


GoalGuru: A React- and FastAPI-based Cloud Application for Predicting the Outcomes of Soccer Games

Technical Reports: CL-2024-42, July 2024

Bernhard Gailer, Timo Gräf, Maria Lyoteva, Tsvetan Stanchev, Apporva Bhoir,
Christoph P. Neumann 

CyberLytics-Lab at the Department of Electrical Engineering, Media, and Computer Science
Ostbayerische Technische Hochschule Amberg-Weiden
Amberg, Germany

Abstract—GoalGuru is a cloud-based web application that presents users with the possibility to predict the outcomes of soccer games between two teams of their choosing. The architecture of GoalGuru comprises a React frontend, FastAPI backend together with Python code for a machine learning model for the game prediction, and a cloud deployment using Docker images on AWS. The data acquisition was done with web scraping from Transfermarkt. This data is stored on the server using a local instance of the document-oriented database TinyDB and subsequently used for training and creating the machine learning model in the backend. The project is deployed on AWS App Runner using a Docker container in AWS ECR. GoalGuru offers scope for expansion, like for example the integration of different machine learning models to improve predictions or the addition of more soccer teams to choose from for making predictions. The overall architecture of GoalGuru ensures scalability because of the cloud deployment and seamless communication between the frontend and backend.

Index Terms—React, Bootstrap, Machine Learning, Docker, AWS, App Runner, REST, FastAPI, Cloud.

I. INTRODUCTION AND OBJECTIVES

Soccer is a very popular sport that is loved by billions of people all around the globe, with more than two-thirds of sports fans in Germany following soccer games [1]. Soccer fans not only support their favorite teams but also engage in lively debates about upcoming games and eagerly try to predict the outcomes. Despite many relying on gut feelings and instincts for predictions, leveraging data from past encounters and match statistics can enhance decision-making regarding potential winners. This approach is used by GoalGuru, a cloud application that uses machine learning to predict the outcomes of soccer games from the German soccer league. GoalGuru presents the user with a UI that allows for the selection of two teams from the German soccer league and a backend that leverages the capabilities of machine learning to generate a prediction. Users can thus predict the outcomes from a pairing of their choice amongst the available teams. GoalGuru also provides a range of additional information about the teams in question to the users, such as the outcomes of the last encounters between the two teams and the team's current form.

In the following sections of this report, the authors will first present the architectural goals in section III followed by

the architecture of GoalGuru in section IV. This involves an overview of the overall system together with the technology stack used as well as the different layers of the application and a look at the deployment. In section V, lessons learned and impediments regarding the development and deployment will be discussed. Lastly, section VI presents possible improvements for future work and concludes.

II. DATA ACQUISITION

To predict the outcomes of soccer games, a large amount of data about previous games is needed. We rely on data from the German website Transfermarkt [2], which provides soccer game data from all the major leagues in European soccer.

For making reasonable predictions, simply using match data is not enough, as this does not give a hint about the strength of the two teams involved. To account for this, we also scraped data from Transfermarkt to enrich the collected data with additional information about the teams, such as the league position and the market value of each team at the time of the game.

To provide users with more background information about the prediction, we also query the OpenLigaDB-API [3] for additional data, like for example the outcomes of the last five games for a team. To sum up, our data acquisition process consists of Web scraping match data alongside additional team information from Transfermarkt.

III. ARCHITECTURAL GOALS

GoalGuru gets data from external sources, which are used to train a machine learning model for predicting soccer game outcomes. The user can interact with GoalGuru by sending a prediction request and getting back a prediction result alongside additional team information. The data sources used by GoalGuru are:

- Transfermarkt website for match data from the past years
- OpenLigaDB-API for additional information (team form)

Figure 1 shows the system context of GoalGuru.

The app should respond quickly to incoming requests from the user and provide an intuitive user interface. The shown

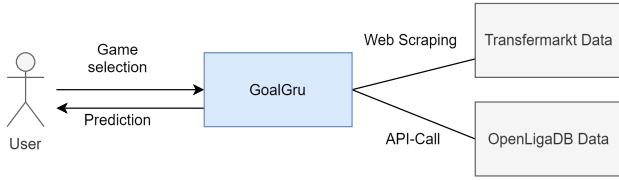


Figure 1. System context diagram for GoalGuru

prediction should be explained and backed up with a tabular display of additional information about both teams.

GoalGuru follows a monolithic architecture, with a Python backend and React frontend components packaged together in a single Docker [4] container for simplified deployment and scaling. This can be split up into dedicated services in further development stages. A RESTful [5] API promotes loose coupling between the backend and frontend, enabling future integration with other clients or services. Python is the programming language of choice for the backend because it is well suited for machine learning tasks. React is used for the frontend since it is a good fit for creating interactive user interfaces and provides a vibrant ecosystem.

The next section takes a closer look at GoalGuru’s architecture.

IV. ARCHITECTURE OF GOALGURU

A. Overall System

The overall architecture of GoalGuru concerning the building block view together with the deployment view can be broken down into several key blocks:

- **Frontend:** A single-page application using client-side rendering built with React. The frontend is responsible for user interaction and displaying predictions.
- **Backend:** Powered by FastAPI [6], a modern Python web framework. The backend handles requests from the frontend, processes data, and interacts with the database and machine learning model.
- **Machine Learning Model:** A Python-based predictive model that analyzes team data to generate game outcome predictions.
- **Database:** TinyDB [7], a lightweight document-oriented database, is used for data persistence. It stores team information and other relevant data required for predictions.
- **Data Acquisition:** Web scraping techniques are employed to gather data from Transfermarkt, ensuring up-to-date information for the prediction model.
- **Deployment:** The application is containerized using Docker and deployed on Amazon Web Services (AWS). This setup ensures easy scalability and management of the application.

The system follows a client-server architecture, with the React frontend serving as the client and the FastAPI backend as the server. This separation of concerns allows for independent development and scaling of the frontend and backend components. Data flow in the system typically follows this pattern:

- 1) User selects teams through the frontend interface.
- 2) Frontend sends a request to the backend API.
- 3) Backend retrieves necessary data from TinyDB.
- 4) Data is processed through the machine learning model.
- 5) Prediction results are sent back to the frontend for display.

This architecture ensures seamless communication between components while maintaining a clear separation of responsibilities. The use of cloud services and containerization provides flexibility for future expansions, such as integrating different machine learning models or adding more teams to the prediction system.

GoalGuru thus can be described as a three-layer application (frontend for presentation, backend for processing frontend requests, and database for persistence) running on a single tier (the Docker container).

In the following sections, the static system components will be described in more detail. We start with the frontend in the presentation layer (subsection IV-B), followed by the backend in the application layer (subsection IV-C) and the data persistence (subsection IV-D). The architecture section closes with a detailed look at the infrastructure and deployment of GoalGuru in subsection IV-E.

B. Frontend

The GoalGuru frontend is a single-page application built with React [8] using the local development server Vite [9]. It consists of a parent component for the app, which comprises several child components. These components are subject to a functional division, with each component being responsible for the rendering of a different part of the frontend. The app component holds the global state of the two selected teams, which is passed down to the child components as properties. Using conditional rendering, the child components showing the prediction and team info only appear when the user has two teams selected and runs the prediction. The app uses React Router [10] to display different routes on the frontend:

- `/`: This is the main page of our application that lets user predict the outcomes of soccer games of their choosing.
- `/gameday`: This route displays the matches from the last gameday in the soccer league.
- `/match/:id`: This is the details page that gets shown when clicking on a match. It renders match details alongside a prediction and an evaluation if the prediction was correct or not.

These routes are defined inside the parent component. If none of the above routes match, the app shows “404 Not Found”.

In the following, the child components are described in more detail regarding the functional division of the frontend.

- 1) **AppNavBar:** This component renders the navigation bar that consists of the GoalGuru logo on the left-hand side, links to the routes, and clickable icons on the right-hand side. The icons allow users to change the language (currently EN and DE are supported) as well as the theme (light and dark mode) of the app. There is also an info icon that shows a modal with information about GoalGuru and how the predictions work when being clicked.

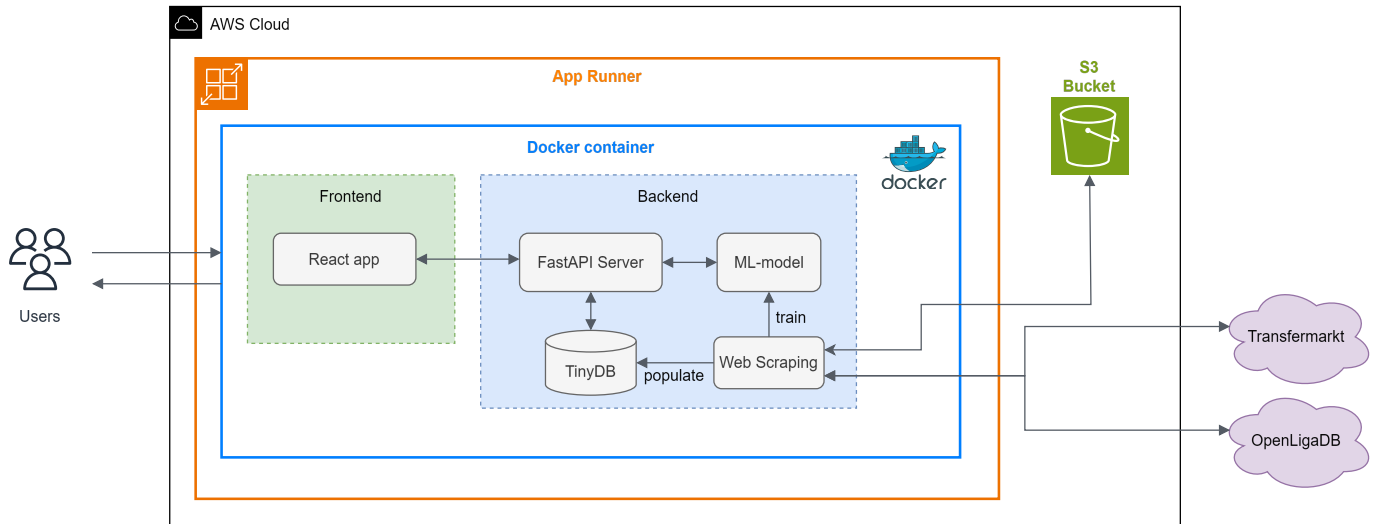


Figure 2. GoalGuru architecture overview: The architecture comprises a React frontend together with a Python backend that uses a document-oriented database, a FastAPI server, a machine learning model, and a web scraping script

- 2) *SoccerIcon*: In the background of the app, a soccer ball icon is displayed. It comes with a scroll-based animation which results in the soccer ball “rolling” to the bottom right corner when a user starts scrolling.
- 3) *GameSelect*: This component renders the select inputs together with the button for predicting the game between the two selected teams. It contains code to check that the prediction button is only active when two different teams are selected. When clicking on the button, *GameSelect* changes the global state (consisting of the two selected teams) in the parent component that is then used by the following child components.
- 4) *GamePrediction*: The *GameSelect* component is responsible for displaying the prediction result of the selected soccer game. It renders a bar consisting of three areas: a green area for the win probability, an orange area for the draw probability, and a red area for the loss probability for the first team. The area with the highest percentage is highlighted. Users can hover over an area to see a tooltip describing the area.
- 5) *TeamMatches*: Here we show a table of the last five matches of each team in the corresponding soccer league. It shows The results of these games as well as a graphical representation of the results (colored, rounded shapes to indicate win, draw, or lose). We make a fetch request to the OpenLigaDB API to retrieve this data.
- 6) *MatchesAgainst*: This component shows a table of up to five of the last matches between the two selected teams against each other. The frontend makes a request to our backend querying the database for matches that involve both of the selected teams and then renders the data in a table.
- 7) *GamedayInfos*: This component is displayed on the /gameday route and displays cards for the last games in the soccer league. When clicking on a card, users get directed

to a details page.

To use consistent styling, we opted for the React bootstrap library [11] that provides React components with Bootstrap [12] styles. This makes it easy to include form elements, tables, buttons, and other styled elements that match the styling of the frontend. Icons are used from the React Icons [13] library.

The frontend has multi-language support (DE and EN). This feature is implemented using the internationalization framework react-i18next [14]. Inside our frontend code, we added a directory called `translations` that consists of two JSON files (German and English versions) that hold the text strings in the corresponding language.

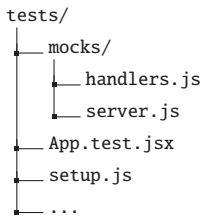
```
translations/
├── de.json
└── en.json
```

All the text content in our frontend is thus displayed dynamically depending on the selected language.

The frontend can be displayed in light mode or dark mode. This feature is implemented by storing the chosen mode in the state of the *AppNavBar* component and changing the `data-bs-theme` attribute on the body element of the HTML code inside a `useEffect` hook. This causes the Bootstrap styles to be changed according to the selected theme.

Frontend testing is implemented by using the testing framework Vitest [15] together with the React testing library [16]. This allows for rendering the components in a Node environment and thus testing if the components include the correct markup.

The frontend testing is located inside the `__test__` directory that contains Test files for the React components together with the test setup:



In order to test components that show data coming from an external API or our backend, we opted for the API mocking library MSW [17] together with the Axios mock adapter [18]. To use these mocking libraries, we chose the popular JavaScript library Axios [19] for the frontend data fetching. To mock the API calls, we created a mock server using MSW and implemented the routes to be intercepted together with dummy data to return, in the file `handlers.js`. This allows us to test if components render the correct markup when fetching data. We also mock requests directly inside the components with the mock adapter.

C. Backend

Our backend uses Python as the programming language and consists of web scraping, machine learning, a server with API endpoints, and a database. Communication between these components in the backend is facilitated by utility functions that provide the glue code for our backend. The main parts of our backend are explained in detail below. The database will be described in section IV-D.

Web Scraping

The web scraping part is implemented in a python script, `webscraping.py`, which contains methods specifically designed to gather data for our machine learning model.

The core components utilized for web scraping are the libraries `requests` [20] and `beautifulsoup4` [21]. These libraries facilitate sending requests to the Transfermarkt website server and processing the returned HTML content.

In addition to generating CSV files for model training, we also create JSON files that are subsequently imported into a TinyDB instance. This approach ensures sufficient speed when querying large datasets.

The `create_dataframe_model_one` method in the web scraping script generates the training data required for our machine learning models. The data collected includes:

- Name of the home team
- Name of the away team
- The result of the match (win, draw, lose)
- The market value of the home team
- The market value of the away team
- Position of the home team in the table before the match
- Position of the away team in the table before the match

The collected data is then one-hot encoded and saved in a CSV file. Initially, we relied primarily on CSV files for data storage. However, in the further course of the project, we opted to store data such as team names, current market values, and past matches in JSON format in a TinyDB database instance. This is accomplished using the `generate_matchdata_json`

and `generate_team_json` methods, which are also located in the web scraping script.

Machine Learning Models

Three distinct model classes were developed and are contained within the `models.py` file in our backend:

- *ModelOne*: A simple model that makes predictions based on the length of the team names.
- *ModelTwo*: A logistic regression model trained on our scraped data. It utilizes various features such as market values and current team standings to calculate the probability of an outcome. Team names are processed using one-hot encoding. The model is implemented with the `scikit-learn` library [22] and uses the *One-versus-Rest* classification strategy (a binary problem is fit for each label of the target column).
- *ModelThree*: A decision tree model trained on our generated data. It employs the same features used in *ModelTwo* for training and is built using the `DecisionTreeClassifier` class from the `scikit-learn` library.

In the current version of our project, we are using *ModelTwo*. A comparison between *ModelTwo* and *ModelThree* is yet to be made.

The structure of the *ModelTwo* and *ModelThree* classes is very similar. In addition to a `predict` method for making predictions, there are also methods for saving and loading the models. Additionally, the `train` method allows for training the model, while the `accuracy` method enables evaluation of the model's performance on a specified dataset.

We currently use *ModelTwo* in our project, which makes use of logistic regression. We opted for this approach because it has some benefits over other machine learning models for predicting categorical data:

- *Explainable Results*: The coefficients of logistic regression are easily interpretable, providing clear insights into how various features impact the probability of an outcome.
- *Low Computational Requirements*: Logistic regression requires less computational power in comparison with more complex models like neural networks or random forests. This results in faster training and prediction times, which is particularly advantageous when handling vast datasets, ensuring high scalability.
- *Less Prone to Overfitting*: Logistic regression is less prone to overfitting compared to highly complex models such as deep decision trees or neural networks, making it a more robust choice for many applications.

In conclusion, our decision to utilize logistic regression for this project is motivated by its interpretability, computational efficiency, scalability, and robustness against overfitting.

Server

Communication between the front- and backend is implemented using a REST interface. The backend exposes three API endpoints that can be queried by the frontend for data retrieval in the presentation layer. The endpoints of this interface are explained in detail below.

/api/teams: This endpoint provides a list of all the teams available during the current season from the backend. This data is used to populate the select options in the frontend.

/api/matches?home_team=<home_team>&away_team=<away_team>: This endpoint provides data on the last five matches against each other for the specified home and away teams. It returns a list of up to five matches including fields for the home team, the away team, the goals scored by the home and away team, and the date of the game. The response data is used to display a table of the last games of the selected teams against each other in the frontend.

/api/predict?home_team=<home_team>&away_team=<away_team>: This endpoint provides prediction results for a match between two teams. The response includes the two teams along with the prediction probabilities made by the machine learning model:

```
{
  "teams": ["Team 1", "Team 2"],
  "probabilities": {
    "home": 0.65,
    "draw": 0.25,
    "away": 0.10
  }
}
```

The response data is used for rendering the prediction bar displaying the predicted probabilities for win, draw, and lose when a user makes a prediction.

Testing

The code in the backend is tested using the popular Python testing library Pytest [23]. Test coverage is generated with the help of the plugin *pytest-cov* [24]. Testing is implemented for the FastAPI web server, web scraping part, machine learning models (mainly ModelOne) as well as for utility functions that glue together server, models, and database.

D. Persistence

The data we get from our Python web scraping script gets saved in CSV format to an AWS S3 bucket [25] and is subsequently used to train the machine learning model.¹

The scraped data consists of the fields depicted in Table I.

Table I
DATA FOR MODEL TRAINING

Field	Description
Result	The result of the game (this is a string that can be "home", "away" or "draw").
MV_HT	The current market value of the home team.
MV_AT	The current market value of the away team.
POS_HT	The current league position of the home team.
POS_AT	The current league position of the away team.
HT	The name of the home team.
AT	The name of the away team.

To avoid re-training the model when the server is started, we train it once with the collected data and then store it using the

¹To properly train our machine learning model, we use one-hot encoding on the teams, so there are a lot more columns to the training data

joblib [26] library. With this approach, we have a pre-trained model in our backend that gets loaded when the server starts.

For persisting data that will be queried by the frontend, we use the document-oriented NoSQL database TinyDB [7], as our data is well-structured but does not benefit from a relational model. TinyDB is directly incorporated into our backend and uses a JSON file for data storage. The Goal Guru backend uses methods provided by the TinyDB library for data insertion and querying, thus directly communicating with the database.

We store data for the past matches of the available soccer teams, as well as data for the team names. The fields of this data are shown in Tables II and III respectively.

Table II
DATA ON MATCHES

Field	Description
Home	The name of the first team (home team).
Away	The name of the second team (away team).
Goals_Home	Goals scored by the home team.
Goals_Away	Goals scored by the away team.
Date	The date when the match was played.

Table III
DATA ON TEAMS

Field	Description
Team	The name of the team.
Market_Value	The market value for the selected team.
ID	The ID of the team from the Transfermarkt site.

The data in Table II consists of match information and is queried by the frontend via backend API-calls for retrieval of the match data between the two selected teams. The data in Table III only consists of three fields: team name, market value, and team ID. These tables are used to provide a list of teams to select from the frontend, and to enrich the data we get from the frontend when a user wants to predict a game with the current market value of these teams.

E. Infrastructure and Deployment

GoalGuru is a containerized application that runs in a single Docker [4] Container. This container exposes the backend server. The frontend is statically served from the backend, and the FastAPI server also contains the TinyDB database.

In order to build the image for the container, we use a Dockerfile with a multi-stage build. The three stages are

- 1) frontend-build
- 2) backend-build
- 3) final

The first stage, named "frontend-build", uses a Node.js Alpine image to build the React frontend. It installs dependencies, copies the frontend code, and builds the production-ready assets. The second stage, "backend-build", uses a Python slim image to set up the FastAPI backend, installing the required Python packages. The final stage, based on the backend build, copies the frontend build artifacts from the first stage into the backend directory. It sets an environment variable for the frontend directory, exposes port 8000, and specifies the command to

start the FastAPI application using the Python-based ASGI web server Uvicorn [27]. This multi-stage approach results in a smaller final image that contains only the necessary components for running the application, eliminating build-time dependencies and reducing the overall image size.

The built image is deployed to the container registry on Amazon Web Services, AWS ECR [28]. From there, our app is deployed as a Docker container using AWS App Runner [29].

The deployment of GoalGuru is automated with a CI/CD workflow (see Figure 3). Therefore, we make use of GitHub Actions [30] to define a workflow inside a YAML file that will run a job to build, push, and deploy our Docker image. The workflow consists of the following jobs:

- 1) Checkout repository
- 2) Configure credentials
- 3) AWS ECR Login
- 4) Build image from Dockerfile and push to ECR

This workflow is triggered every time code is pushed to the prod branch of the GoalGuru GitHub repository. To push the built image to AWS ECR, we supply the access key and access key ID of our AWS IAM user to GitHub secrets. With this approach, no credentials are visible in the Docker runtime as GitHub secrets are stored encrypted. For the CD part of the CI/CD workflow, our App Runner instance is configured to always use the latest Docker image in the registry. This ensures that the app gets redeployed whenever we push on the prod branch and the triggered workflow is successful.

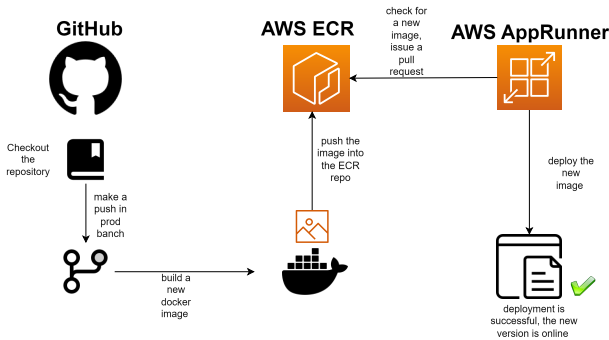


Figure 3. CI/CD workflow to build and deploy our app

V. DISCUSSION

The development of GoalGuru provided valuable insights and presented several challenges, offering important lessons for future projects.

With little prior experience in the development team regarding cloud development, understanding and implementing AWS components required significant time and effort. This experience underscores the importance of allocating sufficient time for learning new technologies in project planning.

Setting up GitHub Actions for continuous integration and deployment to AWS proved more complex than anticipated, particularly regarding credential management. This emphasized

the need for a thorough understanding of security best practices in CI/CD pipelines.

Also, finding a comprehensive API for soccer data was challenging. The decision to use web scraping from Transfermarkt and OpenLigaDB was a pragmatic solution but highlighted the importance of data source evaluation early in the project.

Another impediment at the early development stages was the scope of the Model Training Data: Balancing the need for comprehensive historical data to provide predictions for a wide range of teams proved challenging. This was particularly evident when considering national teams or lower-division clubs with limited available data. Ultimately, the decision to only include teams from the current campaign in the first national league was made, but when expanding the list of teams to choose from, this is an important aspect that should be kept in mind for future development.

These challenges, together with the resulting lessons learned, provide valuable insights for future iterations of GoalGuru and similar projects.

VI. CONCLUSION AND FUTURE WORK

GoalGuru demonstrates the potential of combining web technologies, machine learning, and cloud services for soccer prediction. The current implementation presents a fully functional MVP. Several enhancements regarding the technical implementation are proposed for future development:

- Migrate from TinyDB to a bigger NoSQL database for improved scalability and performance. We can then also run our database in a dedicated container (this ensures consistency across environments) and use Docker Compose [31] for building and orchestrating our services.
- Move the frontend to a separate S3 bucket for optimized content delivery. This would decouple the frontend from the backend and thus favor the principle separation of concerns.
- Rework the web scraping scripts as a service that runs inside a dedicated container. This would make our system more versatile and would move the overall architecture towards a microservice approach.
- Automation of data updates and model re-training using AWS Lambda [32] and AWS EventBridge [33]. This could also be achieved by enhancing our GitHub Actions workflows to trigger web scraping and model training with the newly scraped data on every push to our prod branch.
- Integrating AWS CloudFront [34] in our web app will improve performance by reducing latency through global content caching and enhance security with integrated DDoS protection.
- Integrating AWS EKS [35] will allow us to easily scale our containerized application across multiple nodes, ensuring high availability and reliability as our user base grows.

These improvements would enhance GoalGuru's technical infrastructure and expand its functionality to move towards a microservice architecture with services running inside dedicated containers.

REFERENCES

- [1] *Infographic: The Global Game of Football*. Statista Daily Data. May 24, 2024. URL: <https://www.statista.com/chart/14329/global-interest-in-football> (visited on 06/23/2024).
- [2] *Fußball-Transfers, Gerüchte, Marktwerte und News*. URL: <https://www.transfermarkt.de/> (besucht am 18.06.2024).
- [3] *OpenLigaDB*. URL: <https://www.openligadb.de/> (visited on 06/18/2024).
- [4] Docker. *Docker: Accelerated Container Application Development*. [Online]. URL: <https://www.docker.com/>.
- [5] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000.
- [6] Sebastián Ramírez. *FastAPI*. [Online]. URL: <https://fastapi.tiangolo.com/>.
- [7] *Welcome to TinyDB! — TinyDB 4.8.0 documentation*. URL: <https://tinydb.readthedocs.io/en/latest/> (visited on 06/22/2024).
- [8] Facebook. *React*. [Online]. URL: <https://react.dev/>.
- [9] Evan You. *Vite: Next Generation Frontend Tooling*. [Online]. URL: <https://github.com/vitejs/vite>.
- [10] *Home v6.24.0*. URL: <https://reactrouter.com/en/main> (visited on 06/28/2024).
- [11] *React Bootstrap | React Bootstrap*. URL: <https://react-bootstrap.netlify.app/> (visited on 06/22/2024).
- [12] Mark Otto und Jacob Thornton. *Bootstrap: Build fast, responsive sites*. [Online]. URL: <https://getbootstrap.com/>.
- [13] *React Icons*. URL: <https://react-icons.github.io/react-icons/> (visited on 06/29/2024).
- [14] *Introduction | react-i18next documentation*. Jan. 29, 2024. URL: <https://react.i18next.com> (visited on 06/22/2024).
- [15] Anthony Fu and Matías Capeletto. *Vitest: A Vite-native unit test framework*. [Online]. URL: <https://vitest.dev/>.
- [16] *React Testing Library | Testing Library*. June 3, 2024. URL: <https://testing-library.com/docs/react-testing-library/intro/> (visited on 06/22/2024).
- [17] *Mock Service Worker*. URL: <https://mswjs.io/> (visited on 06/22/2024).
- [18] *axios-mock-adapter*. npm. Sept. 11, 2023. URL: <https://www.npmjs.com/package/axios-mock-adapter> (visited on 06/24/2024).
- [19] *Axios*. URL: <https://axios-http.com/> (visited on 06/24/2024).
- [20] *requests: Python HTTP for Humans*. Version 2.32.3. URL: <https://requests.readthedocs.io> (visited on 06/30/2024).
- [21] *beautifulsoup4: Screen-scraping library*. Version 4.12.3. URL: <https://www.crummy.com/software/BeautifulSoup/bs4/> (visited on 06/30/2024).
- [22] *scikit-learn: machine learning in Python — scikit-learn 1.5.0 documentation*. URL: <https://scikit-learn.org/stable/index.html> (visited on 06/30/2024).
- [23] Holger Krekel and pytest Entwicklungsteam. *Pytest*. [Online]. URL: <https://pytest.org/>.
- [24] Holger Krekel and pytest Entwicklungsteam. *pytest Coverage Reports*. [Online]. URL: <https://pypi.org/project/pytest-cov/>.
- [25] *Amazon Simple Storage Service S3 – Cloud Online-Speicher*. Amazon Web Services, Inc. URL: <https://aws.amazon.com/de/s3/> (besucht am 24.06.2024).
- [26] *Joblib: running Python functions as pipeline jobs — joblib 1.4.2 documentation*. URL: <https://joblib.readthedocs.io/en/stable/> (visited on 06/22/2024).
- [27] Sebastián Ramírez. *Uvicorn*. [Online]. URL: <https://www.uvicorn.org/>.
- [28] *Vollständig verwaltete Container-Registry – Amazon Elastic Container Registry – Amazon Web Services*. Amazon Web Services, Inc. URL: <https://aws.amazon.com/de/ecr/> (besucht am 22.06.2024).
- [29] *Verwalteter Container-Anwendungsdienst – AWS App Runner – Amazon Web Services*. Amazon Web Services, Inc. URL: <https://aws.amazon.com/de/apprunner/> (besucht am 22.06.2024).
- [30] *GitHub Actions documentation*. GitHub Docs. URL: https://docs.github.com/_next/data/Fsuma9vZHsYur7KkeZyWX/en/free-pro-team@latest/actions.json?versionId=free-pro-team%40latest&productId=actions (visited on 07/01/2024).
- [31] Docker. *Compose: Defining and Running Multi-Container Docker Applications*. [Online]. URL: <https://docs.docker.com/compose/>.
- [32] *AWS Lambda Data Processing - Datenverarbeitungsdienste*. Amazon Web Services, Inc. URL: <https://aws.amazon.com/de/lambda/> (besucht am 24.06.2024).
- [33] *Serverless Event Router – Amazon EventBridge – Amazon Web Services*. URL: <https://aws.amazon.com/de/eventbridge/> (visited on 06/24/2024).
- [34] *What is Amazon CloudFront? - Amazon CloudFront*. URL: <https://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/Introduction.html> (visited on 06/30/2024).
- [35] *What is Amazon EKS? - Amazon EKS*. URL: <https://docs.aws.amazon.com/eks/latest/userguide/what-is-eks.html> (visited on 06/30/2024).

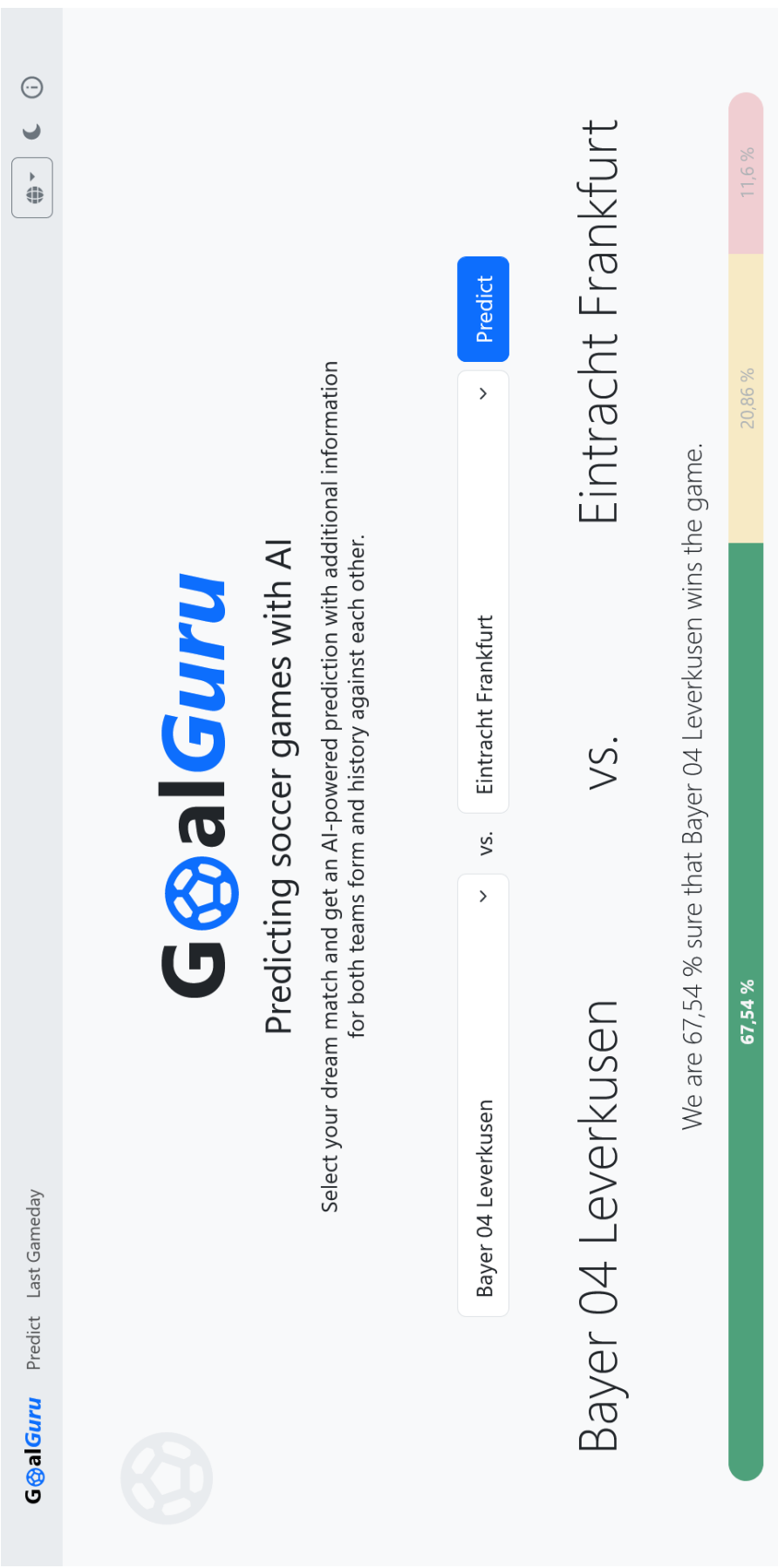


Figure 4. The GoalGuru frontend main route with a prediction for a game between “Bayer 04 Leverkusen” vs. “Eintracht Frankfurt” (the additional team information is located below the shown area)