Esports Organizer

Third Milestone

Prof. Marko Schütz

Introduction to Software Engineering (INSO4101)
Section 080
Professor Marko Schütz
Friday, November 28, 2025

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 centralize all esports-related tools and information into a single, accessible space, 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. Designed to support both casual participants and dedicated competitors, the platform gives each user a space that adapts to their needs as they grow within the community. 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. This modular approach ensures that updates can be introduced with minimal disruption, maintaining a stable and intuitive user experience.

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.

During the later stages of the project, and in order to improve communication and overall project flow, the Player and Teams Profile team and the Identity and Data Systems team were merged. This integration helped minimize misunderstandings, reduce duplicated work, and ensure that player-related features were directly aligned with backend structures and functionality.

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 Situation, 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.
- Promote healthy digital behavior by reinforcing respectful communication and balanced competitive practices across communities.
- Expand opportunities for new organizers by giving emerging leaders simple tools to host events and participate in community growth.

These derived goals differ from the main objectives: while the primary goals focus on making the platform functional and accessible, the derived goals lead to independent cultural, educational, and social outcomes that are valuable 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 that exist in the real-world domain of esports tournaments. It focuses on how players, teams, organizers, and communities participate in competitive events: how teams are formed, how tournaments are announced, how matches are

scheduled and played, and how results affect rankings and future competitions.

The description is organized into several parts. The rough sketch captures informal, everyday statements from people involved in esports, showing how they naturally talk about tournaments, rosters, formats, and schedules. The terminology section then abstracts these observations into stable domain concepts and names. The narrative describes a typical tournament lifecycle, from announcement and registration to bracket progression and closure of the event.

Finally, the sections on events, actions, behaviors, and function signatures structure the dynamic phenomena of the domain. They identify what happens (events), what people or organizers deliberately do (actions), how multi-step flows unfold over time (behaviors), and how these phenomena can be expressed as abstract operations over domain concepts. Together, these views provide a coherent description of the esports tournament domain that later requirements and designs can trace back to.

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.~~

The following are statements people would say in their day to day life.

- "Hey Carlos, I just wanted to say that im very glad I met you at the last tournament. I had a really difficutl time making friends and always wanted to be part of a community that enjoyed the same games as I do. Ever since joining the Smash Bros community here, Ive made a lot of good friends like you and have had a great time participating in events with them."
- "Hey Fabian, Ive been playing Super Smash Bros day and night but dont have anyone to play against. I was thinking of organizing an event in a neat manner where I can find suitable opponents and make a fun competition for everyone. Would you be interested in helping me set it up?"

- "What are we gonna do guys? We need one more member to complete the roster qualifications if we want to compete in the UPRM Marvel Rivals tournament in May. We need to make sure our team is registered two weeks before the tournament too. Any ideas on who we can invite to join us?"
- "Alright people if we're seriours about forming our esports team we need to decide on a captain. This person will be in charge of inviting people, updating the roster and signing us up to tournaments. I think Tomas would be a great choice since he's organized and knows a lot about the games we play. What do you guys think?"
- "Hi Maria, have you seen the requirements posted on the site for the Mario Kart Tournament in February? The organizers stated that you have to be older than 13. They also mentioned that teams need to have 4 members and only people from the west side of the island can enter this specific tournament. Would you be interested in participating with me? Im looking for people from campus that live around here like me in order to be able to form my team."
- "Hi are you one of the organizers for the tournament? I just wanted to say that I noticed how you didnt just put teams randomly in a bracket; instead it seems that teams where seeded based on how well they did in previous tournaments or online qualifiers. I think that was a great idea since I could see and experience first-hand how it made the competition more balanced and fair for everyone."
- "Oh hi Gabriela! Sorry I havent been able to go out with you much this month. Ive been really busy competing on events around the island. Its been fun to play in events with different formats. Some are single elimination so you lose once you're out. Others are double elimination, round robin or that Swiss system where you keep playing people with similar records. Although I've been busy with the torunaments and school work I've had a grat time and have met a lot of people passionate about the same games as me."
- "Hey team, just a reminder that we have a match coming up this Saturday at 3 PM. Make sure to check in on the website at least 15 minutes before the match starts. If we dont check in within the grace period, our team will automatically forfeit the match. Let's make sure we're all on time and ready to play!"
- "Bro, last night's match was insane! Since after every match score were reported you could see how close each team was to winning. I remember in a round robin tournament two teams ended with the same number of wins, so they had to use things like head-to-head results or point differences to decide who placed higher. It made every point count, and it was so exciting to watch!"
- "The popularity the esports community is gaining is surprising. Even people who aren't playing still follow everything. I heard on reddit that some people keep checking the brackets, match schedules, and standings to see who is winning and who got knocked out. Some even stream or share updates on Instagram and Facebook. Yesterday I even heard my grandma talk about how some kid won a five thousand dollar price on Fortnite tournament in Alabama. It's amazing to see how much interest there is in esports nowadays!"

Figure 2.1.1 - Tournament and community interaction flow

This diagram illustrates the relationship between players, organizers, and communities. In this diagram:

- **Blue rectangles** represent entities (players, teams, organizers, communities).
- **Red diamonds** represent actions (register, create, update, check-in, etc.).
- **Arrows** indicate the flow of interactions or dependencies between entities and actions.

If you want to focus on entities, a class diagram could be used, with relations labeled by the actions they enable. Since this is not a standard UML diagram, the graphical elements are explained above for clarity.

2.1.2 Terminology

This section defines key domain concepts observed in esports tournaments. These terms originate from the real-world phenomena described in the rough sketch and serve as stable abstractions used consistently throughout requirements, design, and validation.

Each term emphasizes what exists or happens in the esports domain itself, without referencing interfaces or technical system behavior.

- Player: An individual who participates in competitive gaming events, forms part of a community, and may join or create teams.

- Team: A coordinated group of players who share a name, identity, and roster. Teams participate in tournaments and may designate a captain to manage administrative responsibilities.
- Captain: A team member who takes on leadership tasks such as inviting players, organizing the roster, communicating with organizers, and handling registration-related responsibilities.
- Organizer: A person or group that plans, schedules, and manages esports tournaments, defines eligibility rules, determines formats, and oversees match progression.
- Tournament: A structured competitive event in which teams face one another according to a defined format (e.g., single elimination, double elimination, Swiss, round robin).

Figure Result Report Interaction Flow. This diagram illustrates the flow of the result report creation process.

- **Blue rectangles** represent actions (Create array, trow error, etc).
- **Purple diamonds** represent questions (Is the data empty?, Are there more teams to import.).
- **Arrows** indicate the flow of interactions.
- [.hl-green]#**Tournament Data**: Data on the tournamnets, final standing, KDA etc. used to create the final ranking list.
- **Ranking array**: Final ranking list. every team is placed on the order of where they finished on the ladder.#
- Match: A competitive encounter between two teams within a tournament, whose outcome contributes to bracket progress, standings, or ranking calculations.
- Bracket: The competitive structure that represents match pairings, progression paths, and advancement rules across the tournament's stages.
- Spectator: An individual who follows tournament progress, standings, schedules, or match outcomes, regardless of whether they actively compete.
- Community: A social group of players and teams who share common interests, participate in events, and interact across local or online competitive environments.
- Stage: A recognizable phase in the lifecycle of a tournament, such as announcement, registration, seeding, match progression, and finalization. Each stage has its own domain rules, deadlines, and phenomena that affect how teams advance or prepare for competition.

These terms establish conceptual stability across the documentation, ensuring that later requirements and designs reference the same domain meanings introduced here.

2.1.3 Relationship Between Rough Sketch and Terminology

The Rough Sketch captures informal, everyday statements from players, friends, and organizers, showing how people naturally describe tournaments, rosters, deadlines, formats, and eligibility in real-life esports communities. These unstructured observations reveal the raw phenomena of the domain:

who acts, what happens, and what constraints exist.

The Terminology section formalizes these raw observations into stable domain concepts. Each term provides a precise name for a recurring idea that appears throughout the Rough Sketch but is expressed informally in different ways.

For example, participants talk about “checking in 15 minutes before the match” or “missing the grace period and forfeiting.” These conversational descriptions are abstracted into the concept of a Check-In Policy, which defines expectations and outcomes around attendance before match start times.

Statements about “needing four players,” “finding a fifth teammate,” or “only players from the west side can join” correspond to domain-level Eligibility Constraints and Roster Composition Rules.

Mentions of previous results, qualifiers, or balanced brackets are abstracted into Seeding Criteria, a domain concept describing how teams are ordered before competition begins.

When players discuss single elimination, double elimination, round robin, or Swiss formats, the terminology identifies these as elements of Tournament Format Specification, a stable concept describing the structural logic of competition.

Even casual comments about spectators “checking brackets,” “following standings,” or “watching streams” reflect the presence of Spectator Engagement, which the terminology formalizes as a domain concept capturing non-player participation.

This abstraction process ensures consistency and closure across the documentation: every informal phenomenon described in the Rough Sketch maps cleanly to a concept in the Terminology section, which will later align with requirements, design models, and function signatures.

2.1.4 Expanded Narrative

The esports tournament domain is structured around several key concepts: tournaments, teams, matches, organizers, communities, and stages. A **stage** is a phase in the tournament lifecycle (e.g., registration, bracket generation, match play, results update, closure). Each stage has its own rules, actions, and transitions.

- ~~For single-elimination tournaments, the bracket is structured as a balanced binary tree, ensuring fair seeding based on rank or random assignment.~~

The process begins with an organizer announcing a tournament, specifying its format, rules, and registration window. Players form teams, designate captains, and register for the event. During the registration stage, eligibility is checked (account status, ranking, roster size). If too many teams register, a waitlist is created. When registration closes, the bracket is generated—often as a binary tree for single-elimination, or other structures for Swiss/round-robin formats. Teams may receive byes if the participant count is not a power of two.

- ~~Teams may receive byes if the total participant count is not a power of two.~~

As the tournament progresses through its stages, matches are played round by round. Winners advance to the next stage, losers are eliminated. After each match, results are updated, scores are recorded, and tiebreakers may be applied. The final stage is closure, where results are published and rankings updated. Throughout, communities and spectators follow progress, and organizers manage logistics and communications.

The narrative covers the full lifecycle, emphasizing how each stage (registration, bracket generation, match play, results, closure) plays a distinct role in the domain. The concept of "stage" is central: it structures the flow, determines available actions, and governs transitions between phases.

2.1.5 Unified Domain Concepts and Operations

Entities, actions, events, and behaviors are all phenomena in the esports domain. They are not separated into distinct sections, but are described together to show how they interact and support domain observations.

- **Entities**: Team, Tournament, Match, Player, Organizer, Community, Stage
- **Actions/Operations**: Create tournament, register team, generate bracket, assign captain, update match result, record forfeit, send notification, perform map veto, check-in team
- **Events**: Registration window closes, match completed, bracket generated, player disconnect, roster update deadline reached
- **Behaviors**: Run tournament flow, handle registration, update bracket progression, manage real-time updates

Each action/operation is paired with its function signature, showing how it expresses domain observations. For example:

- **registerTeam(team, tournament)**: Used when a team registers for a tournament. The function requires a team and a tournament; teams are created by players, tournaments by organizers. If you need a team, use a function like `createTeam(player, name, ⋯)` (not shown here, but implied by the domain).
- **updateTeamScore(teamID, points)**: Updates a team's score. Points may be produced by match results, so the function should clarify where points come from (e.g., `recordMatchResult` returns points). If you have only a teamID, you may need to resolve it to a team object via the tournament or a lookup function. Alternatively, `updateTeamScore` could take a team object directly, or the Match object could be responsible for updating scores for all teams involved.

These decisions require discussion and negotiation among the project team. For example, should `updateTeamScore` take a teamID or a team object? Should the Match object update scores for all teams? Each approach has tradeoffs and may require changes to the implementation. The function signatures are grouped with their actions/operations to clarify how they support domain scenarios.

2.1.6 Function Signatures (Grouped with Actions)

Function signatures define how operations are represented programmatically, describing inputs, outputs, and failure conditions. They are grouped with their corresponding actions/operations to clarify how they support domain scenarios and observations.

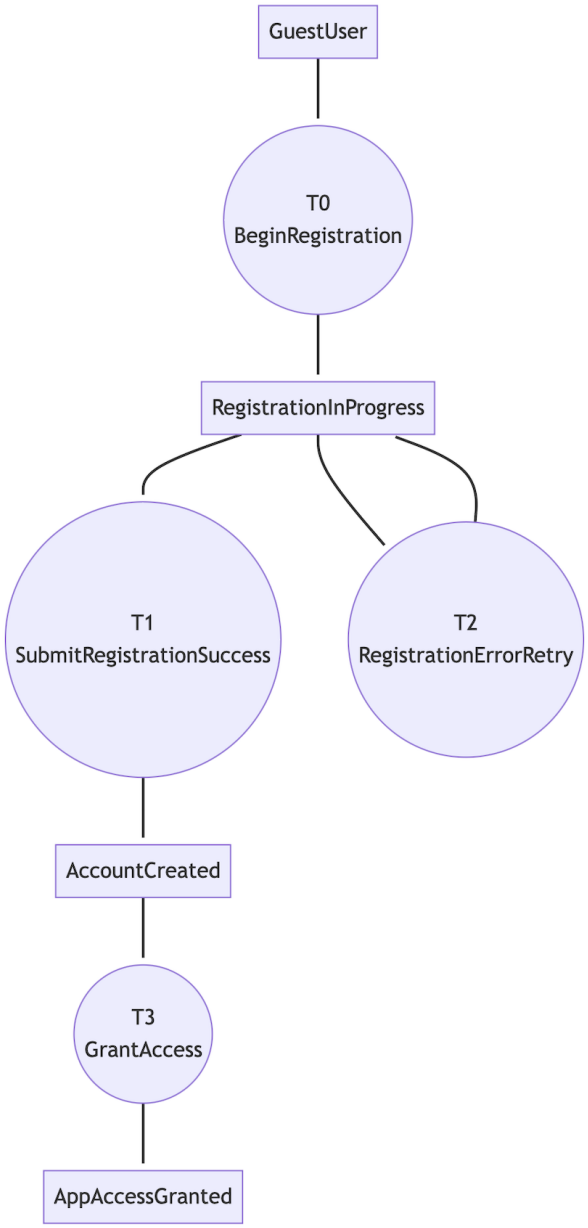


Figure 2.1.2 - Account creation flow

- **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.

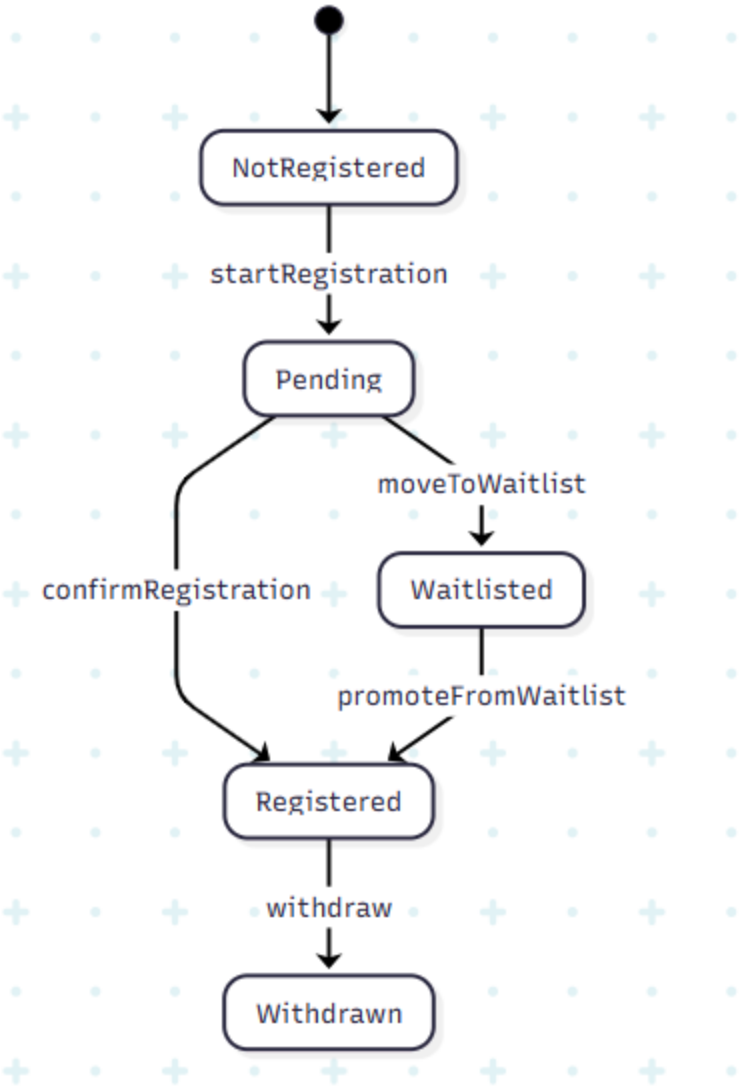


Figure 2.1.3 - Example registration flow

- **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. Points are produced by match results.
- **checkInTeam(teamID, tournamentID) → StatusUpdated | Timeout**
  - Verifies attendance before match start; returns timeout if grace period expires.

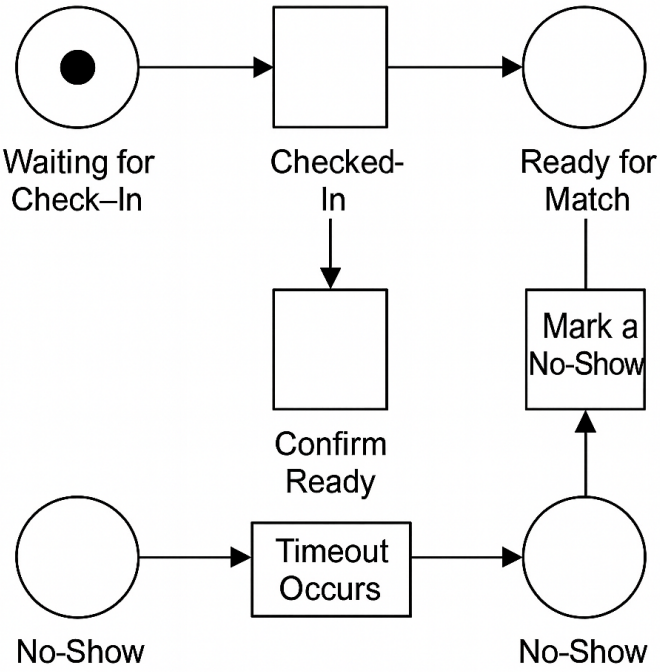


Figure 2.1.4 - Example check in flow

Model Explanation

[.hl-green]#This Petri net model represents the complete player check-in flow within the Esports Organizer system. The goal is to clearly illustrate how a player progresses through the system before a match begins, how delays or no-shows are handled, and how different transitions alter a player's state. Using a token-based representation makes the flow intuitive and highlights how the system behaves under different scenarios.

The model begins with the player in the **Waiting for Check-In** state. A token is placed in this initial place to represent one player who has not yet interacted with the check-in system. When the player arrives and performs the check-in action, the transition **Player Arrives** fires, moving the token into the **Checked-In** state. This represents the user pressing the "Check-In" button in the interface.

Once the player is in **Checked-In**, they must confirm readiness before the match can begin. When the confirmation occurs, the **Confirm Ready** transition fires, moving the token to the **Ready for Match** state. This indicates that the player has completed pre-match requirements. In the real system, this corresponds to backend logic updating the player's ready status.

If the player does not check in within the allowed time, the **Timeout Occurs** transition fires, sending the token from **Waiting for Check-In** to **No-Show**. This is triggered by a countdown or scheduled function. Admins may also manually flag a player absent using the **Mark as No-Show** transition, moving the token from **Checked-In** to **No-Show**.

These transitions illustrate how the system handles normal and exceptional flow paths. The model

ensures that all possible outcomes are explicitly defined—arriving on time, confirming readiness, failing to check in, or being manually marked absent.#

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.

Design decisions about function parameters (e.g., using teamID vs. team object, updating scores via Match or Team) should be discussed and negotiated by the project team. Implementation may need to be revised as the domain model evolves.

2.1.7

Identifying Carrier Sets and Operations for Core Esports Organizer Components

Yasser Alonso Ruiz, November 26, 2025

Introduction

[.hl-green]#This task applies the concepts on algebras in requirements engineering. The goal is to identify the carrier sets, operations, arity, and closure properties for two small components in the Esports Organizer system. This clarifies how each subsystem behaves, what elements it manipulates, and what transformations are valid within that subsystem.3

For this LTT, the selected components are:

1. Teams 2. Notifications

These components were selected because they represent distinct system behaviors: one models group structure and membership, while the other models communication flow and event-triggered updates.

Diagram: Teams and Notifications Algebras

The following diagram visually presents the algebraic structure of the Teams and Notifications components. It shows:

- The carrier set of each component - The set of operations - Arity - Closure properties

Component 1: Teams Algebra

Carrier Set

The carrier set for Teams is:

Team = { all team objects stored in Firestore }

Each element in this set contains fields such as teamId, teamName, members, captainId, createdAt, and ranking or statistics.

Operations

addMember(team, player)

Arity: binary Description: Adds a player to the team's members list. Closure: The output remains in Team. addMember(team, player) → Team

removeMember(team, player)

Arity: binary Description: Removes a player from the team's members list. Closure: The output remains in Team. removeMember(team, player) → Team

changeCaptain(team, player)

Arity: binary Description: Updates the captainId field of the team. Closure: The resulting structure remains part of Team.

updateTeamName(team, newName)

Arity: binary Description: Updates the teamName field. Closure: Output remains a Team element.

Benefits of This Algebra

Defining Teams as an algebra highlights important system invariants:

- A team must always have a valid captain. - All operations maintain a valid Team structure (closure). - No operation produces malformed or incomplete team objects.

This reduces ambiguity, prevents invalid team states, and clarifies the preconditions and postconditions needed for testing.

Component 2: Notifications Algebra

=== Carrier Set The carrier set for Notifications is:

Notification = { all notification objects delivered to users }

Each notification includes notificationId, recipientUserId, type, message, isRead, and timestamp.

Operations

markRead(notification)

Arity: unary Description: Sets isRead to true. Closure: The output remains an element of Notification.

updateMessage(notification, newMessage)

Arity: binary Description: Updates the message text of the notification. Closure: The output remains in Notification.

retarget(notification, newRecipient)

Arity: binary Description: Assigns the notification to a different user. Closure: Output stays inside Notification.

transformType(notification, newType)

Arity: binary Description: Changes the notification type. Closure: The resulting object remains a Notification.

===Benefits of This Algebra This algebraic approach reveals assumptions and prevents errors such as:

- Incorrect deletion or modification of notifications when marked as read
- Producing notifications without required fields
- Targeting invalid or unauthorized recipients

Algebra clarifies valid state transitions and enforces structural consistency.

Summary

By analyzing Teams and Notifications as algebras, the carrier sets, operations, arity, and closure rules become explicit. This method strengthens the requirements model, exposes hidden assumptions, clarifies subsystem behavior, and supports correctness and maintainability across the Esports Organizer system.

2.1.8*Database Algebras*

Events: For events in the esports organizer the following algebra can be used to describe the processes that are applied to them in the database.

(N U B, {addEventToDatabase: E → B, deleteEvent: EventID → B, isEventInDatabase: E → B, getEventFromFirestore: EventID → N})

Where: **N U B** : Is the set of elements on which operations can be applied to.

E : Represents the set of events.

N : Represents the set of events with the addition of a null element, which can be described as $N = E \cup \{none\}$.

B : Represents the set of Boolean values *{true, false}*.

addEventToDatabase : Takes an event object as a parameter, if it is not in the database, it is added to the database. Returns a Boolean to show if the event was added successfully or not; returns true if the event was added and false if the event could not be added.

deleteEvent : Takes an Event's ID, such as its name, and removes the corresponding event from the database if it was found. Returns true if the event was removed and false otherwise.

isEventInDatabase : Takes an Event and checks if an event in the database has the same ID/name as the provided event. Returns a Boolean to indicate if the event was found. **getEventFromFirestore** : takes an Event ID and searches in the database for an event that has a matching ID. Returns the found event and a null object if the event was not found.

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.
- Alex lives for the thrill of competition. Between coding labs and tutoring sessions, he sinks hours into ranked matches of CS:GO and League of Legends, tracking every stat and win rate like a data analyst studying performance trends. His drive to improve is relentless, he watches replays late at night, runs through strategy breakdowns, and celebrates even the smallest jumps in rank.
- **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 dreams of a platform that automates what he does manually: collecting scores, comparing results, and pinpointing weaknesses. He hungers for clear data, sleek dashboards, and a competitive community that keeps him sharp.~~

Persona 2: Maria the Organizer

- **Age:** 34
- **Background:** Works in event management, organizes local gaming tournaments on weekends.
- Maria thrives on order and energy. Her weekdays are filled with logistics meetings and community programs; her weekends revolve around bringing gamers together. She's the mastermind behind local tournaments from planning the brackets and registration forms to juggling the chaotic last-minute messages from participants.
- **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 wants an all-in-one event management platform that gives her time back. Something with simple event creation, instant notifications, and built-in communication tools to keep players updated. For her, efficiency equals stronger, more loyal communities.~~

Persona 3: Liam the Casual Gamer

- **Age:** 26
- **Background:** Plays games for fun after work, sometimes interested in casual competitions.
- After long days designing interfaces and color palettes, Liam logs on to disconnect. He hops into Marvel Rivals or LOL matches with friends, not for glory, but for laughs and connection. Still, he enjoys the occasional weekend competition when it's easy to join.
- **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 wants a smooth, minimalist system that curates nearby casual tournaments, lets him sign up in a few taps, and keeps things friendly. Convenience and community matter more to him than competition.~~

2.2.3 Domain Requirements

- The system-to-be shall require each event to be linked to an existing video game.
- The system-to-be shall require at least one registered user to be assigned as an organizer before an event can be created.
- The system-to-be shall require event modifications and/or cancellations to be performed exclusively by assigned event organizers.
- The system-to-be shall require all events to include a starting date and a location (physical or online URL).
- The system-to-be shall ensure that an event's starting date occurs after the date of creation.
- The system-to-be shall ensure that teams consist of at least one member before being allowed to register for an event.
- ~~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 require organizers to define a maximum number of participants (teams or players) for each event.
- The system-to-be shall ensure that the participant count of any given event must not exceed the participant limit of the event.
- The system-to-be shall ensure that a user may only register once for any given event.
- The system-to-be shall ensure that registration and properties for a specific event cannot be altered once it has concluded.
- The system-to-be shall update user and team participation records according to the results of finished events.
- The system-to-be shall prevent an event from being created without an organizer.
- The system-to-be must enforce unique team and event names within the system.
- ~~The system-to-be shall ensure results of each event are available to users for ranking and visualization.~~

2.2.4 Interface Requirements

- The system-to-be must reject any attempt by a user to create an event, team, or community with a duplicate name of an existing item within the database and provide an appropriate warning upon

creation attempt.

- The system-to-be must provide the means to be able to search for user created events, teams, and communities, filtering these by keywords that might be contained in the item's name.
- The system-to-be must provide a way for users to join teams and communities they are not a part of and leave teams and communities they are a part of.
- ~~The system-to-be must provide a means for users to join teams and communities of interest.~~
- The system-to-be must provide a way for the user to sign in/log into the website using an email and password.
- 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-to-be must save and display the user's public information, such as the username, bio, and profile picture, in the user profile and provide a way to edit these as the user desires.
- The system-to-be must track a user's participated events and favourite games and accurately display these in the user's profile.
- The system-to-be must provide a way for users to be able to change/reset their password within the application.
- 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.
- The system-to-be must provide a way for users with an admin role to send notifications within the application to other users.
- The system-to-be must provide a way for admin users to ban/unban unfit users from the platform.
- The system-to-be must provide a means for users to submit feedback to the application in order to be reviewed and considered to improve the application's design.
- ~~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-to-be must provide a way for users to create events, which must have a start date, a location, an organizer, the subject or game of the event, and an optional associated community.
- 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.
- The system-to-be must provide a form through which users can register for events to mark and record their participation in the event.
- The system-to-be must display the event's primary information, such as the title, date, capacity/number of participants that can attend, the subject game, and the location.
- The system-to-be must reject any attempts from users not in a team when attempting to join an event, providing a message that indicates that the user must be part of a team in order to join.

- The system-to-be must provide a means for users to edit their created event's properties, start date, location, and subject.
- 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-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 reject any attempts users make to register to an event past the event's start date.~~
- The system-to-be must provide a way for users to create a team, providing a team name, a primary game/subject, and optional properties such as a description and social media URLs.
- The system-to-be must provide a way for the team's organizer to edit the team's information, such as the description and subject.
- The system-to-be must provide a way for users to create a community, providing a community name, a primary game/subject, and optional properties such as a description and social media URLs.
- The system-to-be must provide a way for users to create posts for a community and be able to view other users' posts within the community.
- The system-to-be must display any available events related to the community in the community's feed section.

2.2.5 Machine Requirements

- 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-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-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-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 describes how the system realizes the domain phenomena introduced in Section 2.2. Instead of focusing on the "platform", this section explains how actions in the domain—such as updating a player's visibility, retrieving event participation, or assigning badges—become concrete behaviors implemented by software components.

During Milestone 2, the Profiles module evolved from an abstract domain concept ("players have identity, visibility, and participation history") into a working set of coordinated components. The goal is not just to implement features but to reflect the real behaviors observed in the gaming ecosystem the team studied.

A good way to understand this is to imagine walking with a friend to the university cafeteria while explaining what the system actually **does** when a player edits their profile. "Mira, cuando Liam updates his profile picture or hides his stats, that's not just a button on the screen. That action travels through several components—first the React UI, then the backend API, then Firestore. Each step mirrors a behavior we saw in the domain: identity, visibility, and participation. The software simply gives those behaviors a place to live."

The **software architecture** reflects how domain-level concepts transform into operational modules:

- Authentication ensures that only valid users can create, modify, or participate in events—mirroring the domain requirement for verified organizers and participants.
- User profiles are stored persistently, reflecting the domain notion of a "player identity" with attributes that others may see or that the player may choose to hide.
- The Profiles module (implemented using Node.js / Express) acts as the operational expression of domain actions such as "modify profile", "request participation history", or "toggle visibility".
- The React client is the medium through which players perform these behaviors. When the player interacts with the UI, they are essentially causing a domain action to move through modules until the system fulfills it.
- Badge and Privacy Control modules correspond to phenomena such as reputation, privacy preferences, and player-to-player visibility, which emerged naturally during domain exploration.

In this context, the architecture diagram is not meant to be a strict engineering schematic, but rather a conceptual sketch of how domain behaviors travel through the system. To avoid confusion (as noted in the professor's feedback), each type of element in the diagram corresponds to a specific category:

Rectangles: Functional modules that carry out domain operations (Profiles, Badge Module, Privacy Control). **External-service rectangles:** Components outside our system (Firebase Auth, Firestore) that support authentication or persistence. **Solid arrows:** Flows that represent domain actions being executed—such as storing updated profile data, reading badges, or toggling visibility.

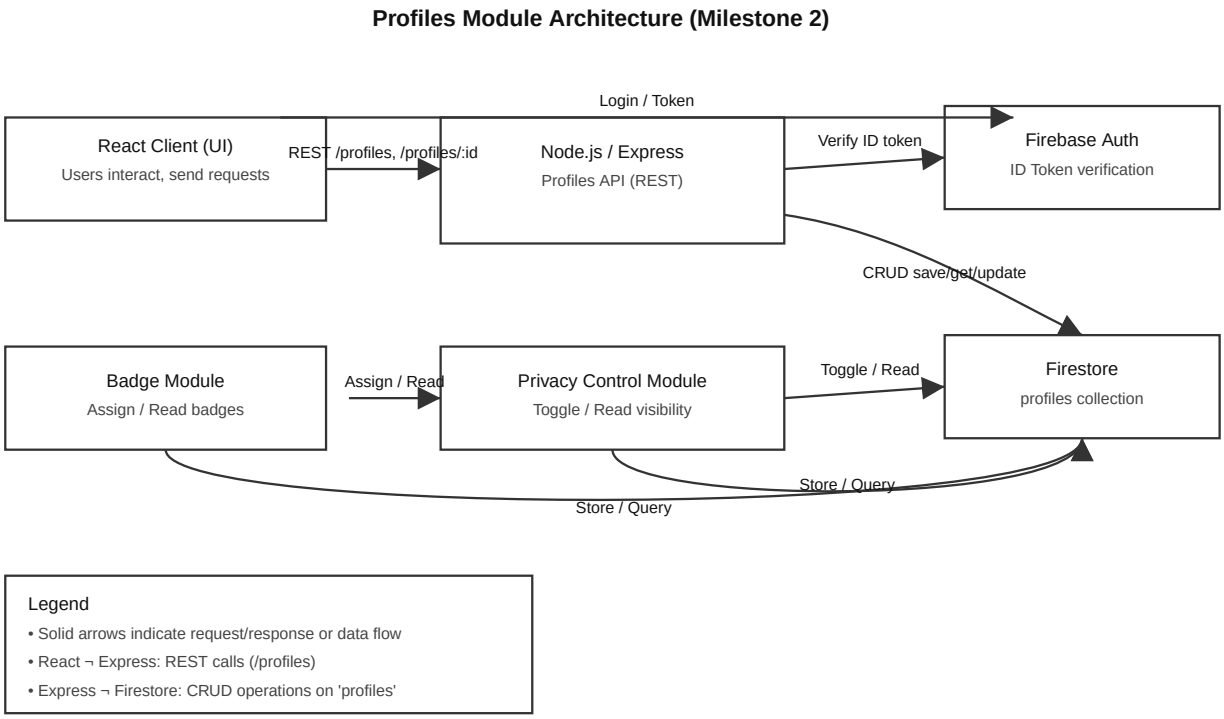


Figure 2.3 - Profiles Module Architecture (Milestone 2)

Thinking again in “cafetería terms”: When Alex wants to check his past tournaments, the system behaves like a small conversation happening behind the scenes. React asks Express, Express checks Firestore, Firestore returns the data, and Express replies. This small chain of interactions corresponds directly to the domain’s requirement to make event results available even after tournaments conclude.

Realization of Domain Behaviors

The **software design** explains how each phenomenon in the domain becomes a concrete operation:

- **Authentication** Reflects the domain restriction that only registered users can create or modify events or profiles.
- **Profile Management** Represents the player entity described in the domain. Updating or viewing a profile is not merely a UI action—it reflects the domain behavior of identifying players, referencing them in events, and ensuring data consistency.
- **Participation Retrieval** Supports the requirement that results and participation remain visible and traceable, enabling rankings, statistics, and post-event visibility.
- **Badge & Visibility Controls** Express aspects of player behavior such as reputation, privacy, or recognition available in real gaming communities, which surfaced during persona and field exploration.

2.3.1 Selected Fragments of the Implementation

Selected fragments appear only to clarify how domain behaviors become implemented actions. They are not included merely for displaying code but for illustrating how the system operationalizes the phenomena discussed in Section 2.2.

The following fragment corresponds to the behavior “view a player’s past participation”, which aligns with the domain requirement ensuring that event results and participation remain accessible for ranking and visualization.

```
async function getParticipatedTournaments() {
    const querySnapshot = await getDocs(collection(db, "User"));

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

    return User;
}
```

Seen from a domain perspective, this function is not “just fetching documents”. It represents the idea that players leave traces of participation behind them, and the system must preserve that history so it can be referenced by rankings, organizers, or other players.

The next code selection reflects domain actions related to players modifying their identity, adjusting visibility, or updating personal attributes—real behaviors that emerged clearly during persona and scenario analysis.

```
export async function saveProfile(profileData) { ... }
export async function getProfile(email) { ... }
export async function updateProfile(email, updates) { ... }
```

These operations embody the concept that “a player in the domain has persistent identity data that evolves over time”. Each function ensures integrity via timestamps, validating that the system remains consistent even as players make changes across tournaments or communities.

This alignment between domain concepts and implementation ensures that the system grows naturally from the world it observes, rather than forcing features into an artificial structure.

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.

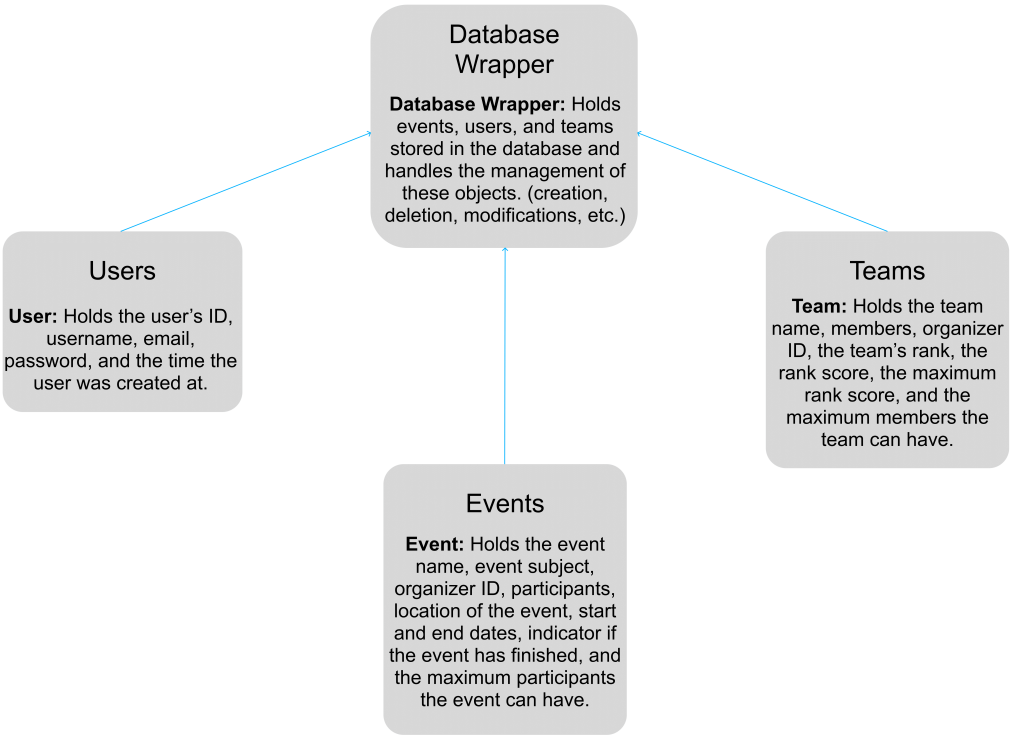


Figure 3.1 - Data architecture

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 Risk Analysis

3.2.1 Narrative Assessment of Organizational Risks in Tournament Management

Introduction

Tournament management within the **Esports Organizer** project involves complex coordination among teams, organizers, schedules, communications, and event policies. While technical risks (bugs, system crashes, data loss) are part of software engineering, this analysis focuses specifically on **non-technical organizational risks** the human, procedural, and managerial factors that threaten smooth tournament operations.

Following the Risk Analysis lecture topic, this document identifies domain-relevant risks, classifies them, evaluates likelihood and impact qualitatively, and proposes managerial mitigation strategies. The assessment is grounded in our project's structure and responsibilities as described in the course specification.

Risk Identification and Classification

Risk 1 - Team No-Shows or Late Arrivals

Category: Organizational / User-Behavior **Description:** Teams frequently arrive late or fail to show for scheduled matches, delaying the bracket and forcing last-minute adjustments.

Likelihood: Moderate **Impact:** High (significant schedule disruption)

Mitigation Strategies:

- Mandatory pre-match check-in window (e.g., 10 minutes prior).
- Automated reminders to team captains.
- Clear penalty structure for repeated no-shows.

Residual Risk: Real-life disruptions may still cause unavoidable absences. Brackets must include buffer slots.

Risk 2 - Scheduling Ambiguity and Overlaps

Category: Organizational / Operational

Description: Two matches may be assigned to the same team simultaneously, or teams may not receive schedule updates in time.

Likelihood: Moderate **Impact:** Severe (cascading delays, bracket instability)

Mitigation Strategies:

- Centralized real-time schedule visible to all participants.
- Manager-level verification before approving schedule adjustments.
- Built-in buffer time between rounds.

Residual Risk: Delays may still occur if matches exceed expected duration.#

Risk 3 - Miscommunication Between Organizers and Participants

[.hl-green]#**Category:** Organizational / User-Behavior

Description: Rules, bracket formats, or procedures may be misunderstood due to inconsistent or unclear communication.

Likelihood: High **Impact:** Moderate (disputes, delays, repeated clarifications)

Mitigation Strategies:

- Single official communication channel (Tournament Dashboard).
- Rules overview and FAQ sections visible during registration.
- Consistent announcements enforced by managers.

Residual Risk: Some users may still ignore updates or fail to read instructions.#

Risk 4 - Administrative Overload and Decision Bottlenecks

[.hl-green]#**Category:** Organizational

Description: Organizers may face simultaneous inquiries, appeals, and administrative tasks during peak periods.

Likelihood: High **Impact:** High (slower decisions, increased errors)

Mitigation Strategies:

- Clear role assignment: appeals, check-ins, rule enforcement.
- Escalation paths involving team leads and managers.
- Standardized decision rubrics.

Residual Risk: Peak-time volume may exceed staff capacity in large tournaments.

Qualitative Summary Table

Risk	Category	Likelihood	Impact
Team no-shows	Organizational / User	Moderate	High
Scheduling ambiguity	Operational	Moderate	Severe
Miscommunication	Organizational	High	Moderate
Administrative overload	Organizational	High	High

Mitigation Philosophy

The Risk Analysis lecture emphasizes **early identification, classification, and mitigation**. In tournament management this means:

- establishing communication systems that prevent ambiguity,
- improving organizational processes to minimize bottlenecks,
- designing escalation policies to maintain fairness,
- planning buffers and fallback procedures for inevitable disruptions.

This document directly applies the lecture topic by converting theoretical principles into practical risks and controls specific to the Esports Organizer project.#

Residual Risk Discussion

Even with strong mitigation strategies, tournament environments retain inherent unpredictability. Residual risks include:

- last-minute team absences,
- communication oversights,
- workload spikes during peak hours,
- disputes arising from edge cases not covered in documentation,

- user behavior influenced by competitive pressure.

Residual risk requires ongoing monitoring, real-time adjustment, and post-event retrospectives.

Conclusion

Non-technical risks in tournament management are often more disruptive than technical ones due to their human and procedural nature. By identifying, classifying, and mitigating these risks and aligning with the organizational roles and communication expectations defined for the project, this analysis strengthens the reliability and fairness of the Esports Organizer's tournament module.

3.2.2 Risk Tree and Analysis for Team Creation

Introduction

This section presents a risk analysis for the **Team Creation** subsystem within the Esports Organizer Manager platform. Team creation is essential for organizing group-based tournaments, assigning captains, managing rosters, and ensuring the integrity of competitive structures. Failures in this subsystem can lead to inconsistent teams, broken tournament logic, or players being assigned to invalid or duplicated team states.

This risk analysis decomposes potential failures into structured branches and identifies where inconsistencies, validation errors, backend–frontend mismatches, or documentation gaps might corrupt the team creation process.

Purpose of the Analysis

The purpose of this analysis is to:

- Identify the **top event** leading to team creation failure.
- Break it down into clear risk branches using a fault-tree model.
- Connect the failure modes to inconsistencies across documentation, API behavior, UX, and data models.
- Provide mitigation strategies and non-functional considerations.
- Support future formal verification or consistency-checking tasks.

This analysis focuses on **how a valid team fails to be created or becomes inconsistent**, not the general management of teams.

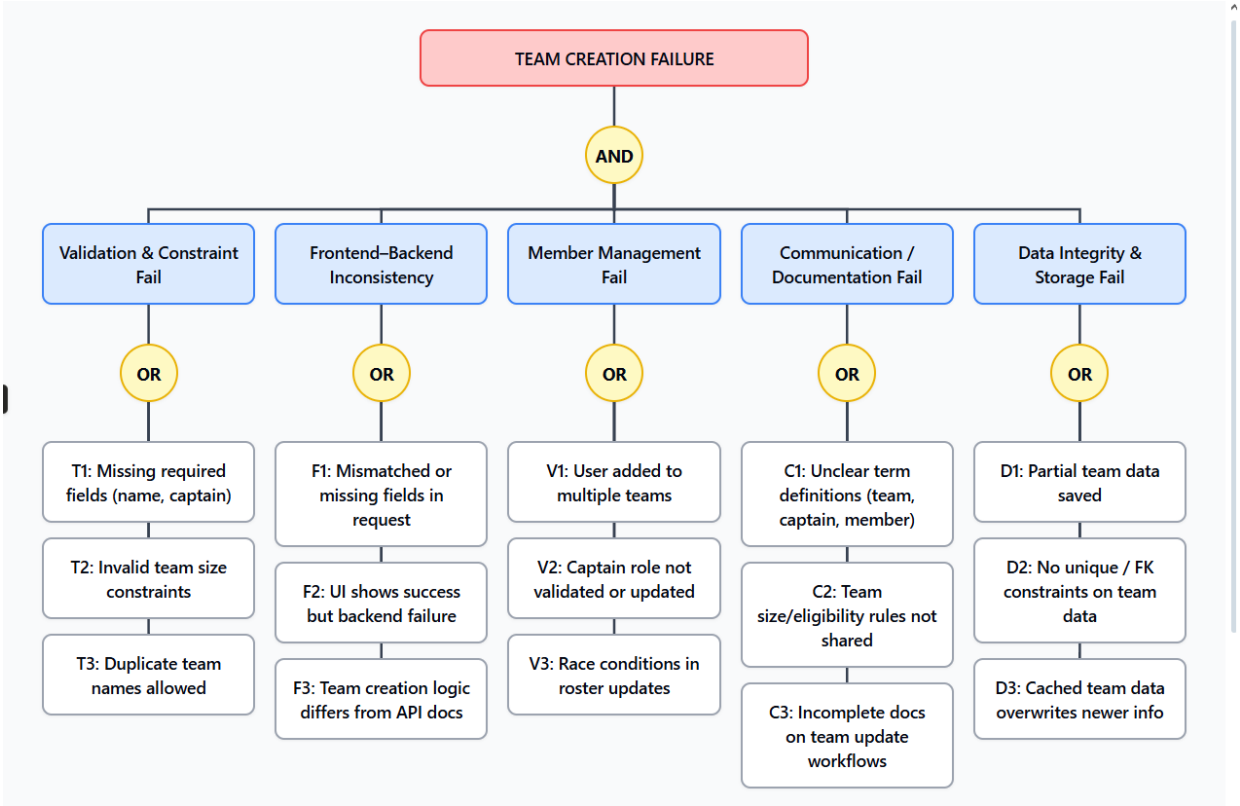
Top Event: Team Creation Failure

 Team Creation Failure A failure occurs when the system creates an invalid team, fails to create a

team, or produces inconsistent data that breaks tournament logic.

Examples include: * Team created without a captain * Duplicate team names * A user assigned to multiple teams simultaneously * Team created with missing or corrupted fields * UI indicates successful team creation but backend fails * Conflicting team size limits across modules

Risk Tree Diagram



Risk Tree Structure and Branches

The following sections detail the major risk branches contributing to Team Creation Failure.

1. Validation & Constraint Risks (T1–T3)

Risk ID	Description	Example Consequence
T1	Team created without required fields	Incomplete or unusable team records
T2	Team size validation missing or incorrect	Teams with too many or too few members
T3	Duplicate team names not prevented	Conflicting team identities in tournaments

Mitigation strategies: * Enforce server-side validation of all required fields. * Apply unique constraints on team names. * Add UI-level validation to prevent incomplete form submissions.

2. Frontend–Backend Inconsistency Risks (F1–F3)

Risk ID	Description	Example Consequence
F1	Mismatched field names or missing request data	Backend receives incomplete team data
F2	UI indicates success but backend rejects creation	Users believe team exists when it does not
F3	Team creation logic differs between frontend and API docs	Conflicting expectations when creating teams

Mitigation strategies: * Synchronize API schemas through OpenAPI. * Require documentation updates for all team-related changes. * Add design-time and runtime request/response validation.

3. Member Management Risks (V1–V3)

Risk ID	Description	Example Consequence
V1	User added to multiple teams simultaneously	Invalid tournament participation
V2	Captain role not properly updated or validated	Teams created without leadership structure
V3	Member removal/addition race conditions	Corrupted or duplicated roster entries

Mitigation strategies: * Enforce membership exclusivity rules. * Validate captain assignment at both UI and backend layers. * Use database transactions when updating rosters.

4. Communication & Documentation Risks (C1–C3)

Risk ID	Description	Example Consequence
C1	Unclear definitions of “team,” “captain,” or “member”	Incorrect assumptions by developers
C2	Team size or eligibility rules not documented	Inconsistent enforcement across modules
C3	Incomplete documentation on team update workflows	Incorrect implementation of roster operations

Mitigation strategies: * Maintain a glossary for all team-related terminology. * Document constraints (size, eligibility, roles) in the wiki. * Track inconsistencies in the `#inconsistencies_channel`.

5. Data Integrity & Storage Risks (D1–D3)

Risk ID	Description	Example Consequence
D1	Partial team data saved due to interrupted request	Broken team records
D2	No unique constraints on team IDs or captain IDs	Duplicate or corrupted relationships
D3	Outdated or cached team data overwritten	Loss of valid roster info

Mitigation strategies: * Ensure atomic DB operations for team creation and updates. * Add unique and foreign-key constraints on team and user tables. * Use versioned updates or optimistic locking.

Link to Lecture Topic Task: Identifying Inconsistencies

Many risks originate from inconsistencies such as:

- Terminological clashes: C1
- Structural mismatches: F1, D1
- Behavioral inconsistencies: F2, V1
- Process/documentation issues: C2, C3

This risk analysis supports the Identifying Inconsistencies task by showing exactly how mismatches propagate into invalid team states.

Conclusion

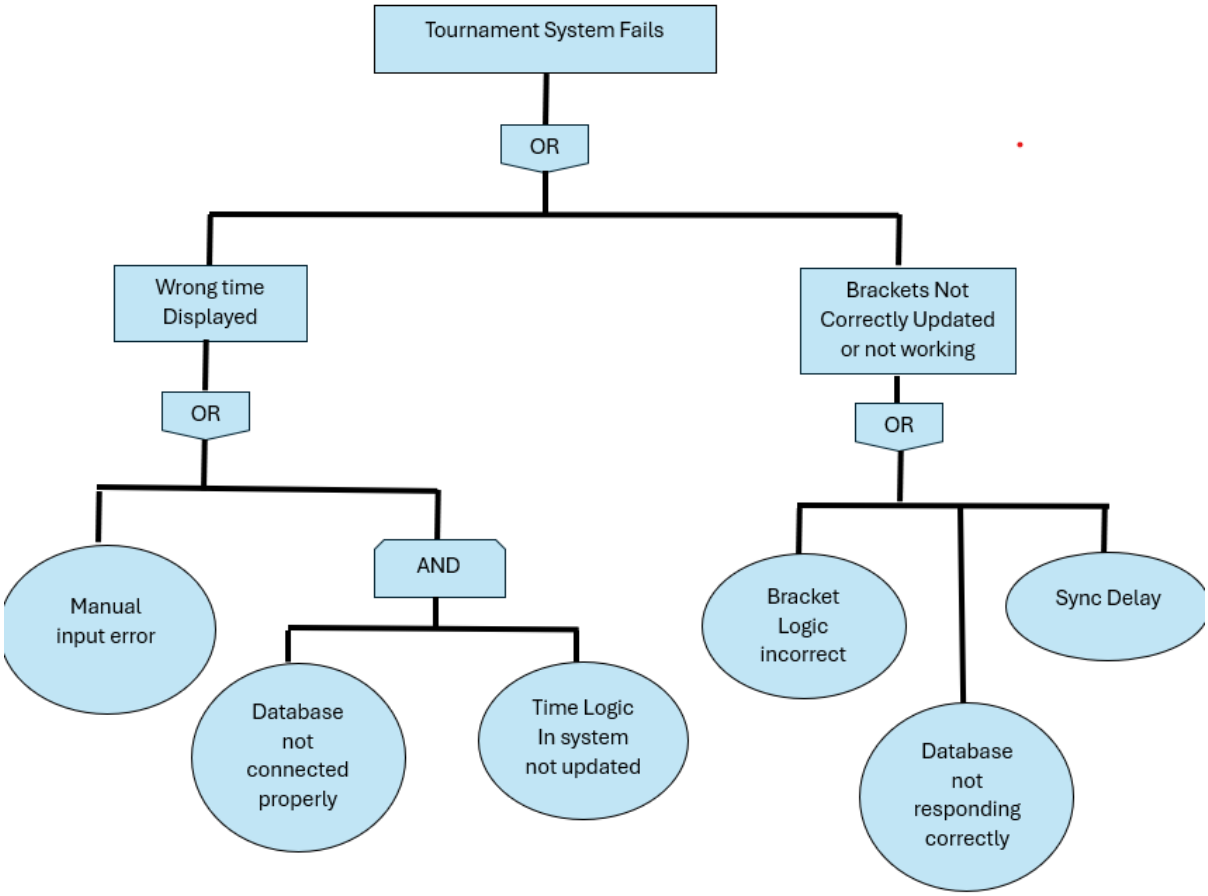
This risk analysis shows how **Team Creation Failure** can arise from validation errors, inconsistent cross-component communication, incorrect member management, or weak data integrity rules. By identifying these risks and applying mitigation strategies, the Esports Organizer Manager improves the reliability of its team management and ensures consistent tournament operations.

3.2.3 Risk Analysis Tree for Tournaments

Introduction

Risk Trees help identify the potential errors or failure points a program might encounter. By breaking down risks into smaller, traceable causes, they allow testers to design more targeted and efficient test cases. This structured view not only improves test coverage but also helps developers anticipate where

bugs, crashes, or unexpected behaviors are most likely to occur during execution.

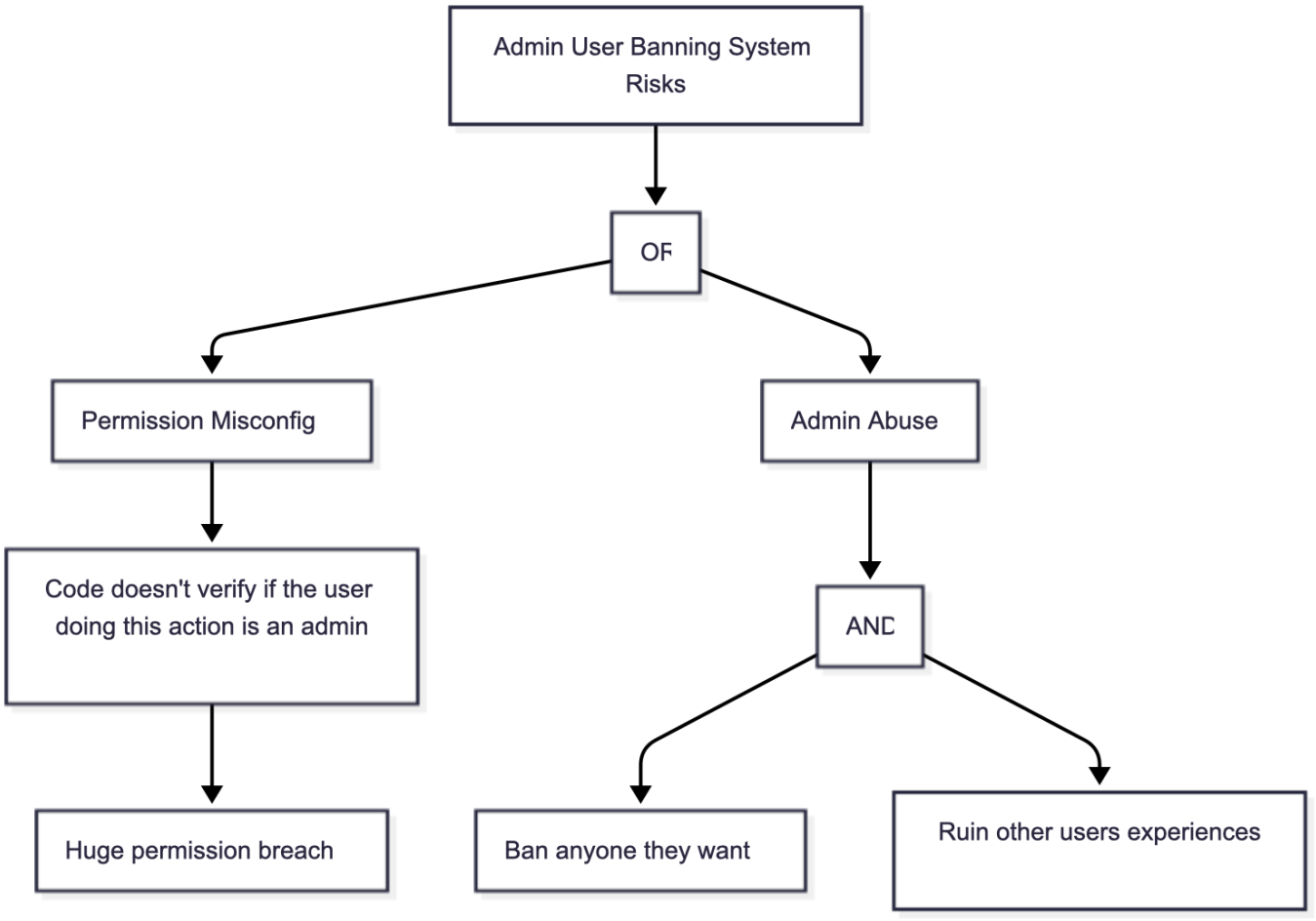


Tournament Risk Analysis Tree

Analysis

Tournaments system can fail in some ways that could be fatal to the product. The first thing that could disrupt the process of tournaments would be the Time display not being correct. This could cause players to miss matches or lose time. This could happen because of a manual input error caused by the manager, but it could also be because the time logic in the system is not updated and database not provide the correct information for the program. Another issue that could interfere with a tournament would be Brackets not updating or failing. This could be caused by Incorrect bracket logic which would mean that brackets do not show up how they are supposed to. There could be a Sync delay within the tournaments causing brackets to show incorrect or incomplete information to a user. Database not responding correctly could also damage the integrity of the brackets since nothing would appear to a user causing a disruption to the process of tournaments in the program. Overall, these risks emphasize the importance of reliable database connectivity and well-designed system logic. By addressing these potential failures, developers and testers can ensure that tournaments run smoothly and maintain their integrity.

3.2.4 Risk Tree and Analysis for Admin Banning User Permission



Admin Banning Permission Risk Tree

This risk tree outlines the potential risks associated with granting the Admin role the permission to ban users. At the top the risk of such permission to an admin can have the risk of 2 things. It could take a permission misconfiguration route on where if not coded well this could lead to the action or permission of banning a user could be used by anyone since it never verified well if the user doing this action is really an admin which lead to a huge permission breach where anyone can just ban whoever they want. The other route is that even if the permission is well coded and verified, there is still a big risk that a rogue admin can be power hungry and ban users for no reason and because they wanted to at any time or any place, which this leads to user dissatisfaction due to the unfair banning of users because nobody likes to be banned for no reason.

To avoid such risks, it is important to implement strict permission checks in the code to ensure that only users with the Admin role can execute the ban user action. Additionally, having some sort alternative to trust users that want to be admins and giving them the role only if they are trusted enough to not abuse their power can help mitigate the risk of rogue admins banning users unfairly. == 3.3 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)

3.4 Checking relevant properties through TLA+

3.4.1 TLA+ Model for Core Database Operations

Acceptance Criteria

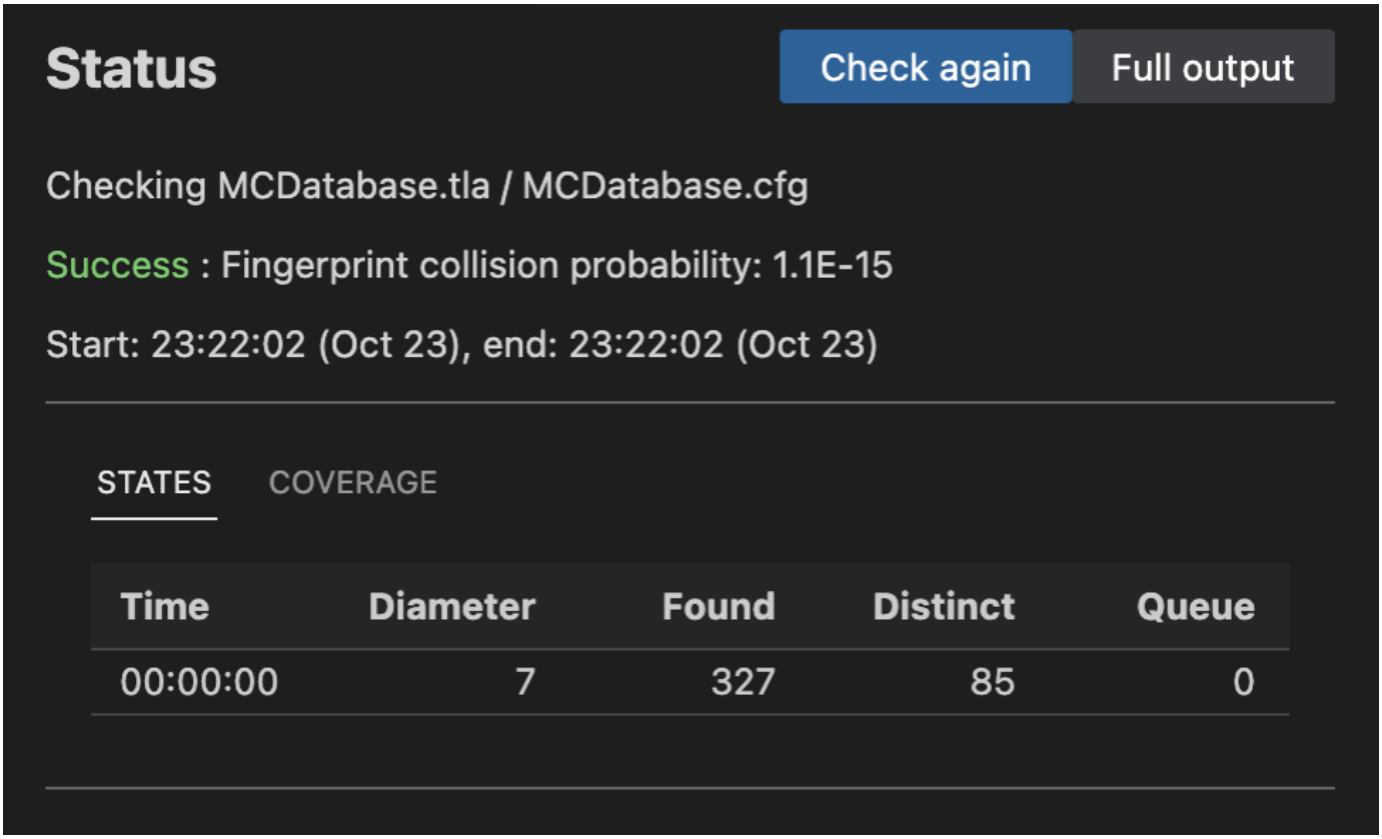
- TLA+ model checking successfully generated 327 states with 85 distinct states found. It managed complete state space exploration within defined constraints, and no safety property violations were detected. Execution was also completed in under a second. Insight into core operations is detailed below.

Findings Report

- After establishing basic functions like `CreateEvent`, `RemoveParticipant`, and `AddParticipant`, I moved onto `ToggleEventFinished`. The logic behind this function was to create an invariant that does not allow events to be toggled as finished if there are active participants in that event. When I went to revise the database logic, I realized there is no specific implementation for event lifecycle management.
- In `Events.js`, the current boolean property `finished` currently acts only as a placeholder for event status. There is no specific method designed for safe event status toggling.

Solution

- The Event class in the database code lacks a method to safely toggle the finished status with proper validation. It includes methods like addParticipant() and removeParticipant(), but no method to safely toggle the finished status while checking if participants exist.
- Implement a method to the event class to safely toggle event status while ensuring there are no active users within it.
- Doing so would prevent both incomplete event management and poor user experience.



TLA+ Model Checking Results

3.4.2 TLA+ Model for Dynamic Match Progression

Overview

TLA+ specification for analyzing tournament bracket progression logic based on the implementations from `MatchProgression.js`, `ResultReport.js`, and `BracketsTournamentPage.jsx`.

Objectives

- Model single-elimination tournament bracket progression
- Verify correct winner selection and elimination tracking
- Detect potential flaws in round advancement logic
- Ensure valid tournament termination properties

Model Configuration

Configuration for 8-team tournament

```
---- MODULE DynamicMatchProgression.cfg ----
SPECIFICATION DynamicMatchProgression

CONSTANTS
    NumTeams = 8
    Teams = 1..8

INVARIANTS
    TypeInvariant
    NoDuplicateEliminations
    EveryTeamEliminatedOrChampion

PROPERTIES
    TournamentTerminates
    NoDeadlock

(* Model checking constraints *)
CHECK_DEADLOCK FALSE
====
```

Key Properties Verified

Safety Properties

- **TypeInvariant**: All tournament state values maintain correct types and ranges
- **NoDuplicateEliminations**: No team appears more than once in the elimination order
- **EveryTeamEliminatedOrChampion**: When a tournament completes, exactly N-1 teams are eliminated and 1 is champion

Liveness Properties

- **TournamentTerminates**: Tournament eventually completes with a champion
- **NoDeadlock**: System never gets stuck in non-terminal state

Analysis Results

Table 1. Possible Flaws Detected

Property	Status	Description
Match double-counting	Safe	Model confirms matches cannot be double-counted
Incomplete round progression	Safe	Rounds only advance when all matches complete
Elimination order consistency	Safe	Elimination order correctly tracks losers
Tournament termination	Safe	Tournament always completes with a single champion

Mapping to the JavaScript Implementation

The following JavaScript behaviors are modeled:

- `initializeBracket()` → `InitializeFirstRound()`
- `progressMatch()` → `ProgressMatch()`
- Round advancement logic → `AdvanceRound()`
- Elimination order tracking → `updatedEliminationOrder`
- Tournament completion → `DeclareChampion`

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.

Agile is a software development approach which prioritized delivering working software in small, frequent iterations while adapting to feedback such as changing system requirement or user

necessities. Agile encourages continuous collaboration, constant improvement and incremental delivery. Our team follows the scrum methodology, which structures development into bi-weekly cycles called sprints. After each sprint it is intended to have a potentially shippable increment of the project.

Team Roles

- Managers: Ensures the scrum process is followed and manages the whole team.
- Team Leaders: They coordinate weekly meetings, verify and approve issues before the managers view them and are up to date with the achievements of every member on their team.
- Development Teams: Database and Backend, events and notifications, Profiles, social features, and UI teams in charge of developing the frontend and backend tasks.

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.

To make an informed decision, both options were compared using Non-Functional Requirements (NFRs). The qualitative attributes of a system rather than its specific functions.

The chosen criteria were:

Speed of Communication - How fast users can exchange information.

Ease of Use - How intuitive and accessible the platform is for all team members.

Information Organization - How effectively messages and files can be structured and retrieved later.

Scalability - How easily the platform can handle team growth in members, channels, and activities.

Process Modeling

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

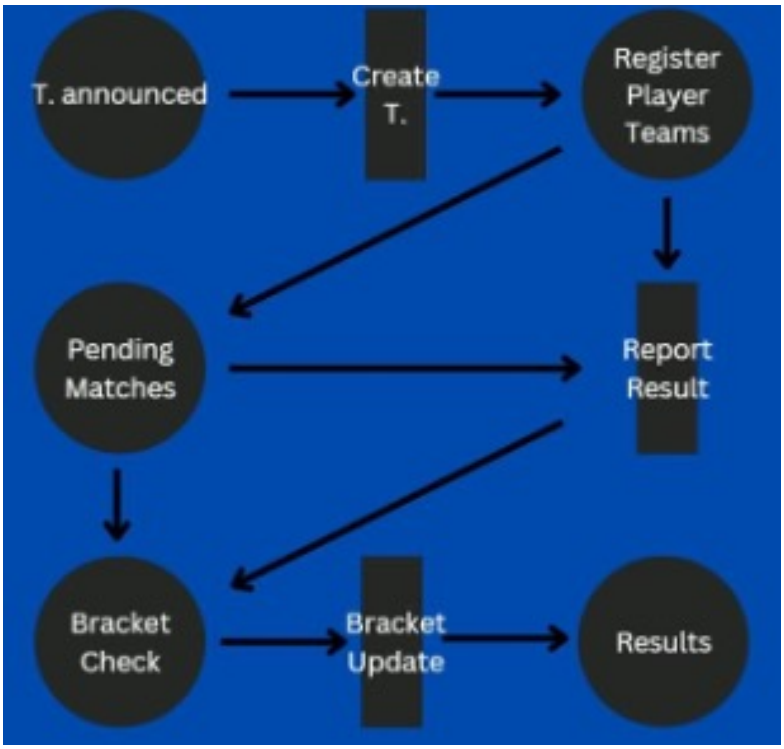
~~Petri Nets help visualize how different backend functions interact dynamically within the tournament system.~~

~~In our project, Petri Nets work as graphical representations of workflows inside the backend system,~~

~~for tournament creation, registration, and progression. Each place represents a specific system state, while each transition models a backend function or event triggered by user or system actions.~~

Petri nets within our project were primarily implemented as guides and visual representations of how different classes are expected to operate and collaborate when needed. They serve as graphical descriptions of what a process should look like, considering all the possible (at least currently known) paths that lead to a successful and complete “circuit.” The goal is to ensure that anyone, even someone outside the project can understand how the workflow described by the Petri Net functions at a theoretical level.

Let us consider, for example, the tournament Petri Net previously created by the database team developers:




```
Tournament Workflow Petri Net
```

We follow its path: circles represent **events**, rectangles represent **actions**, and arrows indicate the **workflow**.

The flow is as follows:

- Tournament is announced
- Organizer creates the tournament
- Player/Team registration process takes place
- Classes handle bracket generation, match creation, and tournament logistics

- The system tracks registration results and how brackets and matches are formed
- Each time a match occurs, brackets update accordingly
- At the end, general results are produced

This Petri Net is oriented toward the general “big picture” of tournament logistics as we understand them, data-focused, high-level, and representative of how tournament information flows through the system.

Across milestones, additional Petri nets similar to this one have been developed to track and clarify other workflows throughout the project.

Real-Time System Design

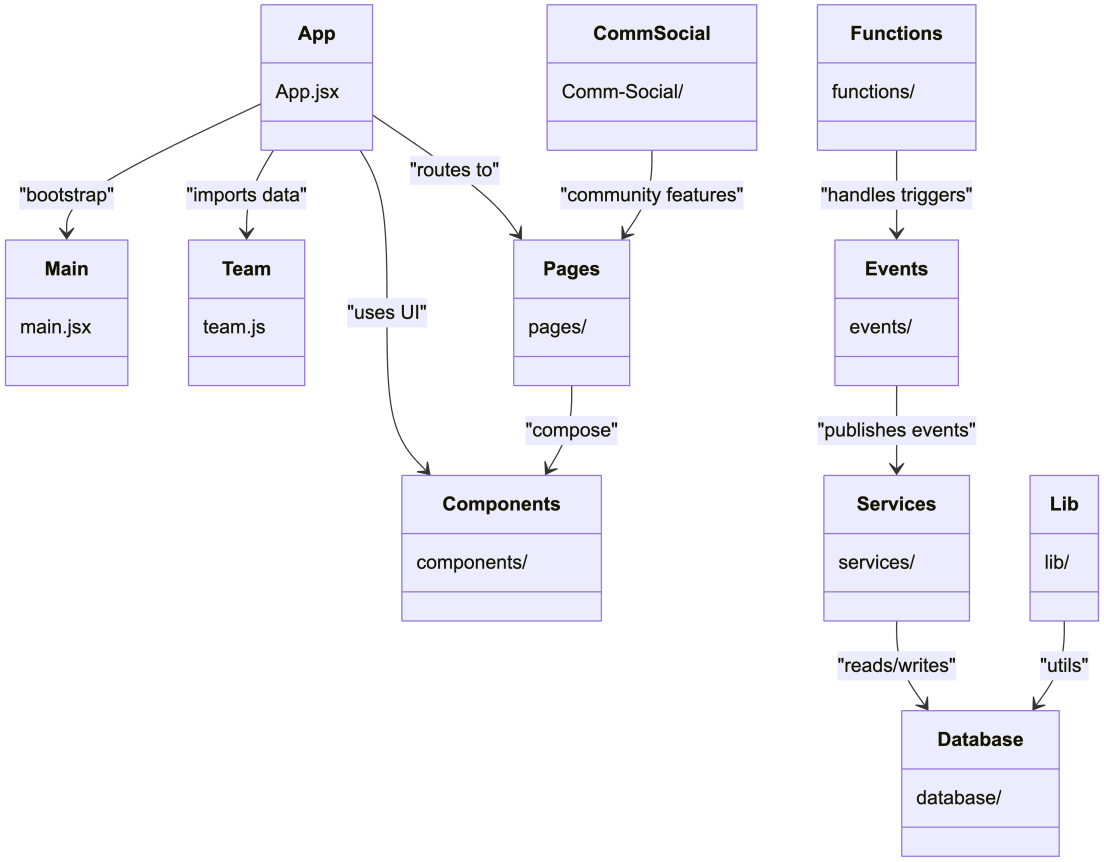
The implementation uses Firebase’s real-time capabilities with React state management for live bracket updates. Cloud Firestore’s `onSnapshot()` listeners provide immediate data synchronization, while React’s `useEffect()` and `useState()` hooks manage component state updates. This enables automatic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

The real-time updating feature ensures that tournament brackets in the eSports web app stay synchronized across all users without requiring manual refreshes.

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



App bootstraps Main and pulls in domain models (Team, Events), UI (Components, Pages, CommSocial), logic (Services, Functions), and 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.

Generative AI Use Disclosure Statement

Our team used AI tools thoughtfully and intentionally throughout the Esports Organizer project. The primary tools used were **GitHub Copilot** (free and Pro versions), **DeepSeek** and **ChatGPT**. Copilot Pro supported faster responses, expanded model options, and unlimited usage, while DeepSeek and **ChatGPT** assisted with conceptual explanations and clarification of complex diagrams and documentation.

Stages of AI Use

AI supported multiple stages of the project:

- **Brainstorming & Problem-Solving:** Generating alternative architectural approaches and breaking down complex tasks, particularly for the **tournament bracket page** and **team visibility controls**.
- **Debugging & Optimization:** Identifying error sources and proposing efficiency improvements for

Firestore operations used in **event data retrieval** and **user authentication workflows**.

- **Code Refinement:** Improving readability, reducing redundancy, and suggesting more maintainable patterns across front-end components.
- **Responsive Design Support:** Troubleshooting alignment and layout issues on the **dashboard**, **profile pages**, and **tournament pages**, including CSS and media queries.
- **Conceptual Understanding:** Clarifying difficult concepts such as Petri nets, TLA+, state charts, and sequence diagrams to support high-level design decisions.

Division of Work

Team members handled all core implementation, architectural decisions, business logic, Firestore integration, and testing. AI-assisted suggestions were reviewed, adapted, or rewritten before integration. AI-generated starting points, such as early bracket logic structures, CSS recommendations, or sample Firestore queries were significantly modified to match the project's requirements.

Verification & Fact-Checking

All AI-assisted code was:

- tested locally,
- validated against official Firestore documentation,
- cross-referenced with course materials when conceptual explanations were involved.

A notable example occurred when AI suggested a forEach-based filtering approach for Firestore queries. After reviewing documentation, the team replaced it with the more efficient and correct `.where()` pattern.

Learning Impact

AI tools improved understanding of complex concepts, accelerated debugging, and revealed more efficient design patterns. This balance of AI assistance and intentional team verification helped deliver a high-quality product while maintaining learning, accountability, and ownership over all final design decisions.

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.
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.

Section Name	Member	Added or Modified	Description
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.
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.

Section Name	Member	Added or Modified	Description
Derived Goals	Jayden Sánchez	Modified	Revised the Derived Goals section to clarify their independence from the main objectives and strengthen the section's purpose.
Application of Topics	Abby Gotay Almonte	Added	Wrote Application of Topics section, taking into account multiple LTTs.
Descriptive Section 2.1.1 - 2.1.6	Héctor Rivera	Modified	Refactored the entire Domain Description subsection to align with professor feedback. Integrated life-like Rough Sketch statements, corrected domain terminology, restructured Events/Actions/Behaviors, and reorganized function signatures under corresponding actions. Ensured all content reflects real-world domain phenomena without referencing system behavior.
Descriptive Section 2.2.3 - 2.2.4	Andres Cruz Zapata	Added/Modified	Added several essential requirements missing from the previous version, removed outdated requirements, and reworded some existing requirements to improve clarity.

Section Name	Member	Added or Modified	Description
Derived Goals	Jayden Sánchez	Modified	Finalizes subsection with refined content, clear goal independence, and required updates to improve clarity and structure.
Introduction Section	Jayden Sánchez	Modified	Reviewed and updated the presentation page, table of contents, logbook legend, and introduction, refining formatting and adding all necessary updates for Milestone 3.
Informative Section 1.1	Yamilet Gómez	Modified	Updated team structure to include the newly formed union between two teams for the final phase of the project. Adjusted Section 1.1 accordingly to reflect the revised organization of members and responsibilities.