HÖHERE TECHNISCHE
BUNDESLEHRANSTALT WIEN 16
Abteilung für Informationstechnologie

HTL
WIEN
WEST

DIPLOMARBEIT

IntelliQ

Ausgeführt im Schuljahr 2024/25

Ricky Raveanu (5BHITM)	Betreuer: Niel Widy BA MA
Nippon Lama (5BHITM)	Betreuer: Niel Widy BA MA
Nikola Petrovic (5BHITM)	Betreuer: Niel Widy BA MA

Selbständigkeitserklärung

Wir erklären, dass wir die vorliegende Diplomarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht haben.

Wien, am 11.04.2025

[redacted]

Ricky Raveanu	Nippon Lama	Nikola Petrovic

Dokumentation der Diplomarbeit

Verfasser	Nippon Lama, Nikola Petrovic, Ricky Raveanu
Jahrgang	5BHITM 2024/25
Thema	IntelliQ is an AI-powered quiz platform
Kooperationspartner	

Aufgabenstellung		
Realisierung		
Ergebnisse		
Teilname an Wettbewerben		
Möglichkeiten der Einsichtnahme der Arbeit		
Abgabevermerk	Datum:	Übernommen von:
Approbation	Datum:	Prüfer:
	Datum:	Abteiluntsvorstand:

Kurzfassung

Das Thema dieses Projekts ist die Entwicklung einer KI-gestützten Quiz-Plattform namens IntelliQ, die das Lernen verbessern soll. Die zentrale Frage lautet: Wie kann künstliche Intelligenz genutzt werden, um dynamische, personalisierte und interaktive Quizze zu erstellen, ohne dabei Datenschutz und Skalierbarkeit zu vernachlässigen? Das erkannte Problem war, dass es kaum Plattformen gibt, die auf den Nutzer zugeschnittene Quiz-Inhalte mit stabiler Echtzeit-Funktionalität bieten.

Das Projekt entstand aus einer Aufgabe, bei der eine innovative Quiz-Plattform entwickelt werden sollte. Das Team wollte frühere Erfahrungen mit der OpenAI API nutzen, um dynamisch generierte Quizze mit KI zu ermöglichen. Die ersten Versionen von IntelliQ waren jedoch noch sehr einfach gehalten. Es fehlten wichtige Funktionen wie Mehrspieler-Modus, effiziente Serverstruktur, Möglichkeiten zur Personalisierung und vor allem die Einhaltung der DSGVO. Auch Skalierbarkeit und Reaktionsgeschwindigkeit mussten stark verbessert werden, um die Plattform breiter nutzbar zu machen.

Das Ziel ist es, eine interaktive und personalisierte Quiz-Plattform zu entwickeln – für Lehrende und Lernende, die ein flexibles und motivierendes Werkzeug zum Lernen und Prüfen brauchen. Wichtige Funktionen sind:

- KI-generierte Quizze, angepasst an Interessen der Nutzer
- Mehrspieler-Funktion für gemeinsames Quizzen in Echtzeit
- Verbesserte Skalierbarkeit im Backend mit Hono und Supabase
- Extras wie eigene Avatare, Quizze aus Dokumenten oder per Zufall, und Lesezeichen-Funktion

Die Umsetzung basiert auf soliden und praktischen Grundlagen. Das Backend wurde von Express.js auf Hono umgestellt. Supabase wurde integriert, um Echtzeit-Datenbankfunktionen und sichere Anmeldung zu ermöglichen. Die Entwicklung erfolgte nach dem agilen Scrum-Modell, was eine gute Aufgabenverteilung und Zusammenarbeit im Team ermöglichte. Für einfache Updates und Deployments wurden CI/CD-Pipelines eingeführt. Beim Design kamen moderne Features wie Quiz-Sharing über OpenGraph-Tags, akustisches Feedback und Quizze aus hochgeladenen Dokumenten dazu. Das UI wurde mit ShadCN verbessert – für bessere Bedienung und Barrierefreiheit.

Das Projekt konnte erfolgreich den Mehrspieler-Modus in Echtzeit mit Raumcodes umsetzen, wodurch gemeinsames Quizzen möglich wurde. Nutzer können jetzt Quizze zu zufälligen Themen oder aus hochgeladenen Dokumenten generieren. Das Nutzererlebnis wurde durch Avatare und anpassbare Designs verbessert.Zudem wird der Datenschutz nach DSGVO eingehalten. IntelliQ ist nun eine vielseitige, interaktive und skalierbare Lernplattform. Das Projekt ist bereit für Wettbewerbe und zeigt technische sowie pädagogische Innovation.

Abstract

The topic of this project is the development of an AI-powered quiz platform, IntelliQ, designed to enhance the learning experience. The key question addressed is: How can artificial intelligence be utilized to deliver dynamic, personalized, and interactive quizzes while ensuring data security and scalability? The identified problem was the lack of platforms offering tailored quiz content with robust real-time functionality.

The project emerged from an assignment aimed at developing an innovative quiz platform. Drawing on prior experience with the OpenAI API, the team sought to leverage AI for dynamic quiz generation. Early iterations of IntelliQ, however, were limited in scope and lacked advanced features such as multiplayer functionality, efficient server-side architecture, user customization options and crucially GDPR compliance. Scalability and responsiveness also required significant improvements to support broader adoption.

Object: The goal is to create a highly interactive and personalized quiz platform designed for educators and students who require a personalized and engaging tool for learning and assessment. Key features include:

- AI-generated quizzes tailored to user interests.
- Multiplayer functionality for real-time engagement.
- Enhanced backend scalability using Hono and Supabase.
- Features like custom avatars, document-based and random quiz generation, and bookmarking quizzes.

The implementation was based on solid theoretical and practical foundations. The backend was transitioned from Express.js to Hono. Supabase was integrated to enable real-time database updates and provide secure user authentication. The development process followed the Agile Scrum methodology, allowing efficient task prioritization and team collaboration. CI/CD pipelines were implemented to streamline deployments and updates. From a design perspective, modern features such as quiz sharing via OpenGraph tags, auditory feedback, and quizzes generated from uploaded documents were introduced. The UI was improved using frameworks like ShadCN, ensuring both accessibility and a seamless user experience.

The project successfully introduced real-time multiplayer functionality using room codes, enabling collaborative quiz sessions. Dynamic quiz creation was implemented, allowing users to generate quizzes with random topics or from uploaded documents. User experience was enhanced with features like avatars, bookmarks, and customizable themes. Performance was optimized through edge server hosting and scalable backend architecture, ensuring low latency. Additionally, the platform adheres to GDPR standards, providing robust data security.

As a result, IntelliQ now offers a versatile, interactive, and scalable learning platform. The project is prepared for submission to competitions, highlighting its technical and educational innovation.

Vorwort

We started this project because we wanted to create a fun and personalized quiz platform. It began as a school assignment in our fourth year: „Make a quiz platform about something.“ We built a basic quiz where users could choose a topic and the AI would generate questions. The feedback from teachers, classmates, and users was so positive that we decided to take it further.

The goal was to solve a common problem—finding quizzes that fit your interests. Instead of searching for quizzes online, IntelliQ creates personalized quizzes for users based on their preferences. This makes learning more engaging and fun.

What started as a simple single-player quiz is now much more. We added a multiplayer mode that lets up to 10 players compete in real-time. We also added the ability for users to upload their own documents and be quizzed on them. This makes the platform more flexible and useful for different learning needs.

IntelliQ is now a platform that provides personalized, competitive, and fun quiz experiences. What began as a school project is now a fully developed platform used by real users, and we're excited to keep improving it.

Inhaltsverzeichnis

Danksagung

We would like to express our sincere gratitude to all IntelliQ users, whose interaction and activity continuously motivated us to deliver the best experience we possibly could.

A special thank you goes to our diploma thesis advisor, Niel Widy, BA MA, for his guidance, support, and valuable feedback throughout the entire process.

1 Introduction

Ausgearbeitet von Nippon Lama

1.1 The idea and the solution

Project Idea

The origins of IntelliQ can be traced back to a school assignment given during the winter semester of the 4th year. The task was to conceptualize and develop a quiz platform. Building on prior experience with OpenAI API and technologies like React and Express.js the team envisioned a platform that could use artificial intelligence to deliver dynamic and personalized quizzes.

While the initial versions of IntelliQ laid a solid foundation, they were limited in scope and lacked several advanced features. These included real-time multiplayer functionality, scalable backend architecture, and user customization. Furthermore, improvements in performance, responsiveness, and GDPR compliance were essential for wider adoption.

To address these challenges, IntelliQ was reimagined as a comprehensive learning tool designed for both individual users and educational institutions. Its key features include:

1. AI-driven quizzes tailored to individual preferences.
2. Real-time multiplayer functionality enabling collaborative and competitive engagement.
3. Dynamic quiz generation, supporting random topics or user-uploaded content.
4. User customization options, such as avatars, themes, and bookmarks.
5. Scalable backend architecture, leveraging modern technologies like Hono and Supabase.

The solution introduces key features to make IntelliQ more interactive, scalable, and user-friendly. Multiplayer functionality allows users to join quizzes via a room code, enabling collaborative and competitive engagement. The user interface will be enhanced using modern UI frameworks to improve visual appeal and accessibility. To address performance and scalability, the backend will transition from Express.js to Hono with TypeScript, incorporating Zod for data validation and Drizzle as the ORM. Backend services will be hosted on edge servers to reduce latency and improve responsiveness. The authentication system will move from standard email-password authentication to an OTP (One-Time Password) based system with Supabase. OTP-based authentication reduces the risk of password-related breaches, such as credential stuffing or brute-force attacks. Moreover, users no longer need to remember passwords.

Non-registered users can access up to three free quizzes daily, offering a preview of the platform's capabilities. Personalization features, such as customizable avatars, adjustable primary colors and light/dark mode options, provide users with a tailored experience. The authentication system has been upgraded to the official Supabase SSR package, improving security and backend efficiency.

Other additions include replayable quizzes, random quiz generation, and the ability to create quizzes from uploaded documents for a more personalized learning experience. Social media sharing is supported through OpenGraph tags, allowing users to share quiz results. Redis caching has been implemented to improve response times, while bookmarking and quiz history management enable users to save and revisit their progress. Together, these updates create a more dynamic and engaging learning environment.

2 Design

Ausgearbeitet von Nikola Petrovic

2.1 Color Palette

[5]

2.1.1 Selection of the colors and justification

The IntelliQ interface embraces a dark, futuristic aesthetic with a minimalist color scheme designed to enhance readability and contrast.

- Background Color (040404)
  - A deep black base that provides a clean and immersive foundation.
- Dark Secondary Background (0c0d0d)
  - A subtle contrast to the primary background, used to define sections and add depth.
- Primary Accent (c8b6ff)
  - A soft neon lavender that highlights interactive elements, ensuring a modern and engaging visual experience.

2.2 Logo

[6], [7]

2.2.1 Selection of the logo and justification

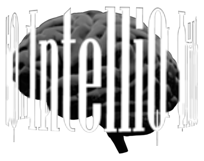


Abbildung 1: Old IntelliQ Logo

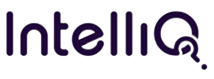


Abbildung 2: New IntelliQ Logo

The redesign of the IntelliQ logo reflects a thoughtful approach to enhancing brand identity and user recognition. IntelliQ, as a platform, incorporates a unique twist in its name, where the „Q“ is stylized as a question mark. This design choice emphasizes the core purpose of the platform—curiosity, learning, and exploration—by integrating the symbolic representation of a question into its visual identity.

Key Challenges of the Previous Logo

1. Legibility Issues:

The word „IntelliQ“ was stylized to curve around the image of a brain in the previous logo. While this approach was visually unique, it significantly reduced readability, making it harder for viewers to immediately recognize the brand.

2. Limited Accessibility:

   The artistic and abstract design required prior familiarity with the platform to understand its purpose. This limited its effectiveness in attracting new audiences and conveying its mission in marketing efforts.

3. Marketing Constraints:

   The abstract elements of the logo, though creative, lacked the clarity needed to ensure broad recognition, especially in contexts where the logo needed to stand out and communicate the brand’s identity quickly.

Key Aspects of the Redesign

1. Improved Readability:

   The updated logo simplifies the design, making the word „IntelliQ“ more legible and ensuring that its message is clear at a glance.

2. Versatility:

   The new logo is designed for use across multiple applications, from digital platforms to printed materials, ensuring it is functional in various marketing and promotional contexts.

3. Modern Aesthetic:

   The updated design aligns with modern branding principles, focusing on clarity and user-friendliness while reflecting the platform’s innovative mission.

2.3 Mockup design

Ausgearbeitet von Nikola Petrovic

[8], [9], [10], [11]

2.3.1 Dashboard

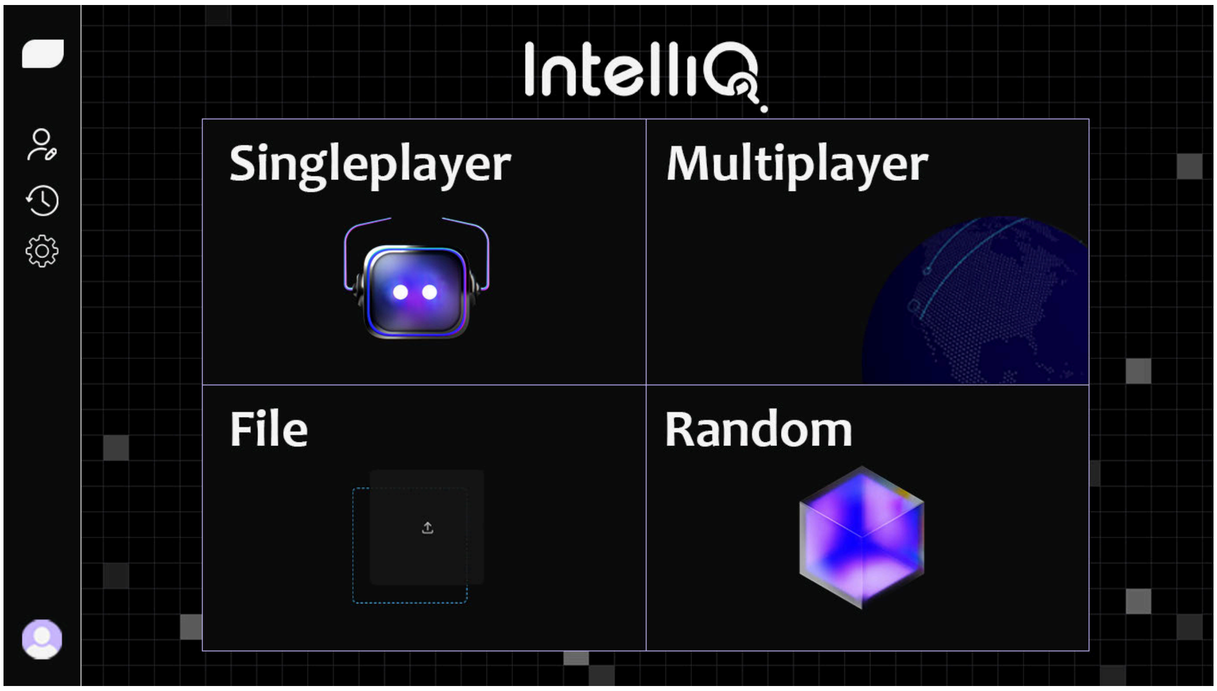


Abbildung 3: Dashboard

The redesigned IntelliQ dashboard focuses on showcasing the newly introduced features. By dividing the interface into four distinct sections, each representing a different quiz mode, the updated layout embraces the expanded functionalities while maintaining efficient access to key features.

1. UX and Navigation
    - Singleplayer: The core experience of IntelliQ remains prominently displayed to ensure continuity for returning users. A bot icon represents this mode, emphasizing the AI-powered nature of the quiz system.
    - Multiplayer: A newly introduced mode that enables competition between users, incorporating a social and competitive learning aspect. This mode is visually represented by a globe, symbolizing global interaction.
    - File-Based Quizzes: A feature that allows users to upload documents and generate quizzes based on their content, making the platform more dynamic and adaptable for self-directed learning where users can actively engage with their study materials. An upload box icon is used to indicate

document-based quiz creation, signalling to the user that they can upload a file to generate a quiz. This design choice intuitively conveys the action required, ensuring clarity and ease of use.

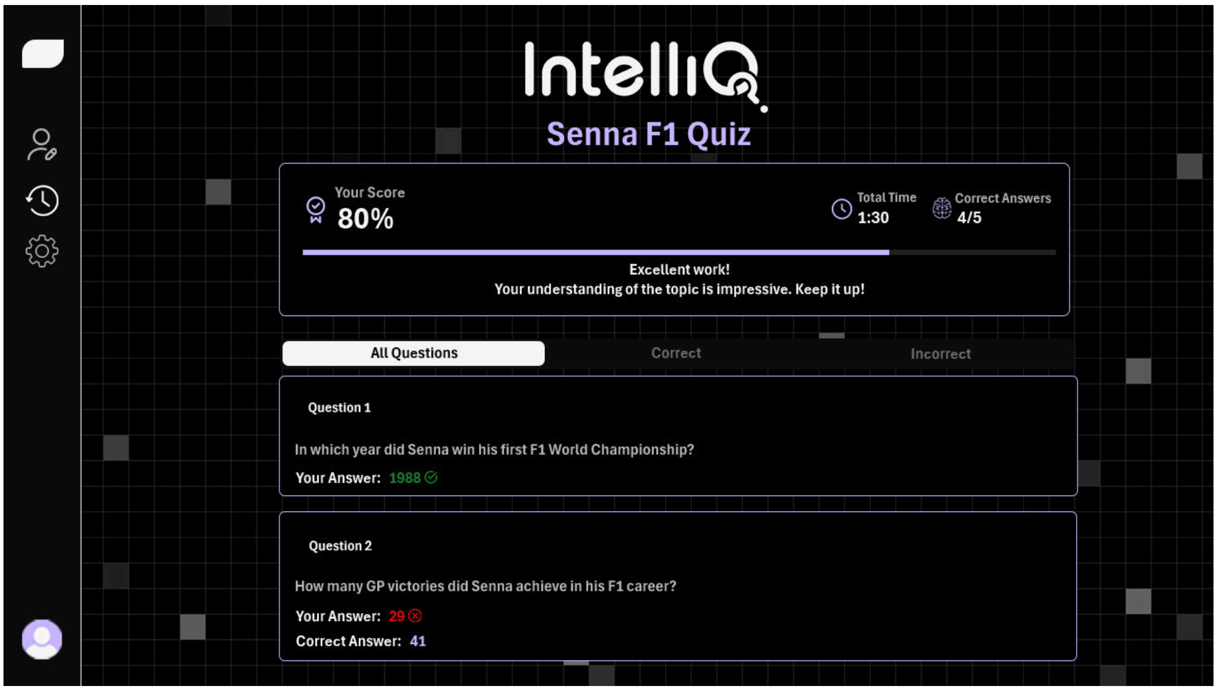
- Random Quizzes: A quiz mode designed to provide a randomized set of questions from various topics, catering to users who prefer an unpredictable challenge. A 3D cube icon represents this mode, visually conveying the unpredictability and diversity of the generated topics.

2. Sidebar Integration

   A Sidebar has been implemented to enhance usability, providing easy access to essential features such as:

   - Home: The central hub for navigating IntelliQ.
   - Quiz Me: A quick-access option for starting a new quiz.
   - Multiplayer: Entry point for joining or creating multiplayer quizzes.
   - Documents: Quick-access for starting a document based quiz.
   - Random: Quick-access for starting a random quiz.
   - History: Enables users to review past quizzes and track progress.
   - Bookmarks: Let's the users see their bookmarked quizzes.
   - Settings: Allows custosmization of user preferences.

2.3.2 Singleplayer Quiz Summary

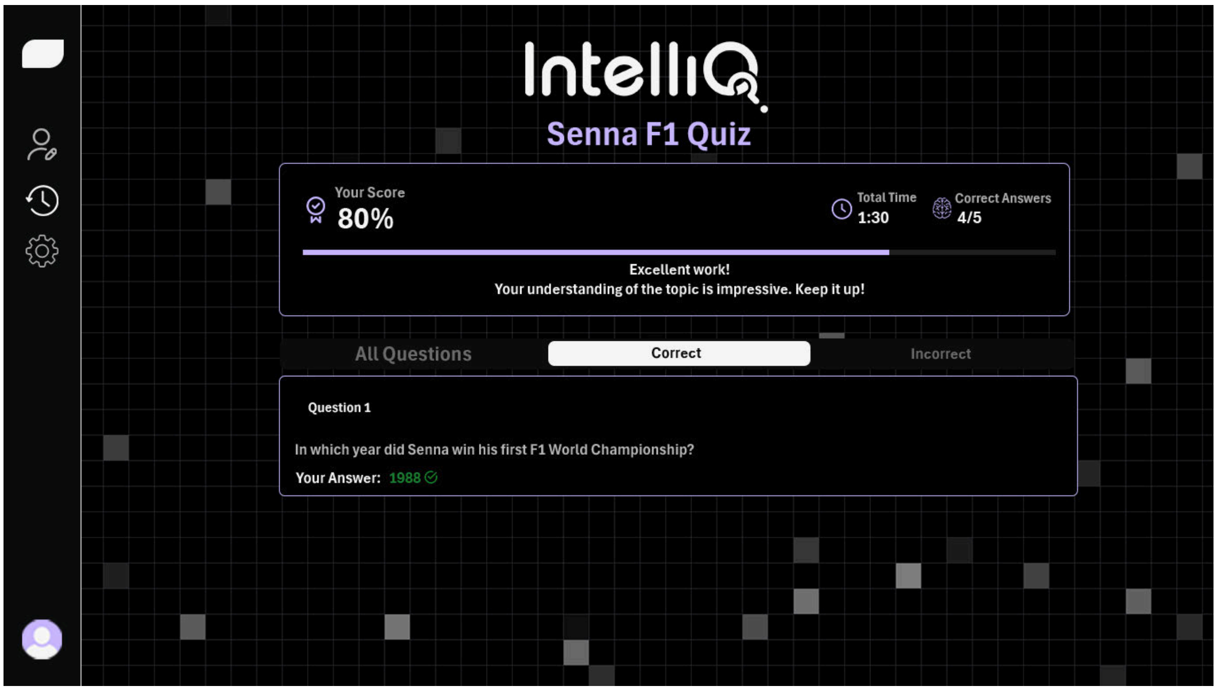


Excellent work!
Your understanding of the topic is impressive. Keep it up!

Question 1
In which year did Senna win his first F1 World Championship?
Your Answer: 1988

Question 2
How many GP victories did Senna achieve in his F1 career?
Your Answer: 29
Correct Answer: 41

Abbildung 4: Single Quiz Summary 1



IntelliQ
Senna F1 Quiz

Your Score
80%

Total Time
1:30

Correct Answers
4/5

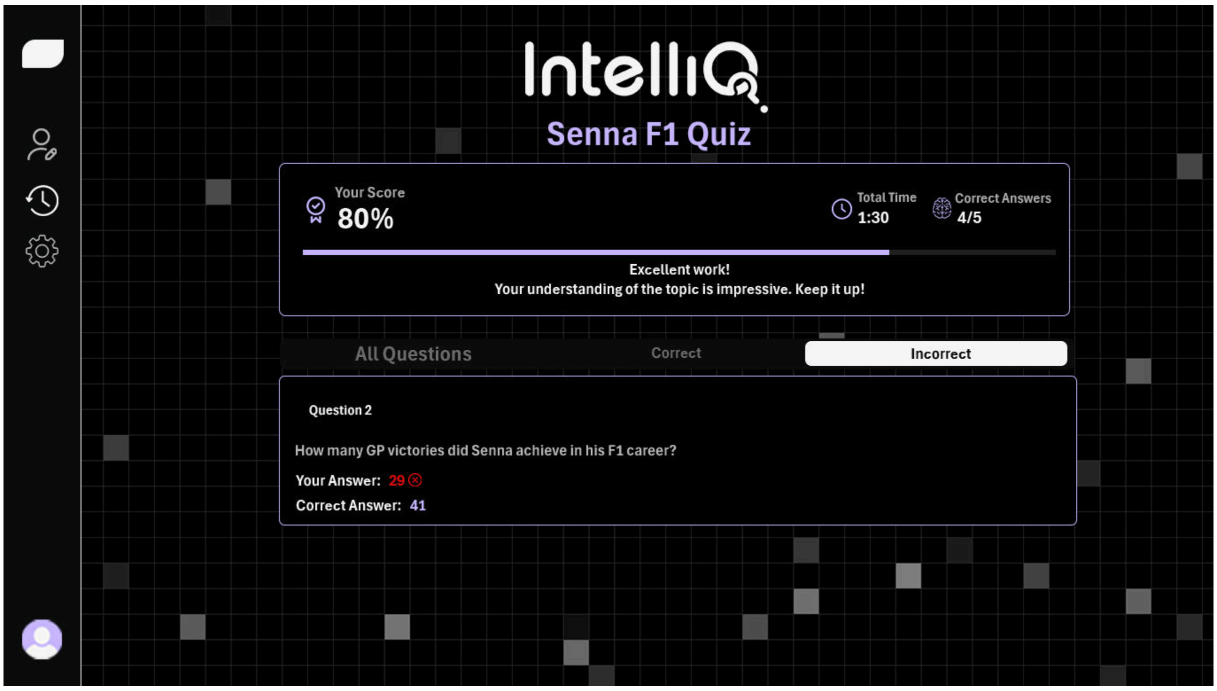
Excellent work!
Your understanding of the topic is impressive. Keep it up!

Question 1

In which year did Senna win his first F1 World Championship?

Your Answer: 1988

Abbildung 5: Single Quiz Summary 2



IntelliQ
Senna F1 Quiz

Your Score
80%

Total Time
1:30

Correct Answers
4/5

Excellent work!
Your understanding of the topic is impressive. Keep it up!

Question 2

How many GP victories did Senna achieve in his F1 career?

Your Answer: 29

Correct Answer: 41

Abbildung 6: Single Quiz Summary 3

The new Singleplayer Quiz Summary was designed to enhance the review process by emphasizing learning from the summary itself, providing users with a structured way to analyze their quiz performance while making it easier to identify correct answers and mistakes. By incorporating these features, the updated design transforms the results screen into an interactive learning tool, ensuring that users can actively engage with their quiz results, reinforce their understanding, and improve retention for future quizzes.

1. Interactive Question Filtering

   To improve the user experience and support different learning needs, the summary includes filtering options that allow users to:

   - View only incorrect questions: This helps users focus specifically on the areas where they made mistakes, reinforcing the learning process by enabling targeted review.
   - View only correct questions: Users can choose to see only the questions they answered correctly, allowing them to reinforce their knowledge on topics they have already mastered.
   - View all questions: For those who prefer to go through the entire quiz, this option provides a full overview of both correct and incorrect answers.

2. Design Enhancements for Effective Learning

   - Question Breakdown: Each question is visually structured, with user responses highlighted and correct answers displayed for easy comparison.
   - Interactive Navigation: The filtering options are placed in a way that ensures quick toggling between views, making the review process efficient and user-friendly.

2.3.3 Join Multiplayer Lobby

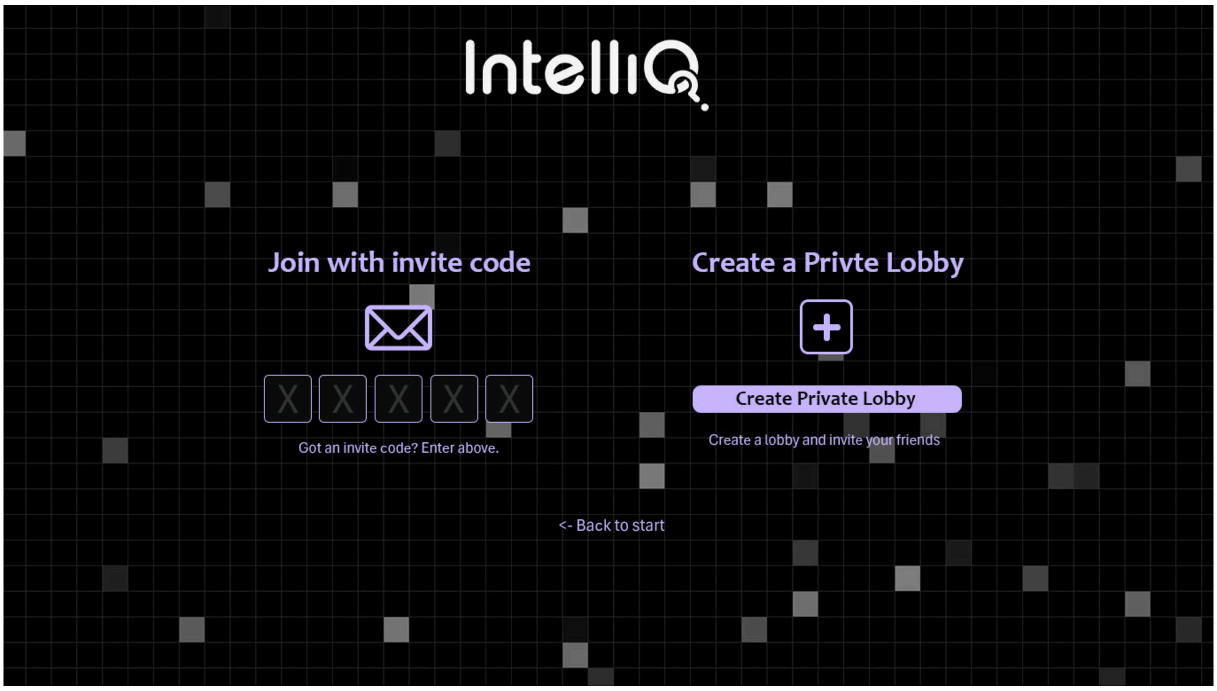


Abbildung 7: Join Multiplayer Lobby

The multiplayer lobby interface is designed for clarity and ease of use, allowing users to either join a lobby with an invite code or create a private lobby.

1. Design Considerations
   - Balanced Layout: A two-column structure ensures equal emphasis on both options.
   - Clear Visual Cues:
     - A mail icon represents the invite code input, reinforcing its purpose.
     - A plus icon symbolizes lobby creation, guiding users intuitively.
   - Navigation Simplicity: The “Back to start” option ensures users can easily return if needed.

2.3.4 Multiplayer Lobby

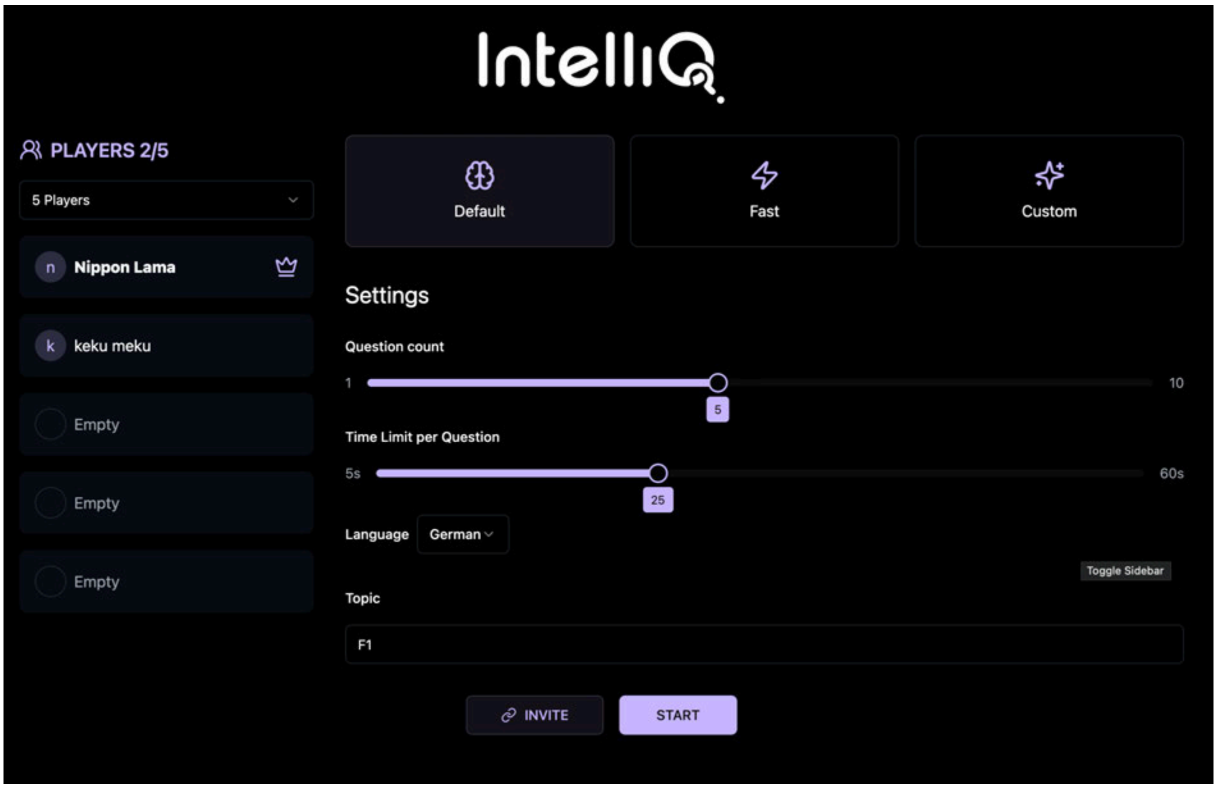


Abbildung 8: Multiplayer Lobby

The multiplayer lobby interface is structured to balance player management and quiz customization, ensuring an efficient user experience. The layout follows a 1/3 - 2/3 split, where the player list remains easily visible while providing enough space for settings adjustments.

1. Layout & Visual Hierarchy

- Player List (1/3 of the screen):
  - Displays all joined players with a dropdown at the top for the lobby owner to set the maximum player count.
  - The compact design ensures that the focus remains on quiz settings while keeping essential player information accessible.
- Quiz Settings (2/3 of the screen):
  - Question Count & Time Limit: Adjustable sliders provide visual and interactive control.
  - Language Selection: A dropdown for seamless multilingual support.
  - Quiz Topic Input: Customizable field to tailor the quiz theme.

2. Presets

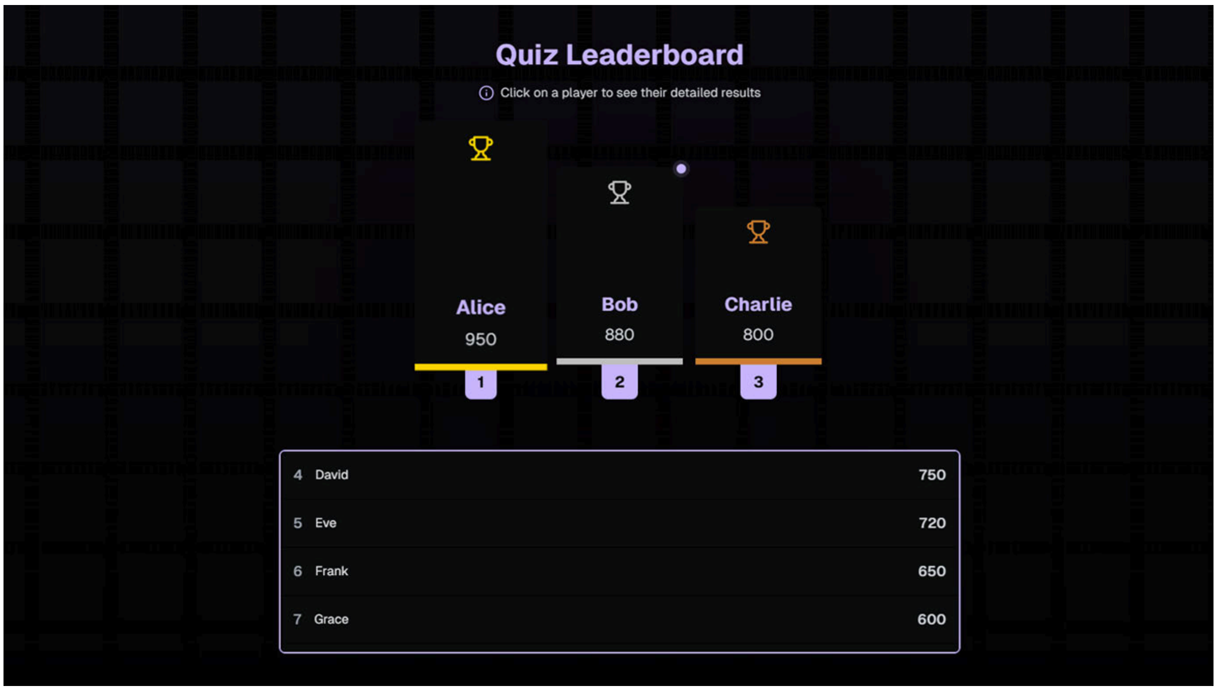
   At the top, three preset buttons allow users to quickly define the quiz experience:

   - Default: A well-balanced standard mode.
   - Fast: A high-speed mode with a lower time limit.
   - Custom: Allows full manual control over settings.

3. Call-to-Action Placement

   - Invite Button: Positioned for quick access to share the lobby.
   - Start Button: Designed with clear prominence to guide users toward launching the quiz.

2.3.5 Multiplayer leaderboard


Quiz Leaderboard

Click on a player to see their detailed results

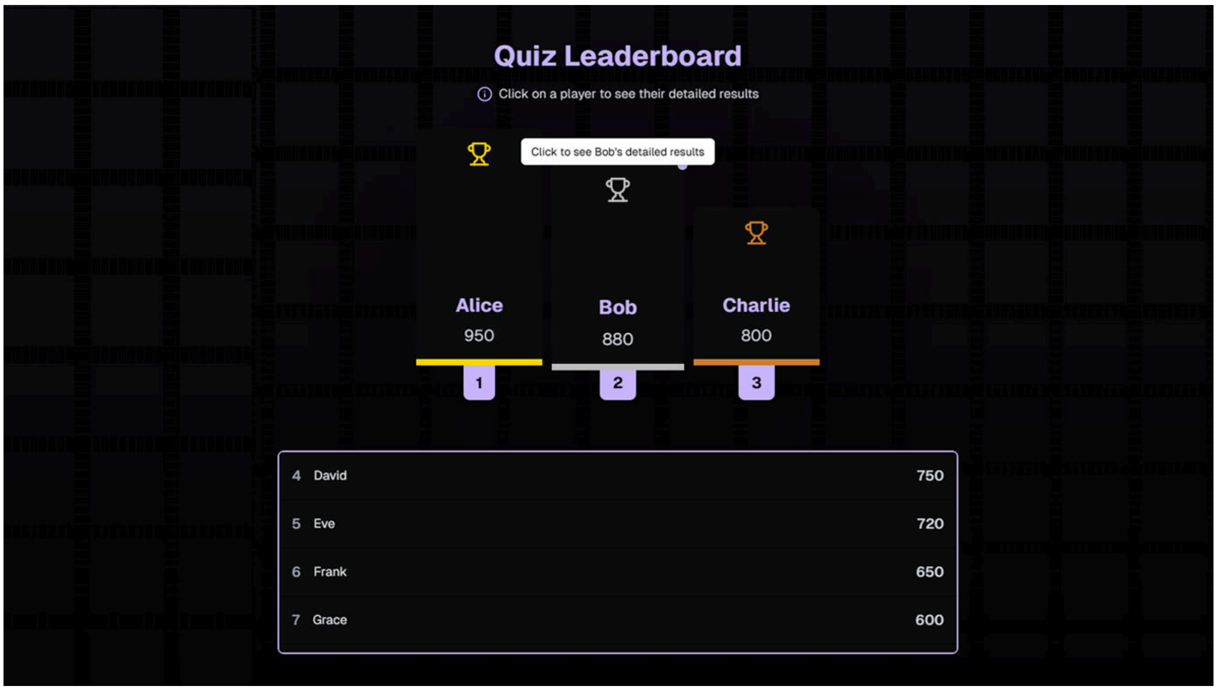
Alice 950 — 1

Bob 880 — 2

Charlie 800 — 3

4	David	750
5	Eve	720
6	Frank	650
7	Grace	600

Abbildung 9: Multiplayer Leaderboard


Quiz Leaderboard

Click on a player to see their detailed results

Click to see Bob's detailed results

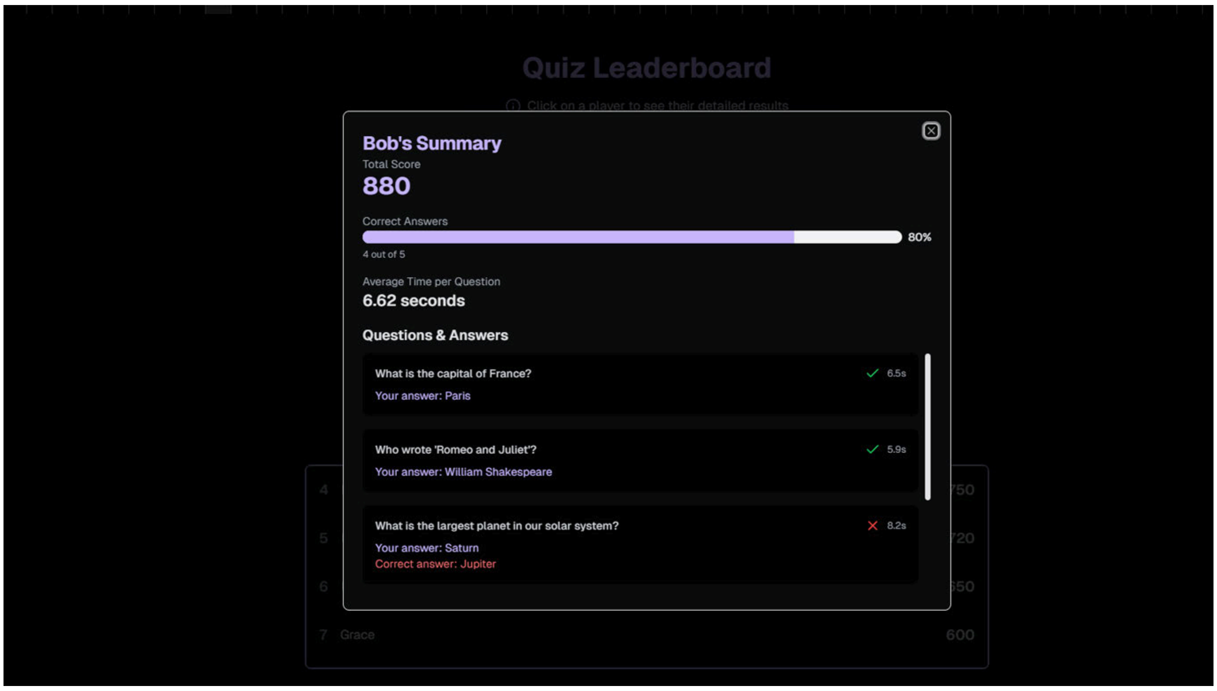
1	Alice	950
2	Bob	880
3	Charlie	800
4	David	750
5	Eve	720
6	Frank	650
7	Grace	600

Abbildung 10: Multiplayer Leaderboard Hover

A podium-style ranking system highlights the top three players, using scaled card heights and color-coded accents (gold, silver, bronze) to establish a clear ranking distinction. Below the podium, remaining players are displayed in a minimalistic leaderboard list, maintaining visibility without overwhelming the UI.

- Visual Hierarchy & Layout
  - Emphasized Top Rankings: The tiered podium design draws attention to the highest-scoring players, creating a clear focal point while reinforcing competition.
  - Minimalist Leaderboard List: Lower-ranked players are presented in a clean, structured list, ensuring clarity while keeping the interface visually balanced.
  - Color-Coded Accents: Gold, silver, and bronze provide instant rank recognition, reducing cognitive load.

2.3.6 Multiplayer Summary


Quiz Leaderboard

Click on a player to see their detailed results

Bob's Summary

Total Score

880

Correct Answers

80%

4 out of 5

Average Time per Question

6.62 seconds

Questions & Answers

What is the capital of France? ✓ 6.5s
Your answer: Paris

Who wrote 'Romeo and Juliet'? ✓ 5.9s
Your answer: William Shakespeare

What is the largest planet in our solar system? ✗ 8.2s
Your answer: Saturn
Correct answer: Jupiter

4		750
5		720
6		650
7	Grace	600

Abbildung 11: Multiplayer Player Summary

The leaderboard is interactive, allowing players to click on their name to access a detailed performance breakdown using a modal that keeps the leaderboard visible in the background, allowing for a seamless transition between rankings and personal results.

- A modal summary overlays the leaderboard, providing:
    - Total Score & Accuracy, visually reinforced by a progress bar.
    - Average Response Time per Question, aiding self-evaluation.
    - Question Breakdown, where correct and incorrect answers are clearly distinguished with color contrast, enhancing readability.
- Color-Coded Feedback:
    - Correct answers are highlighted in a our primary tone.
    - Incorrect answers are displayed with a subtle red highlight, reinforcing learning without overwhelming the user.

2.3.7 Quiz History and Bookmarked Quizzes

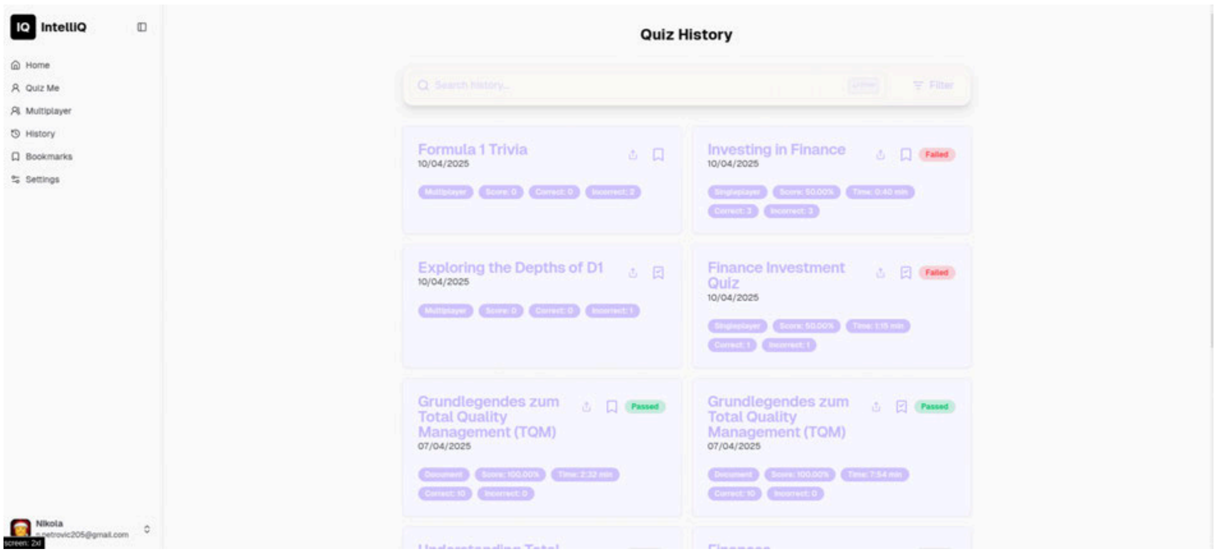
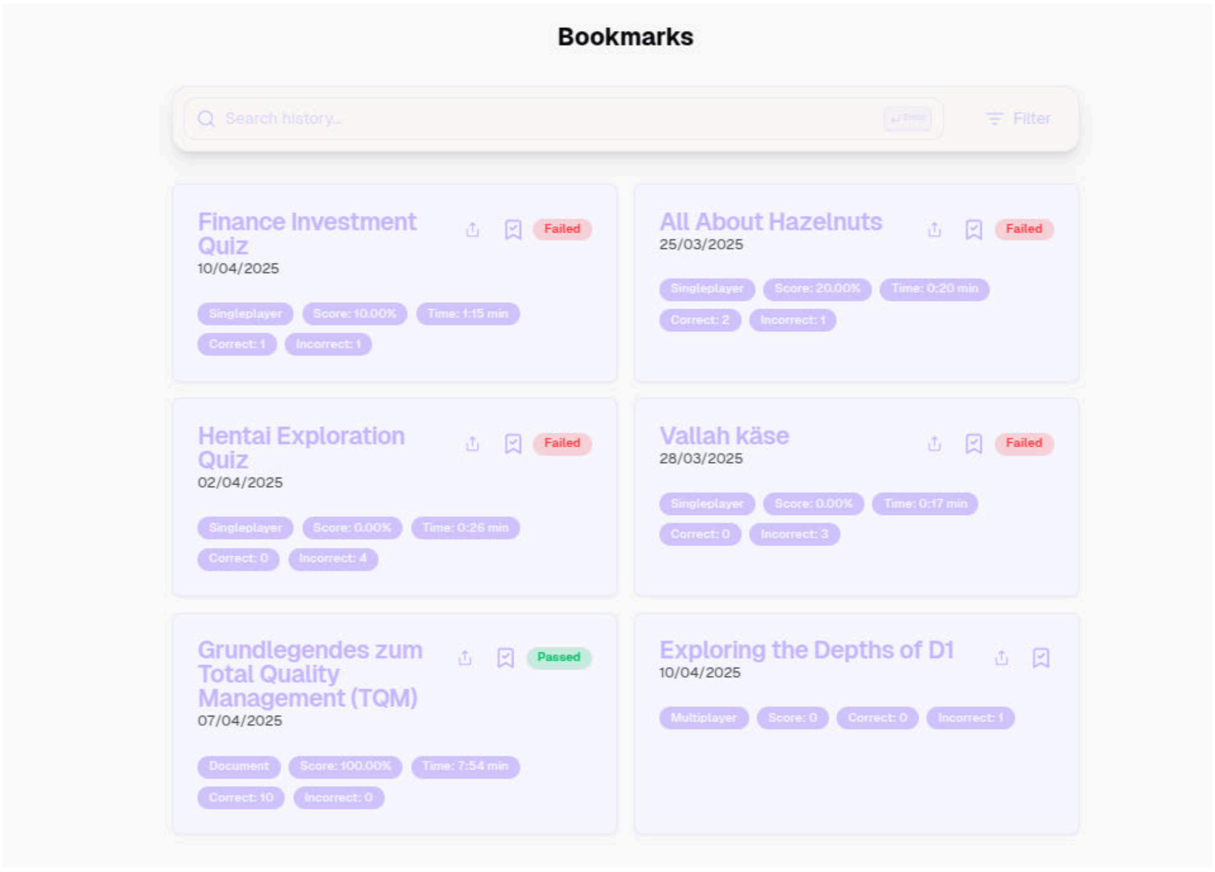


Abbildung 12: Quiz History

This page allows users to efficiently revisit their past quizzes, offering detailed insights into their performance while keeping the interface clean and user-friendly.

- Layout & Structure
    - Search Bar: Positioned at the top for easy filtering, helping users locate specific quizzes by title, date, or other criteria.
    - Grid-Based Quiz Cards: The bookmarked quizzes are displayed in a structured grid layout, giving the user an instant overview of their bookmarked quizzes.
    - Quiz Information on Cards:
        - Quiz Title for instant recognition.
        - Date Taken & Duration to provide context.
        - Score Percentage for performance tracking.
        - Bookmark Icon as a visual indicator, reinforcing that these quizzes are saved.
    - Interactive Elements: The bookmark icon, search bar and Quiz Share Button enhance usability, making navigation effortless.

2.3.7.1 Bookmarked Quizzes



Bookmarks

Finance Investment Quiz
10/04/2025
Failed
Singleplayer | Score: 10.00% | Time: 1:15 min | Correct: 1 | Incorrect: 1

All About Hazelnuts
25/03/2025
Failed
Singleplayer | Score: 20.00% | Time: 0:20 min | Correct: 2 | Incorrect: 1

Hentai Exploration Quiz
02/04/2025
Failed
Singleplayer | Score: 0.00% | Time: 0:26 min | Correct: 0 | Incorrect: 4

Vallah käse
28/03/2025
Failed
Singleplayer | Score: 0.00% | Time: 0:17 min | Correct: 0 | Incorrect: 3

Grundlegendes zum Total Quality Management (TQM)
07/04/2025
Passed
Document | Score: 100.00% | Time: 7:54 min | Correct: 10 | Incorrect: 0

Exploring the Depths of D1
10/04/2025
Multiplayer | Score: 0 | Correct: 0 | Incorrect: 1

Abbildung 13: Bookmarks

This bookmarked quizzes page allows users to efficiently revisit their most relevant quizzes while maintaining a minimalistic design. The layout and structure have been kept the same as for the quiz history to ensure consistency, making navigation seamless and intuitive across different sections of the platform.

2.3.8 Quiz Sharing

The quiz sharing UI provides a simple, user-friendly interface for users to create and share quiz links efficiently. The design focuses on providing all the necessary functionality while keeping the interface clean and intuitive.

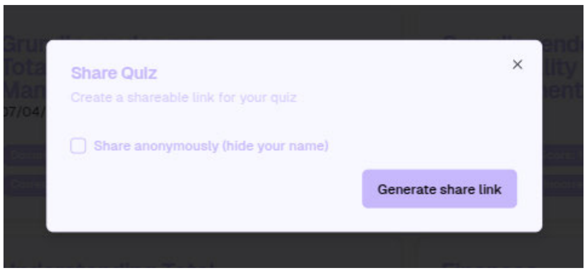
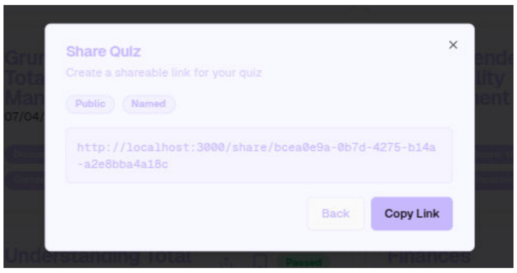
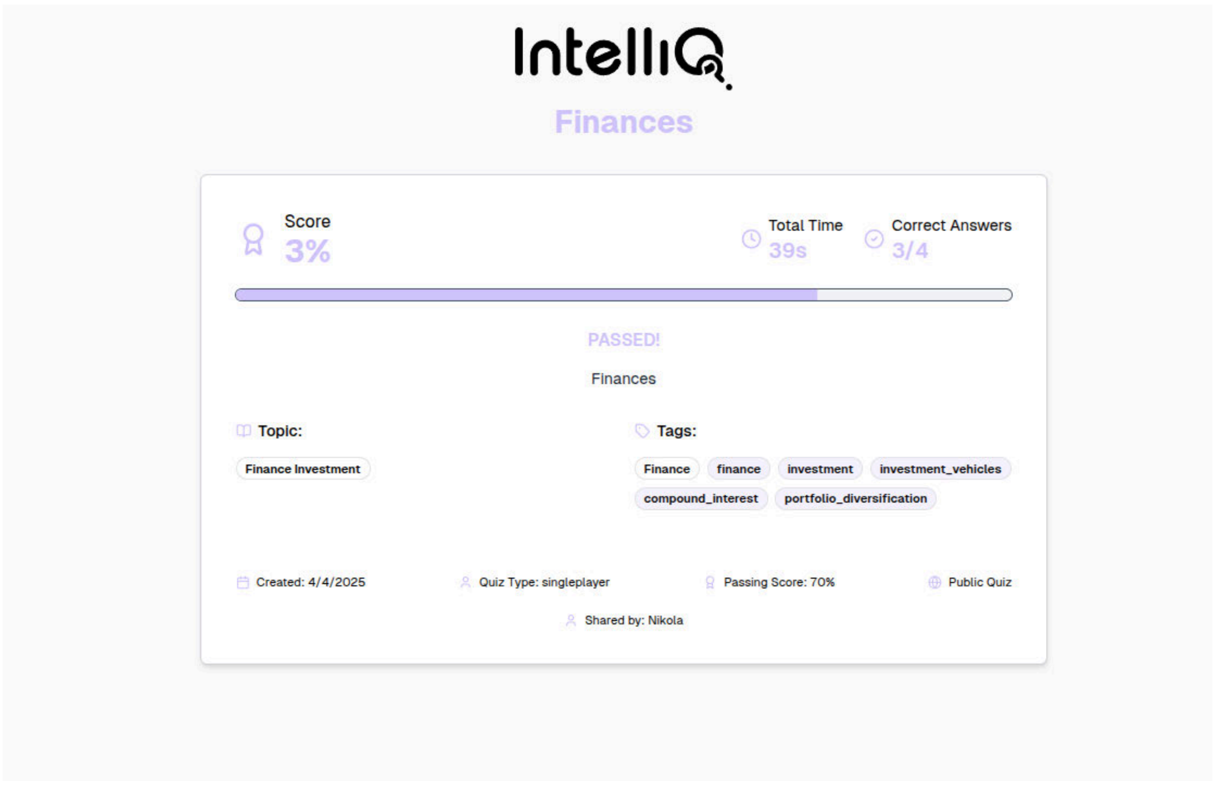

Share Quiz
Create a shareable link for your quiz
Share anonymously (hide your name)
Generate share link

Abbildung 14: Generate share link



http://localhost:3000/share/bcea0e9a-0b7d-4275-b14a-a2e8bba4a18c

Abbildung 15: Copy share link



Score 3%

Total Time 39s

Correct Answers 3/4

PASSED!

Finances

Topic: Finance Investment

Tags: Finance, finance, investment, investment_vehicles, compound_interest, portfolio_diversification

Created: 4/4/2025

Quiz Type: singleplayer

Passing Score: 70%

Public Quiz

Shared by: Nikola

Abbildung 16: Shared Quiz Summary

- Layout & Structure
  - Popup Window: Clicking the share button opens a modal window, where users can generate a shareable link for the selected quiz.
  - Link Generation Options:
    - Anonymity Option: An option is available to share the quiz anonymously, hiding the user's name from the recipients.
  - Action Buttons:
    - Copy Link: Once the link is generated, users can easily copy it to their clipboard with a single click.
    - Generate Link: After configuring the sharing preferences (public or anonymous), the user can click this button to create the link.
  - Design Aesthetic:
    - Minimalist & User-Friendly: The design remains clean and intuitive, focusing on ease of use. The button is prominently displayed on each quiz card, and the modal window follows a minimalistic design with smooth transitions and clear visual hierarchy.
    - Quick Feedback: After generating the link, the popup provides immediate feedback, such as displaying the full URL for easy copying or confirming the anonymity option chosen.

3 Frontend Development

3.1 Overview of Next.js Framework and Key Features

Ausgearbeitet von Nippon Lama

Abbildung 17: Next.js built on React

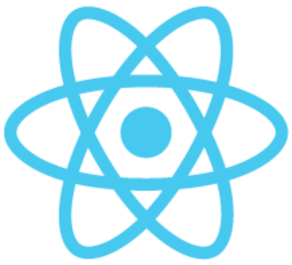


Abbildung 18: React

Next.js is an open-source framework built on React, designed to simplify the development of dynamic, scalable web applications. Unlike traditional React setups, which rely on client-side rendering, Next.js offers a hybrid approach, combining server-side rendering (SSR) and static site generation (SSG). These features enable better performance, SEO, and flexibility for developers. With its app directory architecture, Next.js provides a modern approach to organizing and managing routes, making it a powerful framework for both frontend and backend development. [12]

Key Features of Next.js

1. File-Based Routing: Next.js's app directory introduces a new way of handling routing, where folders and files within the app/ directory map directly to application routes. For example:
   - A folder named `multiplayer` containing a file `page.tsx` corresponds to the `/quiz_ route`.
   - Dynamic routes are defined with brackets, such as `[id]/page.tsx`, which maps to `/multiplayer/:id`.
2. Layouts: Layouts provide a structured way to define reusable components or sections that wrap around specific parts of the application, such as navigation bars, footers, or sidebars.
   - A child layout can inherit a parent layout. For instance, the main navigation bar can be defined in a root layout, while individual sections like dashboard or multiplayer can add their own layouts to extend functionality.

   [13]

IntelliQ uses a sidebar as the primary navigation element, allowing users to seamlessly access various sections of the application. This centralized navigation system ensures that users can efficiently explore the platform's functionalities.

To achieve this, IntelliQ employs a reusable `Layout` component that wraps the entire application. Below is the implementation of the `Layout` component:

```
export default function Layout({ children }: { children:
React.ReactNode }) {
  return (
    <SidebarProvider>
      <div className='...'>
        {/* Sidebar */}
        <AppSidebar />

        {/* Main Content */}
        <div className='...'>
          <SidebarTrigger />
          <main className='...'>
            {children}
          </main>
        </div>
      </div>
    </SidebarProvider>
  );
}
```

The `SidebarProvider` manages the sidebar's open and close states, ensuring consistency across the application without requiring prop drilling. This centralized approach simplifies the codebase and improves maintainability.

The sidebar, implemented through the `AppSidebar` component, provides users with a consistent navigation structure. Whether navigating between single-player quizzes, multiplayer lobbies, or quiz creation tools, the sidebar remains a persistent element, ensuring ease of use.

The `{children}` placeholder in the Layout component allows IntelliQ to dynamically load content based on the current route. For example:

- On the dashboard, it renders all the availabale quiz modes.
- In Multiplayer Mode, it loads lobby details and real-time game updates.
- On the history page, it showcases past quiz results and performance metrics.

This dynamic rendering ensures that IntelliQ can offer a personalized and context-specific user experience for each route without requiring redundant layout components or unnecessary page reloads.

3. SEO:
   - Server-Side Rendering (SSR)
   - Static Site Generation (SSG)
   - Dynamic Metadata Management: Next.js integrates seamlessly with tools like `next/head` to manage metadata dynamically. This includes:
     - Title tags: Customizing page titles such as `Quiz | IntelliQ`
     - Meta descriptions: Adding meaningful descriptions to improve click-through rates.
     - Open Graph tags: Optimizing content sharing on social media platforms.
4. API Routes: The `app/` directory allows developers to define API routes directly within the same structure. This eliminates the need for separate backend services for many use cases.

   Files located within the `app/api` directory are automatically mapped to `/api/*`, functioning as API endpoints rather than standard pages.
5. Edge Computing and Performance: Hosting backend services on edge servers through Next.js significantly reduces latency and improves responsiveness.
6. TypeScript and Modular Styling: Built-in TypeScript support and CSS Modules in the app directory enhance development efficiency, ensuring code reliability and maintainability.

Abbildung 19: Typescript

Comparison: Next.js vs React

Feature	Next.js	React
Rendering Options	Supports SSR, SSG, ISR, and CSR	Only CSR (requires additional tools like Gatsby or custom setups for SSR/SSG)
Routing	File-based routing system	Requires libraries like `react-router` for routing
Performance	Out-of-the-box optimizations like code-splitting and image optimization	Requires manual optimizations or third-party libraries
SEO	Built-in support for SSR and metadata management	Limited to client-side rendering, requiring external libraries like `react-helmet`
Ease of Use	Simplified setup and configuration	Highly customizable but requires additional setup for advanced use cases
Development Time	Faster due to built-in features	Slower as you need to set up tools for SSR, SSG, or routing
API Integration	Built-in API routes	Requires separate backend setup or integration
Ecosystem	Strong community support and Vercel's active development	Larger ecosystem but lacks the streamlined full-stack features of Next.js

Tabelle 1: Next.js vs React

Impact of Next.js on IntelliQ

Next.js's app directory simplified the development process by eliminating the need for additional routing libraries and reducing configuration complexity. The hybrid rendering model, combined with edge server hosting, improved SEO and responsiveness, particularly for real-time multiplayer quizzes.

Why Next.js Was Chosen

Our decision to use Next.js was influenced by our prior experience with React and the limitations we encountered using React Router. Next.js, with its app directory structure and built-in routing, addressed these challenges, offering a more streamlined development process. Its ability to handle SSR and SSG further enhanced scalability and performance, making it the ideal choice for IntelliQ.

3.2 Supabase Introduction

Abbildung 20: Supabase

Supabase is an open-source backend-as-a-service platform that helps developers build scalable and secure applications without managing complex backend infrastructure. It provides a hosted PostgreSQL database with real-time capabilities, user authentication, row-level security, file storage, and automatic APIs based on database tables. In IntelliQ, Supabase plays a central role in managing the backend services, including storing user data, authenticating users, handling quiz results, and enabling real-time communication for multiplayer games. [14]

One of the key features of Supabase is its real-time database updates using PostgreSQL replication, which is important for live multiplayer quizzes. It also offers powerful access control using Row Level Security (RLS), making it easy to manage who can read or write data. Supabase Authentication supports different login methods such as email, password, OTP, and third-party providers like Google or GitHub. For IntelliQ, we use Supabase's OTP-based login system to improve user experience.

Supabase also provides RESTful and GraphQL APIs automatically for each database table, which speeds up development. The platform includes edge functions for serverless computing, allowing us to run custom backend logic close to the user. Additionally, Supabase Storage allows us to store files like images and documents securely.

As a growing developer-first platform, Supabase has a strong community and actively organizes hackathons where developers build creative projects using Supabase. These events help the community grow and contribute new ideas to the open-source project. Supabase is backed by solid documentation and continues to release regular updates, making it a reliable and modern choice for projects like IntelliQ.

The following table compares it with other popular backend solutions:

Feature	Supabase	Firebase	Hasura
Database	PostgreSQL(SQL, relational)	Firestore (NoSQL)	PostgreSQL
Authentication	Email/Password, OTP, OAuth	Email/Password, OAuthPerformance	Requires integration
Real-Time	Built-in via PostgreSQL replication	Built-in with Firestore	Requires manual setup
APIs	REST + GraphQL auto-generated	REST + SDKs	Instant GraphQL
File Storage	Built-in	Built-in	External setup required
Self-hosting	Supported	Not officially supported	Supported
Open Source	Yes	No	Yes
Edge Functions	Yes	Limited	Needs integration
Developer Experience	Simple, SQL-based, well documented	Great but more opinionated	Good but requires knowledge

Tabelle 2: BaaS (Back as a service) comaprison

Benefits of Choosing Supabase for IntelliQ

1. Combines multiple features in a single platform (auth, DB, storage, realtime).
2. Easy SQL-based setup and relational data structure.
3. Real-time features are natively integrated.
4. OTP login improves user convenience.
5. Active open-source community and frequent hackathons.
6. Self-hosting and scalability options available.

[14]

3.3 ShadCN UI: Integration and Benefits for UI Design

ShadCN UI is a modern React component library that focuses on accessibility, design flexibility, and developer experience. It is built on top of Radix UI primitives and uses Tailwind CSS for styling, making it easy to integrate and customize. In IntelliQ, ShadCN UI is used to build the user interface, including the sidebar, navigation bar, buttons, inputs, and other reusable components.

The library provides accessible components by default and allows developers to override or extend styles without losing consistency. Since ShadCN UI follows the „bring your own design system“ approach, it does not lock developers into a specific theme. Instead, it provides base components with well-defined behavior, giving full control over layout, spacing, colors, and interactivity.

ShadCN UI works well with modern frontend stacks like Next.js, Vite, Tailwind CSS, and TypeScript. It also supports server components in the new app directory of Next.js and is compatible with React Server Components (RSC).

ShadCN UI is used throughout IntelliQ’s frontend for building all major components in a clean, accessible, and modular way. It powers many elements of the interface, including:

1. The sidebar used for navigating the platform
2. The topbar used for triggering UI actions like theme switching
3. Buttons used across the quiz and dashboard features
4. Input fields, dropdowns, and form controls for quiz creation
5. Cards and modals used for displaying content and confirmations
6. OTP component for user authentication

The fact that ShadCN supports server components in the Next.js app directory means we could also use its components in server-rendered pages without issues.For example, the layout component that wraps all routes uses a combination of sidebar and header components built with ShadCN UI. Inside pages like the quiz creation interface or quiz history, various form inputs and interactive buttons use the same design system, keeping the UI consistent across the app. The design flexibility of ShadCN UI also allowed us to experiment with light/dark mode and dynamic theming, which we integrated into the user profile settings.

The following table compares ShadCN UI with other popular UI libraries/Frameworks:

Feature	ShadCN UI	Material UI	Chakra UI	Quasar Framework
Design System	Bring your own	Google Material Design	Chakra Design System	Quasar Design System
Styling	Tailwind CSS	CSS-in-JS	Emotion / Style Props	SCSS / CSS
Accessibility	Built-in via Radix	Good	Good	Good
Customizability	High (full control)	Moderate (theme override)	High (theme tokens)	High (via configs and props)
SSR Support	Yes (Next.js App Directory)	Yes	Yes	Partial (requires setup)
Components	Minimal, extendable	Comprehensive	Comprehensive	Large set
Focus	Accessibility, minimalism	Design standards	Simplicity and flexibility	Vue-based SPA/ PWA/Desktop

Tabelle 3: UI Library comparison

[15]

3.4 Tailwind CSS

Ausgearbeitet von Nikola Petrovic [16] Tailwind CSS is a utility-first CSS framework that provides developers with small utility classes to create custom designs without relying on predefined components or themes. Unlike traditional CSS frameworks, which offer a set of pre-designed components, Tailwind gives you full control over the styling of each element. [16]

- Key Features of Tailwind CSS
    - Utility-First Approach: Tailwind promotes a utility-first approach, meaning instead of writing custom CSS or using pre-made components, developers apply utility classes directly to HTML elements.
    - Responsive Design: Tailwind makes it easy to design responsive layouts using its built-in responsive utilities. By adding breakpoint prefixes (like sm:, md:, lg:), you can tailor designs for different screen sizes without writing custom media queries.
    - Easy to Learn and Use: The utility classes are simple and straightforward to use. Developers can quickly grasp Tailwind's approach without needing to learn complex class names or styling conventions.

3.5 Frontend Architecture and Structure

Ausgearbeitet von Nippon Lama

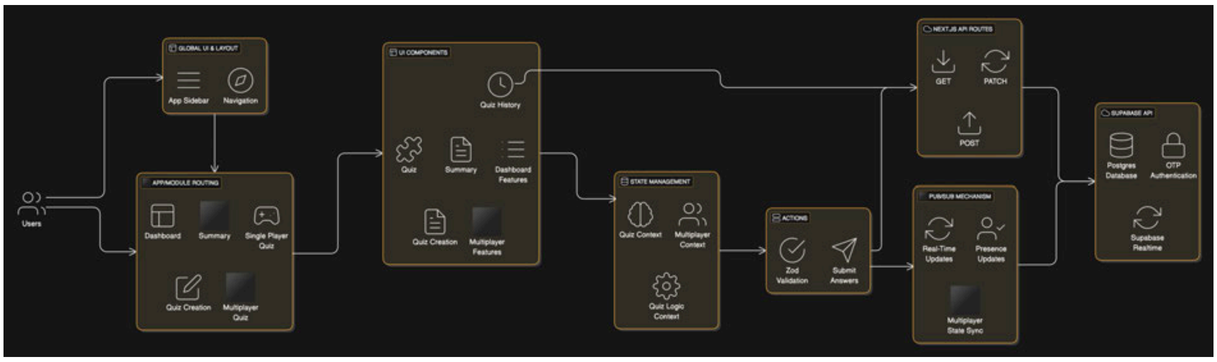


Abbildung 21: Intelliq simplified Frontend Architecture

IntelliQ's frontend architecture is designed for smooth navigation, real-time updates, and efficient state management. Users interact with a structured layout that keeps the sidebar and navigation consistent while dynamically rendering content based on their actions. The application manages quiz data, multiplayer sessions, and game logic, ensuring state updates without unnecessary re-renders. When users submit answers or perform actions, the system validates the data before sending requests to the backend. API routes handle authentication, database updates, and multiplayer synchronization, ensuring real-time feedback. Presence tracking and live updates make interactions responsive and engaging. This structure keeps the application modular, scalable, and optimized for performance.

3.5.1 User Flow Management

IntelliQ is built for a smooth and efficient user experience, managing authentication, navigation, quiz interactions, and result tracking. With a structured approach, it optimizes quiz engagement, multiplayer functionality, and dashboard navigation while keeping the architecture scalable and maintainable.

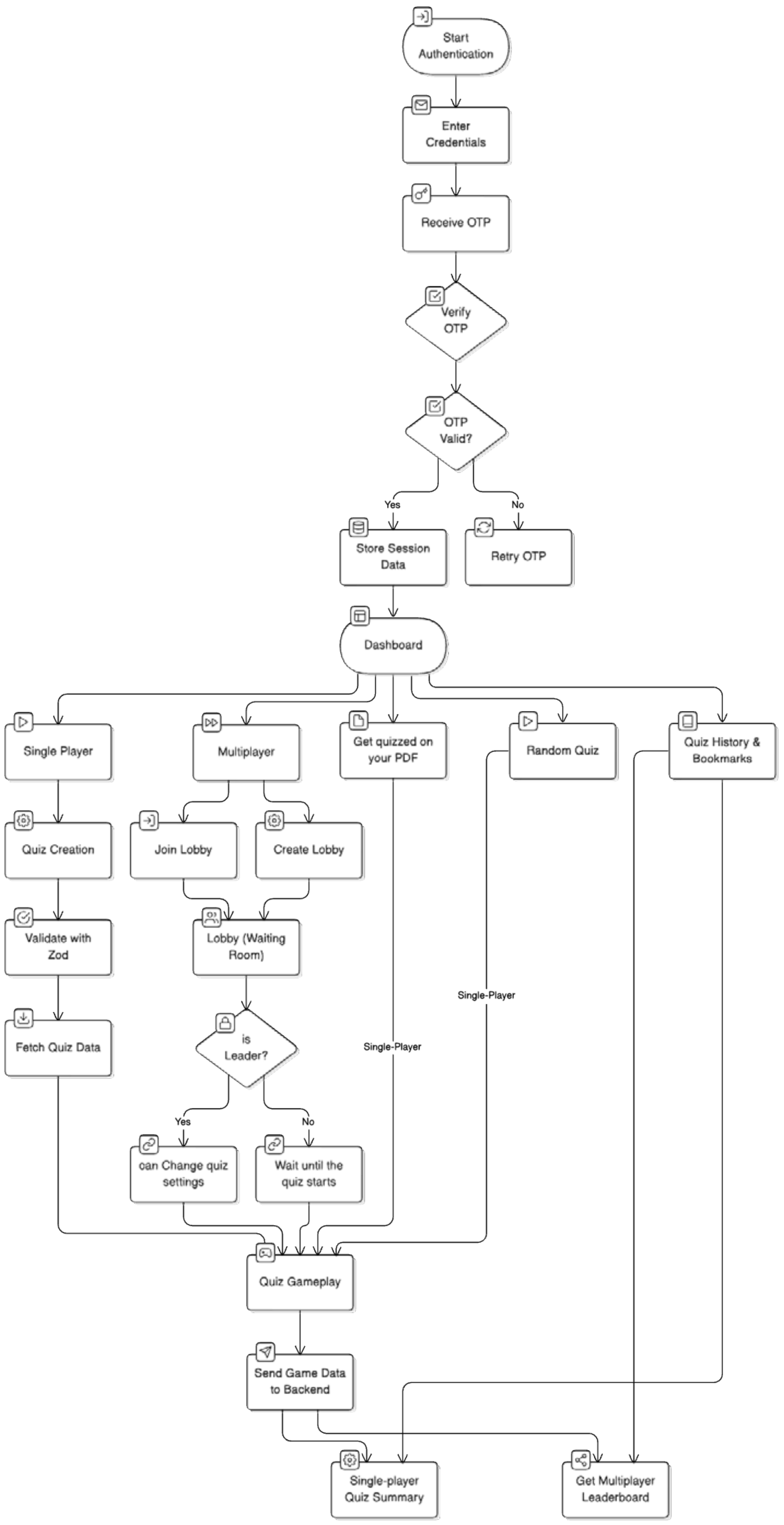


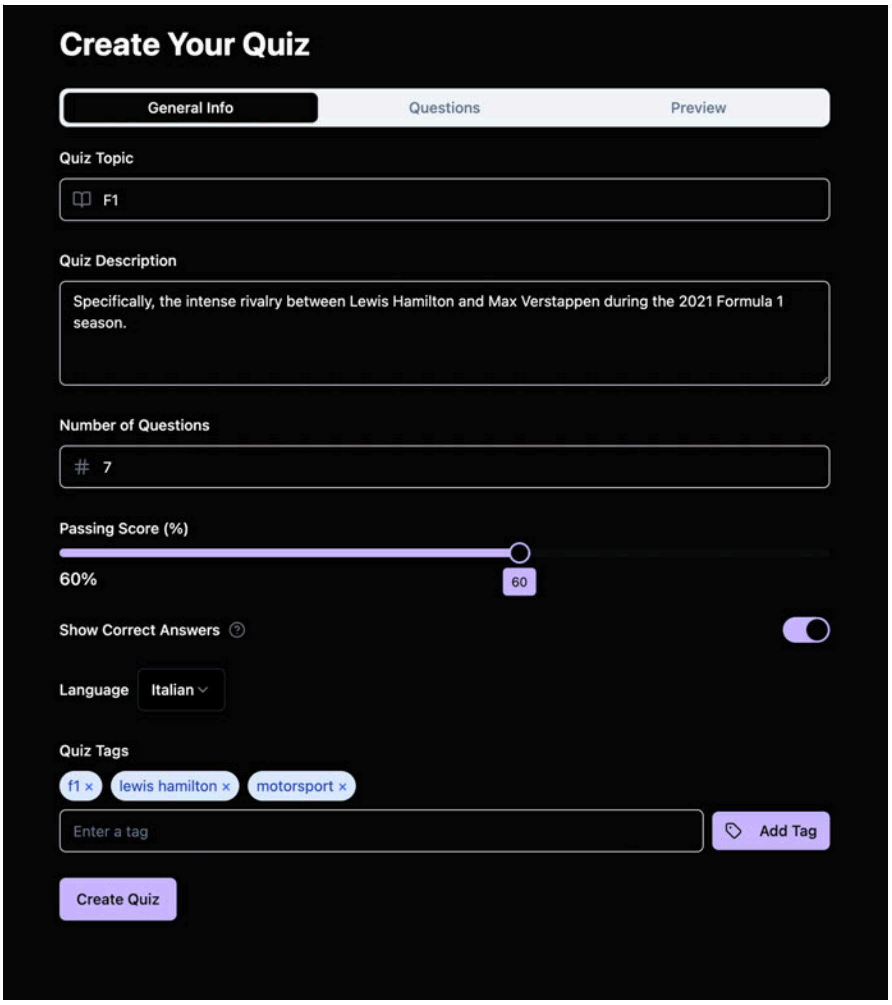
Abbildung 22: Intelliq simplified Frontend Architecture

Users authenticate through an OTP-based login system powered by Supabase, replacing traditional passwords for better security and ease of use. By entering their credentials the users receive an OTP via email. Once verified, session data is stored, allowing users to remain authenticated without repeated logins. If the OTP is invalid, they are prompted to retry. The authentication process follows these steps:

After authentication, users are directed to the dashboard, which serves as the entry point for various features. IntelliQ maintains a persistent sidebar and navigation menu for a seamless experience. Navigation is structured as follows:

1. `Single Player Mode`: Users can start a quiz immediately.
2. `Multiplayer Mode`: Users can either join or create a lobby before starting a game.
3. `Random Quiz`: Users can start a randomly generated quiz on any topic.
4. `Quiz History & Bookmarks`: Past quizzes can be reviewed or bookmarked for later.
5. `Document-Based Quiz`: Users can generate quizzes based on uploaded PDFs.

Single-player mode allows users to configure their quiz experience before starting. Users first choose a topic and provide a brief description that outlines the quiz's purpose. They then set the number of questions, ranging from one to ten, and define a passing score between zero and one hundred percent, which is useful for tracking progress and identifying completed quizzes in the history section. An optional setting enables users to display the correct answer immediately after each question. Users can take quizzes in nine different languages, including `English, German, French, Italian, Polish, Tagalog, Romanian, Spanish, and Serbian`. To organize quizzes efficiently, users must assign at least one tag, though they can add multiple tags for better filtering in the quiz history and bookmarks section. In addition to AI-generated questions, users also have the option to add their own custom questions. This feature enables a more interactive and dynamic experience by allowing users to mix AI-generated content with personalized questions, making the quiz more tailored to their needs. The question input interface ensures that both AI-generated and user-created questions integrate seamlessly within the quiz structure.


Create Your Quiz

General Info | Questions | Preview

Quiz Topic: F1

Quiz Description: Specifically, the intense rivalry between Lewis Hamilton and Max Verstappen during the 2021 Formula 1 season.

Number of Questions: 7

Passing Score (%): 60%

Show Correct Answers: on

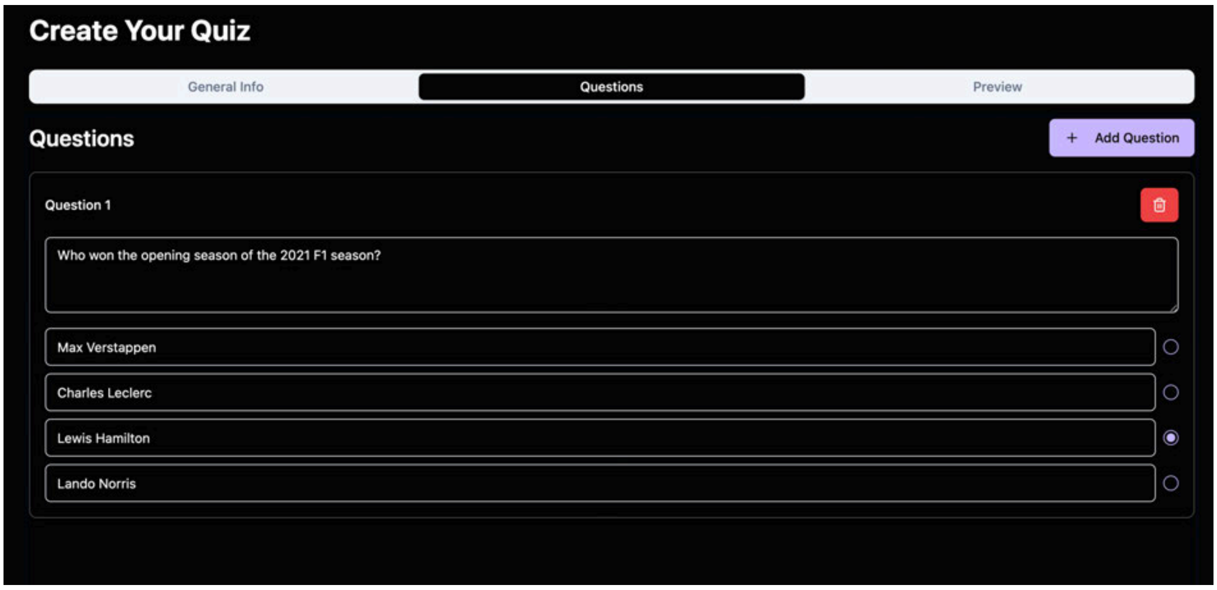
Language: Italian

Quiz Tags: f1 ×, lewis hamilton ×, motorsport ×

Enter a tag | Add Tag

Create Quiz

Abbildung 23: Intelliq single-player quiz creation


Create Your Quiz

General Info | Questions | Preview

Questions | + Add Question

Question 1

Who won the opening season of the 2021 F1 season?

- ○ Max Verstappen
- ○ Charles Leclerc
- ● Lewis Hamilton
- ○ Lando Norris

Abbildung 24: Intelliq single-player own question

Once all settings are configured, the quiz is generated, and users are redirected to the gameplay screen. The quiz follows a multiple-choice format, with a stopwatch tracking the time taken to complete the session. As users progress through the questions, their responses are validated in real time. If the correct answer display option is enabled, the correct answer is shown immediately after each question. At the end of the quiz, a summary page presents the final score, question breakdown, and performance insights, which are stored in the quiz history for future reference.

Multiplayer mode enables real-time quiz engagement by allowing users to either join an existing quiz session or create a new lobby. A multiplayer game requires at least two players and supports a maximum of ten participants. The lobby leader has full control over quiz settings, including language selection, quiz topic, number of questions, player limits, and the time allocated for each question. Throughout the quiz session, only the leader can modify these settings, start the quiz, and control the flow of questions.

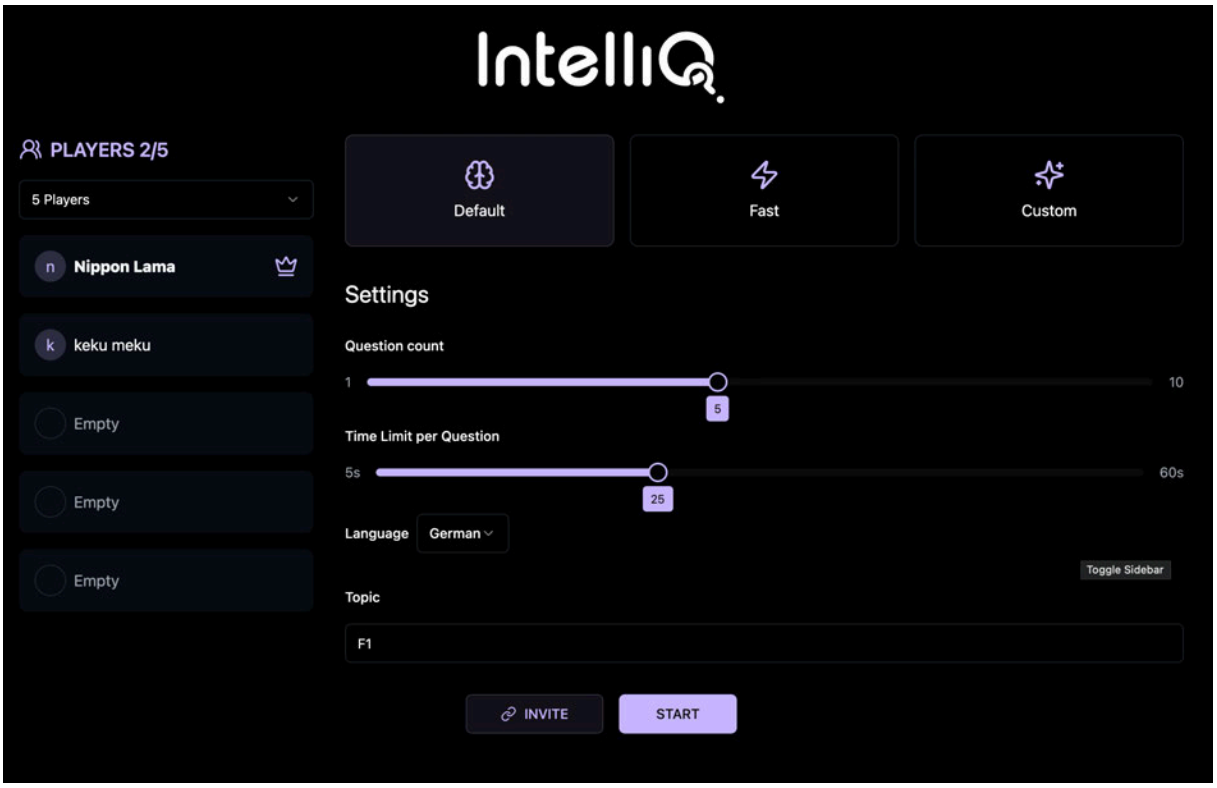


Abbildung 25: Intelliq multiplayer lobby

Once the quiz begins, the gameplay follows the same mechanics as the single-player mode but with the added constraint of a time limit set by the lobby leader. Players must submit answers within the specified timeframe before progressing to the next question. The multiplayer mode synchronizes all players' interactions, ensuring that responses are validated in real time. At the end of the session, a leaderboard ranks participants based on their scores.

The leaderboard provides a transparent performance comparison among players, enhancing engagement and encouraging repeated participation in multiplayer quizzes. All completed quizzes are automatically stored in the Quiz History & Bookmarks section, providing users with a comprehensive record of their past quiz attempts. This feature allows users to revisit previous quizzes at any time and review their performance. Thanks to the structured metadata associated with each quiz, users can efficiently sort through their quiz history based on multiple criteria. Users can filter quizzes by passed or failed status, enabling them to focus on areas that require improvement. Additionally, quizzes are categorized based on tags, allowing users to quickly locate quizzes related to specific topics. IntelliQ also distinguishes between single-player and multiplayer quizzes, making it easier for users to track their performance across different game modes. To further enhance accessibility, users can bookmark specific quizzes for quick access. This functionality is particularly useful for users who want to repeatedly attempt certain quizzes to track their progress over time. Bookmarked quizzes appear in a dedicated section, ensuring they are easily retrievable without needing to search through the full quiz history.

3.5.2 Single-Component Design for Modularity

The idea behind single-component design is based on well-known software development principles that aim to make applications easier to build, manage, and grow. It follows the logic that when each part of an app is small and focused, it becomes easier to reuse, fix, and test. These concepts are not new—they are rooted in software architecture practices that have been used for many years to keep large systems simple and reliable. Theoretical models such as component-oriented programming (COP) further support this approach, arguing that software quality improves when functional units are decoupled and explicitly designed for reuse and adaptability.

Some important design concepts behind single-component architecture include:

1. Modular Programming: Splitting programs into small, self-contained parts.
2. Component-Based Development: Building software with reusable UI blocks.
3. Single Responsibility Principle (SRP): Each component does one job.
4. Atomic Design: UI is built from small elements (atoms) into larger parts.
5. Encapsulation: Hiding internal logic so only the component's output matters.
6. Reusability and Composition: Using components in multiple places and combining them for more complex features.

The IntelliQ-V2 application organizes components in a hierarchical structure:

1. Primitive Components: Basic UI elements like Button, Input, and Card
2. Composite Components: Combinations of primitives that form functional units
3. Page Components: Assemblies of composite components for specific routes
4. Layout Components: Structural components that define the application's visual framework

`Challenge:` Components need to share data and actions with each other, but should still remain independent and reusable.
`Solution:` To manage communication between components, IntelliQ-V2 uses different techniques depending on the situation:

- Context API is used to manage global state that multiple components need to access, such as user settings or quiz data.
- Prop drilling is used for components that are closely related and need to pass data directly, such as a parent component sending information to its child.
- Custom hooks are used to encapsulate shared logic that needs to be reused in different parts of the app.

In the `QAndA` component, props like `quiz`, `questionNumber`, `userAnswer`, `correctAnswer`, and `onAnswerSelected` are passed from a parent. Then, `onAnswerSelected` is passed further into the Answer component. This is essentially prop drilling.

```
type Props = {
  quiz: Quiz[];
  questionNumber: number;
  userAnswer: string | null;
  correctAnswer?: string;
  onAnswerSelected?: (answer: string) => void;
};

const QAndA = ({ quiz, questionNumber, userAnswer, correctAnswer,
onAnswerSelected }: Props) => {

  return (

          answers.map((answer, i) => {
            return (
              <Answer
                key={i}
                answer={answer.slice(3)}
                letter={answer.substring(0, 3)}
                onAnswerSelected={onAnswerSelected}
              ></Answer>
            );
          })
        );
};
```

[17], [18]

3.5.3 Explanation of React Context API

The React Context API is a built-in feature of React that allows developers to share data between components without having to pass props manually through every component in the tree. In traditional React applications, when a parent component needs to pass data to a deeply nested child component, this is usually done through props. However, as the application grows, this method becomes difficult to manage, especially when many components require access to the same data. This is where the Context API becomes useful. This is similar to the `pinia store`, which is a store library for `Vue` that we learned about in our web technology course.

In a Next.js application like IntelliQ, the Context API plays an important role in organizing shared state across different parts of the application. Instead of passing props through multiple components (a process known as prop drilling), context allows us developers to define a provider component that wraps parts of the application and supplies values to any component within that scope. This is particularly helpful in multi-page applications, where global state—such as user information, quiz progress, language preference, or dark mode settings—must be available regardless of which page the user is on.

The Context API has two main parts:

- Provider: Wraps a part of your app and makes data available to all components inside it.
- Consumer (or `useContext` hook): Accesses the shared data in any component inside the Provider.

Example:

```
import { createContext, useContext, useState } from 'react';

const ThemeContext = createContext(null);

export const ThemeProvider = ({ children }) => {
  const [theme, setTheme] = useState('light');
  return (
    <ThemeContext.Provider value={{ theme, setTheme }}>
      {children}
    </ThemeContext.Provider>
  );
};
export const useTheme = () => useContext(ThemeContext);
```

Take `ThemeProvider` as an example: It is wrapped in layout.tsx at the root level of the project so that every component in the project has access to its states, functions, types, etc.

```
import { ThemeProvider } from "@/context/themeContext.tsx";

export default function Layout({ children }: { children: React.ReactNode }) {
  return (
    <ThemeProvider>
      <div className="...">
        <div className="...">
          <main className="...">
            {children}

          </main>
        </div>
      </div>
    </ThemeProvider>
  );
}
```

Now any component can use useTheme() to read or change the theme.

```
//Using destructuring to unpack properties from ThemeContext
const {theme, setTheme} = useTheme();
```

The React Context API is also considered a state management solution. It allows developers to create centralized state holders (contexts) for values that should be accessible in multiple parts of the app. While it does not include advanced features like middleware or side effects out of the box (unlike Redux or Zustand), it is simple, lightweight, and effective for managing global state. For example, in IntelliQ, the quiz state and user authentication status are managed through context providers. Each provider is responsible for one piece of the application's logic. Using context in this way provides several benefits. First, it improves code readability by reducing the number of props that need to be passed between components. Second, it improves performance by allowing developers to control when and how components re-render using memoization and selectors. Third, it supports a more modular architecture, since components only consume the context they need, without depending on unnecessary data or logic.

[19]

Despite its usefulness, the Context API is not always the best solution for every situation. When the application grows in complexity, or when multiple components need to update the same piece of state frequently, other state management libraries may offer better performance and flexibility.

Some popular alternatives to the React Context API include:

- `Zustand`: A minimal and fast state management library based on hooks. It allows developers to create small global stores that are easy to use and scale. Zustand does not require any context providers or complex configuration. One of its core strengths is that components can subscribe only to the parts of the state they use, which improves performance by reducing unnecessary re-renders. Unlike React Context API, Zustand simplifies state access and modification without needing prop drilling or nested providers. It also supports middleware, persistent state (e.g., saving state to local storage), and is compatible with server-side rendering in Next.js.
- `Jotai`: It takes a different approach by using the concept of atoms to manage state. Each atom represents a piece of state and can be used independently across components. Jotai promotes fine-grained control over state, which helps prevent large re-renders across the application. Components that use different atoms only re-render when their specific atom changes. This is a major difference from the Context API, where a single provider update can cause many child components to re-render. Jotai also supports asynchronous atoms, allowing it to manage async data directly without the need for middleware.
- `Redux Toolkit`: It uses a central store to keep all state in one place. It also uses actions and reducers to update the state in a clear and predictable way. Redux Toolkit reduces the amount of setup required compared to older versions of Redux. It also includes great tools for debugging and checking how state changes over time. However, it can still be more complex than Zustand or Jotai, and it’s better suited for apps with many features and a big development team. Compared to Context, Redux is more powerful but takes more time to set up.
- `Recoil`: A library from Meta (Facebook) that uses atoms and selectors to manage state. It’s similar to Jotai but offers more control over how different pieces of state relate to each other. It also works well with React’s Suspense, which helps load data in a smoother way. Recoil keeps track of which components use which atoms, so it updates only what’s necessary. This makes it more efficient than Context for apps with lots of changing data or complex logic.

[20]

3.5.4 Why Redux is not Suitable

Redux is a powerful state management library, widely used in large-scale applications. It introduces concepts such as global stores, reducers, and actions to manage state in a predictable and centralized way. However, in the case of IntelliQ, using Redux would introduce unnecessary complexity and overhead.

IntelliQ uses the React Context API together with the useReducer and follows a clean domain-driven design where each context handles a specific feature area. This approach works well for the app's structure because each major section (like quiz logic, multiplayer, quiz creation, and authentication) is already separated into its own context provider. Each provider only manages the state relevant to its function and exposes helper functions or actions where needed. Redux would introduce unnecessary complexity.

Using Redux would mean:

- Defining global actions and reducers for every state update (e.g., quiz logic, multiplayer setup, submission, leaderboard). It would force the project to flatten this into a single store, making it harder to maintain the clear separation of concerns.
- Writing boilerplate for every interaction (actionTypes, dispatch, connect, etc.)
- Adding middleware like Redux Thunk or Saga for handling async calls such as fetching quiz data, submitting answers, or syncing multiplayer data

Example: In the current QuizContext, we use useReducer and createContext to locally manage the quiz state. The quizReducer handles quiz-specific logic such as:

- `FETCH_QUIZ_REQUEST`
- `FETCH_QUIZ_SUCCESS`
- `SUBMIT_QUIZ_SUCCESS`
- `FETCH_LEADERBOARD_SUCCESS`

```
import type { QuizAction, QuizContextValue } from '@/contexts/quiz-context';

export const quizReducer = (state: QuizContextValue, action: QuizAction):
QuizContextValue => {
  if (action.type === 'FETCH_QUIZ_REQUEST') {
    return { ...state, isLoading: true };
  }

  if (action.type === 'FETCH_QUIZ_SUCCESS') {
    return {
      ...state,
      isLoading: false,
      fetchingFinished: true,
      currentQuiz: action.payload,
      summaryQuiz: null,
    };
  }
 ...
  return state;
};
```

This reducer is scoped to the quiz context and doesn't interfere with multiplayer state or quiz logic state. If Redux were used instead, we would have to:

- Combine all reducers using combineReducers
- Each component would need to dispatch global actions and select state using hooks or connect.
- Quiz, Multiplayer, Auth, Settings, etc. would all live in one shared global store, which can become hard to manage for IntelliQ where state is mostly local to each flow.

[21], [22]

3.5.5 User Context Management

The user context is the central hub for managing user authentication and profile data throughout the application. This structure is used consistently throughout the application to represent user data.

```
export interface User {
  id: string;
  email: string;
  img: string;
  name: string;
  avatar: string;
}
```

Key Features:

1. User State Management: Maintains the current user's information including ID, email, name, and avatar.
2. Authentication Methods: Supports multiple authentication methods:
    - OTP (One-Time Password) authentication
    - OAuth (Google) authentication
3. Profile Management: Allows updating user profile information.

`signInWithOTP(userInput, isNewUser)` : Handles login or registration using email-based OTP (one-time password).

- Calls Supabase's `signInWithOtp()` method.
- Uses `isNewUser` to decide whether to create a new account or log in.

```
const { data, error } = await supabase.auth.signInWithOtp({
  email: userInput.email,
  options: {
    shouldCreateUser: isNewUser,
    data: {
      email: userInput.email,
      name: `${userInput.firstName} ${userInput.lastName}`,
    },
  },
});
```

`signinUsingOAuth()` :Handles signing in with third-party OAuth providers like Google.

- Uses `supabase.auth.signInWithOAuth({ provider: "google" })`.
- Uses `isNewUser` to decide whether to create a new account or log in.

```
await supabase.auth.signInWithOAuth({
  provider: "google",
  // only for development
  options: { redirectTo: "http://localhost:3000/dashboard" },
});
```

`signout()` :Logs the user out and resets the user state.

- Calls supabase.auth.signOut()
- Navigates back to /login route
- Clears loading and OTP flags

```
const signout = async () => {
    try {
      await supabase.auth.signOut();
      setOtpSent(false);
      setIsLoading(false);
      toast({
        title: "User signed out",
        description: "You have been signed out",
      });
      router.push("/login");
    } catch (error) {
      console.log(error);
    }
  };
```

`getUserInfo()` :Fetches user info from the Supabase session when the app loads.

- Called inside a useEffect() when the provider is mounted.
- Extracts metadata like avatar, name, and email from the session.

```
const getUserInfo = async () => {
    const {data: { session },error} = await supabase.auth.getSession();
    if (!session) {
      return;
    }
  const user = session?.user;
  setCurrentUser({
    id: user.id,
    email: user.email,
    img: user.user_metadata.avatar_url,
    name: user.user_metadata.name,
    avatar: user.user_metadata.avatar_url,
});
  };
```

Internal State Flags

- `otpSent`: `true` if OTP was sent successfully to the user's E-mail.
- `isLoading`: `true` while waiting for async auth operations.
- `isNewUser`: `true` if the user is registering, not logging in.

These values help control UI feedback like loading spinners or showing/hiding input fields.

The `NavUser` component uses `useAuth` to display the user's avatar and name in the sidebar, and provides access to user settings and sign-out functionality. It prioritizes the current user data from the user context, ensuring that the most up-to-date user information is displayed.:

```
export function NavUser({ user, isNavbar = false }) {
  const { signout, updateUserProfile, currentUser } = useAuth();

  // Use currentUser.img for the avatar if available, otherwise fall back
to prop
  const currentAvatar = currentUser?.img || user.avatar;
  const currentName = currentUser?.name || user.name;

  // Component renders user avatar and name in the UI
  // ...

  // Provides sign out functionality
  <DropdownMenuItem onClick={signout}>
    <LogOut />
    Log out
  </DropdownMenuItem>
}
```

3.5.6 Quiz Context Implementation

The `QuizContext` is a global state management system designed to handle quizzes in different formats (singleplayer, multiplayer, document-based) for the IntelliQ app. It manages quiz data, user interaction, language settings, and score tracking across the app using React Context and useReducer.

The context uses React's `useReducer` hook to manage complex state transitions in a predictable way. This is particularly important for quizzes because they involve multiple states: loading, active, completed, and error states. The reducer pattern ensures that all state changes are handled consistently and can be easily debugged.

For example, when a user starts a quiz, the context transitions through several states:

1. Initially sets `isLoading` to true
2. Fetches the quiz questions from the API
3. Updates the state with the fetched questions
4. Sets `isLoading` to false and `fetchingFinished` to true

The app supports multilingual quizzes with the SupportedLanguages enum. When generating quizzes, the language is included in the API request, making it easier to serve localized content.

```
export enum SupportedLanguages {
  English = 'en',
  German = 'de',
  French = 'fr',
  Spanish = 'es',
  Italian = 'it',
  Romanian = 'ro',
  Serbian = 'sr',
  Tagalog = 'tl',
  Polish = 'pl',
}
```

A reducer is a pure function that takes in the current state and an action (with a type and optional payload) and returns a new state.

```
const [state, dispatch] = useReducer(quizReducer, initialState);
```

The reducer connects to the app through the QuizProvider. Inside that provider:

- All quiz actions (fetching, submitting, resetting, etc.) are dispatched to the reducer.
- The reducer updates the state, which is then shared with all components using the useQuiz() hook.

Example:

```
dispatch({ type: 'FETCH_QUIZ_SUCCESS' }); //
```

Then inside the reducer:

```
if (action.type === 'FETCH_QUIZ_SUCCESS') {
    return {
      ...state,
      isLoading: false,
      fetchingFinished: true,
      currentQuiz: action.payload,
      summaryQuiz: null,
    };
  }
```

This change triggers a re-render and updates relevant states.

The `CurrentQuiz` interface is the core data structure that represents an active quiz session in IntelliQ. It contains all the essential information needed to manage a quiz, including the questions themselves, the topic, language settings, and scoring criteria. This interface serves as the central data model for quiz functionality, allowing IntelliQ to track what questions are being presented to users, how they should be displayed, and what constitutes a passing performance. The optional properties like `quizId`, `documentId`, and `language` make the interface flexible enough to handle different types of quizzes (standard, document-based, or multiplayer) while maintaining a consistent structure.

`Quiz` Structure:

```
// The Quiz interface represents a single question
export interface Quiz {
  correctAnswer?: string;
  options: string[];
  text: string;
  questionTitle: string;
  id?: string;
}
// ? represents an optional property
interface CurrentQuiz {
  quizId?: string;
  quiz: Quiz[];
  topic: string;
  showCorrectAnswers: boolean;
  documentId?: string;
  language?: SupportedLanguages;
  passingScore?: number;
}
```

`QuizHistory` Structure: This represents the complete history of a quiz attempt, including scores, timing and user's answers.

```
interface HistoryQuestions {
  correctAnswer: string;
  text: string;
  userAnswer: string;
}
export interface QuizHistory {
  quizId: string;
  quizTitle: string;
  quizScore: number;
  totalTime: number;
  correctAnswersCount: number;
  totalQuestions: number;
  passingScore: number;
  questions: HistoryQuestions[];
}
```

[23]

The `fetchQuestions` Function: The `fetchQuestions` function is a versatile function that handles both single-player and multiplayer quiz generation. It adapts its behavior based on the `quizType` parameter, creating different API requests and handling the responses accordingly.

The function accepts two parameters:

- `userQuizData`: Contains quiz configuration (topic, number of questions, language, etc.). This represents the user's request for single-player or multiplayer quiz.
- `roomId`: Optional parameter for multiplayer quizzes

```
const fetchQuestions = async (userQuizData: QuizData, roomId?: string) => {
...
}
```

The function creates different API query parameters based on the quiz type. This conditional logic shows that:

- Multiplayer quizzes require fewer parameters (no description or tags)
- Single-player quizzes include additional metadata (description and tags)

```
const query =
  quizType === 'multiplayer'
    ? {
        quizTopic: interests,
        numberOfQuestions: numberOfQuestions.toString(),
        language,
        quizType,
      }
    : {
        quizTopic: interests,
        quizDescription: userQuizData.description!,
        numberOfQuestions: numberOfQuestions.toString(),
        quizTags: userQuizData.tags,
        language,
        quizType,
      };
```

For multiplayer quizzes, the function performs additional steps:

```
if (quizType === 'multiplayer' && roomId) {
  const multiplayerQuiz = await submitMultiplayerQuiz(roomId, quiz, interests,
language);
  if (multiplayerQuiz && 'questions' in multiplayerQuiz) {
    quiz.quiz = multiplayerQuiz.questions as Quiz[];
    quiz.quizId = multiplayerQuiz.quizId;
  }
}
```

It Calls `submitMultiplayerQuiz` to register the quiz in the database and updates the quiz with the questions and ID returned from the database. This ensures all players in the same room get the same questions

There are several additional functions in the `QuizContext`. Their purpose is clear from their names, and each one plays a role in managing the quiz lifecycle:

- `submitSinglePlayerQuiz`: submits a completed singleplayer quiz to generate summary.
- `getLeaderboard`: fetches the current leaderboard for a multiplayer room.
- `getSinglePlayerSummary`: retrieves the summary of a previously completed quiz.
- `getMultiplayerQuizForPlayers`: used by players in a multiplayer quiz lobby to fetch the quiz that was previously stored in the database by the lobby leader.

3.5.7 Quiz Logic Context Implementation

The `QuizLogicContext` is a specialized context that works in conjunction with the Quiz Context to handle the interactive aspects of quiz-taking. While the Quiz Context manages the overall quiz data and state, the Quiz Logic Context focuses on the user's interaction with the quiz, including answer validation, progress tracking, and feedback mechanisms.

One of the most important aspects of the Quiz Context is how it tracks user interactions. Every answer a user provides is carefully recorded, including:

- The time taken to answer each question
- The selected answer
- Whether the answer was correct or incorrect
- The overall quiz performance

This state variable tracks which question the user is currently viewing, allowing components to display the appropriate question and options. The `quiz` property in the parent context `QuizContext` contains all questions for the quiz, which is an array. Proceeding to the next question is basically an increment of the `questionNumber` state, which then shows the next index question in the array.

```
const [questionNumber, setQuestionNumber] = useState(0);
```

This state creates a percentage value that can be used for progress bars or other visual indicators of quiz completion.

```
const [progressValue, setProgressValue] = useState<number>(
  currentQuiz ? ((questionNumber + 1) / currentQuiz.quiz.length) * 100 : 0,
);
```

One of the most important functions of this context is validating user answers. This is handled through its own reducer pattern. This action is called when the user moves on to the next question, or when the timer runs out in a multiplayer quiz.

The reducer:

- Compares the user's answer with the correct answer
- Increments either the correct or incorrect counter
- Adds the question, correct answer, and user's answer to the `userAnswer` array for tracking purposes.

```
// In the reducer
if (action.type === 'VALIDATE_ANSWER') {
  const { correctAnswer, userAnswer, question } = action.payload;
  let scoreCORRECT = state.correctAnswer;
  let scoreINCORRECT = state.wrongAnswer;
  if (userAnswer !== correctAnswer) {
    scoreINCORRECT += 1;
  } else {
    scoreCORRECT += 1;
  }
  return {
    ...state,
    correctAnswer: scoreCORRECT,
    wrongAnswer: scoreINCORRECT,
    userAnswer: [...state.userAnswer, { question, correctAnswer, userAnswer }],
  };
}
```

The context also provides immediate feedback to users through toast notifications. It displays when the user answers correctly or incorrectly, but only in single-player mode.

```
useEffect(() => {
  if (state.correctAnswer > 0 && !isMultiplayer) {
    showToast('success', 'CORRECT!', "You've answered the question correctly");
  }
}, [state.correctAnswer]);

useEffect(() => {
  if (state.wrongAnswer > 0 && !isMultiplayer) {
      showToast('destructive', 'INCORRECT!', "Oops, that's not the correct
answer. Keep trying!");
  }
}, [state.wrongAnswer]);
```

3.6 Frontend Features

IntelliQ now supports multiplayer mode, document-based quiz, user customization, detailed quiz history with filtering and bookmarking and random quiz.

3.6.1 User Customization

Ausgearbeitet von Ricky Raveanu

The IntelliQ platform implements a comprehensive user customization system that allows users to personalize their experience. This section details the technical implementation of the settings interface and the underlying mechanisms that enable persistent user preferences across devices.

3.6.1.1 Settings Architecture

The user customization system is built with three key components:

User Profile Management:

- Avatar selection with categorized options
- Display name personalization
- Email display for account verification

Theme System:

- Light and dark mode support
- Theme persistence across sessions
- Animated transitions between themes

Preference Controls:

- Sound settings with toggle controls
- Visual effects preferences
- Accessibility options

3.6.1.2 Technical Implementation

The settings functionality is implemented through a combination of React hooks, context providers, and efficient state management. The system leverages Supabase's authentication and storage mechanisms for profile data, while client-side preferences are stored using localStorage.

1. Profile State Management [20]

```
const { updateUserProfile, currentUser } = useAuth();

const handleSaveProfile = async (formData: SettingsFormValues) => {
  try {
    setIsSavingProfile(true);
    await updateUserProfile({
      name: formData.name,
      avatar: avatar,
    });
    toast({
      title: "Profile saved",
      description: "Your profile has been updated.",
    });
  } catch (error) {
    console.error("Error updating profile:", error);
    toast({
      title: "Error updating profile",
      description: "Please try again.",
      variant: "destructive",
    });
  } finally {
    setIsSavingProfile(false);
  }
};
```

2. User Preference Storage: [24]

   Local preferences are persisted using the `useLocalStorage` hook, ensuring that user settings remain consistent across sessions:

```
const [soundEnabled, setSoundEnabled] = useLocalStorage<boolean>(
  "soundEnabled",
  true,
);

const       [particlesEnabled,       setParticlesEnabled]       =
useLocalStorage<boolean>(
  "particlesEnabled",
  true,
);
```

3.6.1.3 Avatar Management System

The platform includes a comprehensive avatar selection system that leverages Supabase Storage for organization and delivery:

1. Avatar Categorization:
   - Avatars organized into distinct style categories
   - Efficient loading through category-based tabs
   - Preview with current selection highlighted
2. Supabase Storage Implementation: [14]

```
const fetchAvatarsByFolder = async (folder: string) => {
  const { data, error } = await supabase.storage
    .from("avatars")
    .list(folder, { limit: 12 });

  if (error) {
    console.error(`Error fetching avatars from ${folder}:`, error);
    return [];
  }

  const urls = await Promise.all(
    data
      .filter((file) => file.name.endsWith(".png"))
      .map(async (file) => {
        const { data: urlData } = await supabase.storage
          .from("avatars")
          .getPublicUrl(`${folder}/${file.name}`);
        return urlData?.publicUrl;
      }),
  );

  return urls.filter(Boolean) as string[];
};
```

3. Progressive Loading: [15]

   The avatar system implements progressive loading with skeleton placeholders to provide a smooth user experience even on slower connections:

```
{initialLoading ? (
  <div className="...">
    {Array.from({ length: 12 }).map((_, i) => (
      <div key={i} className="...">
        <Skeleton className="..." />
      </div>
    ))}
  </div>
) : (
  <div className="...">
    {urls.map((url) => {
      const isSelected = avatar === url;
      return (
        <div
          key={url}
          className="..."
          onClick={() => setAvatar(url)}
        >
          <Avatar
            className={`... ${
              isSelected
                ? "..."
                : "..."
            }`}
          >
            <AvatarImage src={url} alt="Avatar option" />
          </Avatar>
        </div>
      );
    })}
  </div>
)}
```

3.6.1.4 Implementation Considerations

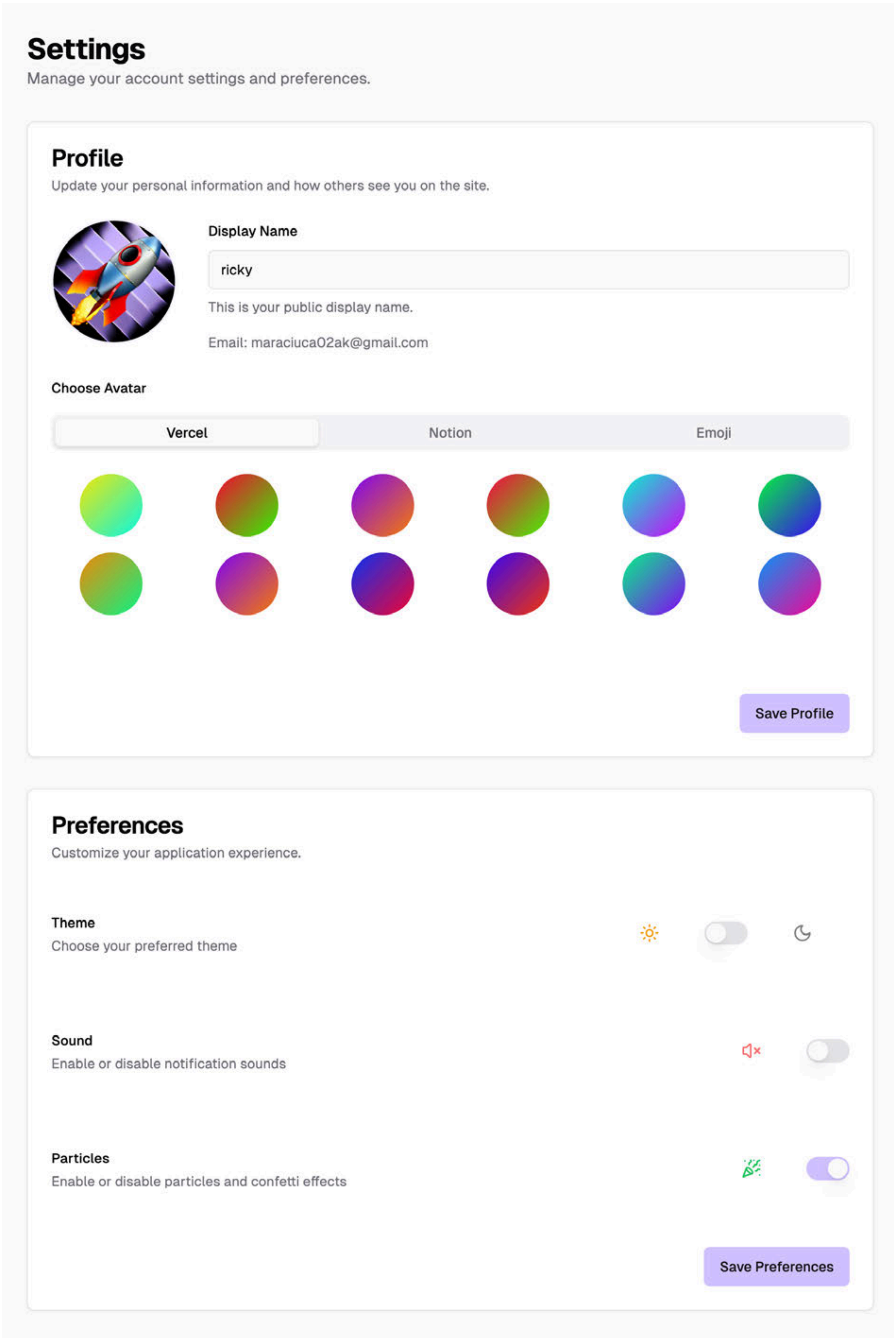
During the development of the user customization system, several factors were considered to ensure optimal performance and user experience:

1. Performance Optimization:
    - Lazy loading of avatar images
    - Efficient state updates to prevent unnecessary re-renders
    - Debounced event handlers for frequent user interactions
2. Multi-Modal Interface: The settings interface is implemented in two variants:
    - Full-page settings view for comprehensive customization
    - Dialog-based quick settings for immediate adjustments

This dual approach provides flexibility while maintaining consistency in the user experience.

3. Security Considerations:
    - User-specific avatar storage paths
    - Authentication checks before profile updates
    - Sanitized inputs for user-provided data

The user customization system exemplifies IntelliQ’s commitment to providing a personalized, accessible experience while maintaining technical excellence and performance optimization.



Settings

Manage your account settings and preferences.

Profile

Update your personal information and how others see you on the site.

Display Name

ricky

This is your public display name.

Email: maraciuca02ak@gmail.com

Choose Avatar

Preferences

Customize your application experience.

Theme

Choose your preferred theme

Sound

Enable or disable notification sounds

Particles

Enable or disable particles and confetti effects

Abbildung 26: IntelliQ Settings Interface

3.6.2 Random Quiz Generation

Ausgearbeitet von Nikola Petrovic

By selecting the Random Quiz card, a new quiz is generated with random questions from a diverse range of topics. This feature provides a unique and exciting quiz experience, as users never know what topic they'll be quizzed on next.

Generating Random Quizzes:

- When users click the Random card on the dashboard, the system automatically generates a quiz with random topics, offering a surprise every time.

Surprise and Variety:

- Each time the Random Quiz is triggered, users are presented with a new set of questions from varied topics, ensuring a fresh and engaging experience with each attempt.

3.6.3 Document-Based Quiz Generation

Ausgearbeitet von Ricky Raveanu

While the backend architecture handles document processing and quiz generation, the frontend implementation provides an intuitive interface for users to upload, manage, and generate quizzes from their documents. This section explores the user experience and frontend components that power the document-based quiz feature.

3.6.3.1 Document Management Interface

The document management interface serves as the central hub for users to interact with their uploaded content:

1. Document Dashboard:
    - Tabbed navigation for different document views (Recent, All, Most Quizzed)
    - Responsive grid layout for document cards
    - Search functionality with real-time filtering
    - Status indicators for document processing stages

The dashboard implementation leverages React's state management to handle document filtering and display:

```
// Tabs implementation for document organization
<Tabs
  defaultValue="recent"
  className="..."
  onValueChange={setActiveTab}
>
  <TabsList className="...">
    <TabsTrigger value="recent">Recent</TabsTrigger>
    <TabsTrigger value="all">All Documents</TabsTrigger>
    <TabsTrigger value="quizzed">Most Quizzed</TabsTrigger>
  </TabsList>

  {/* Tab content implementation */}
  <TabsContent value="recent">
    {/* Recent documents display */}
  </TabsContent>
</Tabs>
```

2. Document Card Component:
    - Visual status indicators for processing stages
    - Direct quiz generation action buttons
    - Document metadata display (size, upload date)
    - Usage statistics showing quiz count

3.6.3.2 File Upload Experience

The file upload component provides an intuitive drag-and-drop interface for document submission:

1. User Interaction Features:
    - Drag-and-drop zone with visual feedback
    - File type validation for supported formats
    - Progress tracking during upload
    - Error handling with user-friendly messages
2. Progressive Feedback:
    - Real-time upload progress indicator
    - Visual confirmation of successful upload
    - Automatic processing status updates
    - Transition to document library upon completion

The upload component incorporates accessibility considerations and intuitive drag-and-drop functionality:

```
// Drag and drop implementation with visual feedback
<div
  className={`... ${
    dragActive
      ? "border-primary bg-primary/5"
      : "border-muted-foreground/25"
  }`}
  onDragEnter={handleDrag}
  onDragLeave={handleDrag}
  onDragOver={handleDrag}
  onDrop={handleDrop}
>
  {/* Upload content */}
</div>
```

3.6.3.3 Processing Status Visualization

The system provides clear visual feedback on document processing stages:

1. Status Indicators:
   - Processing stage labels (Extracting Text, Chunking, etc.)
   - Animated loading indicators
   - Color-coded status representation
   - Error state handling with recovery options
2. Processing Workflow Integration:
   - Automatic polling for status updates
   - Graceful handling of processing delays
   - Transparent feedback on processing stages
   - Smooth transition to quiz generation when ready

The status visualization is implemented with a combination of state management and UI components:

```
// Processing status button implementation
const getProcessingButton = () => {
  if (!document.processingStatus || document.processingStatus === "completed")
{
    return {
      text: (
        <>
          Start Quiz
          <ArrowUpRight className="..." />
        </>
      ),
      variant: "default",
      disabled: false,
    };
  }
  const statusText = {
    pending: "Processing",
    extracting_text: "Extracting Text",
    chunking: "Chunking",
    embedding: "Embedding",
  };
  return {
    text: (
      <>
        {statusText[document.processingStatus]}
        <Loader2 className="..." />
      </>
    ),
    variant: "outline",
    disabled: true,
  };
};
```

3.6.3.4 Quiz Generation Interface

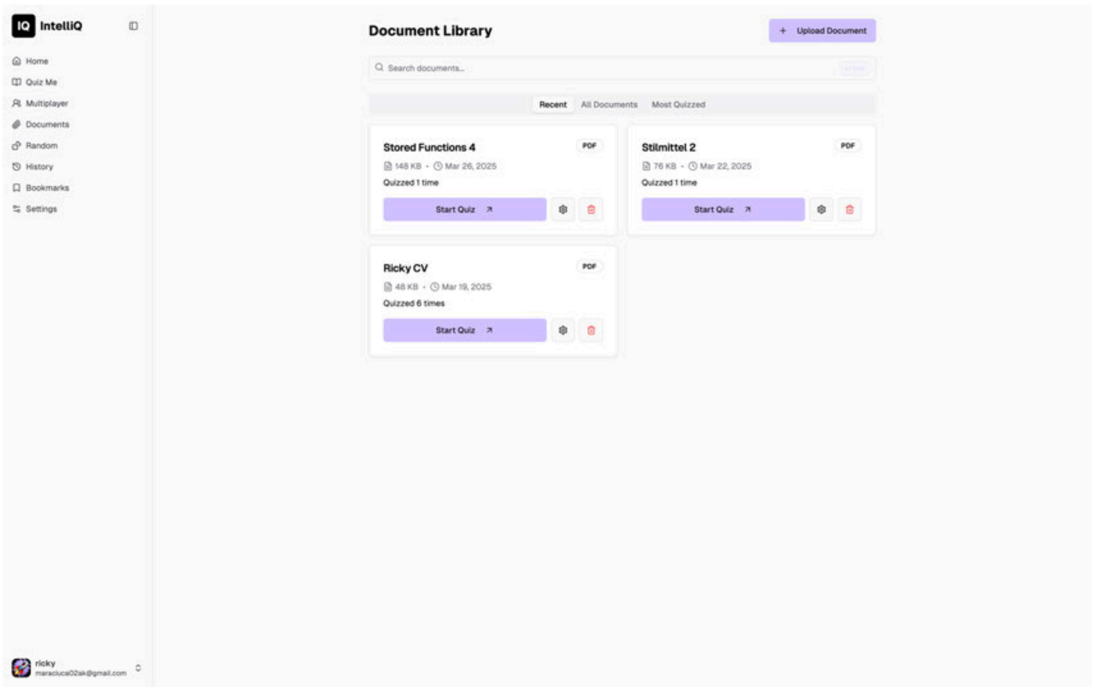
Once a document is processed, the system provides an intuitive interface for quiz generation:

1. Streamlined Quiz Creation:
    - One-click quiz generation with default settings
    - Advanced customization options for tailored quizzes
    - Language selection for multilingual support
    - Question count adjustment based on document length
2. User Feedback During Generation:
    - Animated loading indicators during quiz creation
    - Progress updates for lengthy generation processes
    - Smooth transition to quiz interface when ready
    - Error handling with actionable recovery options

The system incorporates optimistic UI patterns to ensure a responsive experience: [19]

```
// Quiz generation with loading state
const startQuizOnDocument = async (documentId: string) => {
  try {
    setIsGenerating(true);
    if (quizContext?.dispatch) {
      quizContext.dispatch({ type: "RESET_QUIZ" });
    }
    if (quizContext?.fetchDocumentQuestions) {
      const quizData = {
        documentId,
        number: 5,
        quizLanguage: "en" as SupportedLanguages,
        showCorrectAnswers: true,
        passingScore: 50,
        quizType: QuizType.Enum.document,
        questions: [],
        topic: document.title || "Document Quiz",
        description: "Quiz generated from document",
        tags: ["document"],
      };
      await quizContext.fetchDocumentQuestions(quizData);
      router.push("/single-player/quiz/play");
    }
  } catch (error) {
    // Error handling
  } finally {
    setIsGenerating(false);
  }
};
```

The document-based quiz generation frontend provides an intuitive, accessible interface that guides users through the process of transforming static documents into interactive learning experiences. The implementation balances simplicity with powerful customization options, allowing users to quickly generate relevant quizzes from their uploaded content.



Stored Functions 4
148 KB · Mar 26, 2025
Quizzed 1 time

Stilmittel 2
76 KB · Mar 22, 2025
Quizzed 1 time

Ricky CV
48 KB · Mar 19, 2025
Quizzed 6 times

Abbildung 27: IntelliQ Document Dashboard Interface

3.6.4 Auditory Feedback Integration

Ausgearbeitet von Nikola Petrovic

In the system, auditory feedback is designed to enhance the user experience with specific sound cues tied to certain actions:

- Correct/incorrect answer sounds: Clear and distinct sounds that notify the user whether their answer was right or wrong
- Multiplayer soundtrack: A soundtrack plays only when the leaderboard is displayed in multiplayer mode, adding an immersive audio experience during the showcase of the best

3.6.5 Quiz Sharing Functionality

The quiz sharing feature in allows users to easily share their quiz results with others via a shareable link. This feature is integrated into both the History Page and Bookmarks Page, with a share button present on each quiz entry.

- The system generates and stores unique URLs linked to the quiz summary in the backend
- The quiz data is fetched dynamically when the link is accessed by the recipient
- Depending on the sender's privacy choice, the backend adjusts the visibility of the sender's identity

Share Button

- Each quiz entry has a Share Quiz button
- When clicked, the user is prompted with a modal to decide if the recipient should know their identity
- The user can choose to share the quiz anonymously by checking the „Share anonymously (hide your name)" option

Generated Shareable Link

- Once the user makes their selection, they can click Generate share link
- A unique URL is generated and can be copied and shared with others
- The recipient can use this link to view the quiz summary page

Recipient View

- The recipient opens the link in their browser and sees the quiz summary
- Depending on the sender's choice (anonymous or not), the recipient may or may not see the sender's name

Summary Page Details

- The quiz summary page includes the user's score, time taken, correct answers, topic and tags associated with the quiz, and the passing score required to pass the quiz

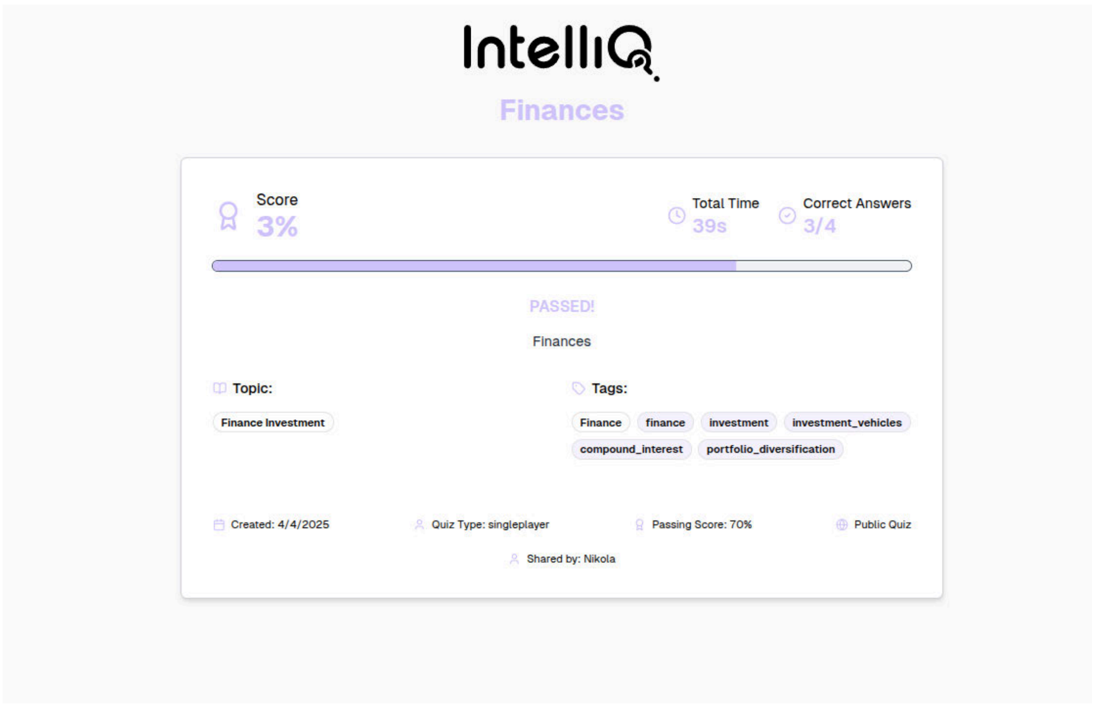


Abbildung 28: IntelliQ Shared Quiz Summary

3.6.6 Bookmarking and Quiz History Management

The bookmarking and quiz history management system is a crucial feature of the platform, allowing users to efficiently manage their completed quizzes by saving and accessing them later. This functionality is primarily supported by the History Page and Bookmarks Page, which work together to provide a seamless user experience.

3.6.6.1 History Management

The History page provides a comprehensive interface for viewing and managing previously played quizzes.

Core Components:

- Main history container with pagination
- Interactive filter system with real-time updates
- Individual history cards with bookmarking and sharing
- Skeleton loading states for smooth UX

State Management Integration:

- Centralized quiz history state
- Loading and error states

Interactive Filtering System:

The filtering system provides dynamic search and categorization capabilities: Filter Categories:

- Quiz type selection (practice, multiplayer)
- Status filtering (passed, failed)
- Tag-based categorization

User Interface Features:

- Animated filter transitions
- Real-time filter updates
- Clear visual feedback
- Accessible filter controls

History Card Visualization:

Each quiz attempt is represented by an interactive card component that provides users with insights into the following:

- Score and performance metrics
- Interactive bookmarking functionality
- Time and date information
- Status indicators and badges

Interactive Elements:

- Animated bookmark toggling
- Click to get detailed summary
- Quiz Sharing

3.6.6.2 Bookmarking Management

Ausgearbeitet von Nikola Petrovic

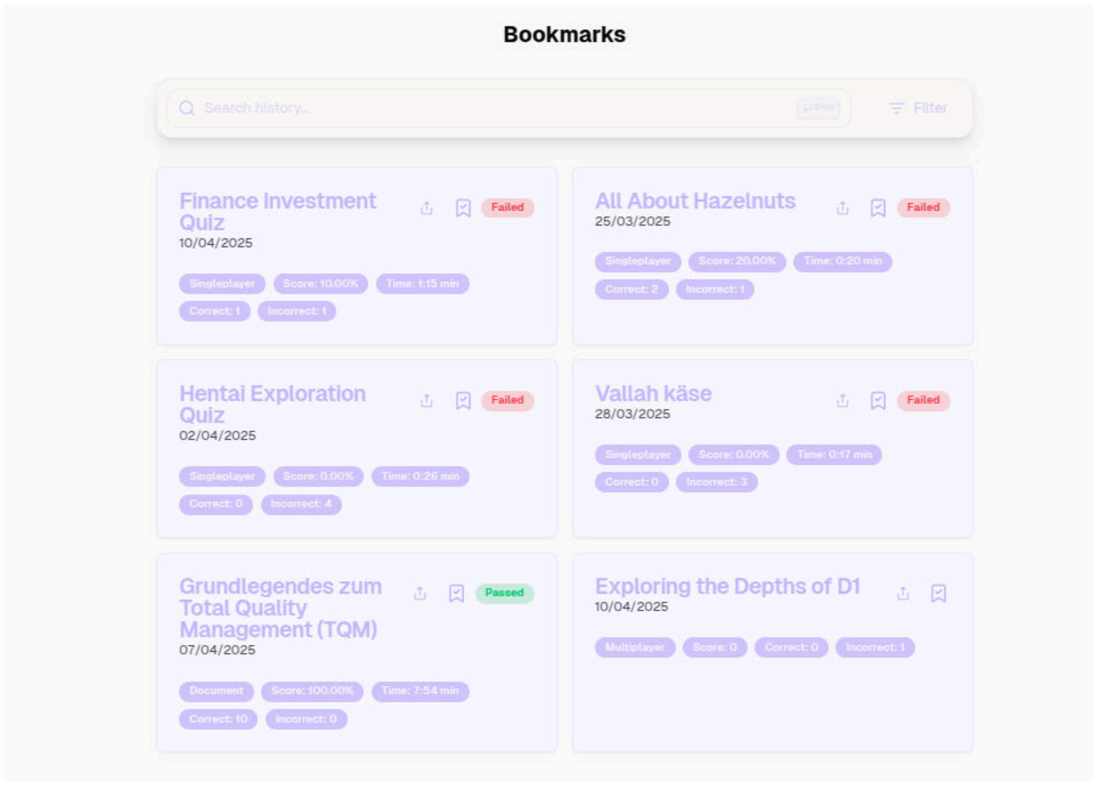
The bookmarking feature allows users to save quizzes they want to revisit later. By pressing the Bookmark button on any quiz entry, users can easily store their preferred quizzes in the Bookmarks Page for quick access. This functionality offers a convenient way to manage and organize quizzes based on personal preference.

Saving Quizzes:

- Each quiz entry on the History Page includes a Bookmark button. When pressed, this button saves the quiz to the Bookmarks Page, allowing users to easily access their saved quizzes later.

Real-Time Feedback:

- The button's status updates immediately, showing whether the quiz has been successfully bookmarked. Users can toggle the bookmark status on or off based on their preference, with changes reflected instantly.



Finance Investment Quiz — Failed
10/04/2025
Singleplayer · Score: 10.00% · Time: 1:15 min · Correct: 1 · Incorrect: 1

All About Hazelnuts — Failed
25/03/2025
Singleplayer · Score: 20.00% · Time: 0:20 min · Correct: 2 · Incorrect: 1

Hentai Exploration Quiz — Failed
02/04/2025
Singleplayer · Score: 0.00% · Time: 0:26 min · Correct: 0 · Incorrect: 4

Vallah käse — Failed
28/03/2025
Singleplayer · Score: 0.00% · Time: 0:17 min · Correct: 0 · Incorrect: 3

Grundlegendes zum Total Quality Management (TQM) — Passed
07/04/2025
Document · Score: 100.00% · Time: 7:54 min · Correct: 10 · Incorrect: 0

Exploring the Depths of D1
10/04/2025
Multiplayer · Score: 0 · Correct: 0 · Incorrect: 1

Abbildung 29: IntelliQ Bookmarks Page

3.6.7 Multiplayer Mode

Ausgearbeitet von Nippon Lama

3.6.7.1 How the multiplayer mode works with Supabase

Supabase provides real-time capabilities using a WebSocket-based system. At the core of its multiplayer support is the Realtime Server, which listens to changes in database or custom messages sent through channels. This is especially useful for building multiplayer apps.

The main mechanism used for multiplayer communication is called a channel. A channel is like a virtual room where multiple users can connect and exchange messages in real time. When a user joins a channel, they open a persistent WebSocket connection to the Supabase Realtime server.

```
const channel = supabase.channel('room-123')
```

Each channel supports presence tracking, which allows the app to know which users are currently connected to that channel. To make this work, a client must call `.track()` after subscribing to the channel. This method sends metadata—like the user's ID, name, or avatar—which is shared with everyone in the same room. All connected clients receive updates when users join or leave the channel. Supabase automatically emits a sync event whenever the presence state changes. You can handle this with a listener and update your local state to reflect the current list of users in the room. Presence state is retrieved with `.presenceState()`, which returns an object containing each connected client's tracked data. This is useful for rendering a real-time player list, showing online indicators, or determining who the host is.

```
channel.subscribe().then(() => {
  channel.track({ userId: 'abc123', name: 'Player1' });
});
```

Another key feature is broadcasting. A client can send custom messages to every other client connected to the same channel using `.send()` with the type `"broadcast"`. These messages include an event name (such as „quiz-start“ or „next-question“) and an optional payload. Every client listening for that event will receive the message and can react accordingly—for example, by navigating to the next quiz question or updating their settings. Broadcasts do not go through the database and are faster because they are sent over the WebSocket connection.

Supabase channels support multiple event types:

- `presence`: for tracking joins, leaves and states.
- `broadcast`: for sending custom real-time messages.
- `postgres_changes`: for watching actual database changes (not commonly used in multiplayer lobbies but still possible).

This system makes it easy to build real-time multiplayer experiences without needing to set up own WebSocket server or custom backend.

```
// Lobby Leader broadcasts
channel.send({
  type: 'broadcast',
  event: 'quiz-start',
  payload: { roomId: 'room-123' }
});

//other players receive it
channel.on('broadcast', { event: 'quiz-start' }, (payload) => {
  // handle quiz start
});
```

To keep the system clean and scalable, Supabase allows you to unsubscribe from channels using removeChannel(). This is important for performance, especially when navigating away from a multiplayer view. It stops the WebSocket connection and clears the client from the presence state.

```
supabase.removeChannel(channel);
```

[25]

3.6.7.2 Core Multiplayer Context Setup

The Multiplayer Context is a sophisticated system that enables real-time, competitive quiz experiences. It works seamlessly with the `QuizContext` and `QuizLogicContext` to create a synchronized, engaging experience for multiple players. By managing room creation, room settings, player synchronization, whether the user is the creator (host) of the room, timer coordination, room metadata like roomId and answer submission, it provides the foundation for multiplayer quiz functionality in IntelliQ.

Internal Flags for room settings:

```
  const [players, setPlayers] = useState<Player[]>([]);
  const [isCreator, setIsCreator] = useState<boolean>(false);
  const [channel, setChannel] = useState<RealtimeChannel | null>(null);
  const [maxPlayers, setMaxPlayers] = useState<number>(5);
  const [questionCount, setQuestionCount] = useState<number>(5);
  const [timeLimit, setTimeLimit] = useState<number>(25);
  const [topic, setTopic] = useState<string>("");
  const [language, setLanguage] = useState<SupportedLanguages>(
SupportedLanguages.English,);
  const [roomId, setRoomId] = useState<string>("");
```

Player Structure:

```
export type Player = {
  id: string;
  userName: string;
  email: string;
  isCreator?: boolean;
  score?: number;
  selectedAnswer: string | null;
  avatar: string;
  settings?: {
    timeLimit: number;
    topic: string;
    language?: string;
  };
}
```

This player type represents the player in a quiz lobby. IntelliQ tracks their useful information such as `userName` and `email`, but also, if this player is a lobby leader `isCreator`, tracks this user's `score`, their `selectedAnswer`, `avatar`, and the lobby setting to synchronise with other players

3.6.7.3 Managing Quiz State Across Players

IntelliQ uses Supabase's real-time channels to implement broadcasting. When the lobby leader (creator) sends a broadcast, it's transmitted to all players connected to the same channel, allowing for real-time synchronization of the quiz state.

Key Broadcast Events:

When the lobby leader moves to the next question, they send a broadcast to all players:

```
if (channel && isCreator) {
  channel.send({
    type: "broadcast",
    event: "next-question",
    payload: {
      questionNumber: newQuestionNumber,
    },
  });
}
```

This broadcast contains the new question number that all players should display. When other players receive this broadcast, they update their local state to match:

```
roomChannel
  .on("broadcast", { event: "next-question" }, async ({ payload }) => {
    // Check if we've reached the end of the quiz
    if (payload.questionNumber >= currentQuiz.quiz.length) {
      setQuizFinished(true);
    }

    // Update the question number for all players
    setQuestionNumber(payload.questionNumber);

    // Hide the correct answer until the timer runs out
    setShowCorrectAnswer(false);

    // Update the progress bar
    setProgressValue(
      (payload.questionNumber / currentQuiz.quiz.length) * 100,
    );
  })
```

When the lobby leader starts the quiz, they send a broadcast to all players:

```
await channel.send({
  type: "broadcast",
  event: "quiz-start",
  payload: { currentQuiz, roomId },
});
```

This broadcast contains the quiz data that all players need. When other players receive this broadcast, they update their quiz state and navigate to the quiz page:

```
.on("broadcast", { event: "quiz-start" }, ({ payload }) => {
  getMultiplayerQuizForPlayers(payload.roomId, payload.currentQuiz);
  dispatch({ type: "FETCH_QUIZ_SUCCESS", payload: payload.currentQuiz });
  router.push(`/multiplayer/${roomCode}/play`);
})
```

When the lobby leader changes quiz settings, they send a broadcast to all players:

```
await channel.send({
  type: "broadcast",
  event: "settings-update",
  payload: { type, value },
});
```

This broadcast contains the updated setting type and value. When other players receive this broadcast, they update their local settings:

```
.on("broadcast", { event: "settings-update" }, ({ payload }) => {
  const { type, value } = payload;

  switch (type) {
    case "numQuestions":
      setQuestionCount(value as number);
      break;
    case "timeLimit":
      setTimeLimit(value as number);
      break;
    case "topic":
      setTopic(value as string);
    case "language":
      setLanguage(value as SupportedLanguages);
      break;
  }
})
```

3.6.7.4 Synchronizing User Presence in Rooms

IntelliQ also uses Supabase's presence feature to track which players are in the room and share their state:

```
roomChannel.on("presence", { event: "sync" }, () => {
  // Code to handle presence sync
})
```

This code sets up an event listener for the `"sync"` event on the presence channel. The `"sync"` event is triggered by Supabase whenever there's a change in the presence state of any player in the room. This could happen when:

- A new player joins the room
- A player leaves the room
- A player's state changes (e.g., they select an answer or their score updates)

```
.subscribe(async (status) => {
  if (status === "SUBSCRIBED" && currentUser) {
    const presenceData = {
      currentUser: {
        id: currentUser.id,
        email: currentUser.email,
        name: currentUser.name,
        score: correctAnswer,
        selectedAnswer,
        isCreator,
      },
    };

    await roomChannel.track({ presenceData });
  }
});
```

This code handles the initial subscription to the channel and sets up the current user's presence: The presence data includes:

1. Basic user information (id, email, name)
2. Current game state (score, selectedAnswer)
3. Creator status (Leader)

```
const newState = roomChannel.presenceState();
const playersList = Object.values(newState)
  .flat()
  .map((player) => {
    const data = player as any;

    if (!data.presenceData?.currentUser) {
      return null;
    }

    return {
      id: data.presenceData.currentUser.id,
      email: data.presenceData.currentUser.email,
      userName: data.presenceData.currentUser.name,
      score: data.presenceData.currentUser.score,
      selectedAnswer: data.presenceData.currentUser.selectedAnswer,
      isCreator: false, // Default to false for all players initially
    } as Player;
  })
  .filter(Boolean);
```

presenceState() is a Supabase method that returns the current state of all players in the channel. The returned object has a structure like:

```
{
  "client-id-1": [
    { presenceData: { currentUser: { id: "user1", name: "Player 1", ... } } }
  ],
  "client-id-2": [
    { presenceData: { currentUser: { id: "user2", name: "Player 2", ... } } }
  ],
  // ... more players
}
```

This code processes the presence state to create a list of players:

1. `Object.values(newState)` - Gets an array of all presence data arrays
2. `.flat()` - Flattens the nested arrays into a single array of presence data objects
3. `.map((player) => {...})` - Transforms each presence data object into a Player object
4. `.filter(Boolean)` - Removes any null values (players with invalid data)

The mapping process extracts user information from the presence data.

```
// Sort players consistently - this ensures stable creator assignment
const sortedPlayers = [...playersList].sort((a, b) =>
  a!.id.localeCompare(b!.id),
);

// If we have players, the first one is the creator
if (sortedPlayers.length > 0) {
  sortedPlayers[0]!.isCreator = true;

}
setPlayers(playersList as Player[]);
```

This code ensures that the creator role is assigned consistently across all clients:

1. `[...playersList].sort((a, b) => a!.id.localeCompare(b!.id))` - Creates a sorted copy of the players list, ordering by user ID
2. `if statement` - Assigns the creator role to the player with the lexicographically smallest ID

This approach ensures that all clients will assign the creator role to the same player. The last piece of code `setPlayers(playersList as Player[]);` updates the local state with the processed list of players. This state is then used throughout the application to display player information, track scores, etc. Keep in mind that `setPlayers` updates the state for all the players in the lobby.

```
  useEffect(() => {
    // The entire code runs inside this useEffect hook
  [supabase,roomCode,router,currentUser,selectedAnswer,
correctAnswer,questionNumber,]);
```

The use of this Hook instructs `React` that the component requires execution of an action following its rendering. It is important to note that a component is re-rendered when a state changes. Consequently, all the state listed in the dependencies, such as `supabase` and `questionNumber`, etc., when they change, causes the code inside `useEffect` to re-run. It is important to note that code blocks in this `useEffect` hook such as event listeners which monitor for broadcasts are re-run when the `questionNumber` changes. It should also be noted that the addition of unnecessary dependencies could break the way the app responds.

4 Workflow Architecture

Ausgearbeitet von Ricky Raveanu

The IntelliQ platform has undergone a significant architectural evolution, transitioning from a traditional AWS infrastructure to a modern edge-computing architecture powered by Cloudflare. This chapter details the current infrastructure design, the rationale behind architectural decisions, and the key components that enable our scalable quiz platform.

4.1 End-to-End Workflow Diagram

The workflow architecture of IntelliQ follows a layered approach, with each layer serving a specific purpose in the application stack. At the core of our architecture is the integration between Cloudflare's edge network and Supabase's database services, bridged by Cloudflare Hyperdrive for optimal performance.

4.1.1 Key Components

DNS Layer:

- Cloudflare DNS handles all incoming traffic to app.intelliq.dev
- SSL/TLS encryption is automatically managed through Cloudflare

Frontend Layer:

- Next.js application hosted on Cloudflare Pages
- Server-side rendering for improved SEO and performance
- Static assets served through Cloudflare's CDN

Backend Layer:

- Cloudflare Workers running Hono framework
- Edge computing for reduced latency
- Cloudflare KV for caching
- Upstash Redis for rate limiting

Database Layer:

- Supabase Postgres as the primary database
- Cloudflare Hyperdrive for database connection pooling
- Average query latency reduced from 200-1000ms to 40-80ms

4.2 Infrastructure Evolution

4.2.1 Initial AWS Architecture

The initial architecture of IntelliQ was built on AWS services:

- Frontend hosted on AWS Amplify [26]
- Backend running on ECS Fargate
- Load balancing through AWS ALB [27]
- Database operations directly to Supabase [28]

Challenges with this approach included:

- High operational costs
- Complex deployment workflows
- Higher latency for non-central European users
- Maintenance overhead of container infrastructure

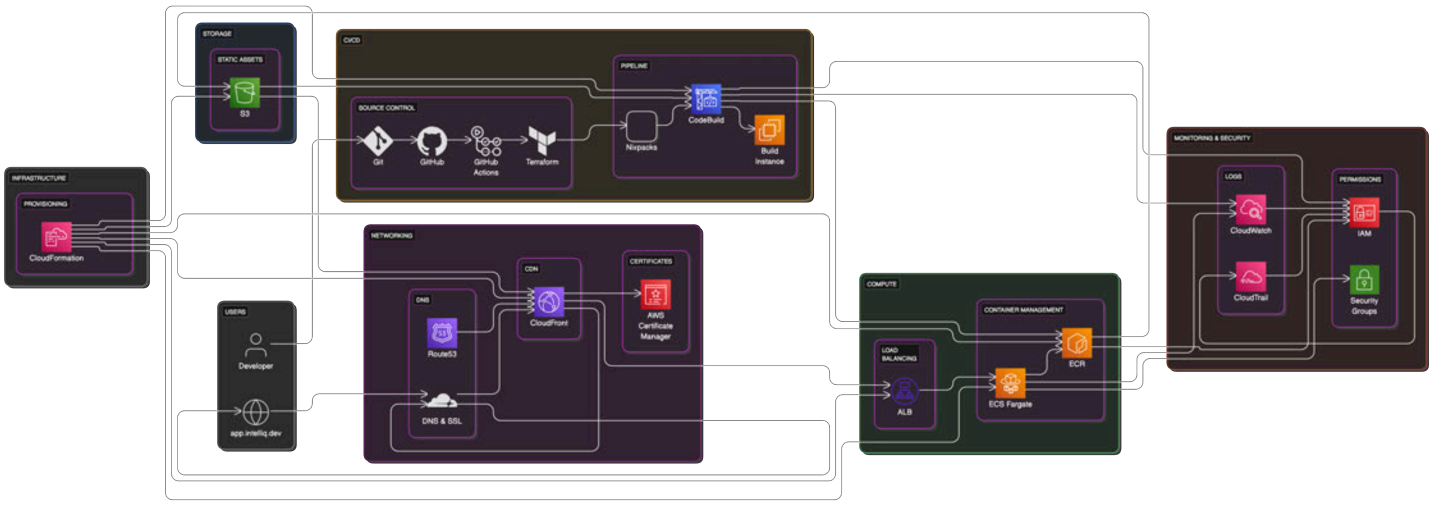


Abbildung 30: IntelliQ Fargate Architecture

4.2.2 Current Edge Architecture

The migration to Cloudflare's edge infrastructure brought several improvements:

1. Deployment Simplicity:
    - Zero-configuration deployments through Cloudflare Pages [29]
    - Automatic builds and previews for each git push
    - Simplified rollback procedures
2. Performance Optimization:
    - Edge computing reduces latency globally [29]
    - Hyperdrive optimizes database connections [30]
    - Multi-layer caching strategy:
        - KV for static data [3]
        - Browser caching for assets [31]
3. Cost Efficiency:
    - Reduced infrastructure costs by 70%
    - Eliminated container management overhead
    - Pay-per-request model for Workers

4.2.3 Security Considerations

The edge architecture implements multiple security layers:

1. Access Control:
    - Rate limiting via Upstash Redis [32]
    - JWT validation at the edge
    - Supabase RLS policies
2. Data Protection:
    - End-to-end encryption
    - Edge isolation of sensitive operations
    - Automated security headers

This architecture has proven to be more resilient, cost-effective, and performant than our previous AWS-based solution, while significantly reducing operational complexity.

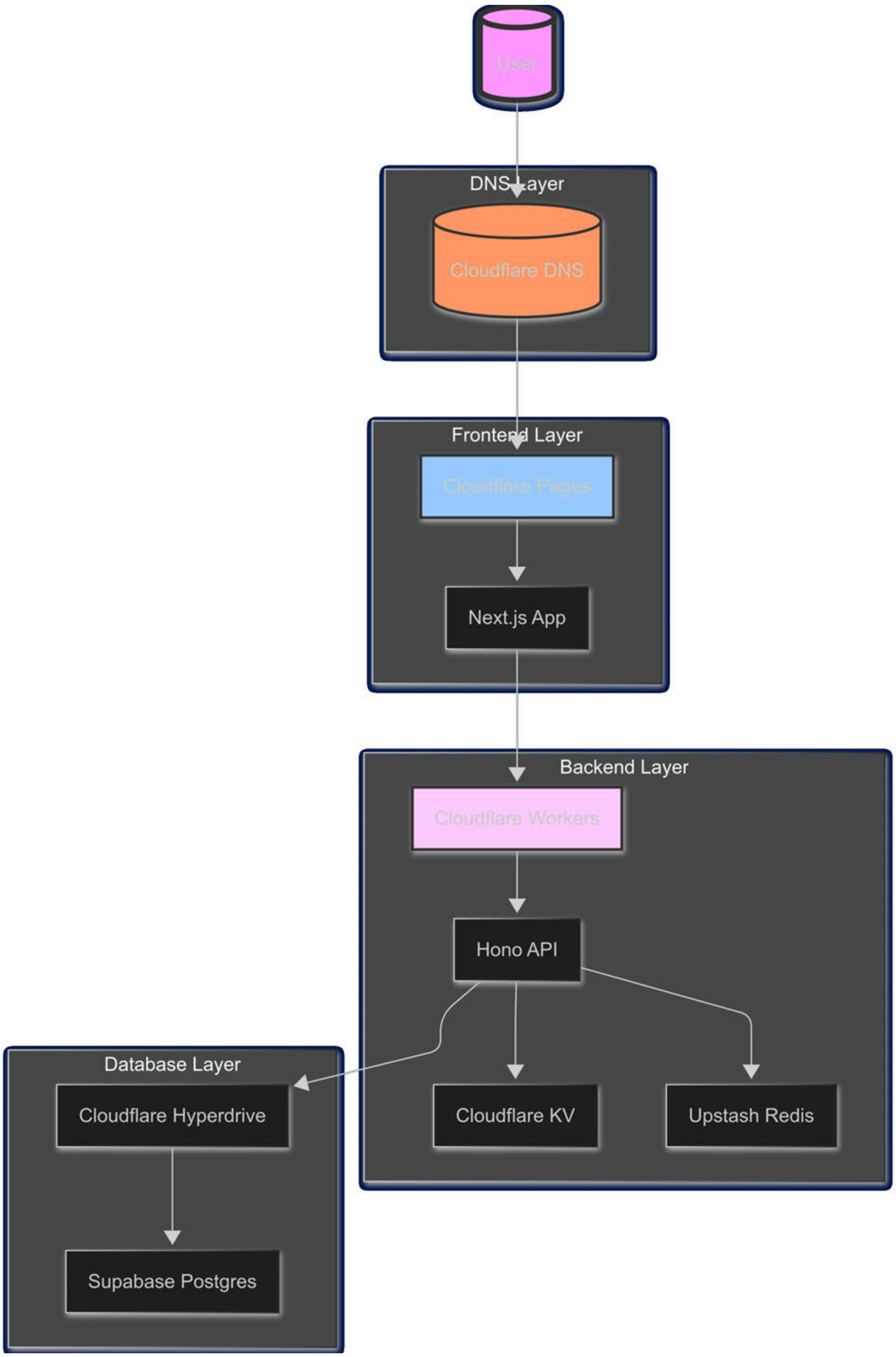


Abbildung 31: IntelliQ End to End Architecture

5 Database Design & Data Migration

Ausgearbeitet von Ricky Raveanu

5.1 Database Architecture Design

The database architecture for IntelliQ was designed to support a scalable, performant quiz platform with real-time multiplayer capabilities. The primary requirements that shaped our database design included:

1. Support for multiple quiz types (singleplayer, multiplayer, document-based, random)
2. Efficient user data management and authentication
3. Real-time multiplayer session handling
4. Document storage and retrieval for quiz generation
5. User activity tracking and analytics
6. Performance optimization through caching
7. GDPR compliance and data security

5.1.1 Core Database Structure

The database schema is built around several key entities:

1. Users: Central entity storing user profiles and authentication data
2. Quizzes: Stores quiz metadata and configuration
3. Questions: Contains individual questions and answers
4. Rooms: Manages multiplayer sessions
5. Documents: Stores uploaded files for document-based quizzes
6. User Responses: Tracks all user answers and performance
7. User Usage Data: Monitors system usage and API consumption
8. Bookmarks: Manages user-saved quizzes

5.1.2 Performance Optimization Strategies

To achieve optimal performance, we implemented a multi-layered caching approach:

1. Database Connection Pooling:

```
const createDb = async (c: Context) => {
  const connectionString = c.env.ENVIRONMENT === "development"
    ? c.env.DATABASE_URL
    : c.env.HYPERDRIVE.connectionString;
  const db = drizzle(postgres(connectionString), {
    schema: {...schema, ...relations}
  });
  return db;
};
```

2. Three-Tiered Caching Strategy [33]:

```
const SHORT_CACHE = 30;    // 30 seconds
export const MEDIUM_CACHE = 150; // 2.5 minutes
export const LONG_CACHE = 1800;  // 30 minutes
```

5.2 Supabase Database Implementation

Our implementation utilizes Supabase as the primary database provider, with several optimizations:

5.2.1 ER Modeling & Optimization [1]

The database schema was designed using Drizzle ORM with careful consideration of relationships and constraints:

1. Strict Foreign Key Relationships:

```
export const quizzesRelations = relations(quizzes, ({one, many}) => ({
  multiplayerQuizSubmissions: many(multiplayerQuizSubmissions),
  userResponses: many(userResponses),
  questions: many(questions),
  bookmarks: many(bookmarks),
  document: one(documents, {
    fields: [quizzes.documentId],
    references: [documents.id]
  })
}));
```

2. Row Level Security Policies:
   - Granular access control for data mutations
   - Automatic user isolation for data privacy
   - Role-based access for administrative functions
3. Performance Optimizations:
   - Indexed foreign keys for faster joins
   - Efficient data types selection (e.g., smallint for scores)
   - Optimized query patterns with materialized views

5.2.2 Migration of Existing User Data

The transition from AWS to Cloudflare infrastructure required careful data migration:

1. Data Migration Process:
    - Schema validation and cleanup
    - Data transformation and normalization
    - Incremental migration to minimize downtime
    - Validation of migrated data integrity
2. Performance Improvements:
    - Original Supabase (Frankfurt) latency: 400ms - 1s
    - Post-migration with Hyperdrive: 80ms - 150ms
    - With added caching layer: 40ms - 80ms

6 Multiplayer System & Real-Time Quiz Flow

Ausgearbeitet von Ricky Raveanu

6.1 Overview of Multiplayer Architecture

The multiplayer quiz system in IntelliQ implements a real-time, synchronized quiz experience across multiple participants. This section explains the technical implementation of the multiplayer flow from quiz creation to results visualization, detailing the API endpoints, data structures, and synchronization mechanisms that enable the collaborative quiz experience.

6.1.1 Multiplayer Data Flow

The multiplayer quiz flow follows a defined sequence of API interactions, as illustrated in the flow diagram below. Each step in the process is handled by specific API endpoints, ensuring proper synchronization and data consistency across all participants.

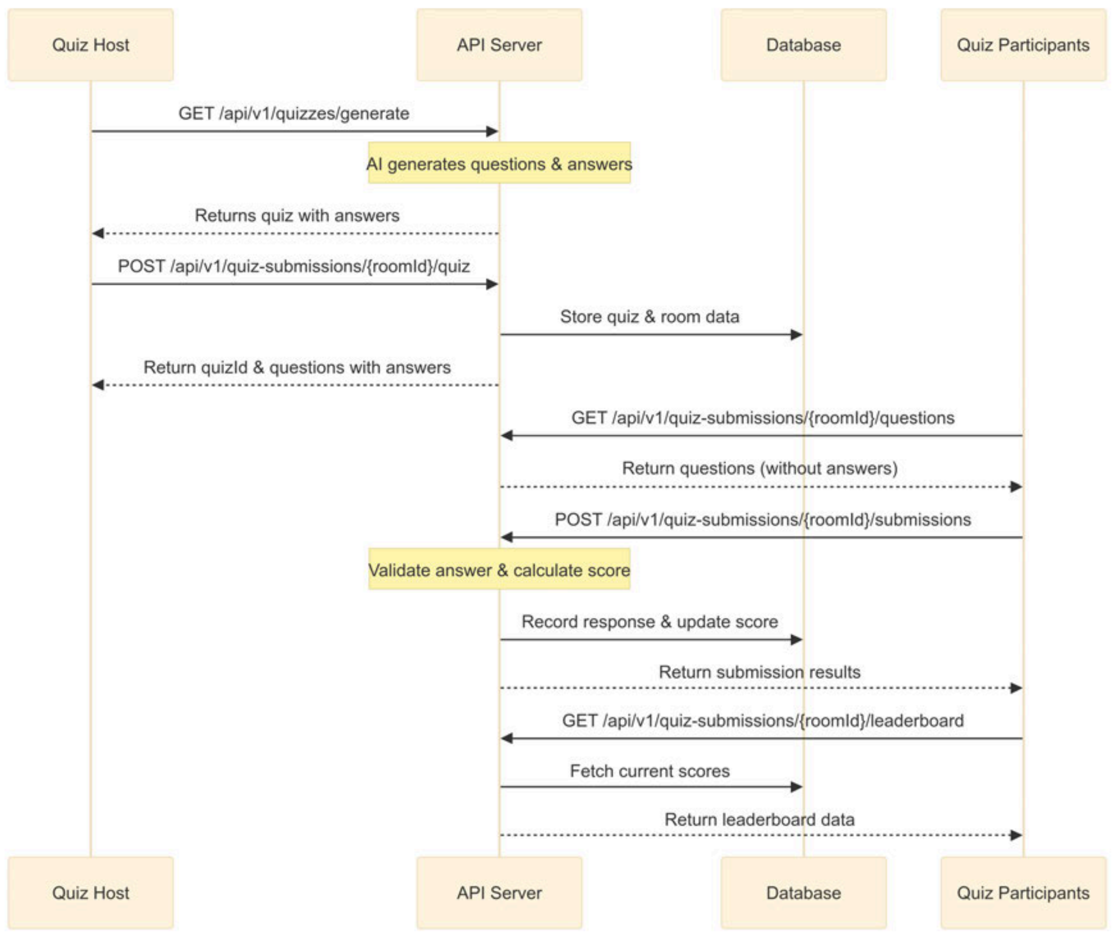

Quiz Host
API Server
Database
Quiz Participants
GET /api/v1/quizzes/generate
AI generates questions & answers
Returns quiz with answers
POST /api/v1/quiz-submissions/{roomId}/quiz
Store quiz & room data
Return quizId & questions with answers
GET /api/v1/quiz-submissions/{roomId}/questions
Return questions (without answers)
POST /api/v1/quiz-submissions/{roomId}/submissions
Validate answer & calculate score
Record response & update score
Return submission results
GET /api/v1/quiz-submissions/{roomId}/leaderboard
Fetch current scores
Return leaderboard data
Quiz Host
API Server
Database
Quiz Participants

Abbildung 32: IntelliQ Multiplayer Flow

6.1.2 Core Components of Multiplayer System

The multiplayer functionality is built upon several key technical components:

1. Room Management: Handles user sessions and participant synchronization
2. Quiz Distribution: Controls the consistent delivery of questions to all participants
3. Answer Validation: Processes user submissions and calculates scores in real-time
4. Leaderboard System: Tracks and displays participant performance

6.2 Technical Implementation [2]

6.2.1 Room Creation Process

The multiplayer quiz begins with the quiz host generating questions and creating a room:

1. Quiz Generation (Host Only):

```
GET
/api/v1/quizzes/generate?quizTopic=formula%20one&numberOfQuestions=5
&language=en&quizType=multiplayer
```

   This endpoint leverages OpenAI's API to generate context-specific questions based on the provided topic. The server returns a structured quiz object containing:

```
{
"quiz": {
  "quizTitle": "Formula One Trivia Challenge",
  "questions": [
      {
        "questionTitle": "The Fastest Lap Record",
        "text": "Which driver holds the record for the fastest lap in
Formula One history?",
        "options": ["a) Lewis Hamilton", "b) Michael Schumacher", "c)
Max Verstappen", "d) Sebastian Vettel"],
        "correctAnswer": "c) Max Verstappen"
      },
    ]
  }
}
```

2. Room Creation (Host Only):

```
POST
/api/v1/quiz-submissions/{roomCode}/quiz
```

The host creates a multiplayer session by submitting the generated quiz to the server.
The server:

- Validates the quiz structure
- Creates a room entity in the database
- Generates a unique room code for participants to join
- Returns the room details and question identifiers

The backend implementation ensures session persistence through Supabase:

```
const result = await db.transaction(async (tx) => {
const { quizTitle, quizTopics, language, questions } = validatedData;

const [createdQuiz] = await tx
  .insert(quizzes)
  .values({
    userId: room.hostId,
    title: quizTitle,
    topic: quizTopics,
    language: language,
    questionsCount: questions.length,
    type: quizType.Enum.multiplayer,
    roomId: roomId,
  })
  .returning({
    id: quizzes.id,
    title: quizzes.title,
  });
```

```
    // Create questions in database
    const createdQuestions = [];
    for (const question of questions) {
      const [createdQuestion] = await tx
        .insert(questionsTable)
        .values({
          text: question.text,
          options: question.options,
          correctAnswer: question.correctAnswer,
          quizId: createdQuiz.id,
        })
        .returning({
          id: questionsTable.id,
          text: questionsTable.text,
          options: questionsTable.options,
          correctAnswer: questionsTable.correctAnswer,
        });

      createdQuestions.push({
        questionTitle: createdQuiz.title,
        ...createdQuestion,
      });
    }

    return {
      quizId: createdQuiz.id,
      questions: createdQuestions,
    };
  });
```

6.2.2 Participant Interaction Flow

Once the room is created, participants can join and interact with the quiz:

1. Question Retrieval (All Participants):

```
GET
/api/v1/quiz-submissions/{roomCode}/questions
```

   Participants use this endpoint to fetch the quiz questions. The response includes all questions but deliberately omits the correct answers:

```
  {
"quizId": "557e75e4-d80e-4463-a282-f449190d4d1e",
"quizTitle": "Formula One Trivia Challenge",
"questions": [
  {
    "id": "1e45e088-aeac-49c5-8314-34638c89eec9",
    "questionTitle": "The Fastest Lap Record",
   "text": "Which driver holds the record for the fastest lap in Formula
One history?",
    "options": [
      "a) Lewis Hamilton",
      "b) Michael Schumacher",
      "c) Max Verstappen",
      "d) Sebastian Vettel"
      ]
    }
  ]
}
```

2. Answer Submission (All Participants):

```
POST
/api/v1/quiz-submissions/{roomCode}/submissions
```

As participants answer questions, they submit their responses to this endpoint. The server:

- Validates the answer against the correct answer in the database
- Calculates the score based on accuracy and response time
- Updates the participant's running score in the session
- Returns the submission result with calculated points

The scoring algorithm factors in both correctness and speed:

```
  // Calculate points based on time taken and correctness
const isCorrect = userAnswer === correctAnswer;
let calculatedScore = 0;

if (isCorrect) {
  // Base score calculation formula
  calculatedScore = Math.round((1 - (timeTaken / timeLimit) / 2.2)
* 1000);

  // Fast answer bonus
  if (timeTaken < (timeLimit * 0.1)) {
    calculatedScore += 50; // Bonus for very quick answers
  }

  // Ensure minimum points for correct answers
  calculatedScore = Math.max(calculatedScore, 100);
}
```

6.3 Performance Optimizations

To ensure the multiplayer system remains responsive even under high load, several performance optimizations have been implemented:

6.3.1 Database Query Optimization

All database interactions are optimized for minimal latency:

1. Transaction-based updates to ensure data consistency
2. Selective column retrieval to reduce data transfer
3. Optimized join operations for related data
4. Index utilization for rapid participant lookup

6.3.2 Edge Computing Advantages

By leveraging Cloudflare Workers and edge computing, the multiplayer system benefits from:

1. Reduced global latency (average response time under 80ms)
2. Higher concurrency handling capability

6.4 User Usage Logging

Implementation of comprehensive usage tracking:

1. Token Usage Monitoring:

```
export const userUsageData = pgTable("user_usage_data", {
  promptTokens: integer("prompt_tokens").notNull(),
  completionTokens: integer("completion_tokens").notNull(),
  totalTokens: integer("total_tokens").notNull(),
  usedModel: text("used_model").notNull(),
  responseTimeTaken: real("response_time_taken").notNull()
});
```

2. Analytics Data Collection:
   - Quiz completion rates
   - Response time tracking
   - Model usage statistics
   - Language preferences

6.4.1 Cache Management [3], [3]

Implementation of an efficient caching strategy using Cloudflare KV:

```
const cacheKey = async (c: Context<{ Bindings: CloudflareEnv }>) => {
  const supabase = c.get("supabase");
  const { data: { user } } = await supabase.auth.getUser();

  if (!user) {
    return `${c.req.url}&v=1#user=anonymous`;
  }

  const kv = c.env.IntelliQ_CACHE_VERSION;
  const userVersion = await getUserCacheVersion(kv, user.id);
  const separator = c.req.url.includes("?") ? "&" : "?";

  return `${c.req.url}${separator}v=${userVersion}#user=${user.id}`;
};
```

6.4.2 Security Implementation

1. Row Level Security:

```
CREATE POLICY "Users can insert their own documents"
ON "documents" AS PERMISSIVE
FOR INSERT TO "authenticated"
WITH CHECK ((auth.uid() = user_id));
```

2. Data Access Controls:
   - User isolation through RLS policies
   - Encrypted sensitive data fields
   - Audit logging for data access
   - Regular security audits

The migration to Cloudflare's infrastructure and implementation of the caching layer resulted in significant performance improvements:

- 75% reduction in average query latency
- 99.9% uptime achievement
- Global access latency under 100ms
- Reduced database load through effective caching

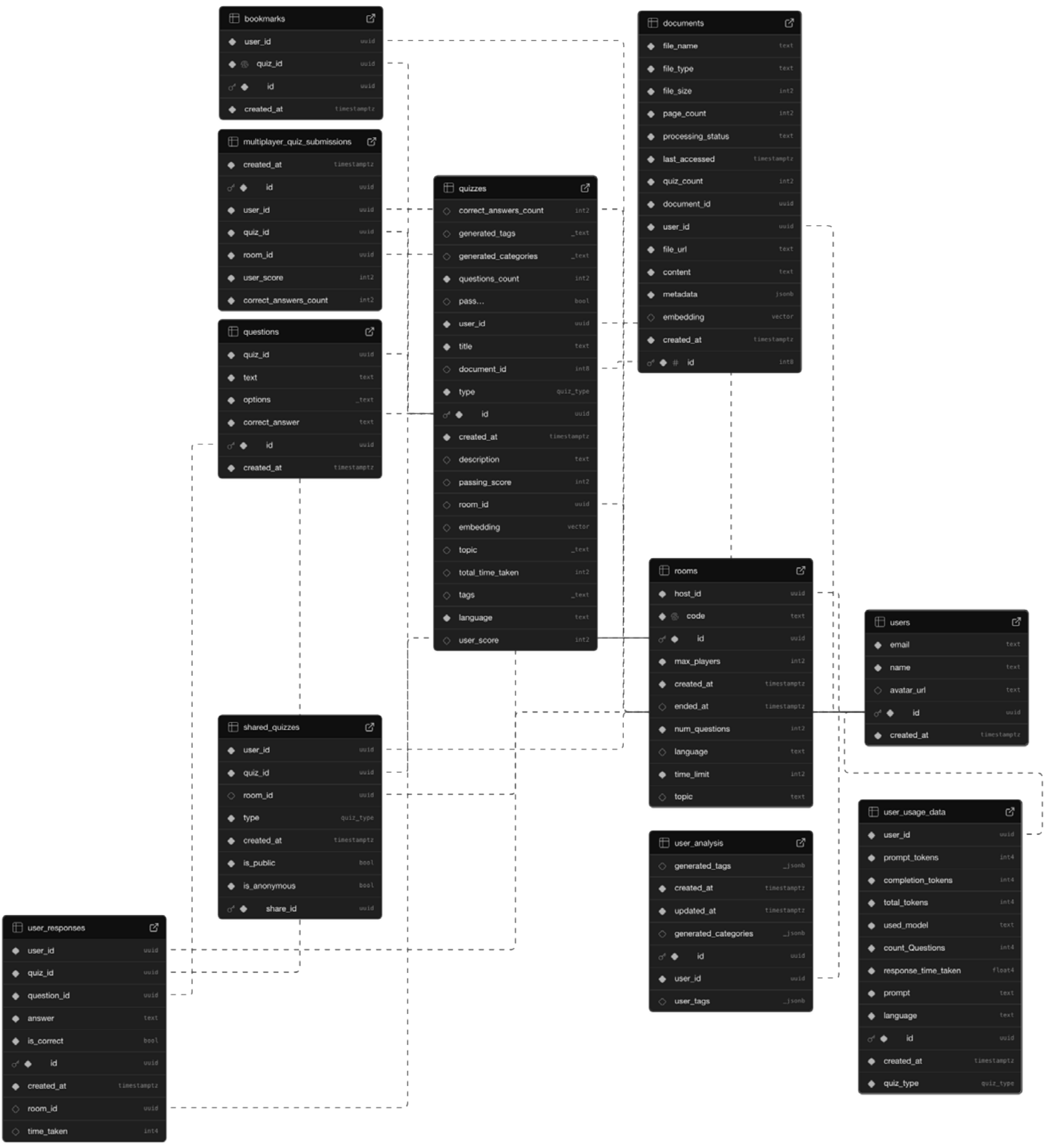


Abbildung 33: IntelliQ DB

7 Document Quiz Flow Implementation

Ausgearbeitet von Ricky Raveanu

7.1 Overview of Document Processing [4]

The IntelliQ platform features an advanced document processing system that allows users to upload documents, automatically extract content, and generate relevant quizzes without manual question creation. This chapter explains the technical implementation of this feature and how it extends the platform's capabilities.

7.2 Architecture Overview

The document quiz implementation in IntelliQ follows these key steps:

1. Document Upload: User uploads a document (currently PDF) through the UI
2. Storage & Processing: The document is stored and queued for asynchronous processing
3. Text Extraction: Content is extracted from the document and stored
4. Content Chunking: For large documents, content is split into meaningful chunks
5. Quiz Generation: AI generates quiz questions based on document content
6. Interactive Quiz: User takes the quiz with customized parameters
7. Result Storage: Quiz results are stored with document reference

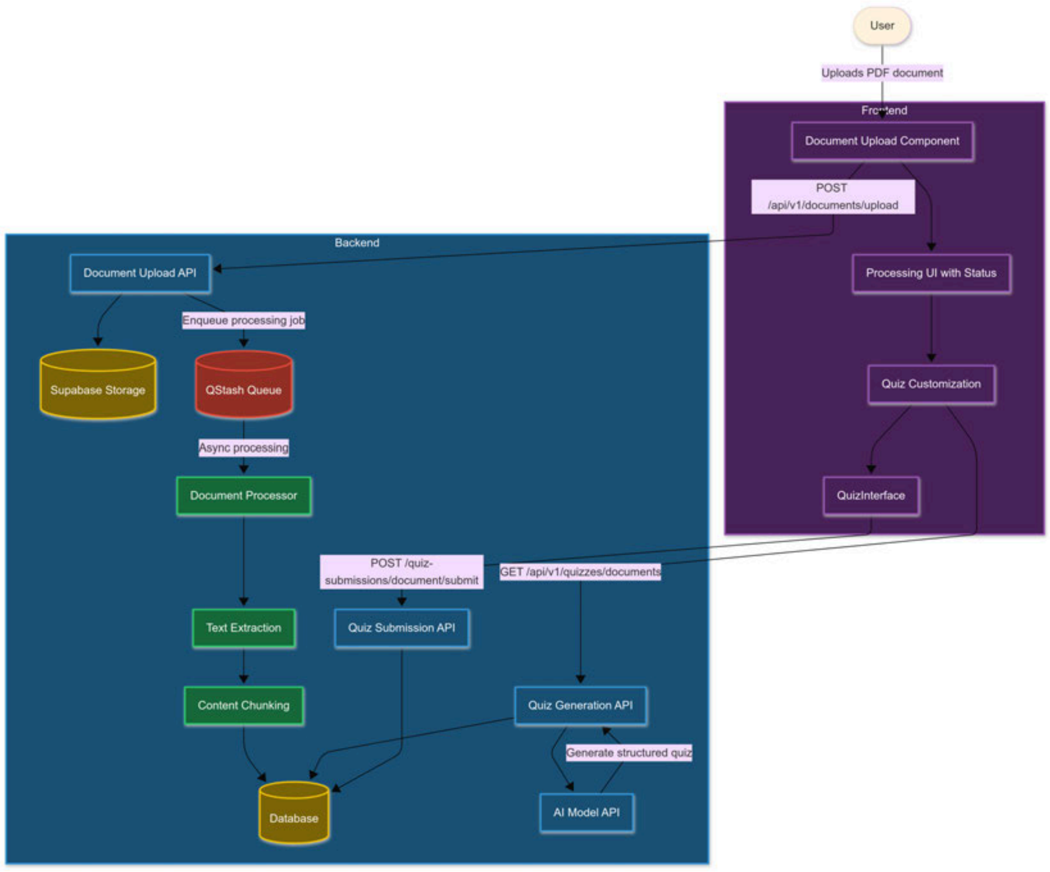


Abbildung 34: IntelliQ Document Quiz Flow Architecture

7.3 Technical Implementation

7.3.1 Document Upload Flow

The document upload process is handled through a comprehensive file upload component in the frontend that supports drag-and-drop functionality and progress tracking. This component communicates with a dedicated document upload API endpoint that performs several critical functions:

- Validates the uploaded file type and permissions
- Uploads the file to Supabase Storage with appropriate security settings
- Creates a database record with metadata about the document
- Queues an asynchronous processing job using QStash

The implementation uses React for the frontend component and Hono for the API routes, maintaining the same architectural patterns used throughout the platform.

7.3.2 Document Processing

Document processing occurs asynchronously through a queued workflow:

- Background worker retrieves the document from storage
- Text extraction is performed based on file type

- For PDFs, content is extracted using PDF.js with page tracking
- Extracted text is stored in the database for future access
- Large documents are automatically split into semantic chunks
- Processing status is updated at each step to provide user feedback

For large documents, a recursive character text splitter is employed that maintains semantic cohesion by properly splitting on paragraph and section boundaries with appropriate overlap.

7.4 Quiz Generation

Once document processing is complete, quiz generation leverages the AI integration:

- API endpoint verifies user ownership of the document
- Processed document content (or chunks) is retrieved
- OpenAI API is used with structured outputs to generate a quiz
- Schema validation ensures consistent quiz format and options
- User usage data is tracked for analytics and billing purposes
- Optional translation service converts questions to requested language

The quiz generation process uses a specific prompt template designed to create four distinct options for each question while ensuring the correct answer pattern remains unpredictable. This approach creates a natural and challenging quiz experience based directly on the document content.

7.5 Data Schema Integration

The document quiz functionality extends the existing database schema with document-specific tables that track:

- Document metadata (file size, type, creation date)
- Processing status and extracted content
- Relationship to generated quizzes
- User ownership and access controls
- Usage statistics for analytics

Quiz submissions maintain a reference to their source document through foreign key relationships, enabling a comprehensive history of document-based learning activities.

7.6 Performance and Security Considerations

7.6.1 Performance Optimizations

Asynchronous Processing: [34]

- Document processing happens asynchronously using QStash
- Prevents blocking the main application flow
- Allows for parallel document processing

Chunking Strategy: [35]

- Large documents are automatically split into smaller chunks
- Avoids AI token limits
- Maintains semantic context between chunks with overlaps

Selective Content:

- For very large documents, only representative chunks are used
- Optimizes token usage
- Focuses on the most relevant content

7.6.2 Security Measures

Document Ownership:

- All API endpoints validate that the requesting user owns the document
- Prevents unauthorized access to documents
- Ensures data isolation between users

Content Validation:

- Input sanitization prevents injection attacks
- Content validation ensures proper document structure
- Error handling prevents exposure of sensitive information

Data Access Controls:

- Supabase RLS policies restrict access to documents
- Encrypted sensitive data fields
- Audit logging for document access
- Regular security audits of document operations

7.7 Integration with Existing Architecture

The document processing system integrates seamlessly with the IntelliQ platform's edge architecture described in previous chapters:

Edge Computing Benefits:

- Document processing API endpoints run on Cloudflare Workers
- Reduced latency for global document uploads
- Consistent performance regardless of user location

Database Integration:

- Documents are stored in the same Supabase instance
- Hyperdrive optimizes document metadata queries
- Foreign key relationships maintain data integrity

Caching Strategy:

- Document metadata is cached at the edge using KV
- Frequently accessed documents benefit from the caching layer
- Reduces database load for repeated document access

7.8 Future Enhancements

The document quiz functionality will be extended with several planned enhancements:

1. Additional File Types: Support for DOCX, TXT, and other document formats
2. Smart Chunking: More sophisticated document chunking based on semantic boundaries
3. Hierarchical Questions: Questions at different difficulty levels or covering different topics
4. Document Search: Full-text and semantic search across documents
5. AI Explanations: Explanations for correct answers based on document content
6. Interactive Document Links: Questions that link back to the relevant document section

8 Advanced Monorepo Architecture & RPC Implementation

Ausgearbeitet von Ricky Raveanu

8.1 Type-Safe Communication Layer

The IntelliQ platform leverages Hono's RPC [36] capabilities to ensure type-safe communication between the frontend and backend services.

This implementation brings several key benefits:

8.1.1 RPC Integration

The RPC system is implemented using a shared type system:

```
// API Type Definitions
export type AppType = typeof routes;

// Client-side implementation
const client = hc<AppType>('https://app.intelliq.dev/api/v1');

// Type-safe API calls
const response = await client.quizzes.generate.$post({
  form: {
    topic: "Formula 1",
    numberOfQuestions: 5,
    language: "en"
  }
});
```

8.1.2 Performance Optimizations

To address potential IDE performance issues with type inference, we implemented several optimizations:

- Route Splitting

```
// Separate routes for better type inference
const v1 = new Hono<{ Bindings: CloudflareEnv }>()
  .route("/quizzes", quizzes)
  .route("/quiz-submissions/multiplayer", multiplayerQuizSubmissionsRoutes)
  .route("/quiz-submissions/singleplayer", singleplayerQuizSubmissionsRoutes)
  .route("/rooms", rooms)
  .route("/history", historyRoutes);

export default v1;
```

8.1.3 Monorepo Benefits

The monorepo structure provides several advantages for IntelliQ:

1. Type Safety:

```
// Shared types across frontend and backend
interface QuizState {
  id: string;
  questions: Question[];
  currentIndex: number;
  submissions: Answer[];
}

// Used in both frontend and backend
type QuizResponse = {
  quiz: QuizState;
  metrics: {
    timeTaken: number;
    score: number;
  };
};
```

2. Version Control:

```
{
  "workspaces": [
    "apps/*",
    "packages/*"
  ],
  "dependencies": {
    "@intelliq/api": "*",
  }
}
```

8.1.4 Error Handling & Type Safety

The RPC implementation includes robust error handling:

```
const actionClient = createSafeActionClient({
  handleServerError(e) {
    if (e instanceof Error) {
      return {
        code: e.name === 'ValidationError' ? 400 : 500,
        message: e.message
      };
    }
    return { code: 500, message: DEFAULT_SERVER_ERROR_MESSAGE };
  }
});
```

8.2 Type-Safe Integration Benefits

1. Compile-Time Error Detection:

```
// This would cause a compile error if the API changes
const response = await client.quizzes.generate.$post({
  form: {
    topic: "Formula 1",
    numberOfQuestions: "5" // Type error: expecting number
  }
});
```

2. Response Type Inference:

```
type QuizResponse = InferResponseType<typeof client.quizzes.generate.
$post>;
// Automatically infers the correct response type
```

3. Enhanced Development Experience:

   - „Automatic type completion in IDEs“
   - „Instant feedback on API changes“
   - „Reduced runtime errors through compile-time checking“

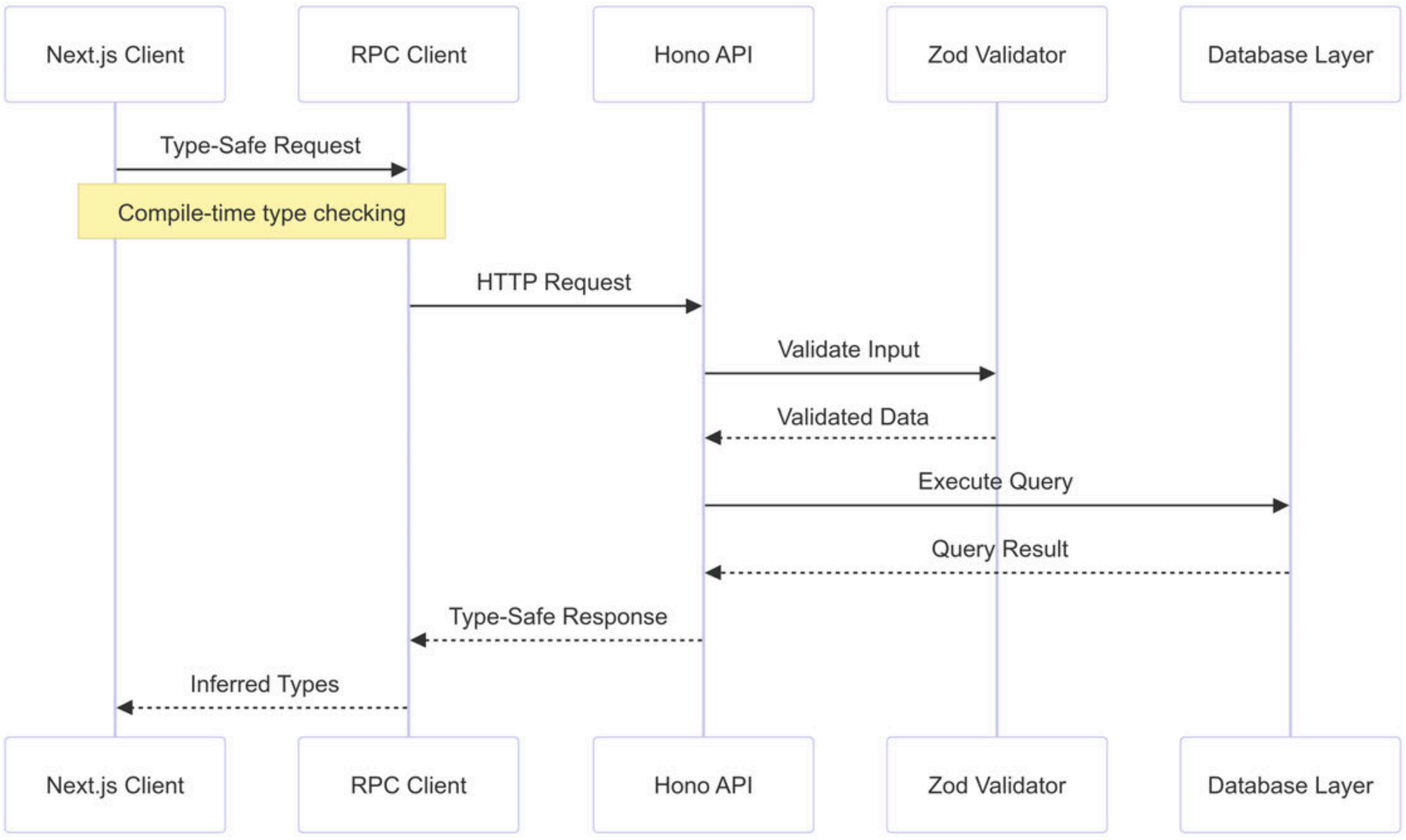


Abbildung 35: IntelliQ RPC [Remote procedure call]

9 Abbildungsverzeichnis

10 Tabellenverzeichnis

11 Literaturverzeichnis

[1] D. Team und Contributors, „Drizzle ORM: A Lightweight TypeScript ORM“. Zugegriffen: 11. April 2025. [Online]. Verfügbar unter: https://orm.drizzle.team/

[2] R. Raveanu, „IntelliQ Docs: Official IntelliQ Documentation“. Zugegriffen: 11. April 2025. [Online]. Verfügbar unter: https://docs.intelliq.dev/

[3] I. Cloudflare, „Cloudflare KV: A Global, Low-Latency Key-Value Store“. Zugegriffen: 11. April 2025. [Online]. Verfügbar unter: https://developers.cloudflare.com/workers/runtime-apis/kv/

[4] R. Raveanu, „IntelliQ Docs: Official IntelliQ Documentation“. Zugegriffen: 11. April 2025. [Online]. Verfügbar unter: https://docs.intelliq.dev/document-quiz-flow

[5] F. Academy, „Ultimate Guide to Choosing Colors for Web Design“. Zugegriffen: 11. April 2025. [Online]. Verfügbar unter: https://www.flux-academy.com/blog/ultimate-guide-to-choosing-colors-for-web-design

[6] Wix, „Good Logo Design Tips“. Zugegriffen: 21. September 2023. [Online]. Verfügbar unter: https://www.wix.com/blog/good-logo-design-tips

[7] S. I. D. Patterns, „How to Redesign Guide“. Zugegriffen: 14. November 2023. [Online]. Verfügbar unter: https://smart-interface-design-patterns.com/articles/how-to-redesign-guide/

[8] Netguru, „Best Practices for UI Web Design“. Zugegriffen: 18. Oktober 2024. [Online]. Verfügbar unter: https://www.netguru.com/blog/best-practices-ui-web-design

[9] Ramshaq, „Dark Mode vs Light Mode: Best Practices for UI Design in 2024“. Zugegriffen: 1. Dezember 2024. [Online]. Verfügbar unter: https://medium.com/@Ramshaq/dark-mode-vs-light-mode-best-practices-for-ui-design-in-2024-bd944c5715a7

[10] Maze, „Modern UI Designs“. Zugegriffen: 27. Juli 2021. [Online]. Verfügbar unter: https://maze.co/collections/ux-ui-design/modern-ui-designs/

[11] H. Developer, „Mastering Responsive Design with Tailwind CSS: Tips and Tricks“. Zugegriffen: 14. August 2024. [Online]. Verfügbar unter: https://dev.to/hitesh_developer/mastering-responsive-design-with-tailwind-css-tips-and-tricks-1f39

[12] Vercel, „Next.js“. Zugegriffen: 1. Dezember 2022. [Online]. Verfügbar unter: https://nextjs.org/

[13] R. Team, „React - A JavaScript library for building user interfaces“. Zugegriffen: 1. Dezember 2022. [Online]. Verfügbar unter: https://reactjs.org/

[14] S. Inc., „Supabase: The open-source Firebase alternative“. Zugegriffen: 1. Dezember 2022. [Online]. Verfügbar unter: https://supabase.io/

[15] Vercel, „ShadCN UI Documentation“. Zugegriffen: 1. März 2023. [Online]. Verfügbar unter: https://ui.shadcn.com/docs

[16] T. Labs, „Tailwind CSS Documentation“. Zugegriffen: 11. April 2025. [Online]. Verfügbar unter: https://v3.tailwindcss.com/resources

[17] L. Laurent, „React's Component-Based Architecture: A Case Study“. Zugegriffen: 31. Juli 2023. [Online]. Verfügbar unter: https://appmaster.io/blog/react-component-based-architecture

[18] J. QIU, „Modularizing React Applications with Established UI Patterns“. Zugegriffen: 16. Februar 2023. [Online]. Verfügbar unter: https://martinfowler.com/articles/modularizing-react-apps.html#LayeredFrontendApplication

[19] R. Team, „React Context API“. Zugegriffen: 1. Dezember 2022. [Online]. Verfügbar unter: https://reactjs.org/docs/context.html

[20] J. Mahadevan, „5 Alternatives to Redux for React State Management“. Zugegriffen: 5. April 2024. [Online]. Verfügbar unter: https://jaydevm.hashnode.dev/5-alternatives-to-redux-for-react-state-management

[21] T. Bose, „Redux vs Context API: When to use them“. Zugegriffen: 28. November 2021. [Online]. Verfügbar unter: https://dev.to/ruppysuppy/redux-vs-context-api-when-to-use-them-4k3p

[22] Billal, „Mastering the Reducer Pattern in React“. Zugegriffen: 20. Dezember 2024. [Online]. Verfügbar unter: https://medium.com/@billalpatel/mastering-the-reducer-pattern-in-react-16e25461dde1

[23] D. Leitch, „Choosing Between “type” and “interface” in React“. Zugegriffen: 9. August 2023. [Online]. Verfügbar unter: https://medium.com/nerd-for-tech/choosing-between-type-and-interface-in-react-da1deae677c91

[24] J. Caron und O.-S. Contributors, „usehooks-ts: Useful React Hooks for TypeScript“. Zugegriffen: 11. April 2025. [Online]. Verfügbar unter: https://github.com/juliencrn/usehooks-ts

[25] S. Inc., „Supabase Realtime Documentation“. Zugegriffen: 1. Dezember 2022. [Online]. Verfügbar unter: https://supabase.io/docs/guides/realtime

[26] I. Amazon Web Services, „AWS Amplify Documentation“. Zugegriffen: 11. April 2025. [Online]. Verfügbar unter: https://aws.amazon.com/amplify/

[27] I. Amazon Web Services, „Application Load Balancer (ALB) Documentation“. Zugegriffen: 11. April 2025. [Online]. Verfügbar unter: https://docs.aws.amazon.com/elasticloadbalancing/latest/application/introduction.html

[28] I. Amazon Web Services, „AWS Fargate Documentation“. Zugegriffen: 11. April 2025. [Online]. Verfügbar unter: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/AWS_Fargate.html

[29] I. Cloudflare, „Cloudflare Developer Platform: Workers, Pages, KV, and More“. Zugegriffen: 11. April 2025. [Online]. Verfügbar unter: https://developers.cloudflare.com/

[30] I. Cloudflare, „Cloudflare Hyperdrive: Turn your existing regional database into a globally distributed database.“. Zugegriffen: 11. April 2025. [Online]. Verfügbar unter: https://developers.cloudflare.com/hyperdrive/

[31] Y. Kobayashi und Contributors, „Hono Built-In Cache Middleware“. Zugegriffen: 11. April 2025. [Online]. Verfügbar unter: https://hono.dev/docs/middleware/builtin/cache

[32] U. Inc., „Upstash Redis: Serverless Database for Redis“. Zugegriffen: 11. April 2025. [Online]. Verfügbar unter: https://upstash.com/redis

[33] D. K. Developer, „Implementing a Three-Tier Caching Strategy for Modern Web Apps“. Zugegriffen: 11. April 2025. [Online]. Verfügbar unter: https://medium.com/@dkdeveloper/three-tier-caching-strategy-web-apps

[34] U. Inc., „QStash: Reliable Messaging for Serverless and Edge Applications“. Zugegriffen: 11. April 2025. [Online]. Verfügbar unter: https://upstash.com/qstash

[35] H. Chase und L. Contributors, „LangChain.js: Building Applications with LLMs in Node.js“. Zugegriffen: 11. April 2025. [Online]. Verfügbar unter: https://js.langchain.com/docs/

[36] Y. Kobayashi und Contributors, „Remote Procedure Call (RPC) with Hono“. Zugegriffen: 11. April 2025. [Online]. Verfügbar unter: https://hono.dev/docs/guides/rpc

A Arbeitsaufteilung

Person	Folgende Punkte der Diplomarbeit wurden inklusive aller Unterpunkte von folgenden Personen verfasst:
Ricky Raveanu	Ricky Raveanu serves as the Lead System Architect, Full-Stack Engineer, and Platform Engineer in the IntelliQ project. As the Lead System Architect, Ricky designs the overall system structure to ensure scalability and seamless integration. They manage backend development, database design, and API integration, ensuring smooth interaction between the frontend and backend. Ricky also handles the platform's infrastructure, real-time multiplayer systems, and document quiz flow, contributing to a scalable, efficient platform.
Nippon Lama	Nippon Lama is a Frontend Engineer, Full-Stack Developer, and Designer. They organize the frontend architecture, focusing on multiplayer mode and document-based quizzes. As a Full-Stack Developer, they ensure smooth integration between frontend and backend features. As a Designer, they create animations and visual elements that enhance user engagement and the overall user experience on the platform.
Nikola Petrovic	Nikola Petrovic works as a Frontend Developer and Designer. As a Frontend Developer, they focus on improving functionality and performance, adding features like random quiz generation and social media sharing. Nikola is also responsible for designing user interfaces and enhancing the quiz experience with auditory feedback. As a Designer, they ensure the platform's visual elements align with the brand identity, including designing the logo and overall aesthetics.