ChatGPT

MERN Project Management Backend Design

This document outlines a production-ready backend architecture for a MERN (MongoDB, Express.js, React, Node.js) project management platform. The design covers schemas, routes, controllers, and access control across multiple domains. We emphasize **modular, scalable, secure design** with clear multi-tenant isolation and RBAC. All schemas use Mongoose with `timestamps` for audit fields [1]. Indexes are added on frequently queried fields for performance [2]. Sensitive data (e.g. tokens) is encrypted or stored securely.

1. Users & Authentication

Schema (User):
- `_id` (ObjectId)
- `clerkUserId`: String – Clerk's user identifier (for Clerk integration)
- `firstName`, `lastName`: String
- `email`: String (unique, indexed)
- `passwordHash`: String (if not fully delegating to Clerk)
- `roles`: [String] – e.g. `["organization_admin","project_manager","member"]`
- `organizationId`: ObjectId (ref to Organization) [3]
- `lastLogin`: Date
- `createdAt`, `updatedAt`: Date (auto-managed via Mongoose `timestamps: true` [1])

Indexes: Unique index on `email`. Index on `organizationId` for scoping queries.

Routes (Express):
- `POST /auth/signup` – Create user (via Clerk or local).
- `POST /auth/login` – Authenticate (Clerk OAuth or JWT).
- `GET /auth/me` – Get current user profile.
- `PATCH /users/:id` – Update user (admin or self).
- `DELETE /users/:id` – Remove user (admin only).

Controller Outlines:
- **createUser:** Validate input, call Clerk signup (or hash password), save User doc, assign default role.
- **loginUser:** Verify credentials via Clerk or JWT, issue token.
- **getProfile:** Retrieve `req.user` (populated from auth middleware), return profile.
- **updateUser:** Check RBAC (e.g. only org-admin or self), update allowed fields.
- **deleteUser:** Check admin role, remove user (optionally cascade to projects/tasks).

Access Control:
- Users belong to an **Organization** (tenant) – no cross-org access [3] [4].
- Roles per user determine permissions. For example, `organization_admin` can manage users in their org, whereas a regular `member` cannot.

2. Projects & Members

Schema (Project):
- `_id` (ObjectId)
- `name`: String
- `description`: String
- `organizationId`: ObjectId (ref Organization) [3]
- `ownerId`: ObjectId (ref User – project creator)
- `memberIds`: [ObjectId] (refs to Users; many-to-many relationship)
- `startDate`, `endDate`: Date
- `status`: String (e.g. `"active"`, `"archived"`)
- `priority`: String (e.g. `"high"`, `"medium"`)
- `createdAt`, `updatedAt`: Date (timestamps)

> **Indexes:** Compound index on `(organizationId, ownerId)` and on `memberIds`.

Routes:
- `GET /projects` – List projects for the user's organization (filter by org and membership).
- `POST /projects` – Create project (org-admin or project-manager).
- `GET /projects/:projectId` – Get project details (members only).
- `PUT /projects/:projectId` – Update project (owner or admin).
- `DELETE /projects/:projectId` – Delete/Archive project (owner or admin).
- `POST /projects/:projectId/members` – Add member by user ID (admin/project-owner).
- `DELETE /projects/:projectId/members/:userId` – Remove member.

Controller Notes:
- **listProjects:** Query by `organizationId` and optionally membership (`memberIds` contains user). Return paginated list.
- **createProject:** Validate org context, assign current user as owner, add owner to `memberIds` by default.
- **updateProject:** Only allow modifying certain fields; cannot move project out of org.
- **addMember/removeMember:** Update `memberIds` array; send notifications.

Access Control:
- All project operations require that `req.user.organizationId` matches `project.organizationId`.
- Only users with `org_admin` or `project_owner` roles can add/remove members or delete a project.
- Regular members can view and update tasks within projects they belong to.

3. Tasks & Nested Subtasks

Schema (Task):
- `_id` (ObjectId)
- `projectId`: ObjectId (ref Project)
- `parentId`: ObjectId (ref Task, nullable) – for nested subtasks [5]
- `ancestors`: [ObjectId] – array of all parent task IDs (Materialized Path pattern) [5]
- `title`: String

- `description`: String
- `status`: String (e.g. `"todo"`, `"in_progress"`, `"done"`)
- `assigneeId`: ObjectId (ref User)
- `priority`: String
- `dueDate`: Date
- **Gantt fields:** `startDate`, `endDate`, `duration`, `baselineStart`, `baselineEnd` (for timeline baseline tracking)
- **Dependencies:** `dependsOn`: [ObjectId] (refs to other Task IDs) – predecessor tasks for critical path.
- **Budget:** `estimateHours`, `actualHours`, `estimateCost`, `actualCost`, `currency`
- `customFields`: Array of {fieldId: ObjectId, value: Mixed} – for dynamic fields.
- `createdAt`, `updatedAt`: Date (timestamps)

> **Hierarchy Modeling:** We use the *materialized path* approach. Each task stores its `parentId` and an `ancestors` array, enabling fast subtree queries [5]. Alternatively, MongoDB's `$graphLookup` can traverse this tree if needed. Embedding all subtasks in one document risks unbounded arrays, so this normalized approach is preferred for unlimited depth [6].
>
> **Indexes:** Compound index on `(projectId, parentId)`. Index on `ancestors` for fast subtree lookup. Index on `assigneeId`. Index on `dueDate` for sorting.

Routes:
- `GET /projects/:projectId/tasks` – List all tasks for a project (with optional filters).
- `POST /projects/:projectId/tasks` – Create new task (optionally with `parentId` for a subtask).
- `GET /tasks/:taskId` – Get task details (includes custom fields and action cards).
- `PUT /tasks/:taskId` – Update task fields (status, assignee, dates, etc.).
- `DELETE /tasks/:taskId` – Delete a task (should also delete/soft-delete its subtasks).
- `GET /tasks/:taskId/subtasks` – List direct subtasks (could use `parentId=taskId`).
- **Special:** `PATCH /tasks/:taskId/move` – Change parent or reorder;
- **Special:** `POST /tasks/:taskId/dependencies` – Set or update predecessor dependencies.

Controller Outlines:
- **listTasks:** Query tasks by `projectId` (and optional filters like status, assignee). Optionally populate nested subtasks (using `$graphLookup` or recursive queries).
- **createTask:** Insert Task document, compute `ancestors` as `parentTask.ancestors + [parentTask._id]` if nested.
- **updateTask:** Save updates; if changing parent, update descendants' `ancestors`.
- **deleteTask:** Perform cascade delete or mark tasks and its subtree as deleted. Use `$graphLookup` on `ancestors` to find descendants.
- **computeCriticalPath:** (Utility) Uses task durations and dependencies to calculate critical path lengths for Gantt – could run periodically or on-demand.

Access Control:
- Must belong to the project's organization. Typically any project member can create or update tasks; only project admin/owner can delete or move tasks across projects.

4. Real-Time Chat (Socket.IO)

Schema (ChatRoom, Message):
- **ChatRoom:**
- `_id` (ObjectId)
- `projectId`: ObjectId (ref Project) – a room scoped to a project (or global rooms).
- `name`: String
- `memberIds`: [ObjectId] (ref Users allowed in room)
- `createdAt`, `updatedAt` (timestamps).
- **Message:**
- `_id` (ObjectId)
- `roomId`: ObjectId (ref ChatRoom)
- `senderId`: ObjectId (ref User)
- `content`: String
- `attachments`: [{url, filename, uploaderId}]
- `createdAt`: Date (timestamp)

> **Indexes:** Index `roomId` on Messages for retrieving history. Possibly index `roomId, createdAt`.

Routes (HTTP):
- `GET /projects/:projectId/chat/rooms` – List chat rooms in a project.
- `POST /projects/:projectId/chat/rooms` – Create a new room.
- `GET /chat/rooms/:roomId/messages` – Paginated fetch of recent messages in room.

Real-Time Socket Endpoints:
- Clients connect via Socket.IO to e.g. `/socket`. Authenticate JWT and join allowed rooms.
- **Events:**
- `message:new` – Send message (server saves to DB, emits to room).
- `room:join` / `room:leave` – Manage room subscriptions.

Controller Outlines:
- **getRooms:** Verify project membership, return rooms.
- **getMessages:** Verify room membership, query last 100 messages sorted by time.
- **socketHandler:** On connection, verify JWT, add user to rooms based on membership. On `message:new`, save Message and broadcast to `roomId`.

Access Control:
- Only users in `ChatRoom.memberIds` (and project members) can join/listen. Enforce both via Socket.IO middleware and route guards.

5. Gantt/Timeline Support

Schema (Gantt Fields):
- Tasks include:

- `startDate`, `endDate` (current schedule)
- `baselineStart`, `baselineEnd` (original plan dates)
- `duration` (computed from dates)
- `dependsOn`: [ObjectId] (predecessor task IDs).

This allows storing timeline data **with baseline tracking**. The `dependsOn` edges are used for **critical path** calculation.

> **Accessing Timeline:** There is no separate Gantt collection; the timeline is derived from Task data.

Routes:
- `GET /projects/:projectId/gantt` – Return all tasks with their start/end/baseline dates and dependencies.
- `GET /tasks/:taskId/critical-path` – Compute or fetch critical path length/points.

Controller Notes:
- **getGanttData:** Collate tasks for a project, return JSON suitable for frontend Gantt chart (list of tasks with dates, deps).
- **computeCriticalPath:** Use topological sort on `dependsOn` graph (per project) to compute earliest/latest start times. This can be done in controller or offloaded to a batch job.

Access Control:
- Same as tasks: project membership required. Only allow org-scoped access.

6. Budget Tracking

Schema (Budget fields):
- **Project:**
- `budgetEstimate`, `budgetActual`, `budgetCurrency` (e.g. USD)
- **Task:**
- `estimateHours`, `actualHours`
- `estimateCost`, `actualCost`, `currency`

Optionally, an **Expense** collection could track individual expenditures or time entries: - **Expense:** `_id, projectId, taskId (optional), amount, description, date, createdBy, currency`.

> **Indexes:** Index on `projectId`, and on `taskId` if tasks have budgets.

Routes:
- `GET /projects/:projectId/budget` – Get overall vs actual budget stats.
- `PATCH /projects/:projectId/budget` – Update budget (authorized users).
- `GET /tasks/:taskId/budget` – Get task budget details.
- `PATCH /tasks/:taskId/budget` – Update task estimates/actuals.
- `POST /projects/:projectId/expenses` – Record a new expense.

Controllers:
- **updateBudget:** When actuals change (e.g. expense added), update sums on project/task or recalc totals.
- **getBudget:** Aggregate task estimates vs actuals, return for reports.

Access Control:
- Usually restricted to project managers or finance roles. Org-admin can configure budgets. Regular members can log time/cost if permitted, but only project-manager updates estimates.

7. Custom Fields & Action Cards

Custom Fields: Allow dynamic fields on tasks or projects.
- **Schema (FieldDefinition):**
- `_id`, `organizationId`, `projectId` (nullable if global), `name`, `type` (e.g. "text","number","date","dropdown"), `options` (for dropdown), `required`: Boolean.
- **Usage:** Task documents have a `customFields: [{ fieldId: ObjectId, value: Any }]` array matching these definitions.

> **Notes:** Storing values within Task leverages atomic update of the document. Alternatively, a dedicated `TaskCustomValue` collection could be used for large numbers of fields. Mongoose population can fetch field metadata.

Action Cards: Generic interactive elements on tasks.
- **Comments:** Stored in a separate `Comment` collection:
- Fields: `_id, taskId, userId, text, createdAt`.
- **Attachments:** Stored in `Attachment` (or use GridFS):
- Fields: `_id, taskId, filename, url, uploadedBy, createdAt, version`.
- **Checklists:** Could embed as sub-doc in Task (`checklist: [{ text, done }]`), or as a `Checklist` collection if complex.

> **Atomicity:** Embedding small checklists or latest comment into Task can use single-document transactions [7], but since these can grow, we use separate collections to avoid large arrays [7].

Routes:
- **Comments:** `GET /tasks/:id/comments`, `POST /tasks/:id/comments`, `DELETE /comments/:commentId`.
- **Attachments:** `GET /tasks/:id/attachments`, `POST /tasks/:id/attachments`, `DELETE /attachments/:id`.
- **Checklists:** `GET /tasks/:id/checklist`, `POST /tasks/:id/checklist`, `PUT /tasks/:id/checklist/item/:itemId`.

Controllers:
- **addComment:** Save a Comment doc. Optionally notify task followers.
- **addAttachment:** Save metadata and upload file to storage (e.g. S3), link in document.
- **updateChecklist:** Modify checklist items within Task (embedded array or separate doc).

Access Control:
- Any project member can comment or attach. Only the uploader or project admins can delete attachments/comments.

8. AI Features (Auto-labeling, Suggestions, Summaries)

Schema: AI outputs can be stored in existing documents or a new collection:
- Add fields to Task (or Project):
- `aiLabels`: [String] – auto-generated labels.
- `aiSummary`: String.
- `aiLastUpdated`: Date.

Alternatively, store per-task suggestions in an `AISuggestion` collection: `{ taskId, type, content, generatedAt }`.

Routes:
- `POST /tasks/:id/ai/labels` – Trigger auto-label generation (calls LLM).
- `POST /tasks/:id/ai/summary` – Generate summary.
- `GET /tasks/:id/ai` – Fetch last AI-generated data.

Controllers:
- **generateLabels:** Collect task data, call LLM (Gemini/GPT), parse labels, save to task.
- **generateSummary:** Similar: prompt LLM to summarize task updates/comments, save.

Notes: AI calls are asynchronous – controllers may enqueue jobs. Store results on success. Ensure LLM credentials are in secure config.

Access Control:
- Any authenticated user can request suggestions on tasks they have access to.

9. Versioning & Proofing

Schema (FileVersion & Approval):
- **File:**
- `_id, projectId, taskId (optional), filename, mimeType, currentVersion: Number, createdAt`.
- **FileVersion (sub-doc or separate):**
- `versionNumber`, `uploadedBy`, `url`, `uploadDate`.
- **Approval:**
- Tracks proofing: `_id, fileId, versionNumber, reviewerId, status ("approved"/"rejected"), comments, reviewedAt`.

> Files may be stored in GridFS or cloud (S3) and referenced by URL. Versions are kept track of numerically.

Routes:
- `POST /files` – Upload a new file (creates File+version1).
- `POST /files/:id/versions` – Upload new version (increments `currentVersion`).
- `GET /files/:id/versions` – List version metadata.
- `POST /files/:fileId/versions/:versionId/approve` – Submit approval decision.
- `GET /files/:id/approvals` – Get approval history.

Controllers:
- **uploadFileVersion:** Store file (e.g. S3), update File doc with new version subdoc and increment.
- **approveVersion:** Create Approval record and update File or version status. Possibly notify uploader.

Access Control:
- Only designated reviewers (project or file-specific role) can approve. Owners can upload new versions.

10. Intake Forms & Templates

Schema:
- **FormTemplate:** `_id, organizationId, name, fields: [{ id, label, type, options, required }], createdBy, createdAt`.
- **FormRequest:** `{ _id, templateId, organizationId, submittedBy, data: {fieldId: value}, status (e.g. "pending","approved","denied"), assignedTo, createdAt, updatedAt }`.

> This allows customizing project intake or request workflows per organization.

Routes:
- `GET /forms/templates` – List form templates (org-scoped).
- `POST /forms/templates` – Create new intake form template.
- `GET /forms/templates/:id` – Get template details.
- `POST /form-requests` – Submit a new request (with `templateId` and `data`).
- `GET /form-requests/:id` – View request status.
- `GET /form-requests` – List own or org's requests (admins see all, users see own).

Controllers:
- **createTemplate:** Save fields definitions.
- **submitRequest:** Validate data against template (e.g. required fields), save FormRequest with `status="pending"`. Notify triage team.
- **updateRequest:** Change status/assignment (admin action).

Access Control:
- Templates are managed by organization admins. Any user can submit a request using a template.

11. CRM Pipeline

Schema:
- **Contact:**
- `_id, organizationId, name, email, phone, company, ownerId (User), source, createdAt, updatedAt`.
- **Deal:**
- `_id, organizationId, name, contactId (ref Contact), value, currency, stage (e.g. "Prospect","Negotiation"), assignedTo (User), createdAt`.
- **CommLog:** Communication logs (calls/emails):
- `_id, organizationId, contactId (ref), dealId (ref), type ("email"/"call"/"meeting"), timestamp, notes, createdBy`.

> **Indexes:** Index on `organizationId`, and on `contactId`, `dealId` for queries.

Routes:
- `GET /contacts`, `POST /contacts`, `GET|PUT|DELETE /contacts/:id`.
- `GET /deals`, `POST /deals`, `GET|PUT|DELETE /deals/:id`.
- `GET /deals/:id/logs`, `POST /deals/:id/logs`.
- `GET /contacts/:id/communications`, `POST /contacts/:id/communications`.

Controllers:
- Standard CRUD, with pipeline-specific logic (e.g. moving deal to next stage).
- **getPipelineStats:** Aggregate deals by stage for dashboard.

Access Control:
- CRM data is org-scoped. Sales reps access their contacts/deals; managers see all in org.

12. Third-Party Integrations

Schema:
- **Integration:**
- `_id, organizationId, type` (e.g. `"slack"`, `"github"`, `"google_drive"`),
- `credentials`: {token, refreshToken, expiresAt}, encrypted in DB.
- `config`: { e.g. Slack workspace ID, channel ID; GitHub repo; Drive folder ID; Stripe account ID; Zoom meeting config; Zapier webhook ID}.
- `enabled`: Boolean
- `lastSync`: Date.
- **SyncLog:**
- `_id, integrationId, action (e.g. "pullIssues", "postMessage"), status ("success"/"fail"), timestamp, details`.

> Store OAuth tokens securely (encrypted) and log all integration activity.

Routes:
- `GET /integrations` – List all integrations for org.
- `POST /integrations/:type/connect` – OAuth callback endpoint or token exchange.
- `DELETE /integrations/:id` – Disable integration.
- `GET /integrations/:id/logs` – View recent sync logs.
- Custom endpoints per integration (e.g. `POST /integrations/slack/test`, `GET /integrations/github/repos`).

Controllers:
- **connectIntegration:** Handle OAuth redirect, save tokens and config.
- **syncData:** Periodic jobs or webhook handlers to sync data (e.g., GitHub issues → tasks, Stripe payments → budget).
- **logSync:** Write to SyncLog for auditing and debugging.

Access Control:
- Typically only organization admins or integration managers can configure integrations.

13. RBAC & Multi-Tenant Structure

Schema (Organization & Roles):
- **Organization:** `_id, name, domain, createdAt`. All data (Projects, Tasks, etc.) references `organizationId` to enforce tenant isolation.
- **User-Role assignments:** Simplest: in `User.roles` we encode per-org roles or add a `role` subdoc: `{ organizationId, roleName }`.
Alternatively, a `RoleAssignment` collection: `{ _id, userId, organizationId, role (e.g. "Admin","Editor","Viewer"), createdAt }`.

> **Tenant Isolation:** Each document includes its `organizationId`. All queries filter on this to prevent cross-tenant access [4] [3]. This enforces data segregation [8].

Access Control Summary:
- Implement middleware that injects `req.user` from auth token (Clerk JWT) and checks `organizationId`.
- Use RBAC policies: e.g. "Admins can manage all, Editors can modify content, Viewers read-only".
- For multi-org users, check the organization context of each request.

> "A multi-tenant system needs to ensure that users from one organization cannot access another organization's data while also managing different permission levels within the same company." [3] This is achieved by scoping all models to `organizationId` and enforcing role checks per endpoint. RBAC provides a clean separation of concerns [9].

Indexing & Performance:
- Add indexes on `(organizationId, [other fields])` for query efficiency.
- Monitor query patterns and add compound indexes as needed [2].

Multi-Tenancy Best Practice:
- Each collection can have a compound shard key including `organizationId` for horizontal scaling if needed.
- Use middleware to automatically add `organizationId` filter to all queries (see [18†L110-L113]).

Conclusion: This design provides a modular backend with clear schemas and routes for each domain. Sensitive operations are protected by RBAC and tenant scoping. Mongoose schemas use `timestamps: true` for auditing [1], and MongoDB best practices (indexed fields, atomic sub-doc updates) ensure performance [2] [7].

Sources:
Design principles are informed by MongoDB schema best practices (indexed queries, embedding vs referencing) [6] [2], and multi-tenant RBAC guidance [4] [3].

---

[1] Mongoose v8.14.2: Mongoose Timestamps
https://mongoosejs.com/docs/timestamps.html

[2] [7] Data Modeling - Database Manual v8.0 - MongoDB Docs
https://www.mongodb.com/developer/products/mongodb/mongodb-schema-design-best-practices/

[3] [9] How to Implement RBAC in an Express.js Application
https://www.permit.io/blog/how-to-implement-rbac-in-an-expressjs-application

[4] Best Practices for Multi-Tenant Authorization
https://www.permit.io/blog/best-practices-for-multi-tenant-authorization

[5] Model Tree Structures - Database Manual v8.0 - MongoDB Docs
https://docs.mongodb.com/manual/applications/data-models-tree-structures/

[6] Tasks and subtasks - Working with Data - MongoDB Developer Community Forums
https://www.mongodb.com/community/forums/t/tasks-and-subtasks/147521

[8] Build a Multi-Tenant Architecture in MongoDB | GeeksforGeeks
https://www.geeksforgeeks.org/build-a-multi-tenant-architecture-in-mongodb/