Competa Arena: A Microservices Architecture Blueprint

Prepared by: Jean d'Amour **Date:** 2025-09-22 **Project Goal:** To build a scalable, high-performance, and real-time competitive learning platform for the Rwandan community and beyond, focusing on Physics, Mathematics, and Computer Science.

1. Introduction: Why Microservices?

This blueprint outlines a microservices-based architecture for Competa Arena. Instead of building one large application (a monolith), we will build it as a collection of smaller, independent services.

This approach was chosen specifically to meet your goals:

- **Scalability:** We can scale individual parts of the application. If a contest becomes very popular, we only need to provide more resources to the `Contest` and `Evaluation` services, not the entire platform.
- **High Performance:** Each service is small and optimized for its specific task.
- **Technology Flexibility:** As you noted, you need Python for future AI features. This architecture allows the `AI Service` to be built in Python, while other services could, in theory, use different technologies if needed.
- **Resilience:** If one service fails (e.g., the `Notifications Service`), the rest of the platform remains online. Users can still compete even if their real-time notifications are temporarily down.

2. The Core Services Blueprint

Here is the breakdown of the essential services needed for Competa Arena.

Service Name	Key Responsibilities	Recommended Python Framework	Rationale & Database Choice
1. User Service	• User registration & login (Authentication). • Profile management (name, school, etc.). • Access control (student vs. admin).	**FastAPI**	**FastAPI** is chosen for its incredible speed and automatic API documentation, which is perfect for a service that will be called by many others. **Database: PostgreSQL**. A robust, relational database is ideal for storing user data, credentials, and relationships.

Service Name	Key Responsibilities	Recommended Python Framework	Rationale & Database Choice
2. Contest Service	• CRUD (Create, Read, Update, Delete) for contests. • Manages questions, subjects, schedules, and rules (timed/untimed).	**FastAPI** or **Django REST Framework**	**FastAPI** for speed. Alternatively, **Django** could be useful here if you have complex admin requirements for creating contests, as it comes with a built-in admin panel. **Database: PostgreSQL** or **MongoDB**. PostgreSQL is great for structured contest data. MongoDB could be an option if your question formats are highly variable. Start with PostgreSQL for simplicity.

Service Name	Key Responsibilities	Recommended Python Framework	Rationale & Database Choice
3. Submission Service	• Accepts user answers (for quizzes) and code (for programming). • Validates the submission format. • **Crucially, it does not evaluate.** It only places the submission into a queue for processing.	**FastAPI**	This service must be extremely fast and lightweight. Its only job is to accept a request, validate it, put it on a queue, and immediately return a "Success" message to the user. FastAPI is perfect for this. **Queue: RabbitMQ** or **Redis**. RabbitMQ is more robust for mission-critical tasks. **Redis** is simpler to set up and is a great starting point.

Service Name	Key Responsibilities	Recommended Python Framework	Rationale & Database Choice
4. Evaluation Service (The "Judge")	• The "brain" of the platform. • Pulls submissions from the queue. • Compiles/runs code in a secure sandbox. • Compares answers against correct solutions. • Assigns a score and a status (e.g., Accepted, Wrong Answer).	**Python (Pure) with Celery**	This service doesn't need a web framework. It's a background worker. **Celery** is the industry-standard Python library for running asynchronous tasks and will work perfectly with RabbitMQ or Redis to process the submission queue. **Sandbox Tech: Docker**. This is non-negotiable for security. Each submission runs in its own isolated container.
5. Leaderboard Service	• Calculates and stores user rankings for each contest. • Provides data for real-time leaderboard updates.	**FastAPI**	Needs to be extremely fast to handle frequent updates and reads. **Database: Redis**. This is the key to a real-time leaderboard. Redis is an in-memory database, making sorting and retrieving ranked sets (like a top 100 list) almost instantaneous.

Service Name	Key Responsibilities	Recommended Python Framework	Rationale & Database Choice
6. Notifications Service	• Sends real-time updates to the user's browser. • Manages other notifications (email, etc.). • Examples: "Your contest is starting," "Your submission has been graded."	**FastAPI with WebSockets**	**WebSockets** are essential for pushing real-time updates from the server to the client without the client having to ask. FastAPI has excellent support for them.

Supporting Infrastructure

- **API Gateway:** This is the single "front door" for your React frontend. All requests go here, and the gateway routes them to the correct internal service. It simplifies your frontend code and adds a layer of security. (e.g., **Kong**, **Tyk**, or even **Nginx**).
- **Message Queue:** The asynchronous backbone of the application. **RabbitMQ** is the professional choice; **Redis** is a great, simpler alternative to start with.

3. The Development Roadmap: A Step-by-Step Guide

Do not build everything at once. Follow this phased approach to stay organized and ensure each part is working perfectly before moving on.

Phase 1: The Foundation (Core Functionality)

Goal: Allow a user to log in and see a list of contests.

1. **Build the `User Service`:**
   - Set up a FastAPI project.
   - Implement `/register`, `/login` (issuing a JWT token), and `/me` (get profile) endpoints.
   - Connect it to a PostgreSQL database.
   - **Containerize it with Docker.** Create a `Dockerfile` for this service.
2. **Build the `Contest Service`:**
   - Set up a second FastAPI project.
   - Implement endpoints to create and list contests (`/contests`).

- For the "create" endpoint, it should require a valid JWT token from the `User Service` to ensure the user is an admin. This will be your first **service-to-service communication**.
- Containerize this service with Docker as well.

3. **Use `docker-compose`:** Create a `docker-compose.yml` file to run both services and your PostgreSQL database with a single command (`docker-compose up`).

Phase 2: The Core Logic (The Competitive Programming Engine)

Goal: Allow a user to submit a solution and have it judged automatically.

1. **Set up the `Submission Service`:**
   - This FastAPI service will have one primary endpoint: `/submit`.
   - It will take the user's code, language, and problem ID.
   - Set up Redis or RabbitMQ and have this service push the submission details into a queue.
2. **Build the `Evaluation Service` (The Judge):**
   - This is a Python project, but not a web server. It's a worker script.
   - Use the **Celery** library to connect to the queue and listen for new submissions.
   - **Create your Judge Docker Images:**
     - `cpp-judge`: A Dockerfile based on Linux that installs `g++`.
     - `python-judge`: A Dockerfile based on Linux that installs `python3`.
   - When the service gets a submission, it will use the Python `docker` library to:
     - Start a new container from the correct judge image (e.g., `cpp-judge`).
     - Set strict memory and CPU limits.
     - Copy the user's code and test case input into the container.
     - Execute a script inside the container that compiles/runs the code, checks the time limit, and compares the output.
     - Capture the result (e.g., "Accepted," "Time Limit Exceeded").
     - Destroy the container.
   - After judging, this service will publish the result to a *new* queue (e.g., `results_queue`).

Phase 3: The User Experience (Real-Time Feedback)

Goal: Show the user their score and update leaderboards in real-time.

1. **Build the `Leaderboard Service`:**
   - This FastAPI service will listen to the `results_queue`.
   - When a new result comes in, it will update the scores stored in its **Redis database**.

- It will have an endpoint like `/leaderboard/{contest_id}` that the frontend can call to get the current rankings.

2. **Build the `Notifications Service`:**
   - This FastAPI service will also listen to the `results_queue`.
   - It will manage **WebSocket connections** to users.
   - When a user's result comes in, it will find their WebSocket connection and send a message like `{"message": "Your submission for Problem X was Accepted!"}`.

Phase 4: Future-Proofing for Payments

Goal: Design the system so that adding payments later is easy.

1. **Plan for a `Billing Service`:** When you are ready to add paid features, you will create a new, separate microservice.
2. **Responsibilities:** This service will handle:
   - Creating subscription plans.
   - Integrating with a payment provider (like Stripe, or a local Rwandan provider like Flutterwave).
   - Checking if a user has an active subscription.
3. **Integration:** Other services will call the `Billing Service` to check for permissions. For example, the `Contest Service`, before showing a premium contest, would ask the `Billing Service`: "Does this user have an active subscription?". This keeps payment logic completely separate from your core contest logic.

4. Final Advice

- **Version Control Everything:** Use Git and GitHub from day one for each service. It's good practice to have a separate repository for each microservice.
- **Automate with GitHub Actions:** Set up simple GitHub Actions to automatically run tests and build your Docker images whenever you push new code. This will save you an incredible amount of time.
- **Focus on the Flow:** The most important concept in this architecture is the flow of data: from the user's browser, through the API Gateway, to a service, onto a queue, processed by a worker, and finally pushed back to the user in real-time.

This blueprint provides a robust, professional-grade foundation for Competa Arena. By following this phased approach, you can manage the complexity and build a powerful platform that will serve the community well. Good luck!