FOR EACH ROW
WHEN NOT EXISTS (SELECT 1 FROM contracts c WHERE c.contract_id = NEW.contract_id)
BEGIN
  SELECT RAISE(ABORT,'Foreign-key violation: documents.contract_id not found in contracts');
END
```

```
CREATE TRIGGER fk_documents_contract_upd
BEFORE UPDATE OF contract_id ON documents
FOR EACH ROW
WHEN NOT EXISTS (SELECT 1 FROM contracts c WHERE c.contract_id = NEW.contract_id)
BEGIN
  SELECT RAISE(ABORT,'Foreign-key violation: documents.contract_id not found in contracts');
END
```

```
CREATE TRIGGER fk_documents_vendor_ins
BEFORE INSERT ON documents
FOR EACH ROW
WHEN NOT EXISTS (SELECT 1 FROM vendors v WHERE v.vendor_id = NEW.vendor_id)
BEGIN
  SELECT RAISE(ABORT,'Foreign-key violation: documents.vendor_id not found in vendors');
END
```

```
CREATE TRIGGER fk_documents_vendor_upd
BEFORE UPDATE OF vendor_id ON documents
FOR EACH ROW
WHEN NOT EXISTS (SELECT 1 FROM vendors v WHERE v.vendor_id = NEW.vendor_id)
BEGIN
  SELECT RAISE(ABORT,'Foreign-key violation: documents.vendor_id not found in vendors');
END
```

```
CREATE TRIGGER fk_opp_stage_matches_pipeline_ins
BEFORE INSERT ON opportunities_task_tickets
FOR EACH ROW
WHEN (SELECT pipeline_id FROM pipeline_stages WHERE stage_id = NEW.stage_id) <>
NEW.pipeline_id
BEGIN
  SELECT RAISE(ABORT, 'stage not in pipeline');
END
```

```
CREATE TRIGGER fk_opp_stage_matches_pipeline_upd
BEFORE UPDATE OF stage_id, pipeline_id ON opportunities_task_tickets
FOR EACH ROW
WHEN (SELECT pipeline_id FROM pipeline_stages WHERE stage_id = NEW.stage_id) <>
NEW.pipeline_id
BEGIN
  SELECT RAISE(ABORT, 'stage not in pipeline');
END
```

```
CREATE TRIGGER fk_opp_ticket_pipeline_ins
BEFORE INSERT ON opportunities_task_tickets
FOR EACH ROW
WHEN NOT EXISTS (SELECT 1 FROM pipelines p WHERE p.pipeline_id = NEW.pipeline_id)
BEGIN
  SELECT RAISE(ABORT,'Foreign-key violation: opportunity.pipeline_id not found');
END
```

```
CREATE TRIGGER fk_opp_ticket_pipeline_upd
BEFORE UPDATE OF pipeline_id ON opportunities_task_tickets
FOR EACH ROW
WHEN NOT EXISTS (SELECT 1 FROM pipelines p WHERE p.pipeline_id = NEW.pipeline_id)
BEGIN
  SELECT RAISE(ABORT,'Foreign-key violation: opportunity.pipeline_id not found');
END
```

```
CREATE TRIGGER fk_opp_ticket_property_ins
BEFORE INSERT ON opportunities_task_tickets
FOR EACH ROW
WHEN NEW.property_id IS NOT NULL
 AND NOT EXISTS (SELECT 1 FROM properties p WHERE p.property_id = NEW.property_id)
BEGIN
  SELECT RAISE(ABORT,'Foreign-key violation: opportunity.property_id not found');
END
```

```
CREATE TRIGGER fk_opp_ticket_property_upd
BEFORE UPDATE OF property_id ON opportunities_task_tickets
FOR EACH ROW
WHEN NEW.property_id IS NOT NULL
 AND NOT EXISTS (SELECT 1 FROM properties p WHERE p.property_id = NEW.property_id)
BEGIN
  SELECT RAISE(ABORT,'Foreign-key violation: opportunity.property_id not found');
```

```
END
```

```
CREATE TRIGGER fk_opp_ticket_stage_ins
BEFORE INSERT ON opportunities_task_tickets
FOR EACH ROW
WHEN NOT EXISTS (SELECT 1 FROM pipeline_stages s WHERE s.stage_id = NEW.stage_id)
BEGIN
  SELECT RAISE(ABORT,'Foreign-key violation: opportunity.stage_id not found');
END
```

```
CREATE TRIGGER fk_opp_ticket_stage_upd
BEFORE UPDATE OF stage_id ON opportunities_task_tickets
FOR EACH ROW
WHEN NOT EXISTS (SELECT 1 FROM pipeline_stages s WHERE s.stage_id = NEW.stage_id)
BEGIN
  SELECT RAISE(ABORT,'Foreign-key violation: opportunity.stage_id not found');
END
```

```
CREATE TRIGGER fk_opp_ticket_vendor_ins
BEFORE INSERT ON opportunities_task_tickets
FOR EACH ROW
WHEN NOT EXISTS (SELECT 1 FROM vendors v WHERE v.vendor_id = NEW.vendor_id)
BEGIN
  SELECT RAISE(ABORT,'Foreign-key violation: opportunities_task_tickets.vendor_id not
found');
END
```

```
CREATE TRIGGER fk_opp_ticket_vendor_upd
BEFORE UPDATE OF vendor_id ON opportunities_task_tickets
FOR EACH ROW
WHEN NOT EXISTS (SELECT 1 FROM vendors v WHERE v.vendor_id = NEW.vendor_id)
BEGIN
  SELECT RAISE(ABORT,'Foreign-key violation: opportunities_task_tickets.vendor_id not
found');
END
```

```
CREATE TRIGGER fk_owner_has_map_bd
BEFORE DELETE ON owner
FOR EACH ROW
WHEN EXISTS (SELECT 1 FROM owner_property_map m WHERE m.owner_id =
OLD.owner_id)
BEGIN
  SELECT RAISE(ABORT,'Delete violates FK: owner still referenced in owner_property_map');
END
```

```
CREATE TRIGGER fk_owner_map_owner_ins
BEFORE INSERT ON owner_property_map
FOR EACH ROW
WHEN NOT EXISTS (SELECT 1 FROM owner ow WHERE ow.owner_id = NEW.owner_id)
BEGIN
  SELECT RAISE(ABORT,'Foreign-key violation: owner_property_map.owner_id not found');
END
```

```
CREATE TRIGGER fk_owner_map_owner_upd
BEFORE UPDATE OF owner_id ON owner_property_map
FOR EACH ROW
WHEN NOT EXISTS (SELECT 1 FROM owner ow WHERE ow.owner_id = NEW.owner_id)
BEGIN
  SELECT RAISE(ABORT,'Foreign-key violation: owner_property_map.owner_id not found');
END
```

```
CREATE TRIGGER fk_owner_map_property_ins
BEFORE INSERT ON owner_property_map
FOR EACH ROW
WHEN NOT EXISTS (SELECT 1 FROM properties p WHERE p.property_id =
NEW.property_id)
BEGIN
  SELECT RAISE(ABORT,'Foreign-key violation: owner_property_map.property_id not found');
END
```

```
CREATE TRIGGER fk_owner_map_property_upd
BEFORE UPDATE OF property_id ON owner_property_map
FOR EACH ROW
WHEN NOT EXISTS (SELECT 1 FROM properties p WHERE p.property_id =
NEW.property_id)
BEGIN
  SELECT RAISE(ABORT,'Foreign-key violation: owner_property_map.property_id not found');
END
```

```
CREATE TRIGGER fk_pipeline_stages_has_opp_bd
BEFORE DELETE ON pipeline_stages
FOR EACH ROW
WHEN EXISTS (SELECT 1 FROM opportunities_task_tickets o WHERE o.stage_id =
OLD.stage_id)
BEGIN
  SELECT RAISE(ABORT,'Delete violates FK: stage still referenced by opportunities/tasks');
END
```

```
CREATE TRIGGER fk_pipeline_stages_pipeline_ins
BEFORE INSERT ON pipeline_stages
FOR EACH ROW
WHEN NOT EXISTS (SELECT 1 FROM pipelines p WHERE p.pipeline_id = NEW.pipeline_id)
BEGIN
  SELECT RAISE(ABORT,'Foreign-key violation: stage.pipeline_id not found in pipelines');
END
```

```
CREATE TRIGGER fk_pipeline_stages_pipeline_upd
BEFORE UPDATE OF pipeline_id ON pipeline_stages
FOR EACH ROW
WHEN NOT EXISTS (SELECT 1 FROM pipelines p WHERE p.pipeline_id = NEW.pipeline_id)
BEGIN
  SELECT RAISE(ABORT,'Foreign-key violation: stage.pipeline_id not found in pipelines');
END
```

```
CREATE TRIGGER fk_pipelines_has_children_bd
BEFORE DELETE ON pipelines
```

```
FOR EACH ROW
WHEN EXISTS (SELECT 1 FROM pipeline_stages s WHERE s.pipeline_id = OLD.pipeline_id)
  OR EXISTS (SELECT 1 FROM opportunities_task_tickets o WHERE o.pipeline_id =
OLD.pipeline_id)
BEGIN
  SELECT RAISE(ABORT,'Delete violates FK: pipeline still referenced');
END
```

```
CREATE TRIGGER fk_properties_has_children_bd
BEFORE DELETE ON properties
FOR EACH ROW
WHEN EXISTS (SELECT 1 FROM owner_property_map m WHERE m.property_id =
OLD.property_id)
  OR EXISTS (SELECT 1 FROM opportunities_task_tickets o WHERE o.property_id =
OLD.property_id)
BEGIN
  SELECT RAISE(ABORT,'Delete violates FK: property still referenced');
END
```

```
CREATE TRIGGER fk_vendor_contacts_vendor_ins
BEFORE INSERT ON vendor_contacts
FOR EACH ROW
WHEN NOT EXISTS (SELECT 1 FROM vendors v WHERE v.vendor_id = NEW.vendor_id)
BEGIN
  SELECT RAISE(ABORT,'Foreign-key violation: vendor_contacts.vendor_id not found in
vendors');
END
```

```
CREATE TRIGGER fk_vendor_contacts_vendor_upd
BEFORE UPDATE OF vendor_id ON vendor_contacts
FOR EACH ROW
WHEN NOT EXISTS (SELECT 1 FROM vendors v WHERE v.vendor_id = NEW.vendor_id)
BEGIN
  SELECT RAISE(ABORT,'Foreign-key violation: vendor_contacts.vendor_id not found in
vendors');
END
```

```
CREATE TRIGGER fk_vendor_scorecard_vendor_ins
BEFORE INSERT ON vendor_scorecard
FOR EACH ROW
WHEN NOT EXISTS (SELECT 1 FROM vendors v WHERE v.vendor_id = NEW.vendor_id)
BEGIN
  SELECT RAISE(ABORT,'Foreign-key violation: vendor_scorecard.vendor_id not found in
vendors');
END
```

```
CREATE TRIGGER fk_vendor_scorecard_vendor_upd
BEFORE UPDATE OF vendor_id ON vendor_scorecard
FOR EACH ROW
WHEN NOT EXISTS (SELECT 1 FROM vendors v WHERE v.vendor_id = NEW.vendor_id)
BEGIN
  SELECT RAISE(ABORT,'Foreign-key violation: vendor_scorecard.vendor_id not found in
```

```
vendors');
END
```

```
CREATE TRIGGER fk_vendors_has_children_bd
BEFORE DELETE ON vendors
FOR EACH ROW
WHEN EXISTS (SELECT 1 FROM communications          c WHERE c.vendor_id =
OLD.vendor_id)
   OR EXISTS (SELECT 1 FROM contracts             c WHERE c.vendor_id = OLD.vendor_id)
   OR EXISTS (SELECT 1 FROM documents             d WHERE d.vendor_id = OLD.vendor_id)
   OR EXISTS (SELECT 1 FROM opportunities_task_tickets o WHERE o.vendor_id =
OLD.vendor_id)
   OR EXISTS (SELECT 1 FROM vendor_contacts        vc WHERE vc.vendor_id =
OLD.vendor_id)
   OR EXISTS (SELECT 1 FROM vendor_scorecard       vs WHERE vs.vendor_id =
OLD.vendor_id)
BEGIN
  SELECT RAISE(ABORT,'Delete violates FK: vendors row still referenced');
END
```

8.3 Full OpenAPI YAML

```
openapi: 3.1.0
info:
  title: CRM Backend API
  version: "1.0.0"
  description: |
    Cookie-authenticated REST API for the CRM MVP.
    - **Auth**: HttpOnly cookies (access_token, refresh_token); CSRF header required on write.
    - **Date format**: DD.MM.YYYY.
    - **Range reads**: GET /{entity}/{start}/{end} return inclusive ID slices.
servers:
  - url: http://localhost:8000
components:
  securitySchemes:
    cookieAuth:
      type: apiKey
      in: cookie
      name: access_token
    csrfHeader:
      type: apiKey
      in: header
      name: X-CSRF-Token
  schemas:
    ErrorEnvelope:
      type: object
      properties:
        error: {"type":"string"}
```

```
        fields:
          type: object
          additionalProperties: true
      required: [error]
    DateDDMMYYYY:
      type: string
      pattern: "^\\d{2}\.\d{2}\.\d{4}$"
    Vendor:
      type: object
      properties:
        vendor_id: {type: integer}
        name: {type: string}
        vendor_type: {type: string}
        status: {type: string}
        phone: {type: string}
        email: {type: string, format: email}
        address: {type: string}
        created_at: {$ref: '#/components/schemas/DateDDMMYYYY'}
        updated_at: {$ref: '#/components/schemas/DateDDMMYYYY'}
      required: [name, vendor_type, status, phone, email, address, created_at, updated_at]
    VendorContact:
      type: object
      properties:
        contact_id: {type: integer}
        vendor_id: {type: integer}
        first_name: {type: string}
        last_name: {type: string}
        job_title: {type: string}
        phone: {type: string}
        email: {type: string, format: email}
        created_at: {$ref: '#/components/schemas/DateDDMMYYYY'}
        updated_at: {$ref: '#/components/schemas/DateDDMMYYYY'}
      required: [vendor_id, first_name, last_name, job_title, created_at, updated_at]
    VendorScorecard:
      type: object
      properties:
        scorecard_id: {type: integer}
        vendor_id: {type: integer}
        period_start: {$ref: '#/components/schemas/DateDDMMYYYY'}
        period_end: {$ref: '#/components/schemas/DateDDMMYYYY'}
        quality_score: {type: number}
        on_time_delivery: {type: number}
        defect_rate: {type: number}
        comments: {type: string}
        created_at: {$ref: '#/components/schemas/DateDDMMYYYY'}
        updated_at: {$ref: '#/components/schemas/DateDDMMYYYY'}
      required: [vendor_id, period_start, period_end, quality_score, on_time_delivery, defect_rate,
created_at, updated_at]
    Contract:
      type: object
```

```
    properties:
      contract_id: {type: integer}
      vendor_id: {type: integer}
      contract_name: {type: string}
      start_date: {$ref: '#/components/schemas/DateDDMMYYYY'}
      end_date: {$ref: '#/components/schemas/DateDDMMYYYY'}
      renewal_terms: {type: string}
      status: {type: string}
      created_at: {$ref: '#/components/schemas/DateDDMMYYYY'}
      updated_at: {$ref: '#/components/schemas/DateDDMMYYYY'}
    required: [vendor_id, contract_name, start_date, end_date, created_at, updated_at, status]
  Document:
    type: object
    properties:
      document_id: {type: integer}
      vendor_id: {type: integer}
      contract_id: {type: integer}
      file_name: {type: string}
      file_path: {type: string}
      uploaded_at: {$ref: '#/components/schemas/DateDDMMYYYY'}
      updated_at: {$ref: '#/components/schemas/DateDDMMYYYY'}
    required: [vendor_id, contract_id, file_name, uploaded_at, updated_at]
  Communication:
    type: object
    properties:
      communication_id: {type: integer}
      vendor_id: {type: integer}
      date_sent: {$ref: '#/components/schemas/DateDDMMYYYY'}
      comm_type: {type: string}
      subject: {type: string}
      content: {type: string}
      user_id: {type: string}
      followup_needed: {type: string, enum: ['0','1']}
      created_at: {$ref: '#/components/schemas/DateDDMMYYYY'}
      updated_at: {$ref: '#/components/schemas/DateDDMMYYYY'}
    required: [vendor_id, date_sent, comm_type, subject, content, followup_needed, created_at,
updated_at]
  Pipeline:
    type: object
    properties:
      pipeline_id: {type: integer}
      pipeline_name: {type: string}
      description: {type: string}
      created_at: {$ref: '#/components/schemas/DateDDMMYYYY'}
      updated_at: {$ref: '#/components/schemas/DateDDMMYYYY'}
    required: [pipeline_name, description, created_at, updated_at]
  PipelineStage:
    type: object
    properties:
      stage_id: {type: integer}
```

```
        pipeline_id: {type: integer}
        stage_name: {type: string}
        stage_order: {type: integer}
        created_at: {$ref: '#/components/schemas/DateDDMMYYYY'}
        updated_at: {$ref: '#/components/schemas/DateDDMMYYYY'}
      required: [pipeline_id, stage_name, stage_order, created_at, updated_at]
    OpportunityTaskTicket:
      type: object
      properties:
        opportunity_id: {type: integer}
        pipeline_id: {type: integer}
        stage_id: {type: integer}
        vendor_id: {type: integer}
        property_id: {type: integer}
        title: {type: string}
        description: {type: string}
        status: {type: string}
        priority: {type: string}
        due_date: {$ref: '#/components/schemas/DateDDMMYYYY'}
        created_at: {$ref: '#/components/schemas/DateDDMMYYYY'}
        updated_at: {$ref: '#/components/schemas/DateDDMMYYYY'}
      required: [pipeline_id, stage_id, vendor_id, property_id, title, description, status, created_at,
updated_at]
    Property:
      type: object
      properties:
        property_id: {type: integer}
        property_name: {type: string}
        address: {type: string}
        status: {type: string}
        created_at: {$ref: '#/components/schemas/DateDDMMYYYY'}
        updated_at: {$ref: '#/components/schemas/DateDDMMYYYY'}
      required: [property_name, address, status, created_at, updated_at]
    Owner:
      type: object
      properties:
        owner_id: {type: integer}
        first_name: {type: string}
        last_name: {type: string}
        phone: {type: string}
        email: {type: string, format: email}
        address: {type: string}
        legal_id: {type: string}
        created_at: {$ref: '#/components/schemas/DateDDMMYYYY'}
        updated_at: {$ref: '#/components/schemas/DateDDMMYYYY'}
      required: [first_name, last_name, phone, email, address, legal_id, created_at, updated_at]
    OwnerPropertyMap:
      type: object
      properties:
        map_id: {type: integer}
```

```
        owner_id: {type: integer}
        property_id: {type: integer}
        created_at: {$ref: '#/components/schemas/DateDDMMYYYY'}
        updated_at: {$ref: '#/components/schemas/DateDDMMYYYY'}
      required: [owner_id, property_id, created_at, updated_at]

paths:
  /login:
    post:
      summary: Login and receive cookies
      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
              properties:
                username: {type: string}
                password: {type: string}
              required: [username, password]
      responses:
        '200':
          description: Logged in
        '401':
          description: Invalid credentials
          content:
            application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}

  /logout:
    post:
      summary: Logout and clear cookies
      security:
        - cookieAuth: []
          # CSRF not required for logout in this MVP (adjust if you enforce it)
      responses:
        '204': {description: Logged out}

  /refresh:
    post:
      summary: Refresh access token using refresh_token cookie
      security:
        - cookieAuth: []
      responses:
        '200': {description: Refreshed}
        '401': {description: Missing/expired refresh}

openapi: 3.1.0
info:
  title: CRM Backend API
  version: "1.0.0"
```

```
  description: |
    Cookie-authenticated REST API for the CRM MVP.
    - **Auth**: HttpOnly cookies (access_token, refresh_token); CSRF header required on write.
    - **Date format**: DD.MM.YYYY.
    - **Range reads**: GET /{entity}/{start}/{end} return inclusive ID slices.
servers:
  - url: http://localhost:8000
components:
  securitySchemes:
    cookieAuth:
      type: apiKey
      in: cookie
      name: access_token
    csrfHeader:
      type: apiKey
      in: header
      name: X-CSRF-Token
  schemas:
    ErrorEnvelope:
      type: object
      properties:
        error: {"type":"string"}
        fields:
          type: object
          additionalProperties: true
      required: [error]
    DateDDMMYYYY:
      type: string
      pattern: "^\\d{2}\\.\\d{2}\\.\\d{4}$"
    Vendor:
      type: object
      properties:
        vendor_id: {type: integer}
        name: {type: string}
        vendor_type: {type: string}
        status: {type: string}
        phone: {type: string}
        email: {type: string, format: email}
        address: {type: string}
        created_at: {$ref: '#/components/schemas/DateDDMMYYYY'}
        updated_at: {$ref: '#/components/schemas/DateDDMMYYYY'}
      required: [name, vendor_type, status, phone, email, address, created_at, updated_at]
    VendorContact:
      type: object
      properties:
        contact_id: {type: integer}
        vendor_id: {type: integer}
        first_name: {type: string}
        last_name: {type: string}
        job_title: {type: string}
```

```
        phone: {type: string}
        email: {type: string, format: email}
        created_at: {$ref: '#/components/schemas/DateDDMMYYYY'}
        updated_at: {$ref: '#/components/schemas/DateDDMMYYYY'}
      required: [vendor_id, first_name, last_name, job_title, created_at, updated_at]
    VendorScorecard:
      type: object
      properties:
        scorecard_id: {type: integer}
        vendor_id: {type: integer}
        period_start: {$ref: '#/components/schemas/DateDDMMYYYY'}
        period_end: {$ref: '#/components/schemas/DateDDMMYYYY'}
        quality_score: {type: number}
        on_time_delivery: {type: number}
        defect_rate: {type: number}
        comments: {type: string}
        created_at: {$ref: '#/components/schemas/DateDDMMYYYY'}
        updated_at: {$ref: '#/components/schemas/DateDDMMYYYY'}
      required: [vendor_id, period_start, period_end, quality_score, on_time_delivery, defect_rate,
created_at, updated_at]
    Contract:
      type: object
      properties:
        contract_id: {type: integer}
        vendor_id: {type: integer}
        contract_name: {type: string}
        start_date: {$ref: '#/components/schemas/DateDDMMYYYY'}
        end_date: {$ref: '#/components/schemas/DateDDMMYYYY'}
        renewal_terms: {type: string}
        status: {type: string}
        created_at: {$ref: '#/components/schemas/DateDDMMYYYY'}
        updated_at: {$ref: '#/components/schemas/DateDDMMYYYY'}
      required: [vendor_id, contract_name, start_date, end_date, created_at, updated_at, status]
    Document:
      type: object
      properties:
        document_id: {type: integer}
        vendor_id: {type: integer}
        contract_id: {type: integer}
        file_name: {type: string}
        file_path: {type: string}
        uploaded_at: {$ref: '#/components/schemas/DateDDMMYYYY'}
        updated_at: {$ref: '#/components/schemas/DateDDMMYYYY'}
      required: [vendor_id, contract_id, file_name, uploaded_at, updated_at]
    Communication:
      type: object
      properties:
        communication_id: {type: integer}
        vendor_id: {type: integer}
        date_sent: {$ref: '#/components/schemas/DateDDMMYYYY'}
```

```
        comm_type: {type: string}
        subject: {type: string}
        content: {type: string}
        user_id: {type: string}
        followup_needed: {type: string, enum: ['0','1']}
        created_at: {$ref: '#/components/schemas/DateDDMMYYYY'}
        updated_at: {$ref: '#/components/schemas/DateDDMMYYYY'}
      required: [vendor_id, date_sent, comm_type, subject, content, followup_needed, created_at,
updated_at]
    Pipeline:
      type: object
      properties:
        pipeline_id: {type: integer}
        pipeline_name: {type: string}
        description: {type: string}
        created_at: {$ref: '#/components/schemas/DateDDMMYYYY'}
        updated_at: {$ref: '#/components/schemas/DateDDMMYYYY'}
      required: [pipeline_name, description, created_at, updated_at]
    PipelineStage:
      type: object
      properties:
        stage_id: {type: integer}
        pipeline_id: {type: integer}
        stage_name: {type: string}
        stage_order: {type: integer}
        created_at: {$ref: '#/components/schemas/DateDDMMYYYY'}
        updated_at: {$ref: '#/components/schemas/DateDDMMYYYY'}
      required: [pipeline_id, stage_name, stage_order, created_at, updated_at]
    OpportunityTaskTicket:
      type: object
      properties:
        opportunity_id: {type: integer}
        pipeline_id: {type: integer}
        stage_id: {type: integer}
        vendor_id: {type: integer}
        property_id: {type: integer}
        title: {type: string}
        description: {type: string}
        status: {type: string}
        priority: {type: string}
        due_date: {$ref: '#/components/schemas/DateDDMMYYYY'}
        created_at: {$ref: '#/components/schemas/DateDDMMYYYY'}
        updated_at: {$ref: '#/components/schemas/DateDDMMYYYY'}
      required: [pipeline_id, stage_id, vendor_id, property_id, title, description, status, created_at,
updated_at]
    Property:
      type: object
      properties:
        property_id: {type: integer}
        property_name: {type: string}
```

```
      address: {type: string}
      status: {type: string}
      created_at: {$ref: '#/components/schemas/DateDDMMYYYY'}
      updated_at: {$ref: '#/components/schemas/DateDDMMYYYY'}
    required: [property_name, address, status, created_at, updated_at]
  Owner:
    type: object
    properties:
      owner_id: {type: integer}
      first_name: {type: string}
      last_name: {type: string}
      phone: {type: string}
      email: {type: string, format: email}
      address: {type: string}
      legal_id: {type: string}
      created_at: {$ref: '#/components/schemas/DateDDMMYYYY'}
      updated_at: {$ref: '#/components/schemas/DateDDMMYYYY'}
    required: [first_name, last_name, phone, email, address, legal_id, created_at, updated_at]
  OwnerPropertyMap:
    type: object
    properties:
      map_id: {type: integer}
      owner_id: {type: integer}
      property_id: {type: integer}
      created_at: {$ref: '#/components/schemas/DateDDMMYYYY'}
      updated_at: {$ref: '#/components/schemas/DateDDMMYYYY'}
    required: [owner_id, property_id, created_at, updated_at]

paths:
  /login:
    post:
      summary: Login and receive cookies
      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
              properties:
                username: {type: string}
                password: {type: string}
              required: [username, password]
      responses:
        '200':
          description: Logged in
        '401':
          description: Invalid credentials
          content:
            application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}
```

```
  /logout:
    post:
      summary: Logout and clear cookies
      security:
        - cookieAuth: []
          # CSRF not required for logout in this MVP (adjust if you enforce it)
      responses:
        '204': {description: Logged out}

  /refresh:
    post:
      summary: Refresh access token using refresh_token cookie
      security:
        - cookieAuth: []
      responses:
        '200': {description: Refreshed}
        '401': {description: Missing/expired refresh}


  /vendors/:
    post:
      summary: Create Vendor
      security:
        - cookieAuth: []
          csrfHeader: []
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Vendor'
      responses:
        '201':
          description: Created
          content:
            application/json:
              schema: $ref: '#/components/schemas/Vendor'
        '400':
          description: Validation error
          content:
            application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}
        '401': {description: Unauthorized}
        '403': {description: Forbidden}
  /vendors/{start}/{end}/:
    get:
      summary: List Vendor by inclusive ID range
      security:
        - cookieAuth: []
      parameters:
        - in: path
```

```
        name: start
        required: true
        schema: {type: integer, minimum: 0}
      - in: path
        name: end
        required: true
        schema: {type: integer, minimum: 0}
    responses:
      '200':
        description: Range result
        content:
          application/json:
            schema:
              type: array
              items: {$ref: '#/components/schemas/Vendor'}
      '400':
        description: Bad range
        content:
          application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}
      '401': {description: Unauthorized}
  /vendors/{vendor_id}:
    put:
      summary: Update Vendor by ID
      security:
        - cookieAuth: []
          csrfHeader: []
      parameters:
        - in: path
          name: vendor_id
          required: true
          schema: {type: integer}
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Vendor'
      responses:
        '200':
          description: Updated
          content:
            application/json:
              schema: $ref: '#/components/schemas/Vendor'
        '400':
          description: Validation error
          content:
            application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}
        '401': {description: Unauthorized}
        '403': {description: Forbidden}
        '404': {description: Not found}
```

```
    delete:
      summary: Delete Vendor by ID
      security:
        - cookieAuth: []
          csrfHeader: []
      parameters:
        - in: path
          name: vendor_id
          required: true
          schema: {type: integer}
      responses:
        '204': {description: Deleted}
        '401': {description: Unauthorized}
        '403': {description: Forbidden}
        '404': {description: Not found}
        '409':
          description: FK/uniqueness constraint violation
          content:
            application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}

  /vendor_contacts/:
    post:
      summary: Create VendorContact
      security:
        - cookieAuth: []
          csrfHeader: []
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/VendorContact'
      responses:
        '201':
          description: Created
          content:
            application/json:
              schema: $ref: '#/components/schemas/VendorContact'
        '400':
          description: Validation error
          content:
            application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}
        '401': {description: Unauthorized}
        '403': {description: Forbidden}
  /vendor_contacts/{start}/{end}/:
    get:
      summary: List VendorContact by inclusive ID range
      security:
        - cookieAuth: []
      parameters:
```

```
      - in: path
        name: start
        required: true
        schema: {type: integer, minimum: 0}
      - in: path
        name: end
        required: true
        schema: {type: integer, minimum: 0}
    responses:
      '200':
        description: Range result
        content:
          application/json:
            schema:
              type: array
              items: {$ref: '#/components/schemas/VendorContact'}
      '400':
        description: Bad range
        content:
          application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}
      '401': {description: Unauthorized}
  /vendor_contacts/{contact_id}:
    put:
      summary: Update VendorContact by ID
      security:
        - cookieAuth: []
          csrfHeader: []
      parameters:
        - in: path
          name: contact_id
          required: true
          schema: {type: integer}
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/VendorContact'
      responses:
        '200':
          description: Updated
          content:
            application/json:
              schema: $ref: '#/components/schemas/VendorContact'
        '400':
          description: Validation error
          content:
            application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}
        '401': {description: Unauthorized}
        '403': {description: Forbidden}
```

```
      '404': {description: Not found}
    delete:
      summary: Delete VendorContact by ID
      security:
        - cookieAuth: []
          csrfHeader: []
      parameters:
        - in: path
          name: contact_id
          required: true
          schema: {type: integer}
      responses:
        '204': {description: Deleted}
        '401': {description: Unauthorized}
        '403': {description: Forbidden}
        '404': {description: Not found}
        '409':
          description: FK/uniqueness constraint violation
          content:
            application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}

  /vendor_scorecard/:
    post:
      summary: Create VendorScorecard
      security:
        - cookieAuth: []
          csrfHeader: []
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/VendorScorecard'
      responses:
        '201':
          description: Created
          content:
            application/json:
              schema: $ref: '#/components/schemas/VendorScorecard'
        '400':
          description: Validation error
          content:
            application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}
        '401': {description: Unauthorized}
        '403': {description: Forbidden}
  /vendor_scorecard/{start}/{end}/:
    get:
      summary: List VendorScorecard by inclusive ID range
      security:
        - cookieAuth: []
```

```
    parameters:
      - in: path
        name: start
        required: true
        schema: {type: integer, minimum: 0}
      - in: path
        name: end
        required: true
        schema: {type: integer, minimum: 0}
    responses:
      '200':
        description: Range result
        content:
          application/json:
            schema:
              type: array
              items: {$ref: '#/components/schemas/VendorScorecard'}
      '400':
        description: Bad range
        content:
          application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}
      '401': {description: Unauthorized}
  /vendor_scorecard/{scorecard_id}:
    put:
      summary: Update VendorScorecard by ID
      security:
        - cookieAuth: []
          csrfHeader: []
      parameters:
        - in: path
          name: scorecard_id
          required: true
          schema: {type: integer}
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/VendorScorecard'
      responses:
        '200':
          description: Updated
          content:
            application/json:
              schema: $ref: '#/components/schemas/VendorScorecard'
        '400':
          description: Validation error
          content:
            application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}
        '401': {description: Unauthorized}
```

```
      '403': {description: Forbidden}
      '404': {description: Not found}
  delete:
    summary: Delete VendorScorecard by ID
    security:
      - cookieAuth: []
        csrfHeader: []
    parameters:
      - in: path
        name: scorecard_id
        required: true
        schema: {type: integer}
    responses:
      '204': {description: Deleted}
      '401': {description: Unauthorized}
      '403': {description: Forbidden}
      '404': {description: Not found}
      '409':
        description: FK/uniqueness constraint violation
        content:
          application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}

/contracts/:
  post:
    summary: Create Contract
    security:
      - cookieAuth: []
        csrfHeader: []
    requestBody:
      required: true
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Contract'
    responses:
      '201':
        description: Created
        content:
          application/json:
            schema: $ref: '#/components/schemas/Contract'
      '400':
        description: Validation error
        content:
          application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}
      '401': {description: Unauthorized}
      '403': {description: Forbidden}
/contracts/{start}/{end}/:
  get:
    summary: List Contract by inclusive ID range
    security:
```

```
      - cookieAuth: []
    parameters:
      - in: path
        name: start
        required: true
        schema: {type: integer, minimum: 0}
      - in: path
        name: end
        required: true
        schema: {type: integer, minimum: 0}
    responses:
      '200':
        description: Range result
        content:
          application/json:
            schema:
              type: array
              items: {$ref: '#/components/schemas/Contract'}
      '400':
        description: Bad range
        content:
          application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}
      '401': {description: Unauthorized}
  /contracts/{contract_id}:
    put:
      summary: Update Contract by ID
      security:
        - cookieAuth: []
          csrfHeader: []
      parameters:
        - in: path
          name: contract_id
          required: true
          schema: {type: integer}
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Contract'
      responses:
        '200':
          description: Updated
          content:
            application/json:
              schema: $ref: '#/components/schemas/Contract'
        '400':
          description: Validation error
          content:
            application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}
```

```
      '401': {description: Unauthorized}
      '403': {description: Forbidden}
      '404': {description: Not found}
  delete:
    summary: Delete Contract by ID
    security:
      - cookieAuth: []
        csrfHeader: []
    parameters:
      - in: path
        name: contract_id
        required: true
        schema: {type: integer}
    responses:
      '204': {description: Deleted}
      '401': {description: Unauthorized}
      '403': {description: Forbidden}
      '404': {description: Not found}
      '409':
        description: FK/uniqueness constraint violation
        content:
          application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}

/documents/:
  post:
    summary: Create Document
    security:
      - cookieAuth: []
        csrfHeader: []
    requestBody:
      required: true
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Document'
    responses:
      '201':
        description: Created
        content:
          application/json:
            schema: $ref: '#/components/schemas/Document'
      '400':
        description: Validation error
        content:
          application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}
      '401': {description: Unauthorized}
      '403': {description: Forbidden}
/documents/{start}/{end}/:
  get:
    summary: List Document by inclusive ID range
```

```
    security:
      - cookieAuth: []
    parameters:
      - in: path
        name: start
        required: true
        schema: {type: integer, minimum: 0}
      - in: path
        name: end
        required: true
        schema: {type: integer, minimum: 0}
    responses:
      '200':
        description: Range result
        content:
          application/json:
            schema:
              type: array
              items: {$ref: '#/components/schemas/Document'}
      '400':
        description: Bad range
        content:
          application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}
      '401': {description: Unauthorized}
  /documents/{document_id}:
    put:
      summary: Update Document by ID
      security:
        - cookieAuth: []
          csrfHeader: []
      parameters:
        - in: path
          name: document_id
          required: true
          schema: {type: integer}
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Document'
      responses:
        '200':
          description: Updated
          content:
            application/json:
              schema: $ref: '#/components/schemas/Document'
        '400':
          description: Validation error
          content:
```

```
          application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}
      '401': {description: Unauthorized}
      '403': {description: Forbidden}
      '404': {description: Not found}
  delete:
    summary: Delete Document by ID
    security:
      - cookieAuth: []
        csrfHeader: []
    parameters:
      - in: path
        name: document_id
        required: true
        schema: {type: integer}
    responses:
      '204': {description: Deleted}
      '401': {description: Unauthorized}
      '403': {description: Forbidden}
      '404': {description: Not found}
      '409':
        description: FK/uniqueness constraint violation
        content:
          application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}

/communications/:
  post:
    summary: Create Communication
    security:
      - cookieAuth: []
        csrfHeader: []
    requestBody:
      required: true
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Communication'
    responses:
      '201':
        description: Created
        content:
          application/json:
            schema: $ref: '#/components/schemas/Communication'
      '400':
        description: Validation error
        content:
          application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}
      '401': {description: Unauthorized}
      '403': {description: Forbidden}
/communications/{start}/{end}/:
  get:
```

```
    summary: List Communication by inclusive ID range
    security:
      - cookieAuth: []
    parameters:
      - in: path
        name: start
        required: true
        schema: {type: integer, minimum: 0}
      - in: path
        name: end
        required: true
        schema: {type: integer, minimum: 0}
    responses:
      '200':
        description: Range result
        content:
          application/json:
            schema:
              type: array
              items: {$ref: '#/components/schemas/Communication'}
      '400':
        description: Bad range
        content:
          application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}
      '401': {description: Unauthorized}
  /communications/{communication_id}:
    put:
    summary: Update Communication by ID
    security:
      - cookieAuth: []
        csrfHeader: []
    parameters:
      - in: path
        name: communication_id
        required: true
        schema: {type: integer}
    requestBody:
      required: true
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Communication'
    responses:
      '200':
        description: Updated
        content:
          application/json:
            schema: $ref: '#/components/schemas/Communication'
      '400':
        description: Validation error
```

```
        content:
          application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}
      '401': {description: Unauthorized}
      '403': {description: Forbidden}
      '404': {description: Not found}
  delete:
    summary: Delete Communication by ID
    security:
      - cookieAuth: []
        csrfHeader: []
    parameters:
      - in: path
        name: communication_id
        required: true
        schema: {type: integer}
    responses:
      '204': {description: Deleted}
      '401': {description: Unauthorized}
      '403': {description: Forbidden}
      '404': {description: Not found}
      '409':
        description: FK/uniqueness constraint violation
        content:
          application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}

/pipelines/:
  post:
    summary: Create Pipeline
    security:
      - cookieAuth: []
        csrfHeader: []
    requestBody:
      required: true
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Pipeline'
    responses:
      '201':
        description: Created
        content:
          application/json:
            schema: $ref: '#/components/schemas/Pipeline'
      '400':
        description: Validation error
        content:
          application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}
      '401': {description: Unauthorized}
      '403': {description: Forbidden}
/pipelines/{start}/{end}/:
```

```
    get:
      summary: List Pipeline by inclusive ID range
      security:
        - cookieAuth: []
      parameters:
        - in: path
          name: start
          required: true
          schema: {type: integer, minimum: 0}
        - in: path
          name: end
          required: true
          schema: {type: integer, minimum: 0}
      responses:
        '200':
          description: Range result
          content:
            application/json:
              schema:
                type: array
                items: {$ref: '#/components/schemas/Pipeline'}
        '400':
          description: Bad range
          content:
            application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}
        '401': {description: Unauthorized}
  /pipelines/{pipeline_id}:
    put:
      summary: Update Pipeline by ID
      security:
        - cookieAuth: []
          csrfHeader: []
      parameters:
        - in: path
          name: pipeline_id
          required: true
          schema: {type: integer}
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Pipeline'
      responses:
        '200':
          description: Updated
          content:
            application/json:
              schema: $ref: '#/components/schemas/Pipeline'
        '400':
```

```
        description: Validation error
        content:
          application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}
      '401': {description: Unauthorized}
      '403': {description: Forbidden}
      '404': {description: Not found}
    delete:
      summary: Delete Pipeline by ID
      security:
        - cookieAuth: []
          csrfHeader: []
      parameters:
        - in: path
          name: pipeline_id
          required: true
          schema: {type: integer}
      responses:
        '204': {description: Deleted}
        '401': {description: Unauthorized}
        '403': {description: Forbidden}
        '404': {description: Not found}
        '409':
          description: FK/uniqueness constraint violation
          content:
            application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}

  /pipeline_stages/:
    post:
      summary: Create PipelineStage
      security:
        - cookieAuth: []
          csrfHeader: []
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/PipelineStage'
      responses:
        '201':
          description: Created
          content:
            application/json:
              schema: $ref: '#/components/schemas/PipelineStage'
        '400':
          description: Validation error
          content:
            application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}
        '401': {description: Unauthorized}
        '403': {description: Forbidden}
```

```
  /pipeline_stages/{start}/{end}/:
    get:
      summary: List PipelineStage by inclusive ID range
      security:
        - cookieAuth: []
      parameters:
        - in: path
          name: start
          required: true
          schema: {type: integer, minimum: 0}
        - in: path
          name: end
          required: true
          schema: {type: integer, minimum: 0}
      responses:
        '200':
          description: Range result
          content:
            application/json:
              schema:
                type: array
                items: {$ref: '#/components/schemas/PipelineStage'}
        '400':
          description: Bad range
          content:
            application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}
        '401': {description: Unauthorized}
  /pipeline_stages/{stage_id}:
    put:
      summary: Update PipelineStage by ID
      security:
        - cookieAuth: []
          csrfHeader: []
      parameters:
        - in: path
          name: stage_id
          required: true
          schema: {type: integer}
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/PipelineStage'
      responses:
        '200':
          description: Updated
          content:
            application/json:
              schema: $ref: '#/components/schemas/PipelineStage'
```

```
      '400':
        description: Validation error
        content:
          application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}
      '401': {description: Unauthorized}
      '403': {description: Forbidden}
      '404': {description: Not found}
    delete:
      summary: Delete PipelineStage by ID
      security:
        - cookieAuth: []
          csrfHeader: []
      parameters:
        - in: path
          name: stage_id
          required: true
          schema: {type: integer}
      responses:
        '204': {description: Deleted}
        '401': {description: Unauthorized}
        '403': {description: Forbidden}
        '404': {description: Not found}
        '409':
          description: FK/uniqueness constraint violation
          content:
            application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}

  /opportunities_task_ticket/:
    post:
      summary: Create OpportunityTaskTicket
      security:
        - cookieAuth: []
          csrfHeader: []
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/OpportunityTaskTicket'
      responses:
        '201':
          description: Created
          content:
            application/json:
              schema: $ref: '#/components/schemas/OpportunityTaskTicket'
        '400':
          description: Validation error
          content:
            application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}
        '401': {description: Unauthorized}
```

```
      '403': {description: Forbidden}
 /opportunities_task_ticket/{start}/{end}/:
   get:
     summary: List OpportunityTaskTicket by inclusive ID range
     security:
       - cookieAuth: []
     parameters:
       - in: path
         name: start
         required: true
         schema: {type: integer, minimum: 0}
       - in: path
         name: end
         required: true
         schema: {type: integer, minimum: 0}
     responses:
       '200':
         description: Range result
         content:
           application/json:
             schema:
               type: array
               items: {$ref: '#/components/schemas/OpportunityTaskTicket'}
       '400':
         description: Bad range
         content:
           application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}
       '401': {description: Unauthorized}
 /opportunities_task_ticket/{opportunity_id}:
   put:
     summary: Update OpportunityTaskTicket by ID
     security:
       - cookieAuth: []
         csrfHeader: []
     parameters:
       - in: path
         name: opportunity_id
         required: true
         schema: {type: integer}
     requestBody:
       required: true
       content:
         application/json:
           schema:
             $ref: '#/components/schemas/OpportunityTaskTicket'
     responses:
       '200':
         description: Updated
         content:
           application/json:
```

```
          schema: $ref: '#/components/schemas/OpportunityTaskTicket'
      '400':
        description: Validation error
        content:
          application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}
      '401': {description: Unauthorized}
      '403': {description: Forbidden}
      '404': {description: Not found}
  delete:
    summary: Delete OpportunityTaskTicket by ID
    security:
      - cookieAuth: []
        csrfHeader: []
    parameters:
      - in: path
        name: opportunity_id
        required: true
        schema: {type: integer}
    responses:
      '204': {description: Deleted}
      '401': {description: Unauthorized}
      '403': {description: Forbidden}
      '404': {description: Not found}
      '409':
        description: FK/uniqueness constraint violation
        content:
          application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}

/properties/:
  post:
    summary: Create Property
    security:
      - cookieAuth: []
        csrfHeader: []
    requestBody:
      required: true
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Property'
    responses:
      '201':
        description: Created
        content:
          application/json:
            schema: $ref: '#/components/schemas/Property'
      '400':
        description: Validation error
        content:
          application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}
```

```
      '401': {description: Unauthorized}
      '403': {description: Forbidden}
 /properties/{start}/{end}/:
  get:
    summary: List Property by inclusive ID range
    security:
      - cookieAuth: []
    parameters:
      - in: path
        name: start
        required: true
        schema: {type: integer, minimum: 0}
      - in: path
        name: end
        required: true
        schema: {type: integer, minimum: 0}
    responses:
      '200':
        description: Range result
        content:
          application/json:
            schema:
              type: array
              items: {$ref: '#/components/schemas/Property'}
      '400':
        description: Bad range
        content:
          application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}
      '401': {description: Unauthorized}
 /properties/{property_id}:
  put:
    summary: Update Property by ID
    security:
      - cookieAuth: []
        csrfHeader: []
    parameters:
      - in: path
        name: property_id
        required: true
        schema: {type: integer}
    requestBody:
      required: true
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Property'
    responses:
      '200':
        description: Updated
        content:
```

```
          application/json:
            schema: $ref: '#/components/schemas/Property'
      '400':
        description: Validation error
        content:
          application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}
      '401': {description: Unauthorized}
      '403': {description: Forbidden}
      '404': {description: Not found}
  delete:
    summary: Delete Property by ID
    security:
      - cookieAuth: []
        csrfHeader: []
    parameters:
      - in: path
        name: property_id
        required: true
        schema: {type: integer}
    responses:
      '204': {description: Deleted}
      '401': {description: Unauthorized}
      '403': {description: Forbidden}
      '404': {description: Not found}
      '409':
        description: FK/uniqueness constraint violation
        content:
          application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}

/owner/:
  post:
    summary: Create Owner
    security:
      - cookieAuth: []
        csrfHeader: []
    requestBody:
      required: true
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Owner'
    responses:
      '201':
        description: Created
        content:
          application/json:
            schema: $ref: '#/components/schemas/Owner'
      '400':
        description: Validation error
        content:
```

```
          application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}
      '401': {description: Unauthorized}
      '403': {description: Forbidden}
  /owner/{start}/{end}/:
    get:
      summary: List Owner by inclusive ID range
      security:
        - cookieAuth: []
      parameters:
        - in: path
          name: start
          required: true
          schema: {type: integer, minimum: 0}
        - in: path
          name: end
          required: true
          schema: {type: integer, minimum: 0}
      responses:
        '200':
          description: Range result
          content:
            application/json:
              schema:
                type: array
                items: {$ref: '#/components/schemas/Owner'}
        '400':
          description: Bad range
          content:
            application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}
        '401': {description: Unauthorized}
  /owner/{owner_id}:
    put:
      summary: Update Owner by ID
      security:
        - cookieAuth: []
          csrfHeader: []
      parameters:
        - in: path
          name: owner_id
          required: true
          schema: {type: integer}
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Owner'
      responses:
        '200':
          description: Updated
```

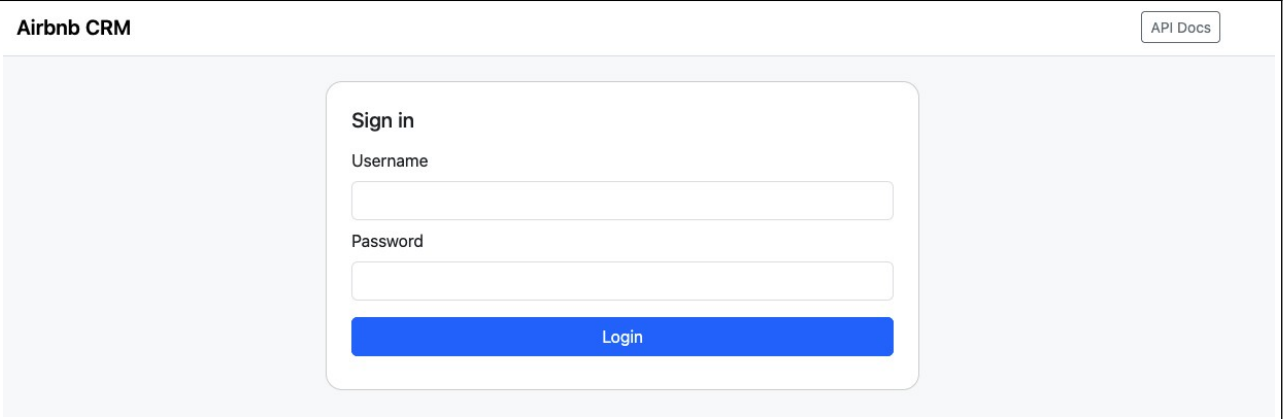
```
        content:
          application/json:
            schema: $ref: '#/components/schemas/Owner'
      '400':
        description: Validation error
        content:
          application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}
      '401': {description: Unauthorized}
      '403': {description: Forbidden}
      '404': {description: Not found}
  delete:
    summary: Delete Owner by ID
    security:
      - cookieAuth: []
        csrfHeader: []
    parameters:
      - in: path
        name: owner_id
        required: true
        schema: {type: integer}
    responses:
      '204': {description: Deleted}
      '401': {description: Unauthorized}
      '403': {description: Forbidden}
      '404': {description: Not found}
      '409':
        description: FK/uniqueness constraint violation
        content:
          application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}

/owner_property_map/:
  post:
    summary: Create OwnerPropertyMap
    security:
      - cookieAuth: []
        csrfHeader: []
    requestBody:
      required: true
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/OwnerPropertyMap'
    responses:
      '201':
        description: Created
        content:
          application/json:
            schema: $ref: '#/components/schemas/OwnerPropertyMap'
      '400':
        description: Validation error
```

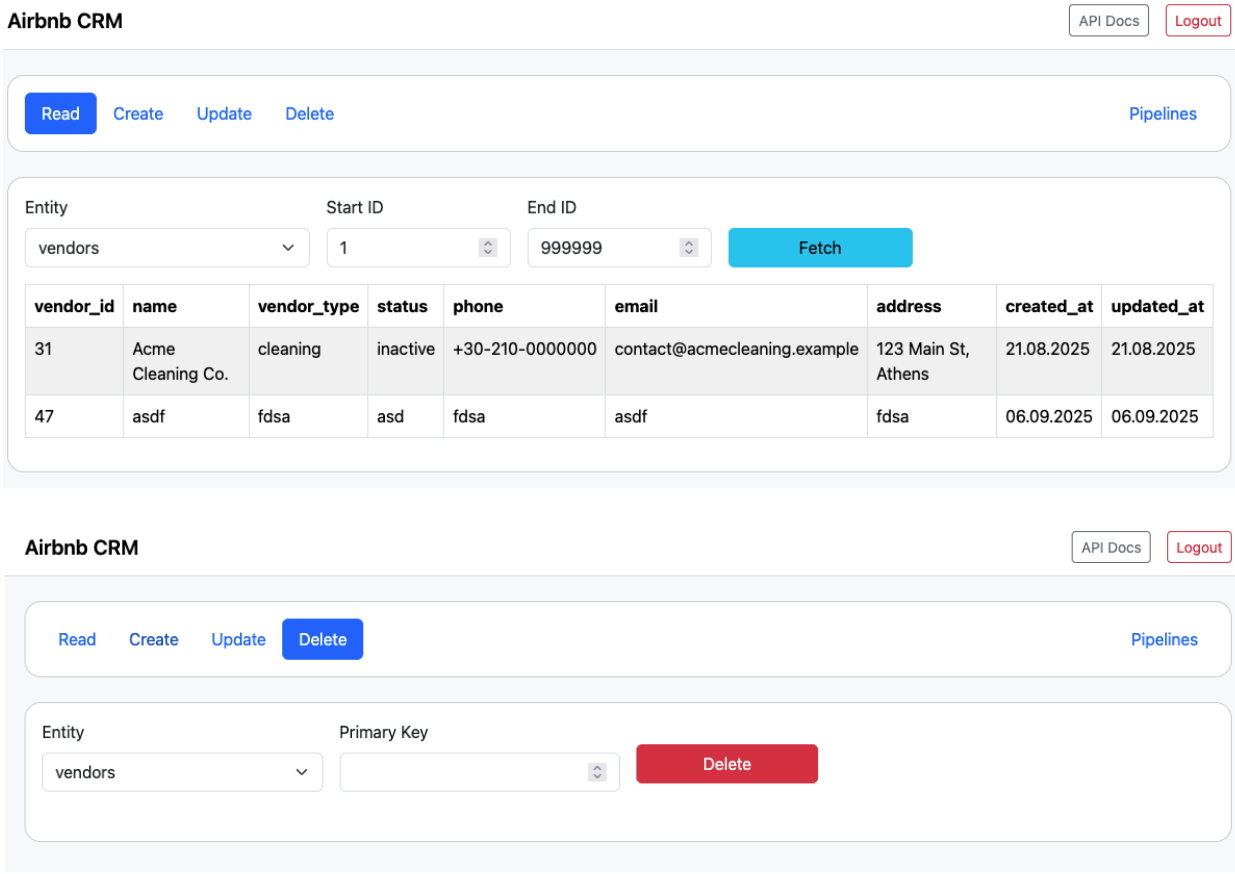
```
        content:
          application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}
      '401': {description: Unauthorized}
      '403': {description: Forbidden}
 /owner_property_map/{start}/{end}/:
  get:
    summary: List OwnerPropertyMap by inclusive ID range
    security:
      - cookieAuth: []
    parameters:
      - in: path
        name: start
        required: true
        schema: {type: integer, minimum: 0}
      - in: path
        name: end
        required: true
        schema: {type: integer, minimum: 0}
    responses:
      '200':
        description: Range result
        content:
          application/json:
            schema:
              type: array
              items: {$ref: '#/components/schemas/OwnerPropertyMap'}
      '400':
        description: Bad range
        content:
          application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}
      '401': {description: Unauthorized}
 /owner_property_map/{map_id}:
  put:
    summary: Update OwnerPropertyMap by ID
    security:
      - cookieAuth: []
        csrfHeader: []
    parameters:
      - in: path
        name: map_id
        required: true
        schema: {type: integer}
    requestBody:
      required: true
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/OwnerPropertyMap'
    responses:
      '200':
```

```
        description: Updated
        content:
          application/json:
            schema: $ref: '#/components/schemas/OwnerPropertyMap'
      '400':
        description: Validation error
        content:
          application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}
      '401': {description: Unauthorized}
      '403': {description: Forbidden}
      '404': {description: Not found}
  delete:
    summary: Delete OwnerPropertyMap by ID
    security:
      - cookieAuth: []
        csrfHeader: []
    parameters:
      - in: path
        name: map_id
        required: true
        schema: {type: integer}
    responses:
      '204': {description: Deleted}
      '401': {description: Unauthorized}
      '403': {description: Forbidden}
      '404': {description: Not found}
      '409':
        description: FK/uniqueness constraint violation
        content:
          application/json: {schema: {$ref: '#/components/schemas/ErrorEnvelope'}}
```

8.4 Screenshots & UI walk-throughs

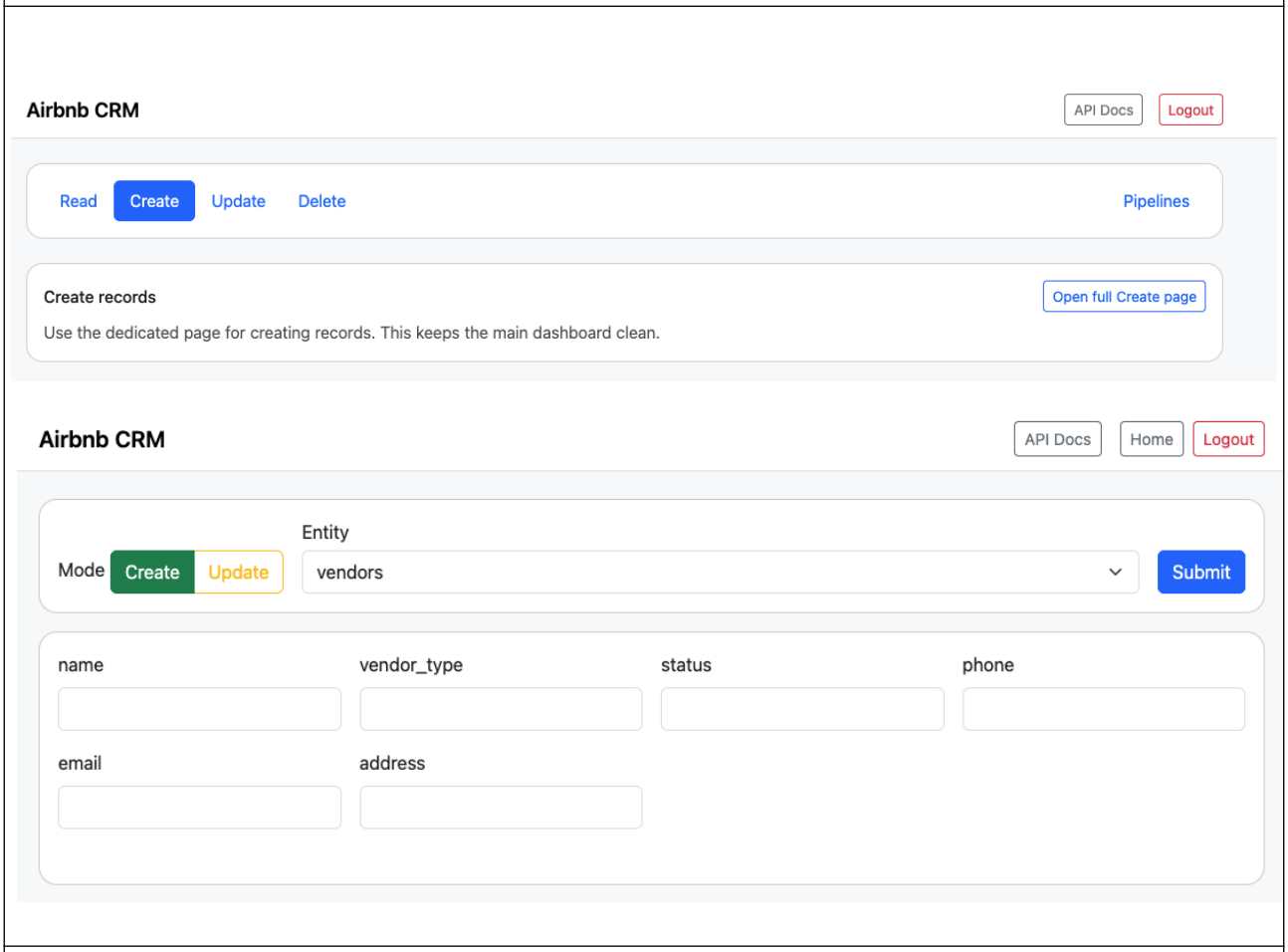
Login

This is the screen the user is welcomed by in order to login to the system. There are fields to input their credentials and a button named login in order to login to the system. On the top right corner there is always in the system pages a button to the system documentation.

Read-delete menu



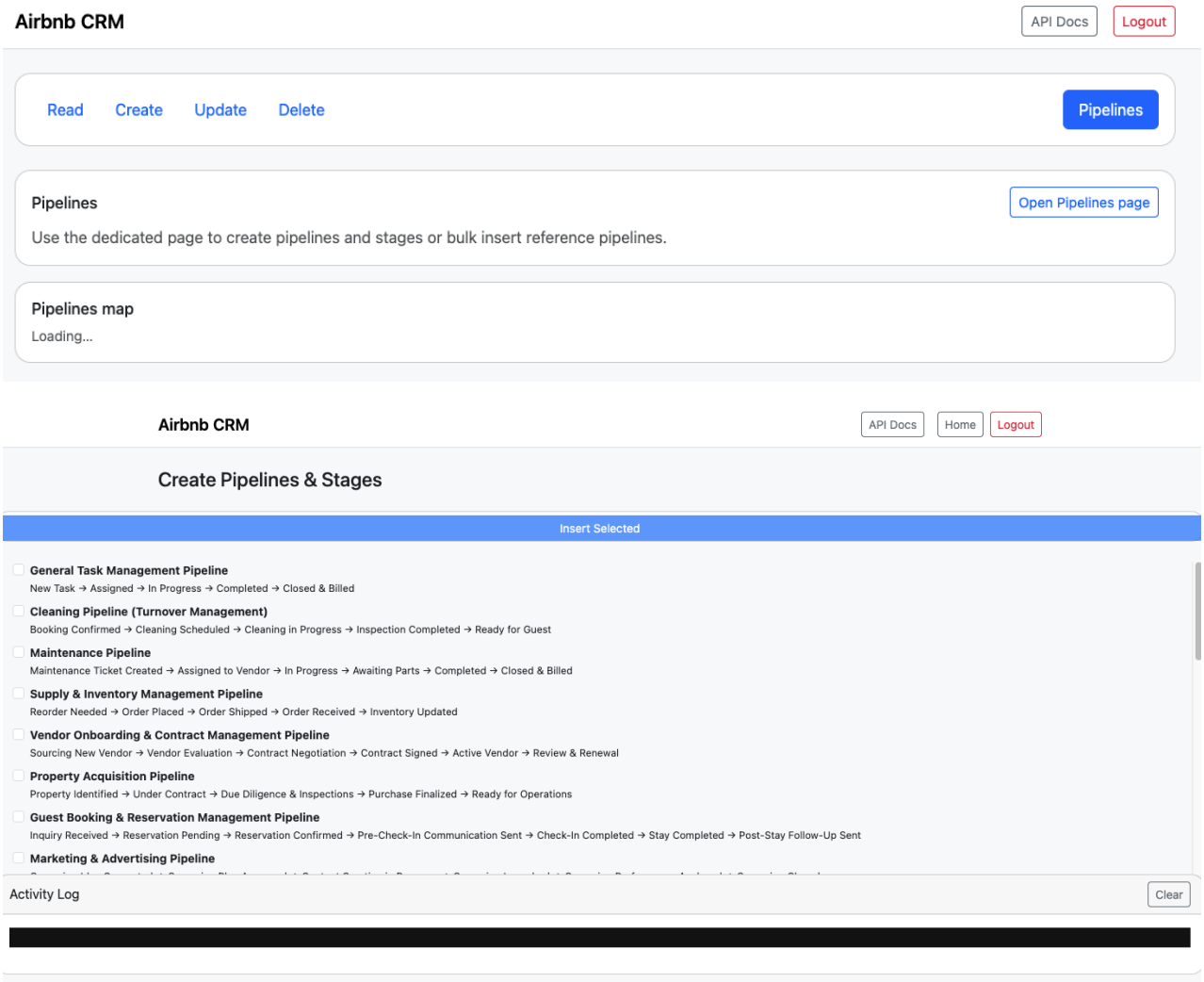
These two screenshots belong to the same page that follows a successfull login; a tabbed menu of operations over the database tables and another tab for the pipeline inserts. The first screenshot in this section shows the read operation from a table; this section shows the required fields needed (the table, input from a scrolldown menu, an id range) and a get button, after its press the frontend renders the data to a table bellow the input fields. Delete follows form, with a table selection ellement, a record key input field and a button to trigger the table record delete.

Create-update menu



The create-update menu can be accessed by the button in the create and update tabs of the index.html page as presented here. Through this page the use can select the operation required (insert/update) from the apropriate buttons, choose the table he wants to work on, fill the needed data fields and finalise the operation through the submit button.

Pipelines menu



The pipelines map insert allows for a multi-table insert (that reflects a set of operations needed to be done per property) to the database. The menu can be accessed by the pipelines tab and button on the index.html page.

8.6 Database 3rd normal form

What 3NF means for a database schema.

A schema is in **3rd Normal Form** if:

1. It is in **2NF** (i.e., no partial dependency on composite keys).
2. **No transitive dependencies**: Non-key attributes must depend **only on the primary key**, and **nothing else**.

In other words:

- Each table should represent a **single concept**.

- All fields in a table should describe **only that concept's primary key**, and not other things derived from it.

Problem

- Fields like `vendor_type`, `status`, or possibly even `address` may be **repeating values** or contain **embedded structure**.
- Address might contain embedded city/zip/country.
- `legal_id` likely has a format worth validating separately.
- Phone/email might be reused across roles.

General Patterns for 3NF

Current Practice	3NF Transformation
Repeated values (`status`, `type`)	Move to **lookup tables**
Embedded fields (`address`, `period`)	Split into **composite tables**
Redundant contact info across roles	Centralize into **shared contact table**
Foreign keys stored as loose strings	Reference **ID-based relations**

Side Effects of 3NF

Pros

- Enforces **semantic clarity** and reduces data redundancy.
- Enables shared **reporting models** (e.g., multiple vendors per performance period).
- Makes schema more **flexible and maintainable** in the long run.

Cons

- Requires **more joins**, which could affect performance.
- Adds **complexity** to queries and CRUD logic (especially for manual forms).
- Can be **overkill for small or read-heavy systems** like embedded CRMs.

Introducing **Third Normal Form (3NF)** into your schema resolves several structural and long-term integrity issues. Here's a clear, categorized list of what 3NF **actively prevents or improves**:

Issues Resolved by 3rd Normal Form

1. Data Redundancy

- **Problem**: Repeating values like `vendor_type = "cleaner"` in multiple rows.
- **3NF Fix**: Move such values into a separate `vendor_types` table.
- **Benefit**: Reduces storage waste and avoids inconsistency (e.g., "Cleaner" vs. "cleaner").

2. Update Anomalies

- **Problem**: Changing a vendor type name ("Plumber" → "Plumbing Service") requires updating it in **every vendor row**.

- **3NF Fix**: Update it once in the lookup table.
- **Benefit**: Centralized, atomic updates.

3. Insert Anomalies

- **Problem**: You can’t insert a new vendor type unless a vendor row uses it.
- **3NF Fix**: Create a record in the `vendor_types` table independently.
- **Benefit**: Independent population of reference data.

4. Delete Anomalies

- **Problem**: Deleting the last vendor with a given status may **erase knowledge** of that status.
- **3NF Fix**: Status lives in its own table, decoupled from vendors.
- **Benefit**: Safe deletion of data without cascading loss of meaning.

5. Transitive Dependencies

- **Problem**: A column depends not on the primary key, but on another non-key column.(E.g., `vendor_type → default_contract_terms`)
- **3NF Fix**: Break out such logic into a related table.
- **Benefit**: Only the primary key determines the rest of the row, ensuring semantic clarity.

6. Ambiguous Data Semantics

- Problem: `address` field contains multiple concepts: street, zip, city, country.
- 3NF Fix: Normalize into an `addresses` table with structured fields.
- Benefit: Easier parsing, querying, and location-based filtering.

7. Poor Queryability for Reporting

- **Problem**: Want to find vendors by region or owners by country—but `address` is a blob.
- **3NF Fix**: Structural decomposition allows targeted queries.
- Benefit: Improved analytics and easier integration with other systems (e.g., mapping APIs).

8. Duplication of Logic in Code

- **Problem**: Validation logic for things like status or type must be repeated in every language (Python, JS).
- **3NF Fix**: The DB enforces what values are allowed.
- **Benefit**: Simplifies validation and reduces bugs.

9. Non-Scalable Design

- **Problem**: In flat schemas, adding fields like `country_tax_rules` or `region_defaults` means bloating existing tables.
- **3NF Fix**: Lookup tables can be extended modularly.
- **Benefit**: Schema grows **horizontally**, not vertically—keeps each entity simple.

10. Inconsistent Referential Integrity

Problem: Manually entered free-text values can lead to mismatched references.
3NF Fix: Foreign key enforcement ensures linked data exists.
Benefit: Prevents orphan records and logical contradictions.

Migration plan (field-by-field)

vendors	• Move unique vendor_type values to vendor_type; set vendors.vendor_type_id accordingly. Same for vendor_status. Drop old text columns. • Create address rows from vendors.address; replace with address_id. • If you want to keep phones/emails per company, either put them into contact_channel linked to a “company contact” or keep vendor.phone/email as is (still 3NF if considered attributes of vendor, but centralizing channels reduces duplication).
vendor_contacts	• For each row, create acontact (first_name,last_name), attach channels in contact_channel for phone/email, and keep job_title. Replace with (vendor_contact_id, vendor_id, contact_id, job_title).
vendor_scorecard	• Build performance_period distinct pairs from (period_start,period_end), assign period_id; drop period_start/period_end; add period_id.
contracts	• Create contract_status list from distinct values; wire contract_status_id; drop free-text status.
documents	• No structural change required beyond FK integrity; optional document_type later.
communications	• Create comm_type list from distinct comm_type strings; map to comm_type_id. Convert user_id TEXT to FK into user.
pipelines / stages / tickets	• Keep pipeline and pipeline_stage as is (already well-formed). Move ticket status/priority to lookups and rename to ticket for clarity.
owner / properties / owner_property_map	• Convert owner to point at contact + address as above; property gets address_id. Keep the M\:N map.

A proposed 3rd NF schema

```
CREATE TABLE vendor_type (
  vendor_type_id   INTEGER PRIMARY KEY,
  name             TEXT NOT NULL UNIQUE
);
```

```
CREATE TABLE vendor_status (
  vendor_status_id INTEGER PRIMARY KEY,
  name             TEXT NOT NULL UNIQUE
);
```

```
CREATE TABLE contract_status (
  contract_status_id INTEGER PRIMARY KEY,
  name               TEXT NOT NULL UNIQUE
);
```

```
CREATE TABLE comm_type (
  comm_type_id    INTEGER PRIMARY KEY,
  name            TEXT NOT NULL UNIQUE
);
```

```
CREATE TABLE ticket_status (
  ticket_status_id  INTEGER PRIMARY KEY,
  name              TEXT NOT NULL UNIQUE
);
```

```
CREATE TABLE ticket_priority (
  ticket_priority_id INTEGER PRIMARY KEY,
  name               TEXT NOT NULL UNIQUE
);
```

```
CREATE TABLE address (
  address_id   INTEGER PRIMARY KEY,
  line1        TEXT NOT NULL,
  line2        TEXT,
  city         TEXT,
  region       TEXT,
  postal_code  TEXT,
  country      TEXT
);
```

```
CREATE TABLE contact (
  contact_id   INTEGER PRIMARY KEY,
  first_name   TEXT NOT NULL,
  last_name    TEXT NOT NULL
);
```

```
CREATE TABLE contact_channel (
  channel_id   INTEGER PRIMARY KEY,
  contact_id   INTEGER NOT NULL,
  kind         TEXT NOT NULL CHECK (kind IN ('phone','email')),
  value        TEXT NOT NULL,
```

```
  is_primary   INTEGER NOT NULL DEFAULT 0 CHECK (is_primary IN (0,1)),
  FOREIGN KEY (contact_id) REFERENCES contact(contact_id) ON DELETE CASCADE ON
UPDATE CASCADE
);
```

```
CREATE INDEX idx_contact_channel_contact ON contact_channel(contact_id);
```

```
-- One-and-only-one primary channel per contact (when is_primary=1)
CREATE TRIGGER trg_contact_channel_one_primary
BEFORE INSERT ON contact_channel
FOR EACH ROW
WHEN NEW.is_primary = 1
BEGIN
  -- Demote any existing primaries for the same contact
  UPDATE contact_channel
     SET is_primary = 0
   WHERE contact_id = NEW.contact_id
     AND is_primary = 1;
END;
```

```
CREATE TRIGGER trg_contact_channel_one_primary_upd
BEFORE UPDATE OF is_primary ON contact_channel
FOR EACH ROW
WHEN NEW.is_primary = 1
BEGIN
  UPDATE contact_channel
     SET is_primary = 0
   WHERE contact_id = NEW.contact_id
     AND channel_id <> NEW.channel_id
     AND is_primary = 1;
END;
```

```
CREATE TABLE "user" (
  user_id     INTEGER PRIMARY KEY,
  user_name   TEXT NOT NULL UNIQUE,
  user_role   TEXT NOT NULL
);
```

```
CREATE TABLE vendor (
  vendor_id         INTEGER PRIMARY KEY,
  name              TEXT NOT NULL,
  vendor_type_id    INTEGER NOT NULL,
  vendor_status_id  INTEGER NOT NULL,
  address_id        INTEGER,
  created_at        TEXT DEFAULT (datetime('now')),
  updated_at        TEXT,
  FOREIGN KEY (vendor_type_id)   REFERENCES vendor_type(vendor_type_id)   ON
DELETE RESTRICT ON UPDATE CASCADE,
  FOREIGN KEY (vendor_status_id) REFERENCES vendor_status(vendor_status_id) ON
DELETE RESTRICT ON UPDATE CASCADE,
  FOREIGN KEY (address_id)       REFERENCES address(address_id)          ON DELETE SET
NULL ON UPDATE CASCADE
);
```

```
CREATE INDEX idx_vendor_type   ON vendor(vendor_type_id);
CREATE INDEX idx_vendor_status ON vendor(vendor_status_id);
CREATE INDEX idx_vendor_addr   ON vendor(address_id);
```

```
CREATE TABLE vendor_contact (
  vendor_contact_id INTEGER PRIMARY KEY,
  vendor_id         INTEGER NOT NULL,
  contact_id        INTEGER NOT NULL,
  job_title         TEXT,
  role_note         TEXT,
  UNIQUE (vendor_id, contact_id),
  FOREIGN KEY (vendor_id)  REFERENCES vendor(vendor_id)   ON DELETE CASCADE ON
UPDATE CASCADE,
  FOREIGN KEY (contact_id) REFERENCES contact(contact_id) ON DELETE RESTRICT ON
UPDATE CASCADE
);
```

```
CREATE INDEX idx_vendor_contact_vendor  ON vendor_contact(vendor_id);
CREATE INDEX idx_vendor_contact_contact ON vendor_contact(contact_id);
```

```
CREATE TABLE property (
  property_id   INTEGER PRIMARY KEY,
  property_name TEXT NOT NULL,
  address_id    INTEGER,
  status        TEXT,
  FOREIGN KEY (address_id) REFERENCES address(address_id) ON DELETE SET NULL ON
UPDATE CASCADE
);
```

```
CREATE INDEX idx_property_addr ON property(address_id);
```

```
CREATE TABLE owner (
  owner_id    INTEGER PRIMARY KEY,
  contact_id  INTEGER NOT NULL,
  legal_id    TEXT,
  address_id  INTEGER,
  FOREIGN KEY (contact_id) REFERENCES contact(contact_id) ON DELETE RESTRICT ON
UPDATE CASCADE,
  FOREIGN KEY (address_id) REFERENCES address(address_id) ON DELETE SET NULL ON
UPDATE CASCADE
);
```

```
CREATE INDEX idx_owner_contact ON owner(contact_id);
CREATE INDEX idx_owner_addr    ON owner(address_id);
```

```
CREATE TABLE owner_property_map (
  map_id     INTEGER PRIMARY KEY,
  owner_id   INTEGER NOT NULL,
  property_id INTEGER NOT NULL,
  UNIQUE (owner_id, property_id),
  FOREIGN KEY (owner_id)   REFERENCES owner(owner_id)       ON DELETE CASCADE
ON UPDATE CASCADE,
  FOREIGN KEY (property_id) REFERENCES property(property_id) ON DELETE CASCADE
ON UPDATE CASCADE
```

```
);
```

```
CREATE INDEX idx_opm_owner    ON owner_property_map(owner_id);
CREATE INDEX idx_opm_property ON owner_property_map(property_id);
```

```
CREATE TABLE pipeline (
  pipeline_id   INTEGER PRIMARY KEY,
  pipeline_name TEXT NOT NULL,
  description   TEXT
);
```

```
CREATE TABLE pipeline_stage (
  stage_id     INTEGER PRIMARY KEY,
  pipeline_id  INTEGER NOT NULL,
  stage_name   TEXT NOT NULL,
  stage_order  INTEGER NOT NULL,
  UNIQUE (pipeline_id, stage_order),
  UNIQUE (pipeline_id, stage_name),
  FOREIGN KEY (pipeline_id) REFERENCES pipeline(pipeline_id) ON DELETE CASCADE
ON UPDATE CASCADE
);
```

```
CREATE INDEX idx_stage_pipeline ON pipeline_stage(pipeline_id);
```

```
CREATE TABLE ticket (
  opportunity_id     INTEGER PRIMARY KEY,
  pipeline_id        INTEGER NOT NULL,
  stage_id           INTEGER NOT NULL,
  vendor_id          INTEGER,
  property_id        INTEGER,
  title              TEXT NOT NULL,
  description        TEXT,
  due_date           TEXT,
  ticket_status_id   INTEGER,
  ticket_priority_id INTEGER,
  FOREIGN KEY (pipeline_id)        REFERENCES pipeline(pipeline_id)           ON DELETE
RESTRICT ON UPDATE CASCADE,
  FOREIGN KEY (stage_id)           REFERENCES pipeline_stage(stage_id)        ON DELETE
RESTRICT ON UPDATE CASCADE,
  FOREIGN KEY (vendor_id)          REFERENCES vendor(vendor_id)               ON DELETE
SET NULL ON UPDATE CASCADE,
  FOREIGN KEY (property_id)        REFERENCES property(property_id)           ON DELETE
SET NULL ON UPDATE CASCADE,
  FOREIGN KEY (ticket_status_id)   REFERENCES ticket_status(ticket_status_id) ON DELETE
SET NULL ON UPDATE CASCADE,
  FOREIGN KEY (ticket_priority_id) REFERENCES ticket_priority(ticket_priority_id) ON
DELETE SET NULL ON UPDATE CASCADE
);
```

```
CREATE INDEX idx_ticket_pipeline  ON ticket(pipeline_id);
CREATE INDEX idx_ticket_stage     ON ticket(stage_id);
CREATE INDEX idx_ticket_vendor    ON ticket(vendor_id);
CREATE INDEX idx_ticket_property  ON ticket(property_id);
```

```
CREATE INDEX idx_ticket_status    ON ticket(ticket_status_id);
CREATE INDEX idx_ticket_priority  ON ticket(ticket_priority_id);
```

```
-- Ensure a ticket's stage belongs to the same pipeline
CREATE TRIGGER trg_ticket_stage_pipeline_consistency_ins
BEFORE INSERT ON ticket
FOR EACH ROW
BEGIN
  SELECT
    CASE
      WHEN (SELECT pipeline_id FROM pipeline_stage WHERE stage_id = NEW.stage_id) <>
NEW.pipeline_id
      THEN RAISE(ABORT, 'ticket.stage_id must belong to ticket.pipeline_id')
    END;
END;
```

```
CREATE TRIGGER trg_ticket_stage_pipeline_consistency_upd
BEFORE UPDATE OF stage_id, pipeline_id ON ticket
FOR EACH ROW
BEGIN
  SELECT
    CASE
      WHEN (SELECT pipeline_id FROM pipeline_stage WHERE stage_id = NEW.stage_id) <>
NEW.pipeline_id
      THEN RAISE(ABORT, 'ticket.stage_id must belong to ticket.pipeline_id')
    END;
END;
```

```
CREATE TABLE contract (
  contract_id         INTEGER PRIMARY KEY,
  vendor_id           INTEGER NOT NULL,
  contract_name       TEXT NOT NULL,
  start_date          TEXT NOT NULL,
  end_date            TEXT,
  renewal_terms       TEXT,
  contract_status_id  INTEGER NOT NULL,
  FOREIGN KEY (vendor_id)         REFERENCES vendor(vendor_id)                ON DELETE
RESTRICT ON UPDATE CASCADE,
  FOREIGN KEY (contract_status_id) REFERENCES contract_status(contract_status_id) ON
DELETE RESTRICT ON UPDATE CASCADE
);
```

```
CREATE INDEX idx_contract_vendor  ON contract(vendor_id);
CREATE INDEX idx_contract_status  ON contract(contract_status_id);
```

```
CREATE TABLE document (
  document_id INTEGER PRIMARY KEY,
  vendor_id   INTEGER NOT NULL,
  contract_id INTEGER NOT NULL,
  file_name   TEXT NOT NULL,
  file_path   TEXT NOT NULL,
  uploaded_at TEXT NOT NULL DEFAULT (datetime('now')),
  FOREIGN KEY (vendor_id)   REFERENCES vendor(vendor_id)     ON DELETE RESTRICT
```

```
ON UPDATE CASCADE,
  FOREIGN KEY (contract_id) REFERENCES contract(contract_id) ON DELETE CASCADE
ON UPDATE CASCADE
);
```

```
CREATE INDEX idx_document_vendor   ON document(vendor_id);
CREATE INDEX idx_document_contract ON document(contract_id);
```

```
-- A document's vendor must match its contract's vendor
CREATE TRIGGER trg_document_vendor_matches_contract_ins
BEFORE INSERT ON document
FOR EACH ROW
BEGIN
  SELECT
    CASE
      WHEN (SELECT vendor_id FROM contract WHERE contract_id = NEW.contract_id) <>
NEW.vendor_id
      THEN RAISE(ABORT, 'document.vendor_id must match contract.vendor_id')
    END;
END;
```

```
CREATE TRIGGER trg_document_vendor_matches_contract_upd
BEFORE UPDATE OF vendor_id, contract_id ON document
FOR EACH ROW
BEGIN
  SELECT
    CASE
      WHEN (SELECT vendor_id FROM contract WHERE contract_id = NEW.contract_id) <>
NEW.vendor_id
      THEN RAISE(ABORT, 'document.vendor_id must match contract.vendor_id')
    END;
END;
```

```
CREATE TABLE performance_period (
  period_id   INTEGER PRIMARY KEY,
  period_name TEXT,
  start_date  TEXT NOT NULL,
  end_date    TEXT NOT NULL,
  CHECK (date(start_date) <= date(end_date))
);
```

```
CREATE TABLE vendor_scorecard (
  scorecard_id     INTEGER PRIMARY KEY,
  vendor_id        INTEGER NOT NULL,
  period_id        INTEGER NOT NULL,
  quality_score    NUMERIC DEFAULT 0,
  on_time_delivery NUMERIC DEFAULT 0,
  defect_rate      NUMERIC DEFAULT 0,
  comments         TEXT,
  UNIQUE (vendor_id, period_id),
  FOREIGN KEY (vendor_id) REFERENCES vendor(vendor_id)               ON DELETE
CASCADE ON UPDATE CASCADE,
  FOREIGN KEY (period_id) REFERENCES performance_period(period_id)     ON DELETE
```

```
RESTRICT ON UPDATE CASCADE
);
```

```
CREATE INDEX idx_scorecard_vendor ON vendor_scorecard(vendor_id);
CREATE INDEX idx_scorecard_period ON vendor_scorecard(period_id);
```

```
CREATE TABLE communication (
  communication_id INTEGER PRIMARY KEY,
  vendor_id        INTEGER NOT NULL,
  date_sent        TEXT NOT NULL,
  comm_type_id     INTEGER NOT NULL,
  subject          TEXT NOT NULL,
  content          TEXT NOT NULL,
  user_id          INTEGER,
  followup_needed  INTEGER NOT NULL DEFAULT 0 CHECK (followup_needed IN (0,1)),
  FOREIGN KEY (vendor_id)    REFERENCES vendor(vendor_id)     ON DELETE CASCADE
ON UPDATE CASCADE,
  FOREIGN KEY (comm_type_id) REFERENCES comm_type(comm_type_id) ON DELETE
RESTRICT ON UPDATE CASCADE,
  FOREIGN KEY (user_id)      REFERENCES "user"(user_id)       ON DELETE SET NULL ON
UPDATE CASCADE
);
```

```
CREATE INDEX idx_comm_vendor   ON communication(vendor_id);
CREATE INDEX idx_comm_type     ON communication(comm_type_id);
CREATE INDEX idx_comm_user     ON communication(user_id);
CREATE INDEX idx_comm_date     ON communication(date_sent);
```

8.7 Security concideration and solutions
Authentication & security

In the current code base these issus can cause signifficant issues in the long run if the service is deployed:

- Hard-coded users and secrets beyond their storage to a JSON file.

A fix plan for the current security plan, using JWT:

This requires further code rewrite and testing which go beyond the scope of the assignment.

Alternatives for JWT include protocols like oAuth[12], SAML[13] or OIDC[14] impose similar issues.

8.8 Additions for the service tests

- Include the use of lint and inclusion of headless browsers for further tests of the service
- Auth matrix integration test (hitting your Bottle app):Unauth → 401, Auth user → 403 on POST/PUT/DELETE, Auth admin → 200 on POST/PUT/DELETE. Also assert the error body contract.
- Date parsing edge cases in validators (negative): 31.11.2025, 29.02.2025, 2025-01-01,

01/01/2025.

- Delete protection on all FK paths, not just one (e.g., vendors ↔ vendor_contacts, properties ↔ opportunities_task_tickets, contracts ↔ documents).
- Range GET bounds: exact endpoints (1..1), empty ranges, reversed range rejection (if applicable), large range performance guard (fast return with limit).
- JWT expiry & revocation: create a token with a past exp; ensure /logout adds to revocation set → token then rejected.

On a more design than testing front, the migration to ISO-8601 dates is recommended for more production ready systems and tests should follow to reflect that.

8.9 Installation & run manual

1) Prerequisites

Python 3.10+ (3.11 recommended)

2) Create a virtual environment

macOS/Linux	python3 -m venv .venv source .venv/bin/activate python -m pip install --upgrade pip
Windows (PowerShell)	py -m venv .venv .\.venv\Scripts\Activate.ps1 python -m pip install --upgrade pip

3) Install dependencies

pip install bottle PyJWT

for tests:
pip install pytest

Or drop this as `requirements.txt`:

bottle
PyJWT
pytests

…and then:

pip install -r requirements.txt

4) Database

Place the provided SQLite file in the project root:

airbnb.db

5) Configure (optional)

You can run with defaults. If you prefer environment variables:

macOS/Linux	export PORT=8000 export SECRET_KEY="change-me" export DB_PATH="./airbnb.db"
Windows (PowerShell)	$env:PORT="8000" $env:SECRET_KEY="change-me" $env:DB_PATH="./airbnb.db"

6) Run the app

Your main entry point is:

APP_MAIN.10.py

Server starts at:

https://localhost:8000/index.html

7) Basic usage

- Open the HTML pages (e.g., `index.html`) in a browser, or serve them via the app if routes are provided.
- Login sets HttpOnly cookies; subsequent requests include cookies automatically; writes add `X-CSRF-Token`

8) Run tests

pytest -q
with coverage:
pytest --cov=. --cov-report term-missing --cov-report html

notes:

Set service secret, user name and admin password running SECRET_USERS_PW_ADMIN.py script:

-python3 SECRET_USERS_PW_ADMIN.py SECRET_KEY
-python3 admin <YOUR_NEW_PASSWORD>
-python3 <username> <YOUR_NEW_PASSWORD>

A user named Jane has been already defined. Any other user can be added as the user using the with

the last form of the terminal command.

8.10 Table Design Issues and Future Improvements

Vendors	• Issue: schema / validator drift (some drafts showed a single score, others split into quality_score, on_time_delivery, defect_rate). • Resolution: unified schema with the three numeric dimensions; OpenAPI updated; validator and DTOs aligned. • Future: enforce score ranges (0–100) at validator layer.
Vendor Contacts	• Issue: phone/email initially required; business reality requires optional (a contact may only have one). • Resolution: validator corrected to make them optional; OpenAPI matches. • Future: add uniqueness guard ((vendor_id,email)).
Vendor Scorecard	• Issue: early schema named a single score field, later corrected to three metrics. • Resolution: docs, OpenAPI, and validator harmonized. • Future: add trigger to ensure period_start < period_end and no overlapping scorecards for same vendor/period.
Contracts	• Issue: end_date wrongly required in some places. • Resolution: made optional in validator (open-ended contracts allowed). • Future: add state machine (draft → active → expired) and triggers for non-overlap.
Documents	• Issue: mismatch between created_at vs uploaded_at. • Resolution: adopted uploaded_at (source of truth) + updated_at. • Future: enforce FK consistency (must point to an existing vendor and contract).
Communications	• Issue: followup_needed type mismatch (bool vs string). • Resolution: standardized to string enum "0"/"1". • Future: migrate to boolean once the ORM/JSON layer can map it reliably.
Pipelines & Stages	• Issue: schema drafts showed stage_name as INT. • Resolution: corrected to string (name), with stage_order as INT. • Future: add uniqueness (pipeline_id, stage_order).
Opportunities Task Tickets	• Issue: description originally typed as INT. • Resolution: corrected to string. • Future: enforce FK coverage (pipeline_id, stage_id) and stage progression constraints.
Properties	• Issue: none major; consistent across schema and validators.

	• Future: add UNIQUE(property_name,address) to avoid duplicates.
Owners	• Issue: none major; consistent. • Future: consider adding natural key (legal_id unique).

Owner–Property Map (current design)

Issues identified:
- Duplicates: no UNIQUE(owner_id,property_id) → same pair repeated.
- Updates can silently create duplicates.
- Naming drift: map_id vs owner_property_map_id.
- Delete UX: safest by PK, but UI often only knows the pair.
- Resolution (current system): standardized on single PK map_id + UNIQUE(owner_id,property_id) to prevent duplicate pairs. Thesis documents trade-off with composite PK alternative.

Owner–Property Contract (future update)

To better reflect reality, this relationship will be refactored into a temporal contract table:

New semantics: an owner–property link is valid over a time interval (start_date … end_date), with lifecycle status ∈ {trial, active, paused, ended}.

Rationale: allows multiple legitimate relationships over time (trials, pauses, reactivations, terminations) without creating duplicates.

Theoretical grounding:

Valid time vs transaction time separation.

Invariants:
- I1: No overlaps for same (owner, property).
- I2: At most one open interval (end_date=NULL).
- I3: Well-formed interval (start < end).
- I4: State machine constraints(*).

(*):What we’re modeling: a set of contracts between an Owner and a Property, each active over a closed-open time interval [start_date, end_date). There may be multiple contracts across time (trial → active → paused → active → ended), but no overlaps for the same pair.

Valid time: start_date … end_date (business reality window).

Transaction time: the existing created_at, updated_at (system recordkeeping; keept as is).

State: status ∈ {trial, active, paused, ended} with a small state machine.

This generalizes the current “unique pair” discussion (which led to duplicates/ambiguity) into a

proper temporal relation, addressing exactly the issues as documented around owner_property_map duplicates and ambiguous updates/deletes.

A commented SQL migration script

What it does

- Creates owner_property_contract (temporal table) with contract_id PK, owner_id, property_id, start_date, end_date (NULL=open), status (trial|active|paused|ended), reason, created_at, updated_at preserved for transaction-time
- Backfills from owner_property_map, deriving start_date := created_at, end_date := NULL when updated_at is empty/same as created (treated as open), status := 'active' for open; 'ended' otherwise
- Adds constraints/indexes: unique open interval per (owner_id, property_id), helper index (owner_id, property_id, start_date)
- Enforces invariants via triggers: well-formed interval (start < end), no overlapping intervals for the same pair (closed–open semantics)
- Compatibility view owner_property_map_compat: exposes the latest (open or most recent) contract per pair as the legacy shape, helps avoid breaking consumers during transition

```
-- Migration: owner_property_map -> owner_property_contract (temporal, non-overlapping intervals)
-- Goals
--  - Introduce time-bounded owner ↔ property contracts with a single surrogate PK.
--  - Preserve existing data; derive start_date/end_date/status from created_at/updated_at.
--  - Enforce invariants:
--      I1: No overlapping intervals per (owner_id, property_id)
--      I2: At most one open interval (end_date IS NULL) per pair
--      I3: start_date < end_date when end_date present
--  - Keep backward compatibility via a compatibility VIEW (optional).

BEGIN TRANSACTION;
PRAGMA foreign_keys=OFF;

-- 1) New table
CREATE TABLE IF NOT EXISTS owner_property_contract (
  contract_id   INTEGER PRIMARY KEY,
  owner_id      INTEGER NOT NULL,
  property_id   INTEGER NOT NULL,
  start_date    TEXT NOT NULL,  -- DD.MM.YYYY
  end_date      TEXT,           -- NULL = open interval
  status        TEXT NOT NULL CHECK (status IN ('trial','active','paused','ended')),
  reason        TEXT,
  created_at    TEXT NOT NULL,
  updated_at    TEXT NOT NULL,
  FOREIGN KEY (owner_id)    REFERENCES owner(owner_id)       ON DELETE CASCADE ON UPDATE CASCADE,
  FOREIGN KEY (property_id) REFERENCES property(property_id) ON DELETE CASCADE ON UPDATE CASCADE
```

```
);

-- 2) Indexes/constraints
CREATE UNIQUE INDEX IF NOT EXISTS uq_opc_open
  ON owner_property_contract(owner_id, property_id)
  WHERE end_date IS NULL;

CREATE INDEX IF NOT EXISTS idx_opc_pair_start
  ON owner_property_contract(owner_id, property_id, start_date);

-- 3) Backfill from owner_property_map
INSERT INTO owner_property_contract (
  contract_id, owner_id, property_id, start_date, end_date, status, reason, created_at, updated_at
)
SELECT
  map_id AS contract_id,
  owner_id,
  property_id,
  COALESCE(NULLIF(created_at, ''), strftime('%d.%m.%Y','now')) AS start_date,
  CASE
    WHEN updated_at IS NULL OR updated_at = '' OR updated_at = created_at THEN NULL
    ELSE updated_at
  END AS end_date,
  CASE
    WHEN updated_at IS NULL OR updated_at = '' OR updated_at = created_at THEN 'active'
    ELSE 'ended'
  END AS status,
  NULL AS reason,
  COALESCE(NULLIF(created_at, ''), strftime('%d.%m.%Y','now')) AS created_at,
  COALESCE(NULLIF(updated_at, ''), COALESCE(NULLIF(created_at, ''), strftime('%d.%m.
%Y','now'))) AS updated_at
FROM owner_property_map
WHERE NOT EXISTS (
  SELECT 1 FROM owner_property_contract c WHERE c.contract_id =
owner_property_map.map_id
);

-- 4) Triggers for interval invariants
DROP TRIGGER IF EXISTS trg_opc_interval_wellformed_ins;
CREATE TRIGGER trg_opc_interval_wellformed_ins
BEFORE INSERT ON owner_property_contract
FOR EACH ROW
WHEN NEW.end_date IS NOT NULL
 AND date(substr(NEW.start_date,7,4)||'-'||substr(NEW.start_date,4,2)||'-'||
substr(NEW.start_date,1,2))
    >= date(substr(NEW.end_date,7,4)||'-'||substr(NEW.end_date,4,2)||'-'||substr(NEW.end_date,1,2))
BEGIN
  SELECT RAISE(ABORT, 'invalid interval: start_date must be before end_date');
END;
```

```
DROP TRIGGER IF EXISTS trg_opc_interval_wellformed_upd;
CREATE TRIGGER trg_opc_interval_wellformed_upd
BEFORE UPDATE OF start_date, end_date ON owner_property_contract
FOR EACH ROW
WHEN NEW.end_date IS NOT NULL
 AND date(substr(NEW.start_date,7,4)||'-'||substr(NEW.start_date,4,2)||'-'||
substr(NEW.start_date,1,2))
   >= date(substr(NEW.end_date,7,4)||'-'||substr(NEW.end_date,4,2)||'-'||substr(NEW.end_date,1,2))
BEGIN
  SELECT RAISE(ABORT, 'invalid interval: start_date must be before end_date');
END;

DROP TRIGGER IF EXISTS trg_opc_no_overlap_ins;
CREATE TRIGGER trg_opc_no_overlap_ins
BEFORE INSERT ON owner_property_contract
FOR EACH ROW
WHEN EXISTS (
  SELECT 1
  FROM owner_property_contract x
  WHERE x.owner_id    = NEW.owner_id
    AND x.property_id = NEW.property_id
    AND (
      date(substr(x.start_date,7,4)||'-'||substr(x.start_date,4,2)||'-'||substr(x.start_date,1,2))
        <= COALESCE(date(substr(NEW.end_date,7,4)||'-'||substr(NEW.end_date,4,2)||'-'||
substr(NEW.end_date,1,2)), date('9999-12-31'))
      AND
      date(substr(NEW.start_date,7,4)||'-'||substr(NEW.start_date,4,2)||'-'||substr(NEW.start_date,1,2))
        <  COALESCE(date(substr(x.end_date,7,4)||'-'||substr(x.end_date,4,2)||'-'||
substr(x.end_date,1,2)), date('9999-12-31'))
    )
)
BEGIN
  SELECT RAISE(ABORT, 'overlapping owner-property contract interval');
END;

DROP TRIGGER IF EXISTS trg_opc_no_overlap_upd;
CREATE TRIGGER trg_opc_no_overlap_upd
BEFORE UPDATE OF start_date, end_date, owner_id, property_id ON owner_property_contract
FOR EACH ROW
WHEN EXISTS (
  SELECT 1
  FROM owner_property_contract x
  WHERE x.owner_id    = NEW.owner_id
    AND x.property_id = NEW.property_id
    AND x.contract_id <> OLD.contract_id
    AND (
      date(substr(x.start_date,7,4)||'-'||substr(x.start_date,4,2)||'-'||substr(x.start_date,1,2))
        <= COALESCE(date(substr(NEW.end_date,7,4)||'-'||substr(NEW.end_date,4,2)||'-'||
substr(NEW.end_date,1,2)), date('9999-12-31'))
      AND
```

```
    date(substr(NEW.start_date,7,4)||'-'||substr(NEW.start_date,4,2)||'-'||substr(NEW.start_date,1,2))
      <  COALESCE(date(substr(x.end_date,7,4)||'-'||substr(x.end_date,4,2)||'-'||
substr(x.end_date,1,2)), date('9999-12-31'))
   )
)
BEGIN
  SELECT RAISE(ABORT, 'overlapping owner-property contract interval');
END;

-- 5) Compatibility view (mimic old owner_property_map by showing latest contract per pair)
DROP VIEW IF EXISTS owner_property_map_compat;
CREATE VIEW owner_property_map_compat AS
WITH ranked AS (
  SELECT
    c.contract_id AS map_id,
    c.owner_id, c.property_id,
    c.start_date AS created_at,
    COALESCE(c.end_date, c.updated_at) AS updated_at,
    ROW_NUMBER() OVER (
      PARTITION BY c.owner_id, c.property_id
      ORDER BY
        CASE WHEN c.end_date IS NULL THEN 1 ELSE 0 END DESC,
        date(substr(c.start_date,7,4)||'-'||substr(c.start_date,4,2)||'-'||substr(c.start_date,1,2)) DESC
    ) AS rn
  FROM owner_property_contract c
)
SELECT map_id, owner_id, property_id, created_at, updated_at
FROM ranked
WHERE rn = 1;

PRAGMA foreign_keys=ON;
COMMIT;
```

8.11 Self-Assessment Summary

Introduction

In order to demonstrate critical reflection, the following matrix evaluates the project deliverables against the assignment criteria. This is a candidate’s perspective, not a substitute for the examiner’s evaluation. The intention is to make strengths and shortcomings explicit, to show that the identified limitations are acknowledged, and to highlight the documented roadmap for future improvements (see Future Work).

The scores shown are approximate self-ratings, based on the official rubric, and are supported by evidence in the thesis, codebase, and documentation. In areas marked lower, mitigation strategies and upgrade plans are already documented (e.g. normalization of tables, temporal owner–property contracts, DTO migration).

Assessment Matrix

Criterion	Weight	Self-Rating	Reflection
Domain model & Database	15%	9 / 10	Strong schema with triggers and validators; issues in owner_property_map are explicitly acknowledged and a temporal redesign is documented as future work.
Repositories / DTO / Controllers	25%	8.5 / 10	CRUDBase and route factory work well; validators double as DTOs. The associated risk of drift is noted, and a migration to DTO-driven OpenAPI is planned.
Authentication & Security	10%	8.5 / 10	JWT with HttpOnly cookies, refresh, CSRF headers, and roles. Adequate for an MVP. More advanced features (rotation, anomaly detection) reserved for future iterations.
Frontend	10%	8 / 10	Functional Bootstrap UI that covers CRUD and auth flows. Minimal styling/UX; sufficient for requirements but leaves room for future polish.
Docs & OpenAPI	15%	8.5 / 10	Endpoints, error envelopes, and date formats documented; spec aligned with code. Drift tests and DTO auto-generation are planned enhancements.
Testing	15%	8.5 / 10	Unit, CRUD, validator, and black-box tests present. CI coverage established. Role-matrix tests and contract drift tests remain in roadmap.
Use of LLMs	10%	9 / 10	LLM use documented with methodology, provenance, and human oversight. Provides transparency and academic integrity.