ChatGPT

Coordination Module Phase 1 – UI + Backend Specification

Phase 1: Database Schema

```
// File: apps/api/db/schema.ts (Phase 1 – Database Schema using Drizzle ORM)
import { pgTable, uuid, text, timestamp, boolean, unique } from "drizzle-orm/pg-
core";

// **Tasks Table** – Stores core task info. Every task is linked to one primary
project and department.
export const tasks = pgTable("tasks", {
  id: uuid("id").defaultRandom().primaryKey().notNull(),
  title: text("title"),                      // Task title (allow null for voice/
email tasks, can be filled later)
  description: text("description"),          // Detailed description (optional,
can be added/edited later)
  status: text("status").default("TO_DO").notNull(),
  // Status lifecycle: one of 'TO_DO', 'IN_PROGRESS', 'OUTWARD', 'STALLED',
'REVIEW', 'COMPLETE', 'CANCELLED'
  projectId: uuid("project_id").notNull(),   // Primary project reference (FK to
projects.id)
  department: text("department").notNull(), // Primary department tag (e.g.
"Operations", "Accounting", etc.)
  ballInCourtUserId: uuid("ball_in_court_user_id"), // Current responsible user
(FK to users.id, null if unassigned/external)
  ballInCourtNote: text("ball_in_court_note"),      // Optional note from the
last ball pass (if provided)
  origin: text("origin").default("UI").notNull(),   // Origin of task creation:
'UI', 'email', or 'voice'
  voiceUrl: text("voice_url"),               // File path or URL of voice
recording (if task created via voice)
  voiceTranscript:
text("voice_transcript"),// Transcript text of voice note (if available)
  createdBy: uuid("created_by"),             // User who created task (FK to
users.id, null if external email/voice)
  createdAt: timestamp("created_at", { withTimezone:
true }).defaultNow().notNull(),
  updatedAt: timestamp("updated_at", { withTimezone:
true }).defaultNow().notNull(),
}, (table) => {
  // (Optional) Ensure status and department are within allowed sets
  unique("tasks_status_check").on(sql`${table.status} IN
('TO_DO','IN_PROGRESS','OUTWARD','STALLED','REVIEW','COMPLETE','CANCELLED')`),
  unique("tasks_dept_check").on(sql`${table.department} IN
('Operations','Procurement','Accounting','Service','Estimating','Scheduling')`),
});

// **Task Link Tables** – Support linking tasks to multiple projects or
departments (beyond the primary ones).
export const tasksProjects = pgTable("tasks_projects", {
  taskId: uuid("task_id").notNull(),      // Task ID (FK to tasks.id)
  projectId:
uuid("project_id").notNull() // Additional linked project ID (FK to projects.id)
}, (table) => {
  table.primaryKey(["task_id","project_id"]);
});

export const tasksDepartments = pgTable("tasks_departments", {
  taskId: uuid("task_id").notNull(),        // Task ID (FK to tasks.id)
  department: text("department").notNull()  // Additional department tag
}, (table) => {
  table.primaryKey(["task_id","department"]);
});

// **Task Activity Log** – Records all task events: status changes, ball passes,
comments.
export const taskLogs = pgTable("task_logs", {
  id: uuid("id").defaultRandom().primaryKey().notNull(),
  taskId: uuid("task_id").notNull(),        // Related task (FK to tasks.id)
  type: text("type").notNull(),             // Event type: 'STATUS',
'BALL_PASS', 'COMMENT'
  createdBy:
uuid("created_by"),            // User who performed the action (FK to users.id,
null for external/system)
  oldStatus: text("old_status"),            // Previous status (for status
change events)
  newStatus: text("new_status"),            // New status (for status change
events)
  fromUserId: uuid("from_user_id"),         // Prior ball holder (for ball pass
events)
  toUserId: uuid("to_user_id"),             // New ball holder (for ball pass
events)
  body: text("body"),                      // Content of comment or note (for
COMMENT or BALL_PASS events)
  createdAt: timestamp("created_at", { withTimezone:
true }).defaultNow().notNull(),
}, (table) => {
  // Indexes to query by task
  unique("idx_task_logs_task_time").on(table.taskId, table.createdAt)
});
```

Schema Notes: Each task has a **primary project** (`projectId`) and a **primary department**. The `tasksProjects` and `tasksDepartments` tables allow tagging a task to multiple projects or departments if needed. The `status` field is constrained to the lifecycle values **TO_DO**, **IN_PROGRESS**, **OUTWARD**, **STALLED**, **REVIEW**, **COMPLETE**, **CANCELLED**. The **Ball-in-Court** fields (`ballInCourtUserId` and optional `ballInCourtNote`) track who currently holds responsibility and any note left when the "ball" was passed. These are independent of `status` – changing status doesn't automatically change the ball-in-court user, and vice versa.

The **origin** field tracks how a task was created (`UI` for normal UI input, `email` for email intake, `voice` for voice intake). Voice-originated tasks store an audio file reference (`voiceUrl`) and a transcript (`voiceTranscript`). Tasks can initially be created with minimal info (even a null title/description for voice/email cases) and then updated later.

All significant actions are recorded in **taskLogs**: status changes (with old/new status), ball passes (with from/to user and note), and comments (with comment text in `body`). This provides a full activity history for each task. (Additional tables like `task_comments` or `ball_history` could be used, but here we unify events in one log for simplicity.)

Phase 2: Backend API Route Scaffolds (Express)

```
// File: apps/api/src/routes/tasks.ts (Phase 2 – Express route scaffolds for
Task module)
import express from 'express';
export const tasksRouter = express.Router();

// **Task CRUD Routes**
tasksRouter.get('/projects/:projectId/tasks', (req, res) => {
  // TODO: Fetch all tasks for the given projectId
  res.json([]);  // placeholder
});
tasksRouter.post('/projects/:projectId/tasks', (req, res) => {
  // TODO: Create a new task under the given projectId
  res.status(201).json({});  // placeholder
});
tasksRouter.get('/tasks', (req, res) => {
  // TODO: Fetch tasks (optionally filter by ?projectId, etc.)
  res.json([]);  // placeholder
});
tasksRouter.get('/tasks/:id', (req, res) => {
  // TODO: Fetch a single task detail by ID
  res.json({});  // placeholder
});
```

```
tasksRouter.put('/tasks/:id', (req, res) => {
  // TODO: Update task fields (partial updates allowed)
  res.json({});  // placeholder
});

// **Task Action Routes**
tasksRouter.post('/tasks/:id/status', (req, res) => {
  // TODO: Update task status (with proper lifecycle checks, if any)
  res.json({});  // placeholder
});
tasksRouter.post('/tasks/:id/ball-pass', (req, res) => {
  // TODO: Pass ball-in-court to another user (with optional note)
  res.json({});  // placeholder
});

// **Task Comments Routes**
tasksRouter.get('/tasks/:id/comments', (req, res) => {
  // TODO: Fetch all comments for task (activity log filtered to comments)
  res.json([]);  // placeholder
});
tasksRouter.post('/tasks/:id/comments', (req, res) => {
  // TODO: Add a new comment to the task
  res.status(201).json({});  // placeholder
});
```

Explanation: We define an Express router for task-related endpoints.

- CRUD endpoints:
- `GET /api/projects/:projectId/tasks` – list all tasks in a given project.
- `POST /api/projects/:projectId/tasks` – create a new task in the given project.
- `GET /api/tasks` – list tasks (optionally filtered, e.g. by project via query param).
- `GET /api/tasks/:id` – get detailed info for a specific task.
- `PUT /api/tasks/:id` – update an existing task's details (title, description, etc. Partial updates supported).

- Action endpoints:

- `POST /api/tasks/:id/status` – change a task's status (e.g. move from TO_DO to IN_PROGRESS).
- `POST /api/tasks/:id/ball-pass` – transfer the "ball-in-court" (responsibility) to another user, optionally including a note.

- Comment endpoints:

- `GET /api/tasks/:id/comments` – get all comments for a task.

- `POST /api/tasks/:id/comments` – add a new comment to a task.

At this stage, each handler is just a stub (sending placeholder responses). In **Phase 4**, we will implement the logic (database operations, validations, activity logging) for these routes.

(Integration Note:) In your Express server setup (e.g. `app.js` or `index.ts`), mount this router under the API path, for example:

```
import { tasksRouter } from './routes/tasks';
app.use('/api', tasksRouter);
```

This will ensure the endpoints are served as `/api/tasks/...` and `/api/projects/...` as specified.

Phase 3: Frontend Layout (React Pages & Key Components)

In this phase, we scaffold the React frontend for the Coordination module. The UI is optimized for desktop (using Tailwind CSS classes for layout), and structured to accommodate future mobile and guest-access needs. We'll create pages for listing tasks (within a project context), viewing task details, and a modal for creating tasks. Key interactive components like a status selector and a ball-in-court chip are also included.

```
// File: apps/coordination_ui/src/pages/ProjectDetailPage.tsx
import React, { useState, useEffect } from 'react';
import { useParams, useNavigate } from 'react-router-dom';

export function ProjectDetailPage() {
  const navigate = useNavigate();
  const { projectId } = useParams();  // Expect route like /projects/:projectId
  const [project, setProject] = useState(null);
  const [tasks, setTasks] = useState([]);
  const [error, setError] = useState('');
  const [showNewTaskModal, setShowNewTaskModal] = useState(false);
  const [newTask, setNewTask] = useState({
    title: '',
    department: '',
    priority: 'normal',    // example extra field (could be ignored by backend
if not supported yet)
    // (We include priority as a placeholder; statuses are handled separately.)
  });

  useEffect(() => {
    if (projectId) {
      // Load project details and tasks for that project
      fetch(`/api/projects/${projectId}`)
        .then(res => res.json())
        .then(data => setProject(data))
        .catch(err => console.error('Failed to load project', err));
      fetch(`/api/projects/${projectId}/tasks`)
        .then(res => res.json())
        .then(data => setTasks(data))
        .catch(err => console.error('Failed to load tasks', err));
    }
  }, [projectId]);

  const handleCreateTask = (e) => {
    e.preventDefault();
    if (!newTask.title || !newTask.department) {
      setError('Title and Department are required');
      return;
    }
    // Create task via API
    fetch(`/api/projects/${projectId}/tasks`, {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify(newTask)
    })
      .then(res => {
        if (!res.ok) throw new Error(`Failed to create task (status $
{res.status})`);
        return res.json();
      })
      .then(createdTask => {
        setTasks([...tasks, createdTask]);   // add to list
        setShowNewTaskModal(false);
        setNewTask({ title: '', department: '', priority: 'normal' });
        setError('');
      })
      .catch(err => {
        console.error(err);
        setError(err.message);
      });
  };

  return (
    <div className="p-4 max-w-5xl mx-auto">
      {/* Back link */}
      <div className="mb-4">
        <a href="/" className="text-blue-600 text-sm">&larr; Back to Projects</
a>
      </div>

      {/* Project Header */}
      {project && (
        <h1 className="text-2xl font-bold mb-6">
```

```
          {project.name || 'Project'} – Tasks
        </h1>
      )}

      {/* Error Banner */}
      {error && (
        <div className="bg-red-100 text-red-800 p-3 rounded mb-4">
          {error}
        </div>
      )}

      {/* Tasks List */}
      <div className="mb-6">
        {tasks.length === 0 ? (
          <p className="text-gray-600">No tasks yet. Create one below!</p>
        ) : (
          <div className="grid gap-4">
            {tasks.map(task => (
              <div key={task.id}
                   className="bg-white border border-gray-200 rounded p-4
cursor-pointer hover:shadow"
                   onClick={() => navigate(`/tasks/${task.id}`)}>
                <div className="flex justify-between">
                  <div>
                    <h3 className="text-lg font-semibold">{task.title ||
<em>(Untitled Task)</em>}</h3>
                    <p className="text-sm text-gray-600">
                      Dept: {task.department} • Status: {task.status}
                    </p>
                  </div>
                  {/* Example Ball-in-Court Chip usage */}
                  {task.ballInCourtUserId && (
                    <BallInCourtChip userId={task.ballInCourtUserId} />
                  )}
                </div>
              </div>
            ))}
          </div>
        )}
      </div>

      {/* New Task Button */}
      <button
        onClick={() => setShowNewTaskModal(true)}
        className="bg-green-600 text-white px-4 py-2 rounded shadow">
        + New Task
      </button>
      {/* New Task Modal */}
      {showNewTaskModal && (
        <div className="fixed inset-0 bg-black bg-opacity-50 flex items-center
justify-center z-50">
          <div className="bg-white rounded p-6 w-full max-w-md">
            <h2 className="text-xl font-bold mb-4">Create New Task</h2>
            <form onSubmit={handleCreateTask} className="space-y-4">
              <div>
                <label className="block text-sm font-medium mb-1">Title</label>
                <input
                  type="text"
                  className="w-full border border-gray-300 rounded px-3 py-2"
                  value={newTask.title}
                  onChange={e => setNewTask({ ...newTask, title:
e.target.value })}
                  placeholder="Task title (optional for voice tasks)"
                />
              </div>
              <div>
                <label className="block text-sm font-medium mb-1">Department</
label>
                <select
                  className="w-full border border-gray-300 rounded px-3 py-2"
                  value={newTask.department}
                  onChange={e => setNewTask({ ...newTask, department:
e.target.value })}
                >
                  <option value="">-- Select Department --</option>
                  <option>Operations</option>
                  <option>Procurement</option>
                  <option>Accounting</option>
                  <option>Service</option>
                  <option>Estimating</option>
                  <option>Scheduling</option>
                </select>
              </div>
              {/* Additional fields like priority, attachments could go here */}
              <div className="flex justify-end items-center">
                <button type="button" onClick={() =>
setShowNewTaskModal(false)} className="text-gray-600 mr-4">Cancel</button>
                <button type="submit" disabled={newTask.title === '' ||
newTask.department === ''}
                        className="bg-blue-600 text-white px-4 py-2 rounded
disabled:opacity-50">
                  Create Task
                </button>
              </div>
            </form>
          </div>
        </div>
      )}
    </div>
  );
}

// A small component to display ball-in-court user (current responsible)
function BallInCourtChip({ userId }) {
  // (In a real app, we'd look up the user's name; here we just show the ID or a
placeholder)
  return (
    <span className="bg-blue-100 text-blue-800 text-xs font-medium px-2 py-1
rounded self-start">
      Ball with User {userId}
    </span>
  );
}
```

```
// File: apps/coordination_ui/src/pages/TaskDetailPage.tsx
import React, { useState, useEffect } from 'react';
import { useParams } from 'react-router-dom';

export function TaskDetailPage() {
  const { taskId } = useParams();
  const [task, setTask] = useState(null);
  const [comments, setComments] = useState([]);
  const [newComment, setNewComment] = useState('');
  const [ballPassTarget, setBallPassTarget] = useState('');
  const [ballPassNote, setBallPassNote] = useState('');
  const [error, setError] = useState('');

  useEffect(() => {
    if (!taskId) return;
    // Fetch task details
    fetch(`/api/tasks/${taskId}`)
      .then(res => res.json())
      .then(data => setTask(data))
      .catch(err => console.error('Failed to load task', err));
    // Fetch comments (activity log entries of type COMMENT)
    fetch(`/api/tasks/${taskId}/comments`)
      .then(res => res.json())
      .then(data => setComments(data))
      .catch(err => console.error('Failed to load comments', err));
  }, [taskId]);
  const handleStatusChange = (e) => {
    const newStatus = e.target.value;
    if (!task) return;
    if (newStatus === task.status) return;
    // Update status via API
    fetch(`/api/tasks/${task.id}/status`, {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ status: newStatus })
    })
      .then(res => {
        if (!res.ok) throw new Error(`Status update failed (status $
{res.status})`);
        return res.json();
      })
      .then(updatedTask => {
        setTask(updatedTask);
        setError('');
      })
      .catch(err => setError(err.message));
  };

  const handleAddComment = (e) => {
    e.preventDefault();
    if (!newComment.trim()) return;
    // Post new comment
    fetch(`/api/tasks/${task.id}/comments`, {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ text: newComment })
    })
      .then(res => {
        if (!res.ok) throw new Error('Failed to add comment');
        return res.json();
      })
      .then(createdComment => {
        setComments([...comments, createdComment]);
        setNewComment('');
        setError('');
      })
      .catch(err => setError(err.message));
  };

  const handleBallPass = (e) => {
    e.preventDefault();
    if (!ballPassTarget.trim()) {
      setError('Please specify a user ID to pass the ball');
      return;
    }
    // Call ball-pass API
    fetch(`/api/tasks/${task.id}/ball-pass`, {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ toUserId: ballPassTarget, note: ballPassNote })
    })
      .then(res => {
        if (!res.ok) throw new Error('Failed to pass ball');
        return res.json();
      })
      .then(updatedTask => {
        setTask(updatedTask);
        setBallPassTarget('');
        setBallPassNote('');
        setError('');
      })
      .catch(err => setError(err.message));
  };

  if (!task) {
    return <div className="p-4">Loading task...</div>;
  }

  return (
    <div className="p-4 max-w-4xl mx-auto">
      {/* Task Header */}
      <div className="mb-4">
        <h2 className="text-2xl font-bold">{task.title || 'Untitled Task'}</h2>
        <p className="text-sm text-gray-600">
          Project: {task.projectId} • Department: {task.department}
        </p>
        {task.voiceUrl && (
          <div className="mt-2">
            {/* If this is a voice task, allow audio playback */}
            <audio controls src={task.voiceUrl} className="w-full mb-1" />
            {task.voiceTranscript && (
              <p className="text-sm italic text-gray-700">Transcript:
{task.voiceTranscript}</p>
            )}
          </div>
        )}
      </div>

      {/* Error Message */}
      {error && (
        <div className="bg-red-100 text-red-800 p-3 rounded mb-4">
          {error}
        </div>
      )}

      {/* Status Control */}
      <div className="mb-4">
        <label className="block font-medium mb-1">Status:</label>
        <StatusSelect value={task.status} onChange={handleStatusChange} />
      </div>

      {/* Ball-in-Court Section */}
      <div className="mb-6">
        <h3 className="font-medium mb-2">Current Ball-in-Court:</h3>
        {task.ballInCourtUserId ? (
          <p className="mb-2">This task is currently with <strong>User
{task.ballInCourtUserId}</strong>
            {task.ballInCourtNote && (<span> – Note: <em>{task.ballInCourtNote}
</em></span>)}
          </p>
        ) : (
          <p className="mb-2"><em>No one currently holds the ball for this
task.</em></p>
        )}
        <form onSubmit={handleBallPass} className="flex items-center gap-2">
          <input
            type="text"
            placeholder="Target User ID"
            value={ballPassTarget}
            onChange={e => setBallPassTarget(e.target.value)}
            className="border border-gray-300 rounded px-2 py-1 text-sm"
          />
          <input
            type="text"
            placeholder="Optional note"
            value={ballPassNote}
            onChange={e => setBallPassNote(e.target.value)}
            className="border border-gray-300 rounded px-2 py-1 text-sm flex-1"
          />
          <button type="submit" className="bg-blue-600 text-white text-sm px-3
py-1 rounded">
            Pass Ball
          </button>
        </form>
      </div>

      {/* Comments Section */}
      <div className="mb-6">
        <h3 className="font-medium mb-2">Comments:</h3>
        {comments.length === 0 ? (
          <p className="text-gray-600 text-sm">No comments yet.</p>
        ) : (
          <div className="space-y-2 mb-4">
            {comments.map(comment => (
              <div key={comment.id}
className="bg-gray-50 border border-gray-200 rounded p-2 text-sm">
                <span className="text-gray-700">{comment.body}</span>
                <div className="text-xs text-gray-500">
                  — by User {comment.createdBy || 'Unknown'} on {new
Date(comment.createdAt).toLocaleString()}
                </div>
              </div>
            ))}
          </div>
        )}

        {/* Add Comment Form */}
        <form onSubmit={handleAddComment} className="flex items-center gap-2">
          <input
            type="text"
            placeholder="Type a comment..."
            value={newComment}
            onChange={e => setNewComment(e.target.value)}
            className="border border-gray-300 rounded px-3 py-1 flex-1 text-sm"
          />
          <button type="submit" className="bg-gray-800 text-white text-sm px-4
py-1 rounded">
            Add
          </button>
        </form>
      </div>
    </div>
  );
}

// Component for status selection (dropdown)
function StatusSelect({ value, onChange }) {
  const statusOptions =
['TO_DO','IN_PROGRESS','OUTWARD','STALLED','REVIEW','COMPLETE','CANCELLED'];
  return (
    <select value={value} onChange={onChange} className="border border-gray-300
rounded px-2 py-1">
      {statusOptions.map(status => (
        <option key={status} value={status}>
          {status.replace('_', ' ')}
        </option>
      ))}
    </select>
  );
}
```

UI Explanation:

- **ProjectDetailPage:** Shows a list of tasks for a specific project. It fetches project info and tasks from the backend. Each task card displays the title, department, status, and (if set) a **Ball-in-Court chip** indicating who currently has responsibility. There is a **"+ New Task"** button that opens a modal form to create a new task. The form collects at least a Title and Department (required for creation). Additional fields like priority are included as placeholders. On submission, it calls the backend to create the task and then updates the local list. (We ensure tasks always have a project and department before creation. If voice tasks are created later via this UI, Title can be left blank and filled by transcript afterwards.)

- **TaskDetailPage:** Shows details for a single task. It fetches the task data and that task's comments (from the activity log) on load. It displays the task's title, project ID, department, and if it's a voice task, an audio player and transcript. The **Status** is shown with a dropdown (`StatusSelect` component) to change it – selecting a new status triggers `handleStatusChange`, which calls the `/api/tasks/:id/status` endpoint and updates the UI with the new status. The **Ball-in-Court** section displays who currently holds the task (the current responsible user) and any note from the last handoff. It provides a form to **pass the ball** to another user: the user enters a target user ID and an optional note, and submits to call the `/api/tasks/:id/ball-pass` endpoint. After a successful pass, the UI updates the task's current owner and clears the form. We do not tie ball-in-court changes to status changes – for example, a task could remain "IN_PROGRESS" but change hands between users, or be marked "OUTWARD" while with an external party – illustrating that the status is independent.

- The **Comments** section in TaskDetailPage lists all comments (if any) with their author (user ID for now) and timestamp. A form at the bottom allows adding a new comment, which calls the `/api/tasks/:id/comments` endpoint and on success appends the new comment to the list. Comments and other activities (status changes, ball passes) are all logged in the backend, but here we only fetch and display the comment entries for simplicity. In a future iteration, we could merge all `taskLogs` entries into a single chronological timeline UI.

All components use Tailwind CSS classes (e.g. `max-w-5xl mx-auto`, `bg-gray-50`, etc.) for styling, ensuring the layout is responsive-ready. The design is optimized for desktop (e.g. modal dialogs, multi-column layouts) but uses flexible container sizes and responsive utility classes, making it easier to adapt for mobile later. We avoid fixed pixel widths; instead using relative units and max-width containers so the UI can scale. Also, by using React Router (with `useNavigate` and `useParams`), the structure can support direct linking (useful for future guest access to specific task URLs).

Phase 4: Implementing Commenting & Ball-Pass Logic (Backend)

Now we fill in the backend logic for the task routes – particularly for adding comments and passing the ball, as well as updating status and creating tasks. This includes database operations and writing to the activity log. We'll use the schema from Phase 1 and assume a configured Drizzle ORM (`db`) and authentication middleware that provides `req.user.id`.

```
// File: apps/api/src/routes/tasks.ts  (Phase 4 – Implemented logic)
import express from 'express';
import { db } from '../db';  // assume we have a database instance
import { tasks, taskLogs, tasksProjects, tasksDepartments } from '../db/schema';
import { eq, and } from 'drizzle-orm';  // Drizzle query helpers

export const tasksRouter = express.Router();

// Helper: fetch a task by ID (returns undefined if not found)
async function findTaskById(taskId) {
  const [task] = await db.select().from(tasks).where(eq(tasks.id, taskId));
  return task;
}

// **GET /projects/:projectId/tasks** – list tasks for a project
tasksRouter.get('/projects/:projectId/tasks', async (req, res) => {
  const projectId = req.params.projectId;
  try {
    const projectTasks = await
db.select().from(tasks).where(eq(tasks.projectId, projectId));
    return res.json(projectTasks);
  } catch (err) {
    console.error('Error fetching tasks for project', err);
    return res.status(500).json({ error: 'Failed to fetch tasks' });
  }
});

// **POST /projects/:projectId/tasks** – create new task in a project
tasksRouter.post('/projects/:projectId/tasks', async (req, res) => {
  const projectId = req.params.projectId;
  const userId = req.user?.id;  // assume user authentication middleware
  const { title, description, department, priority, linkedProjectIds = [],
linkedDepartments = [] } = req.body;
  if (!department) {
    return res.status(400).json({ error: 'Primary department is required' });
  }
  try {
    const taskData: any = {
      projectId,
      title: title || null,
      description: description || null,
      department,
      status: 'TO_DO',
      ballInCourtUserId: userId || null,  // default ball-in-court to creator
(if available)
      ballInCourtNote: null,
      createdBy: userId || null,
      // origin defaults to 'UI' by schema; no need to set unless overriding
    };
    // Insert the new task
    const [newTask] = await db.insert(tasks).values(taskData).returning();
    const taskId = newTask.id;
    // Link additional projects if provided
    if (Array.isArray(linkedProjectIds) && linkedProjectIds.length > 0) {
      for (const pid of linkedProjectIds) {
        if (pid && pid !== projectId) {
          await db.insert(tasksProjects).values({ taskId, projectId: pid });
        }
      }
    }
    // Link additional departments if provided
    if (Array.isArray(linkedDepartments) && linkedDepartments.length > 0) {
      for (const dept of linkedDepartments) {
        if (dept && dept !== department) {
          await db.insert(tasksDepartments).values({ taskId, department:
dept });
        }
      }
    }
    // (Optionally, log task creation as an event)
    // Respond with the created task
    return res.status(201).json(newTask);
  } catch (err) {
    console.error('Error creating task', err);
    return res.status(500).json({ error: 'Failed to create task' });
  }
});

// **GET /tasks** – list tasks (optional filtering)
tasksRouter.get('/tasks', async (req, res) => {
  const { projectId } = req.query;
  try {
    let result;
    if (projectId) {
      // filter by project if query param is provided
      result = await db.select().from(tasks).where(eq(tasks.projectId,
projectId));
    } else {
      result = await
db.select().from(tasks).limit(50); // e.g., limit to 50 for safety
    }
    return res.json(result);
  } catch (err) {
    console.error('Error fetching tasks', err);
    return res.status(500).json({ error: 'Failed to fetch tasks' });
  }
});

// **GET /tasks/:id** – get task detail
tasksRouter.get('/tasks/:id', async (req, res) => {
  const taskId = req.params.id;
  try {
    const task = await findTaskById(taskId);
    if (!task) return res.status(404).json({ error: 'Task not found' });
    return res.json(task);
  } catch (err) {
    console.error('Error fetching task', err);
    return res.status(500).json({ error: 'Failed to fetch task' });
  }
});

// **PUT /tasks/:id** – update task (partial info)
tasksRouter.put('/tasks/:id', async (req, res) => {
  const taskId = req.params.id;
  const { title, description, department, linkedProjectIds, linkedDepartments }
= req.body;
  try {
    const task = await findTaskById(taskId);
    if (!task) return res.status(404).json({ error: 'Task not found' });
    // Prepare fields to update (excluding status or ball-in-court here for
safety)
    const updateData: any = {};
    if (title !== undefined) updateData.title = title;
    if (description !== undefined) updateData.description = description;
    if (department !== undefined) updateData.department = department;
    // (We could allow changing primary project via this route if needed)
    if (Object.keys(updateData).length > 0) {
      updateData.updatedAt = new Date();
      await db.update(tasks).set(updateData).where(eq(tasks.id, taskId));
    }
    // Handle updates to linked projects
    if (Array.isArray(linkedProjectIds)) {
      // remove all existing extra links and add new ones
      await db.delete(tasksProjects).where(eq(tasksProjects.taskId, taskId));
      for (const pid of linkedProjectIds) {
        if (pid && pid !== task.projectId) {
          await db.insert(tasksProjects).values({ taskId, projectId: pid });
        }
      }
    }
    // Handle updates to linked departments
    if (Array.isArray(linkedDepartments)) {
      await db.delete(tasksDepartments).where(eq(tasksDepartments.taskId,
taskId));
      for (const dept of linkedDepartments) {
        if (dept && dept !== task.department) {
          await db.insert(tasksDepartments).values({ taskId, department:
dept });
        }
      }
    }
    const updatedTask = await findTaskById(taskId);
    return res.json(updatedTask);
  } catch (err) {
    console.error('Error updating task', err);
    return res.status(500).json({ error: 'Failed to update task' });
  }
});

// **POST /tasks/:id/status** – update task status
tasksRouter.post('/tasks/:id/status', async (req, res) => {
  const taskId = req.params.id;
  const { status: newStatus } = req.body;
  const userId = req.user?.id;
  if (!newStatus) {
    return res.status(400).json({ error: 'New status is required' });
  }
  try {
    const task = await findTaskById(taskId);
    if (!task) return res.status(404).json({ error: 'Task not found' });
    const oldStatus = task.status;
    if (newStatus === oldStatus) {
      return res.json(task); // no change
    }
    // (Optional: enforce valid status transitions here if needed)
    // Update status
    const [updatedTask] = await db.update(tasks)
      .set({ status: newStatus, updatedAt: new Date() })
      .where(eq(tasks.id, taskId))
      .returning();
    // Log the status change event
    await db.insert(taskLogs).values({
      taskId,
      type: 'STATUS',
      createdBy: userId || null,
      oldStatus: oldStatus,
      newStatus: newStatus,
      body: null,
      fromUserId: null,
      toUserId: null
    });
    return res.json(updatedTask);
  } catch (err) {
    console.error('Error updating status', err);
    return res.status(500).json({ error: 'Failed to update status' });
  }
});

// **POST /tasks/:id/ball-pass** – pass ball-in-court to another user
tasksRouter.post('/tasks/:id/ball-pass', async (req, res) => {
  const taskId = req.params.id;
  const { toUserId, note } = req.body;
  const userId = req.user?.id;
  if (!toUserId) {
    return res.status(400).json({ error:
'Target user ID is required to pass the ball' });
  }
  try {
    const task = await findTaskById(taskId);
    if (!task) return res.status(404).json({ error: 'Task not found' });
    const fromUserId = task.ballInCourtUserId || null;
    // Update the task's ball-in-court fields
    const [updatedTask] = await db.update(tasks).set({
        ballInCourtUserId: toUserId,
        ballInCourtNote: note || null,
        updatedAt: new Date()
      })
      .where(eq(tasks.id, taskId))
      .returning();
    // Log the ball-pass event
    await db.insert(taskLogs).values({
      taskId,
      type: 'BALL_PASS',
      createdBy: userId || null,
      oldStatus: null,
      newStatus: null,
      fromUserId: fromUserId,
      toUserId: toUserId,
      body: note || null
    });
    // (Optionally, could trigger notification to new ball owner here)
    // Note: Ball pass does *not* change task status in this design.
    // If needed, an auto-return can be implemented by tracking ball history:
    // we have fromUserId stored, so the system could auto-pass back after a
timeout.
    return res.json(updatedTask);
  } catch (err) {
    console.error('Error passing ball', err);
    return res.status(500).json({ error: 'Failed to pass ball' });
  }
});

// **GET /tasks/:id/comments** – retrieve comments for a task
tasksRouter.get('/tasks/:id/comments', async (req, res) => {
  const taskId = req.params.id;
  try {
    // Fetch all log entries of type 'COMMENT' for this task, oldest first
    const comments = await db.select().from(taskLogs)
      .where(and(eq(taskLogs.taskId, taskId), eq(taskLogs.type, 'COMMENT')))
      .orderBy(taskLogs.createdAt);
    return res.json(comments);
  } catch (err) {
    console.error('Error fetching comments', err);
    return res.status(500).json({ error: 'Failed to fetch comments' });
  }
});

// **POST /tasks/:id/comments** – add a new comment
tasksRouter.post('/tasks/:id/comments', async (req, res) => {
  const taskId = req.params.id;
  const userId = req.user?.id;
  const { text } = req.body;
  if (!text) {
    return res.status(400).json({ error: 'Comment text is required' });
  }
  try {
    const task = await findTaskById(taskId);
    if (!task) return res.status(404).json({ error: 'Task not found' });
    // Insert new comment into log
    const [logEntry] = await db.insert(taskLogs).values({
      taskId,
      type: 'COMMENT',
      createdBy: userId || null,
      body: text,
      oldStatus: null,
      newStatus: null,
      fromUserId: null,
      toUserId: null
    }).returning();
    return res.status(201).json(logEntry);
  } catch (err) {
    console.error('Error adding comment', err);
    return res.status(500).json({ error: 'Failed to add comment' });
  }
});
```

Backend Logic Details:

- **Task Creation (`POST /projects/:projectId/tasks`):** We require at least a department (and in UI we ensure title & department are provided for normal tasks). The new task is inserted with default status `TO_DO`. We set the **ball-in-court** to the creator by default (so the creator is responsible until reassigned). We also handle optional arrays `linkedProjectIds` and `linkedDepartments`: for each extra project or department provided, we create entries in the join tables. (Primary project and department are stored on the task itself; we avoid duplicating them in the link tables.) The response returns the newly created task. *(Note: In this implementation, the task's `origin` will default to "UI". Voice/email endpoints will override this field accordingly.)*

- **List Tasks (`GET /projects/:projectId/tasks` and `GET /tasks`):** These retrieve tasks from the database. We allow filtering by project ID either via the route or query parameter. In a larger system, we would implement pagination and more filters (status, assignee, etc.), but for Phase 1 this basic retrieval is sufficient. The results are returned as JSON arrays.

- **Get Task Detail (`GET /tasks/:id`):** Fetches one task by ID. We simply return the task record (which includes fields like `ballInCourtUserId`, `ballInCourtNote`, etc.). In a real app, we might join related tables (e.g. to include the project name or user names) or populate computed fields (like `ball_owner_email` as seen in the UI code), but those enhancements can be added later. Currently, the frontend uses this data directly or already has context (project info, etc.).

- **Update Task (`PUT /tasks/:id`):** Allows partial updates of a task's editable fields (title, description, department, etc.). We explicitly ignore status and ball-in-court changes here to avoid bypassing their dedicated logic – for those, the API expects clients to use the specialized endpoints. If `linkedProjectIds` or `linkedDepartments` arrays are provided, we reset the extra links for the task: first delete all existing links, then insert the new ones. This ensures the task's additional project/department associations can be changed (for example, linking it to a new project). The updated task (after changes) is returned. **Partial Info Support:** Because this route does not require all fields, users can create a task with minimal info and later call this to fill in missing details (e.g. add a description or change department).

- **Change Status (`POST /tasks/:id/status`):** This endpoint updates a task's status. We retrieve the current task to get the old status (and to ensure the task exists). If the new status is the same as current, we simply return the task (no change). Otherwise, we update the task's `status`. (In a more advanced system, we could enforce allowed transitions – e.g. not allowing going directly from TO_DO to COMPLETE – but that logic can be added as needed. Here any status change is allowed for flexibility in Phase 1.) We then insert a log entry of type `'STATUS'` recording the old and new status and who made the change. This way, the status change will appear in the task's activity history. We return the updated task object.

- **Pass Ball (`POST /tasks/:id/ball-pass`):** This is used to hand off responsibility for the task. It requires a `toUserId` (the ID of the user to whom the ball is being passed). We find the task to get the current ball holder (`fromUserId`). Then we update the task's `ballInCourtUserId` to the new user and store the optional note in `ballInCourtNote`. This note is also recorded and returned as part of the task, so the new holder can see any instructions/remarks. We then create a log entry of type `'BALL_PASS'` with `fromUserId`, `toUserId`, and the note. The `createdBy` field logs who initiated the pass (usually the current holder or a coordinator). **Auto-return support:** The design allows us to implement auto-return logic later. Since every pass is logged with from/to, the system can determine the previous holder. For example, if a task is in an OUTWARD status (perhaps handed off to an external partner) and we want it to "auto-return" after a deadline, a background job could look at the latest ball-pass log and automatically pass the ball back to the `fromUserId` (previous owner) when conditions are met. In Phase 1, we do not implement the timer/trigger, but the data model and log make it possible. The ball pass endpoint does **not** alter the task's status; the task could remain IN_PROGRESS or be marked OUTWARD by a separate status change – this separation ensures that status reflects *what* is happening with the task, while ball-in-court reflects *who* is currently responsible.

- **Get Comments (`GET /tasks/:id/comments`):** Fetches all comment events for the task. We query the `taskLogs` for entries of type `'COMMENT'` for that task, ordering by creation time. The frontend uses this to display the comment thread. (In the future, we might also fetch other event types for a full activity feed, but for now only comments are exposed via UI.)

- **Add Comment (`POST /tasks/:id/comments`):** Adds a new user comment to the task. We require a text body. We insert a `taskLogs` entry with type `'COMMENT'`, linking it to the task and setting `createdBy` to the current user. The new log entry (which includes an `id`, the comment text in `body`, the author, and timestamp) is returned in the response. The frontend uses this response to immediately display the new comment. All comments are persisted in the unified log, and since we don't separate a dedicated comments table in this design, editing or deleting comments would involve updating the log entry (not implemented in Phase 1).

Throughout these handlers, errors are caught and a 500 response is returned if something goes wrong. The logic assumes a valid authenticated user (for `req.user.id`) for internal actions. In cases of email/voice tasks where there might be no authenticated user, we allowed `createdBy` or `createdBy` in log to be null.

With these implementations, the backend will now properly handle task creation, updates, status changes, ball handoffs, and commenting – all while maintaining a complete history of actions. The frontend from Phase 3 will interact with these endpoints to provide a functional coordination UI.

Phase 5: Voice/Email Task Intake Stubs

In Phase 5, we introduce endpoints to handle task creation via **email** and **voice**. These would typically be called by an automated service (for example, an email webhook or a voice transcription service) rather than a user directly. For now, we provide stub implementations: they create tasks with the given information and mark their origin appropriately, but do not integrate with any external email server or voice recognition service in this phase. This sets the stage for full integration later.

```
// (Append to apps/api/src/routes/tasks.ts – Phase 5 voice/email intake)

// **POST /api/tasks/email-intake** – create a task from an incoming email
(stubbed)
tasksRouter.post('/tasks/email-intake', async (req, res) => {
  const { subject, body, from, projectId, department } = req.body;
  // Basic validation
  if (!projectId || !department) {
    return res.status(400).json({ error: 'projectId and department are required
for email intake' });
  }
  try {
    const title = subject && subject.trim() !== '' ? subject : 'Email Task ' +
new Date().toISOString();
    const taskData: any = {
      projectId,
      title,
      description: body?.substring(0, 1000) || null,  // use email body
(truncated) as initial description
      department,
      status: 'TO_DO',
      ballInCourtUserId: null,    // unassigned initially (or assign to a
default user if desired)
      ballInCourtNote: null,
      origin: 'email',
      createdBy: null             // no authenticated user (could map `from` to
a user if exists)
    };
    const [newTask] = await db.insert(tasks).values(taskData).returning();
    // Log creation via email (optional)
    await db.insert(taskLogs).values({
      taskId: newTask.id,
      type: 'STATUS',
      createdBy: null,
      oldStatus: null,
      newStatus: 'TO_DO',
      body: `Task created via email from ${from || 'unknown sender'}`,
    });
    return res.status(201).json(newTask);
  } catch (err) {
    console.error('Error intaking email task', err);
    return res.status(500).json({ error: 'Failed to create task from email' });
  }
});

// **POST /api/tasks/voice-intake** – create a task from a voice recording
(stubbed)
```

```
tasksRouter.post('/tasks/voice-intake', async (req, res) => {
  const { title, projectId, department, audioUrl, transcript } = req.body;
  if (!projectId || !department) {
    return res.status(400).json({ error: 'projectId and department are required
for voice intake' });
  }
  try {
    const taskTitle = title && title.trim() !== '' ? title : 'Voice Task ' +
new Date().toISOString();
    const taskData: any = {
      projectId,
      title: taskTitle,
      description: transcript ? transcript : null,  // we could use transcript
as description
      department,
      status: 'TO_DO',
      ballInCourtUserId: null,   // e.g., assign to a default coordinator or
leave unassigned
      ballInCourtNote: null,
      origin: 'voice',
      voiceUrl: audioUrl || null,
      voiceTranscript: transcript || null,
      createdBy: req.user?.id || null  // if voice input is from a logged-in
user, otherwise null
    };
    const [newTask] = await db.insert(tasks).values(taskData).returning();
    // We might trigger an async transcription service here if no transcript
provided.
    // Log creation via voice (optional note)
    await db.insert(taskLogs).values({
      taskId: newTask.id,
      type: 'STATUS',
      createdBy: taskData.createdBy,
      oldStatus: null,
      newStatus: 'TO_DO',
      body: `Task created via voice${transcript ? '' : ' (transcription
pending)'}`,
    });
    return res.status(201).json(newTask);
  } catch (err) {
    console.error('Error intaking voice task', err);
    return res.status(500).json({ error: 'Failed to create task from voice' });
  }
});
```

Intake Endpoints Explanation:

- **Email Intake (`POST /api/tasks/email-intake`):** This endpoint simulates processing an incoming email to create a task. We expect the request to provide at least a `projectId` and `department` to categorize the task (in a real integration, these might be derived from the email's recipient address or header tags). The `subject` becomes the task title (or a timestamped placeholder if empty), and the email `body` becomes the task description (truncated to a reasonable length for storage). The `origin` is set to `'email'` and `createdBy` is left `null` (since the sender might be an external user; if `from` matches an internal user email, we could look up and set `createdBy` accordingly in a future enhancement). We leave the task unassigned (`ballInCourtUserId: null`), assuming a coordinator will triage it. As a stub, we immediately insert the task into the database. We also add an activity log entry to note that the task was created via email (including the sender's email if provided). The response returns the created task. *(In a complete system, this endpoint would be secured and likely invoked by a mail webhook with proper validation.)*

- **Voice Intake (`POST /api/tasks/voice-intake`):** This endpoint handles tasks created from voice input. The request should include `projectId` and `department`, and either an `audioUrl` (pointing to the stored voice recording) or the raw audio file (in Phase 1, we assume an `audioUrl` is provided since we are not handling file uploads proper). It can also include a preliminary `transcript`. We create a task with `origin: 'voice'`. The task title can be provided or we default to "Voice Task <timestamp>" if not. If a transcript is available (for example, if voice-to-text was done on the fly), we set it on the task (and also copy it to `description` for convenience). If not, we leave `voiceTranscript` null – indicating transcription is pending – and the task description can be empty or a placeholder. The task is inserted with no one holding the ball initially (or we could assign to a default responsible user like a dispatcher — this can be decided by business rules later). We log an event that the task was created via voice (and note if transcription is pending). In a real implementation, this is where we might kick off an asynchronous transcription job if needed. The new task is returned in the response.

These intake endpoints are **stubs**: they perform basic data handling and use our existing creation logic, but do not integrate with external systems yet. They ensure the system is designed to accommodate email and voice-created tasks. The UI we built earlier does not directly call these (as they are meant for automated intakes), but any tasks created through them will appear in the normal UI lists and can be managed through the Task Detail view like other tasks. For example, a coordinator could see a new task that came from a client email or a voice message, open it in the UI, listen to the attached audio (the `voiceUrl` is stored), read the transcript, then update details or assign it to the appropriate team member.

Mobile & Guest Considerations: While these features are desktop-oriented in Phase 1, the separation of front-end and back-end concerns lays groundwork for expansion. The responsive-friendly Tailwind classes mean the UI can be adapted to smaller screens with minimal changes. Routing and permission checks will allow implementing a read-only guest view (e.g. sharing a task link with a client) by adding token-based access later on. All actions are logged and can trigger notifications, which will be important for real-time updates on mobile devices in future phases.

End of Phase 1 Specification. All code blocks above can be copied into the monorepo structure (`/apps/api` for backend, `/apps/coordination_ui` for frontend) to establish the initial Coordination module. This includes database schema definitions, Express API routes with the required logic, and React components/pages for the UI.
