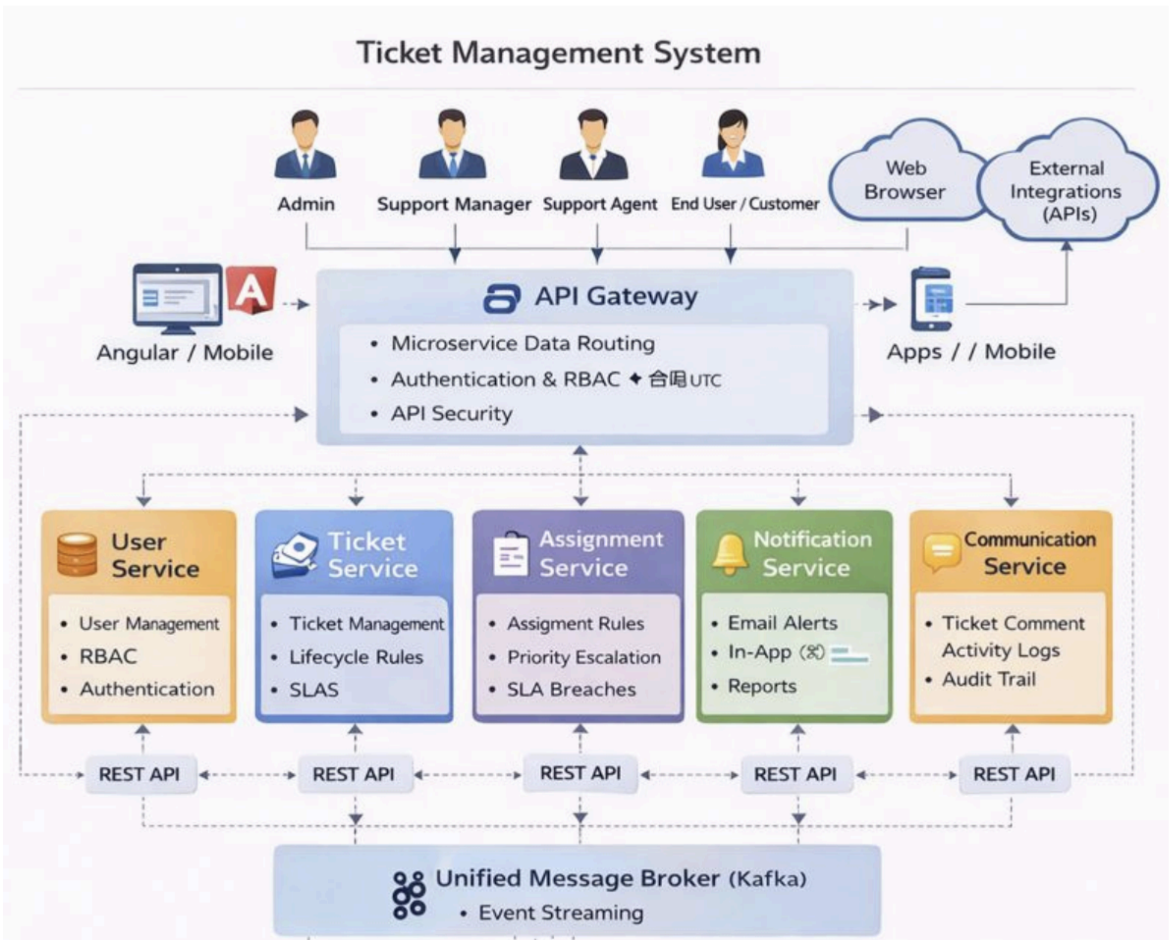
SOFTWARE DESIGN DOCUMENT(SDD)

Smart Ticket & Issue Management System



---

1. DOCUMENT CONTROL

Item	Details
Project Name	Smart Ticket & Issue Management System
Version	1.0
Author	Varakantham Pranitha
Date	27-12-2025
Status	Final

2. PURPOSE OF THE DOCUMENT

This document describes the system architecture, design decisions, component structure, APIs, data models, and non-functional aspects of the Smart Ticket & Issue Management System.

It is intended for:

- Developers
- Reviewers
- Interview discussions
- Maintenance & enhancement planning

3. SYSTEM OVERVIEW

Business Objective

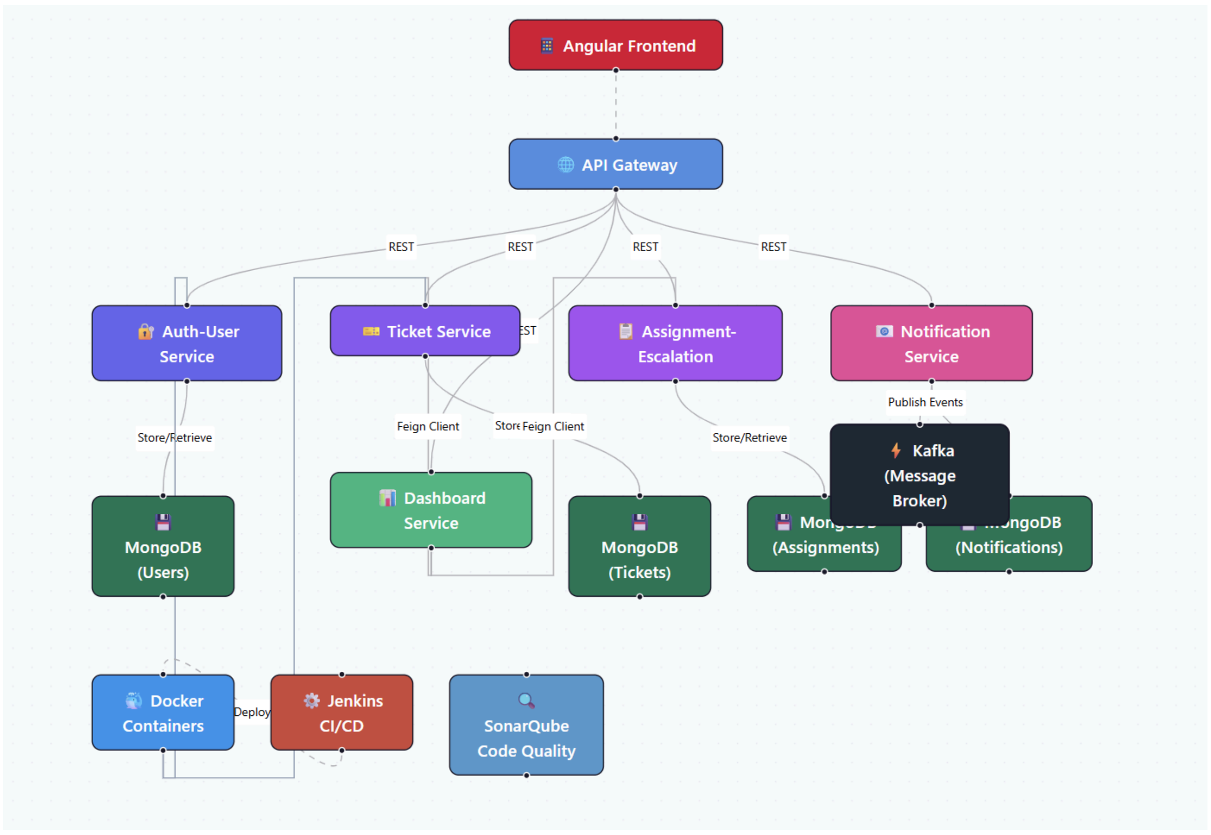
Provide a scalable, secure, and maintainable system to:

- Manage tickets (creation, update, resolution, closure, escalation)
- Assign tickets manually or automatically to agents
- Enforce SLA rules and escalation policies
- Provide dashboards for Admins, Managers, Agents, and End-Users

High-Level Features

- User Registration & Secure Login
- Real-time Ticket creation & tracking
- Manual and auto assignment
- SLA enforcement & escalation
- Role-based access (USER / ADMIN / MANAGER / AGENT)

4. ARCHITECTURE OVERVIEW (HLD)



Architecture Style

- Microservices Architecture
- REST-based communication
- Containerized deployment

Core Components

1. Angular Frontend
2. API Gateway (optional)
3. Auth-User Service
4. Ticket Service
5. Assignment-Escalation Service
6. Notification Service (Kafka + SMTP)
7. Dashboard Service (aggregates stats via Feign)
8. MongoDB (per service)
9. CI/CD Pipeline

5. TECHNOLOGY STACK

Backend

- Java 17
- Spring Boot
- Spring WebFlux (reactive)
- Spring Data MongoDB
- Spring Validation

Frontend

- Angular
- Bootstrap / Angular Material
- RxJS
- Chart.js (via ng2-charts) for dashboard visualizations

Database

- MongoDB (one DB per microservice)

DevOps

- Docker
- Docker Compose
- Jenkins
- SonarQube

Testing

- JUnit 5
- Mockito
- Spring Boot Test

6. MICROSERVICES DESIGN

6.1 Auth-User Service

Responsibilities

- User registration & authentication
- Role management (Admin, Manager, Agent, User)
- Password reset/change flows

APIs

```
POST   /auth/register -> Register new user
POST   /auth/login -> Authenticate User
POST   /auth/change-password -> Change password with old/new
POST   /auth/reset-password -> Reset password with token
POST   /auth/request-reset -> Request Password Reset
PUT    /users/{id} -> Update user details
GET    /users -> List all users
GET    /users/{id} -> Get user by ID
GET    /auth/{id} -> Fetch user email by ID
DELETE /auth/{id} -> Delete user by ID (Soft delete)
```

Database

- users collection

6.2 Ticket Service

Responsibilities

- Ticket CRUD operations
- Lifecycle Management, Status transitions
  (OPEN → IN_PROGRESS → RESOLVED → CLOSED → REOPENED)
- Reopen/Cancel
- Category management
- Ticket activity timeline (comments, actions, SLA reports)

Ticket APIs

```
POST   /tickets/create -> Create Ticket
PUT    /tickets/{id} -> Update Ticket
PUT    /tickets/{id}/close -> Close Ticket
PUT    /tickets/{id}/resolve -> Resolve Ticket
PUT    /tickets/{id}/reopen -> Reopen Ticket
GET    /tickets/{id} -> Get Tickets by ID
GET    /tickets/user/{userId} -> Get Tickets by User ID
GET    /tickets/recent -> Get recent tickets
GET    /tickets/recent/{userId} -> Get recent tickets by user ID
```

```
DELETE /tickets/{id} -> Delete ticket by ID
```

Ticket Activity APIs

```
POST /tickets/{ticketId}/activity/comment -> Post the ticket activity by ID
GET  /tickets/{ticketId}/activity/ -> Get ticket activity by ID
```

Category Activity APIs

```
POST   /categories/create -> Create a new category
PUT    /categories/{id} -> Update category by ID
GET    /categories -> Get all categories
GET    /categories -> Get category by ID
DELETE /categories/{id} -> Delete category by ID
```

Database

- tickets collection
- ticket_activities collection
- categories collection

6.3 Assignment-Escalation Service

Responsibilities

- Manual assignment of Tickets
- Auto assignment (least workload → hybrid skill+workload)
- SLA Rules enforcement & escalation checks
- Reassignment logic

Assignment-Escalation APIs

```
POST  /assignments/manual -> Manual assignment
POST  /assignments/{ticketId}/auto -> Auto assignment
PUT   /assignments/{ticketId}/complete -> Complete assignment
PUT   /assignments/{ticketId}/check-escalation -> Manual assignment
```

Sla-Rules APIs

```
POST /sla-rules -> Create custom SLA_RULE
PUT  /sla-rules/{id} -> Update SLA_RULE
GET  /sla-rules -> Get all SLA_RULEs
DELETE  /sla-rules/{id} -> Delete SLA_RULE
```

Database

- assignment collection
- sla_rules collection

6.4 Notification Service

Responsibilities

- Send email notifications (assignment, escalation, resolution etc)
- Kafka event publishing for async communication
- Notification history per user

APIs

```
POST /notifications/send -> Send Notification
GET  /notifications/status/{id} -> Get notification status
GET  /notifications/history/{userEmail} -> Get notification history
```

Database

- notifications collection

6.5 Dashboard Service

Responsibilities

- Aggregate stats from Ticket & Assignment services via Feign clients
- Provide unified dashboard endpoints for Admin/Manager views

APIs

```
GET  /dashboard/tickets/status-summary -> Ticket status distribution
GET  /dashboard/tickets/status-priority-summary -> Ticket priority
distribution
GET  /dashboard/category-summary -> Ticket category distribution
GET  /dashboard/assignments/agent-summary -> Agent workload summary
GET  /dashboard/assignments/escalation-summary -> Escalation summary
```

Database

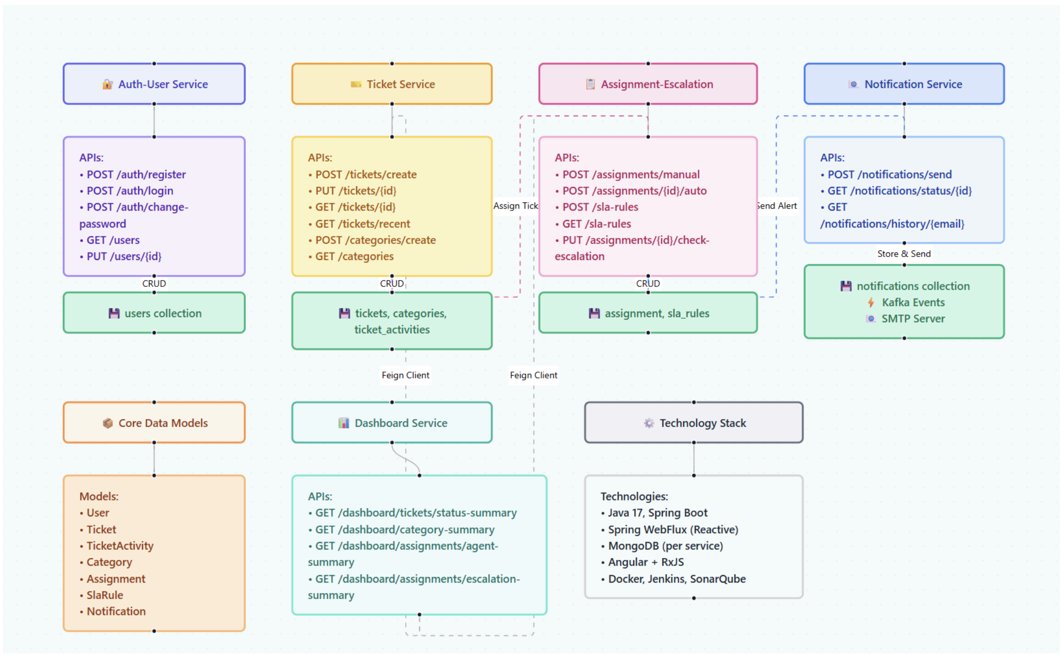
- No dedicated DB (aggregates data from other services)

6.6 Api Gateway

Responsibilities

- Forwarding api points

7. DATA DESIGN (LLD)



\User Document

```
{
  "id": "u101",
  "displayId": "USR-001",
  "username": "agent_john",
  "password": "encrypted",
  "email": "john.doe@test.com",
  "roles": ["AGENT", "USER"],
  "enabled": true,
  "passwordLastChanged": "2026-01-01T09:00:00",
  "resetToken": "abc123token",
  "resetTokenExpiry": "2026-01-01T10:00:00Z",
  "agentLevel": "L1"
}
```

Purpose: Stores user identity, roles, and security metadata.

Category Document

```
{
  "id": "c101",
  "name": "Authentication",
  "description": "Issues related to login and password",
  "linkedSlaId": "sla001",
  "active": true
}
```

Purpose: Defines ticket categories and links to SLA rules

Ticket Document

```
{
  "id": "t201",
  "displayId": "TCK-001",
  "title": "Login issue",
  "description": "Unable to login with correct credentials",
  "categoryId": "c101",
  "status": "OPEN",
  "priority": "HIGH",
  "createdBy": "u101",
  "assignedTo": "u202",
  "createdAt": "2026-01-01T10:00:00",
  "updatedAt": "2026-01-01T10:05:00"
}
```

Purpose: Represents ticket lifecycle and metadata.

TicketActivity Document

```
{
  "id": "a301",
  "ticketId": "t201",
  "actorId": "u202",
  "actionType": "COMMENT",
  "details": "Investigating login issue",
  "timestamp": "2026-01-01T10:15:00Z"
}
```

Purpose: Logs timeline of actions on a ticket.
.

Assignment Document

```
{
  "id": "as501",
  "ticketId": "t201",
  "agentId": "u202",
  "assignedAt": "2026-01-01T10:05:00Z",
  "dueAt": "2026-01-01T12:05:00Z",
  "unassignedAt": null,
  "breached": false,
  "breachedAt": null,
  "status": "ASSIGNED",
  "type": "AUTO",
  "escalationLevel": 0
}
```

SlaRule Document

```
{
  "id": "sla001",
  "priority": "HIGH",
  "responseMinutes": 30,
  "resolutionMinutes": 120
}
```

Purpose: Defines SLA rules per priority.

Notification Document

```
{
  "id": "n701",
  "recipient": "john.doe@test.com",
  "subject": "Ticket Assigned",
  "body": "Ticket TCK-001 has been assigned to you.",
  "eventType": "ASSIGNMENT",
  "createdAt": "2026-01-01T10:06:00Z"
}
```

Purpose: Stores notifications sent to users.

Dashboard DTOs

StatusSummaryDto

```
{
  "status": "OPEN",
  "count": 15
}
```

PrioritySummaryDto

```
{
  "priority": "HIGH",
  "count": 8
}
```

CategorySummaryDto

```
{
  "categoryId": "c101",
  "count": 12
}
```

AgentSummaryDto

```
{
  "agentId": "u202",
  "assignedCount": 10,
  "resolvedCount": 7,
  "overdueCount": 2,
  "escalationLevel": 1,
  "averageResolutionTimeMinutes": 45.5
}
```

Purpose: Lightweight aggregation objects for dashboard visualizations.

8. API DESIGN & VALIDATION

- **RESTful APIs**: All services expose endpoints following REST conventions (`GET`, `POST`, `PUT`, `DELETE`).
- **JSON Payloads**: Requests and responses use JSON format for interoperability.
- **Reactive Programming**: APIs return `Mono` or `Flux` for non-blocking I/O.
- Service-level business rule enforcement
- Standard HTTP status codes

  `200 OK` → Successful read/update
  `201 CREATED` → Resource created
  `400 BAD REQUEST` → Validation or business rule violation
  `401 UNAUTHORIZED` → Invalid credentials
  `403 FORBIDDEN` → Role/permission denied
  `404 NOT FOUND` → Resource not found
  `409 CONFLICT` → Duplicate resource (e.g., user already exists)
  `500 INTERNAL SERVER ERROR` → Unexpected system error

9. ERROR HANDLING STRATEGY

Global Exception Handling

- Centralized using `@ControllerAdvice`
- Converts exceptions into standard JSON error responses.

```
{
  "timestamp": "2026-01-01T10:00:00",
  "status": 400,
  "error": "Validation Error",
  "message": "Priority must be one of [LOW, MEDIUM, HIGH, CRITICAL]"
}
```

10. SECURITY DESIGN

- Password encryption (BCrypt)
- Role-based access control
- JWT authentication
- Secure API access

11. NON-FUNCTIONAL REQUIREMENTS

Area	Design Decision
Scalability	Stateless services, Containerization, Database per Service
Performance	Pagination, async calls, Reactive APIs
Availability	Independent services, Health Checks
Maintainability	POM-like layered backend
Security	Validation, encryption
Observability	Centralized Logging, monitoring, Error Tracking, Audit Trails

12. TESTING STRATEGY

Backend

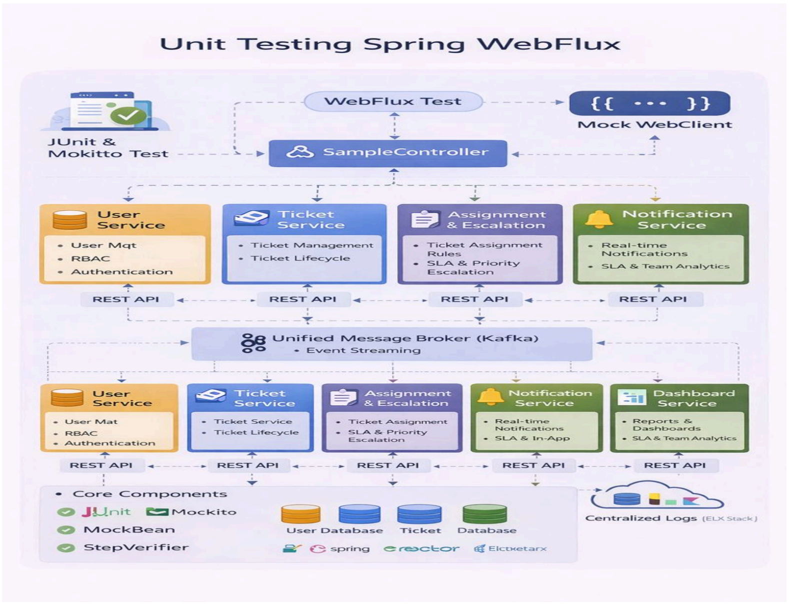
- Unit tests (Service layer)
- Controller tests (MockMvc)
- Repository Tests
- Minimum 90% coverage

Frontend

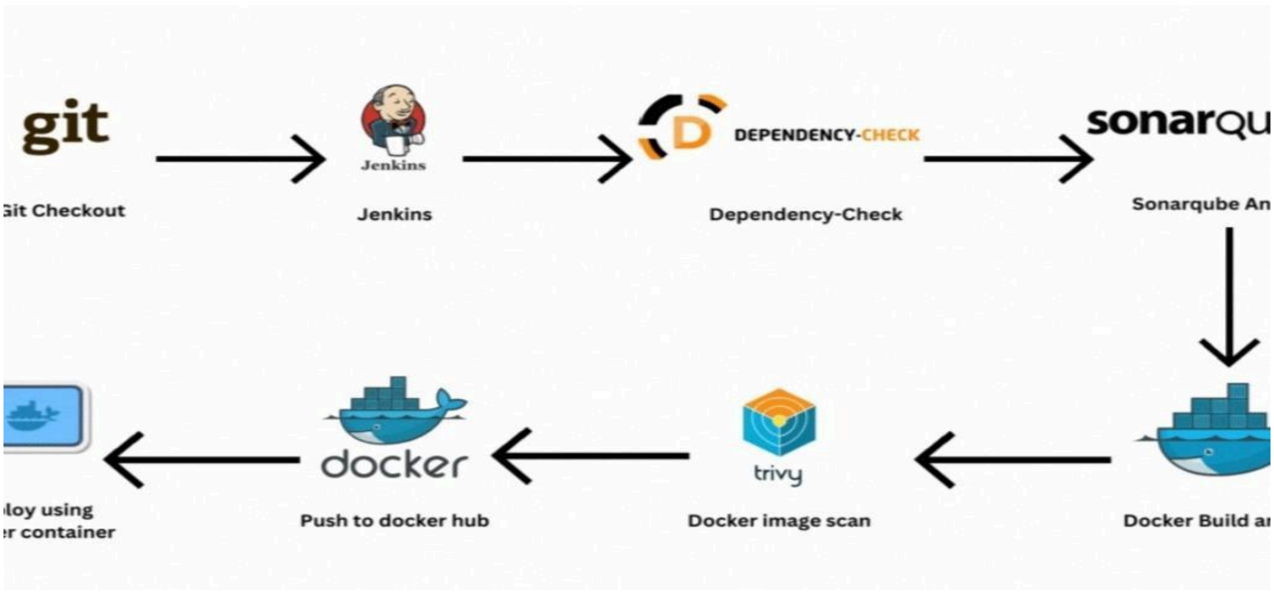
- Component tests
- Service tests

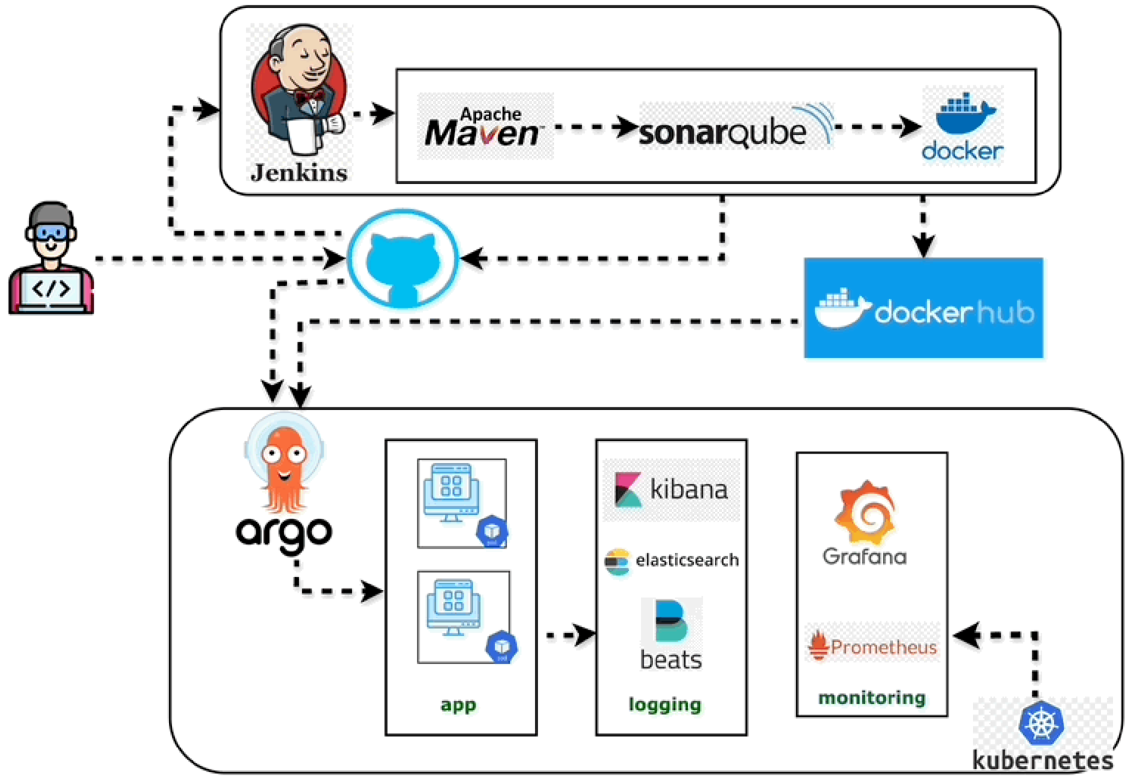
Quality Gates

- SonarQube enforced
- Build fails on violations



13. CI/CD DESIGN





Pipeline Flow

1. Git Commit
2. Jenkins Build
3. Unit Tests
4. SonarQube Scan
5. Quality Gate Check
6. Docker Build
7. Docker Compose Deploy

---

14. DEPLOYMENT DESIGN

- Docker image per microservice
- docker-compose for orchestration
- Environment-specific configs

---

15. ASSUMPTIONS & CONSTRAINTS

Assumptions

- Services communicate over REST
- MongoDB available
- Docker environment present

Constraints

- No distributed transactions
- Event-driven architecture out of scope

---

16. FUTURE ENHANCEMENTS

- Kubernetes deployment
- AI-Driven Auto Assignment
- Multi-Channel Notifications
- Advanced SLA Management

---

17. CONCLUSION

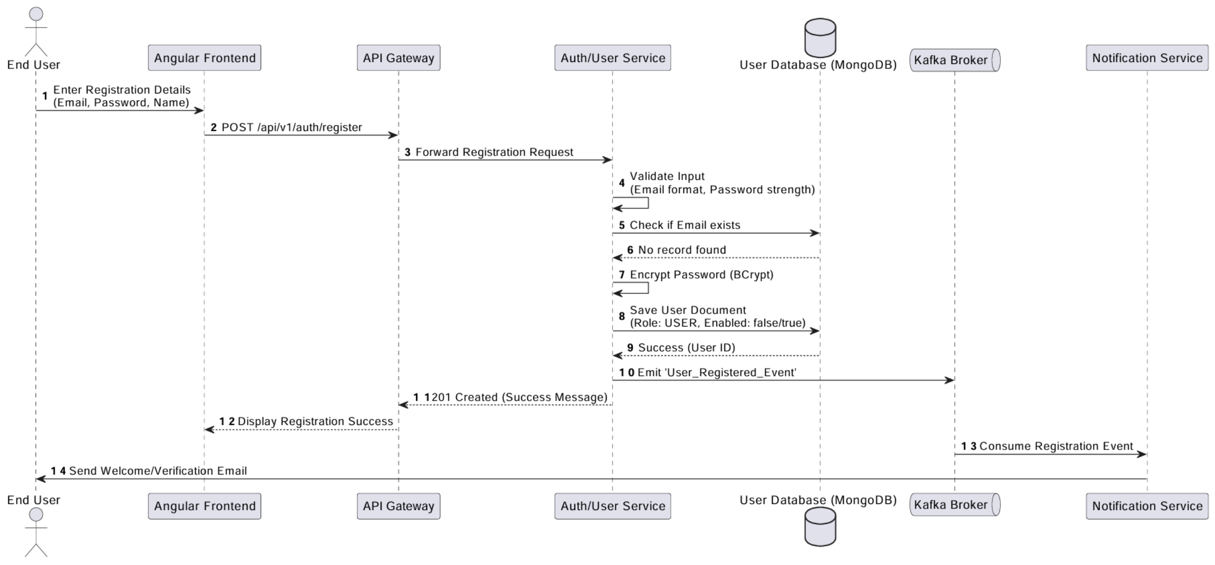
This design ensures:
✔ Clean separation of concerns
✔ Scalability & maintainability
✔ Testability & CI/CD readiness
✔ Interview-ready explanation

SEQUENCE DIAGRAMS

1. User Registration — Sequence Diagram

Scenario

A new user registers using the Angular UI.


End User
Angular Frontend
API Gateway
Auth/User Service
User Database (MongoDB)
Kafka Broker
Notification Service
1 Enter Registration Details (Email, Password, Name)
2 POST /api/v1/auth/register
3 Forward Registration Request
4 Validate Input (Email format, Password strength)
5 Check if Email exists
6 No record found
7 Encrypt Password (BCrypt)
8 Save User Document (Role: USER, Enabled: false/true)
9 Success (User ID)
10 Emit 'User_Registered_Event'
11 201 Created (Success Message)
12 Display Registration Success
13 Consume Registration Event
14 Send Welcome/Verification Email

Flow

1. User enters details in Angular UI
2. Angular sends `POST /users/register` to User Service
3. User Service validates request (email, password)
4. User Service checks email uniqueness in MongoDB
5. Password is encrypted
6. User is saved in MongoDB
7. Success response sent back to UI

Participants

- User
- Angular UI
- User Service
- MongoDB

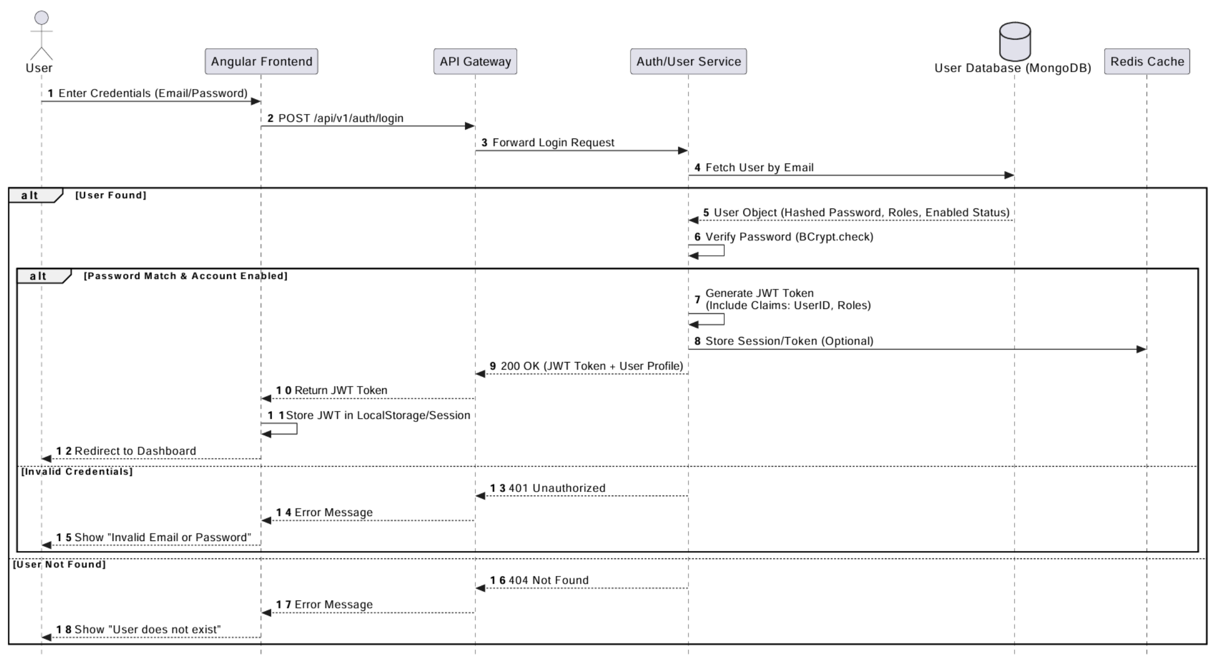
Key Design Points

- Validation at API boundary
- Encryption at service layer
- No direct DB access from UI

2. User Login — Sequence Diagram

Scenario

Registered user logs into the system.


User
Angular Frontend
API Gateway
Auth/User Service
User Database (MongoDB)
Redis Cache
1 Enter Credentials (Email/Password)
2 POST /api/v1/auth/login
3 Forward Login Request
4 Fetch User by Email
alt [User Found]
5 User Object (Hashed Password, Roles, Enabled Status)
6 Verify Password (BCrypt.check)
alt [Password Match & Account Enabled]
7 Generate JWT Token (Include Claims: UserID, Roles)
8 Store Session/Token (Optional)
9 200 OK (JWT Token + User Profile)
10 Return JWT Token
11 Store JWT in LocalStorage/Session
12 Redirect to Dashboard
[Invalid Credentials]
13 401 Unauthorized
14 Error Message
15 Show "Invalid Email or Password"
[User Not Found]
16 404 Not Found
17 Error Message
18 Show "User does not exist"

Flow

1. User submits login form
2. Angular calls `POST /users/login`
3. User Service fetches user from MongoDB
4. Password is verified
5. (Optional) JWT token generated
6. Response sent to Angular
7. Angular stores token and navigates user

Participants

- User
- Angular UI
- User Service
- MongoDB

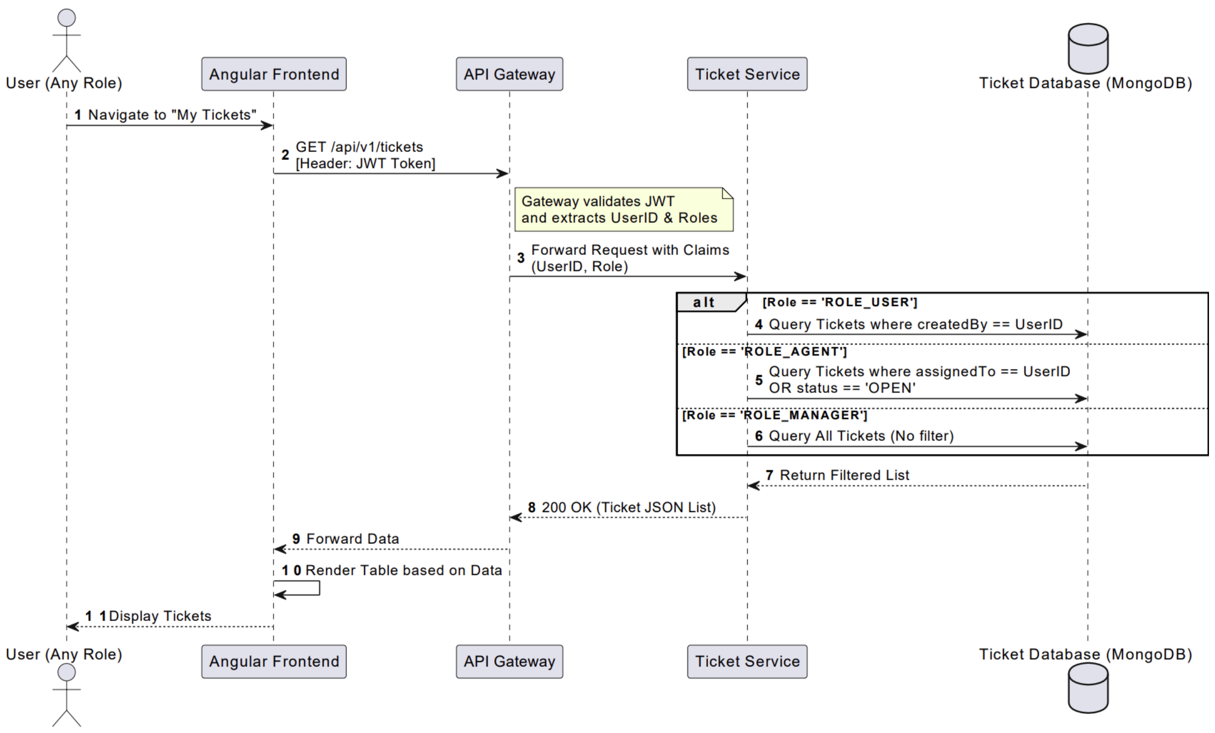
Failure Cases

- Invalid credentials → 401
- Inactive user → 403

3. View Tickets (Role-Based) — Sequence Diagram

Scenario

A logged-in user requests to view tickets, and the system filters the results based on their assigned role (End User, Agent, or Manager).


User (Any Role)
Angular Frontend
API Gateway
Ticket Service
Ticket Database (MongoDB)
1 Navigate to "My Tickets"
2 GET /api/v1/tickets [Header: JWT Token]
Gateway validates JWT and extracts UserID & Roles
3 Forward Request with Claims (UserID, Role)
alt
[Role == 'ROLE_USER']
4 Query Tickets where createdBy == UserID
[Role == 'ROLE_AGENT']
5 Query Tickets where assignedTo == UserID OR status == 'OPEN'
[Role == 'ROLE_MANAGER']
6 Query All Tickets (No filter)
7 Return Filtered List
8 200 OK (Ticket JSON List)
9 Forward Data
10 Render Table based on Data
11 Display Tickets

Flow

1. The user navigates to the ticket list on the Angular UI.
2. Angular UI `GET /tickets` sends a request to the Ticket Service, including the JWT Token in the header.
3. Ticket Service extracts the User ID and Role from the JWT claims.
4. Ticket Service determines the query filter based on the role:
   - **End User:** Filter by `createdBy == UserID`.
   - **Agent:** Filter by `assignedTo == UserID` or specific categories.
   - **Manager:** No filter (fetches all records).
5. Ticket Service queries MongoDB with the appropriate filter
6. MongoDB returns the filtered result set.
7. Ticket Service sends the data back to the UI.
8. Angular UI renders the list for the use

Participants

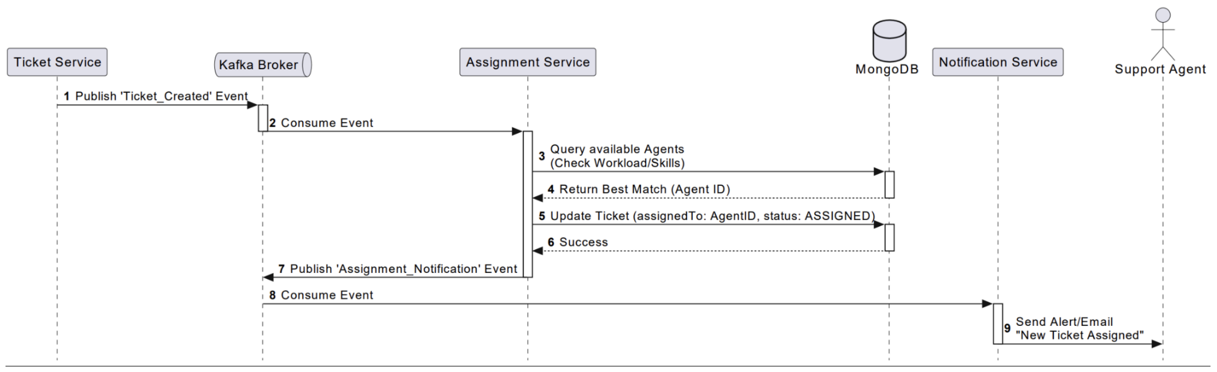
- Angular UI
- Product Service
- MongoDB
- User

---

4. Ticket Assignment — Sequence Diagram

Scenario

Once a ticket is created, the system automatically assigns it to an available Support Agent based on workload, or a Manager manually overrides the assignment.


Ticket Service
Kafka Broker
Assignment Service
MongoDB
Notification Service
Support Agent
1 Publish 'Ticket_Created' Event
2 Consume Event
3 Query available Agents (Check Workload/Skills)
4 Return Best Match (Agent ID)
5 Update Ticket (assignedTo: AgentID, status: ASSIGNED)
6 Success
7 Publish 'Assignment_Notification' Event
8 Consume Event
9 Send Alert/Email "New Ticket Assigned"

Flow

1. Ticket Service publishes a `Ticket_Created` event to Kafka
2. Assignment-Escalation Service consumes the event and triggers the assignment logic.
3. Assignment Service queries MongoDB to find an available Support Agent (least busy or specific category).
4. Assignment Service updates the Ticket Service (via REST or Shared DB) with the `assignedTo` ID and sets status to `ASSIGNED`.
5. Assignment Service sends an event to Kafka for the Notification Service.
6. Notification Service sends an email/alert to the Support Agent.
7. Success Response is logged in the Ticket Activity audit trail.

Participants

- Ticket Service
- Angular UI
- Kafka Broker
- Assignment-Escalation Service
- MongoDB
- Support Agent

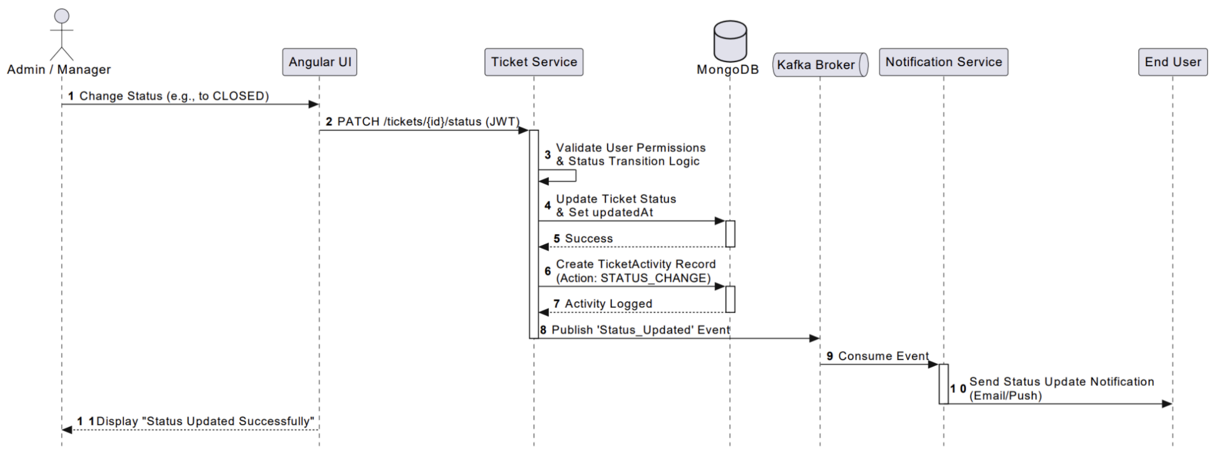
Key Design Decisions

- Inventory validation before order creation
- Atomic stock update logic
- Failure stops order creation

5. Ticket Status Update — Sequence Diagram

Scenario

An Agent or Manager updates the status of an existing ticket (e.g., from `ASSIGNED` to `IN_PROGRESS` or `CLOSED`) through the Angular UI.



Admin / Manager
Angular UI
Ticket Service
MongoDB
Kafka Broker
Notification Service
End User
1 Change Status (e.g., to CLOSED)
2 PATCH /tickets/{id}/status (JWT)
3 Validate User Permissions & Status Transition Logic
4 Update Ticket Status & Set updatedAt
5 Success
6 Create TicketActivity Record (Action: STATUS_CHANGE)
7 Activity Logged
8 Publish 'Status_Updated' Event
9 Consume Event
10 Send Status Update Notification (Email/Push)
11 Display "Status Updated Successfully"

Flow

1. Admin selects a new status from a dropdown in the Angular UI and submits.
2. Angular UI sends a `PUT /tickets/{id}/status` request to the Ticket Service with the JWT Token.
3. Ticket Service extracts the User Role from the JWT to ensure the user has permission to change statuses.
4. Ticket Service validates the status transition (e.g., a `CLOSED` ticket cannot be moved back to `OPEN` without specific rights).
5. Ticket Service updates the status and `updatedAt` timestamp in MongoDB.
6. Ticket Service calls the Communication Service (or uses an internal helper) to log the change in the TicketActivity collection.
7. Ticket Service publishes a `Status_Updated` event to Kafka.
8. Notification Service consumes the event and notifies the End User that their ticket status has changed.
9. Ticket Service returns a `200 OK` response to the UI.

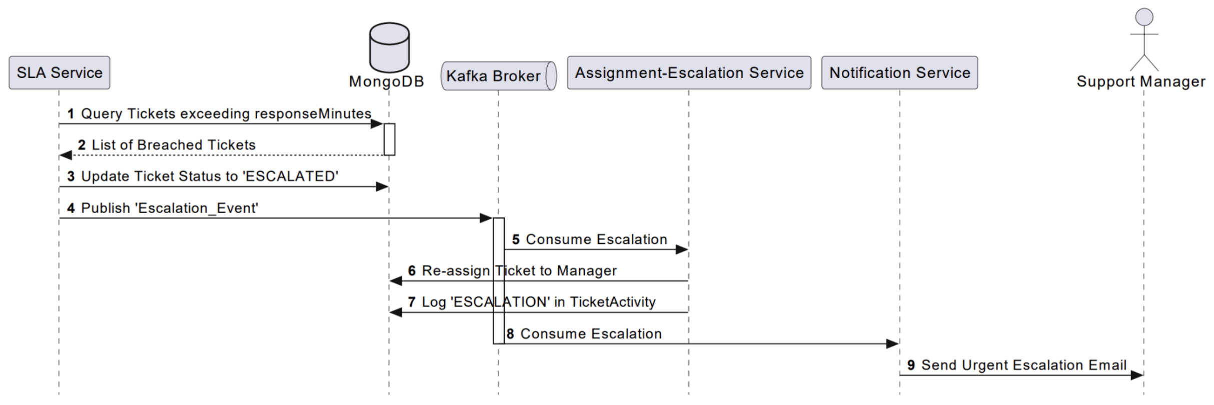
Valid Transitions

- OPEN→ ASSIGNED
- ASSIGNED→ IN_PROGRESS
- IN_PROGRESS→ RESOLVED

6. SLA Breach & Escalation — Sequence Diagram

Scenario

The system automatically detects that a ticket has not been resolved within the time limit defined in the SlaRule and escalates it to a Manager.


SLA Service
MongoDB
Kafka Broker
Assignment-Escalation Service
Notification Service
Support Manager
1 Query Tickets exceeding responseMinutes
2 List of Breached Tickets
3 Update Ticket Status to 'ESCALATED'
4 Publish 'Escalation_Event'
5 Consume Escalation
6 Re-assign Ticket to Manager
7 Log 'ESCALATION' in TicketActivity
8 Consume Escalation
9 Send Urgent Escalation Email

Flow

1. SLA Service runs a background scheduled task.
2. It queries the Ticket DB for OPEN or IN_PROGRESS tickets where currentTime > SLA_Deadline.
3. SLA Service identifies a breach and updates the ticket status to ESCALATED.
4. SLA Service publishes an Escalation_Event to Kafka.
5. Assignment Service consumes the event and re-assigns the ticket to a Manager.
6. Notification Service consumes the event and alerts the Manager immediately.
7. The TicketActivity log is updated to record the automated escalation.

Participants

- SLA Service
- Ticket DB
- Kafka Broker
- Assignment Service
- Notification Service
- Support Manager

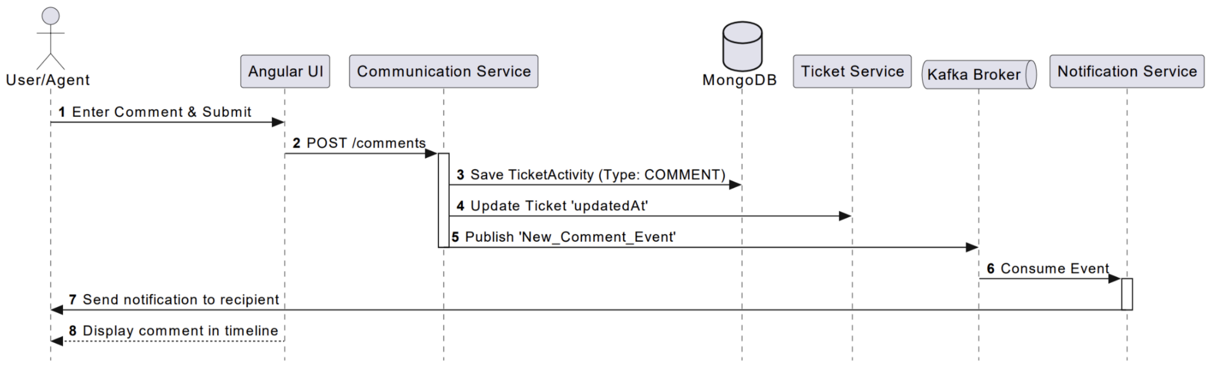
Key Design Points

- Automated Governance: No human intervention is needed to flag overdue tickets.
- Decoupled Architecture: Using Kafka ensures that the escalation logic doesn't block the monitoring service.

7.Ticket Commenting & Communication — Sequence Diagram

Scenario

A User or Agent adds a comment to a ticket to provide updates or ask questions.


User/Agent
Angular UI
Communication Service
MongoDB
Ticket Service
Kafka Broker
Notification Service
1 Enter Comment & Submit
2 POST /comments
3 Save TicketActivity (Type: COMMENT)
4 Update Ticket 'updatedAt'
5 Publish 'New_Comment_Event'
6 Consume Event
7 Send notification to recipient
8 Display comment in timeline

Flow

1. User/Agent enters a comment in the Angular UI.
2. Angular UI sends `POST /tickets/{id}/comments` to the Communication Service.
3. Communication Service saves the comment in the Activity DB.
4. Communication Service updates the Ticket Service to refresh the `updatedAt` timestamp.
5. Communication Service sends a notification event to Kafka.
6. Notification Service sends a real-time alert to the other party (User or Agent).

Participants

- User / Agent
- Angular UI
- Communication Service
- Activity DB
- Ticket Service
- Notification Service

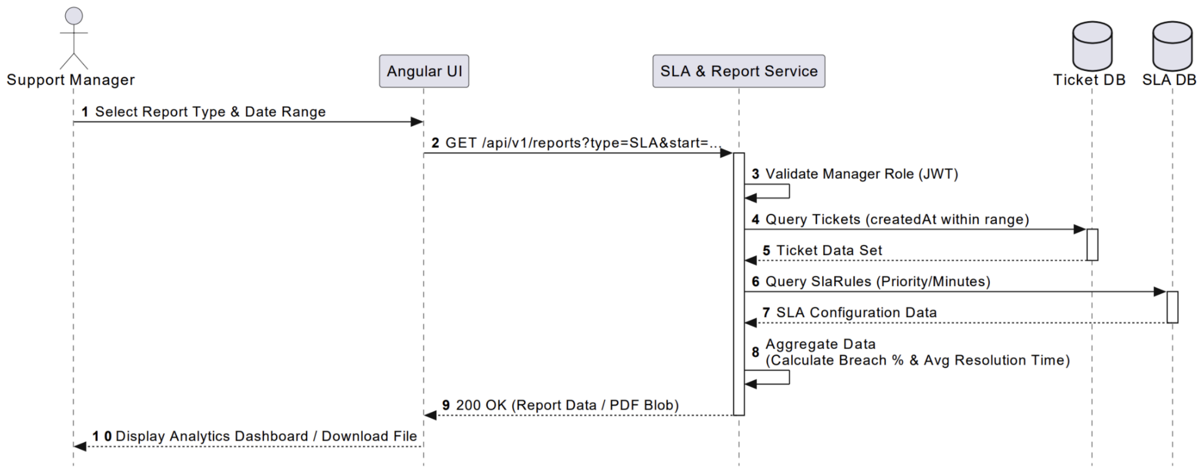
Key Design Points

- Audit Trail: Every comment is stored as a `TicketActivity` record
- Sync vs Async: Comment saving is synchronous, but the notification is asynchronous.

8. Reporting & Analytics — Sequence Diagram

Scenario

A Support Manager or Administrator requests a performance report (e.g., Ticket Resolution Trends or SLA Compliance) through the Angular dashboard to monitor team efficiency.



Support Manager
Angular UI
SLA & Report Service
Ticket DB
SLA DB
1 Select Report Type & Date Range
2 GET /api/v1/reports?type=SLA&start=...
3 Validate Manager Role (JWT)
4 Query Tickets (createdAt within range)
5 Ticket Data Set
6 Query SlaRules (Priority/Minutes)
7 SLA Configuration Data
8 Aggregate Data (Calculate Breach % & Avg Resolution Time)
9 200 OK (Report Data / PDF Blob)
10 Display Analytics Dashboard / Download File

Flow

1. Angular UI sends a `GET /reports` request with parameters and JWT Token to the SLA & Report Service.
2. SLA & Report Service validates the Manager's role via the JWT.
3. SLA & Report Service queries the Ticket Database for tickets within the timeframe and their final statuses.
4. SLA & Report Service queries the SLA Database to compare actual resolution times against `SlaRule` targets (response vs. resolution minutes).
5. The service performs Data Aggregation (calculating averages, breach percentages, and counts).
6. The service generates a JSON payload for charts or a PDF/Excel file for download.
7. SLA & Report Service returns the data/file to the UI.
8. Angular UI renders the graphical charts (Bar/Pie charts) for the Manager.

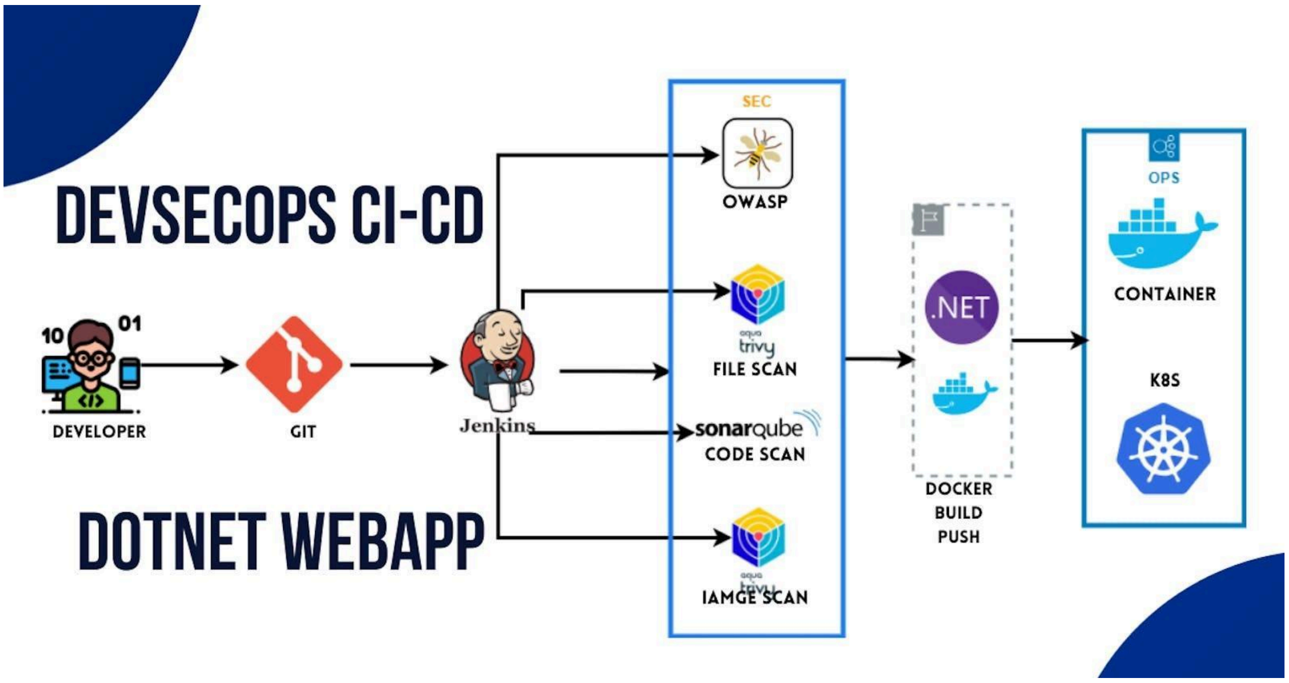
Participants

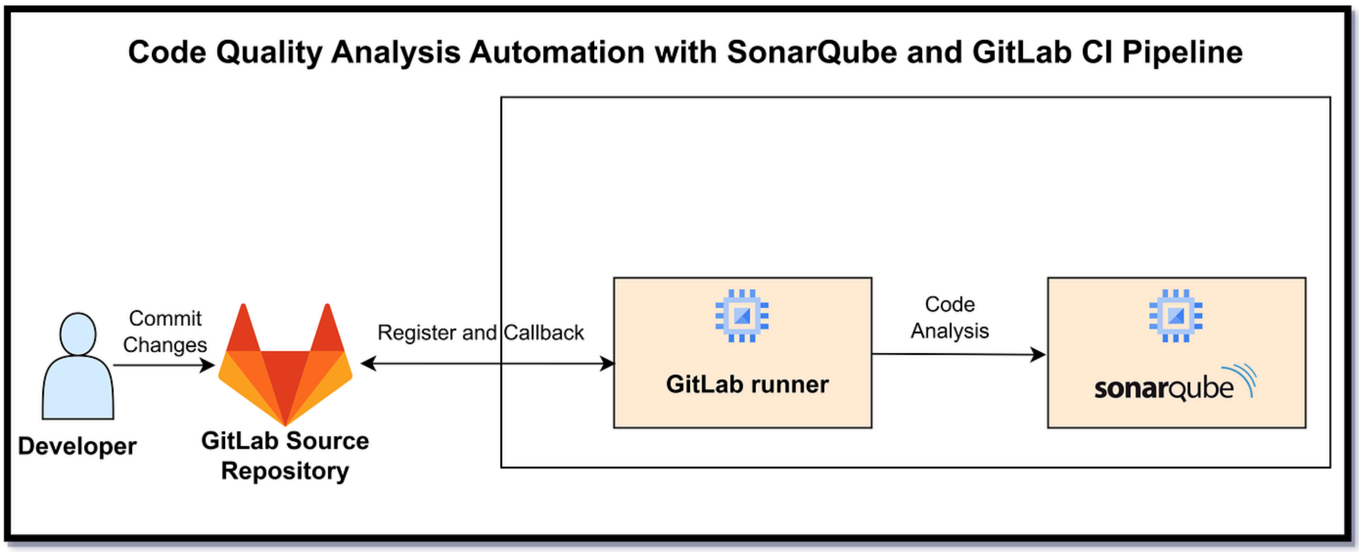
- Support Manager
- Angular UI
- SLA & Report Service
- Ticket Database (MongoDB)
- SLA_Rule Collection (MongoDB)

Key Design Points

- Data Aggregation: Complex calculations are handled at the service layer to keep the UI lightweight.

9. CI/CD Pipeline — Sequence Diagram





Scenario

Developer pushes code to Git repository.

Flow

1. Developer pushes code
2. Jenkins pipeline triggered
3. Jenkins runs unit tests

4. Jenkins runs SonarQube scan
5. Quality gate checked
6. Docker images built
7. Docker Compose deploys services

Participants

- Developer
- Git
- Jenkins
- SonarQube
- Docker

---

10. Error Handling — Sequence Diagram

Scenario

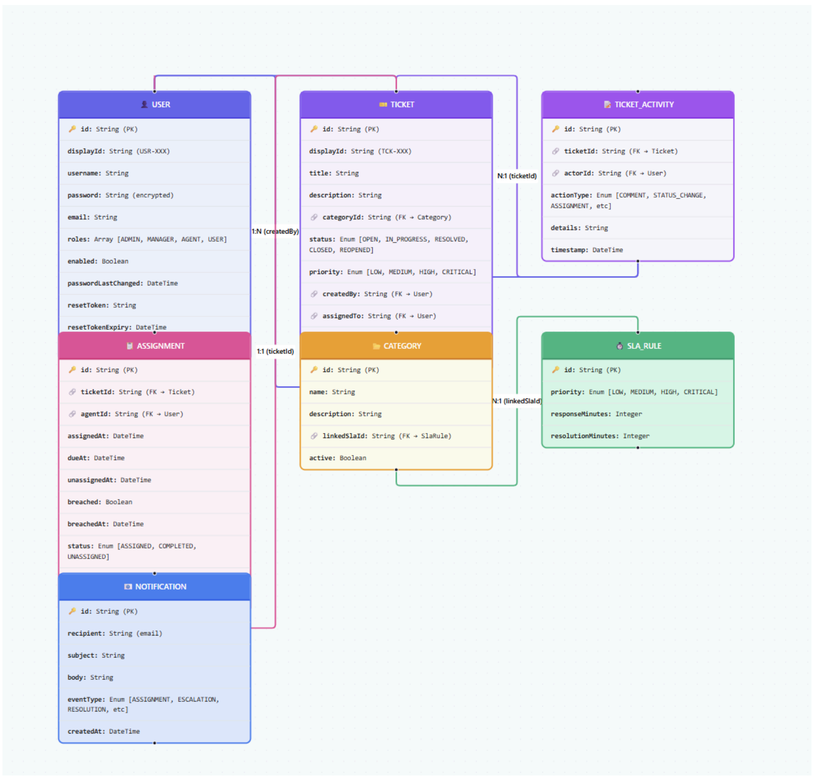
Invalid request sent to backend.

Flow

1. Angular sends invalid request
2. Controller validation fails
3. Global exception handler triggered
4. Standard error response returned

Key Point

- Consistent error structure across services


USER
id: String (PK)
displayId: String (USR-XXX)
username: String
password: String (encrypted)
email: String
roles: Array [ADMIN, MANAGER, AGENT, USER]
enabled: Boolean
passwordLastChanged: DateTime
resetToken: String
resetTokenExpiry: DateTime
TICKET
id: String (PK)
displayId: String (TCK-XXX)
title: String
description: String
categoryId: String (FK → Category)
status: Enum [OPEN, IN_PROGRESS, RESOLVED, CLOSED, REOPENED]
priority: Enum [LOW, MEDIUM, HIGH, CRITICAL]
createdBy: String (FK → User)
assignedTo: String (FK → User)
TICKET_ACTIVITY
id: String (PK)
ticketId: String (FK → Ticket)
actorId: String (FK → User)
actionType: Enum [COMMENT, STATUS_CHANGE, ASSIGNMENT, etc]
details: String
timestamp: DateTime
1:N (createdBy)
N:1 (ticketId)
1:1 (ticketId)
N:1 (linkedSlaId)
ASSIGNMENT
id: String (PK)
ticketId: String (FK → Ticket)
agentId: String (FK → User)
assignedAt: DateTime
dueAt: DateTime
unassignedAt: DateTime
breached: Boolean
breachedAt: DateTime
status: Enum [ASSIGNED, COMPLETED, UNASSIGNED]
CATEGORY
id: String (PK)
name: String
description: String
linkedSlaId: String (FK → SlaRule)
active: Boolean
SLA_RULE
id: String (PK)
priority: Enum [LOW, MEDIUM, HIGH, CRITICAL]
responseMinutes: Integer
resolutionMinutes: Integer
NOTIFICATION
id: String (PK)
recipient: String (email)
subject: String
body: String
eventType: Enum [ASSIGNMENT, ESCALATION, RESOLUTION, etc]
createdAt: DateTime

1. USER SERVICE (user-service)

```
user-service
└─ src/main/java/com/example/user
    ├─ UserServiceApplication.java
    │
    ├─ controller
    │   └─ UserController.java
    │
    ├─ service
    │   ├─ UserService.java
    │   └─ UserServiceImpl.java
    │
    ├─ repository
    │   └─ UserRepository.java
    │
    ├─ model
    │   └─ User.java
    │
    ├─ dto
    │   ├─ UserRegisterRequest.java
    │   ├─ LoginRequest.java
    │   └─ UserResponse.java
    │
    ├─ exception
    │
    │
    │   └─ GlobalExceptionHandler.java
    │
    └─ config
        └─ SecurityConfig.java
```

Key Responsibilities

- `UserController` → API layer
- `UserService` → Business rules
- `UserRepository` → MongoDB access
- `SecurityConfig` → Password encoding / JWT (optional)

---

2. TICKET SERVICE (ticket-service)

```
ticket-service
└─ src/main/java/com/example/ticket
    ├─ TicketServiceApplication.java
    │
    ├─ controller
    │   └─ TicketController.java
    │
    ├─ service
    │   ├─ TicketService.java
    │   └─ TicketServiceImpl.java
    │
    ├─ repository
    │   └─ TicketRepository.java
    │
    ├─ model
    │   └─ Ticket.java
    │
    ├─ dto
    │   ├─ TicketRequest.java
    │   ├─ TicketResponse.java
    │   └─ TicketTrendDto.java
    │
    ├─ exception
    │   ├─ TicketNotFoundException.java
    │   └─ GlobalExceptionHandler.java
    │
    └─ config
        └─ MongoConfig.java
```

Key Responsibilities

- `TicketController` → API endpoints for CRUD & reports
- `TicketServiceImpl` → Business rules (create, assign, resolve, trend reports)
- `TicketRepository` → Reactive MongoDB access
- `MongoConfig` → Reactive Mongo setup

---

3. CATEGORY SERVICE (category -service)

```
category-service
└─ src/main/java/com/example/category
    ├─ CategoryServiceApplication.java
    │
    ├─ controller
    │   └─ CategoryController.java
    │
    ├─ service
    │   ├─ CategoryService.java
    │   └─ CategoryServiceImpl.java
    │
    ├─ repository
    │   └─ CategoryRepository.java
    │
    ├─ model
    │   └─ Category.java
    │
    ├─ dto
    │   ├─ CategoryRequest.java
    │   └─ CategoryResponse.java
    │
    └─ exception
        ├─ CategoryNotFoundException.java
        └─ GlobalExceptionHandler.java
```

Key Responsibilities

- `CategoryController` → Manage categories
- `CategoryServiceImpl` → Business logic for category CRUD
- `CategoryRepository` → MongoDB access

---

4. AGENT SERVICE (agent-service)

```
agent-service
└─ src/main/java/com/example/agent
   ├─ AgentServiceApplication.java
   │
   ├─ controller
   │   └─ AgentController.java
   │
   ├─ service
   │   ├─ AgentService.java
   │   └─ AgentServiceImpl.java
   │
   ├─ repository
   │   └─ AgentRepository.java
   │
   ├─ model
   │   └─ Agent.java
   │
   ├─ dto
   │   ├─ AgentRequest.java
   │   └─ AgentResponse.java
   │
   └─ exception
       ├─ AgentNotFoundException.java
       └─ GlobalExceptionHandler.java
```

Key Responsibilities

- `AgentController` → APIs for agent management & performance board
- `AgentServiceImpl` → Business logic for assignment, stats, dashboards
- `AgentRepository` → MongoDB access

---

5. REPORT SERVICE (report-service)

```
angular-ui
└─ src/app
   ├─ core
   │   ├─ services
   │   │   ├─ auth.service.ts
   │   │   ├─ product.service.ts
   │   │   └─ order.service.ts
   │   │
   │   └─ guards
   │       └─ auth.guard.ts
   │
```

```
├─ modules
│   ├─ auth
│   │   ├─ login.component.ts
│   │   └─ register.component.ts
│   │
│   ├─ product
│   │   └─ product-list.component.ts
│   │
│   └─ order
│       ├─ checkout.component.ts
│       └─ order-history.component.ts
│
├─ shared
│   └─ models
│       ├─ user.model.ts
│       ├─ product.model.ts
│       └─ order.model.ts
│
└─ app.module.ts
```

Key Responsibilities

- `ReportController` → APIs for ticket trends, agent performance, category stats
- `ReportServiceImpl` → Aggregation logic for analytics

---

6. COMMON / SHARED CONCEPTS (IMPORTANT)

```
common
└─ src/main/java/com/example/common
   ├─ exception
   │   └─ ApiErrorResponse.java
   │
   ├─ util
   │   └─ Constants.java
   │
   └─ config
       └─ SwaggerConfig.java
```

---

7. TEST STRUCTURE

```
src/test/java
└─ com/example
   ├─ controller
   │   ├─ TicketControllerTest.java
   │   └─ AgentControllerTest.java
   │
   ├─ service
   │   ├─ TicketServiceTest.java
   │   ├─ AgentServiceTest.java
   │   └─ ReportServiceTest.java
   │
   └─ repository
       ├─ TicketRepositoryTest.java
       └─ CategoryRepositoryTest.java
```

7. ANGULAR FRONTEND (angular-ui)

```
src/
 └── app/
      ├── core/                    # Core services & guards
      │    ├── services/
      │    │    ├── auth.service.ts
      │    │    ├── ticket.service.ts
      │    │    ├── activity.service.ts
      │    │    └── report.service.ts
      │    ├── guards/
      │    │    └── role.guard.ts
      │    └── interceptors/
      │         └── jwt.interceptor.ts
      │
      ├── shared/                  # Shared UI components & models
      │    ├── components/
      │    │    └── navbar.component.ts
      │    ├── models/
      │    │    ├── ticket.model.ts
      │    │    ├── activity.model.ts
      │    │    └── report.model.ts
      │    └── utils/
      │         └── date-format.pipe.ts
      │
      ├── modules/
      │    ├── tickets/
      │    │    ├── ticket-list/
      │    │    │    └── ticket-list.component.ts
      │    │    ├── ticket-detail/
      │    │    │    └── ticket-detail.component.ts
      │    │    ├── ticket-form/
      │    │    │    └── ticket-form.component.ts
      │    │    └── timeline/
      │    │         └── timeline.component.ts
      │    │
      │    ├── dashboard/
      │    │    ├── status-summary/
      │    │    │    └── status-summary.component.ts
      │    │    ├── priority-summary/
      │    │    │    └── priority-summary.component.ts
      │    │    ├── category-summary/
      │    │    │    └── category-summary.component.ts
      │    │    └── sla-compliance/
      │    │         └── sla-compliance.component.ts
      │    │
      │    ├── auth/
      │    │    ├── login/
      │    │    │    └── login.component.ts
      │    │    └── profile/
      │    │         └── profile.component.ts
      │    │
      │    └── notifications/
      │         └── notification-list/
      │              └── notification-list.component.ts
      │
      ├── app-routing.module.ts     # Route definitions
      ├── app.component.ts          # Root component
      └── app.module.ts             # Root module
```

8. DEVOPS & CI/CD FILES

```
capstone-project
├─ Jenkinsfile
├─ docker-compose.yml
├─ ticket-service/Dockerfile
├─ category-service/Dockerfile
├─ agent-service/Dockerfile
├─ report-service/Dockerfile
├─ angular-ui/Dockerfile
└─ README.md
```

9. DOCKER COMPOSE

```
version: '3.8'

services:
  mongo:
    image: mongo:6.0
    container_name: mongo
    ports:
      - "27017:27017"
    volumes:
      - mongo_data:/data/db

  zookeeper:
    image: confluentinc/cp-zookeeper:7.5.0
    container_name: zookeeper
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
    ports:
      - "2181:2181"

  kafka:
    image: confluentinc/cp-kafka:7.5.0
    container_name: kafka
    depends_on:
      - zookeeper
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka:9092
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
    ports:
      - "9092:9092"

  eureka:
    build: ./smartticket-service-registry
    container_name: smartticket-service-registry
    ports:
      - "8761:8761"

  config-server:
    build: ./smartticket-config-server
    container_name: smartticket-config-server
    ports:
      - "8888:8888"
    depends_on:
```

```
    - eureka

api-gateway:
  build: ./smartticket-api-gateway
  container_name: smartticket-api-gateway
  ports:
    - "8765:8765"
  depends_on:
    - eureka
    - config-server

auth-service:
  build: ./auth-user-service
  container_name: auth-user-service
  ports:
    - "8081:8081"
  env_file:
    - .env
  depends_on:
    - mongo
    - kafka
    - eureka
    - config-server

dashboard-service:
  build: ./dashboard-service
  container_name: dashboard-service
  ports:
    - "8082:8082"
  env_file:
    - .env
  depends_on:
    - mongo
    - kafka
    - eureka
    - config-server

notification-service:
  build: ./notification-service
  container_name: notification-service
  ports:
    - "8083:8083"
  env_file:
    - .env
  depends_on:
    - mongo
    - kafka
    - eureka
    - config-server

assignment-escalation-service:
  build: ./assignment-escalation-service
  container_name: assignment-escalation-service
  ports:
    - "8084:8084"
  env_file:
    - .env
  depends_on:
    - mongo
    - kafka
    - eureka
    - config-server

ticket-service:
    build: ./ticket-service
    container_name: ticket-service
    ports:
      - "8085:8085"
    env_file:
      - .env
    depends_on:
      - mongo
      - kafka
      - eureka
      - config-server

volumes:
  mongo_data:
```

---

7. JENKINSFILE (CI/CD)

```
pipeline {
  agent any

  stages {
    stage('Checkout') {
      steps { git 'https://github.com/your-org/smart-order-management.git' }
    }

    stage('Build & Test') {
      steps { sh 'mvn clean test' }
    }

    stage('SonarQube Scan') {
      steps {
        withSonarQubeEnv('SonarQube') {
          sh 'mvn sonar:sonar'
        }
      }
    }

    stage('Quality Gate') {
      steps {
        timeout(time: 2, unit: 'MINUTES') {
          waitForQualityGate abortPipeline: true
        }
      }
    }

    stage('Docker Build & Deploy') {
      steps {
        sh 'docker-compose up -d --build'
      }
    }
  }
}
```

8. README.md (MINIMAL TEMPLATE)

```
# Smart Order Management System

## Tech Stack
- Spring Boot Microservices
- Angular
- MongoDB
- Docker
- Jenkins + SonarQube

## Run Locally
docker-compose up -d

## Services
- User Service: 8081
- Product Service: 8082
- Order Service: 8083
- UI: http://localhost:4200
```

CORE BUSINESS RULES

1. Users & Access

- Users must log in to use the system.
- Each user has a role: Admin, Manager, Agent, User.
- Users can only do actions allowed by their role.

2. Ticket Creation

- Any logged-in user can raise a ticket.
- Tickets must have category, priority, and description.
- Ticket is created with status CREATED.

3. Ticket Lifecycle

Tickets follow this order only:

```
CREATED → ASSIGNED → IN_PROGRESS → RESOLVED → CLOSED
```

- Invalid status changes are not allowed.

4. Ticket Assignment

- Only the Support Manager can assign tickets.
- A ticket can be assigned to one support agent at a time.

5. Ticket Handling

- Only the assigned agent can work on the ticket.
- The agent must move the ticket to IN_PROGRESS before resolving.

6 Resolution & Closure

- Tickets must have a resolution summary before RESOLVED.
- Tickets can be CLOSED after resolution.
- Closed tickets cannot be updated.

7 Reopen & Cancel

- Users can reopen a resolved/closed ticket.
- Tickets can be cancelled only before work starts.

8 SLA & Escalation

- Each priority has an SLA time.
- If SLA is crossed, the ticket is escalated automatically.

9 Notifications

- User gets notified when:
  - Ticket is created
  - Ticket status changes
  - Ticket is resolved

10 Reporting

- Dashboard shows ticket counts and status.
- Managers can view team performance and SLA status.