perplexity

Build out the workflow in the attached document, for implementation as a microservice in a web application for a coding academy

Based on the workflow described in the attached document, here is a microservice design for the coding academy portion of the web application.

Microservice Design: Coding Academy Workflow

This document outlines the technical design for a set of microservices that implement the coding academy workflow specified in the `Goal.pdf` document[1]. The architecture is event-driven, leveraging both synchronous API calls for direct actions and asynchronous messaging for decoupled background processes[2].

1. Core Service Boundaries and Responsibilities

The workflow is decomposed into several independent microservices, each with a clear business capability[3].

Microservice	Core Responsibility	Technologies	Communication
Task Service	Manages the lifecycle of academy exercises (`Task` entities). Handles task creation, assignment, and status transitions (e.g., from `TODO` to `SUBMITTED`).	Node.js, PostgreSQL, Prisma	REST API, Publishes events
Review Service	Manages the `Review` process. Consumes submitted tasks, facilitates human review, and records outcomes (`APPROVED`/`REJECTED`)[4].	Node.js, PostgreSQL, Prisma	REST API, Subscribes to events
CI Orchestrator	Listens for webhooks from code repositories (e.g., GitHub). Triggers automated checks (linters, tests) and publishes the results as events.	Go or Node.js	Webhooks, Publishes events
User & Auth Service	Manages user profiles, roles (`Coding Student`, `Academy Reviewer`, `Super Admin`), and authentication. Issues JWTs with role claims[1].	Clerk/Auth0	REST API
Notification Service	Sends emails or in-app notifications based on workflow events (e.g., "Your submission was reviewed").	Node.js, RabbitMQ[5]	Subscribes to events

2. Data Models (PostgreSQL with Prisma)

The data models are directly derived from the `Goal.pdf` document and form the foundation of the Task and Review services[1].

```
// From Goal.pdf, adapted for the academy
enum Role {
  CODING_STUDENT
  ACADEMY_REVIEWER
  SUPER_ADMIN
  // Other internal roles omitted for clarity
}

enum TaskStatus {
  TODO          // Assigned to student
  IN_PROGRESS   // Student is working on it
  SUBMITTED     // Awaiting review
  UNDER_REVIEW  // A reviewer is actively looking at it
  APPROVED      // Reviewer approved
  REJECTED      // Reviewer rejected
}

model User {
  id        String   @id @default(cuid())
  clerkId   String   @unique // From Auth Provider
  role      Role
  xp        Int      @default(0) // For student progression
  tasks     Task[]   @relation("Assignee")
  reviews   Review[] @relation("Reviewer")
}

model Task {
  id         String     @id @default(cuid())
  title      String
  status     TaskStatus @default(TODO)
  repoUrl    String     // URL to the student's forked repository
  assignee   User       @relation("Assignee", fields:[assigneeId], references:[id])
  assigneeId String
  reviews    Review[]
}

model Review {
  id         String   @id @default(cuid())
  outcome    String   // "APPROVED" or "REJECTED"
  comment    String?
  task       Task     @relation(fields:[taskId], references:[id])
  taskId     String
  reviewer   User     @relation("Reviewer", fields:[reviewerId], references:[id])
  reviewerId String
}
```

3. Event-Driven Choreography

The services communicate asynchronously via a message broker like RabbitMQ to reduce coupling and improve resilience[2] [5].

Key Events:

- `task.submitted`: Published by the Task Service when a student submits their work. The CI Orchestrator and Review Service subscribe.
- `ci.result.passed`: Published by the CI Orchestrator. The Review Service subscribes to move the task into the reviewer's queue.
- `ci.result.failed`: Published by the CI Orchestrator. The Task Service subscribes to move the task status back to `IN_PROGRESS` and the Notification Service alerts the student.
- `review.created`: Published by the Review Service when an `APPROVED` or `REJECTED` decision is made. The Task Service subscribes to update task status and award XP, and the Notification Service alerts the student.

4. API Endpoints

The workflow is driven by a REST API, with access controlled by JWT roles.

Method	Route	Description	Authorized Role(s)
`POST`	`/tasks/:id/submit`	A student submits their completed exercise for review.	`CODING_STUDENT` (must be assignee)
`GET`	`/reviews/queue`	Fetches a list of submitted tasks awaiting review. This populates the reviewer's dashboard[4].	`ACADEMY_REVIEWER`, `SUPER_ADMIN`
`POST`	`/reviews`	An academy reviewer submits their approval or rejection of a task[4].	`ACADEMY_REVIEWER`, `SUPER_ADMIN`
`PATCH`	`/admin/tasks/:id`	A super admin manually overrides a task's status or reassigns it[1] [5].	`SUPER_ADMIN`
`PATCH`	`/admin/reviews/:id`	A super admin overrides a review outcome or grade[1].	`SUPER_ADMIN`
`POST`	`/admin/impersonate`	A super admin gets a token to act as another user for debugging purposes[1] [5].	`SUPER_ADMIN`

5. Coding Academy Workflow Implementation

This workflow mirrors the "submit → review → approve/reject" loop described in the `Goal.pdf` and demonstrated in the review workflow video[1] [6] [4].

1. **Task Assignment**: A student enrolls, and the **Task Service** assigns them a starter exercise by creating a `Task` record with `status: TODO`.
2. **Student Submission**: The student completes the work in their repository and calls `POST /tasks/:id/submit`.

- The **Task Service** updates the task status to `SUBMITTED`.
- It then publishes a `task.submitted` event to the message broker[2].

3. **Automated CI Check**:
   - The **CI Orchestrator** receives the `task.submitted` event.
   - It runs the pre-configured tests (e.g., lint, unit tests) against the student's `repoUrl`.
   - If tests fail, it publishes a `ci.result.failed` event. The **Task Service** reverts the status, and the **Notification Service** informs the student.
   - If tests pass, it publishes a `ci.result.passed` event.
4. **Human Review**:
   - The **Review Service** receives the `ci.result.passed` event and makes the task available in the review queue.
   - An `Academy Reviewer` fetches their queue via `GET /reviews/queue`[4].
   - The reviewer examines the code and submits their verdict using `POST /reviews`, providing an `outcome` and `comment`.
5. **Final Outcome**:
   - The **Review Service** publishes a `review.created` event.
   - The **Task Service** consumes this event. If the outcome is `APPROVED`, it updates the task status and increments the student's `xp` points. If `REJECTED`, it updates the status and notifies the student to make changes.
   - The **Notification Service** informs the student of the outcome.

Throughout this process, a `Super Admin` can intervene at any step, using the admin-specific API endpoints to manage users, re-assign tasks, or override review outcomes, ensuring full platform control[1] [2] [5].

⁂

1. Goal.pdf
2. https://www.youtube.com/watch?v=HiwOx-W1TIA
3. https://www.udemy.com/course/design-microservices-architecture-with-patterns-principles/
4. https://www.youtube.com/watch?v=L25xafJgD1E
5. https://kalacademy.com/microservices-using-c/
6. tools.build_systems