Dashboard development roadmap

we are planning to build a dashboard for projects, tasks, work reports and etc. management based on role + department (cause each department have their diff diff role and work management flow) how can we approach from scratch to end of development give me brief overview how can we manage and what would be the possibilities

Nice — this is a great product to build. Below is a compact, practical roadmap from idea → production, plus architecture patterns, data model examples, feature ideas and pitfalls. I'll keep it actionable so you can hand it to PMs/engineers and start slicing an MVP.

1) High-level approach (phases, not time)

- **Discovery** — interview stakeholders per department, capture role-specific workflows, must-have fields, compliance needs, reporting KPIs.
- **Define MVP** — smallest set that delivers value (projects, tasks, basic role-based access, reports export).
- **Design** — UX flows & component library, permission UI, templates per department.
- **Architecture & Data model** — pick DB, auth, integrations.
- **Implement** — backend APIs, frontend components, unit + integration tests.
- **Deploy & Monitor** — CI/CD, logs, error/usage monitoring, rollout by department.
- **Iterate** — add custom workflows, analytics, automation.

2) Core concepts / domain model (entities)

Keep the model generic but extensible:

- **Organization / Department**
  - id, name, settings (workflows, templates)
- **User**
  - id, name, email, department_id, role_ids, metadata
- **Role**
  - id, name, scope (global / dept), default permissions

- **Permissions / Policies**
  - CRUD on Projects/Tasks/Reports, special actions (approve, assign, escalate)
- **Project**
  - id, owner, department_id, metadata, status, tags
- **Task / Work Item**
  - id, project_id, assignee(s), type, status, priority, estimates, custom fields
- **Workflow / State Machine** (configurable per dept)
  - states, transitions, triggers, guards
- **Work Report / Timesheet / Log**
  - user_id, task_id, date, hours, description, attachments
- **Audit / Activity Log**
  - who, when, what changed
- **Attachment / Comment / Notification**

3) Authorization model (recommended)

Use **hybrid RBAC + ABAC**:

- **RBAC**: roles define baseline permissions (viewer, editor, manager).
- **ABAC**: attributes (department, project.owner, task.type) control fine-grained rules (e.g., QA role can move tasks only in QA dept).
- Support **scoped roles**: same role can have department scope.
- Admin UI to manage: roles, permission matrix, department-specific overrides.

4) Handling department-specific flows

- **Configurable Workflows** per department: edit states and who can transition.
- **Field templates**: allow department to add custom fields for tasks/reports.
- **Role mapping**: map department roles to global capabilities (so same “QA Lead” in QA vs BAU can have different perms).
- **Feature flags / toggles** so you can enable department-specific features without branching code.

5) UX patterns (what to build first)

- **Role-based dashboards**: quick KPIs, pending approvals, assigned items.
- **Project view**: summary, timeline, members, milestones.
- **Task board**: Kanban + list + calendar views toggle.
- **Work report**: log hours, generate PDF/export, manager approve/reject.
- **Admin panel**: users, roles, department settings, workflow editor.

- **Notifications center**: in-app + email + webhooks.

6) APIs & backend design (example endpoints)

- Auth: `POST /auth/login`, `POST /auth/refresh`, `GET /auth/me`
- Users: `GET /users`, `POST /users`, `PUT /users/:id`
- Projects: `GET /projects`, `POST /projects`, `GET /projects/:id`
- Tasks: `GET /projects/:id/tasks`, `POST /tasks`, `PATCH /tasks/:id`
- Reports: `GET /reports`, `POST /reports/:taskId`
- Admin: `GET /roles`, `PUT /roles/:id`, `GET /departments/:id/workflow`
  Implement **policy enforcement at API layer** (mid-tier) and again in UI to hide disabled actions.

7) Data storage / tech choices (suggestions & tradeoffs)

- **Relational (Postgres)** — good for strong data integrity, joins, reports. Use when you need transactions and complex queries.
- **Document DB (MongoDB)** — good for flexible custom fields per department. Use if schemas vary a lot.
- **Hybrid**: Postgres for core entities, ElasticSearch for search, Redis for caching/queues.
- **ORM/Tools**: Prisma (Postgres), TypeORM/NestJS, Mongoose (Mongo).

8) Infrastructure & scaling

- Stateless API servers (Node.js / NestJS / Django) behind load balancer.
- Background jobs with worker queue (BullMQ / RabbitMQ) for notifications, report generation, exports.
- File storage: S3-compatible for attachments.
- CI/CD pipelines for automated tests and deployments.
- Observability: Sentry for errors, Prometheus/Grafana for metrics, ELK or Loki for logs.

9) Reporting & analytics

- Basic: task counts, throughput, completion time, overdue tasks, billable hours.
- Advanced: build a metrics pipeline (events → OLAP store or data warehouse) for ad-hoc reports.
- Export: CSV / Excel / PDF and API endpoints for report automation.

10) Integrations & automation

- Slack/Teams notifications, email, calendar (iCal/Google Calendar), single-sign-on (SAML/OAuth).
- Webhooks for external automation.
- Automations engine: allow triggers like “when task moved to QA → assign reviewer”.

11) Security & compliance

- Strong auth: OAuth2 + refresh tokens, optional SSO (SAML/Google).
- Field-level access if necessary (sensitive fields).
- Audit trails for edits.
- Rate limiting, input validation, OWASP checks.
- Data retention policies (especially for reports).

12) Testing strategy

- Unit tests for business logic (RBAC/ABAC rules).
- E2E tests for critical flows (create project → create task → report).
- Contract tests for APIs (consumer-driven tests if multiple services).
- Load tests for peak usage (task bulk updates, export generation).

13) MVP feature checklist (minimal but useful)

- Multi-department org model + users import.
- Roles + permission matrix (basic RBAC).
- Projects CRUD, Tasks CRUD, assign/ comment/ attachments.
- Simple configurable workflow per department (2–3 states).
- Work report entry + export (CSV, PDF).
- Notifications (in-app + email).
- Admin UI for roles/departments and user management.
- Audit log and basic analytics dashboard.

14) Advanced/optional features (post-MVP)

- Per-department advanced workflow engine.
- Templates for projects/tasks by department.
- SLA/escalation automation.
- Gantt/timeline views and dependency management.
- Mobile app or PWA.

- AI-assist: auto-assign, summarize reports.

15) Example DB schema (short)

Projects:

```json
{
  "id":"uuid",
  "name":"Website Revamp",
  "department_id":"dept_1",
  "owner_id":"user_12",
  "status":"active",
  "meta": {"budget":50000}
}
```

Tasks:

```json
{
  "id":"uuid",
  "project_id":"proj_1",
  "title":"Design landing",
  "type":"design",
  "status":"in_review",
  "assignees":["user_5"],
  "custom_fields": {"browser_support":"all"}
}
```

RolePermission:

```scss
(role_id, resource, action, allowed, scope)
```

16) Common pitfalls & tradeoffs

- **Over-customizing early**: trying to support every department's nuance pre-MVP will delay launch. Ship common primitives + templates.
- **Permissions complexity**: hard to debug; build admin UI with simulation (preview a user's effective permissions).

- **Schema sprawl**: custom fields everywhere become hard to query—consider a hybrid approach (core fields relational + JSON for extras).
- **Workflow edge-cases**: model transitions and approvals explicitly; add testing for each transition.

17) Monitoring success — KPIs to track

- Time-to-complete tasks (by dept/role)
- Tasks overdue rate
- Adoption: active users per dept
- Report submission rate
- Workflow bottlenecks (where tasks pile up)

18) Next practical steps (concrete)

1. Run workshops with 2–3 departments, capture workflows & required reports.
2. Define MVP scope based on impact (use the checklist above).
3. Create UI wireframes for dashboard, task board, admin panel.
4. Define core DB schema and auth model (RBAC + ABAC plan).
5. Build a vertical slice: auth → project → task → report → notifications. Ship that as the first internal release.

If you want, I can:

- Draft a concrete **MVP backlog** (user stories) for sprint-ready tickets.
- Sketch a **roles × permission matrix** template for your admin UI.
- Produce a minimal **ER diagram** (visual) or **sample Postgres schema**.

Which one should I generate next?