Low-Level Design (LLD) — Enterprise Ticketing System

Table of Contents – Low Level Design (LLD)

3.3 Basic Project Management (Kanban, Epics, Stories, Sub-tasks)

- Presentation Layer (UI)
- API Layer
- Backend Layer
- Data Layer

3.4 Workflow Automation

- Presentation Layer (UI)
- API Layer
- Backend Layer
- Data Layer

3.5 Collaboration & Communication

- Presentation Layer (UI)
- API Layer
- Backend Layer
- Data Layer

3.6 Reporting & Analytics

- Presentation Layer (UI)
- API Layer
- Backend Layer
- Data Layer

4. External Interface Requirements

4.1 User Interfaces (UI/UX Guidelines)
4.2 Software Interfaces (APIs, Microservices, Gateways)
4.3 Communications Interfaces (SMTP, Notifications)

5. Non-Functional Requirements & Design Strategy

5.1 Performance

1. Introduction

1.1 Purpose

This document (LLD) translates the SRS and HLD into actionable, implementation-ready design guidance for the Enterprise Ticketing System MVP. It is intended to be the single source of truth for:

- API design choices (payload shapes, error envelope, security model),
- data model decisions and DDL-ready constraints,
- integration/factory choices (storage, queueing, worker model),
- operational and deployment expectations (backup, monitoring, scalability),
- coding & infra conventions that developers must follow.

Goal: Developers should be able to implement features (frontend + backend + worker + infra) with minimal ambiguity using this LLD.

1.2 Document Conventions

Naming & formatting

- Database fields: `snake_case` (e.g., `created_at`, `ticket_key`).
- API JSON fields (frontend ↔ API): `camelCase` (e.g., `createdAt`, `ticketKey`).
- Pydantic models follow `PascalCase` class names.
- React components: `PascalCase` file/component names.
- All timestamps are ISO-8601 in UTC (e.g., `2025-09-22T10:00:00Z`).

HTTP & error envelope

- Standard response envelope for errors:

```
{
  "error": "E_SHORT_CODE",
  "message": "Human readable explanation",
  "details": { /* optional structured info */ }
}
```

- Success: return resource body (no wrapper) except list endpoints which return `{ items: [...], meta: { page, pageSize, total } }`.

Security

- JWTs use RS256 (asymmetric). Public key is used by API servers to verify; private key used by auth service to sign.
- Secrets are stored in a secrets manager (Vault / AWS Secrets Manager); do not hardcode.

Coding & style

- Backend Python: target **Python 3.11**, FastAPI, black formatter, isort, flake8.
- Frontend: React 18 + TypeScript, ESLint, Prettier.
- DB migrations: Alembic (Postgres).
- CI: GitHub Actions; PRs must pass lint + unit tests.

Versioning

- API versions: `/api/v1/...`. When incompatible changes occur, bump version to v2.
- Database migrations: strictly via Alembic with migration files checked into repo.

1.3 Intended Audience

- Backend developers (FastAPI, workers)
- Frontend developers (React + TypeScript)
- DevOps / SRE (k8s, CI/CD, monitoring)
- QA engineers and Test automation engineers
- Product Owners & Technical Writers (for acceptance criteria)

This document is authoritative for implementation details. Tests, mock servers and stubs should adhere to this LLD.

1.4 Project Scope (MVP)

The MVP delivers the core ticketing and lightweight project-management capability:

In-scope (MVP):

- Authentication & RBAC (Admin, Manager, Team Member, Client)
- Ticket CRUD with lifecycle (open → in_progress → resolved → closed + reopen)
- Kanban board UI (drag & drop status updates)
- Epics → Stories → Subtasks data model and basic UI
- Commenting, mentions, attachments (uploads via pre-signed S3)
- Workflow automation rules (JSON condition → action) executed by background worker
- In-app notifications and email notifications
- Basic reporting: dashboard KPIs & export pipeline (async for big exports)

- Audit logging (`ticket_history`) for critical operations

Out-of-scope (MVP):

- Mobile native apps
- Full-text search across all artifacts (deferred — use Postgres indexes now)
- Real-time collaborative editing (deferred) — UI will use polling/WS hook points
- Advanced analytics (ML/forecasting)

1.5 Definitions, Acronyms, and Abbreviations

- **API** — Application Programming Interface
- **DB** — Database
- **DDL** — Data Definition Language (SQL schema)
- **FK / PK** — Foreign Key / Primary Key
- **JSONB** — Postgres JSON Binary column type
- **JWT** — JSON Web Token
- **RBAC** — Role-Based Access Control
- **SLA** — Service Level Agreement
- **S3** — Object storage (AWS S3 or compatible)
- **PITR** — Point-in-time Recovery
- **MVP** — Minimum Viable Product
- **CI/CD** — Continuous Integration / Continuous Deployment

2. Overall Description

This section describes the product perspective, the set of features, user classes, environment, constraints and the exact technology choices and high-level architecture.

2.1 Product Perspective

The system is a web-first SaaS platform to manage tickets, small projects, and team workflows. It is a single-tenant or multi-tenant-capable platform (MVP: single-tenant per deployment; multi-tenant flags designed but deferred). The frontend is a Single Page Application (SPA) that uses REST APIs to communicate with the backend. Background tasks are executed asynchronously by Celery workers.

Primary flows

- User logs in → views dashboard (aggregates) → creates ticket → system evaluates rules → notifications sent → team works via Kanban.

Key integrations

- SMTP (emails)
- S3-compatible object storage for file attachments
- Managed Postgres (recommended for production)
- Redis for caching + Celery broker

2.2 Product Features (concise)

1. **Authentication & RBAC** — secure login, short-lived access tokens, refresh tokens, role enforcement.
2. **Ticketing** — create/read/update/list/soft-delete; ticket history.
3. **Project Management** — Epics, Stories, Subtasks; Kanban board.
4. **Workflow Automation** — admin-defined JSON rules, worker engine.
5. **Collaboration** — comments, mentions, attachments, in-app & email notifications.
6. **Reporting** — dashboard KPIs, export (CSV/XLSX/PDF) with async generation for large datasets.
7. **Operational** — logging, metrics, monitoring, backups, alerts.

2.3 User Classes and Characteristics

Admin

- Full system access (manage users, teams, rules, all tickets).
- Actions: user CRUD, rule creation, global settings, export all reports.

Manager

- Manage projects/teams, assign tickets inside their teams, view team reports.
- Actions: create tickets, assign, move on Kanban, create stories/epics.

Team Member

- Work on assigned tickets, comment, update status, upload attachments.

Client

- Create tickets, view own tickets, comment.
- Limited permissions (no assignments, no admin features).

Notes: Roles are enforced by API dependency injection (FastAPI `Depends`) and middleware; frontend hides UI elements based on role, but backend enforces.

2.4 Operating Environment

Local development

- Docker Compose with services: `api`, `db` (Postgres), `redis`, `minio` (S3-compatible), `celery_worker`, `frontend`.
- Python 3.11, Node 18+.

Staging / Production

- Containerized workloads on Kubernetes (EKS/GKE/AKS).
- Managed Postgres (RDS / Cloud SQL recommended).
- Redis: managed (ElastiCache / Memorystore) or cluster.
- Object storage: AWS S3 (recommended) with server-side encryption (SSE-KMS or SSE-S3).

- TLS termination at Ingress (ALB / NGINX) with certificates managed by ACM / cert-manager.

Resource sizing (starting guidance)

- API: 2 replicas (t2.medium / equivalent), autoscale based on CPU/requests.
- Celery: 2 worker replicas initially; autoscale by queue backlog.
- Postgres: db.m5.large class (or cloud equivalent) with backups.
- Redis: single instance / cluster depending on load.

2.5 Design and Implementation Constraints

- **Security & Compliance**
  - All endpoints must be HTTPS.
  - Sensitive PII must not be logged (mask emails/passwords in logs).
  - Passwords hashed with bcrypt (cost/work factor ≥ 12).
  - Token signing uses RSA key pair and supports rotation.
- **File uploads**
  - Max single file size: **20 MB** (configurable).
  - Allowed MIME types: `image/png`, `image/jpeg`, `application/pdf`, `application/msword`, `application/vnd.openxmlformats-officedocument.wordprocessingml.document`, others whitelisted.
  - Uses pre-signed S3 URLs; backend stores metadata only.
- **Audit**
  - Every ticket change persists an entry into `ticket_history` with `old_value` and `new_value`.
- **Concurrency**

    - Expect heavy board updates — use optimistic concurrency (`updated_at` timestamp / ETag) to detect conflicting writes.

- **Internationalization**
    - Timestamps in UTC; frontend shows in user local timezone (user profile `timeZone`).

2.6 Assumptions and Dependencies

Assumptions

- An SMTP provider is available (SES, SendGrid) for emails.
- S3-compatible storage is available for attachments.
- The initial user base is moderate (100–1,000 active users); later scale will be addressed via autoscaling.
- Single-tenant deployment per organization is acceptable for MVP.

Dependencies

- PostgreSQL 13+ (tested on 13/14)
- Redis 6+
- Celery 5.x
- FastAPI 0.95+ (or latest stable)
- React 18+, TypeScript 5+

2.7 High-Level Architecture & Technology

Architectural pattern

- 3-tier + workers:
    - Presentation: React SPA (client)
    - API Gateway / Backend: FastAPI monolith (modular services) — split into modules: `auth`, `users`, `tickets`, `projects`, `comments`, `rules`, `reports`.

    - Background Workers: Celery for async tasks (notifications, scanning, exports, rule evaluation).
    - Data Layer: PostgreSQL (primary), Redis (cache + broker), S3 (attachments).

Rationale for choices

- **FastAPI**: high-performance, automatic OpenAPI docs, Pydantic validation, async support.
- **Postgres**: robust relational model, JSONB support (for flexible fields), mature tooling.
- **Redis + Celery**: well-known worker + broker fit for background jobs and scaling.
- **S3 pre-signed uploads**: efficient, secure file transfer without routing file bytes through API.
- **Kubernetes**: production-grade orchestration and autoscaling; supports separate lifecycle for API & workers.

Security & secrets

- Use a secrets manager (Vault / AWS Secrets Manager) for DB credentials, JWT private key, SMTP credentials.
- TLS endpoints; HSTS; security headers.

Observability

- Metrics: Prometheus exporter on API & Celery, default metrics: request latency, error rates, DB connections, queue length.
- Tracing: OpenTelemetry integration.
- Logs: structured JSON logs with correlation IDs (`X-Request-ID`).
- Alerts: PagerDuty / Opsgenie integration via webhook for SRE.

CI / CD

- CI: GitHub Actions runs lint, unit tests, build images.

- CD: GitHub Actions / ArgoCD / Helm deploys to K8s on `main` merge (after tests).

Initial configuration defaults

- Access token TTL: **15 minutes**
- Refresh token TTL: **30 days**
- Rate limit for login endpoint: **10 attempts / 10 minutes** (per IP)
- Default page size for paginated lists: **25**, max page size **100**
- Password policy: min 8 chars, at least 1 number and 1 letter (regex: `(?=^.{8,}$)(?=.*\d)(?=.*[A-Za-z]).*$`)

Quick decisions to lock in now (so code can be scaffolded)

1. **Auth model**: JWT (access short), refresh tokens stored hashed in DB, logout revokes refresh token; access tokens cannot be revoked individually (use short TTL + revocation list in Redis for emergency).
2. **Attachment flow**: pre-signed URL request → client uploads to S3 → client calls `complete` endpoint → backend schedules virus scan → mark `available` on success.
3. **Rule engine**: JSONB rules stored in DB; worker reads rules by `when` (event) and runs a simple matcher (key equality, tag contains, regex on title/content). Complex rules deferred.
4. **Ticket key**: DB sequence `ticket_seq` + prefix `TSK-`. Use `SELECT nextval('ticket_seq')` in transaction to build `ticket_key`.
5. **Timezones**: store UTC; frontend converts to user's timezone based on `users.time_zone` or browser setting.

Directory Mapping:

backend/

└─ app/

  ├─ api/

  │ └─ v1/

  ├─ schemas/

  ├─ models/

  ├─ services/

  ├─ workers/

  ├─ db/

  │ └─ migrations/

  ├─ utils/

  ├─ metrics/

  └─ core/

frontend/

└─ src/

  ├─ api/

  ├─ components/

  │ ├─ Reports/

  │ ├─ Tickets/

  │ ├─ Board/

  │ ├─ Projects/

│ ├─ Workflows/

│ ├─ Comments/

│ ├─ Attachments/

│ └─ Notifications/

├─ hooks/

├─ pages/

│ └─ reports/

└─ context/

infrastructure/

└─ k8s/

3.1 User Management & Access Control — Complete Detail

Summary

Handles authentication (login/refresh/logout), user CRUD (create/read/update/deactivate), roles (admin, manager, team_member, client), password reset, and RBAC enforcement. Provides JWT-based auth with short-lived access tokens + refresh tokens (secure cookie or stored hashed in DB). Includes account lockout and rate-limiting for protection.

3.1.1 Actors & Permission Matrix (quick)

- **Admin**: full CRUD on users, manage roles, view all tickets & reports, create workflow rules.
- **Manager**: create users within department / team, assign tickets within own teams, view team reports.
- **Team Member**: view/modify assigned tickets, comment, upload attachments.

- **Client**: create tickets, view own tickets, comment.

RBAC is enforced server-side; frontend only hides UI per role (server never trusts frontend).

3.1.2 Presentation Layer (UI)

Inputs (forms / fields)

1. **Login Form**
    - `email` (string)
    - `password` (string)
    - `rememberMe` (boolean) — optional, controls refresh token cookie expiration
2. **User Create / Invite Form** (Admin/Manager)
    - `name` (string)
    - `email` (string)
    - `role` (enum: `admin` | `manager` | `team_member` | `client`)
    - `teamId` (optional int)
    - `departmentId` (optional int)
    - `sendInvite` (boolean) — if true, email invite sent
3. **User Edit Form**
    - `name`, `role`, `departmentId`, `teamId`, `status` (`active` or `inactive`), `timeZone`, `phone`
4. **Password Reset**
    - Request: `email`
    - Reset: `token`, `newPassword`, `confirmPassword`

Client-side validations (immediate)

- `email`: validate with RFC-like regex; show message `Invalid email format.`
- `password`: min 8 chars, must have at least one letter and one digit; show `Password must be at least 8 characters and contain letters and numbers.`
- `name`: non-empty, trimmed.
- `role`: must be one of allowed values.
- `teamId`/`departmentId`: optional but chosen from preloaded dropdown (fetch `GET /api/v1/teams` / `GET /api/v1/departments`).
- `password` / `confirmPassword` must match on reset page.
- Show UX help text: password policy, session timeout, two-step details.

UI behavior & outputs

- On successful login: store `accessToken` in memory and `refreshToken` in **httpOnly secure cookie** (recommended). Update global auth context (user object & role).
- On create user + `sendInvite=true`: show "Invitation sent" toast.
- On errors: show user-friendly inline messages (map API error codes to messages).
- Provide `Logout` action that calls `POST /api/v1/auth/logout` and clears local tokens.

Accessibility & UX

- Mark inputs with labels and ARIA attributes.
- Keyboard accessible forms and error focus.
- Provide strength meter for passwords (optional).

Frontend Files (where to implement)

```
frontend/src/components/Auth/LoginForm.tsx
frontend/src/components/Auth/LogoutButton.tsx
frontend/src/components/User/UserCreateForm.tsx
frontend/src/components/User/UserEditForm.tsx
```

```
frontend/src/hooks/useAuth.ts
frontend/src/context/AuthContext.tsx
frontend/src/api/auth.ts
```

3.1.3 API Layer (Gateway) — Endpoints, Payloads, Contracts

Endpoints (core)

- `POST /api/v1/auth/login` — login
- `POST /api/v1/auth/refresh` — exchange refresh token for access token
- `POST /api/v1/auth/logout` — revoke refresh token
- `POST /api/v1/auth/request-password-reset` — request reset email
- `POST /api/v1/auth/reset-password` — reset password using token
- `GET /api/v1/users` — list users (admin)
- `POST /api/v1/users` — create user (admin/manager depending)
- `GET /api/v1/users/{id}` — get user
- `PUT /api/v1/users/{id}` — update user
- `DELETE /api/v1/users/{id}` — soft-delete / deactivate
- `GET /api/v1/me` — current auth user info

Request / Response examples

Login request

```
POST /api/v1/auth/login
Content-Type: application/json

{
  "email": "alice@example.com",
  "password": "Secret123!"
}
```

Login response (success)

- Response status: `200 OK`
- Response body:

```
{
  “access_token”: "<JWT_ACCESS_TOKEN>",
  "expiresIn": 900,            // seconds (15 minutes)
  "user": { "id": 42, "email": "alice@example.com",
"name": "Alice", "role": "manager" }
}
```

- Additionally: server sets `Set-Cookie: refreshToken=<token>; HttpOnly; Secure; SameSite=Strict; Path=/api/v1/auth/refresh; Max-Age=2592000`

Refresh token endpoint

```
POST /api/v1/auth/refresh
Cookie: refreshToken=<token>
```

Response: `200 OK` with new `accessToken` (and optionally new refresh token cookie rotated).

Logout

```
POST /api/v1/auth/logout
Authorization: Bearer <accessToken>
Cookie: refreshToken=<token>
```

- Response: `204 No Content` and server clears refresh token cookie and blacklists refresh token id.

API-level Validations

- Content-Type enforcement.
- JSON schema validation (Pydantic).
- Authentication dependency (FastAPI `Depends`) for protected routes; `get_current_user()` sets `request.state.user`.
- Rate limiting:
  - `POST /api/v1/auth/login` limit: default `10 attempts per 10 minutes per IP` (configurable).

    - Endpoint returns `429 Too Many Requests` on limit exceed with `Retry-After` header.
- Input size limits (e.g., 8KB per request body for auth endpoints).

Error responses (standardized)

Examples:

- `400 Bad Request`

```
{ "error": "E_INVALID_PAYLOAD", "message": "Missing email or password" }
```

- `401 Unauthorized` (invalid credentials)

```
{ "error": "E_AUTH_INVALID", "message": "Email or password is incorrect" }
```

- `423 Locked` (account locked after repeated failed attempts)

```
{ "error": "E_ACCOUNT_LOCKED", "message": "Account locked due to too many failed attempts. Try again after 30 minutes." }
```

- `429 Too Many Requests`

```
{ "error": "E_RATE_LIMIT", "message": "Too many requests. Try again later." }
```

- `403 Forbidden` (role restrictions)

```
{ "error": "E_FORBIDDEN", "message": "You do not have permission to perform this action." }
```

- `409 Conflict` (user already exists)

```
{ "error": "E_USER_EXISTS", "message": "A user with this email already exists." }
```

API Files (gateway)

`backend/app/api/v1/routes_auth.py`

```
backend/app/api/v1/routes_users.py
backend/app/api/deps.py           # get_current_user,
require_roles
backend/app/core/security.py     # JWT helpers
backend/app/core/config.py       # rate limits, token TTL
```

3.1.4 Backend Layer (Business Logic)

Responsibilities

- Authenticate credentials (compare hashed password).
- Generate access JWT and refresh token (signed and persisted hashed).
- Enforce account lockout policy and record failed logins.
- Provide user management operations: create, update, soft-delete.
- Enforce RBAC on critical ops.
- Send invitation & password reset emails via worker.
- Maintain audit logs (e.g., user created/updated actions).

Authentication flow (detailed)

1. **Login**
    - Receive `email` + `password`.
    - Rate-limit check (per IP & per account). If exceeded -> 429.
    - Lookup user by `email`. If not found -> return 401 without revealing existence.
    - If found, check `status='active'`, else return 423 or 403.
    - Compare password with stored hash (bcrypt `checkpw`).
        - If wrong: increment failed login counter (Redis or DB), if counter exceeds threshold (e.g., 5), set `locked_until = now() + 30 minutes` and log event.
        - If correct: reset failed login counter; generate `access_token` (JWT) and `refresh_token` (random UUID or signed JWT),

store refresh token hashed in DB (or store token id) with `expires_at`.

- Return access token in body and set refresh token in httpOnly cookie.

2. **Refresh**
    - Client calls `POST /auth/refresh` with cookie containing refresh token.
    - Server validates refresh token exists and not revoked; if valid, create new access token; optionally rotate refresh token (issue new refresh token and revoke old one).
    - Rotation reduces risk of refresh token replay.
3. **Logout**
    - Revoke refresh token: mark stored token as revoked in DB or store in revocation list in Redis.
    - Clear refresh cookie.
4. **Password reset**
    - `POST /auth/request-password-reset` with `email`. If email exists, generate `reset_token` (short TTL e.g., 1 hour, store hashed with user id) and emit email job via Celery to send reset link (`/reset-password?token=xxx`). Return `200 OK` (do not reveal existence).
    - `POST /auth/reset-password` with `token`, `newPassword`: validate token, set `password_hash` with new bcrypt hash, delete/reset token record, log event.

Business validations (backend)

- Only `admin` can set role to `admin`.
- Manager can create users with roles `team_member` and `client` for their teams; cannot create admins.
- Email uniqueness enforced; return `E_USER_EXISTS` if violation occurs.
- All modifications produce audit entry: `user_audit` table (actor_id, target_id, action, before, after).

Token structure: access JWT (example claims)

```
{
  "iss": "ticketing.example.com",
  "sub": "42",                 // user id
  "email": "alice@example.com",
  "role": "manager",
  "iat": 1695379200,
  "exp": 1695380100,          // short TTL
  "jti": "uuid-v4"            // optional JWT id for
traceability
}
```

Refresh tokens stored in DB as:

```
refresh_tokens (
  id BIGSERIAL PRIMARY KEY,
  user_id BIGINT REFERENCES users(id),
  token_hash TEXT NOT NULL,
  issued_at TIMESTAMPTZ,
  expires_at TIMESTAMPTZ,
  revoked BOOLEAN DEFAULT FALSE
)
```

Store only the hash of the token (compare via bcrypt/sha256) to avoid plaintext tokens saved.

Audit logging

- `user_audit` table records `action` (`create`, `update`, `delete`, `password_reset`), `changed_by` (user id or system), `old_value`, `new_value`, `created_at`.

Security considerations

- Hash refresh tokens (never store plaintext).
- Use strong RNG for refresh tokens (UUIDv4 + HMAC).
- On suspicious activity (multiple failed login attempts, login from new IP/country), optionally trigger MFA or admin alert.

Implementation files (backend)

```
backend/app/services/auth_service.py
backend/app/services/user_service.py
backend/app/models/user.py          # SQLAlchemy model
backend/app/models/refresh_token.py
backend/app/schemas/auth.py         # Pydantic
LoginRequest/LoginResponse
backend/app/schemas/user.py
backend/app/db/session.py
backend/app/db/migrations/*.py      # alembic migrations
backend/app/workers/tasks_email.py # send invite/reset
emails
backend/app/core/security.py        # token create/verify
utilities
```

Backend pseudocode (login)

```
def login(email, password, ip):
    if is_rate_limited(ip): raise RateLimitError
    user = db.query(User).filter(User.email ==
email).one_or_none()
    # do not reveal which condition failed
    if user is None or user.status != 'active':
        record_failed_attempt(user, ip)
        raise AuthInvalid
    if not bcrypt.check_password(password,
user.password_hash):
        record_failed_attempt(user, ip)
        if failed_count_exceeds_threshold(user):
lock_account(user)
        raise AuthInvalid
    reset_failed_attempts(user)
    access_token = create_jwt(user.id, user.role)
    refresh_token = create_refresh_token()
    store_refresh_token_hash(user.id, refresh_token)
    set_refresh_cookie(refresh_token)
    return access_token, user
```

Testing the backend behavior

- Unit tests for:

    - `authenticate_user` returns user on correct password.
    - `authenticate_user` fails and increments failed attempt counter on wrong password.
    - Token rotation logic: refresh rotates tokens.
    - Access to admin-only endpoints returns 403 for non-admins.
- Integration tests:
    - Full login -> refresh -> access protected resource -> logout flows.

3.1.5 Data Layer (Database)

Tables (DDL snippets — Alembic-ready)

users

Column Name	Data Type	Constraints / Default	Description / Notes
id	BIGSERIAL	PRIMARY KEY	Auto-incrementing unique ID
uuid	UUID	UNIQUE, DEFAULT gen_random_uuid()	Universally unique identifier
name	VARCHAR(200)		User's full name
email	VARCHAR(320)	NOT NULL, UNIQUE	User's email address, must be unique
password_hash	TEXT	NOT NULL	Hashed password for authentication
role	VARCHAR(32)	NOT NULL	User role (e.g., admin, user, manager)
status	VARCHAR(16)	NOT NULL, DEFAULT 'active'	User status: 'active', 'inactive', 'banned'
team_id	BIGINT	REFERENCES teams(id)	Foreign key to the `teams` table
department_id	BIGINT	REFERENCES departments(id)	Foreign key to the `departments` table
time_zone	VARCHAR(64)		User's time zone
created_at	TIMESTAMPTZ	NOT NULL, DEFAULT now()	Timestamp when the user was created
updated_at	TIMESTAMPTZ	NOT NULL, DEFAULT now()	Timestamp when the user was last updated

refresh_tokens

Column Name	Data Type	Constraints / Default	Description / Notes
id	BIGSERIAL	PRIMARY KEY	Auto-incrementing unique ID
user_id	BIGINT	NOT NULL, REFERENCES users(id) ON DELETE CASCADE	Links token to a specific user; deleted user cascades delete of tokens
token_hash	TEXT	NOT NULL	Hashed value of the refresh token
issued_at	TIMESTAMPTZ	NOT NULL, DEFAULT now()	Timestamp when the token was issued
expires_at	TIMESTAMPTZ		Optional expiration timestamp for the token
revoked	BOOLEAN	DEFAULT FALSE	Indicates if the token has been revoked
created_at	TIMESTAMPTZ	DEFAULT now()	Timestamp when the token record was created

Indexes:

Index Name	Columns	Notes
idx_refresh_tokens_user	user_id	Speeds up queries by `user_id`

failed_logins (optional Redis alternative; DB used for audit)

Column Name	Data Type	Constraints / Default	Description / Notes
id	BIGSERIAL	PRIMARY KEY	Auto-incrementing unique ID
user_id	BIGINT		Optional reference to the user who attempted login
ip_address	VARCHAR(45		IP address from which the login attempt was
attempted_at	TIMESTAMPTZ	DEFAULT now()	Timestamp when the failed login attempt occurred

user_audit

Column Name	Data Type	Constraints / Default	Description / Notes
id	BIGSERIAL	PRIMARY KEY	Auto-incrementing unique ID

actor_id	BIGINT		ID of the user who performed the action
target_user_id	BIGINT		ID of the user on whom the action was performed
action	VARCHAR(64)		Description of the action performed (e.g., update, delete)
old_value	JSONB		Previous state of the data before the action
new_value	JSONB		New state of the data after the action
created_at	TIMESTAMPTZ	DEFAULT now()	Timestamp when the audit record was created

Constraints & indexes

- `UNIQUE(email)` on users.
- Index on `users.role` for quick role-based queries.
- `refresh_tokens.token_hash` not unique (multiple tokens per user allowed but typically rotate old tokens revoked).
- Use `pgcrypto` or `gen_random_uuid()` extension for UUID generation.

SQLAlchemy model (condensed)

```
from sqlalchemy import Column, Integer, String, Text,
ForeignKey, TIMESTAMP, Boolean
from sqlalchemy.sql import func
from app.db.base_class import Base

class User(Base):
    id = Column(Integer, primary_key=True, index=True)
    uuid = Column(UUID(as_uuid=True),
server_default=text("gen_random_uuid()"), unique=True)
    name = Column(String(200))
    email = Column(String(320), unique=True, index=True,
nullable=False)
    password_hash = Column(Text, nullable=False)
    role = Column(String(32), nullable=False)
    status = Column(String(16), nullable=False,
default='active')
    team_id = Column(Integer, ForeignKey('teams.id'),
nullable=True)
```

```
    department_id = Column(Integer,
ForeignKey('departments.id'), nullable=True)
    time_zone = Column(String(64))
    created_at = Column(TIMESTAMP(timezone=True),
server_default=func.now())
    updated_at = Column(TIMESTAMP(timezone=True),
server_default=func.now(), onupdate=func.now())
```

3.1.6 Error Codes & How to Map Them (for frontend devs)

API Status	Error Code	Message (example)	Frontend handling
400	E_INVALID_PAYLOAD	"Missing email"	show inline field error
401	E_AUTH_INVALID	"Email or password is incorrect"	show above-form error
423	E_ACCOUNT_LOCKED	"Account locked due to failed attempts"	show modal with unlock instructions
403	E_FORBIDDEN	"No permission"	hide UI controls / show toast
409	E_USER_EXISTS	"Email already exists"	show inline on email field
429	E_RATE_LIMIT	"Too many requests"	show retry notice / disable button
500	E_INTERNAL	"Something went wrong"	generic toast, log correlation id

Include `X-Request-ID` in all responses to help correlate logs (backend sets or echoes `X-Request-ID` from client).

3.1.7 Tests & QA Checklist (to include in CI)

Unit tests

- `test_login_success`
- `test_login_wrong_password_increments_failed_count`
- `test_login_lockout_after_threshold`
- `test_refresh_rotates_token`
- `test_logout_revokes_refresh_token`
- `test_admin_can_create_user`
- `test_manager_cannot_create_admin`

Integration tests

- End-to-end login → create ticket (requires auth) → logout flow.
- Password reset flow:
  - Request reset generates token and enqueues email job.
  - Reset endpoint with token updates password and invalidates all refresh tokens.

Security tests

- Verify `refreshToken` cookie has `HttpOnly`, `Secure`, `SameSite=Strict`, `Path=/api/v1/auth/refresh`.
- Ensure `accessToken` not stored in localStorage if front wants to prefer memory usage; if stored, warn.

Load tests

- Rate-limiting on `POST /auth/login` to ensure thresholds trigger.

3.1.8 Implementation Notes, Best Practices & Edge Cases

- **Where to store tokens in the browser?**
  - Best: Put `refreshToken` in `httpOnly` cookie; store `accessToken` in memory (AuthContext) and refresh automatically via refresh cookie when needed.
  - If storing `accessToken` in localStorage, be aware of XSS risk. Use CSP and strict XSS mitigations.
- **Token revocation & blacklisting**
  - Access tokens are short-lived (15 min). Use a Redis blacklist for emergency revocations (store `jti` with TTL equal to token expiry).
  - Refresh tokens persisted in DB. Revoke on logout or rotation.
- **Account lockout**
  - Use short-term lockout (30 minutes) after 5 failed attempts. Exponential backoff option available.

- **Password hashing**
    - Use bcrypt with cost 12 (configurable). Consider Argon2 if available.
- **Email flows**
    - Invitation & password reset emails should contain a `token` for single use (store hashed).
    - Email templates must be configurable and must not include sensitive tokens in logs.
- **Audit logging**
    - Every create/update/delete operation on users should write into `user_audit` table.
- **Rate limiting**
    - Use a shared rate-limiter (Redis) per-IP and per-account (account-only after account exists).
- **User deletion**
    - Soft-delete: set `status='inactive'` not physical delete to preserve FK integrity and history.
- **RBAC enforcement**
    - Centralize role checks in `require_role()` dependency rather than sprinkling role checks throughout route handlers.

3.1.9 Acceptance Criteria for Feature Completion

- Login flow works end-to-end: valid credentials → `200` with access token and refresh cookie; invalid credentials → `401` and failed attempt logged.
- Refresh flow issues new access tokens and rotates refresh token (if rotation enabled).
- Logout revokes refresh token and results in no further refresh being possible.
- Admin can create user, cannot create duplicate email (returns `409`).

- Manager can create team_member and client users limited to his/her team/ department.
- Password reset flow issues an email with one-time token and the token allows updating password; token invalid after use/expiry.
- RBAC test coverage: verify protected admin endpoints return 403 for non-admin users.

3.2 Ticket Creation & Tracking — Complete Detail

Summary

Ticketing subsystem supports creating, reading, updating, listing, searching, and soft-deleting tickets. It enforces lifecycle rules (`open → in_progress → resolved → closed`), records every change in `ticket_history`, supports optimistic concurrency, publishes domain events (`ticket.created`, `ticket.updated`, `ticket.status_changed`) for background workers (workflow rules, notifications, reporting), and integrates with attachments/comments.

3.2.1 Actors & Use Cases

- **Creator (any authenticated user)**: create a ticket.
- **Assignee (team member/manager/admin)**: take ownership, update progress, resolve/close.
- **Manager/Admin**: change priority, reassign, change status, manage tickets across teams (admin).
- **Client**: create and comment on their own tickets, view status updates.

Core use cases:

- Create new ticket
- View ticket details
- Update ticket fields (title, description, priority, assignee)
- Change ticket status

- List/filter tickets (for board and list views)
- Soft-delete ticket (admin)
- Record history for every change
- Publish events for rules & notifications

3.2.2 Presentation Layer (UI)

UI Components & Pages

- `TicketCreateModal` — used to create a new ticket.
- `TicketDetailPage` — shows full ticket, comments, attachments, history.
- `TicketCard` — compact card used on boards.
- `KanbanBoard` — groups tickets by status columns; supports drag & drop.
- `TicketList` — table/list view with filters and pagination.

Inputs (fields and UX flows)

Ticket create form fields:

- `title` (string) — required, max 400 chars.
- `description` (rich text / markdown) — optional.
- `type` (enum): `bug` | `task` | `incident` | `service_request` — default `task`.
- `priority` (enum): `low` | `medium` | `high` — default `medium`.
- `assigneeId` (int, optional) — selected from dropdown/search.
- `teamId` (int, optional) — select team/board.
- `labels/tags` (array of strings) — optional.
- `dueDate` (ISO date, optional).
- `attachments` (file[]) — handled via presigned upload flow.
- `sensitive` (boolean) — flag for restricted visibility (optional feature).

Board actions

- Drag ticket card to change status (client checks allowed transitions).
- Inline edit (quick edits): priority, assignee.

Client-side Validations

- `title` non-empty, trimmed, length ≤ 400.
- `priority` and `type` must be allowed enums.
- `assigneeId` must be selected from known users (client may still send arbitrary id but server validates).
- `dueDate` must be valid date >= today (if business requires).
- Attachments: allowed MIME types and ≤ 20MB (client enforces and shows progress).

UX: Outputs & Optimistic UI

- On successful create: close modal, show created ticket in board/list, navigate to detail.
- Use optimistic updates when creating/updating tickets to improve UX; reconcile with server response.
- Display server-provided `ticketKey` (TSK-xxxx) and createdAt timestamp.
- Show in-card indicators: priority, assignee avatar, tags, due date.

Errors & Handling in UI

- Inline validation errors mapped from API `details`.
- Display toasts for server errors with correlation ID (`X-Request-ID`).
- If optimistic update fails, show error and revert UI.
- For conflict (HTTP 409), display a dialog with server state diff and options: reload / merge / overwrite.

Frontend Files

`frontend/src/components/Tickets/TicketCreateModal.tsx`

```
frontend/src/components/Tickets/TicketDetailPage.tsx
frontend/src/components/Board/KanbanBoard.tsx
frontend/src/components/Tickets/TicketCard.tsx
frontend/src/api/tickets.ts
frontend/src/hooks/useTickets.ts
```

3.2.3 API Layer (Gateway) — Endpoints & Contracts

Endpoints

- `POST /api/v1/tickets` — create ticket
- `GET /api/v1/tickets/{ticketKey}` — get ticket detail
- `PUT /api/v1/tickets/{ticketKey}` — update ticket (partial)
- `PUT /api/v1/tickets/{ticketKey}/status` — change status
- `GET /api/v1/tickets` — list tickets (filters, pagination)
- `DELETE /api/v1/tickets/{ticketKey}` — soft-delete (admin)
- Attachments:
  - `POST /api/v1/tickets/{ticketKey}/attachments/request-upload`
  - `POST /api/v1/tickets/{ticketKey}/attachments/complete`
- `GET /api/v1/tickets/{ticketKey}/history` — ticket history entries (paginated)

Payloads — Examples

Create ticket request

```
POST /api/v1/tickets
Content-Type: application/json
Authorization: Bearer <accessToken>

{
  "title": "Checkout button 500 error",
  "description": "<p>Stacktrace...</p>",
```

```
  "type": "bug",
  "priority": "high",
  "assigneeId": 17,
  "teamId": 3,
  "tags": ["checkout","urgent"],
  "dueDate": "2025-10-01"
}
```

Create ticket response (201)

```
{
  "id": 1245,
  "ticketKey": "TSK-11024",
  "title": "Checkout button 500 error",
  "description": "<p>Stacktrace...</p>",
  "type": "bug",
  "priority": "high",
  "status": "open",
  "creatorId": 42,
  "assigneeId": 17,
  "teamId": 3,
  "tags": ["checkout","urgent"],
  "dueDate": "2025-10-01",
  "createdAt": "2025-09-22T10:12:00Z",
  "updatedAt": "2025-09-22T10:12:00Z"
}
```

Update ticket (partial)

```
PUT /api/v1/tickets/TSK-11024
Content-Type: application/json
Authorization: Bearer <accessToken>
If-Unmodified-Since: 2025-09-22T10:12:00Z   // optional
optimistic header

{
  "title": "Checkout button throws 500 when coupon
applied",
  "assigneeId": 19
}
```

Change status

```
PUT /api/v1/tickets/TSK-11024/status
Content-Type: application/json
Authorization: Bearer <accessToken>

{ "status": "in_progress" }
```

List tickets (filters)

```
GET /api/v1/tickets?
teamId=3&status=open&priority=high&page=1&pageSize=25
```

API Validation Rules (gateway-level)

- All protected endpoints require `Authorization` bearer token.
- Validate JSON schema via Pydantic.
- Validate `ticketKey` format `^TSK-\d+$`.
- Validate `If-Unmodified-Since` header if provided (ISO timestamp).
- Enforce rate limits per user for create operations (configurable).

API Errors (standardized)

- `400 E_INVALID_PAYLOAD` — missing/invalid fields (returns `details` with field errors).
- `401 E_AUTH_INVALID` — authentication missing/invalid.
- `403 E_FORBIDDEN` — insufficient role/permission to perform action (e.g., client trying to assign).
- `404 E_TICKET_NOT_FOUND` — ticketKey doesn't exist.
- `409 E_CONFLICT` — optimistic concurrency conflict (include server `updatedAt` in response).
- `413 E_PAYLOAD_TOO_LARGE` — attachments over size limit when using direct upload endpoints.
- `422 E_INVALID_STATUS_TRANSITION` — trying illegal status change.
- `500 E_INTERNAL` — unhandled server error.

API Files

```
backend/app/api/v1/routes_tickets.py
backend/app/schemas/ticket.py    # Pydantic models for
create/update/out
backend/app/api/deps.py          # require_role,
pagination, get_db
```

Pydantic models (examples)

`schemas/ticket.py`

```
class TicketCreate(BaseModel):
    title: constr(min_length=1, max_length=400)
    description: Optional[str] = None
    type:
Literal['bug','task','incident','service_request'] =
'task'
    priority: Literal['low','medium','high'] = 'medium'
    assigneeId: Optional[int] = None
    teamId: Optional[int] = None
    tags: Optional[List[str]] = []
    dueDate: Optional[date] = None
```

`TicketOut` includes fields from DB with `createdAt`, `updatedAt`.

3.2.4 Backend Layer (Business Logic)

Responsibilities

- Validate business-level rules (RBAC, allowed transitions).
- Generate unique `ticket_key` transactionally.
- Persist ticket and `ticket_history`.
- Publish domain events for workers.
- Handle attachments lifecycle (cooperate with `attachment_service`).
- Support optimistic concurrency and conflict resolution.
- Implement filters and board grouping for UI.

Key business rules & validations

- **Role-based rules**:
  - `client` cannot set `assigneeId` on create or update. If provided by client, ignore or return `403`.
  - `team_member` can only assign to themselves (if policy says), or cannot assign — configurable.
  - `manager` can assign within their department/team; `admin` can assign to anyone.
- **Status lifecycle**:
  - Allowed transitions: `open -> in_progress`, `in_progress -> resolved`, `resolved -> closed`, `closed -> reopened -> in_progress`.
  - Disallow illegal transitions (e.g., `open -> closed` unless explicit `forceClose` provided by admin).
- **Priority changes**:
  - Anyone with proper role can change priority; changing to `high` may trigger workflow rules.
- **Assignee validation**:
  - provided `assigneeId` must exist and be `active`.
  - If adding assignee that belongs to another team, verify role allows cross-team assignment.

Ticket key generation (safe)

- Use Postgres sequence for strong concurrency safety:
  1. Start Alembic migration to create `ticket_seq`:

```
CREATE SEQUENCE ticket_seq START 1;
```

2. On create, in same DB transaction:

   seq = nextval('ticket_seq')

3. ticket_key = 'TSK-' || to_char(seq + 1000, 'FM999999') -- offset for aesthetics
4. INSERT INTO tickets(ticket_key, title, ...) VALUES(ticket_key,...)

- Alternatively, store `ticket_key` as computed column or use a dedicated table to generate keys.

Database transaction & events

- Begin DB tx.
- Insert `tickets` row.
- Insert `ticket_history` (`action='created'`, `new_value` JSONB).
- Commit transaction.
- After commit, publish `ticket.created` event to message broker (Redis/Celery). Publishing after commit ensures workers only consume stable state.

Optimistic concurrency / conflict handling

- `tickets` table includes `updated_at TIMESTAMPTZ`.
- For updates, clients can provide `If-Unmodified-Since` header or `If-Match: <etag>` (etag = hash of `updated_at`).
- Backend checks: if `ticket.updated_at` != provided header → return `409 E_CONFLICT` with current server state and `currentUpdatedAt`.
- For drag/drop board updates where multiple users may update status, recommend using `PUT /tickets/{ticketKey}/status` with the `If-Unmodified-Since` header.

Publishing events (payload)

`ticket.created` payload:

```
{
  "ticketId": 1245,
  "ticketKey": "TSK-11024",
  "creatorId": 42,
  "assigneeId": 17,
  "teamId": 3,
  "priority": "high",
  "type": "bug",
  "createdAt": "2025-09-22T10:12:00Z"
}
```

Workers subscribe to `ticket.created` and `ticket.updated`.

Attachments interaction

- On `create` call, frontend may omit attachments and do attachments after ticket exists.
- `attachment_service` handles presigned URL generation and completes attachment lifecycle (see 3.5 section for details).

Audit / ticket_history rules

- For every `create`, `update`, `status_change`, `assign`, `comment_added`, create `ticket_history` record:
  - `ticket_id`, `changed_by`, `action`, `old_value` (JSONB), `new_value` (JSONB), `created_at`.
- `old_value` / `new_value` should only include changed fields to limit payload size.

Example backend functions (service layer)

- `ticket_service.create_ticket(current_user, ticket_payload) -> TicketOut`
- `ticket_service.update_ticket(current_user, ticket_key, update_payload, if_unmodified_since=None) -> TicketOut`
- `ticket_service.change_status(current_user, ticket_key, new_status, if_unmodified_since=None) -> TicketOut`

- `ticket_service.list_tickets(filters, page, pageSize) -> Paginated[TicketOut]`
- `ticket_service.get_ticket(ticket_key) -> TicketOut`

Pseudocode (create_ticket)

```
def create_ticket(current_user, payload):
    # validate role and payload
    if current_user.role == 'client' and
payload.get('assigneeId'):
        raise Forbidden("Clients cannot assign tickets")

    # ensure assignee exists if provided
    if payload.assigneeId:
        assignee =
user_repo.get_by_id(payload.assigneeId)
        if not assignee or assignee.status != 'active':
            raise BadRequest('E_ASSIGNEE_NOT_FOUND')

    with db.transaction() as tx:
        seq = tx.execute("SELECT
nextval('ticket_seq')").scalar()
        ticket_key = f"TSK-{1000 + seq}"
        ticket = Ticket(
            ticket_key=ticket_key,
            title=payload.title,
            description=payload.description,
            type=payload.type,
            priority=payload.priority,
            status='open',
            creator_id=current_user.id,
            assignee_id=payload.assigneeId,
            team_id=payload.teamId,
            tags=payload.tags,
            due_date=payload.dueDate
        )
        tx.add(ticket)
        tx.flush()
        tx.add(TicketHistory(ticket_id=ticket.id,
changed_by=current_user.id, action='created',
new_value=payload.dict()))
    # publish event AFTER commit
```

```
    event_bus.publish('ticket.created', {...})
    return TicketOut.from_orm(ticket)
```

Indexing & performance

- Index on `tickets (team_id, status, priority, assignee_id, created_at)` for frequent queries.
- GIN index on `tags` if stored as `text[]` or `jsonb`.
- Consider materialized view for frequently used board queries if listing becomes slow.

Search & filters

- Provide filter options:
  - `status`, `priority`, `assigneeId`, `teamId`, `createdFrom`, `createdTo`, `text` (search in title/description)
- For `text` search, in MVP use `ILIKE` with `%term%`; for scale plan, integrate `pg_trgm` or external search service.

Error handling in backend

- Map DB exceptions to API errors:
  - Unique/constraint violations → `400` with `E_CONSTRAINT_VIOLATION` or `E_TICKET_KEY_CONFLICT`.
  - FK violations → `400` with `E_FK_VIOLATION`.
  - Concurrency conflict detected → `409 E_CONFLICT`.
- Include helpful details: `currentUpdatedAt` to allow client to retry.

Eventual consistency considerations

- Rule engine and notifications run asynchronously — UI should show expected delay (e.g., "Assigning...").

Backend Files

```
backend/app/services/ticket_service.py
backend/app/models/ticket.py
```

```
backend/app/models/ticket_history.py
backend/app/schemas/ticket.py
backend/app/workers/tasks_rules.py
backend/app/workers/tasks_notifications.py
backend/app/utils/event_bus.py  # wrapper for publishing
events (Celery/Redis)
```

3.2.5 Data Layer (Database) — DDL & Models

`ticket_seq` (sequence)

```
CREATE SEQUENCE ticket_seq START 1;
```

`tickets` table

Column Name	Data Type	Constraints / Default	Description / Notes
id	BIGSERIAL	PRIMARY KEY	Auto-incrementing unique ID
ticket_key	VARCHAR(32)	NOT NULL, UNIQUE	Unique ticket identifier
title	VARCHAR(400)	NOT NULL	Title of the ticket
description	TEXT		Detailed description of the ticket
type	VARCHAR(32)	NOT NULL	Type of ticket (e.g., bug, feature, task)
priority	VARCHAR(16)	NOT NULL, DEFAULT 'medium'	Priority of the ticket (low, medium, high)
status	VARCHAR(32)	NOT NULL, DEFAULT 'open'	Current status of the ticket (open, in-progress, closed)
creator_id	BIGINT	NOT NULL, REFERENCES users(id)	User who created the ticket
assignee_id	BIGINT	REFERENCES users(id)	User assigned to work on the ticket
team_id	BIGINT	REFERENCES teams(id)	Team responsible for the ticket
tags	JSONB	DEFAULT '[]'::jsonb	Tags for categorization and filtering
due_date	DATE	NULL	Optional due date for the ticket
created_at	TIMESTAMPTZ	NOT NULL, DEFAULT now()	Timestamp when the ticket was created
updated_at	TIMESTAMPTZ	NOT NULL, DEFAULT now()	Timestamp when the ticket was last updated

Indexes:

Index Name	Columns / Type	Notes

idx_tickets_team_status	team_id, status	Speeds up queries filtering by team & status
idx_tickets_assignee	assignee_id	Speeds up queries by assignee
idx_tickets_created_at	created_at	Speeds up queries sorted/filtered by creation time
idx_tickets_priority	priority	Speeds up queries filtered by priority
idx_tickets_tags_gin	tags (GIN index)	Efficient search/filtering on JSONB tags

`ticket_history` table

Column Name	Data Type	Constraints / Default	Description / Notes
id	BIGSERIAL	PRIMARY KEY	Auto-incrementing unique ID
ticket_id	BIGINT	NOT NULL, REFERENCES tickets(id)	ID of the ticket this history entry belongs to
changed_by	BIGINT	REFERENCES users(id)	ID of the user who made the change
action	VARCHAR(64)	NOT NULL	Type of action (e.g., created, updated, status_change, assigned, comment_added, rule_applied)
old_value	JSONB		Previous state of the ticket data
new_value	JSONB		New state of the ticket data
created_at	TIMESTAMPTZ	NOT NULL, DEFAULT now()	Timestamp when the history record was created

Indexes:

Index Name	Columns	Notes
idx_ticket_history_ticket	ticket_id	Speeds up queries filtering by ticket_id

Optional: `ticket_assignments` (audit of assignments)

If more granular assignment history is needed.

Column Name	Data Type	Constraints / Default	Description / Notes

id	BIGSERIAL	PRIMARY KEY	Auto-incrementing unique ID
ticket_id	BIGINT	REFERENCES tickets(id)	ID of the ticket being reassigned
old_assignee_id	BIGINT		ID of the previous assignee (can be NULL if unassigned previously)
new_assignee_id	BIGINT		ID of the new assignee
changed_by	BIGINT		ID of the user who made the assignment
created_at	TIMESTAMPTZ	DEFAULT now()	Timestamp when the assignment was made

SQLAlchemy models (brief)

`models/ticket.py` maps fields with SQLAlchemy `JSON` for `tags`, use `TIMESTAMP` with `server_default=func.now()`.

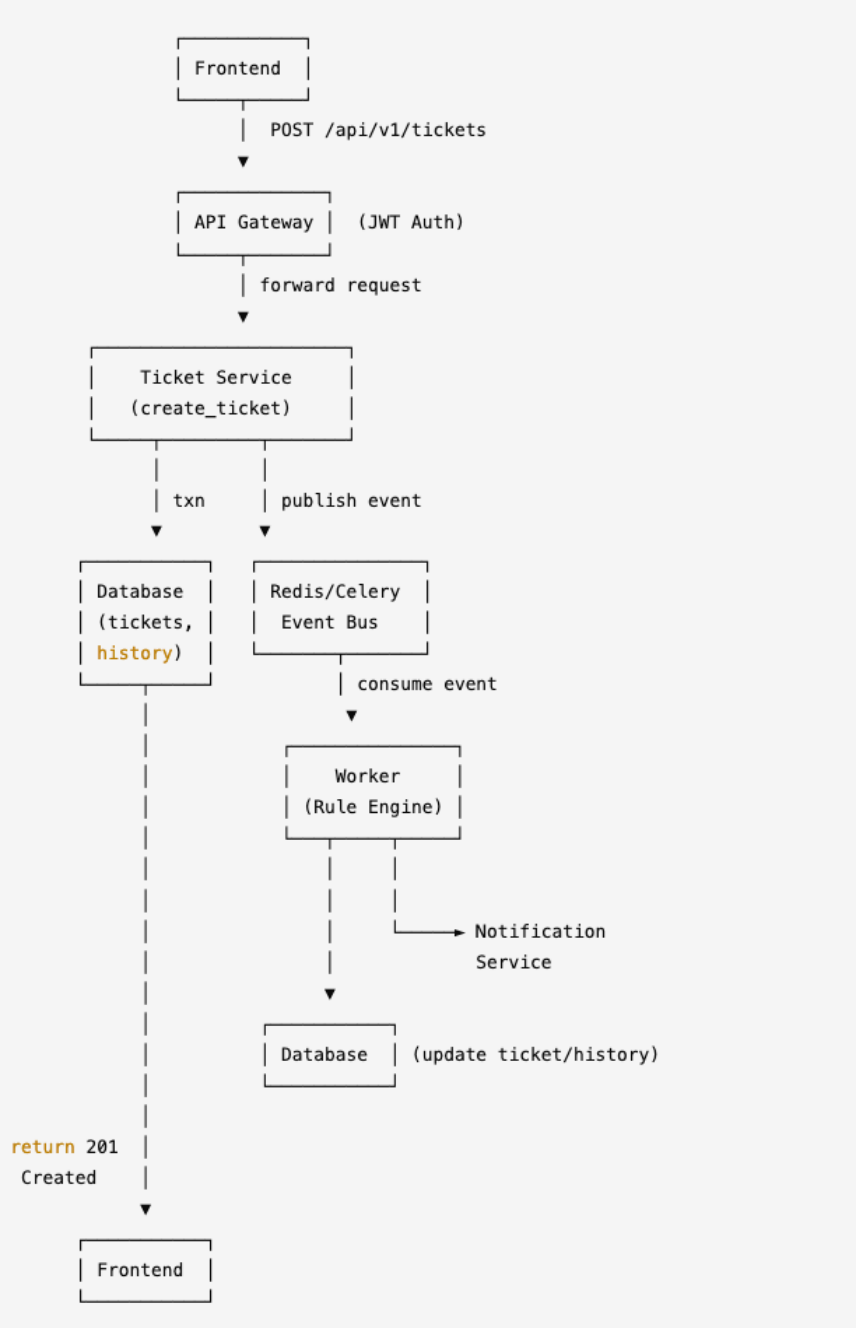
3.2.6

Attachments Interaction (brief cross-link)

Attachments are handled via presigned S3 flow; the attachment metadata is stored in `attachments` table (see 3.5 Collaboration section for full details). Key sequence:

1. `POST /tickets/{ticketKey}/attachments/request-upload` → returns `uploadId` and presigned URL.
2. Frontend PUTs file to S3.
3. `POST /tickets/{ticketKey}/attachments/complete` → backend validates size/type, marks `status='pending_scan'`, enqueues scan, then `available`.

3.2.7 Sequence Flow (Create Ticket) — Step-by-step



3.2.8 Testing & QA (Suggested tests)

Unit tests

- `test_create_ticket_by_client_without_assignee` (clients cannot assign)

- `test_create_ticket_assign_to_nonexistent_user` -> returns 400
- `test_ticket_key_generated_unique` (use sequence or stub)
- `test_status_transition_allowed` (open->in_progress allowed)
- `test_status_transition_disallowed` (open->closed without force -> 422)

Integration tests

- Full create flow integrates DB, then worker consumes `ticket.created` and applies rule (e.g., auto-assign to team).
- Test concurrent updates: simulate two clients updating same ticket with same `updated_at` — second should receive `409`.

E2E tests (frontend)

- Create ticket via UI -> check ticket appears on board.
- Drag & drop the ticket to change status -> backend status changed and history recorded.

Load tests

- Bulk create tickets stress test to verify `ticket_seq` handling and DB write throughput.
- Board listing performance test with 10k tickets in a team (use indexes/ materialized view).

3.2.9 Error Messages & Developer Guidance (mapping)

Condition	API response (HTTP + code)	Message

Missing required field (title)	400 / E_INVALID_PAYLOAD	"Field 'title' is required."
Invalid enum (priority)	400 / E_INVALID_PAYLOAD	"Invalid value for 'priority'."
Client attempted to assign	403 / E_FORBIDDEN	"Clients cannot assign tickets."
Assignee not found	400 / E_ASSIGNEE_NOT_FOUND	"Assignee not found or inactive."
Ticket not found	404 / E_TICKET_NOT_FOUND	"Ticket 'TSK-...' not found."
Invalid status transition	422 / E_INVALID_STATUS_TRANSITION	"Cannot change from 'open' to 'closed'."
Concurrency conflict	409 / E_CONFLICT	"Ticket updated by another user.", include `currentUpdatedAt`
DB constraint (FK) failure	400 / E_FK_VIOLATION	"Referenced resource not found."
Server error	500 / E_INTERNAL	"Unexpected server error."

3.2.11 Acceptance Criteria (developer-ready)

- `POST /api/v1/tickets` creates a ticket and returns full ticket object with `ticketKey`.
- Ticket `ticketKey` format validated and unique.
- Ticket `created_at` and `updated_at` timestamps set and returned in ISO8601 UTC.
- `ticket_history` entry created for `created` action and for subsequent updates/status changes.
- `ticket.created` event published after commit for worker processing.
- Attachments can be added via presigned upload flow (metadata in `attachments` table).
- Optimistic concurrency enforced: `If-Unmodified-Since` or ETag headers yield `409` on conflict.
- Role enforcement: clients cannot assign, managers cannot create admins, etc.
- Tests cover create/update/status-change flows and conflict behavior.

3.3 Basic Project Management (Kanban, Epics, Stories, Sub-tasks)

3.3.1 Presentation Layer (Frontend/UI)

Inputs:

- **Epic creation modal** → `{ title, description, dueDate, teamId }`.
- **Story creation form** → `{ epicId, title, description, assigneeId, priority }`.
- **Sub-task creation inline form** → `{ storyId, title }`.
- **Kanban board interactions**:
  - Drag-and-drop tickets/stories between columns (`Backlog`, `To Do`, `In Progress`, `Done`).
  - Inline edit of titles, assignee, and priority on cards.

Validations (client-side):

- Title: required, 1–200 chars.
- Epic must be linked to a team.
- Story must belong to an Epic.
- Sub-task must belong to a Story.
- Dates: dueDate must not be in the past.
- Kanban drag-drop only allowed for permitted roles (Manager/Admin).

Outputs:

- Board re-renders instantly after drag-drop.
- Created Epic/Story/Sub-task appears immediately (optimistic update).
- Hierarchical view (Epic → Stories → Sub-tasks).
- Card badges: assignee avatar, priority color, due date, status.

Errors (UI messages):

- “Epic not found” (404).

- “Invalid drag-drop target” (422).
- “Permission denied: You cannot update this board” (403).
- “Network/server error. Please retry.” (500).

3.3.2 API Gateway Layer

Endpoints:

- `POST /api/v1/projects/epics` → create epic.
- `GET /api/v1/projects/epics/{id}` → get epic.
- `PUT /api/v1/projects/epics/{id}` → update epic.
- `DELETE /api/v1/projects/epics/{id}` → delete epic.
- `POST /api/v1/projects/stories` → create story.
- `PUT /api/v1/projects/stories/{id}` → update story.
- `POST /api/v1/projects/subtasks` → create sub-task.
- `PUT /api/v1/projects/subtasks/{id}` → update sub-task.
- `GET /api/v1/projects/kanban/{teamId}` → get board.
- `PUT /api/v1/projects/kanban/update` → update card positions.

Inputs (payloads):

- Create Epic: `{ title, description?, teamId, dueDate? }`
- Create Story: `{ epicId, title, description?, assigneeId?, priority? }`
- Create Sub-task: `{ storyId, title, status? }`
- Kanban Update: `{ itemId, fromStatus, toStatus, position }`

Validations:

- JSON schema validation (Pydantic).

- `epicId`, `storyId`, `teamId` must exist in DB.
- Only Managers/Admins can reorder board items.
- Team members can only update their own stories/sub-tasks.

Outputs:

- Standardized responses with objects:

```
{
  "id": 101,
  "title": "Epic - Checkout Refactor",
  "status": "open",
  "dueDate": "2025-10-01"
}
```

- Kanban returns grouped JSON by status:

```
{
  "columns": [
    { "status": "Backlog", "items": [ { "id": 1, "title":
"Story 1" } ] },
    { "status": "In Progress", "items": [ { "id": 2,
"title": "Story 2" } ] }
  ]
}
```

Errors:

- `400 E_INVALID_PAYLOAD` → schema mismatch.
- `403 E_FORBIDDEN` → unauthorized drag-drop.
- `404 E_NOT_FOUND` → epic/story/subtask missing.
- `409 E_CONFLICT` → reorder conflict.

3.3.3 Backend Layer (Business Logic)

Inputs:

- Validated payloads from API Gateway.

- DB records (Epics, Stories, Sub-tasks).

Validations (server-side):

- Epic must belong to an accessible team.
- Story must belong to an existing Epic.
- Sub-task must belong to an existing Story.
- Board reorder must be valid transition (e.g., not moving a "Done" epic back to "Backlog").

Outputs:

- Creates/updates Epics, Stories, Sub-tasks.
- Returns board grouped by status.
- Cascading updates (sub-task → story → epic).

Errors:

- Orphan Story/Sub-task creation → `422`.
- Unauthorized board update → `403`.
- Invalid status transition → `422`.

Business Rules:

- **Cascade closure:**
  - If all sub-tasks → `done`, story auto-transitions → `done`.
  - If all stories in an epic → `done`, epic auto-transitions → `completed`.
- **Status transitions:**
  - Allowed: `backlog → todo → in_progress → done`.
  - Forbidden: skip intermediate states unless Manager/Admin uses force flag.
- **Notifications:**

- On create/update, publish events (`epic.created`, `story.updated`, etc.) → consumed by notifications & workflow automation workers.

3.3.4 Data Layer (Database Models)

`epics` table

Column Name	Data Type	Constraints / Default	Description / Notes
id	BIGSERIAL	PRIMARY KEY	Auto-incrementing unique ID
team_id	BIGINT	NOT NULL, REFERENCES teams(id)	Team responsible for the epic
title	VARCHAR(200)	NOT NULL	Title of the epic
description	TEXT		Detailed description of the epic
status	VARCHAR(32)	NOT NULL, DEFAULT 'open'	Current status of the epic (e.g., open, in-progress, closed)
due_date	DATE		Optional due date for the epic
created_at	TIMESTAMPTZ	DEFAULT now()	Timestamp when the epic was created
updated_at	TIMESTAMPTZ	DEFAULT now()	Timestamp when the epic was last updated

`stories` table

Column Name	Data Type	Constraints / Default	Description / Notes
id	BIGSERIAL	PRIMARY KEY	Auto-incrementing unique ID
epic_id	BIGINT	NOT NULL, REFERENCES epics(id)	ID of the epic this story belongs to
title	VARCHAR(200)	NOT NULL	Title of the story
description	TEXT		Detailed description of the story
priority	VARCHAR(16)	DEFAULT 'medium'	Priority of the story (low, medium, high)
assignee_id	BIGINT	REFERENCES users(id)	User assigned to the story

status	VARCHAR(32)	NOT NULL, DEFAULT 'backlog'	Current status of the story (backlog, in-progress, done, etc.)
created_at	TIMESTAMPTZ	DEFAULT now()	Timestamp when the story was created
updated_at	TIMESTAMPTZ	DEFAULT now()	Timestamp when the story was last updated

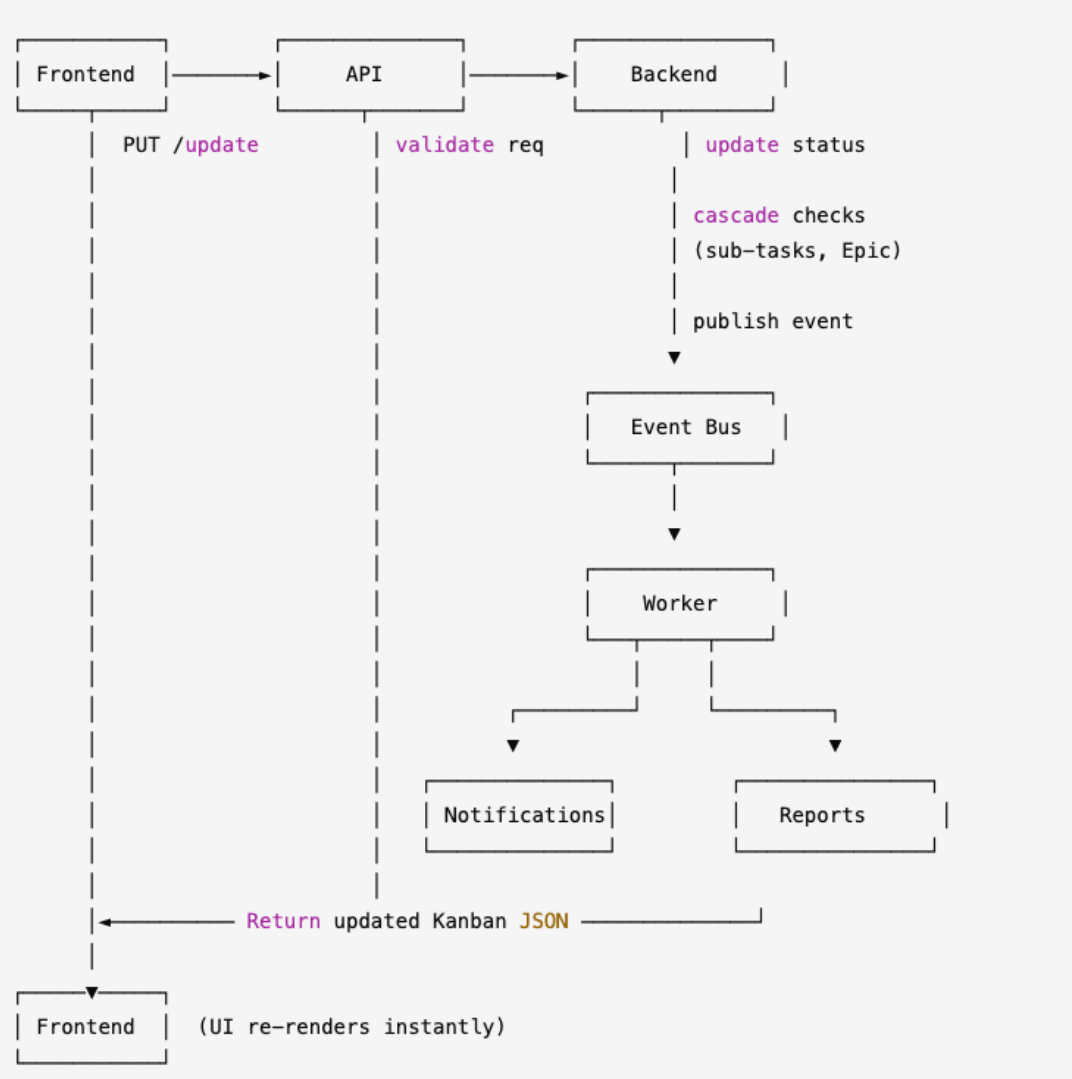
`sub_tasks` table

Column Name	Data Type	Constraints / Default	Description / Notes
id	BIGSERIAL	PRIMARY KEY	Auto-incrementing unique ID
story_id	BIGINT	NOT NULL, REFERENCES stories(id)	ID of the story this sub-task belongs to
title	VARCHAR(200)	NOT NULL	Title of the sub-task
status	VARCHAR(32)	NOT NULL, DEFAULT 'todo'	Status of the sub-task (e.g., todo, in-progress, done)
created_at	TIMESTAMPTZ	DEFAULT now()	Timestamp when the sub-task was created
updated_at	TIMESTAMPTZ	DEFAULT now()	Timestamp when the sub-task was last updated

Indexes

Index Name	Columns	Notes
idx_subtasks_story	story_id	Speeds up queries filtering sub-tasks by story

3.3.5 Sequence Flow — Kanban Drag-Drop



3.3.6 Error Mapping Table

Condition	HTTP Code	Error Code	Message
Epic not found	404	E_EPIC_NOT_FOUND	"Epic does not exist"
Story not found	404	E_STORY_NOT_FOUND	"Story does not exist"
Sub-task not found	404	E_SUBTASK_NOT_FOUND	"Sub-task does not exist"
Unauthorized board update	403	E_FORBIDDEN	"You cannot modify this board"
Invalid drag-drop transition	422	E_INVALID_TRANSITION	"Invalid workflow transition"
Reorder conflict (optimistic lock)	409	E_CONFLICT	"Board updated by another user"

Payload schema mismatch	400	E_INVALID_PAYLOAD	"Invalid input format"

3.3.7 Acceptance Criteria (Dev-Ready)

- Epics, Stories, Sub-tasks can be created, updated, deleted via API & UI.
- Kanban board supports drag-drop, updates persist across reload.
- Cascade closure works (sub-task → story → epic).
- Role enforcement:
  - Manager/Admin can reorder.
  - Team member can only update their items.
- Optimistic concurrency enforced → `409` returned on conflicts.
- Events (`epic.created`, `story.status_changed`) published to worker queue.
- Full audit log maintained in DB (project history table can be added if needed).
- Tests cover:
  - Create/update flows
  - Invalid transitions
  - Cascade closure logic
  - Role-based restrictions

3.4 Workflow Automation

3.4.1 Presentation Layer (Frontend/UI)

Inputs:

- **Workflow Rule Builder UI** (Admin/Manager only):
  - Rule Name

    - Event Trigger (dropdown: `ticket.created`, `ticket.updated`, `status.changed`, `comment.added`)
    - Conditions (key/value rules, e.g., `priority = high`, `status = open`, `tags contains "urgent"`)
    - Actions (dropdown: `assignTo`, `changePriority`, `notifyUser`, `addTag`)
    - Active flag (checkbox).

Validations (client-side):

- Rule name required, ≤ 150 chars.
- At least one condition OR action must exist.
- Trigger must be one of allowed values.
- No empty action payloads (e.g., assign requires a valid userId).

Outputs:

- Workflow rule list (table/grid with status toggle).
- Rule creation success toast.
- Visual confirmation of automation actions applied (audit logs).

Errors (UI messages):

- "Invalid rule configuration" (422).
- "Permission denied" (403).
- "Workflow name already exists" (409).
- Network/server error fallback (500).

3.4.2 API Gateway Layer

Endpoints:

- `POST /api/v1/workflows` → create workflow rule.
- `GET /api/v1/workflows` → list rules (paginated).

- `GET /api/v1/workflows/{id}` → fetch rule detail.
- `PUT /api/v1/workflows/{id}` → update rule.
- `DELETE /api/v1/workflows/{id}` → soft-delete rule.
- `PUT /api/v1/workflows/{id}/toggle` → enable/disable rule.

Inputs (payloads):

- Create Rule:

```
{
  "name": "Auto-assign urgent bugs",
  "trigger": "ticket.created",
  "conditions": [
    { "field": "priority", "operator": "=", "value":
"high" },
    { "field": "type", "operator": "=", "value": "bug" }
  ],
  "actions": [
    { "type": "assignTo", "userId": 17 },
    { "type": "notifyUser", "userId": 42 }
  ],
  "active": true
}
```

Validations:

- JSON schema validation.
- Each condition must specify `{field, operator, value}`.
- Each action must specify `type` and valid parameters.
- Max 10 conditions per rule, 5 actions per rule.

Outputs:

- On success: return stored rule object with `id`.
- On list: return paginated list of rules.

Errors:

- `400 E_INVALID_PAYLOAD` → schema mismatch.

- `403 E_FORBIDDEN` → insufficient role (only Admin/Manager).
- `404 E_WORKFLOW_NOT_FOUND` → rule not found.
- `409 E_DUPLICATE_RULE_NAME` → rule name already exists.

3.4.3 Backend Layer (Business Logic)

Inputs:

- Payloads from API.
- DB records of workflows, users, teams.
- Domain events from Ticket/Project modules (via Celery/Redis).

Validations (server-side):

- Only Admin/Manager can create rules.
- Condition fields must map to known schema fields (`priority`, `status`, `type`, `tags`, etc.).
- Action types must be supported (`assignTo`, `changePriority`, `notifyUser`, `addTag`).
- Rule cycles are prevented (self-referential actions blocked).

Outputs:

- Rule persisted in DB.
- Rules applied asynchronously when matching event arrives.
- Logs in `workflow_history` table.

Errors:

- Invalid condition → reject with `422`.
- User specified in action not found → `400`.
- Circular rule detected → `422`.

Business Logic Rules:

- **Trigger evaluation:**
    - On event (e.g., `ticket.created`), worker fetches active rules for that trigger.
    - Conditions matched using AND logic by default.
- **Action execution:**
    - If rule matches, actions executed in transaction-safe order.
    - Example: Assign user → Update DB → Record history → Publish event.
- **Rule conflicts:**
    - If multiple rules conflict (e.g., two assignTo actions), last-matched wins (configurable).
- **Audit:**
    - Every rule execution logged in `workflow_history`.

3.4.4 Data Layer (Database Models)

`workflow_rules` table

Column Name	Data Type	Constraints / Default	Description / Notes
id	BIGSERIAL	PRIMARY KEY	Auto-incrementing unique ID
name	VARCHAR(150)	NOT NULL, UNIQUE	Unique name of the workflow rule
trigger	VARCHAR(64)	NOT NULL	Trigger event for the workflow (e.g., ticket_created, status_changed)
conditions	JSONB	NOT NULL	JSON object defining conditions for when the rule should execute
actions	JSONB	NOT NULL	JSON object defining actions to take when the rule is triggered
active	BOOLEAN	DEFAULT true	Whether the rule is currently active
created_by	BIGINT	REFERENCES users(id)	ID of the user who created the workflow rule
created_at	TIMESTAMPTZ	DEFAULT now()	Timestamp when the workflow rule was created

updated_at	TIMESTAMPTZ	DEFAULT now()	Timestamp when the workflow rule was last updated

Indexes:

Index Name	Columns	Condition	Notes
idx_workflow_trigger	trigger	WHERE active = true	Optimizes queries filtering by trigger on active rules only

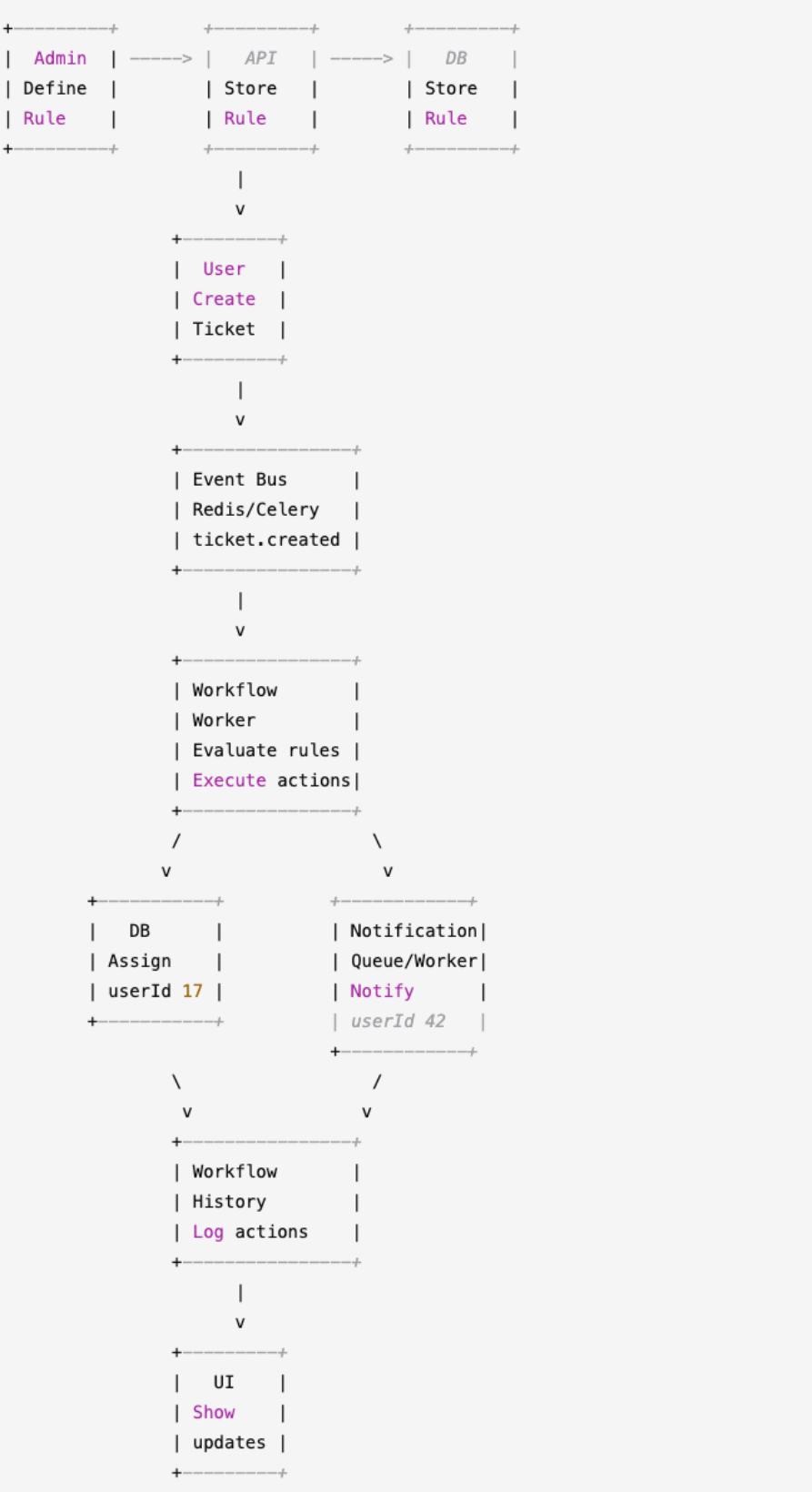
`workflow_history` table

Column Name	Data Type	Constraints / Default	Description / Notes
id	BIGSERIAL	PRIMARY KEY	Auto-incrementing unique ID
workflow_id	BIGINT	REFERENCES workflow_rules(id)	The workflow rule that was triggered
triggered_event	VARCHAR(64)		The event that triggered this workflow execution
matched	BOOLEAN	NOT NULL	Whether the workflow conditions matched (true/false)
executed_actions	JSONB		JSON object describing the actions that were executed
executed_at	TIMESTAMPTZ	DEFAULT now()	Timestamp when the workflow was executed

Indexes:

Index Name	Columns	Notes
idx_workflow_history_workflow	workflow_id	Optimizes queries for fetching workflow execution history per rule

3.4.5 Sequence Flow — Workflow Rule Execution



3.4.6 Error Mapping Table

Condition	HTTP Code	Error Code	Message
Invalid rule payload	400	E_INVALID_PAYLOAD	"Workflow rule payload invalid"
Unauthorized role	403	E_FORBIDDEN	"Only Admin/Manager can manage workflows"
Workflow rule not found	404	E_WORKFLOW_NOT_FOUND	"Workflow rule not found"
Duplicate rule name	409	E_DUPLICATE_RULE_NAME	"Workflow name already exists"
Invalid action parameter	422	E_INVALID_ACTION	"Invalid action configuration"
Condition refers to unknown field	422	E_INVALID_CONDITION	"Condition field not supported"
Workflow caused circular reference	422	E_CIRCULAR_RULE	"Circular dependency detected"

3.4.7 Acceptance Criteria (Dev-Ready)

- Admin/Manager can create workflow rules with triggers, conditions, and actions.
- Rules stored in `workflow_rules` with JSONB fields.
- Rules are evaluated **asynchronously** when matching events occur.
- Matching rules execute all actions in safe order.
- Rule executions logged in `workflow_history`.
- RBAC enforced (only Admin/Manager can manage workflows).
- Validation ensures conditions and actions are valid.
- Optimistic concurrency on workflow updates (update conflict returns `409`).
- Tests:
    - Create valid/invalid rules.
    - Trigger workflow and confirm actions executed.
    - Ensure disabled rules do not execute.
    - Detect and block circular workflows.

3.5 Collaboration & Communication

3.5.1 Presentation Layer (Frontend/UI)

Inputs:

- **Comment Box (Rich text editor)**:
  - Input: `{ content, mentions[], attachments[] }`
  - Support for `@mentions` of users.
  - Inline image/file attachments (max 20 MB).
- **Attachment Upload UI**:
  - Drag-and-drop file upload.
  - Shows upload progress.
- **Notification Panel**:
  - Real-time in-app notifications (via WebSocket or polling fallback).
  - Filter notifications by type (ticket update, mention, rule, system).

Validations (Client-Side):

- Comment content: max 2000 chars.
- Mentions: must be valid user IDs.
- Attachments: allowed MIME types only (`png`, `jpg`, `pdf`, `docx`).
- Max 5 attachments per comment.
- File size limit: 20 MB each.

Outputs (UI):

- Comment immediately displayed (optimistic update).
- Mentions highlighted (e.g., `@username`).
- Attachments preview thumbnails.

- Notification badge increments when new notification arrives.

Errors (UI Messages):

- "Attachment exceeds 20MB limit."
- "Invalid file type."
- "Comment too long."
- "Server error, please retry."

3.5.2 API Gateway Layer

Endpoints:

Comments:

- `POST /api/v1/tickets/{ticketKey}/comments` → add comment.
- `GET /api/v1/tickets/{ticketKey}/comments` → list comments.
- `DELETE /api/v1/comments/{id}` → soft-delete comment (Admin/Manager).

Attachments:

- `POST /api/v1/attachments/request-upload` → get presigned S3 URL.
- `POST /api/v1/attachments/complete` → finalize upload.
- `GET /api/v1/attachments/{id}` → fetch metadata (URL).

Notifications:

- `GET /api/v1/notifications` → list notifications (paginated).
- `PUT /api/v1/notifications/{id}/read` → mark as read.
- `PUT /api/v1/notifications/read-all` → mark all as read.

Inputs (Payloads):

Add Comment Request:

```
{
  "content": "Please fix this ASAP @john",
  "mentions": [12],
  "attachments": [101, 102]
}
```

Request Upload (Attachment):

```
{
  "filename": "error_log.pdf",
  "contentType": "application/pdf"
}
```

Validations:

- Comment content non-empty.
- Mentions must exist in users table.
- Attachments must exist and belong to ticket.
- Notification pagination params: `page`, `pageSize` (default 25, max 100).

Outputs:

- Comment response:

```
{
  "id": 5001,
  "ticketId": 11024,
  "authorId": 42,
  "content": "Please fix this ASAP @john",
  "mentions": [12],
  "attachments": [
    { "id": 101, "url": "https://s3/.../error_log.pdf" }
  ],
  "createdAt": "2025-09-22T11:00:00Z"
}
```

- Presigned URL response:

```
{
  "uploadId": "uuid",
  "url": "https://s3.bucket/uuid?signature=...",
```

```
  "attachmentId": 101
}
```

- Notifications list:

```
{
  "items": [
    { "id": 701, "type": "mention", "message": "You were
mentioned in TSK-11024", "read": false }
  ],
  "meta": { "page": 1, "pageSize": 25, "total": 120 }
}
```

Errors:

- `400 E_INVALID_PAYLOAD` — invalid JSON.
- `403 E_FORBIDDEN` — unauthorized to comment/delete.
- `404 E_ATTACHMENT_NOT_FOUND` — attachment missing.
- `413 E_ATTACHMENT_TOO_LARGE` — file exceeds size limit.

3.5.3 Backend Layer (Business Logic)

Inputs:

- Validated payloads from API.
- DB lookups for users, tickets, attachments.
- Event queue for notifications.

Validations:

- Comment content sanitized (prevent XSS).
- Mentions resolved against users table.
- Attachments validated (exist + size/type checks).
- User must have access to ticket/team.

Outputs:

- Comment persisted.

- Mentions resolved to notifications.
- Attachments linked to ticket/comment.
- Notifications inserted + events published.

Errors:

- User without ticket access → `403`.
- Invalid attachment reference → `400`.
- Notification delivery failure → logged + retried (async).

Business Logic Rules:

- Mentions: Each mentioned user receives `mention` notification.
- Comment events: `comment.created` published to workers.
- Attachments virus scanned asynchronously (mark `status=clean`).
- Notifications:
  - In-app inserted into `notifications` table.
  - Email notifications queued via Celery.
- Auto-cleanup: deleted comments remain soft-deleted for audit.

3.5.4 Data Layer (Database Models)

`comments` table

Column Name	Data Type	Constraints / Default	Description / Notes
id	BIGSERIAL	PRIMARY KEY	Auto-incrementing unique ID
ticket_id	BIGINT	NOT NULL, REFERENCES tickets(id)	The ticket this comment belongs to
author_id	BIGINT	NOT NULL, REFERENCES users(id)	The user who wrote the comment
content	TEXT	NOT NULL	The actual comment text
mentions	JSONB	DEFAULT '[]'::jsonb	List of mentioned users (e.g., `["@user1", "@user2"]`)

created_at	TIMESTAMPTZ	DEFAULT now()	Timestamp when the comment was created
updated_at	TIMESTAMPTZ	DEFAULT now()	Timestamp when the comment was last updated
deleted	BOOLEAN	DEFAULT false	Soft delete flag (true = comment is deleted, but still stored for history/ audit)

Indexes:

Index Name	Columns	Notes
idx_comments_ticket	ticket_id	Optimizes queries to fetch comments for a specific ticket

`attachments` table

Column Name	Data Type	Constraints / Default	Description / Notes
id	BIGSERIAL	PRIMARY KEY	Auto-incrementing unique ID
ticket_id	BIGINT	REFERENCES tickets(id)	The ticket this attachment is linked to
filename	VARCHAR(255)	NOT NULL	Original file name of the uploaded attachment
content_type	VARCHAR(100)	NOT NULL	MIME type of the file (e.g., `image/png`, `application/pdf`)
size	BIGINT	NOT NULL	File size in bytes
storage_url	TEXT	NOT NULL	Location of the file in storage (e.g., S3, GCS, local path)
status	VARCHAR(32)	DEFAULT 'pending'	Attachment status: `pending`, `clean`, `infected`
uploaded_by	BIGINT	REFERENCES users(id)	The user who uploaded the file
created_at	TIMESTAMPTZ	DEFAULT now()	Timestamp when the file was uploaded

Indexes:

Index Name	Columns	Notes
idx_attachments_ticket	ticket_id	Optimizes queries to fetch all attachments for a ticket

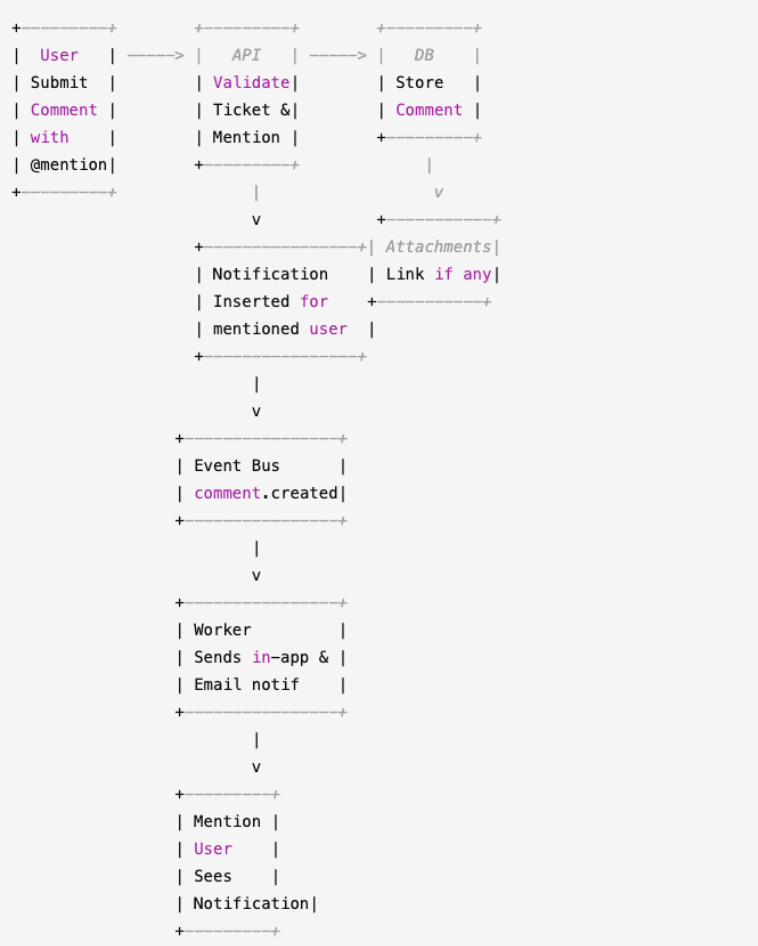
`notifications` table

id	BIGSERIAL	PRIMARY KEY	Auto-incrementing unique notification ID
user_id	BIGINT	NOT NULL, REFERENCES users(id)	The recipient user of the notification
type	VARCHAR(64)	NOT NULL	Type/category of notification (e.g., `comment`, `assignment`, `workflow`)
message	TEXT	NOT NULL	Notification content/message text
read	BOOLEAN	DEFAULT false	Whether the notification has been read by the user
created_at	TIMESTAMPTZ	DEFAULT now()	Timestamp when the notification was created

Indexes:

Index Name	Columns	Notes
idx_notifications_user	user_id	Optimizes queries to fetch all notifications for a user

3.5.5 Sequence Flow — Mention in Comment



3.5.6 Error Mapping Table

Condition	HTTP Code	Error Code	Message
Invalid comment payload	400	E_INVALID_PAYLOAD	"Invalid comment format"
Ticket not found	404	E_TICKET_NOT_FOUND	"Ticket not found"
Unauthorized to comment	403	E_FORBIDDEN	"You cannot comment on this ticket"
Mention user not found	400	E_INVALID_MENTION	"Mentioned user not found"
Attachment too large	413	E_ATTACHMENT_TOO_LARGE	"File exceeds size limit"
Attachment type not allowed	422	E_INVALID_ATTACHMENT	"Invalid file type"
Notification user not found	404	E_NOTIFICATION_NOT_FOUND	"Notification recipient not found"
Server error	500	E_INTERNAL	"Unexpected server error"

3.5.7 Acceptance Criteria (Dev-Ready)

- Users can add, edit, and delete comments on tickets.
- Mentions generate notifications (in-app + email).
- Attachments uploaded via presigned S3 URLs, virus scanned before `status=clean`.
- Notifications appear in-app in real-time, with read/unread toggle.
- Role enforcement:
  - Team members can comment on their team's tickets.
  - Clients can only comment on their own tickets.
  - Admin/Manager can delete comments.
- Audit: Deleted comments remain soft-deleted (`deleted=true`).
- Tests:
  - Add comment with mention → notification created.
  - Invalid mention → rejected.
  - Upload invalid file → rejected.

- Attachment virus scan sets status correctly.
- Notifications pagination works.

3.6 Reporting & Analytics — Complete, developer-ready design

Below is the exhaustive LLD for **Reporting & Analytics**. It covers UI, API, backend processing, data model (DDL), sequence flows, error mapping, performance/caching strategy, export mechanics, test checklist, and exact file/directory mapping so developers can implement without ambiguity.

3.6.1 Overview / Goals

Purpose: Provide dashboards and exportable reports that give managers and admins actionable insights: ticket counts by status/priority/team, SLA/resolution metrics, workload per assignee, trends over time, and a CSV/XLSX/PDF export pipeline for ad-hoc and scheduled reports.

Non-functional targets (measurable):

- Dashboard API median latency ≤ 500 ms for typical queries (with caching/materialized views).
- Export pipeline handles large exports (100k rows) via streaming/worker with no blocking of API.
- Report generation failures < 0.5% (retries + logging).

Supported report types (MVP):

- Dashboard summary (KPIs): open tickets, in_progress, resolved, average resolution time last 30/90 days.
- SLA breaches report (tickets exceeding SLA target).
- Tickets by priority/team/assignee.
- Time-to-resolution histogram.
- Export formats: CSV, XLSX, PDF (PDF generated from HTML/PDF engine; async for large sets).

3.6.2 Presentation Layer (Frontend / UI)

Pages & Components

- `ReportsDashboard` — KPI cards, charts (bar, line), filters.
- `ReportsExplorer` — list of saved/scheduled reports, run now.
- `ReportFilterPanel` — date range, team, assignee, priority, status, tags.
- `ExportModal` — choose format, email when ready toggle, synchronous/async notice.
- `ReportHistory` — status of previously requested exports, download link.

Inputs (user interactions & fields)

- Filters:
  - `dateFrom` (ISO date), `dateTo`, `teamId`, `assigneeId`, `priority[]`, `status[]`, `tags[]`.
- Aggregation options:
  - Time granularity: `daily|weekly|monthly`.
  - Group by: `status|assignee|team|priority`.
- Export:
  - `format` : `csv | xlsx | pdf`
  - `columns` : list of fields (select columns to include)
  - `async` flag (frontend indicates if dataset larger than threshold will force async)
  - `emailOnReady` boolean
- Scheduling (future, optional): cron expression / daily/weekly preset (not required for MVP but UI should reserve space)

Client-side validations

- `dateFrom` ≤ `dateTo`.

- Date range maximum for synchronous export = 31 days (configurable). Larger ranges switch to async and return 202.
- `format` within allowed set.
- `columns` subset of allowed fields (frontend provides checkboxes).
- Filters must be valid ids (prefetch lists for teams/assignees).

Outputs (UI)

- KPI cards (counts): total tickets, open, in_progress, avg resolution time (hrs).
- Time series chart: tickets created/resolved over time.
- Table with pagination for report explorer.
- Export dialog shows progress and download link when ready. For async exports, show reportId and poll or show push notification.

Error & UX messaging

- Validation error inline: "Start date must be earlier than end date."
- Synchronous export too large: "Export too large — generating in background; we'll email you the link."
- Export failed: "Report generation failed — retry or check logs (support)."
- Not authorized: "You don't have permission to view this report."

Frontend file mapping

```
frontend/src/pages/reports/Dashboard.tsx
frontend/src/pages/reports/Explorer.tsx
frontend/src/components/Reports/FilterPanel.tsx
frontend/src/components/Reports/ExportModal.tsx
frontend/src/api/reports.ts
frontend/src/hooks/useReports.ts
```

3.6.3 API Layer (Gateway): Endpoints & Contracts

Endpoints (MVP)

- `GET /api/v1/reports/dashboard` — returns KPI aggregates and charts data.
- `GET /api/v1/reports/metrics` — raw metrics for custom charts (accepts filters).
- `GET /api/v1/reports/export` — synchronous export (small), query params include filters + `format`.
- `POST /api/v1/reports/export` — async export request (preferred for large sets), body contains filters, columns, format and `emailOnReady` flag → returns `202 Accepted` + `reportId`.
- `GET /api/v1/reports/status/{reportId}` — check status + download URL when ready.
- `GET /api/v1/reports/history` — list of user's past exports.
- `POST /api/v1/reports/schedule` — create scheduled report (future feature stub allowed).
- `DELETE /api/v1/reports/{reportId}` — cancel/delete report (if pending).

Request/Response Examples

Dashboard request

```
GET /api/v1/reports/dashboard?
teamId=3&dateFrom=2025-09-01&dateTo=2025-09-22
Authorization: Bearer <token>
```

Dashboard response (200)

```
{
  "kpis": {
    "totalTickets": 3452,
    "open": 210,
    "inProgress": 78,
    "resolvedLast30Days": 1820,
    "avgResolutionHours": 26.7
  },
  "timeSeries": {
```

```
    "labels": ["2025-09-01","2025-09-02",...],
    "created": [12,8,...],
    "resolved": [10,14,...]
  },
  "byPriority": {"high": 560, "medium": 1800, "low":
1092}
}
```

Async export request

```
POST /api/v1/reports/export
Authorization: Bearer <token>
Content-Type: application/json

{
  "filters": { "dateFrom": "2025-01-01", "dateTo":
"2025-09-22", "teamId": 3 },
  "format": "xlsx",
  "columns":
["ticketKey","title","status","priority","assignee","crea
tedAt","resolvedAt"],
  "emailOnReady": true
}
```

Accepted response (202)

```
{
  "reportId": "rpt_3f2a9d",
  "status": "pending",
  "message": "Report queued. You will be notified when
it's ready."
}
```

Status when ready (200)

```
{
  "reportId": "rpt_3f2a9d",
  "status": "ready",
  "downloadUrl": "https://s3.example.com/reports/
rpt_3f2a9d.xlsx?sig=...",
  "expiresAt": "2025-10-06T10:00:00Z"
}
```

API validations & rate-limiting

- Validate filters and date formats.
- Enforce role-based access: Managers can request team reports; Admin can request org-wide.
- Synchronous export size limit (e.g., max rows 50k) — if exceeded return `413` or suggest async via `202`.
- Rate-limit `POST /reports/export` per user (e.g., 5 exports per hour) to avoid abuse.

API Error codes (examples)

- `400 E_INVALID_PARAMS` — invalid filters or columns.
- `401 E_UNAUTHORIZED`
- `403 E_FORBIDDEN`
- `413 E_EXPORT_TOO_LARGE`
- `429 E_RATE_LIMIT`
- `500 E_INTERNAL`

API files

```
backend/app/api/v1/routes_reports.py
backend/app/schemas/report.py
backend/app/api/deps.py  # role checks, pagination
```

3.6.4 Backend Layer (Business Logic)

Responsibilities

- Serve KPI and chart aggregations efficiently.
- Decide sync vs async for export requests.
- For async exports: persist request, enqueue Celery task `generate_report`.
- Worker streams DB query, writes CSV/XLSX, uploads to S3, updates `reports` table, notifies user.

- Support pagination and filtered queries for `GET /reports/metrics` (for frontend charts/table).
- Use caching and materialized views for expensive aggregates.

Implementation Details

Dashboard queries

- Use pre-aggregated / materialized views for heavy aggregates:
  - `tickets_aggregates_by_day(team_id, date)` materialized view updated every 5 minutes via scheduled job or incremental update.
- Fallback: on-the-fly SQL with indexes for small queries.

Materialized view example (concept)

```
CREATE MATERIALIZED VIEW mv_ticket_counts AS
SELECT team_id, status, date_trunc('day', created_at) AS
day, COUNT(*) AS count
FROM tickets
GROUP BY team_id, status, date_trunc('day', created_at);
-- Refresh mechanism: REFRESH MATERIALIZED VIEW
CONCURRENTLY mv_ticket_counts;
```

Export decision logic

- Estimate row count using `COUNT(*)` with same filters:
  - If `count <= SYNC_LIMIT` (e.g., 5000 rows), generate synchronously and return file stream.
  - Else enqueue worker and return `202` with `reportId`.

Generate report worker (outline)

- Task `generate_report(reportId)`:
  1. Lock `reports` row; set `status='running'`.
  2. Use server-side cursor (psycopg2 named cursor) to stream rows.
  3. Write to temp file in streaming fashion (CSV using `csv.writer`, XLSX using `openpyxl` streaming writer or `xlsxwriter` with streaming).

4. Upload file to S3 with server-side encryption and set ACL/private.
5. Update `reports` with `status='ready'`, `s3_key`, `completed_at`.
6. Create `notifications` row and send email if `emailOnReady`.
7. On failure, set `status='failed'` and add `error_message`, create retry/job logs.

Security

- Pre-signed S3 download URLs expire (configurable; e.g., 7 days).
- Files stored for limited retention (e.g., 30 days) with background cleanup job.

Performance & caching

- Cache common dashboard results in Redis (TTL 60–300s).
- For heavy dynamic filters, use DB indexes and avoid full-table scans.
- Use connection pooling for large exports and ensure worker memory usage is controlled (streaming).

Logging & monitoring

- Track job metrics: duration, bytes written, rows processed, failures.
- Expose metrics to Prometheus: `report_jobs_total`, `report_jobs_failed`, `report_job_duration_seconds`.

Edge cases

- Export generation should be idempotent — if worker restarts, check if `s3_key` already present and `status='ready'`.
- If upload fails, retry with exponential backoff (Celery retries).

Backend file mapping

```
backend/app/services/report_service.py
backend/app/workers/tasks_reports.py
backend/app/models/report.py
backend/app/db/materialized_views.sql
```

```
backend/app/utils/csv_streamer.py
backend/app/utils/xlsx_streamer.py
backend/app/utils/s3.py
```

3.6.5 Data Layer (Database)

`reports` table (track export jobs)

Column Name	Data Type	Constraints / Default	Description / Notes
id	BIGSERIAL	PRIMARY KEY	Auto-incrementing unique row ID
report_id	UUID	NOT NULL, UNIQUE	External-facing unique identifier for the report (useful for APIs)
created_by	BIGINT	REFERENCES users(id)	User who requested/generated the report
filters	JSONB		Stores report filters (dynamic key-value pairs, e.g., `{ "team_id": 5, "date_range": "last_30_days" }`)
columns	JSONB		Stores which columns/fields are included in the report
format	VARCHAR(16)	NOT NULL	Report format (`csv`, `xlsx`, `pdf`)
status	VARCHAR(32)	NOT NULL DEFAULT 'pending'	Report state: `pending`, `running`, `ready`, `failed`, `canceled`
s3_key	TEXT		Location of the generated report file in storage (e.g., AWS S3 path)
error_message	TEXT		Error details if report generation fails
email_on_ready	BOOLEAN	DEFAULT false	Whether to email user when the report is ready
created_at	TIMESTAMPTZ	DEFAULT now()	Timestamp when report request was created
completed_at	TIMESTAMPTZ		Timestamp when report finished (success/failure)

Indexes:

Index Name	Column	Notes
idx_reports_status	status	Optimizes queries by report status (e.g., fetching `pending` or `running` jobs)

idx_reports_created_by	created_by	Speeds up queries fetching reports by user

Materialized view: `mv_ticket_aggregates` (example)

Column Name	Data Type	Description
team_id	BIGINT	The team to which the ticket belongs (`tickets.team_id`)
day	DATE	The day (derived from `created_at` via `date_trunc`) when tickets were created
status	VARCHAR	The current status of tickets (`open`, `in_progress`, `resolved`, etc.)
cnt	BIGINT	Count of tickets created for the team on that day with this status
avg_resolution_hours	DOUBLE PRECISION	Average resolution time (in hours), computed as `resolved_at - created_at`(NULL if unresolved)

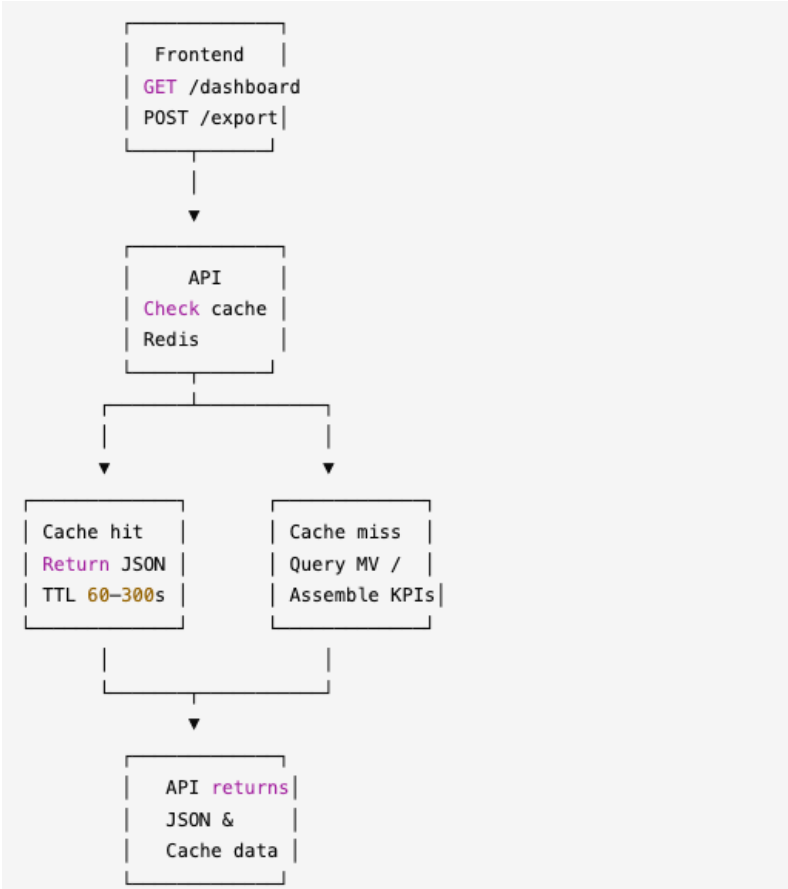
Notes

- `resolved_at` field must exist on `tickets` (nullable) for resolution time metrics.
- Refresh interval and method must be configured (concurrent refresh to avoid locks).

Index suggestions

- `tickets(created_at)`, `tickets(status)`, `tickets(team_id)`, `tickets(assignee_id)` for filter speed.
- GIN index on `tickets.tags` (`jsonb` or `text[]`) if tags are used heavily.

3.6.6 Sequence Flows



3.6.7 Error Mapping Table

Condition	HTTP	Code	Message
Invalid filter or columns	400	E_INVALID_PARAMS	"Invalid report parameters"
Not authorized for team/org	403	E_FORBIDDEN	"Not allowed to run this report"
Export too large for sync	413	E_EXPORT_TOO_LARGE	"Export too large; use async"
Export queued	202	E_QUEUED	"Report queued; you will be notified"
Export failed (worker)	500	E_EXPORT_FAILED	"Report generation failed"
Report not found	404	E_NOT_FOUND	"Report not found"

All responses include `X-Request-ID` for traceability.

3.6.8 Acceptance Criteria & Tests (Dev-Ready)

Acceptance

- Dashboard endpoint returns KPIs and time-series within SLA (median latency target).
- Small synchronous export works and returns file stream (CSV/XLSX) with requested columns.
- Large export enqueues job and returns `202` with `reportId`.
- Worker generates export, uploads to S3, sets `status='ready'`, and notification/email sent.
- Download link is a pre-signed URL and valid for configured TTL (e.g., 7 days).
- Materialized view refresh mechanism exists and is scheduled or triggered.
- Reports older than retention (e.g., 30 days) are cleaned up by job.

Unit tests

- `test_dashboard_aggregates_cached` — cache hit vs miss.
- `test_export_sync_small` — small dataset exported synchronously.

- `test_export_async_enqueued` — large dataset returns 202 and creates `reports` row.
- `test_worker_generates_report` — worker writes file, s3 upload simulated, DB updated.
- `test_export_filters_validation` — invalid dates rejected.

Integration tests

- Full end-to-end: request export async → worker runs → get status → retrieve pre-signed URL and download contents match filters.

Load tests

- Generate performance test for dashboard queries with production-like dataset (hundreds of thousands tickets) to confirm indexing/materialized view effectiveness.

3.6.9 Operational Considerations

- **Monitoring**: expose Prometheus metrics for report job durations, queue sizes, success/failure counts.
- **Alerting**: alert on high failure rates or unusually long job durations (> 2 hours).
- **Retention policy**:
    - Keep generated files in S3 for 30 days; set lifecycle policy to delete after retention.
    - Keep `reports` DB rows for 90 days; mark older rows archived/ removed depending on policy.
- **Security**: generated exports may contain PII; ensure S3 bucket encryption and limited access. Use server-side encryption and ACLs.
- **Costs**: large exports incur S3/storage and egress; consider compressing CSV/ XLSX or store zipped files.
- **Backups**: backup materialized views not necessary; backups focus on base tables (tickets, ticket_history).

3.6.10 Sample SQL Queries (for implementation & tests)

Tickets created per day (filter team)

```
SELECT date_trunc('day', created_at)::date AS day,
COUNT(*) AS created_count
FROM tickets
WHERE team_id = :team_id AND created_at
BETWEEN :date_from AND :date_to
GROUP BY day ORDER BY day;
```

Average resolution time by team

```
SELECT team_id,
  AVG(EXTRACT(epoch FROM (resolved_at - created_at))/
3600) AS avg_resolution_hours
FROM tickets
WHERE resolved_at IS NOT NULL
  AND created_at BETWEEN :date_from AND :date_to
GROUP BY team_id;
```

SLA breaches

```
SELECT ticket_key, creator_id, assignee_id, priority,
created_at, resolved_at,
  EXTRACT(epoch FROM (COALESCE(resolved_at, now()) -
created_at))/3600 AS hours_open
FROM tickets
WHERE EXTRACT(epoch FROM (COALESCE(resolved_at, now()) -
created_at))/3600 > :sla_hours
  AND created_at BETWEEN :date_from AND :date_to;
```

4. External Interface Requirements

4.1 User Interfaces (UI/UX Guidelines)

Design Principles

- **Consistency**:
  - Use a unified design system (e.g., Tailwind + shadcn/ui on frontend).
  - Standardized component library: inputs, modals, tables, charts, notifications.
- **Responsiveness**:
  - Must work on desktop (primary) and adapt to tablets.
  - Mobile experience: simplified but functional (view-only for some dashboards).
- **Accessibility**:
  - WCAG 2.1 AA compliance baseline.
  - Contrast ratios for text/icons.
  - Keyboard navigability.
  - ARIA labels for forms, charts, buttons.
- **Localization (Future-proofing)**:
  - Text strings stored in i18n files.
  - Date/time formatted via locale.

UI Components Used Across System

- **Navigation**:
  - Sidebar: `Dashboard`, `Tickets`, `Projects`, `Reports`, `Admin`.
  - Breadcrumbs on detail pages.
- **Form Components**:
  - Text input, dropdown, autocomplete, date picker, file upload.
  - Validation messages displayed inline (e.g., under field).
- **Charts & Data Visualization**:
  - Recharts / Chart.js with color-blind-friendly palettes.

    - Loading skeletons while data is fetched.

- **Feedback & Notifications**:
    - Snackbar/toast for success/error.
    - Notification bell for in-app events.

4.2 Software Interfaces (APIs, Microservices, Gateways)

API Gateway

- **Protocol**: REST + JSON.
- **Auth**: JWT (RS256), bearer tokens in `Authorization` header.
- **Versioning**: `/api/v1/...`.
- **Pagination**: `?page=1&pageSize=25` (max 100).
- **Filtering**: query params (`?status=open&priority=high`).
- **Sorting**: `?sort=createdAt:desc`.
- **Errors**: JSON structured, standard format:

```
{
  "error": {
    "code": "E_INVALID_PAYLOAD",
    "message": "Invalid request body",
    "details": { "field": "title" }
  }
}
```

Microservices / External Integrations

- **Email (SMTP / SendGrid / Outlook 365)**
    - Protocol: SMTP with TLS.
    - Used for notifications and report emails.
    - Configurable sender domains.
- **File Storage (AWS S3 / Azure Blob)**

    - Presigned URL upload + download.
    - Server-side encryption enabled.
    - Lifecycle rules (delete after 30 days for exports).
- **Authentication Provider (OIDC / Azure AD / Google Workspace)**
    - Support SSO integration with OAuth2/OpenID Connect.
    - JWT tokens validated at API gateway.
- **WebSocket Service**
    - Used for real-time notifications (ticket updates, mentions).
    - Fallback: long-polling.
    - Protocol: WSS, token-authenticated.

4.3 Communications Interfaces (SMTP, Notifications)

Email

- **Transport**:
    - Primary: SMTP relay (e.g., Microsoft 365, SendGrid).
    - Fallback: Local postfix relay (dev/test only).
- **Formats**:
    - HTML + plain text fallback.
    - Templates stored in backend `/templates/email/`.
- **Types of Emails**:
    - New ticket assigned.
    - Mentioned in a comment.
    - SLA breach alert.
    - Report ready for download.

- **Retry**:
    - If mail server unreachable, retry with exponential backoff (via Celery).

In-App Notifications

- Stored in `notifications` table.
- Displayed in UI bell + optional email.
- Delivered via WebSocket.
- Mark as read/unread via API.

Push Notifications (Future scope)

- Mobile/web push via Firebase or VAPID.
- Not MVP, but directory structure reserved.

5. Non-Functional Requirements & Design Strategy

This section details performance, security, reliability, usability, and maintainability strategies. Each sub-section is expanded for **developer-ready guidance** with metrics, design decisions, and implementation notes.

5.1 Performance

Goals:

- Ensure low-latency responses for APIs and dashboards.
- Support concurrent users without degradation.
- Efficient data processing for large exports, reports, and dashboards.

Metrics / Targets:

- API median latency ≤ 200 ms for CRUD operations (ticket/comment management).
- Dashboard query median latency ≤ 500 ms for cached queries.

- Asynchronous export jobs can handle datasets up to 100k rows without blocking API.
- WebSocket notifications latency < 2 seconds from event trigger to client receipt.

Implementation Strategy:

- **Caching:** Use Redis for frequently accessed KPIs, ticket counts, and reference data (teams, users, priorities).
- **Database Optimization:**
  - Indexing on high-query columns (tickets: `status`, `priority`, `team_id`, `assignee_id`).
  - Materialized views for heavy aggregate queries (e.g., reporting KPIs).
  - Use server-side cursors for large exports to reduce memory consumption.
- **Async Processing:**
  - Celery or similar worker queue for report generation, notifications, and attachment virus scanning.
- **Load Balancing:**
  - Horizontal scaling via multiple API instances behind a load balancer.
- **Frontend Optimization:**
  - Debounce search/filter inputs.
  - Virtualized tables for large datasets.

5.2 Security

Goals:

- Protect sensitive ticket data, user information, and attachments.
- Ensure authentication, authorization, and data integrity.

Implementation Strategy:

- **Authentication & Authorization:**
  - JWT tokens for API access.
  - Role-based access control (RBAC) for features: Admin, Manager, Team Member, Client.
- **Data Protection:**
  - TLS/HTTPS for all client-server communications.
  - Server-side encryption for attachments and report exports (AWS S3 SSE-S3/SSE-KMS).
- **Input Validation & Sanitization:**
  - Prevent XSS and SQL injection on forms, comments, and reports.
  - Escape/encode HTML content in comments and tickets.
- **Audit Logging:**
  - Log user activity: login, ticket edits, comment actions, report generation.
  - Immutable audit tables for sensitive operations.
- **Error Handling:**
  - Do not expose stack traces in production.
  - Use standard error responses (JSON) with codes for client handling.

Security Policies:

- Password policies if local authentication used (min 12 characters, complexity rules).
- Optional 2FA for admins/managers.
- Periodic access token expiration (e.g., 1 hour), refresh via refresh token.

5.3 Reliability & Availability

Goals:

- Ensure system uptime ≥ 99.9%.

- Minimize service disruptions during peak usage.

Strategies:

- **Redundancy:** Multiple API and worker instances across availability zones.
- **Health Checks & Auto-Recovery:**
    - API and worker health endpoints monitored by orchestration platform (Kubernetes, ECS, or similar).
    - Auto-restart unhealthy pods/services.
- **Database Replication:**
    - Primary + replicas for read-heavy workloads (e.g., reporting dashboards).
- **Retry Mechanisms:**
    - Exponential backoff for transient errors (API calls, email notifications, file uploads).
- **Backups:**
    - Nightly DB snapshots, daily incremental backups.
    - Attachment & export storage backup policies.
- **Monitoring & Alerting:**
    - Prometheus + Grafana dashboards for system metrics: API latency, job queue length, error rates, DB performance.

5.4 Usability

Goals:

- Ensure intuitive navigation and task efficiency.
- Make reporting, ticketing, and collaboration straightforward.

Strategies:

- **Consistent Design System:**

    - Unified colors, spacing, fonts, icons (lucide-react, react-icons).
- **Responsive UI:**
    - Functional across desktop and tablet devices.
- **Inline Feedback:**
    - Real-time validation messages, loading spinners, success notifications.
- **Accessibility:**
    - Keyboard navigable, ARIA labels, contrast ratios for readability.
- **Search & Filter Optimization:**
    - Quick search in tickets, users, reports.
    - Auto-complete for assignees, teams, tags.

5.5 Maintainability

Goals:

- Ensure codebase is modular, testable, and easy to extend.
- Minimize technical debt.

Strategies:

- **Code Organization:**
    - Clear separation: frontend (`components`, `pages`, `hooks`), backend (`models`, `schemas`, `services`, `workers`, `api/routes`).
- **Coding Standards:**
    - TypeScript strict mode for frontend.
    - PEP8 + type hints for Python backend (FastAPI).
    - ESLint / Prettier for consistent formatting.
- **Documentation:**
    - Inline docstrings for backend services.

    - Storybook for reusable frontend components.
    - API Swagger/OpenAPI documentation auto-generated from FastAPI.
- **Testing:**
    - Unit tests for each service, frontend component, API route.
    - Integration tests for workflows (ticket creation → comment → notification → reporting).
    - CI/CD pipelines with automated test runs.
- **Logging & Observability:**
    - Structured logging with correlation IDs for tracing requests.
    - Centralized log aggregation (ELK/CloudWatch).
- **Modular Microservices:**
    - Decouple ticketing, reporting, notifications for easier updates and scaling.
- **Dependency Management:**
    - Pin major versions, security scanning (Dependabot / pip-audit / npm audit).

6. Other Requirements

6.1 Deliverables

This section defines all **artifacts, code, documentation, and deployments** that must be produced and handed over during the lifecycle of the project. These deliverables ensure that the development team, QA, operations, and client stakeholders have all the necessary assets to build, test, deploy, and maintain the system.

6.1.1 Source Code Deliverables

- **Frontend Codebase**

    - Location: `/frontend/`
    - Tech Stack: React + TypeScript + TailwindCSS + shadcn/ui
    - Deliverables:
        - Source files (`.tsx`, `.ts`, `.css`).
        - Component library with reusable UI widgets (buttons, forms, tables).
        - Routing structure for all major pages (`/login`, `/tickets`, `/reports`).
        - State management setup (Redux Toolkit / Zustand).
        - Unit tests for components (`Jest`, `React Testing Library`).
- **Backend Codebase**
    - Location: `/backend/`
    - Tech Stack: Python FastAPI + PostgreSQL + Redis + Celery
    - Deliverables:
        - API endpoints implementation (`/api/v1/...`).
        - Business logic services (`services/`, `usecases/`).
        - Database models & migrations (`models/`, `migrations/`).
        - Background workers for notifications, reporting, and file handling.
        - Role-Based Access Control (RBAC) middleware.
        - Unit and integration tests with pytest.
- **Infrastructure-as-Code (IaC)**
    - Location: `/infra/`
    - Deliverables:
        - Dockerfiles for frontend, backend, workers.

        - Kubernetes manifests (Deployments, Services, Ingress, ConfigMaps, Secrets).
        - CI/CD pipeline scripts (GitHub Actions / GitLab CI).
        - Terraform / Pulumi scripts for cloud infra provisioning (optional).

6.1.2 Documentation Deliverables

- **Software Requirements Specification (SRS)**
    - Already produced (baseline for this LLD).
- **High-Level Design (HLD)**
    - Architectural diagrams, component relationships, system boundaries.
- **Low-Level Design (LLD)** (this document)
    - Detailed breakdown: features, data models, flows, validations.
- **API Documentation**
    - Auto-generated via FastAPI + Swagger.
    - Hosted at `/docs` endpoint.
- **Developer Documentation**
    - README.md with setup instructions.
    - Contribution guide (`CONTRIBUTING.md`).
    - Coding standards (linting rules, formatting, naming).
- **User Documentation (Client-facing)**
    - Quick start guide (PDF/HTML).
    - Step-by-step instructions for ticket creation, tracking, and reporting.
    - FAQs and troubleshooting section.

6.1.3 Testing Deliverables

- **Test Plans & Test Cases**
  - Covering functional + non-functional requirements.
  - Stored in `/tests/docs/`.
- **Automated Test Suites**
  - **Frontend:** Jest + React Testing Library.
  - **Backend:** Pytest + coverage reports.
  - **Integration Tests:** API-level workflows (ticket lifecycle, reporting).
  - **End-to-End Tests:** Cypress for UI workflows.
- **Load & Performance Testing**
  - Scripts using JMeter/Locust.
  - Benchmarks stored in `/tests/performance/`.
- **Security Testing**
  - Static analysis (Bandit, ESLint security plugin).
  - Vulnerability scans (npm audit, pip-audit).
  - Pen-test results if applicable.

6.1.4 Deployment Deliverables

- **Staging Environment**
  - Running build with test data.
  - Accessible by QA and business stakeholders.
- **Production Environment**
  - Fully configured, secured environment.
  - Autoscaling enabled for APIs and workers.
  - Monitoring dashboards configured (Prometheus, Grafana).
- **Release Artifacts**

    - Docker images published to container registry.
    - Helm charts / deployment manifests tagged with release versions.
- **CI/CD Pipeline Deliverables**
    - Continuous integration with linting, unit tests, integration tests.
    - Continuous deployment to staging environment.
    - Manual approval step for production release.

6.1.5 Training & Handover Deliverables

- **Developer Training**
    - Walkthrough of codebase and architecture.
    - Sessions on coding standards, branching strategy, and CI/CD usage.
- **Administrator Training**
    - How to manage users, teams, and roles.
    - How to monitor system health, logs, and alerts.
- **End-User Training**
    - Recorded video tutorials.
    - Hands-on training sessions for managers and team members.

7. High-Level Data Model

This section describes the **core entities** in the system, their attributes, and relationships. The model is designed around **ticketing, project management, workflow automation, and collaboration**.

All tables follow these conventions:

- **Primary Key:** `id BIGSERIAL`
- **Timestamps:** `created_at`, `updated_at` (default: `now()`)

- **Foreign Keys:** Use `ON DELETE CASCADE` for dependent entities (comments, sub-tasks).
- **Indexing:** Each FK has an index for query performance.

7.1 User Entity

Purpose: Represents users (admins, managers, team members, clients).

Table: `users`

Column Name	Data Type	Constraints / Rules	Description
`id`	`BIGSERIAL`	Primary Key	Unique identifier for each user
`name`	`VARCHAR(100)`	NOT NULL	Full name of the user
`email`	`VARCHAR(150)`	UNIQUE, NOT NULL	User's email address (also used as login ID)
`password_hash`	`TEXT`	NOT NULL	Securely hashed password (bcrypt/ argon2 recommended)
`role`	`VARCHAR(32)`	NOT NULL, CHECK (`IN ('admin','manager','member','client')`)	User's system-wide role for access control
`status`	`VARCHAR(32)`	DEFAULT `'active'`, CHECK (`IN ('active','inactive','suspended')`)	Account status (active/ inactive/suspended)
`last_login`	`TIMESTAMPTZ`	Optional	Timestamp of the user's most recent login
`created_at`	`TIMESTAMPTZ`	DEFAULT `now()`	Timestamp when user was created
`updated_at`	`TIMESTAMPTZ`	DEFAULT `now()`	Timestamp when user record was last updated

```
CREATE INDEX idx_users_email ON users(email);
```

Relationships:

- One user can belong to multiple teams (`user_teams`).
- One user can be assigned to many tickets.

- Users can author comments, attachments, and reports.

7.2 Team Entity

Purpose: Represents organizational units managing tickets.

Table: `teams`

Column Name	Data Type	Constraints / Rules	Description
`id`	`BIGSERIAL`	Primary Key	Unique identifier of the user
`name`	`VARCHAR(100)`	NOT NULL	Full name of the user
`email`	`VARCHAR(150)`	UNIQUE, NOT NULL	User's email (used for login + communication)
`password_hash`	`TEXT`	NOT NULL	Hashed password (using bcrypt/argon2, never plaintext)
`role`	`VARCHAR(32)`	NOT NULL, CHECK (`IN ('admin','manager','member','client')`)	Global system role of the user
`status`	`VARCHAR(32)`	DEFAULT `'active'`, CHECK (`IN ('active','inactive','suspended')`)	Current account status
`last_login`	`TIMESTAMPTZ`	Optional	Last login timestamp
`created_at`	`TIMESTAMPTZ`	DEFAULT `now()`	Timestamp when the user was created
`updated_at`	`TIMESTAMPTZ`	DEFAULT `now()`	Timestamp when the user was last updated

Mapping table: `user_teams`

Column Name	Data Type	Constraints / Rules	Description
`id`	`BIGSERIAL`	Primary Key	Unique identifier of the user-team mapping

`user_id`	`BIGINT`	NOT NULL, FK → `users(id)`, `ON DELETE CASCADE`	User that belongs to the team
`team_id`	`BIGINT`	NOT NULL, FK → `teams(id)`, `ON DELETE CASCADE`	Team the user is associated with
`role_in_team`	`VARCHAR(32)`	DEFAULT `'member'`, CHECK (`IN ('member','lead','manager','viewer')`)	Role of the user inside the team (permissions level)
`created_at`	`TIMESTAMPTZ`	DEFAULT `now()`	Timestamp when the mapping was created

📂 Indexes

Index Name	Columns	Purpose
`idx_user_teams_user`	`(user_id)`	Query all teams a user belongs to
`idx_user_teams_team`	`(team_id)`	Query all users in a given team
`idx_user_teams_unique`	`(user_id, team_id)` UNIQUE	Prevent duplicate entries of the same user in a team

Relationships:

- One team has many users (via `user_teams`).
- Teams own tickets, epics, stories.

7.3 Ticket Entity

Purpose: Core unit of the system representing a task, bug, or request.

Table: `tickets`

Table Name	Column Name	Data Type	Constraints / Notes
tickets	id	BIGSERIAL	PRIMARY KEY
	ticket_key	VARCHAR(20)	UNIQUE, NOT NULL (e.g., TSK-1001)
	title	VARCHAR(200)	NOT NULL

	description	TEXT	Optional
	status	VARCHAR(32)	NOT NULL, CHECK: 'open','in_progress','done','closed','archived'
	priority	VARCHAR(16)	CHECK: 'low','medium','high','critical', DEFAULT
	type	VARCHAR(32)	CHECK: 'bug','task','incident','service_request',
	reporter_id	BIGINT	NOT NULL, FK → users(id), ON DELETE
	assignee_id	BIGINT	FK → users(id), ON DELETE SET NULL
	team_id	BIGINT	FK → teams(id), ON DELETE SET NULL
	due_date	DATE	Optional
	resolved_at	TIMESTAMPTZ	Optional
	created_at	TIMESTAMPTZ	DEFAULT now()
	updated_at	TIMESTAMPTZ	DEFAULT now()
Indexes			status, team_id, assignee_id, due_date
ticket_comments	id	BIGSERIAL	PRIMARY KEY
	ticket_id	BIGINT	NOT NULL, FK → tickets(id), ON DELETE
	user_id	BIGINT	NOT NULL, FK → users(id), ON DELETE
	message	TEXT	NOT NULL
	created_at	TIMESTAMPTZ	DEFAULT now()
	updated_at	TIMESTAMPTZ	DEFAULT now()
Indexes			ticket_id, user_id

Relationships:

- Tickets belong to teams.
- Tickets can have comments, attachments, workflow rules triggered.
- Tickets can be part of stories or epics.

7.4 Comment Entity

Purpose: Allows collaboration through discussions on tickets.

Table: `comments`

Column Name	Data Type	Constraints / Rules	Description
id	BIGSERIAL	Primary Key	Unique identifier of the comment
ticket_id	BIGINT	NOT NULL, FK → tickets(id), ON DELETE CASCADE	The ticket this comment belongs to
author_id	BIGINT	NOT NULL, FK → users(id)	The user who wrote the comment
content	TEXT	NOT NULL	The actual comment text
mentions	JSONB	DEFAULT '[]'::jsonb	List of mentioned users (e.g., ["@user1", "@user2"])
created_at	TIMESTAMPTZ	DEFAULT now()	Timestamp when the comment was created
updated_at	TIMESTAMPTZ	DEFAULT now()	Timestamp when the comment was last updated
deleted	BOOLEAN	DEFAULT false	Soft delete flag (true = comment hidden but retained for audit)

```
CREATE INDEX idx_comments_ticket ON comments(ticket_id);
```

Relationships:

- Each comment belongs to a ticket.
- Mentions generate notifications.

7.5 Work Hierarchy (Epics, Stories, Sub-tasks)

Epics

Column Name	Data Type	Constraints / Rules	Description
id	BIGSERIAL	Primary Key	Unique identifier of the epic
team_id	BIGINT	NOT NULL, FK → teams(id)	Team that owns the epic
title	VARCHAR(200)	NOT NULL	Title/summary of the epic
description	TEXT	Optional	Detailed description of the epic

status	VARCHAR(32)	DEFAULT 'open', CHECK (IN ('open','in_progress','done','archived'))	Current status of the epic
due_date	DATE	Optional	Planned completion date for the epic
created_at	TIMESTAMPTZ	DEFAULT now()	Timestamp when the epic was created
updated_at	TIMESTAMPTZ	DEFAULT now()	Timestamp when the epic was last updated

Stories

Column Name	Data Type	Constraints / Rules	Description
id	BIGSERIAL	Primary Key	Unique identifier of the story
epic_id	BIGINT	NOT NULL, FK → epics(id), ON DELETE CASCADE	Parent epic that this story belongs to
title	VARCHAR(200)	NOT NULL	Title/summary of the story
description	TEXT	Optional	Detailed description of the story
priority	VARCHAR(16)	DEFAULT 'medium', CHECK (IN ('low','medium','high','critical'))	Priority level of the story
assignee_id	BIGINT	FK → users(id)	User assigned to the story (if any)
status	VARCHAR(32)	NOT NULL, DEFAULT 'backlog', CHECK (IN ('backlog','todo','in_progress','done','blocked'))	Current workflow status of the story
created_at	TIMESTAMPTZ	DEFAULT now()	Timestamp when the story was created
updated_at	TIMESTAMPTZ	DEFAULT now()	Timestamp when the story was last updated

```
CREATE INDEX idx_stories_epic ON stories(epic_id);
```

Sub-tasks

Column Name	Data Type	Constraints / Rules	Description
id	BIGSERIAL	Primary Key	Unique identifier of the sub-task
story_id	BIGINT	NOT NULL, FK → stories(id), ON DELETE CASCADE	Parent story that this sub-task belongs to
title	VARCHAR(200)	NOT NULL	Title/summary of the sub-task
status	VARCHAR(32)	DEFAULT 'todo', CHECK (IN ('todo','in_progress','done','blocked'))	Current status of the sub-task
created_at	TIMESTAMPTZ	DEFAULT now()	Timestamp when the sub-task was created
updated_at	TIMESTAMPTZ	DEFAULT now()	Timestamp when the sub-task was last updated

```
CREATE INDEX idx_subtasks_story ON sub_tasks(story_id);
```

Relationships:

- Epic → multiple Stories → multiple Sub-tasks.
- Stories & sub-tasks link to tickets for workflow tracking.

7.6 Attachments Entity

Purpose: Manage files uploaded for tickets/comments.

Table: `attachments`

Column Name	Data Type	Constraints / Rules	Description
id	BIGSERIAL	Primary Key	Unique identifier of the attachment
ticket_id	BIGINT	FK → tickets(id), ON DELETE CASCADE	Linked ticket (if file is tied directly to a ticket)
comment_id	BIGINT	FK → comments(id), ON DELETE CASCADE	Linked comment (if uploaded with a comment)
filename	VARCHAR(255)	NOT NULL	Original filename of uploaded file

content_type	VARCHAR(100)	NOT NULL	MIME type (e.g., image/png, application/pdf)
size	BIGINT	NOT NULL, recommended CHECK (size <= 20971520)	File size in bytes (max 20MB if business rule enforced)
storage_url	TEXT	NOT NULL	URL/path to file in storage (e.g., S3 presigned URL)
status	VARCHAR(32)	DEFAULT 'pending', CHECK (IN ('pending','clean','infected'))	File status after virus scan
uploaded_by	BIGINT	FK → users(id)	User who uploaded the file
created_at	TIMESTAMPTZ	DEFAULT now()	When file was uploaded

Indexes

Index Name	Columns	Purpose
idx_attachments_ticket	(ticket_id)	Fetch all attachments for a ticket
idx_attachments_comment	(comment_id)	Fetch all attachments tied to a comment
idx_attachments_user	(uploaded_by)	Query attachments uploaded by a specific user

7.7 Notifications Entity

Purpose: Store in-app notifications triggered by comments, mentions, or rules.

Table: notifications

Column Name	Data Type	Constraints / Rules	Description
id	BIGSERIAL	Primary Key	Unique identifier of the notification
user_id	BIGINT	NOT NULL, FK → users(id), ON DELETE CASCADE	User who will receive the notification

type	VARCHAR(64)	NOT NULL, CHECK (IN ('mention','assignment','sla_breach','report_ready','comment','status_change'))	Type of notification
message	TEXT	NOT NULL	Human-readable notification message
related_ticket	BIGINT	FK → tickets(id), ON DELETE CASCADE	Linked ticket (if applicable)
related_comment	BIGINT	FK → comments(id), ON DELETE CASCADE	Linked comment (if applicable, e.g., mention)
read	BOOLEAN	DEFAULT false	Whether the notification has been read
severity	VARCHAR(16)	DEFAULT 'info', CHECK (IN ('info','warning','critical'))	Severity level of the notification
created_at	TIMESTAMPTZ	DEFAULT now()	Timestamp when notification was created
updated_at	TIMESTAMPTZ	DEFAULT now()	Last updated (e.g., when marked read)

Indexes

Index Name	Columns	Purpose
idx_notifications_user	(user_id)	Lookup notifications for a specific user
idx_notifications_ticket	(related_ticket)	Fetch notifications tied to a ticket
idx_notifications_type	(type)	Filter notifications by type
idx_notifications_unread	(user_id, read) WHERE read=false	Optimized lookup for unread notifications per user

7.8 Reports Entity

Purpose: Track async report requests and exports.

Table: `reports`

id	BIGSERIAL	PRIMARY KEY	Auto-incrementing internal ID for the report
report_id	UUID	NOT NULL, UNIQUE	External UUID for tracking/report reference
created_by	BIGINT	NOT NULL, REFERENCES users(id)	User who generated the report
filters	JSONB		JSON object storing filter criteria used in the report
columns	JSONB		JSON array/object describing which columns to include
format	VARCHAR(16)	NOT NULL	Export format (`csv`, `xlsx`, `pdf`)
status	VARCHAR(32)	NOT NULL, DEFAULT `'pending'`	Report generation status (`pending`, `running`, `ready`, `failed`, `canceled`)
s3_key	TEXT		Path/key of the generated file in storage (e.g., S3 bucket)
error_message	TEXT		Error details if report generation fails
email_on_ready	BOOLEAN	DEFAULT false	Whether to send an email when the report is ready
created_at	TIMESTAMPTZ	DEFAULT now()	Timestamp when the report was requested
completed_at	TIMESTAMPTZ		Timestamp when the report finished (success/failure)

7.9 Relationships Summary (ERD Style)