CS302 Final Report (Dancesport System)

Lucas Liao
weishunliao.2022@scis.smu.edu.sg

Chloe Ong
chloeong.2022@scis.smu.edu.sg

Haris Shariff
harisms.2022@scis.smu.edu.sg

Axel Tan
axel.tan.2022@scis.smu.edu.sg

Lim Wei Jie
weijielim.2022@scis.smu.edu.sg

Oh Sheow Woon
swoh.2022@scis.smu.edu.sg

Abstract—This report describes the development of a digital scoring system for DanceSport competitions that replaces traditional paper-based methods with a real-time microservices platform. The system manages authentication, competition setup, scoring, event routing, and live display, with users interacting through standard web interfaces. Internal communication relies on asynchronous messaging to ensure accuracy and responsiveness. The project also applies DevSecOps practices, Infrastructure as Code, and automated continuous delivery to maintain quality and consistency. The solution improves efficiency, reduces errors, and enhances the overall competition experience.

Index Terms—dancesport, scoring system, microservices, real time, DevOps, cloud deployment

I. Introduction

DanceSport competitions traditionally rely on manual, paper-based scoring methods. These approaches often lead to administrative delays, human errors, and inefficient data processing. As competitions scale in size and frequency, the need for a modernised scoring system becomes increasingly important [1].

This proposal outlines a digital scoring solution designed to modernise adjudication, improve accuracy, and enhance the overall competition experience.

II. Key scenarios

This section presents the core usage scenarios that illustrate how the proposed digital scoring system supports the end-to-end competition workflow, from event setup to live adjudication and result publication. The scenarios highlight the interactions between adjudicators, competitors, and backend services, and demonstrate how the system reduces manual effort while improving transparency and responsiveness. The sequence diagrams illustrating the workflows for all scenarios described below are appended in Appendix B.

A. Scenario 1

Before the competition begins, the Chief Adjudicator sets up the event using the Competition Service. The admin creates competition categories and dances before generating the registration links.

Competitors and adjudicators submit their details through these links, and the information is stored in the Postgres database. Once the registration closes, the system automatically provisions the rounds and heats, assigns bib numbers to competitors, and allocates them to their respective heats.

B. Scenario 2

During a live heat, each adjudicator uses a tablet to enter their rankings or scores. Once submitted, these scores are forwarded to the Scoring Service, which aggregates inputs from all adjudicators.

The Scoring Service applies the official DanceSport tie-breaking rules to compute the final rankings. Results are updated immediately, ensuring that the latest standings are available in real time.

C. Scenario 3

As the round progresses, the audience can see the leaderboard update in real time as the adjudicators submit their scores. Once all adjudicators have finished scoring, the system computes the final rankings.

The top-ranked couples automatically advance to the next round, and the Competition Service records the official results and prepares the subsequent heats.

III. System architecture

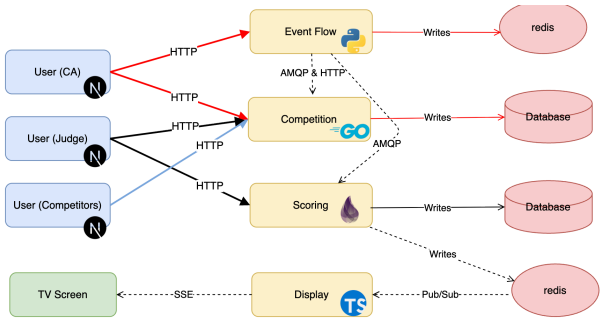

Fig. 1. System Architecture

The system architecture (Fig. 1) is built around five core microservices: Competition, Scoring, Display, Event Flow, and Auth. Each service is responsible for a specific functional area.

Users interact with the platform through standard HTTP requests, communicating directly with the relevant

microservice for actions such as logging in, registering competitors, configuring heats, or submitting scores.

Internally, microservices communicate using a combination of AMQP messaging and HTTP, depending on the type of interaction required. AMQP, implemented through RabbitMQ, is used for publishing and consuming events such as score submissions, heat updates, and competition changes. This allows the services to remain loosely coupled and enables reliable asynchronous communication. The Event Flow service functions as an event router and ensures that key competition-related events are propagated to the appropriate services.

To support audience-facing presentation, the Display service delivers real-time leaderboard data and heat status updates to public screens and dashboards using Server-Sent Events (SSE), enabling continuous, low-latency updates. Overall, this architecture supports scalability, fault isolation, and real-time responsiveness for live competition environments.

IV. Microservices Architecture

Building on the high-level architecture, this section provides a detailed description of each microservice and its role within the competition workflow.

A. Services Overview

The architecture separates concerns into specialized services:

Competition Service (Golang): Serves as the core domain service managing competition lifecycle, discipline definitions, registration workflows, and heat generation. It maintains authoritative state in PostgreSQL and publishes domain events via RabbitMQ when significant state changes occur (e.g., sub-events created).

Scoring Service (Elixir/Phoenix, Lua): Provides HTTP APIs for adjudicators to submit scores during the event. It is built with Elixir for fault tolerance through supervision trees and process-based memory isolation. Upon retrieval of scores, they are pushed to Redis via pre-evaluated Lua scripts, calculating rankings for all the competitors, before publishing to the Redis pub/sub channels for downstream consumers.

Event Flow Service (Python/FastAPI): Orchestrates the progression of competition events through their lifecycle (started → ongoing → stopped). It subscribes to RabbitMQ for sub-event creation notifications, fetches hierarchical event structures from the Competition service, flattens them into linear sequences, and stores event graphs in Redis for efficient retrieval. It also exposes HTTP APIs for the frontend to get event information (e.g., the current running event, next event).

Display Service (Node.js/Express): Provides real-time leaderboard visualisation for audience screens. It subscribes to Redis pub/sub channels (‘display::event_flow‘ and score update channels) and streams live updates to frontends via Server-Sent Events (SSE). The service implements redundant polling as a fallback mechanism to ensure data consistency even if pub/sub messages are missed or on service restarts, for increased consistency.

Frontend Service (Next.js 15/TypeScript): Delivers web interfaces for different user roles including Chief Adjudicators (competition management), Adjudicators (score submission), and Competitors (registration). It is built with Next.js for server-side rendering and optimized client-side navigation and styled with Tailwind CSS v4.

Auth Service (Golang): Handles authentication and authorisation using JWT tokens with RS256 signing. Integrates with Twilio Verify API for SMS-based one-time password (OTP) verification, where users receive a 6-digit code via SMS to their mobile number. The service validates OTPs through Twilio’s verification check endpoint, which returns approval status. Upon successful OTP verification, the service issues signed JWT tokens containing user claims for subsequent API authorisation, providing secure access control for sensitive operations like competition provisioning and score submission.

B. Communication Patterns

The system employs both synchronous and asynchronous communication patterns optimised for different use cases:

Synchronous REST APIs: Services expose RESTful HTTP endpoints for request-response interactions. The Competition service provides endpoints for competition CRUD operations, registration management, and heat retrieval. The Scoring service exposes APIs for score submission and ranking queries. All endpoints accept and return ‘application/json‘ with appropriate HTTP status codes for error handling.

Asynchronous Messaging (RabbitMQ): Domain events are published via RabbitMQ using fanout exchanges for pub/sub patterns. When the Competition service creates sub-events, it publishes ‘SubEventCreatedEvent‘ messages to the ‘sub_event_created‘ exchange. The Event Flow service subscribes to this exchange, receiving notifications to trigger event graph generation without tight coupling between services.

The Event Flow service acts as the primary event coordinator, publishing multiple event types to downstream consumers. When an event graph is successfully generated, it publishes ‘event_graph_ready‘ messages containing the competition ID and head event ID to notify the Competition service. As competitions progress through their lifecycle, the Event Flow service publishes ‘event_started‘ events when the first heat begins, enabling the Competition service to update its state machine accordingly.

Redis Pub/Sub: Real-time score updates flow through Redis channels. The Scoring service publishes score changes to the ‘display::event_flow‘ channel, which the Display service subscribes to for immediate propagation

to connected frontends. This pattern provides low-latency updates suitable for live audience displays.

Server-Sent Events (SSE): The Display service maintains persistent HTTP connections with frontend clients using SSE, enabling unidirectional streaming of score updates without WebSocket complexity. Clients automatically reconnect on connection loss, and the service handles multiple concurrent streams efficiently.

V. Data Management

Each service maintains its own data stores following the database-per-service pattern:

PostgreSQL Databases: The Competition service uses PostgreSQL with schema migrations managed via golang-migrate, storing competitions, disciplines, registrations, and heats in normalised tables with foreign key constraints. The Scoring service similarly uses PostgreSQL for score persistence and ranking history, with its own schema migration system. Database connection pools are configured with health checks and connection lifetime limits for reliability.

Redis Cache: Used for high-velocity data including event graphs (Event Flow), live scores (Scoring), and session state. Redis provides sub-millisecond read latency critical for real-time display updates. Data is typically ephemeral with TTLs set based on competition duration.

Schema Versioning: Database migrations are version-controlled and applied automatically during deployment. The Competition service includes migration scripts in its container image, executed by a separate 'competition_migrate' container before the main service starts, ensuring zero-downtime deployments. Similarly, the Scoring service runs its migrations on server start.

VI. Devops Practices

A. CI/CD Pipeline Architecture

The project follows a trunk-based development within a monorepo, using GitLab's parent-child pipeline configuration to orchestrate service-specific CI/CD workflows. The parent pipeline (.gitlab-ci.yml) triggers child pipelines for the Competition, Display, Event Flow, Frontend, Scoring, and Auth services, enabling independent builds and deployments while maintaining consistent standards across all microservices.

These practices form the basis of the system's automated CI/CD workflow. To provide clarity and consistency across all microservices, the pipeline is structured into well-defined stages, which are outlined below.

1) CI Pipeline Stages

The CI pipeline is triggered on merge requests to the 'main' branch. Each microservice follows a standardised CI pipeline.

Prepare: Downloads dependencies and caches modules for faster subsequent builds.

Lint: Enforces code formatting standards using language-specific tools (e.g. gofmt for Go, ESLint for TypeScript).

Test: Executes unit tests with coverage reporting (e.g., using gotestsum for Go services), generating JUnit XML reports for GitLab integration.

Integration: Runs integration tests using Docker-in-Docker with Testcontainers, enabling realistic testing against databases and message queues.

Security: Performs Static Application Security Testing (SAST) using Semgrep and secret detection to prevent credential leaks. GoSec provides Go-specific vulnerability scanning at severity level 0 for comprehensive coverage.

2) CD Pipeline Stages

The Continuous Delivery (CD) pipeline is triggered on merge commits to main. The stages in the CD pipeline are:

Package: Builds Docker images tagged with Git commit SHA and pushes to Azure Container Registry (ACR). Image digests are extracted and stored as artifacts for immutable deployments.

SBOM: Generates Software Bill of Materials using Syft to track dependencies and improve supply-chain security transparency.

Sign: Signs and verifies container images using Cosign to verify authenticity and integrity before deployment.

Deploy: Updates Kubernetes manifests in the separate deployment repository through automated merge requests.

B. GitOps Deployment

1) Deployment Automation

The CD pipeline implements a GitOps workflow by automatically updating Kubernetes manifests in a separate deployment repository ('dancesport-deployment'). This separation ensures declarative configuration management and maintains Git as the single source of truth. The deployment update process follows these steps:

a) Clone the deployment repository using authenticated credentials
b) Create a feature branch named 'ci/service/commit-sha'
c) Update the container image digest in the Kubernetes manifest using 'yq'
d) Commit changes with descriptive message
e) Push branch and create merge request via GitLab API
f) Enable auto-merge after pipeline checks pass

The script uses image digests rather than tags to ensure immutable deployments, preventing drift caused by tag mutations.

2) Merge Request Automation

The pipeline checks for existing merge requests before creating new ones, ensuring idempotency. Auto-merge is enabled with retry logic (up to 6 attempts with 5-second intervals) to handle timing issues when merge requests are

not immediately ready. Source branches are automatically deleted after merge to maintain repository cleanliness.

C. Agile Project Management

Agile methodologies were employed to facilitate the incremental delivery of features and improvements across our microservices architecture. Development was organised around iterative workflows supported by GitLab's merge request processes, enabling close collaboration among team members working on various components.

Regular communication through team meetings and code reviews maintained alignment and transparency throughout the project lifecycle. These practices encouraged rapid adaptation to evolving requirements and the continuous integration of feedback from integration testing and cross-team dependencies.

All code changes required approval from teammates before merging to the main branch, ensuring collective code ownership and quality standards. Merge request conventions enforced structured commit messages following the Angular convention (e.g., 'feat:', 'fix:', 'docs:'), providing clear change histories and facilitating automated release notes.

This systematic approach to project management ensured that development efforts remained aligned with project goals while maintaining code quality through peer review and automated CI/CD pipelines that validated changes before merging.

D. API Documentation

As part of our DevOps practices, we prioritised creating and maintaining comprehensive, up-to-date API documentation to facilitate seamless collaboration between teams and ensure efficient microservice integration. We adopted a contract-first approach where each team published their service contracts before implementation, enabling teams to work independently and in parallel.

For our Go-based services (Authentication and Competition), we leveraged Swagger annotations directly in the source code to automatically generate interactive API documentation accessible at '/swagger/' endpoints, ensuring documentation remained synchronised with implementation.

For other services, we maintained detailed 'endpoints.md' files as authoritative contracts, with the EventFlow team publishing their specifications on the project wiki for enhanced visibility. These contracts included complete endpoint specifications, request/response schemas with example JSON payloads, authentication methods, error codes, and integration examples.

By exchanging service contracts early in development, teams could mock dependencies and validate integration points before writing code, significantly improving collaboration and reducing integration issues in our distributed microservices architecture.

E. Post-Mortem

Following integration issues and system incidents, we conducted analysis sessions to identify root causes, document resolution steps, and capture lessons learned for future prevention. These sessions emphasised learning and improvement over assigning blame, fostering a culture of transparency and collaborative problem-solving.

When critical RedisJSON compatibility challenges arose during integration testing between the Display and Scoring services, we collaboratively diagnosed the issue through team meetings and discovered that our Redis Cluster deployment did not support RedisJSON commands. Through collaborative problem-solving, we decided to deploy a separate standalone Redis instance and modify the application code to parse JSON output directly rather than relying on RedisJSON-specific commands, ensuring compatibility across different Redis deployments while maintaining system functionality.

This approach aligns with the "Third Way" of DevOps, which focuses on continuous learning and experimentation. Continuous feedback loops through integration testing, cross-service validation, and team communication ensured members had the necessary information to address issues proactively, ultimately leading to a more resilient and maintainable distributed system.

VII. Self-Directed Research

This project required the incorporation of self-directed research which led us to explore DevSecOps, Infrastructure-as-Code, and GitOps. These approaches were selected because they enabled rapid iteration while maintaining security, scalability, and reliablity. By adopting automated security checks, reproducible infrastructure, and declarative deployments with easy rollbacks, the team was able to move quickly without compromising system integrity.

A. DevSecOps

In this project, several DevSecOps practices were applied by integrating static code analysis, security scanning, secret detection, and automated within the workflow. For linting and static code analysis, tools such as Ruff for Python and Go Vet for Go were used to enforce coding standards, detect style issues, and highlight potential logic mistakes early. On security, SAST tools like Semgrep, which helped identify vulnerabilities such as a SQL injection risk and weak cryptography usage were applied.

These issues were fixed, and other alerts were validated as false positives. Secret detection was also enabled to ensure that sensitive credentials such as API keys or tokens were never accidentally committed.

Alongside these checks, unit and integration tests were developed for each microservice to verify correctness and stability. All these tools and scans run automatically during merge requests, ensuring that only secure, high-quality, and tested code is merged into the main branch.

B. Infrastructure-as-Code with Terraform

Terraform was adopted to manage our cloud resources through Infrastructure as Code. It allowed for the creation of consistent development, staging, and production environments and reduced manual configuration errors. Terraform made it easy to replicate environments and apply changes quickly; for example, the staging environment was replicated and production-level components such as a private networks, SSL, and Application Gateway were added. This research demonstrated how IaC improves consistency, efficiency, and maintainability.

C. GitOps with ArgoCD

For continuous delivery, Argo CD was used to automate and visualise our Kubernetes deployments. It continuously monitored the live cluster and ensured it always matched the desired state defined in our Git repository. This eliminated many repetitive manual tasks, such as manually applying updated manifests, and reduced operational overhead. Argo CD also provided a clear interface for viewing the health and sync status of each service, giving fast feedback whenever something drifted from the intended configuration.

By separating CI and CD, the workflow became more organised, with CI handling builds and tests while Argo CD managed declarative deployments. Rollbacks also became much simpler, since reverting to a previous Git commit automatically restored the earlier configuration in the cluster. Overall, automatic reconciliation, easy rollbacks, and continuous monitoring helped improve the efficiency, reliability, and manageability of our deployments.

VIII. Reflections

A. What went well

Our team worked extremely well together throughout the project. We have built a strong culture of collaboration over many past projects, which allowed us to communicate openly and effectively. We frequently jumped onto calls to discuss ideas, solve issues immediately, and support one another whenever someone faced difficulties. Collaborative debugging became a natural part of our workflow, and our familiarity with each other's strengths and weaknesses helped us divide work efficiently and maintain steady progress. This strong teamwork made the entire development process smooth and enjoyable.

We are also proud of how quickly and confidently we picked up new technologies. This project pushed us to explore tools such as Terraform, Argo CD, GitLab CI CD, automated testing, SBOM generation, image signing, and SAST scanning. These tools were either new to us or only lightly touched on in class, but through research and experimentation we managed to apply them effectively. We also managed to implement observability and performance monitoring using metrics and dashboards via Azure Log Analytic Workspace (Kusto Query Language) which helped us detect deployment issues and track resource usage across our infrastructure. Setting up a complete CI CD pipeline, implementing Infrastructure as Code, and integrating security checks allowed us to deepen our understanding of DevOps philosophy. Seeing everything come together in a fully automated workflow was both rewarding and educational, and it helped us appreciate how modern software delivery is done in real industry settings.

We also enjoyed the opportunity to embrace DevOps culture in a more holistic way. Unlike previous projects, where the focus was mainly on building features, this project allowed us to think about reliability, automation, consistency, and operational efficiency. By combining the tools, were able to iterate quickly and safely without fear of breaking the system, since every change was tested, reviewed, traced, and reversible. Monitoring deployments through Argo CD, managing resources through Terraform, and enforcing quality checks through CI all gave us hands-on experience with practices that professional teams rely on. This made the project feel richer and more meaningful compared to traditional feature-only assignments.

Lastly, we felt motivated by the fact that our project addressed a real-world problem. Ballroom dance competitions still rely heavily on manual processes that are slow and error prone and building a system that could genuinely improve the experience for organisers, judges, and audiences made the work feel purposeful. Designing a real time scoring system with live leaderboards and event driven communication showed us how software can directly solve practical issues. Knowing that our solution could be used in an actual competition setting made the project especially fulfilling and gave us a sense of pride in what we built.

B. What could be better

Although our project went well overall, there is always room for improvement. One key area is communication, especially when project requirements begin to evolve. Despite having API contracts established early on, a few unexpected edge cases emerged as the scope expanded. This led to minor integration issues, particularly within our choreography-style architecture. Improving the clarity and consistency of communication during requirement changes would help prevent these misalignments in future projects.

We were also unable to fully automate end-to-end testing due to the timing and sequencing of service integrations. Although we explored Playwright and managed to script parts of the flow, the overall system wasn't in a stable enough state for a complete, reliable e2e pipeline. Several service dependencies were still evolving or not yet aligned, which meant the test paths frequently broke or couldn't reach a full user journey. As a result, our e2e efforts ended up being partial and more manual than intended, limiting our ability to validate the full system early.

C. Future improvements and next steps

In the future, we plan to refine our microservices to improve reliability, error handling, and overall system performance. We already introduced contract testing, mocking, and integration tests to support service isolation and incremental development. A natural next step is to expand these practices into a more systematic end-to-end testing framework, allowing broader scenario coverage and earlier validation of workflows even while services continue to evolve.

The user interfaces for judges and displays can be made more accessible by improving layout clarity, increasing contrast, and simplifying interactions to better support quick decision-making during live events. On the DevOps side, we hope to further improve our Terraform and Argo CD setups to make them more modular and production ready by applying automated validation and introducing a structured pipeline for plan/apply and state management.

Additionally, implementing Kubeseal for production-grade secrets management would address address key limitations of environment variables and GitLab CI/CD variables, which cannot safely be safely stored in Git and are exposed in plaintext during CI execution. Kubeseal uses asymmetric encryption to create SealedSecrets that can be safely stored in the GitOps repository, which only the Kubernetes cluster's private key can decrypt. This would strengthen our existing DevSecOps pipeline and align with GitOps best practices for declarative infrastructure management.

Finally, a key next step is to test the system with actual users from the DanceSport community and gather feedback to prepare the platform for real-world use.

References

[1] N. Khare, "Why Every Dance Competition Needs Smart Scoring Software," Danceplace Business Blog, Jul. 20, 2025. [Online]. Available: https://www.danceplace.com/business/blog/upgrade-dance-competition-scoring-software/ [Accessed: Nov. 16, 2025].

Appendix A

TABLE I
Member Contributions

Name	Contributions
Lucas Liao	I was the primary owner of the Authentication service, where I designed and implemented the core authentication and authorization workflows for the system. I also provisioned the production environment by extending the staging setup with API gateways, private networking, domain configuration, and SSL termination. In addition, I integrated Twilio for one-time password delivery and JWT issuance and connected the authentication flow to the frontend to enable secure, user-specific access across the platform.
Chloe Ong	My contributions focused on the Display service, DevSecOps implementation, and system orchestration. I developed the Display service using TypeScript with Redis pub/sub and polling for real-time score delivery via Server-Sent Events, along with its frontend visualization interface. For DevSecOps, I integrated GitLab's SAST and Secret Detection across all microservices with configured exclusion patterns and mandatory CI/CD security scanning. I also help maintain the root 'docker-compose.yml' orchestrating all six microservices and dependencies, enabling consistent development environments and integration testing. Furthermore, I have contributed to the Competition Service by implementing a couple the APIs during a critical phase of development, helping ensure feature completeness while other core tasks were in progress.
Haris Shariff	I served as the primary owner of the Competition Service, where I designed the architecture, built most of the API surface, and coordinated integration work across the services. In addition to service development, I led platform and pipeline engineering for the project by designing the root GitLab CI/CD parent-child pipeline structure, integrating SBOM generation and Cosign signing, and maintaining automated deployment workflows. I also provisioned and operated our staging environment on Azure (AKS, ACR, Argo CD, Postgres, Redis and LAW) and authored the Kubernetes manifests shared across services. These responsibilities spanned multiple layers of the system, and I work closely with teammates to ensure everything integrated smoothly.
Axel Tan	My primary contribution was the development of the Event Flow Service, implemented in Python using FastAPI. I also designed and managed the AMQP-based message bus architecture for the system, including the fanout exchanges that enable loose coupling between microservices. In addition, I authored the root 'docker-compose.yml' configuration to support local development and service integration, and designed several Competition Service APIs in Go to facilitate integration with the Event Flow Service.
Lim Wei Jie	I led the entire frontend and UX development of the application, building a clean, intuitive interface and ensuring consistent user flows. I created the frontend Dockerfile and CI steps for linting, formatting, and automated tests, and designed both unit and component test suites to maintain code quality. I also integrated all backend services locally and performed manual end-to-end testing to validate system behaviour. Throughout the process, I worked closely with each service owner to resolve integration issues and ensure all components functioned seamlessly together.
Oh Sheow Woon	I mainly worked on the Scoring service, planning and building it to be fast, reliable and fault tolerant. This included the writing and debugging of the Redis Lua scripts which enabled atomic and fast scoring calculations. Beyond that, I also designed the pub/sub system between Scoring and Display service and gave insights on how to improve fault tolerance on service restarts, ensuring graceful recovery.

Appendix B

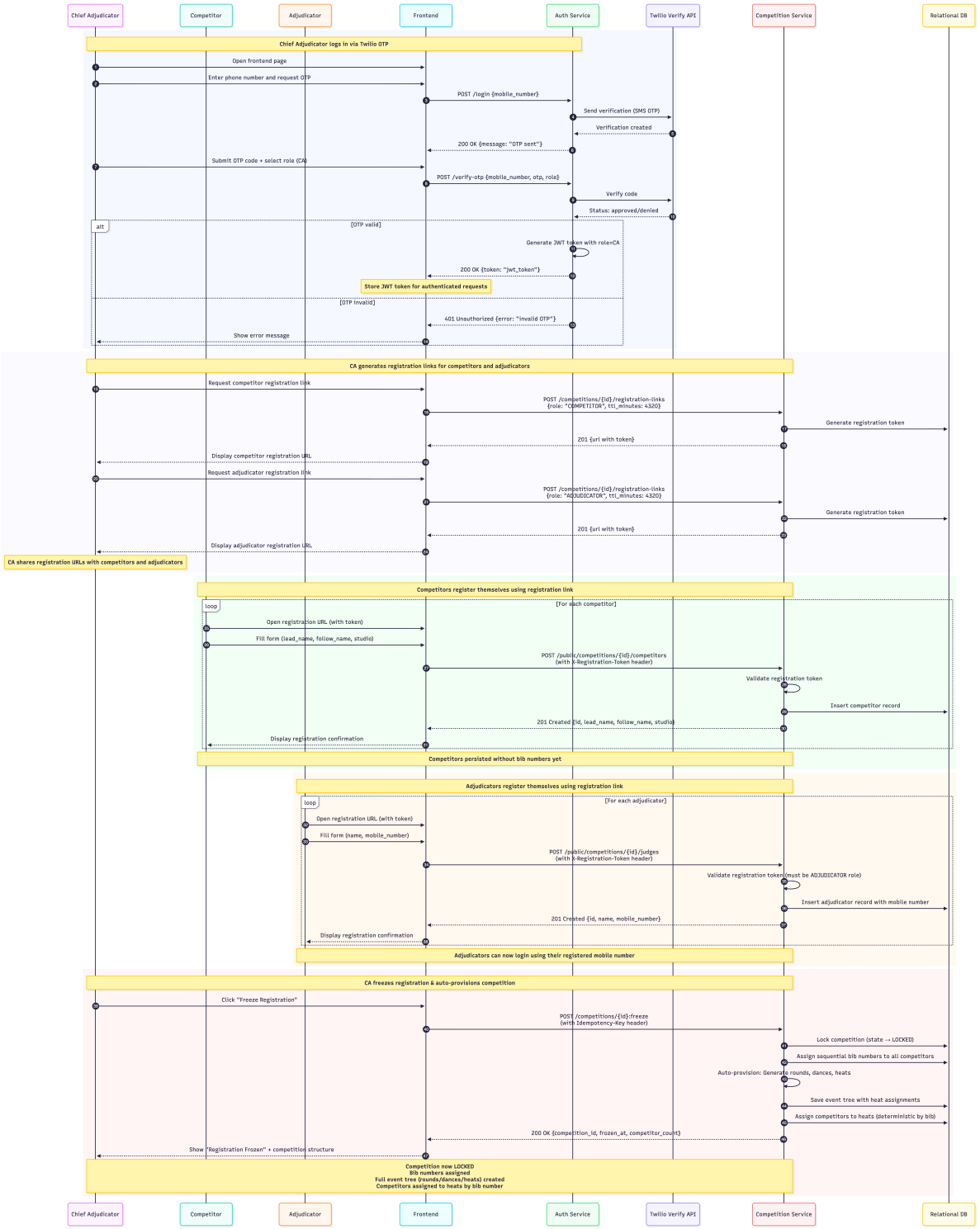

Fig. 2. Sequence diagram for Scenario 1:

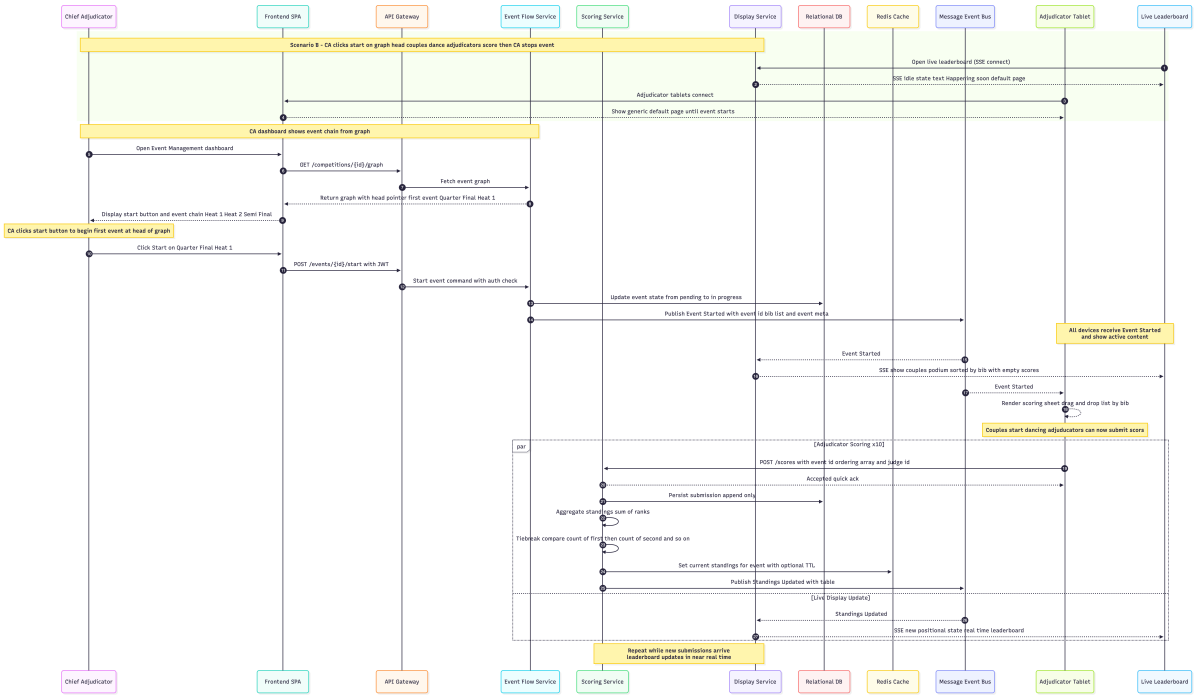

Fig. 3. Sequence diagram for Scenario 2:

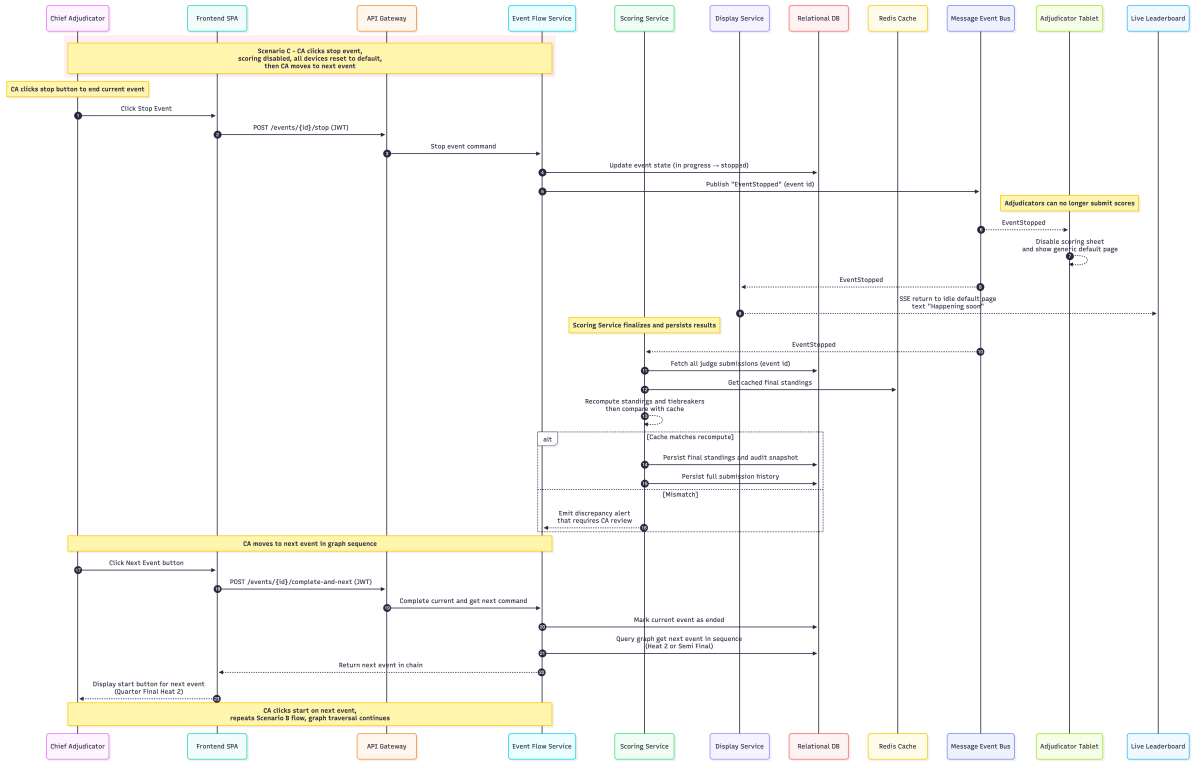

Fig. 4. Sequence diagram for Scenario 3: