Bier Pongo

Kevin Fischer   
*OTH Amberg-Weiden*  
k.fischer2@oth-aw.de

Simon Kleber  
*OTH Amberg-Weiden*   
s.kleber1@oth-aw.de Markus Fleischmann  
*OTH Amberg-Weiden*   
m.fleischmann2@oth-aw.de

Fabian Klinger  
*OTH Amberg-Weiden*   
f.klinger1@oth-aw.de

Christian Renner  
*OTH Amberg-Weiden*   
c.renner@oth-aw.de Darren Fürst  
*OTH Amberg-Weiden*   
d.fuerst@oth-aw.de

Jakob Lindner  
*OTH Amberg-Weiden*   
j.lindner3@oth-aw.de

*Abstrakt*— Cloudfähige, skalierbare, multiuser- und multidevice-fähige Bier Pong Social Media Platform.

Schlüsselwörter—Beer Pong, Cloud, Data, Team

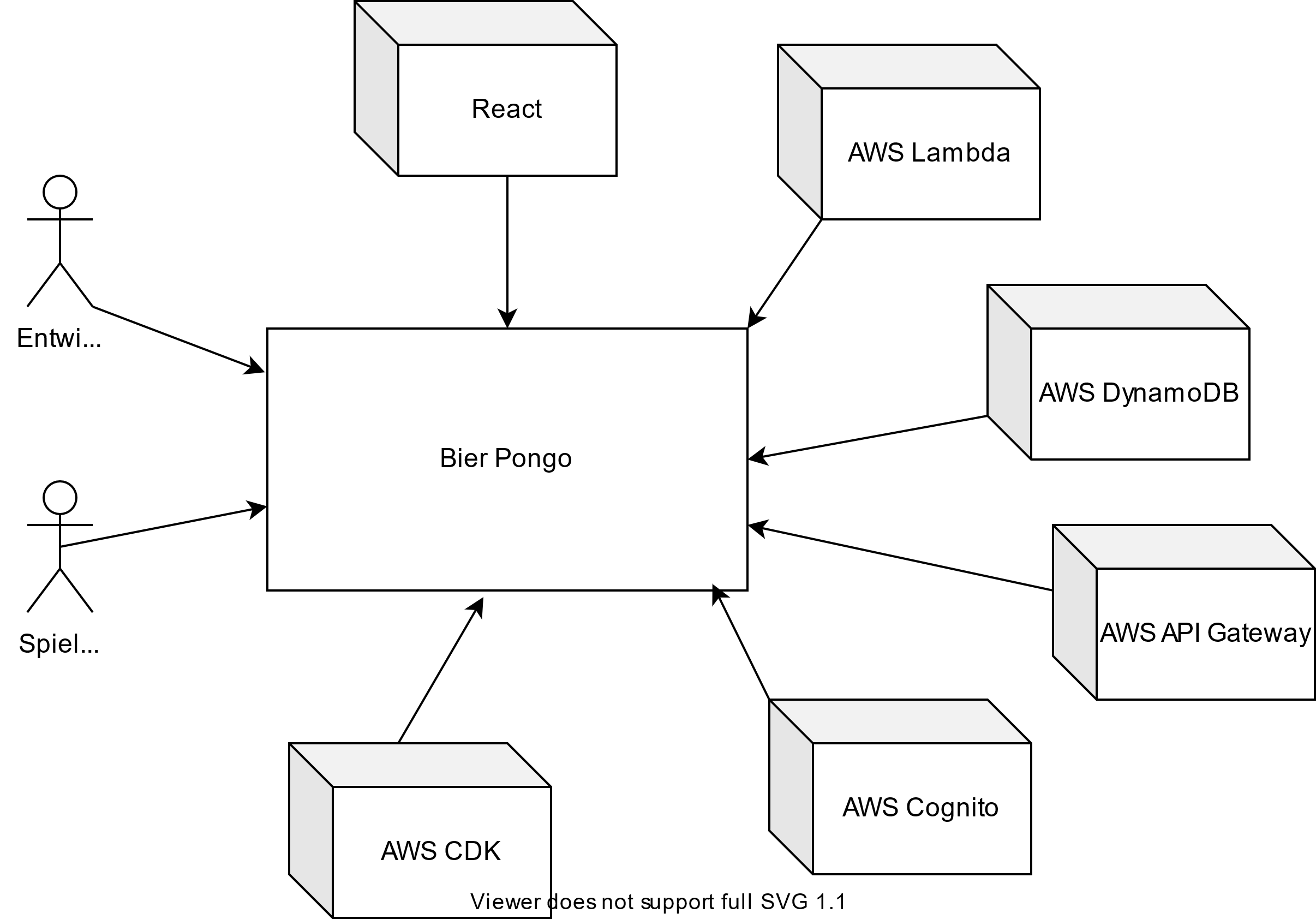
# Problemstellung

## Mission statement

Bier Pongo ist eine Webanwendung, welche es Bierpong-Spielern ermöglicht, auch räumlich voneinander getrennt zusammen zu spielen und immer den aktuellen Stand des Spielfelds vor sich zu haben. Benötigt wird dafür nur ein Gerät mit Webbrowser und Internetverbindung. Bier Pongo maximiert den Spielspaß dank intuitiver Bedienung, die keiner ausführlichen Erklärung bedarf.

## Alleinstellungsmerkmal

Im Folgenden wird das Gesamtsystem als Blackbox dargestellt. Alle verwendeten externen Systeme werden als Box um die Anwendung "Bier Pongo" herum dargestellt. Der menschliche Akteur "Spieler" kann in diesem Fall als Plural aufgefasst werden, da an einem Bierpong-Spiel immer zwei Spieler beteiligt sind.



Für die Entwicklung des Frontend wird die JavaScript Library "React" verwendet. Das Backend bilden die Dienste "Lambda" und "DynamoDB" von AWS. Als Schnittstelle dient der Dienst "AWS API Gateway", um Daten im jeweilig gewünschten Format abzurufen. Für die Benutzerverwaltung wird "AWS Cognito" verwendet. Um alle Dienste von AWS per Infrastructure as Code zu orchestrieren, wird das "AWS CDK" (Cloud Development Kit) benutzt.

TODO: Wo wird der React-Projektordner abgelegt? (EC2, S3)

## Qualitätsziele

|  |  |
| --- | --- |
| **Qualitätsziel** | **Beschreibung** |
| Gute Benutzbarkeit | Bier Pongo ist für die Spieler intuitiv zu bedienen und ohne Erklärung spielbar. |
| Hohe Zuverlässigkeit | Das System steht den Spielern jederzeit zur Verfügung und bildet den Spielstand live ab. |
| Gute Wartbarkeit | Bier Pongo ist ohne großen Administrationsaufwand betreibbar und leicht um zusätzliche Features erweiterbar. |

## Enscheidende Rahmenbedingungen

|  |  |
| --- | --- |
| **Vorgabe** | **Beschreibung** |
| Zeitlicher Rahmen | Das Projekt ist innerhalb von 5 Kalenderwochen abzuschließen. |
| Technischer Anspruch | Das Projekt muss mind. einen der Themenbereiche BigData / Cloud / NoSQL in einer hinreichend technischen Tiefe bedienen. |
| Einfachheit in der Anwendung | Die Anwendung soll aber so einfach gehalten werden, dass sie im gegebenen Zeitraum umgesetzt werden kann. |

# Lösungsstrategie

## Lösungsansätze

Die zuvor benannten Qualitätsziele wurden beim Projekt Bier Pongo folgendermaßen brücksichtigt.

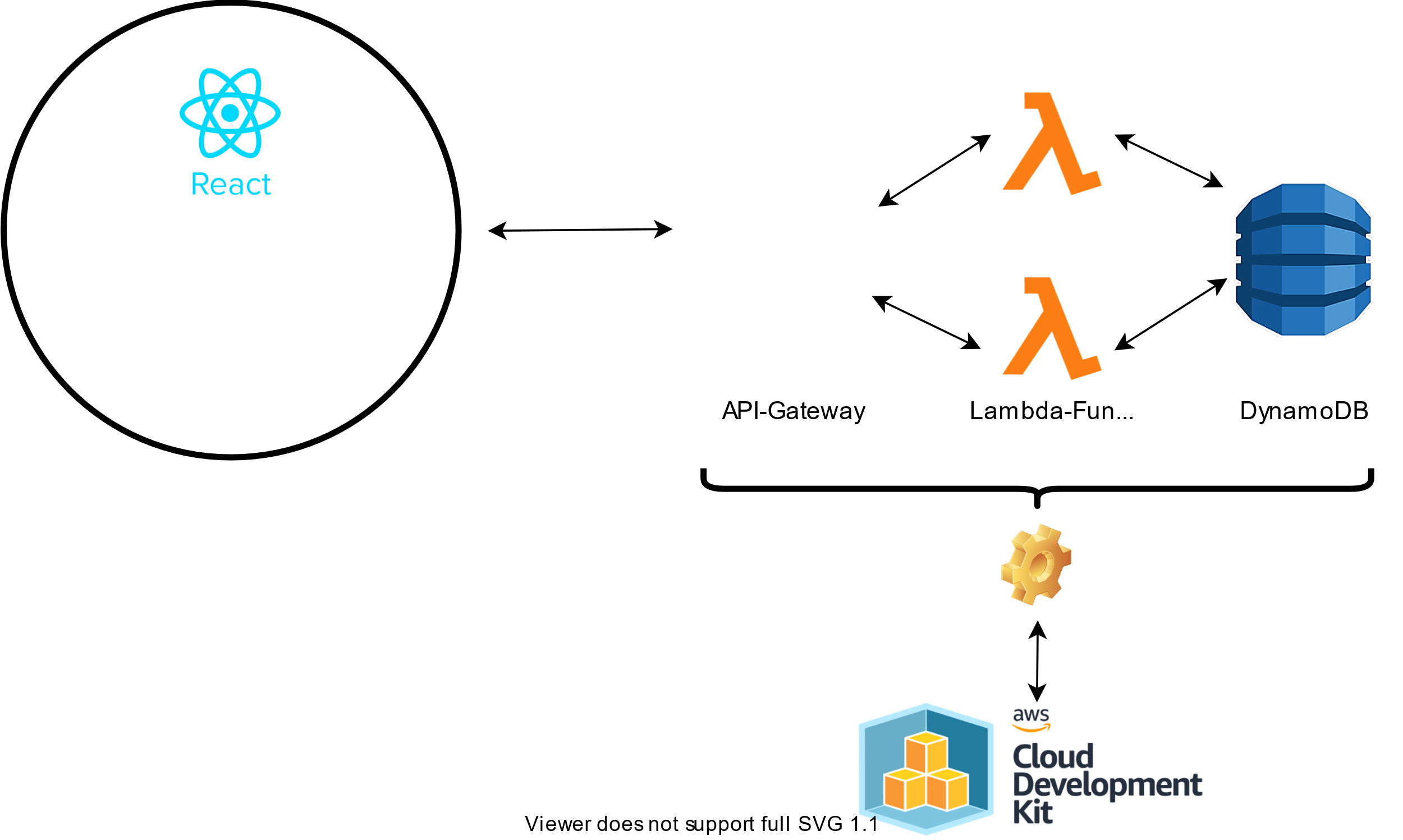
|  |  |
| --- | --- |
| **Qualitätsziel** | **Lösungsansatz** |
| Gute Benutzbarkeit |  Dank der simpel gehaltenen React-Oberfläche kann die Anwendung direkt ohne Tutorial o.Ä. benutzt werden.   Kennt man die Regeln des "Real-Life"-Spiels kommt man ohne Einweisung direkt zurecht.   Die Benutzer benötigen lediglich einen Webbrowser. |
| Hohe Zuverlässigkeit |  Da sämtliche Infrastruktur durch Dienste von AWS bereitgestellt wird, ist die Anwendung praktisch ohne zu erwartende Ausfälle verfügbar.   Den einzigen Flaschenhals stellt die Internetverbindung des Benutzers dar. |
| Gute Wartbarkeit |  Durch Nutzung des AWS Clous Development Kit können etwaige Änderungen an der Infrastruktur im Code getätigt und anschließend deployed werden.   Zusätzliche Features können für das separate React-Frontend entwickelt werden, ohne dadurch das Backend auf AWS zu beeinflussen. |

## Technologiestack

Das Frontend von Bier Pongo besteht aus einer React-Anwendung in einem Docker-Container.

Die Backend-Infrastruktur besteht aus Diensten von AWS. Die Brücke zwischen React und AWS bildet eine REST-API, realisiert durch AWS API-Gateway. Die zur Verfügung gestellten Endpoints nehmen API-Aufrufe von Seiten React entgegen und bedienen diese, indem AWS Lambda-Funktionen aufgerufen werden. Durch die Lambda-Funktionen werden die Spiel-Daten aus der AWS DynamoBD gelesen und geschrieben.

Für die Verwaltung der AWS Infrastruktur wird per Infrastructure as Code das AWS Cloud Development Kit verwendet.



# Lösungsdetails

## Architekturentscheidung

Die Wahl des Frontend-"Frameworks" fiel auf die JavaScript-Library React, da sich mit ihr leicht moderne User Interfaces erstellen lassen und im Entwicklerteam schon Erfahrungen mit React und JavaScript im Allgemeinen vorhanden waren. Für React sprach auch die Modularität mittels Components. Diese sind wiederverwertbar und austauschbar und konnten unabhängig voneinander entwickelt werden.

Damit die Applikation von allen Entwicklern ohne Kompatilitätsprobleme auf z.T. unterschiedlichen Betriebssystemen gebaut, entwickelt und getestet werden kann, lag es nahe dafür einen Docker-Container zu verwenden. Auch nach der Entwicklungszeit ist geplant, den Docker-Container auf AWS Elastic Container Service zu hosten.

Das Backend wird komplett durch Dienste von AWS gebildet. Sämtliche Services werden über das AWS Cloud Development Kit per Infrastructure as Code definiert.

Das AWS API-Gateway stellt verschiedene Endpoints zur Verfügung, damit die React-App auf Spiel-Daten zugreifen kann. Für die API-calls wird die Bibliothek axios verwendet. Das API-Gateway ruft abhängig vom Typ der Anfrage die entsprechende Lambda-Funktion auf.

Alle Lambda-Funktionen sind in der Programmiersprache Python geschrieben, da das Team hier auch schon Erfahrung hatte. Mithilfe der Python-Bibliothek boto3 werden die Spieldaten, welche in einer Datenbanktabelle der AWS DynamoBD liegen, in der Lambda-Funktion gelesen und geschrieben.

In der AWS DynamoDB Datenbank liegt lediglich eine Tabelle, welche für jedes Spiel die Spiel-ID sowie den Spielstand (getroffene Becher) speichert. Die Daten gelangen anschließend über das API-Gateway zurück zur aufrufenden React-App. Entsprechend gilt der beschriebene Weg für das Updaten eines Spiels.

## Verhalten

Im Folgenden wird der Ablauf eines Spiels beispielhaft in Stichpunkten beschrieben:

* Spieler 1 öffnet die URL und erstellt ein Spiel.
* Über den "post"-Endpoint wird das API-Gateway dazu aufgefordert, die entsprechende post-Lambda-Funktion aufzurufen.
* Die Lambda-Funktion erstellt eine zufällige, eindeutige Spiel-ID und speichert diese in der Datenbanktabelle ab.
* Spieler 2 öffnet die URL und tritt dem von Spieler 1 erstellten Spiel durch Eingabe der Spiel-ID bei
* Über den "join"-Endpoint wird das API-Gateway dazu aufgefordert, die entsprechende join-Lambda-Funktion aufzurufen.
* Die Lambda-Funktion erhöht die Spieler-Anzahl des entsprechenden Spiels um 1
* Nun nutzen beide Spieler die gleiche URL, welche die Spiel-ID enthält, um zu spielen
* **TODO: ANZEIGE DES AKTIVEN SPIELERS? SPIELER-ID?**
* Spieler 1 trifft beispielsweise einen Becher, klickt diesen an und beendet anschließend seinen Zug
* **TODO: WIE SIEHT DER STRING AUS, DER ÜBERGEBEN WIRD?**
* String wird per API-call dem API-Gateway übergeben.
* Der API-Gateway ruft die put-Lambda-Funktion auf, um die Spieldaten in der Datenbank zu aktualisieren
* Dabei wird dem State (Spielstand) der String angehängt
* Spieler 1 und 2 spielen abwechselnd, bis eine Seite alle Becher getroffen hat
* **TODO: WAS PASSIERT, WENN DAS SPIEL BEENDET IST?**

# Fazit und Ausblick

## Derzeitiger Stand

Im Folgenden sind die im Konzeptpapier angegebenen User-Stories aufgelistet:

 MUSS

* Spiel erstellen & beitreten - geschafft
* Spiel eintragen - geschafft

 SOLL

* Userverwaltung - nicht geschafft
* Historie - nicht geschafft

 KANN

* Historie - nicht geschafft
* Spiel ansehen – nicht geschafft
* Passwort vergessen – nicht geschafft

## Nächste Schritte

Die nächsten Schritte des Projekts werden hier aufgelistet:

 Fertigstellung der geplanten Features

 Zukünftiges hosten des Docker-Containers in AWS ECS