

I will provide raw data, notes, or my understanding. Your task is NOT to fix grammar — only to correct logical mistakes, fill missing reasoning, restructure ideas for clarity, and expand the content with deeper insights. Preserve my intent but make the explanation accurate, complete, and logically strong.

After improving it, also provide a short summary of the key corrected points.

Ask questions if something is unclear.

Wait for my input.

Got it — I'll wait for your input.



Smart Ticketing System:
system to manage ticketing inside the organization or industry with the incident management and tracking system.
in this system we are managing tickets with some data. first have a look on structure first.

system is having:
organizations: (organization registration and login){ organization will have branches(for multiple locations), departments(specifically for branches), roles(inside the department), employee(employee will be created for specific role).

category and sub_categories for ticketing:
ticketing management will have categories(like hardware, software, incident or any other) and sub category (like mouse, keyboard or else).

along with all this data the ticket will be created.
now the ticket is being created for department like when employee is raising ticket then it should be raised for department or individually. (in database we have to maintain the state as DEPARTMENT/INDIVIDUALLY). we

have to maintain the reference data accordingly.
ticket is having base data like:
id, subject, description, attachments, category & sub
category

in reference data the ticket shold have:
from raised{
userid/employee id
dep id
branch id
more needy references
}

and to raised{
department/individually (status)
raised department id/raised employee id
option to raise copy(RC) which will share this ticket's copy
with another person too
option to raise hidden copy(RHC) which will share this
ticket copy with the person but in hidden format.

along with all this data we have to maintain the states:
RAISED | PENDING | WORKING_ON_IT | CLOSED |
REJECTED | APPROVED | DELIVERED | and more needy
status.

}

Below is a **restructured, clarified, and logically complete** version of your description.
I preserved your intent but corrected gaps, inconsistencies, and reasoning issues, and
added deeper insights.

After that, you'll find a **summary of key corrected points**.

If anything needs adjustment, tell me.

Improved and Structured Explanation

Smart Ticketing System Overview

The Smart Ticketing System is designed to manage incidents and service requests within
an organization or across multiple organizations. It combines structured organizational

data with a robust ticket lifecycle, enabling controlled ticket assignment, tracking, and resolution.

1. Organizational Structure

Each registered organization contains the following hierarchical components:

1.1 Organization → Branches

- An organization can operate in multiple geographic or functional locations.
- Each branch acts as an independent unit for ticket routing and resource assignment.

1.2 Branches → Departments

- Every branch contains several departments (e.g., IT, HR, Finance).
- Tickets can be directed to a specific department based on the nature of the issue.

1.3 Departments → Roles

- Departments define roles (e.g., Support Engineer, Manager, Technician).
- Roles determine access permissions and responsibility levels.

1.4 Roles → Employees

- Employees are created under a specific role.
- Each employee inherits role- and department-based access privileges.

This hierarchical structure ensures that every ticket can be traced to a specific organizational context (organization → branch → department → employee).

2. Ticket Categorization

Ticket classification is essential for routing and reporting:

2.1 Categories

High-level groups indicating the type of issue, e.g.:

- Hardware
- Software
- Incident
- Maintenance
- Access Management

2.2 Sub-categories

More granular details under each category, e.g.:

- Hardware → Mouse, Keyboard, Printer
- Software → OS Issue, Application Bug

These rules help automate ticket assignment and analytics.

3. Ticket Creation Model

When a ticket is created, two sets of data must be captured:

3.1 Base Ticket Data

This is the essential information common to all tickets:

- **Ticket ID**
 - **Subject**
 - **Description**
 - **Attachments**
 - **Category**
 - **Sub-category**
-

3.2 Reference Data

Reference data defines *who raised the ticket, who it is assigned to, and how it flows through the organization.*

3.2.1 “From” (Raised By) Data

These fields identify the source of the ticket:

- **Employee ID** (who raised it)
- **Department ID**
- **Branch ID**
- Any additional metadata (e.g., device info, priority, requester contact details)

This ensures proper auditability.

3.2.2 “To” (Assigned To) Data

This determines how the ticket is routed:

Assignment Mode

- **DEPARTMENT** – ticket goes to a department queue
- **INDIVIDUAL** – ticket is assigned to a specific employee

Assignment References

- If DEPARTMENT → **Assigned Department ID**
- If INDIVIDUAL → **Assigned Employee ID**

Copying Options

- **RC (Raise Copy)** – sends a visible copy to another employee
- **RHC (Raise Hidden Copy)** – sends a hidden/invisible copy to selected individuals (similar to BCC in email)

These options help notify concerned personnel without altering the primary assignee.

4. Ticket Lifecycle / Status Management

Every ticket must follow a controlled lifecycle. Standard states may include:

- **RAISED** – Ticket is created but not yet processed
- **PENDING** – Awaiting action or approvals
- **WORKING_ON_IT** – Assigned team/employee is actively addressing the issue
- **APPROVED** – Request is approved (if approval workflow is enabled)
- **REJECTED** – Ticket or request is declined
- **DELIVERED** – Solution or service has been delivered
- **CLOSED** – Successfully resolved and validated
- **Additional states** as needed (ON_HOLD, ESCALATED, REOPENED, etc.)

A clear status model allows automated workflows, escalation, reporting, and SLA management.

Summary of Key Improvements

1. Structured your content logically: Organization → Categories → Ticket → Lifecycle.
2. Clarified relationships: Org → Branch → Department → Role → Employee.
3. Fixed conceptual gaps: Explained why each hierarchy layer is needed.
4. Defined assignment modes clearly: DEPARTMENT vs. INDIVIDUAL.
5. Corrected copy logic: RC (visible) and RHC (hidden).
6. Added lifecycle insights: Why statuses exist and how they support workflows.
7. Expanded reasoning: How categorization and references help routing and reporting.

If you want, I can also help you:

- convert this into a technical specification,
- design the database schema,
- design APIs,
- create workflow diagrams,
- or prepare documentation for developers or management.

Would you like the next step?



Smart Ticketing System:

system to manage ticketing inside the organization or industry with the incident management and tracking system.

in this system we are managing tickets with some data. first have a look on structure first.

system is having:

organizations: (organization registration and login){ organization will have branches(for multiple locations), departments(specifically for branches), roles(inside the department), employee(employee will be created for specific role).

category and sub_categories for ticketing:

ticketing management will have categories(like hardware, software, incident or any other) and sub category (like mouse, keyboard or else).

along with all this data the ticket will be created.

now the ticket is being created for department like when employee is raising ticket then it should be raised for department or individually. (in database we have to maintain the state as DEPARTMENT/INDIVIDUALLY). we have to maintain the reference data accordingly.

ticket is having base data like:

id, subject, description, attachments, category & sub category

in reference data the ticket should have:

```
from raised{  
    userid/employee id  
    dep id  
    branch id  
    more needy references  
}
```

and to raised{

```
    department/individually (status)  
    raised department id/raised employee id  
    option to raise copy(RC) which will share this ticket's copy  
    with another person too  
    option to raise hidden copy(RHC) which will share this  
    ticket copy with the person but in hidden format.
```

along with all this data we have to maintain the states:

```
RAISED | PENDING | WORKING_ON_IT | CLOSED |  
REJECTED | APPROVED | DELIVERED | and more needy  
status.
```

yet this is all about the ticket we have to maintain the ticket state as RETURNED | RELEASED | REPLIED to whom the ticket is raised can reply to the ticket, at the time it will go in replied/ returned. when ticket is raised then the both nodes can have communication with each other until the ticket is closed.

if the ticket is closed/ cancelled/ rejected then the feedback should be raised

```
}
```

Below is a **clean, logically consistent, expanded version** of your updated description. I preserved your meaning but fixed reasoning gaps, clarified workflows, and made the structure internally consistent.

A short summary of corrections is included at the end.

If anything should be interpreted differently, just tell me.

Improved and Structured Explanation

1. Smart Ticketing System Overview

The Smart Ticketing System is an internal incident and service request management platform. It is designed to help organizations track issues, route them to the correct department or individual, maintain clear communication, record ticket states, and collect feedback after closure.

The system relies on a well-defined organizational structure and a detailed ticket lifecycle.

2. Organizational Structure

Each organization that registers in the system will maintain the following hierarchy:

2.1 Organization → Branches

- Organizations may operate in multiple locations.
- Each location is represented as a **branch**.

2.2 Branches → Departments

- Each branch contains departments (e.g., IT, HR, Admin).
- A ticket can be routed to a specific department.

2.3 Departments → Roles

- Departments define roles (e.g., Technician, Support Engineer, Manager).
- Roles are used to define permissions and responsibilities.

2.4 Roles → Employees

- Employees are created within a specific role.
 - Every employee belongs to one department and branch.
 - This determines who can raise or respond to tickets.
-

3. Ticket Categorization Structure

The ticket system uses classification to help route and analyze issues:

3.1 Categories

Examples:

- Hardware

- Software
- Incident
- Maintenance

3.2 Sub-categories

Examples:

- Hardware → Mouse, Keyboard, Monitor
- Software → Application Issue, OS Problem

Categories drive assignment rules and reporting.

4. Ticket Creation Model

Tickets consist of two types of data: **base ticket information** and **reference information**.

4.1 Base Ticket Information

This is the mandatory information for every ticket:

- Ticket ID
 - Subject
 - Description
 - Attachments
 - Category
 - Sub-category
-

4.2 Reference (Relational) Information

Reference data defines **who raised the ticket**, **who it is assigned to**, and **extra routing options**.

4.2.1 From (Raised By)

Captures the origin of the ticket:

- Employee ID / User ID
- Department ID
- Branch ID
- Any additional metadata (device details, priority, contact information)

This ensures the ticket can always be traced back to the requester.

4.2.2 To (Assigned To)

The ticket can be raised to a department or to an individual employee.

This must be stored as a state: **DEPARTMENT / INDIVIDUAL**.

Assignment Data

- If raised to a department → store **Assigned Department ID**
- If raised to an individual → store **Assigned Employee ID**

Copy Options

- **RC (Raise Copy)**
Sends a visible copy of the ticket to another user.
- **RHC (Raise Hidden Copy)**
Sends a hidden copy (similar to BCC).

This is useful for supervisors or stakeholders who need visibility without being part of the main workflow.

5. Ticket Lifecycle and Status Management

The system must support both **major workflow states** and **interaction-related states**.

5.1 Primary Workflow States

These represent the ticket's lifecycle:

- **RAISED** — Ticket created
- **PENDING** — Waiting for response, approval, or action
- **WORKING_ON_IT** — Assigned user/department is actively working
- **APPROVED** — If approval workflow is involved
- **REJECTED** — Request or ticket is rejected
- **DELIVERED** — Work has been completed or service delivered
- **CLOSED** — Ticket resolution finalized and accepted

Additional states (ON_HOLD, ESCALATED, REOPENED, etc.) can be added as needed.

5.2 Interaction-Based States

These states reflect communication between parties:

- **REPLIED** — The assigned party or requester has replied
- **RETURNED** — The ticket has been sent back to the requester (e.g., for more information or wrong routing)
- **RELEASED** — Ownership or assignment has been released (e.g., moved back to the department queue)

These states are used to track conversational or routing transitions within the ticket lifecycle.

6. Communication Flow Between Nodes

Once a ticket is raised:

- Both sides (**requester** and **assigned party**) can communicate.
- Messages, comments, and updates can be exchanged.
- Communication continues until the ticket is **closed**.

If the assigned party returns the ticket (RETURNED), the requester may provide clarification, causing it to move back into a working/active state.

7. Feedback Process

Feedback is required when a ticket reaches one of the end states:

- **Closed**
- **Cancelled**
- **Rejected**

The feedback mechanism allows the requester to rate the service, comment on the resolution, and help improve the system.

Summary of Key Improvements

1. Clarified organizational hierarchy and its purpose.
2. Separated base ticket data from reference data for cleaner logic.
3. Defined **DEPARTMENT** vs. **INDIVIDUAL** assignment clearly.
4. Added clear meaning for **RC** vs. **RHC** options.
5. Introduced two types of states:

- Primary workflow states
- Interaction-based states (REPLIED, RETURNED, RELEASED)

6. Explained ticket communication flow logically.
 7. Added a structured feedback process after ticket closure or cancellation.
-

If you want, I can also help you next with:

- database schema design (tables + relationships)
- API design (REST/GraphQL)
- workflow diagrams
- sequence diagrams
- UI flow for ticket creation and assignment

Would you like any of these?



in this system where you have maintained the status whether the ticket is raised for department or individually

In your system, the place to maintain whether a ticket is raised for a department or for an individual should be inside the “To (Assigned To)” reference data as a mandatory field.

Here is the clean and correct way to store it:

Where to Store the Assignment Type (DEPARTMENT / INDIVIDUAL)

Field Name (recommended):

assignment_type

Possible Values:

- "DEPARTMENT"
- "INDIVIDUAL"

Why?

Because this field determines **how the ticket is routed and which reference field to use** (Department ID or Employee ID).

✓ How to Structure It

To (Assigned To) Reference Block

cpp

 Copy code

```
to_assigned {  
    assignment_type: "DEPARTMENT" | "INDIVIDUAL"  
  
    assigned_department_id: (if assignment_type == "DEPARTMENT")  
    assigned_employee_id: (if assignment_type == "INDIVIDUAL")  
  
    raise_copy: [list of employee_ids] // RC  
    raise_hidden_copy: [list of employee_ids] // RHC  
}
```

✓ Explanation

When a ticket is raised to a department:

ini

 Copy code

```
assignment_type = "DEPARTMENT"  
assigned_department_id = <department_id>  
assigned_employee_id = null
```

When a ticket is raised to an individual:

ini

 Copy code

```
assignment_type = "INDIVIDUAL"  
assigned_employee_id = <employee_id>  
assigned_department_id = null
```

⚠ Why not mix the status with the main ticket status?

Because:

- Main ticket statuses: RAISED, WORKING_ON_IT, CLOSED, REJECTED, etc.
These represent the **lifecycle** of the ticket.
- Assignment type (DEPT/INDIVIDUAL) represents a **routing decision** and should not interfere with workflow statuses.

Keeping them separate avoids confusion and makes the system scalable.

📌 Summary (Short Answer)

You should maintain whether a ticket is raised for a department or for an individual in the “**To (Assigned To)**” reference section, using a field like:

python

 Copy code

```
assignment_type: "DEPARTMENT" or "INDIVIDUAL"
```

This field controls which reference (department_id or employee_id) is used.

If you want, I can also show you the **database table design** for this part.



Ticket Creation Model:

Ticket Base:

ticket id, subject, description, attachments, category, sub category, assignment type(DEPARTMENT/INDIVIDUAL)

Ticket references:

ticket references from:

employee id/user id, department id, branch id, (device details)

ticket references to:

if raised to dept: ASSIGNED DEPARTMENT ID

if raised to employee: ASSIGNED EMPLOYEE ID

branch id, organization id

Ticket Status management:
RAISED, PENDING, WORKING_ON_IT, APPROVED,
REJECTED, DILIVERED, CLOSED, RETURNED, RELEASED,
TO_HIGHER_AUTHORITY, ON_HOLD, RESPONDED,
ESCALATED.

now you have this ticket information, create a proper schema in object format

Below is a **clean, consistent, and production-ready object schema** based on the ticket structure you provided.

I resolved ambiguities, normalized fields, and added structure where necessary while keeping your intent fully intact.

If you want, I can also generate **database tables** (SQL), **MongoDB schemas**, or **API contract format** after this.

Smart Ticket Object Schema (Recommended Structure)

json

 Copy code

```
{  
  "ticket_id": "string / uuid",  
  
  "base": {  
    "subject": "string",  
    "description": "string",  
    "attachments": [  
      {  
        "file_id": "string",  
        "file_name": "string",  
        "file_url": "string"  
      }  
    ],  
    "category_id": "string",  
    "sub_category_id": "string",  
  
    "assignment_type": "DEPARTMENT | INDIVIDUAL"  
  },  
}
```

```
"references": {  
    "from": {  
        "employee_id": "string",  
        "department_id": "string",  
        "branch_id": "string",  
        "device_details": {  
            "ip_address": "string",  
            "device_type": "string",  
            "os": "string",  
            "browser": "string",  
            "additional_info": "string"  
        }  
    },  
  
    "to": {  
        "assigned_department_id": "string | null",  
        "assigned_employee_id": "string | null",  
        "branch_id": "string",  
        "organization_id": "string"  
    }  
},  
  
"status": {  
    "current_status": "RAISED | PENDING | WORKING_ON_IT | APPROVED | REJECTED | D  
    "status_history": [  
        {  
            "status": "string",  
            "changed_by": "employee_id",  
            "changed_at": "timestamp",  
            "remarks": "string"  
        }  
    ]  
},  
  
"copy_options": {  
    "raise_copy": [  
        "employee_id"  
    ],  
    "raise_hidden_copy": [  
        "employee_id"  
    ]  
},
```

```
"timestamps": {  
    "created_at": "timestamp",  
    "updated_at": "timestamp",  
    "closed_at": "timestamp | null"  
}  
}
```

✓ Explanation of Schema Structure

1. Ticket Base

Holds core ticket information:

- subject, description, category, sub-category
- assignment_type = DEPARTMENT / INDIVIDUAL

Assignment type controls which reference field is used in the "to" block.

2. References Section

From

Details of who raised the ticket:

- employee_id
- department_id
- branch_id
- device_details (optional metadata)

To

Details of where the ticket is assigned:

- assigned_department_id (if DEPARTMENT)
- assigned_employee_id (if INDIVIDUAL)
- branch_id (target branch)
- organization_id (target org)

3. Status Section

current_status

Stores the active state of the ticket.

status_history

Tracks all transitions (useful for auditing and SLA calculations).

4. Copy Options

Stores visible and hidden copies:

- `raise_copy` (RC)
 - `raise_hidden_copy` (RHC)
-

5. Timestamps

Standard tracking for lifecycle events.

★ If you want next:

I can produce:

- SQL table schema (normalized)
- NoSQL (Mongo) schema
- API request/response models
- Workflow diagrams
- State machine design

Would you like the **database schema** next?



how this data can be managed in database. can you suggest

Below is a **clear and practical guide** for how this ticketing system can be stored in a database.

I'll give you two recommended approaches:

- 1. Relational Database Model (SQL) — Best for enterprises, strict structure, reporting, SLA management**

2. Document Database Model (MongoDB) — Best for dynamic structures and flexible ticket metadata

You can choose whichever fits your system.

1. RELATIONAL DATABASE MODEL (SQL – Recommended for Enterprise)

Relational DBs handle hierarchical org structures, permissions, ticket audit trails, and reporting very well.

A. Organizational Structure Tables

organizations

field	type
id (PK)	uuid
name	varchar

branches

field	type
id (PK)	uuid
organization_id (FK)	uuid
name	varchar

departments

field	type
id (PK)	uuid
branch_id (FK)	uuid
name	varchar

roles

field	type
id (PK)	uuid
department_id (FK)	uuid
name	varchar

employees

field	type
id (PK)	uuid
role_id (FK)	uuid
department_id (FK)	uuid
branch_id (FK)	uuid
name	varchar

B. Category Structure**categories**

id name

sub_categories

id category_id name

C. Ticket Tables**tickets (MAIN TABLE)**

field	type	notes
id (PK)	uuid	ticket id
subject	text	

field	type	notes
description	text	
category_id (FK)	uuid	
sub_category_id (FK)	uuid	
assignment_type	enum(DEPARTMENT, INDIVIDUAL)	controls routing
created_at	timestamp	
updated_at	timestamp	
closed_at	timestamp	nullable

ticket_references_from

Stores who raised the ticket.

field	type
id (PK)	uuid
ticket_id (FK)	uuid
employee_id (FK)	uuid
department_id (FK)	uuid
branch_id (FK)	uuid
device_ip	varchar
device_type	varchar

ticket_references_to

Stores assignment details.

field	type
id (PK)	uuid
ticket_id (FK)	uuid
assigned_department_id (nullable)	uuid
assigned_employee_id (nullable)	uuid
branch_id	uuid
organization_id	uuid

Only one of `assigned_department_id` OR `assigned_employee_id` is filled based on `assignment_type`.

ticket_status_history

Tracks all state transitions.

field	type
id (PK)	uuid
ticket_id (FK)	uuid
status	enum
changed_by	employee_id
changed_at	timestamp
remarks	text

ticket_copy_options

Holds RC & RHC entries.

field	type
id (PK)	uuid

field	type
ticket_id	uuid
employee_id	uuid
type	enum(RC, RHC)

ticket_attachments

| id | ticket_id | file_name | file_url |

This SQL design is **fully normalized**, easy to scale, great for analytics, and suitable for large organizations.

2. DOCUMENT DATABASE MODEL (MongoDB – Recommended for Fast Development)

Store each ticket as a **single document** with embedded sub-objects.

Example:

json

 Copy code

```
{
  "_id": "uuid",
  "subject": "...",
  "description": "...",
  "category_id": "uuid",
  "sub_category_id": "uuid",
  "assignment_type": "DEPARTMENT",

  "from": {
    "employee_id": "uuid",
    "department_id": "uuid",
    "branch_id": "uuid",
    "device_details": {}
  },
}
```

```
"to": {  
    "assigned_department_id": "uuid",  
    "assigned_employee_id": null,  
    "branch_id": "uuid",  
    "organization_id": "uuid"  
},  
  
"attachments": [],  
"copy_options": {  
    "rc": [],  
    "rhc": []  
},  
  
"status": {  
    "current": "PENDING",  
    "history": [  
        {  
            "status": "RAISED",  
            "changed_by": "uuid",  
            "changed_at": "timestamp"  
        }  
    ]  
},  
  
"timestamps": {  
    "created_at": "timestamp",  
    "updated_at": "timestamp",  
    "closed_at": null  
}  
}
```

MongoDB is great if:

- You want flexibility
- Fields may evolve over time
- You prefer embedding over joins
- You want fast development
- Real-time updates & nested structures
- Less strict reporting needs

★ WHICH DATABASE SHOULD YOU USE?

Requirement	Recommendation
Enterprise-level system	SQL (PostgreSQL / MySQL)
Strict reporting, SLAs, audits	SQL
Flexible structure, rapid development	MongoDB
Large organization hierarchy	SQL
Dynamic ticket metadata	MongoDB
Need ACID & joins	SQL

⭐ If you want next:

I can generate:

- Full SQL schema (DDL scripts) for all tables
- Optimized indexing strategy
- ER diagram
- MongoDB collection design
- API designs based on this schema

Would you like the **SQL DDL scripts or diagram** next?



if he replied/ returned/ responded then the from person can send the message and there shoud be communication in between both. find the fields or data to handle this communication

Got it! For handling **two-way communication between ticket participants** (requester and assignee), we need a **dedicated structure** that supports messages, replies, and audit tracking. I'll outline the **fields and data structure** needed.

This is **independent of the main ticket status** but linked to the ticket.

1. Communication Table / Object

Each ticket can have multiple messages exchanged between "from" and "to" participants.

Fields / Data Needed

Field	Type	Description
id	uuid	Unique identifier for the message
ticket_id	uuid	Link to the parent ticket
sender_id	uuid	Employee ID of the sender
receiver_ids	array[uuid]	Employee(s) receiving the message (could be multiple for RC/RHC)
message_text	text	Message content
attachments	array	Optional attachments for this message
message_type	enum("NORMAL","RC","RHC")	Type of message; matches copy options
parent_message_id	uuid (nullable)	If this is a reply to another message
status	enum("SENT", "READ", "RETURNED", "RESPONDED")	Current state of the message
created_at	timestamp	When the message was sent
updated_at	timestamp	Last update on this message
hidden	boolean	True if the message is hidden (RHC)

2. How This Handles Ticket Communication

1. Two-way messaging

- Sender can be **from** (requester) or **to** (assignee)
- Receiver(s) determined by assignment + RC/RHC

2. Replies / Threading

- `parent_message_id` allows threading and conversation flow.
- Can track multiple responses and replies.

3. Status tracking

- A message can be marked as `READ` by the receiver(s).
- If a ticket is `RETURNED`, that action can be logged in the message as `status=RETURNED`.
- Similarly, `RESPONDED` tracks that the message has been answered.

4. Copy options

- RC and RHC messages are included in the same table.
- Hidden messages (`hidden=true`) are visible only to selected users.

5. Audit / History

- All messages are timestamped and linked to the ticket and sender.
- Full audit trail of communication between requester and assignee.

3. Optional Enhancements

- **Read receipts:** Track who has read each message (`read_by: array[employee_id]`)
- **Priority flag:** Mark urgent messages
- **Inline comments / annotations** on attachments

4. Example Object (JSON)

```
json
```

 Copy code

```
{  
  "id": "msg_123",  
  "ticket_id": "ticket_456",  
  "sender_id": "emp_001",  
  "receiver_ids": ["emp_002", "emp_003"],  
  "message_text": "Please provide additional details for the ticket.",  
  "attachments": [  
    {"file_name": "screenshot.png", "file_url": "..."}  
  ],  
  "message_type": "NORMAL",  
  "parent_message_id": null,  
  "status": "SENT",  
  "hidden": false,  
  "created_at": "2025-12-12T10:00:00Z",  
  "read_by": []  
}
```

```
"updated_at": "2025-12-12T10:00:00Z"  
}
```

Summary

- Use a separate `ticket_messages` table / collection linked to tickets.
- Include sender, receiver, message content, attachments, type, and status.
- Support threading via `parent_message_id`.
- Support RC / RHC (visible/hidden copy).
- Maintain a full audit trail for replies, returns, and responses.

If you want, I can **combine this with your ticket schema** and show a **full relational or MongoDB schema** including **tickets + references + status + messages** so it's ready for implementation.

Do you want me to do that?

