Esports Organizer

Second Milestone

Prof. Marko Schütz

Introduction to Software Engineering (INSO4101)

Section 080

Professor Marko Schütz

Friday, October 24, 2024

Table of Contents

Legend

Added data

Removed data

Paraphrased data

0 Introductory Part

0.1 Purpose

Esports Organizer is an online platform created to make it easier for players, teams, and communities to connect and manage their activities in the world of competitive gaming. Its main goal is to bring everything related to esports organization into one place, so users can track their progress, show their achievements, and join events without confusion. The project focuses on accessibility and teamwork. With a clean and modern design, Esports Organizer allows users to interact in a way that feels smooth and enjoyable. It is more than just a management tool; it is a space that helps players grow, communicate, and build stronger gaming communities.

0.2 Skeleton

The Esports Organizer web app is built using React for the frontend and Firebase for the backend, making the interface interactive, clean, and easy to update. The structure of the app is organized into different parts that work together to give users a complete experience. Some of the main parts include the Teams section, which lets users discover and manage their team, and the Communities section, where users can connect and interact with others who share similar interests. All parts of the app are connected and designed to grow in the future, allowing new features and improvements to be added while keeping everything simple, responsive, and easy to use for everyone.

1 Informative Part

1.1 Team Members

Our project's team is composed entirely of students participating in the Esports Organizer project. Every team member is directly involved in the development, design, and decision-making processes. While the project does not have external clients or sponsors, it still involves several types of stakeholders. Stakeholders include players, organizers, spectators,community members / clubs, developers / team members themselves that have an interest by the system's outcomes. Recognizing this distinction helps the team consider diverse perspectives when making design and implementation decisions.

Roles and responsibilities are distributed among the team members as follows:

- Experience Design Team - Focuses on the user interface and overall user experience of the platform. This includes designing intuitive layouts, ensuring visual appeal, and optimizing how users interact with the system to create a seamless and pleasant experience.
- Identity and Data Systems Team - Manages the backend infrastructure, and the data handling. This team ensures that the system can securely store, process, and retrieve user and event information while maintaining performance and reliability.
- Events and Notifications - Handles functionality related to event management and notifications. This includes handling event creation, scheduling, and ensuring timely notifications and updates are delivered to participants and spectators.
- Player and Teams Profiles Team - Oversees features related to player and team identity within the platform. This includes profile creation, team registration, and ensuring accurate representation of players and teams across tournaments and events.

To ensure smooth progress across all teams, we are implementing strong communication channels between them following Agile Development within the project. The Experience Design, Identity and Data Systems, and Event Management teams will hold weekly alignment meetings to minimize dependencies and prevent delays in design or implementation. This collaborative workflow ensures that decisions about data structures, user flows, and system needs are made jointly, maintaining consistency across the project.

1.2 Current situations, needs, ideas

Current situation

Video games are a widely popular form of entertainment where people can either enjoy casual fun with friends or compete in more structured, competitive settings. However, the competitive gaming scene is broad and diverse, which makes it harder for players to navigate. Most of the information about esports events is scattered across multiple platforms and communities, and because genre preferences vary widely, a lot of events go unnoticed. This lack of variety prevents player from discovering tournaments, joining them, or finding a community they like, reducing opportunities for players to connect with others who share similar interests.

To better understand this situation, the team conducted domain engineering research, analyzing how grassroots and student esports tournaments are currently organized. Platforms such as Start.gg, Battlefy, and Tournament were studied to identify both strengths and gaps in existing systems.

Needs

From this situation a few needs were identified.

- Players need a **clear, reliable and centralized** platform where they can easily discover and register for esports events.

- Organizers need a reliable and effective channel to promote their events and reach their audience.
- Communities need more visibility for their events so they can attract new members and maintain engagement.
- Spectators and fans need an accessible way to follow competitions, track results, and feel part of the event experience.
- Developers need well-defined software requirements and architecture to ensure scalability, usability, and long-term maintainability.

Ideas

Our project aims to address these needs by implementing a web-base platform for esports tournaments. This includes both software engineering processes and user-centered design principles and the following feature:

- Conducting domain engineering and user-centered design to ensure the platform meets community needs.
- Community engagement activities, such as integrating feedback loops and social interaction features to help players connect with others and follow competitions.
- Implementing a web-based platform with UI/UX principles that ensure accessibility, adaptability, and usability.
- Centralized events hub where player can **browse and discover** tournaments by game, date, community.
- Registration and management system to simplify how participants sign up and how organizers handle teams and brackets.
- Notification and updates feature to keep participants and spectators informed in real time.
- Community-driven interface that allows players to connect with others, share interests, and follow ongoing competitions.
- Testing and validation activities to ensure performance, usability, and reliability, covering more than just code functionality.

These ideas were designed to reduce fragmentation, improve the visibility of events, and create a stronger and more connected esports community.

1.3 Scope, Span, and Synopsis

Scope

This project belongs to the broad domain of entertainment, gaming and event organization, with a focus on the growing field of esports. This includes players, organizers, or simply spectators who participate or follow gaming tournaments and events, whether it's online or in person. The scope covers all phases of the software engineering required for the project, including domain description,

requirements engineering, software architecture, component design, implementation, and finally testing. It also considers aspects of user experience, communication, and community engagement, as these are essential in the competitive gaming environment.

Additionally, it includes testing, validation, and community-focused UX activities beyond pure code development.

Span

While the domain of the project is the global esports community, the specific focus (span) of this web-based hub is on small to medium scale tournaments organized by local communities, including student groups and other independent gaming organizations. The web is designed to help these organizers manage registration, brackets, communication and visibility in a way that is easy to use. Unlike large-scale esport platforms, this project emphasizes inclusivity, adaptability, and simplicity making it suitable for semi-formal competitions.#

Synopsis

This project aims to design and develop a web-based platform that facilitates the organization and participation of esports tournaments. The web page will centralize key processes such as event discovery, player registration, tournament scheduling, and results publishing. It will also provide real-time notifications, and public results for spectators.

The project will be conducted through standard software engineering phases:

- Domain description to understand the current practices of student esport events.
- Requirements analysis to capture user and stakeholder needs.
- Software architecture and design to define the system's structure and components.
- Implementation of core features in line with the defined requirements.
- Testing and validation to ensure functionality, usability, and reliability.

In summary, this project delivers not only a working system but also serves as a solution to improve how other esports web pages organize tournaments and shape user experiences.

1.4 Other activities than just developing source code

Our project is not limited to just source code. Through research, each team worked to ensure the system is meaningful and effective for the gaming community, addressing problems seen in other platforms.

1.5 Derived Goals

In addition to its main goals of centralizing event discovery, simplifying registration, and strengthening

community engagement, the Esports Organizer project also seeks to achieve several secondary outcomes that emerge from the main objectives but go beyond the platform's basic purpose:

- Promote fair play and inclusivity in all competitive tournaments by keeping competition fair and tournament tools easy to use across different events and communities.
- Increase the visibility of local and student-led esports initiatives so they can be recognized and stay active even outside the platform.
- Encourage partnerships among various gaming communities to help them connect and work together beyond a single tournament or event.
- Support the learning process and skill development for players and teams by giving structure and feedback that help them improve over time.
- Raise consciousness about esports culture and its values as a positive social and educational influence beyond event hosting.

These derived goals differ from the main objectives: while the primary goals focus on making the platform functional and accessible, the derived ones lead to broader cultural, educational, and social benefits for the esports community.

These derived goals add value beyond the platform's main purpose, helping to strengthen communities, improve collaboration, and build long-term engagement in esports.

2 Descriptive Part

2.1 Domain Description

This section describes the essential components and interactions within an the esports tournament platform. It defines how players, teams, and organizers interact through the platform and identifies the terminology and conceptual abstractions that structure the system. The description includes a rough sketch of the domain, key terms, a narrative of typical processes, and breakdowns of events, actions, behaviors, and function signatures.

2.1.1 Rough Sketch

- Players register in a community and can either create new teams or join existing ones advertised within the platform.
- Each team designates a captain, who manages invitations, roster changes, and tournament registrations.
- Organizers publish tournament announcements with eligibility rules such as age limits, roster size, or regional restrictions.
- Teams submit their rosters before the registration deadline, depending on the ruleset, limited roster chances may be permitted later.

- Tournament seeding is determined by prior rankings, past results, or qualifiers.
- Supported formats include single-elimination, double-elimination, Swiss, or round-robin.
- Teams must check in before each match; missing a grace period may cause a forfeit.
- Results are recorded; tiebreakers decide standings in round-robins or Swiss.
- Spectators and media follow brackets, standings, and schedules.

Figure 2.1.1 – Tournament and Community Interaction Flow. This diagram illustrates the relationship between players, organizers, and communities. It shows how tournaments are advertised, how teams are formed and registered, and how interactions move from creation to competition within the ecosystem.

2.1.2 Terminology

- **Player**: An individual who registers on the platform to participate in esports tournaments.
- **Team**: A structured group of players with shared identity, team name, scorem and confirmation status (isConfirmed), led by a captain.
- **Captain**: A designated team member responsible for managing team activities, including roster changes and tournament registrations.
- **Tournament**: A competitive event where teams compete against each other in a structured format.
- **Match**: A single game or series of games played between two teams within a tournament
- **Organizer**: An individual or group responsible for setting up and managing tournaments.
- **Spectator**: An individual who watches the tournaments and follows the progress of teams and

matches.

- **Community**: A group of players and teams that share common interests and participate in tournaments together.

2.1.3 domain terminology in relation to domain rough sketch

The Rough Sketch describes the dynamic, real-world actions that occur within the platform — such as “team registration,” “match reporting,” or “roster updates.” These represent domain events and procedural details observed in operation.

```
*The Terminology section abstracts these raw observations into reusable, stable concepts.
For example:
*“15-minute grace period” → becomes Check-In Policy
**“Missed match due to delay” → becomes Forfeit Event
*“Map veto order” → becomes Match Preparation Protocol
*“Roster changes allowed until Day 1” → becomes Roster Update Policy
```

This abstraction process establishes consistency across the documentation, allowing the same concept (e.g., “check-in”) to be used precisely throughout requirements, design, and implementation. It also enables closure — where each defined concept connects seamlessly to a corresponding function or entity in the system model (e.g., a Team object includes isConfirmed to represent check-in status).

2.1.4 Narrative

The esports tournament process begins with the organizer, who creates and publishes the tournament on the platform. The organizer defines important parameters such as the tournament name, format, ruleset, and registration window. Once registration opens, teams—created by players within communities—can submit their rosters. Each team must designate a captain, who serves as the primary contact for communications, match coordination, and administrative updates.

During the registration phase, eligibility is verified (account status, ranking limits, and roster size). If the number of registered teams exceeds capacity, the extra teams are placed on a waitlist queue. When registration closes, the tournament bracket generation algorithm initializes:

```
*For single-elimination tournaments, the bracket is structured as a balanced binary tree,
ensuring fair seeding based on rank or random assignment.
*Teams may receive byes if the total participant count is not a power of two.
```

As the event progresses, matches are conducted round by round. Winners advance automatically to the next stage, while losers are eliminated. After every match, results are updated in real time through the system’s front end, using listeners or state updates that propagate score and bracket changes across

the platform. Once the final match concludes, the tournament state transitions to “closed,” and the results are recorded for ranking and statistical purposes.

This process represents the complete lifecycle of an esports competition, integrating the behaviors of players, teams, and organizers into a unified flow that the platform supports and tracks programmatically.

2.1.5 Events, Actions, Behaviors

Events, actions, and behaviors describe the system’s dynamic elements during tournament operations. Based on the tournament flow described in the reference document, these elements can be categorized as follows:

- Event (instantaneous state change)
  - "Team registration window has closed."
  - "Match #3 has been completed with a score of 2–1."
  - "Tournament bracket successfully generated."
  - "Player connection lost during match."
  - "Roster update deadline reached."
- Action (an act carried out once)
  - Create a tournament.
  - Register a team for a tournament.
  - Generate tournament bracket.
  - Assign a team captain.
  - Update match results.
  - Record a forfeit.
  - Send notification to players.
  - Perform map veto.
- Behavior (multi-step process composed of actions/events)
  - Run Tournament Flow: Create → Register → Generate Bracket → Conduct Matches → Update Results → Close Tournament.
  - Handle Registration: Check eligibility → Validate team → Add to confirmed or waitlist → Confirm participation.
  - Update Bracket Progression: Identify match → Update scores → Propagate winners to the next match → Refresh tournament state. **Manage Real-Time Updates: Listen for database changes → Trigger UI updates → Re-render bracket with new data.

These behavioral groupings form the operational backbone of the esports system. Each process is

modeled to support agile and event-driven design principles, ensuring the platform reacts to real-time changes (e.g., new registrations, match completions, or team updates) without manual intervention.

2.1.6 Function Signatures

Function signatures define how the system's operations are represented programmatically — describing inputs, outputs, and failure conditions. Derived from the registration, bracket generation, and update logic described in the reference file, the following function abstractions represent key behaviors in the tournament system:

*createTournament(organizer, name, format, maxSlots, rules) → Tournament | Failure Creates a new tournament entry with defined parameters. *registerTeam(team, tournament) → Confirmation | Waitlist | Denied Registers a team if it meets eligibility requirements and if slots are available. *generateBrackets(tournament) → BracketStructure | Failure Creates a bracket layout based on participant count and format (single/double elimination). *recordMatchResult(matchID, winnerID, score) → UpdatedBracket | Failure Updates match data and propagates the winning team to the next round. *updateTeamScore(teamID, points) → UpdatedTeam | Failure Adjusts a team's total score and updates leaderboard standings. *checkInTeam(teamID, tournamentID) → StatusUpdated | Timeout Verifies attendance before match start; returns timeout if grace period expires.

Each function maintains closure within the domain: the output (e.g., a BracketStructure or UpdatedTeam) can directly serve as input to another function in the system, supporting modularity and composability in software design.

2.2 Requirements

2.2.1 User Stories, Epics, Features

Epics

- As a gamer, I want to discover local tournaments and communities, so that I can participate on the tournaments and meet players with similar interests to myself.
- The system must provide games with tools to search, filter and join local and online tournaments by game, skill level and location, this will foster community engagement and participation.
- As a tournament organizer, I want to announce and manage events so that I can attract the maximum number of participants and grow the competitive scene.
- The system must provide organizers with a platform to create, publish and manage tournament events, this includes bracket generation, participant registration and notification.
- As a competitive player, I want to track my performance and rankings, so that I can measure progress and compare myself with others.
- The system must support the collection, storage and display of player data, allowing users to view ranking, match histories, and statistics on their performance.

- As a casual player, I want to join or create teams for my favorite games so that I can play cooperatively and find new friends to play my favorite games.
- The system must support users in creating, joining and managing gaming communities, this includes team formation, messaging and even participation, in order to enhance long-term user engagement.

User Stories

- As a gamer, I want to follow specific communities so that I receive notifications about upcoming events.
- As a gamer, I want to create a new community for a game without an existing one so that I can gather players with similar interests.
- As a competitive player, I want to view my local ranking so that I can see how I compare with others in my region.
- As a competitive gamer, I want to join a team for my favorite game so that I can participate in team-based competitions.
- As an organizer, I want to create tournament brackets so that matches are structured and easy to follow.
- As an organizer, I want to notify users about new tournaments so that they are aware and can sign up.
- As a games, I require a system that must be able of providing a way to follow specific communities and automatically receive notifications for upcoming tournaments and events.
- As a gamer, I require a system that must be able of supporting the creation of new communities for games not yet listed.
- As a competitive player, I require a system that must display accurate local ranking based on official tournaments results.
- As a competitive player, I require a system that must provide a leaderboard interface that updates after each match.
- As an organizer, I require a system that must be able of providing tools for automatic bracket generation and real-time result tracking.
- As an organizer, I require a system that must be able of sending notification to registered participants when a new tournament or event is created.
- As a casual gamer, I require a system that must provide a simple interface to join or create teams for friendly or competitive play.
- As an organizer, i require a system that must provide communication tools, such as an announcement board, to interact with participants.

Features

- Tournament and event search by both game and location.
- Community following and customizable notifications.
- Local and regional ranking system based on tournament results.
- Team creation and management tools.
- Bracket generation for competitions.
- Organizer tools for posting and updating events.
- Option to create new communities for games without an existing competitive scene.
- Tournament Discovery: The system must provide search filters by game, skill tier, and proximity to help users find events.
- Tournament Management: The system must allow organizers to configure tournament details, such as rules, registration limits, and brackets as well as publishing updates.
- Ranking and Statistics: The system must compute and update leaderboards after each tournament, storing individual and team statistics.
- Community Management: The system must provide means for users to create communities, invite members and post discussion threads.
- Notification and Alert System: The system must notify users about new tournaments, team invitations, and ranking changes.
- Team Coordination: The system must enable players to form teams and register as one for competitions.

2.2.2 Personas

Persona 1: Alex the Competitive Player

- **Age:** 21
- **Background:** University student, plays multiple esports titles, highly motivated by rankings and performance.
- **Goals:**
  - Find tournaments to test and improve skills.
  - Track rankings and stats across games.
- **Frustrations:**
  - Difficult to keep up with scattered tournament announcements.
  - Lacks a centralized platform to measure performance. Alex is a computer science major who spends his free time practicing games like CSGO and League of Legends. Outside gaming, He balances university coursework with a part-time job tutoring. He is data driven and

competitive, ofter analyzing his match performance and stats. His ideal platform is one that automatically records tournament results, visualizes ranking and helps him identify skill gaps. He also values social feature that help him connect with other players for practice sessions.

Persona 2: Maria the Organizer

- **Age:** 34
- **Background:** Works in event management, organizes local gaming tournaments on weekends.
- **Goals:**
  - Announce tournaments easily to the right audience.
  - Manage brackets and notify participants quickly.
- **Frustrations:**
  - Promotion spread thin across many platforms.
  - Hard to build consistent communities for recurring events. Maria works full time for a community recreation center and runs gaming events as a passion project. She is detail oriented, managing the logistics of the events as well as promotion across multiple social media platforms. Her biggest challenge is keeping registration data organized and communicating efficiently with participants. She needs a platform that makes event promotion and bracket management easier as well as updating results that help her build a loyal player base.

Persona 3: Liam the Casual Gamer

- **Age:** 26
- **Background:** Plays games for fun after work, sometimes interested in casual competitions.
- **Goals:**
  - Discover local communities for his favorite games.
  - Join teams to participate in friendly competitions.
- **Frustrations:**
  - Overwhelmed by too many platforms and event sources.
  - Wants simple notifications without constantly monitoring social media. Liam works as a graphic designer and enjoys unwinding with friends in games like Rocket League and R.E.P.O. He values simplicity and convenience, he does not want to browse ten discord servers just to end up finding a small weekend tournament. He is motivated by social interactions rather than simply winning, and wants a system that shows casual events nearby, has easy sing ups and minimal notifications.

2.2.3 Domain Requirements

- Events must be associated with an existing video game.

- The system-to-be shall require each event to be linked to an existing video game.
- Every event created must have at least one organizer.
- The system-to-be shall require at least one registered user to be assigned as an organizer before an event can be created.
- Only the event organizer should have permission to edit or cancel their events.
- The system-to-be shall require event modifications and/or cancellations to be performed exclusively by assigned event organizers.
- Each event must have a starting date, ending date, and location, whether it is physical or online.
- The system-to-be shall require all events to include a starting date, ending date, and a location (physical or online URL).
- The system-to-be shall ensure that the ending date is later than the starting date.
- Each team must consist of one or more users.
- The system-to-be shall enforce that each team registered in an event consists of at least one user account.
- The system-to-be shall block registration of empty teams.
- Events must have a limit of participants.
- The system-to-be shall require organizers to define a maximum number of participants (teams or players) for each event.
- Once the limit is reached, no more users should be allowed to register for the event.
- The system-to-be shall prevent additional registrations for an event once the defined participant limit has been reached.
- Once an event is over, no registrations or modifications to the event should be allowed.
- The system-to-be shall automatically disable registration and editing for a specific event once it has concluded.
- The results of an event must be recorded once the event is finished.
- The system-to-be shall ensure results of each event are available to users for ranking and visualization.
- Appropriate ranking updates must be done based on the results of finished events.
- The system-to-be shall update team and/or player rankings based on recorded results of completed events.
- The system-to-be shall prevent an event from being created without an organizer.
- The system-to-be must prevent duplicate teams or event names.

2.2.4 Interface Requirements

- The system must allow for users to create and manage events.

- The system-to-be must provide a way for users to create events, which must have a start and end date, a location, an organizer (usually the user that created the event) and a subject or game the event will be about. Event properties must be stored as: a date, in the format year, month (by index starting at 0), day, hour, and minute, a location name or address, a user ID, and a subject name or event. In addition, the system-to-be must provide a means to edit events created by them.
- The system-to-be must reject any attempt by a user to create an event/team with a duplicate name of an existing event/team within the database and provide an appropriate warning upon creation attempt.
- The system must allow users to search for events and communities within specific locations.
- The system-to-be must provide the means to be able to search for user created events and communities and provide a way to filter these by categories, such as by location or subject.
- The system must track and update user rankings as needed.
- The system-to-be must rank users based on their performance in events they have participated in, utilizing a custom algorithm, and provide a means for users to see their rankings and compare them with other user rankings.
- The system must allow users to join teams and communities of their choice.
- The system-to-be must provide a means for users to join teams and communities of interest and provide a means for users to communicate via messaging with one another within those teams and communities.
- If a community or team does not exist for a certain game, the system must allow the user to create one.
- If a community or team does not exist for a certain game or subject, the system-to-be must provide a way for the user to create one with the desired subject.
- The system must allow event organizers to edit their events, start and end dates and location, until the event has started.
- The system-to-be must provide a means for users to edit their event properties, start and end dates, location, and subject, requiring that the user originally created the event and the event has not passed its start date.
- The system-to-be must provide a form through which users can register for events to mark and record their participation in the event.
- Any registrations passed the start or end of an event must not be allowed.
- The system-to-be must reject any attempts users make to register to an event past the event's start date.
- The system must allow event organizers to record event results once the event has finished.
- The system-to-be must provide a way for an organizer of an event to view the results and/or progress of the event, including after the event has finished.
- The system must notify users of new events relevant to their interests.
- The system-to-be must notify, whether by email or SMS, users of new or existing events, teams, and

communities relevant to the user's interest.

- The system-to-be must provide a means for tournament organizers to publish announcements regarding upcoming events or community updates that must be visible to all participants.

2.2.5 Machine Requirements

- The system must support at least 450 users at a time with an average response time of 2 seconds or less.
- The system-to-be must be able to process simultaneous requests, such as joining and creating events or community interactions, from at least 450 active users without exceeding an average response time of 2 seconds per request.
- The system must be available at least 99% of the time per month.
- The system-to-be must maintain an uptime of at least 99% of the time per month. Excluding the scheduled maintenance that must not exceed 3 hours per session.
- The system must be able to keep user data secure within the website.
- The system-to-be must ensure the integrity and availability of user data. Including secure password storage and role-based access to restrict access to sensitive data.
- The system must be able to handle at least 750 registered users without major performance decrease.
- The system-to-be must ideally maintain a latency of 200 - 600 ms, in user interactions such as post creation functionality, tournament registration and community interactions with a user base as big as 750 concurrent players.

2.3 Implementation

The implementation stage translates the requirements described in Section 2.2 into a working system. While requirements define **what** the platform must achieve, the implementation details **how** these goals were realized.

For Milestone 2, this section has been expanded to include current progress on the Profiles module, showing concrete examples of backend and frontend integration that transform previous design concepts into a functional system.

The **software architecture** captures the big picture of the system: * Main modules include authentication, user profiles, tournament management, match reporting, and community management. * These modules communicate through well defined APIs and shared data stored in Firestore. * A backend service (Node.js/Express) now handles the Profiles module, which interacts with Firestore and the React client through REST endpoints. * The local environment has been successfully tested and launched, confirming connectivity between modules and databases. The `/profiles` endpoint was used for testing profile retrieval functionality in the local development environment.

The **software design** explains how each component is realized: * **Authentication:** Implemented with Firebase Authentication and OAuth2 providers such as Google, Discord, and Twitch. * **User profiles:** Stored in Firestore, including usernames, emails, stats, teams, and communities. * **Tournament management:** Collections store brackets, matches, and schedules, enforcing limits on participants like defined in Section 2.2.3 Domain Requirements. * **Match reporting:** Organizers and players update results, which automatically update user rankings. * **Communities and teams:** Users can create or join them, ensuring inclusivity and adaptability. * **Profiles module (Milestone 2 addition):** Integrates CRUD operations and privacy settings. Implemented functions such as `getParticipatedTournaments()` within the Profiles module retrieve user data directly from Firestore, ensuring up-to-date synchronization between the backend and the database. Additionally, asynchronous functions like `saveProfile`, `getProfile`, and `updateProfile` manage the creation, retrieval, and updating of user profiles within Firestore, maintaining accurate and timestamped records.

Architecture Diagram: The following UML/Architecture diagram illustrates how the Profiles module connects the frontend React client with the backend API, Firebase Authentication, Firestore, and related modules.

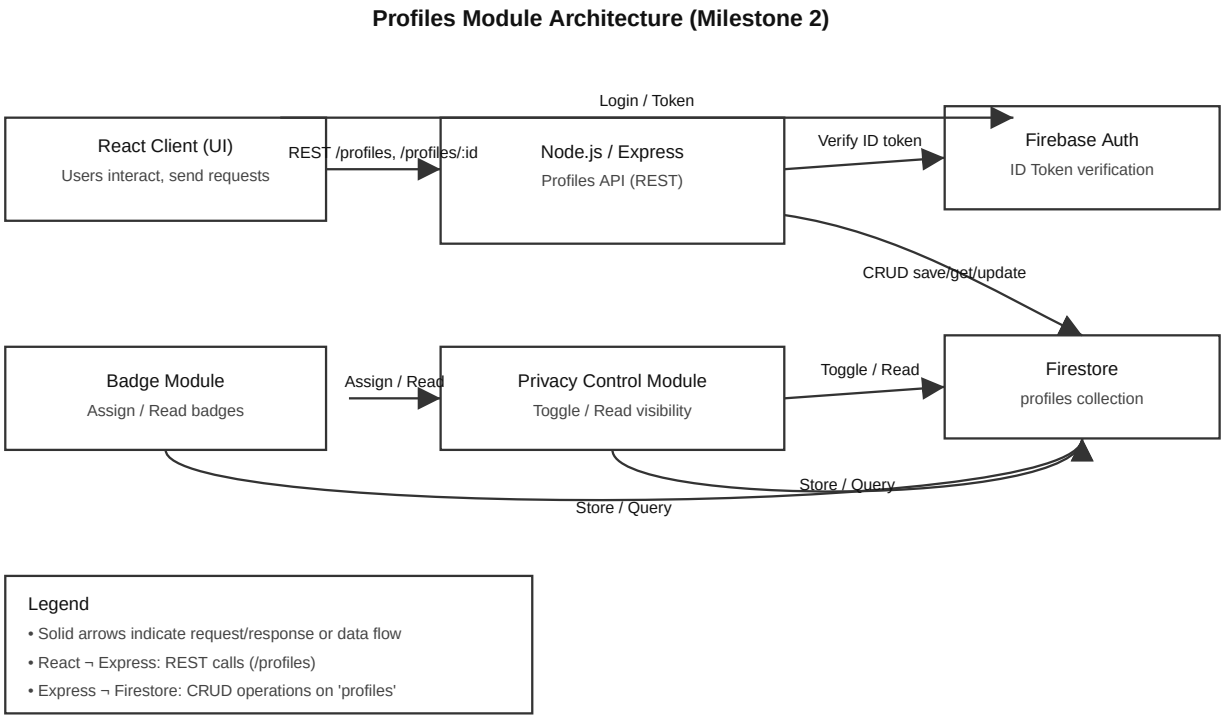


Figure 1. Image: Profiles Module Architecture (Milestone 2)

Diagrams and screen mockups complement the text: * Architecture diagrams highlight module interactions. * Sequence diagrams show workflows such as user login and event creation. * A UML or architecture diagram now illustrates how the Profiles module interacts with the Badges and PrivacyControls components (see `profiles_architecture.svg`). * Figma mockups illustrate how the interface supports usage scenarios.

2.3.1 Selected Fragments of the Implementation

Selected fragments are only included when they clarify explanations and complement the documentation.

For Milestone 2, the following code excerpts from the Profiles module demonstrate Firestore integration and CRUD functionality. These fragments show how user-related data is retrieved, created, and updated through asynchronous operations, confirming full connectivity between the backend and the Firestore database.

```
import { collection, getDocs } from "firebase/firestore";
import { db } from '../lib/firebase.js';

// All console logs are for debugging purposes to understand the Firestore data and see
if we are fetching it correctly.
// The debugging would be seen in the browser console.
async function getParticipatedTournaments() {
    const querySnapshot = await getDocs(collection(db, "User"));

    console.log("=== FIRESTORE DATA STRUCTURE ===");
    console.log("Total documents:", querySnapshot.size);

    querySnapshot.forEach((doc) => {
      console.log("Document ID:", doc.id);
      console.log("Document data:", doc.data());
      console.log("All fields in this document:", Object.keys(doc.data()));
      console.log("---");
    });

    const User = querySnapshot.docs.map((doc) => ({
      id: doc.id,
      ...doc.data()
    }));

    return User;
}

export default getParticipatedTournaments;
```

Explanation: This function retrieves all documents from the “User” collection in Firestore and displays key information through console logs. It verifies that Firestore connectivity and data structure are working correctly in the current environment. This implementation satisfies the system requirement that **“the system must provide a way to retrieve stored user data from Firestore.”**

The following fragment complements the previous example by showing how user profile information is created, retrieved, and updated. This code implements the CRUD logic for the Profiles module and

ensures that user data is stored with proper timestamps and validation.

```
import { db } from '../lib/firebase.js';
import { doc, setDoc, getDoc, updateDoc, serverTimestamp } from 'firebase/firestore';

/**
 * @param {Object} profileData
 * @param {string} profileData.email
 * @param {string} profileData.name
 * @param {string} profileData.photoUrl
 * @param {string} profileData.role
 */
export async function saveProfile(profileData) {
  if (!profileData.email || !profileData.name || !profileData.role) {
    throw new Error('Campos obligatorios faltantes: email, name o role');
  }

  const ref = doc(db, 'profiles', profileData.email);
  await setDoc(ref, {
    ...profileData,
    createdAt: serverTimestamp()
  });

  return { success: true };
}

/**
 * @param {string} email
 */
export async function getProfile(email) {
  const ref = doc(db, 'profiles', email);
  const snap = await getDoc(ref);

  if (!snap.exists()) {
    return null;
  }
  return snap.data();
}

/**
 * @param {string} email
 * @param {Object} updates
 */
export async function updateProfile(email, updates) {
  const ref = doc(db, 'profiles', email);
  await updateDoc(ref, {
    ...updates,
```

```
    updatedAt: serverTimestamp()
  });

  return { success: true };
}
```

Explanation: These asynchronous functions (`saveProfile`, `getProfile`, and `updateProfile`) manage CRUD operations for user profiles. They interact directly with Firestore, adding timestamps (`createdAt`, `updatedAt`) to track data modifications. This code demonstrates how the system fulfills requirements such as **“the system must provide a way to create and manage user profile data”** and **“the system must provide means to update stored information reliably.”**

Best practices followed in this documentation: * **No screenshots** of code — only properly formatted snippets. * **Scalable images** (SVG, PDF) for diagrams and mockups instead of raster images (JPEG, PNG). * **Fragments included only when they clarify**, never for their own sake.

3 Analytic Part

3.1 Concept Analysis

Based on our understanding of the esports domain and the problem space, we identify the following key concepts with their derivation from stakeholder input:

Game vs Gaming Community: Games are the software titles (Tekken, Valorant), while **Gaming Communities** are groups of players who compete in specific games within geographic regions.

Derived from: "people who play Tekken" and "Valorant players in our city" indicate distinct player groups organized around specific game titles within geographic boundaries.

Tournament vs Match: Tournaments are organized competitive events with multiple participants, while **Matches** are individual competitions between players/teams within tournaments. The bracket reference indicates tournaments contain structured match progressions.

Derived from: "organize tournaments," "track results from multiple events," and "bracket" references demonstrate the hierarchical relationship between tournaments and their constituent matches.

Geographic Locality: Multiple references to "local," "in our city," and "geographic areas" reveal that competitive gaming operates within **Local Competitive Scenes** - geographically-bounded communities where players can feasibly attend in-person events.

Derived from: "hard to know who’s actually good in our area," "local competitive scene," and "in our city" emphasize the geographic boundaries that define competitive communities.

Performance and Rankings: The "3rd place" and "ranking across events" statements show that **Competition Results** and **Player Rankings** are important domain concepts that currently exist in

fragmented form.

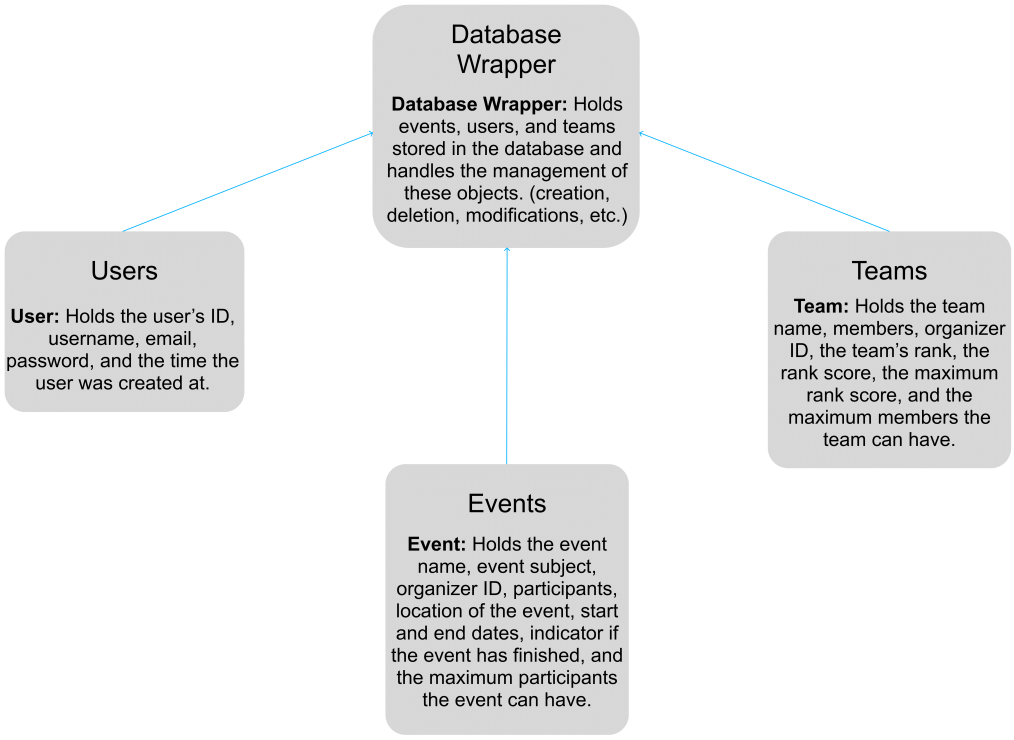
Derived from: "I got 3rd place at X tournament," "track my performance across events," and "hard to know who's actually good", indicates that performance tracking and comparative rankings are central concerns.

Individual vs Team Competition: Some games support both individual and team play (e.g., fighting games typically individual, MOBAs typically team-based). Our domain model must accommodate both.

Derived from: References to "players" (individual) and the diversity of game types mentioned (Tekken as fighting game, Valorant as team-based) imply different competition structures.

Casual vs Competitive Players: The distinction between someone who "plays games" and someone who "competes in tournaments" is crucial for our domain focus.

Derived from: "people who play Tekken" versus "organize tournaments" and "track results from multiple events" distinguish recreational players from competitive participants who engage in structured competition.



The diagram above illustrates our system's data architecture, which directly implements the domain concepts identified in our analysis. The architecture consists of four primary components:

Database Wrapper: Serves as the central data management layer, handling the creation, deletion, and modification of events, users, and teams. This component ensures consistent data operations across all entity types and maintains data integrity throughout the system.

Users Entity: Represents individual competitive players (mapping to our "Competitive Players" concept), storing user ID, username, email, password, and account creation timestamp. This entity distinguishes tournament participants from casual players.

Teams Entity: Accommodates team-based competition (supporting our "Individual vs Team Competition" concept), containing team name, members, organizer ID, rank metrics, and capacity constraints. This enables team-based games while maintaining performance tracking.

Events Entity: Implements our "Tournament" concept, storing event name, game subject, organizer, participants, location (supporting "Geographic Locality"), dates, completion status, and participant capacity. This entity provides the foundation for tracking "Competition Results" across the "Local Competitive Scene."

The relationships between entities enable tracking individual and team performance across multiple events, managing team compositions, associating events with geographic locations, and linking organizers to their tournaments.

3.2 validation and verification

Validation determines whether stakeholders agree with our understanding of the esports domain as we have documented it.

Domain Concept Validation:

Present our identified concepts to stakeholders and ask for their agreement:

- Present our concept of **Gaming Community** as "groups of players who compete in specific games within geographic regions" to community members and verify this reflects their experience
- Share our understanding of **Local Competitive Scene** as "geographically-bounded communities where players can feasibly attend in-person events" with tournament organizers and participants

Domain Understanding Validation:

Present our domain analysis directly to stakeholders:

- Share our understanding that tournaments contain structured match progressions organized in brackets, and verify this reflects how competitive events actually operate
- Show our understanding that competition results and player rankings currently exist in fragmented form across different platforms, and ask stakeholders if this characterizes their current situation

Terminology Validation:

Present our terminology definitions to stakeholders and ask for confirmation:

- Confirm that our definition of **Match** as individual competitions within tournaments aligns with

how stakeholders use this term

- Check that our concept boundaries between different domain entities match stakeholder understanding

Verification Strategy

All concepts in the domain are used consistently across documentation, requirements, and architecture. Requirements clearly trace back to domain properties, and every property that affects the system generates the right requirements. The software architecture covers all specified requirements without gaps, with components having clear responsibilities. The data model represents all domain concepts without conflicts. Implementation matches the design, with unit tests covering all components and interfaces working as specified. Finally, every requirement has a matching test case, and testing environments reflect the operational conditions defined.

Conceptual Consistency: All concepts in the domain are used consistently across documentation, requirements, and architecture. Requirements clearly trace back to domain properties, and every property that affects the system generates the right requirements. Architectural Completeness: The software architecture covers all specified requirements without gaps, with components having clear responsibilities. The data model represents all domain concepts without conflicts. Implementation Alignment: Implementation matches the design, with comprehensive test coverage ensuring correctness. Every requirement has a matching test case using the following test types: Unit Tests: Cover individual component logic, domain model behavior, and business rule validation. These tests verify that individual classes and methods function correctly in isolation. Integration Tests: Verify interactions between system components, database operations, and service layer functionality. These tests ensure that the Database Wrapper correctly manages Users, Teams, and Events entities. API Tests: Validate interface contracts, request/response formats, and endpoint behavior. These tests confirm that external interfaces adhere to specifications and handle edge cases appropriately. End-to-End Tests: Confirm complete user workflows from tournament creation through result tracking. These tests simulate real user scenarios, such as creating an account, joining a team, registering for an event, and viewing rankings. Acceptance Tests: Verify that implemented features meet stakeholder requirements as defined in user stories. These tests ensure that the system delivers the value promised to competitive gaming communities. Testing environments reflect the operational conditions defined in requirements, with test data representing realistic competitive gaming scenarios.

Success Criteria

Validation Success Indicators: - Stakeholders recognize their experiences in our domain scenarios and confirm our understanding is accurate - Stakeholders agree with our concept definitions and the relationships we've identified between domain entities - When stakeholders suggest modifications, they represent refinements rather than fundamental misunderstandings of the domain

Verification Success Indicators: - All cross-references between project documents are accurate and consistent - Domain concepts are used consistently across all development phases - Requirements properly trace to domain properties without gaps or contradictions - Software architecture adequately addresses all specified requirements without conflicts - Each requirement maps to at least one test case

of the appropriate type(s)

Application of Topics

Algebras and Closure Under Operations

The project demonstrates algebraic closure through refined function signatures across multiple domains. Event and Team classes evolved from long parameter lists to single initializer objects with implicit defaults, ensuring valid object states. Tournament bracket generation implements closure through the `generateBrackets(tournament) → BracketStructure | Failure` function, which takes tournament parameters and returns a complete bracket structure or explicit failure state.

Agile Practices

Following Scrum methodology with bi-weekly sprints, the team organized around specialized roles including Managers, Team Leaders, and Development Teams (Database/Backend, Events, Profiles, Social Features, UI). Weekly ceremonies included sprint planning, progress reviews, and retrospectives. GitHub issues tracked feature development through branching and pull requests. This structure enabled continuous delivery of backend features and improved cross-team communication.

Decision-Making Processes

The team applied systematic evaluation methods for technical choices, such as the communication platform selection comparing Email versus Discord. Using weighted scoring matrices based on non-functional requirements (speed, ease of use, organization, scalability), Discord was selected with a score of 1.8 versus Email's 0.4. Similar evaluation processes were used for database design and architecture decisions.

Process Modeling

Petri nets modeled competing processes in tournament registration, particularly for limited slot allocation. Sequence diagrams (Figure 1) illustrated tournament lifecycle interactions between hosts and participants. State charts (Figure 2) defined tournament progression through creation, registration, match progression, and completion states.

Real-Time System Design

The implementation uses Firebase's real-time capabilities with React state management for live bracket updates. Cloud Firestore's `onSnapshot()` listeners provide immediate data synchronization, while React's `useEffect` and `useState` hooks manage component state updates. This enables automatic bracket propagation when match results change.

Tournament Algorithms

Single elimination bracket generation implements mathematical models using power-of-two calculations, bye allocations, and balanced binary tree structures. The system handles various tournament formats with algorithms for participant counting, round calculation (`ceiling(log2 n)`), and fair seeding through random shuffling or rank-based sorting.

Diagrams

```
classDiagram
direction TB

%% App source files
class App {
  App.jsx
}
class Main {
  main.jsx
}
class Team {
  team.js
}
class CommSocial {
  Comm-Social/
}
class Components {
  components/
}
class Pages {
  pages/
}
class Services {
  services/
}
class Database {
  database/
}
class Lib {
  lib/
}
class Events {
  events/
}
class Functions {
  functions/
}
```

```
App --> Main : "bootstrap"
App --> Team : "imports data"
App --> Components : "uses UI"
App --> Pages : "routes to"
Pages --> Components : "compose"
Services --> Database : "reads/writes"
Functions --> Events : "handles triggers"
Lib --> Database : "utils"
CommSocial --> Pages : "community features"
Events --> Services : "publishes events"
```

App bootstraps Main and pulls in domain models (Team, Events), UI (Components, Pages, CommSocial), logic (Services, Functions), persistence (Database, Lib). Arrows indicate directional dependencies (e.g., Pages compose Components, Services read/write Database, Functions handle event triggers).

In practice this means user interactions flow from Pages → Services → Database, yielding clear separation of concerns, easier testing, and safer evolution of UI and storage/persistence layers.

LogBook

Section Name	Member	Added or Modified	Description
Rough Sketch	Pedro Bonilla	Added	Added a Flowchart to better explain the way communities, players, teams, and organizers work.
Informative Section 1.1	Yamilet Gomez	Modified	Clarified distinction between clients and stakeholders; added explanation on internal team communication to prevent dependency issues; rephrased statement about project being carried out only by students.

Section Name	Member	Added or Modified	Description
Descriptive Section 2.1.1 - 2.1.6	Hector Rivera	Modified	Modified several sections to improve clarity and fix formatting issues in the Narrative, Domain Terminology, and Function Signatures sections.
Introduction Section	Jayden Sánchez	Added	Added the presentation page, table of contents, and legend to define the structure of the documentation and introduced the introductory part to explain the project's purpose and organization.
Analytical Part Section 3-1 - 3-2	Ricardo Burgos	Modified	Enhanced concept analysis with quote derivations; added database architecture diagram explanation; expanded verification strategy with five test types and coverage metrics.
Requirements Section 2.2.3 - 2.2.5	Andrés Cruz Zapata	Modified	Rephrased all requirements to improve clarity and provide more detailed descriptions for each requirement established.

Section Name	Member	Added or Modified	Description
Informative Section 1.2 - 1.4	Yamilet Gómez	Modified	Integrated “situation, needs, and ideas” with clearer examples of domain and requirements engineering. Rephrased “platform” to “hub” to avoid confusion with “system”, and added focus on small-scale tournaments.
Informative Section 1.2 - 1.4	Yamilet Gómez	Modified	Integrate section 1.4 with 1.2 and 1.3, in order to completely delete section 1.4
Desciptive Sections 2.2.1-2.2.2	Pedro Bonilla	Modified	Rephrased to improve clarity and provide more detail as per feedback.
Derived Goals	Jayden Sánchez	Modified	Revised the Derived Goals section to clarify their independence from the main objectives and strengthen the section’s purpose.