System Overview

- **Purpose**: Design and architecture for a web-based system with organizational hierarchy, role-based access control (RBAC), task scheduling, analytics, notifications, and external integrations. Focus on enterprise productivity, problem tracking by branch location, data-driven performance analysis, and support for future mobile app integration.
- **Scope**:
    - Manage tasks (daily, weekly, monthly).
    - Track operational problems across branches.
    - Analyze performance using advanced analytics.
    - Integrate with third-party systems (Telegram, CRM tools like Salesforce, HubSpot, Zoho).
    - Customize branding and themes.
    - Scale securely using Dockerized infrastructure.
- **Product Perspective**: Multi-tenant SaaS-style web application where multiple organizations exist independently, each with departments, teams, and hierarchical roles. Users access features based on assigned roles.
- **User Classes**:
    - Super Admin: System-wide administration.
    - Organization Admin: Manage organization, users, roles.
    - Manager: Assign tasks, review reports.
    - Staff/User: Execute tasks, receive notifications.
    - Analyst: Access reports & analytics.

Functional Requirements

- **Authentication & Authorization**:
    - **User Signup**: SMTP-based email verification, allowed email domain restriction (e.g., @company.com), email verification required before activation.
    - **User Sign-In**: Secure login using email & password, JWT-based authentication, session expiration and refresh tokens.
- **Role-Based Access Control (RBAC)**:
    - Create, edit, delete, and assign roles.
    - Permission-based access (read/write/update/delete).
    - Role inheritance support.
    - Example Roles: Admin, Manager, Analyst, User.
- **Organization & Hierarchy Management**:
    - **Organization**: Create multiple organizations, organization-level branding & configuration.
    - **Department Management**: Create and manage departments, assign users to departments.
    - **Team Management**: Create teams under departments, assign team leads and members.

    - **Hierarchy**: Organization → Department → Team → User, with hierarchical visibility & access control.
- **UI Customization**:
    - Upload organization logo.
    - Theme customization: Primary color, secondary color, dark/light mode.
    - Applied system-wide per organization.
- **Task Management System**:
    - **Task Boards**: Daily Tasks, Weekly Tasks, Monthly Tasks.
    - Each task includes: Title, Description, Priority, Assigned user/team, Start date & due date, Task category.
    - **Task Status**: Not Started, Started, Pending, Completed.
- **Notifications**:
    - Channels: Web notifications, Email (SMTP), Telegram Bot integration.
    - Triggers: Task creation, task deadline reminders (daily, weekly, monthly), task status changes, problem assignment updates.
- **Reporting & Rating System**:
    - **Reports**: Daily task report, Weekly task report, Monthly task report.
    - **Rating System**: Performance rating based on completion metrics (e.g., ⭐⭐⭐⭐⭐ Excellent, ⭐⭐⭐⭐ Very Good, ⭐⭐⭐ Good, ⭐⭐ Poor, ⭐ Very Poor).
- **Advanced Analytics**:
    - Insights: Task completion trends, productivity heatmaps, user performance scoring, department efficiency comparison, predictive analytics (task delay probability), SLA breach analysis.
    - Types: Line charts, bar charts, radar charts, heatmaps, forecast models.
- **Problem Area Tracking Module**:
    - Purpose: Track operational issues across branch locations.
    - Record Fields: Location (Branch), Problem Type, Assigned Date, Customer Name, Assigned Person, Problem Fixed Date, Resolution Time (auto-calculated), Status (Open / In Progress / Fixed).
    - Analytics: Most frequent problem types, resolution time per branch, staff efficiency ranking.
- **External Integrations**:
    - **Telegram**: Task notifications, problem assignment alerts, daily summary reports.
    - **CRM & Third-Party Tools**: REST API integration with CRMs (Salesforce, HubSpot, Zoho), webhooks for event-based data sync, OAuth2 authentication for external tools.

Non-Functional Requirements

- **Performance**: API response < 300ms for 95% of requests, scalable microservice-ready architecture.
- **Security**: Encrypted passwords (bcrypt), JWT authentication, role-based authorization, email domain restriction, HTTPS enforced.

- **Scalability**: Dockerized services, horizontal scaling support, load-balancer ready.
- **Availability**: 99.9% uptime target, database backups & recovery strategy.

System Architecture

- **Technology Stack**:

Layer	Technology
Frontend	Angular
Backend	Python (FastAPI / Django REST)
Database	PostgreSQL
Deployment	Docker
Notifications	SMTP, Telegram API
Analytics	Python ML/Stats libraries
API Style	RESTful

- **High-Level Architecture**:
    - Angular Web App → API Gateway → Python Backend Services → PostgreSQL Database → Analytics Engine.

Database Design (High-Level)

- Key Tables: Users, Roles, Permissions, Organizations, Departments, Teams, Tasks, Task_Reports, Problem_Areas, Notifications, Audit_Logs.

Deployment Architecture

- Docker containers for: Frontend, Backend, Database.
- Environment-based configuration.
- CI/CD pipeline ready.
- Cloud-agnostic (AWS / Azure / GCP).

Mobile App Integration

- REST APIs exposed for mobile clients.
- Token-based authentication.
- Push notification-ready.
- Web-first, mobile-compatible architecture.

Future Enhancements

- AI-based task recommendation.
- Auto-task generation.
- Mobile native apps.
- Voice command integration.
- Chatbot for task updates.

Implementation Guidelines

1. Start by setting up the backend with FastAPI or Django REST Framework, implementing authentication and RBAC first.
2. Design the database schema in PostgreSQL based on the key tables.
3. Build the Angular frontend with components for task boards, hierarchies, and custom themes.
4. Integrate notifications via SMTP and Telegram API.
5. Implement analytics using Python libraries like Pandas, Matplotlib, Scikit-learn for insights and charts.
6. Ensure all APIs are RESTful, secure, and performant.
7. Dockerize the application and provide a docker-compose file for local deployment.
8. Write unit tests for critical components (e.g., auth, task assignment).
9. Document any assumptions or deviations from the SDD.

Output the complete codebase in a structured format, including source files, database migration scripts, and deployment instructions. If generating code, use best practices for readability and modularity.