Duale Hochschule Baden-Württemberg Mannheim

**Praxisarbeit**

# Konzeption und Implementierung einer Companion App für Arc Raiders.

**Studiengang Informatik**

**Studienrichtung Informationstechnik**


| | |
|---|---|
| Verfasser(in): | Paul Wegfahrt |
| Matrikelnummer: | 2415837 |
| Kurs: | TINF23IT1 |
| Studiengangsleiter: | Prof Dr. Gerhards |
| Wissenschaftlicher Betreuer: | Jürgen Schultheis |
| Bearbeitungszeitraum: | 14.10.2025 – 14.04.2026 |
| Eingereicht am: | 14.04.2026 |

# Ehrenwörtliche Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit mit dem Titel "*Konzeption und Implementierung einer Companion App für Arc Raiders.*" selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Ort, Datum Paul Wegfahrt

# Sperrvermerk

Der Inhalt dieser Arbeit darf weder als Ganzes noch in Auszügen Personen außerhalb des Prüfungsprozesses und des Evaluationsverfahrens zugänglich gemacht werden, sofern keine anders lautende Genehmigung der Ausbildungsstätte vorliegt.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Quelltextverzeichnis

# Abkürzungsverzeichnis

| | |
|---|---|
| **DHBW** | Duale Hochschule Baden-Württemberg |
| **PvPvE** | Player versus Player versus Entities |
| **API** | Application Programming Interface |
| **TTI** | Time to Interactivity |
| **MVP** | Minimum Viable Product |
| **ER** | Entity-Relationship |
| **MCDA** | Multi-Criteria Decision Analysis |
| **DDD** | Domain-Driven Design |
| **BEM** | Block Element Modifier |
| **OOCSS** | Object Oriented CSS |
| **FCP** | First Contentful Paint |
| **MUI** | Material-UI |
| **SPA** | Single Page Application |
| **DAG** | Directed Acyclic Graph |
| **SSG** | Static Site Generation |
| **SSR** | Server-Side Rendering |
| **CSR** | Client-Side Rendering |
| **ISR** | Incremental Static Regeneration |
| **PPR** | Partial Prerendering |
| **SEO** | Search Engine Optimization |
| **UI** | User Interface |
| **UX** | User Experience |
| **HTML** | Hypertext Markup Language |
| **CSS** | Cascading Style Sheets |
| **JIT** | Just-In-Time |
| **NPM** | Node Package Manager |
| **CLI** | Command Line Interface |
| **CVA** | Class Variance Authority |
| **ARIA** | Accessible Rich Internet Applications |
| **WAI** | Web Accessibility Initiative |

| | |
|---|---|
| **SWR** | Stale-While-Revalidate |
| **BaaS** | Backend as a Service |
| **FaaS** | Function as a Service |
| **REST** | Representational State Transfer |
| **ORM** | Object-Relational Mapping |
| **RLS** | Row-Level Security |
| **CI** | Continuous Integration |
| **CD** | Continuous Deployment |
| **INVEST** | Independent, Negotiable, Valuable, Estimable, Small, Testable |
| **SMART** | Specific, Measurable, Achievable, Relevant, Time-bound |
| **MoSCoW** | Must-Have, Should-Have, Could-Have, Won't-Have |
| **VPS** | Virtual Private Server |
| **MPA** | Multi-Page Application |
| **RSC** | React Server Components |
| **IAM** | Identity and Access Management |
| **SaaS** | Software as a Service |
| **PaaS** | Platform as a Service |
| **IaaS** | Infrastructure as a Service |
| **CDN** | Content Delivery Network |
| **ER** | Entity-Relationship |
| **CTE** | Common Table Expression |
| **JSONB** | JavaScript Object Notation Binary |
| **DAG** | Directed Acyclic Graph |
| **SUS** | System Usability Scale |
| **DAU** | Daily Active Users |
| **E2E** | End-to-End |

# Abstract

Abstract...

# Zusammenfassung

Zusammenfassung....

# 1 Einleitung

## 1.1 Motivation

Der globale Gaming-Markt verzeichnet ein beispielloses Wachstum: Mit einer Bewertung von 298,98 Milliarden USD im Jahr 2024 wird prognostiziert, dass der Markt bis 2030 auf 600,74 Milliarden USD anwachsen wird [52]. Diese Entwicklung zeigt deutlich, dass Anwendungen in diesem Bereich ein großes Potenzial haben.

Innerhalb dieses dynamischen Marktes hat sich der Extraction Shooter als besonders anspruchsvolles und komplexes Genre etabliert. Spiele wie Escape from Tarkov und Hunt: Showdown definieren dieses Genre durch ihre charakteristischen Merkmale: hochriskante Player versus Player versus Entities (PvPvE)-Gameplay-Mechaniken, komplexe Progressionssysteme mit zahlreichen Quest-Lines, ressourcenbasierte Upgrade-Systeme und die permanente Gefahr des Verlusts aller mitgeführten Items bei einem Spieltod. Arc Raiders, entwickelt von Embark Studios und im Oktober 2025 veröffentlicht, positioniert sich als ambitionierter Vertreter dieses Genres mit dem Ziel, die Komplexität von Tarkov mit einer zugänglicheren Spielerfahrung zu verbinden.

Die inhärente Komplexität von Extraction Shootern stellt Spieler jedoch vor erhebliche Herausforderungen: Multiple Quest-Lines mit unterschiedlichen Zielen und Abhängigkeiten, begrenzter Inventarplatz, knappe Ressourcen und die Notwendigkeit koordinierter Squad-Basierter Strategien erfordern ein hohes Maß an Planung und Informationsmanagement. Companion Apps haben sich in der Gaming-Industrie als effektive Lösung etabliert, um solche Komplexitäten zu bewältigen. Es bestehen zahlreiche Beispiele aus verschiedenen Genres wie League of Legends, Destiny 2 oder eben Extraction-Shootern wie Escape from Tarkov, welche demonstrieren, wie externe Anwendungen das Spielerlebnis durch Statistik-Tracking, Ressourcenmanagement und Team-Koordination signifikant verbessern können [29] [14].

## 1.2 Problemstellung

Spieler von Arc Raiders sehen sich mit mehreren miteinander verknüpften Herausforderungen konfrontiert:

**Informationsasymmetrie und mangelnde Übersicht:** Das Spiel bietet zahlreiche Quest-Lines mit unterschiedlichen Zielen, zeigt jedoch nur die aktiven Quests an. Spieler haben dadurch keinen vollständigen Überblick über verfügbare Quests, deren Abhängigkeiten, den optimalen Pfad zur Erfüllung ihrer Ziele oder benötigter Materialien für die Zukunft. Diese Informationsfragmentierung erschwert strategische Planung und führt zu ineffizienten Entscheidungen.

**Ressourcenmanagement:** Die Upgrade-Systeme für Workstations erfordern multiple Ressourcentypen über mehrere Stufen hinweg. Bei begrenztem Inventarplatz und knappen Ressourcen fehlen Spielern Werkzeuge zur Kalkulation benötigter Materialien und zur Priorisierung von Upgrades basierend auf ihren individuellen Spielzielen.

**Squad-Koordination:** Arc Raiders basiert auf Squad-orientiertem Gameplay, doch wenn jedes Squad-Mitglied nur seine eigenen Ziele verfolgt, entstehen Konflikte bei der Routenplanung und Ressourcenverteilung. Die fehlende zentrale Übersicht über Squad-Ziele behindert effektive Koordination und optimale Ressourcennutzung.

**Fehlen offizieller Planungstools:** Zum aktuellen Stand (November 2025) existieren keine offiziellen Tools oder Application Programming Interface (API) zur Lösung dieser Probleme. Daten werden von Spielern über diverse Plattformen gesammelt und bereitgestellt.

Diese Problemstellung ist nicht singulär für Arc Raiders, sondern repräsentativ für die Herausforderungen moderner Extraction Shooter mit komplexen Meta-Progression-Systemen. Die Lösung dieser Probleme durch eine dedizierte Companion App könnte somit über Arc Raiders hinaus als Referenzimplementierung für ähnliche Spiele dienen.

# 2 Aufgabenstellung und Entwicklungsmethodik

## 2.1 Operationalisierung der Problemstellung

Die in Abschnitt 1.2 identifizierten Herausforderungen für Spieler von Arc Raiders – Informationsasymmetrie bei Quest-Lines, ineffizientes Ressourcenmanagement und unzureichende Squad-Koordination – werden durch systematische Operationalisierung in konkrete Entwicklungsaufgaben überführt. Operationalisierung bezeichnet dabei die systematische Ableitung von Services und technischen Constraints aus übergeordneten Zielen [39].

Die identifizierten Probleme lassen sich wie folgt operationalisieren:

- **Problem "Informationsasymmetrie und mangelnde Übersicht"**
  → **Ziel** "Vollständiger Überblick über Quest-Lines und Abhängigkeiten"
  → **Service** "Quest Tracking mit Kanban/Flow-Chart-Visualisierung"
  → **Technische Anforderung** "Datenmodell für Quest-Abhängigkeiten, Filterung und Statusverwaltung"
- **Problem "Ressourcenmanagement bei begrenztem Inventar"**
  → **Ziel** "Optimierte Materialnutzung und Upgrade-Priorisierung"
  → **Service** "Material Calculator & Workstation Planner"
  → **Technische Anforderung** "Berechnung für Upgrade-Kosten über multiple Stufen"
- **Problem "Fehlende Squad-Koordination"**
  → **Ziel** "Zentrale Übersicht über Squad-Ziele und effiziente Routenplanung"
  → **Service** "Squad-basierte Routenoptimierung mit interaktiven Karten"
  → **Technische Anforderung** "Darstellung von Zielen und Karten mit Tools zur Routenoptimierung"

Diese Operationalisierung adressiert das in Abschnitt 1.2 beschriebene Fehlen offizieller Planungstools und nutzt die von der Community bereitgestellten Daten als Grundlage. Sie bildet die methodische Basis für die nachfolgend beschriebene Entwicklung einer Companion App, die als Referenzimplementierung für ähnliche Extraction Shooter dienen kann.

## 2.2 Entwicklungsmethodik und Phasenmodell

### 2.2.1 Phase 1: Requirements Engineering

In agilen Entwicklungsumgebungen ist Requirements Engineering integraler Bestandteil zur Sicherstellung, dass sich entwickelnde Bedürfnisse und Erwartungen der Stakeholder während des gesamten Entwicklungsprozesses erfasst werden [1].

#### 1.1 Anforderungserhebung (Requirements Elicitation)

- **1.1.1 Empirische Datenerhebung aus Spieltests:** Die in Abschnitt 1.1 erwähnte Teilnahme an einem Spieltest von Arc Raiders bildet die empirische Grundlage für die Anforderungserhebung. Durch systematische Beobachtung und Protokollierung werden konkrete Pain Points bei Quest-Management und Ressourcenplanung identifiziert.
- **1.1.2 Stakeholder-Interviews:** Durchführung direkter Gespräche mit Stakeholdern zur Extraktion von Bedürfnissen und Ideen [77]. Strukturierte Interviews mit Arc Raiders-Spielern verschiedener Erfahrungsstufen sowie gemeinsame Brainstorming-Sessions zur Feature-Ideenfindung ermöglichen die Erfassung spezifischer Anforderungen für Squad-basiertes Gameplay. Die in Abschnitt 1.2 identifizierten Problemstellungen werden dabei validiert und präzisiert.
- **1.1.3 Comparative Analysis:** Wie in Abschnitt 1.1 dargelegt, haben sich Companion Apps in der Gaming-Industrie als effektive Lösung etabliert. Die systematische Analyse umfasst primär etablierte Companion Apps für Extraction Shooter. Dies identifiziert Standard-Features und Innovationspotenziale, während die heuristische Evaluation von UI/UX-Patterns für komplexe Informationsdarstellung Best Practices aufzeigt.
- **1.1.4 Ableitung von User Stories:** Transformation der identifizierten Nutzerbedürfnisse in User Stories nach dem Format „Als [Rolle] möchte ich [Funktion], um [Nutzen] zu erreichen“. Beispiele bezogen auf die Problemstellung:
  - „Als Spieler möchte ich alle verfügbaren Quests und deren Abhängigkeiten sehen, um meine Ziele zu planen“,
  - „Als Squad-Leader möchte ich die Quest-Ziele meiner Teammitglieder auf einer Karte visualisieren, um eine effiziente Route für das gesamte Team zu planen“
  - „Als Spieler möchte ich kalkulieren, welche Materialien ich für geplante Workstation-Upgrades, Quests sowie Projekte benötige, um mein begrenztes Inventar optimal zu nutzen“.

### 1.2 Anforderungsanalyse (Requirements Analysis)

- **1.2.1 Kategorisierung und zeitliche Strukturierung:** Gruppierung in funktionale Anforderungen nach Seite (z.B. Dashboard, Quests, Workstations) und Implementierungsphase (z.B. P1-Visualization, P2-Progression) sowie nicht-funktionale Anforderungen (Performance, Usability, Verfügbarkeit). Identifikation von Abhängigkeiten zwischen Anforderungen (z.B. Routenplanung benötigt Maps) zur Planung der Implementierungsreihenfolge.
- **1.2.2 Priorisierung:** Implementierung der höchstpriorisierten Anforderungen zuerst zur Maximierung des Stakeholder-ROI [3]. Anwendung der MoSCoW-Methode (Must, Should, Could, Won't) mit Bewertung nach Business Value (Lösung der Kernprobleme aus Abschnitt 1.2) und technischer Komplexität. Dokumentation in Form eines Kanban Boards sowie Gantt-Diagramms unter Berücksichtigung von Abhängigkeiten zwischen Features.
- **1.2.3 Spezifikation von Akzeptanzkriterien:** Definition messbarer Erfolgskriterien für jede User Story nach SMART-Kriterien (Specific, Measurable, Achievable, Relevant, Time-bound). Beispiele umfassen:
  - „Quest-Suche liefert Ergebnisse in <500ms Time to Interactivity (TTI) für 95% der Anfragen“
  - „Material Calculator berechnet Upgrade-Kosten für beliebige Kombinationen von Workstations korrekt“.

### 1.3 Anforderungsvalidierung

- **1.3.1 Prototyping:** Erstellen von Minimum Viable Product (MVP)-Prototypen zur Visualisierung und Validierung der Anforderungen mit Stakeholdern. Interaktive Mockups und Wireframes ermöglichen frühes Feedback zu UI/UX-Designs und Funktionalitäten, um sicherzustellen, dass die entwickelten Lösungen den identifizierten Bedürfnissen entsprechen.
- **1.3.3 Abgleich mit Problemstellung:** Systematischer Abgleich der definierten Anforderungen mit der ursprünglichen Problemstellung zur Sicherstellung vollständiger Abdeckung aller drei Hauptherausforderungen: Informationsasymmetrie bei Quest-Lines, ineffizientes Ressourcenmanagement und unzureichende Squad-Koordination.

### 2.2.2 Phase 2: Architektur und Design

#### 2.1 Systemarchitektur

- **2.1.1 Technologie-Assessment:** Wie in Abschnitt 1.1 erwähnt, rechtfertigt das enorme Wachstumspotenzial des Gaming-Marktes die Wahl skalierbarer Technologien. Die systematische Evaluation umfasst Frontend-Frameworks (React, Vue, Angular), Backend-Technologien (Next.js API Routes, separate Backend-Lösung), Datenbank-Systeme (PostgreSQL, MongoDB, Firebase) sowie Hosting-Plattformen (Vercel, Netlify, AWS) unter Berücksichtigung von Kosteneffizienz für Hobby-Projekte. Multi-Criteria Decision Analysis (MCDA) ermöglicht die Technologieauswahl basierend auf Kriterien wie Performance, Entwicklerfreundlichkeit, Skalierbarkeit und Kosten.
- **2.1.2 Architekturentwurf:** Definition der Systemgrenzen und Komponenten (Frontend, Backend, Datenbank, externe Datenquellen) sowie Auswahl geeigneter Architekturmuster unter Berücksichtigung der Komplexität. Die Dokumentation erfolgt durch Architekturdiagramme nach dem C4-Modell (Context, Container, Component, Code), um verschiedene Abstraktionsebenen abzubilden und Stakeholdern unterschiedliche Detailgrade zu ermöglichen.
- **2.1.3 Risikoanalyse:** Identifikation technischer Risiken, insbesondere die in Abschnitt 1.2 erwähnte Abhängigkeit von Community-bereitgestellten Daten statt offizieller API. Bewertung nach Eintrittswahrscheinlichkeit und Impact sowie Definition von Mitigationsstrategien: Backup-Strategien für Datenquellen (Web-Scraping des offiziellen Fandoms gemäß robots.txt, User-Generated Content mit Qualitätssicherung), Caching-Mechanismen zur Reduzierung der Abhängigkeit von externen Quellen sowie Validierung und Qualitätssicherung von Community-Daten.

#### 2.2 Datenmodellierung

- **2.2.1 Konzeptionelle Modellierung:** Erstellung von Modellierungsdiagrammen wie Entity-Relationship (ER)-Digrammen zur Abbildung der Hauptentitäten (Quests, Workstations, Materialien, Spielerprofile) und deren Beziehungen oder Context Diagrams aus dem Domain-Driven Design (DDD) zur Identifikation von Bounded Contexts. Berücksichtigung der in Abschnitt 1.2 beschriebenen Anforderungen an Flexibilität und Erweiterbarkeit für zukünftige Features.
- **2.2.2 Datenquellen-Strategie:** Aufgrund des in Abschnitt 1.2 beschriebenen Fehlens offizieller Tools müssen mehrere Community-Datenquellen miteinander Verglichen und möglicherweise kombiniert werden. Hierbei wird ähnlich wie bei der Technologie-Assessment-Methode eine MCDA angewendet, um die Zuverlässigkeit, Aktualität und

Vollständigkeit der Datenquellen zu bewerten. Strategien zur Datenintegration und -synchronisation werden definiert, um eine konsistente und aktuelle Datenbasis sicherzustellen.

### 2.2.3 Phase 3: Implementierung in Iterationen

Agile Entwicklung betont die iterative Natur mit kontinuierlicher Verfeinerung und Validierung [20].

#### 3.1 Entwicklung

- **3.1.1 Iterative Entwicklung:** Umsetzung der priorisierten User Stories in Iterationen unter Berücksichtigung von Clean-Code Prinzipien. Jede Iteration umfasst Planung, Implementierung, Testing und Review. Kontinuierliche Integration von Feedback aus Reviews zur Anpassung des Backlogs und Verbesserung der Implementierung.
- **3.1.2 Continuous Integration:** Automatisierte Builds bei jedem Commit über GitHub Actions oder ähnliche CI-Tools sowie automatisierte Testausführung der gesamten Test-Suite. Code Quality Checks umfassen Linting (ESLint für JavaScript/TypeScript). Automatisches Deployment auf Staging-Umgebung (z.B. Vercel Preview Deployments) für frühzeitiges Testen.

#### 3.2 Review und Retrospektive

- **3.2.1 Review:** Demonstration implementierter Features an Stakeholder zur Einholung von Feedback bezüglich der Lösungen für die Probleme aus Abschnitt 1.2. Backlog-Refinement basierend auf gewonnenen Erkenntnissen sowie Anpassung der Priorisierung bei Bedarf.
- **3.2.2 Retrospektive:** Reflexion des Entwicklungsprozesses mit den Leitfragen: Was lief gut? Was kann verbessert werden? Identifikation von Verbesserungspotenzialen in Prozess, Definition konkreter Aufgaben für den nächsten Zyklus sowie Anpassung der Entwicklungspraktiken basierend auf Lessons Learned.

### 2.2.4 Phase 4: Evaluation und Reflexion

#### 4.1 Quantitative Evaluation

- **4.1.1 Performance-Metriken:** Messung von Page Load Time durch Metriken wie der TTI für alle Hauptseiten. Messung der API-Antwortzeiten für kritische Endpunk-

te sowie Überwachung der Server- und Datenbank-Performance (CPU-, Speicher- und Datenbank-Latenz) unter Lastbedingungen.

- **4.1.2 User-Engagement-Metriken:** Sofern Veröffentlichung erfolgt: Daily Active Users (DAU), Feature Usage Statistics zur Identifikation der am häufigsten genutzten Funktionalitäten sowie User Retention Rate zur Bewertung des langfristigen Nutzens.

### 4.2 Qualitative Evaluation

- **4.2.2 User Experience Interviews:** Durchführung von User Experience Interviews mit Spielern zur Bewertung, ob die in Abschnitt 1.2 identifizierten Probleme gelöst wurden. Bewertung der Zielerreichung bezüglich Verbesserung der Quest-Übersicht, Effizienzsteigerung im Ressourcenmanagement sowie Optimierung der Squad-Koordination.
- **4.2.3 Vergleich mit Anforderungen:** Systematischer Vergleich der implementierten Features mit initialen Anforderungen und Akzeptanzkriterien. Identifikation von vollständig erfüllten, teilweise erfüllten und nicht erfüllten Anforderungen mit Begründung der Abweichungen.

### 4.3 Kritische Reflexion

- **4.3.1 Technologie-Entscheidungen:** Diskussion des gewählten Technologie-Stacks (React/Next.js, Supabase/PostgreSQL, Vercel) mit Fokus auf Stärken, Schwächen und Lessons Learned. Reflexion der Datenhaltungsstrategie (Community-Daten, Caching, Synchronisation) und der in Abschnitt 1.2 beschriebenen Herausforderung einer fehlenden offiziellen API.
- **4.3.2 Architektur-Bewertung:** Bewertung der gewählten Architektur hinsichtlich Eignung für die Anforderungen, Entwicklungseffizienz, Skalierbarkeit und Wartbarkeit sowie Diskussion von Trade-offs und alternativen Ansätzen. Reflexion, ob die Architekturentscheidungen im Kontext des in Abschnitt 1.1 beschriebenen Marktwachstums zukunftsfähig sind.
- **4.3.3 Lessons Learned:** Dokumentation gewonnener Erkenntnisse aus dem Entwicklungsprozess: erfolgreiche Praktiken, aufgetretene Herausforderungen und deren Lösungen sowie Empfehlungen für zukünftige Projekte. Reflexion der agilen Methodik und deren Eignung für die Entwicklung einer Gaming Companion App.
- **4.3.4 Generalisierbarkeit:** Bewertung, inwieweit die entwickelte Lösung als Referenzimplementierung für andere Extraction Shooter dienen kann, wie in Abschnitt 1.2 postuliert. Identifikation von spielspezifischen Aspekten und übertragbaren Konzepten.

## 2.3 Methoden und Verfahren

"Welche Methoden und Verfahren werden verwendet?"

### 2.3.1 Verwendete Tools und Technologien

Verwendete Tools und Technologien mit Versionen/Datum (wie z.B. package.json aufbereiten), Ergebnis der Arbeit muss replizierbar sein

# 3 Grundlagen

## 3.1 Arc Raiders & Gaming Tools

### 3.1.1 Companion Applications im digitalen Ökosystem

Companion Applications haben sich als eigenständige Software-Kategorie etabliert, die primäre Anwendungen durch zusätzliche Funktionalität ergänzt, ohne das Hauptprodukt zu ersetzen. Im Gaming-Kontext erfüllen diese Anwendungen mehrere Schlüsselfunktionen: Sie erweitern die Spielerfahrung über die Session hinaus, bieten Planungs- und Analysewerkzeuge und ermöglichen Social Features außerhalb der Spielumgebung.

Companion Apps für Gaming-Konsolen verzeichneten seit 2020 ein signifikantes Wachstum, wobei die PC-Gaming-Plattform Steam mit etwa 33,4 Millionen Downloads im zuletzt gemessenen Quartal führend war [12]. Diese Entwicklung zeigt, dass Spieler bereit sind, zusätzliche Tools zu nutzen, um ihre Gaming-Erfahrung zu optimieren.

Die technische Architektur von Companion Apps variiert je nach Anwendungsfall:

- **Offizielle Plattform-Apps:** Direkte Integration mit Herstellersystemen (PlayStation App, Xbox App, Steam Mobile)
- **Game-spezifische Tools:** Fokus auf ein einzelnes Spiel oder Franchise
- **Community-getriebene Lösungen:** Von Spielern entwickelte Third-Party-Tools

### 3.1.2 Gaming Companion Apps: Funktionale Kategorisierung

Basierend auf der Analyse existierender Lösungen lassen sich Gaming Companion Apps in folgende funktionale Kategorien einteilen:

**Informations- und Datenbank-Tools:** Diese Tools bieten Zugriff auf Spieldaten wie Item-Statistiken, Charakterwerte oder Mechaniken. Apps wie Handbook for EFT bieten Spielern offline verfügbare Informationen zu Karten, Munitionsvergleichen, Waffen-Performance, Ausrüstung, Quest-Guides und Key-Informationen [43].

**Progress-Tracking-Systeme:** Apps wie The Hideout: Tarkov Sidekick ermöglichen Quest-Tracking, Hideout-Modul-Verwaltung und Team-Quest-Tracking, wo Spieler den Status ihrer Freunde sehen können [32].

**Markt- und Wirtschafts-Tools:** Funktionen wie Flea Market-Preisanzeige, Preishistorien bis zu einem Jahr zurück und Preis-Alerts für Flea Market-Preise helfen Spielern bei wirtschaftlichen Entscheidungen [32].

**Karten- und Navigationshilfen:** Interactive Maps-Apps bieten detaillierte, community-erstellte Karten mit Loot-Spots, Extraktionspunkten, Key-Locations und Quest-Details [44].

**Build-Planer und Kalkulatoren:** Weapon Builder und Damage Calculator („Tarkov'd Simulator") ermöglichen theoretisches Durchspielen von Szenarien vor der Implementierung im Spiel [32].

### 3.1.3 Extraction Shooter: Genre-spezifische Anforderungen

Extraction Shooter basieren auf Konzepten der Verlustaversion und verzögerten Gratifikation, wobei jeder Raid ein Risiko darstellt und der erfolgreiche Abschluss eines wertvollen Durchgangs intensive dopamingesteuerte Befriedigung freisetzt [35]. Diese psychologischen Mechanismen erzeugen spezifische Anforderungen an unterstützende Tools.

**Risiko-Management:** Die Permadeath-Mechanik von Extraction Shootern erfordert sorgfältige Planung. Companion Apps können helfen, Risiken zu minimieren durch:

- Vorausschauende Ressourcenplanung
- Optimale Route-Planung zur Minimierung von Begegnungen
- Wert-Kalkulation von Loot vs. Risiko

**Komplexitätsreduktion:** Das Extraction-Shooter-Genre gilt aufgrund seiner Komplexität als schwer zugänglich für den Massenmarkt [34]. Arc Raiders hat durch seine Betonung von Teamsynergien und intuitiveren Spielerführungssystemen einen zugänglicheren Einstiegspunkt geschaffen [35].

Der aktuelle Markt konzentriert sich auf Escape from Tarkov (ca. 60.000 gleichzeitige Nutzer), Hunt Showdown (ca. 20.000) und Dark and Darker (ca. 10.000), zusammen etwa 100.000 gleichzeitige Nutzer [34]. Die Herausforderung für neue Titel liegt in der Etablierung eigener Tool-Ökosysteme.

### 3.1.4 Arc Raiders: Technische Charakteristika und Systeme

ARC Raiders ist ein 2025 veröffentlichter Third-Person-Extraction-Shooter, entwickelt mit der Unreal Engine 5 für PlayStation 5, Windows und Xbox Series X/S [82]. Bis zum 11. November 2025 hatte das Spiel weltweit über vier Millionen Exemplare verkauft [82], was eine signifikante Nutzerbasis für Companion-Tools darstellt.

Das Spiel implementiert mehrere Systeme, die durch externe Tools unterstützt werden können:

**Quest-System:** Spieler erfüllen Quests für Händler mit unterschiedlichen Motiven und Agenden, was sich in komplexen Quest-Chains mit Abhängigkeiten manifestiert [21]. Das Ingame-Interface zeigt nur aktive Quests, was einen Gesamtüberblick erschwert.

**Crafting und Ressourcen:** Spieler müssen Workshop-Stationen upgraden und Blueprints lernen, um fortgeschrittenere Items zu craften [21]. Bei begrenztem Inventarplatz ist strategisches Ressourcen-Management essentiell.

**Skill-Progression:** Der ARC Raiders Skill-Tree verzweigt sich in drei Pfade: Survival, Mobility und Conditioning [21], was Langzeitplanung erfordert.

**Multiplayer-Koordination:** Das Spiel unterstützt nahtloses Cross-Platform-Spiel zwischen PlayStation, Xbox und PC, wobei Spieler in Squads bis zu drei Personen oder solo spielen können [21].

### 3.1.5 Referenzimplementierungen: Tarkov Companion Ecosystem

Das Escape from Tarkov Ökosystem bietet wertvolle Erkenntnisse für die Entwicklung von Companion Apps.

Tarkov Companion als Overwolf-App bietet Quest-Management sortiert nach Händler, Location oder Status, Browse-Funktionen für Karten mit Extraktionen, Loot-Hotspots und Quest-Items sowie Key-Suche und Hideout-Upgrade-Tracking [67]. Diese Features repräsentieren den aktuellen Standard für Extraction-Shooter-Companion-Apps.

Tarkov.dev bietet ein freies, community-erstelltes und Open-Source-Ökosystem von Escape from Tarkov-Tools inklusive Informationen zu Items, Crafts, Barters, Maps, Loot-Tiers, Hideout-Profiten und einer freien API [65]. Die Verfügbarkeit einer API ermöglicht die Entwicklung vielfältiger Third-Party-Tools.

## 3.2 Moderne Web-Technologien

Die Wahl geeigneter Web-Technologien ist entscheidend für den langfristigen Erfolg komplexer Webanwendungen. Für die Entwicklung einer Arc Raiders Companion App kommen moderne, produktionsreife Technologien zum Einsatz, die sowohl Entwicklerproduktivität als auch Anwendungsperformance optimieren.

### 3.2.1 TypeScript

TypeScript hat sich als De-facto-Standard für moderne JavaScript-Entwicklung etabliert. Als Superset von JavaScript fügt TypeScript statische Typisierung und erweiterte Sprachfeatures hinzu, die besonders für große Projekte wertvoll sind [36].

**Type Safety in großen Projekten:** Type Safety ist eines der herausragenden Merkmale von TypeScript und adressiert eine kritische Limitierung des dynamischen Typsystems von JavaScript [36]. In größeren Anwendungen, wo mehrere Entwickler am selben Codebase arbeiten, können unterschiedliche Annahmen über Datentypen zu unerwartetem Verhalten und schwer auffindbaren Bugs führen [36]. TypeScript ermöglicht es Entwicklern, Typen explizit für Variablen, Funktionsparameter und Rückgabewerte zu definieren, wodurch potenzielle Fehler zur Compile-Zeit statt zur Laufzeit erkannt werden [36].

Organisationen übernehmen TypeScript zunehmend für Large-Scale-Anwendungen aufgrund seiner Fähigkeit, Fehler zur Compile-Zeit zu erkennen, wodurch Laufzeitfehler reduziert und die Code-Qualität verbessert wird [23]. Der strukturierte Ansatz von TypeScript hilft Teams, komplexe Codebases effizienter zu verwalten [23]. In größeren Teams verbessert Type Safety die Zusammenarbeit durch Reduzierung der kognitiven Last und Förderung klarerer Kommunikation [36].

**Developer Experience:** TypeScript verbessert nicht nur die Code-Qualität, sondern steigert auch signifikant die Developer Experience durch überlegenes Tooling und Error Reporting [36]. Die Integration mit modernen IDEs bietet erweiterte Features wie intelligente Code-Vervollständigung, automatisches Refactoring und Code-Navigation [16].

Tooling und IDE-Unterstützung für TypeScript erfuhren 2024 signifikante Verbesserungen, wobei Entwickler von besserem IntelliSense, Auto-Completion und Refactoring-Tools profitieren, was den Entwicklungsprozess reibungsloser und effizienter gestaltet [66]. TypeScript wird zunehmend mit modernen Frameworks wie React, Angular und Vue.js integriert, was Entwicklern ermöglicht, die Vorteile von TypeScripts Type-Checking zu nutzen und gleichzeitig die leistungsstarken Features dieser Frameworks für den Aufbau von Benutzeroberflächen zu verwenden [23].

Durch die Nutzung des statischen Typsystems von TypeScript können Entwickler sichereren und wartbareren Code schreiben und die Gesamtqualität und Zuverlässigkeit ihrer Anwendungen verbessern [16]. Mit dem Fokus auf statische Typisierung und Developer-Ergonomie ist TypeScript gut positioniert, um den sich entwickelnden Anforderungen der Webentwicklung auch zukünftig gerecht zu werden [9].

### 3.2.2 React

React hat sich als führende JavaScript-Bibliothek für den Aufbau von Benutzeroberflächen etabliert, primär für Single Page Application (SPA) [28]. Eines der wichtigsten Features von React ist seine komponentenbasierte Architektur, die Entwicklern ermöglicht, skalierbare und wartbare Anwendungen effizient zu erstellen.

**Komponentenbasiertes User Interface (UI):** In React ist eine Komponente eine wiederverwendbare, eigenständige Einheit einer Benutzeroberfläche [28]. Komponenten ermöglichen es, eine Anwendung in kleinere, unabhängige Teile zu zerlegen, die effizient verwaltet und wiederverwendet werden können [28]. Diese modulare Struktur macht die Anwendung einfacher zu entwickeln, zu warten und zu skalieren, da Komponenten über verschiedene Teile der App oder sogar in unterschiedlichen Projekten wiederverwendet werden können [28].

Jede React-Anwendung besteht aus einem Baum von Komponenten, wobei jede Komponente ihre eigene Logik, ihren State und ihre UI-Repräsentation hat [28]. Die Vorteile dieser Architektur umfassen:

- **Wiederverwendbarkeit:** Komponenten können mehrfach in verschiedenen Teilen einer Anwendung verwendet werden
- **Modularität:** Jede Komponente handhabt ein spezifisches Stück Funktionalität, was die Anwendung strukturierter macht
- **Skalierbarkeit:** Große Anwendungen können durch Zusammensetzen kleinerer, wiederverwendbarer Komponenten entwickelt werden
- **Wartbarkeit:** Updates und Bugfixes sind einfacher, da Änderungen auf spezifische Komponenten lokalisiert sind [28]

**Komposition over Inheritance:** React verfügt über ein leistungsstarkes Kompositionsmodell, und die Verwendung von Komposition anstelle von Vererbung wird empfohlen, um Code zwischen Komponenten wiederzuverwenden [24]. Bei Facebook verwenden die Entwickler React in tausenden von Komponenten, und es wurden keine Anwendungsfälle gefunden, bei denen die Erstellung von Komponenten-Vererbungshierarchien empfohlen würde [24]. Props und Komposition bieten die gesamte Flexibilität, die benötigt wird, um das Aussehen und Verhalten einer Komponente auf explizite und sichere Weise anzupassen [24].

Komposition ist eine Technik, bei der eine Komponente durch Zusammensetzen anderer Komponenten aufgebaut wird, ähnlich wie man ein Lied aus verschiedenen musikalischen Noten komponieren würde [56]. React fördert die Verwendung von Komposition anstelle von Vererbung aus mehreren Gründen:

- **Flexibilität:** Komposition gibt mehr Flexibilität bei der gemeinsamen Nutzung von Funktionalität zwischen Komponenten [56]

- **Einfachheit:** Sie fördert einfachere Hierarchien mit Komponenten, die isoliert verstanden werden können, ohne eine Vererbungskette kennen zu müssen [56]
- **Wiederverwendbarkeit:** Für Komposition entworfene Komponenten sind oft einfacher wiederzuverwenden, da sie keine Annahmen über den Kontext treffen, in dem sie verwendet werden [56]
- **Vermeidung von Tight Coupling:** Vererbung kann zu enger Kopplung zwischen Komponenten führen, was die Codebasis fragil und schwer zu refaktorisieren macht [56]

React fördert Komposition gegenüber Vererbung, weil sie größere Flexibilität und Trennung von Belangen ermöglicht [41]. Vererbung kann manchmal zu enger Kopplung zwischen Komponenten führen, was es schwieriger macht, Anwendungen zu modifizieren oder zu skalieren [41]. Durch die Übernahme von Komposition anstelle von Vererbung können Entwickler die Wiederverwendbarkeit von Komponenten vereinfachen und ihren Code flexibler und wartbarer gestalten [41].

### 3.2.3 Full Stack React Frameworks

Während React als UI-Bibliothek exzelliert, benötigen produktionsreife Anwendungen zusätzliche Funktionalitäten wie Routing, Server-Side Rendering und Daten-Fetching. Full-Stack React Frameworks wie Next.js adressieren diese Anforderungen durch Bereitstellung einer kompletten Lösung für moderne Webanwendungen.

**Next.js als React-Backend-Framework:** Next.js hat sich zu einem Kraftpaket für den Aufbau performanter und Search Engine Optimization (SEO)-freundlicher React-Anwendungen entwickelt [54]. Der App Router ist ein dateibasierter Router, der Reacts neueste Features wie Server Components, Suspense und Server Functions nutzt [70]. Next.js verwendet ein dateisystembasiertes Routing, bei dem Ordner zur Definition von Routen verwendet werden [75]. Jeder Ordner repräsentiert ein Routen-Segment, das einem URL-Segment zugeordnet wird [75].

**File-Based Routing:** Mit dem App Router ermutigt Next.js zu einem dateibasierten Routing-System, bei dem die Verzeichnisstruktur die URL-Struktur widerspiegelt [46]. In Next.js nutzt der App Router das dateibasierte Routing, was bedeutet, dass die Position der `page.tsx`- oder `route.ts`-Datei innerhalb des `app`-Verzeichnisses definiert, wie sie auf eine gegebene URL abgebildet wird [49].

Next.js folgt weiterhin dem dateibasierten Routing, aber mit der Einführung des App Routers haben Dateien und Ordner nun strikt definierte Rollen [4]:

- **Ordner:** Ordner definieren die Routen der Anwendung. Ein Routen-Segment ist ein Pfad vom Root-Ordner bis zu einem Blatt-Ordner, der eine `page.ts`-Datei enthält [4]
- **Dateien:** Dateien erstellen die UI für ein Routen-Segment [4]

Eine spezielle `page.ts`-Datei wird verwendet, um Routen-Segmente öffentlich zugänglich zu machen [75]. Dynamic Route Segments können durch Umschließen des Ordnernamens mit eckigen Klammern erstellt werden, wie `[productId]` [4].

**Rendering-Strategien:** Next.js ist zu einem führenden Framework geworden, indem es verschiedene Rendering-Strategien anbietet: Static Site Generation (SSG), Server-Side Rendering (SSR), Client-Side Rendering (CSR), Incremental Static Regeneration (ISR) und das experimentelle Partial Prerendering (PPR) [69]. Diese wurden entwickelt, um Performance, SEO und User Experience in verschiedenen Situationen zu optimieren [69].

**SSG:** Bei SSG wird die initiale Hypertext Markup Language (HTML) zur Build-Zeit generiert, was zu schnellen Ladezeiten führt [54]. SSG ist ideal für Inhalte, die sich nicht häufig ändern, wie Blog-Posts, Dokumentation oder Marketing-Seiten [54]. Im App Router-Modell rendert SSG automatisch jede Komponente statisch, die keine server-spezifischen Funktionen nutzt [68].

**SSR:** SSR generiert HTML auf dem Server für jede Anfrage [54]. Diese Strategie ist optimal für echtzeitbezogene, nutzerspezifische Inhalte wie personalisierte Dashboards, Account-Profile oder News-Feeds [68]. SSR garantiert, dass jeder Besucher bei jedem Laden der Seite die neuesten Daten sieht [68].

**ISR:** ISR baut auf SSG auf und fügt die Fähigkeit hinzu, Inhalte zu aktualisieren, ohne einen vollständigen Rebuild der Site zu erfordern [54]. Bei ISR wird weiterhin die initiale HTML zur Build-Zeit generiert, aber Next.js wird auch mitgeteilt, wie oft nach Updates geprüft und die HTML-Dateien revalidiert werden sollen [54]. ISR ist geeignet für Inhalte, die sich periodisch ändern, wie Blog-Posts oder Produkt-Listings [68].

**PPR:** PPR ist eine der neuesten Ergänzungen zu Next.js und befindet sich derzeit im experimentellen Status [68]. PPR ermöglicht es, dass eine Seite teilweise mit statischen und dynamischen Segmenten kombiniert pre-gerendert wird [68]. Dies ist besonders nützlich für Seiten mit Sektionen, die progressiv laden können, während kritischer Content sofort erscheint [68].

Partial Prerendering kombiniert ultra-schnelle statische Edge-Delivery mit vollständig dynamischen Fähigkeiten und hat das Potenzial, das Standard-Rendering-Modell für Webanwendungen zu werden [73]. PPR bietet ein vereinheitlichtes Modell, das die Zuverlässigkeit und Geschwindigkeit von ISR mit den dynamischen Fähigkeiten von SSR verbindet [73].

PPR basiert auf React Suspense Boundaries: zur Build-Zeit wird der gesamte Content bis zur Suspense-Boundary zusammen mit den Fallbacks statisch generiert und suspendiert das Rendering zur Build-Zeit [53]. Zur Request-Zeit wird die pre-gerenderte statische Shell sofort an den Client geliefert, während Next.js das Rendering dort fortsetzt, wo es zur Build-Zeit suspendiert wurde [53]. Sobald die suspendierten Children auflösen, wird die UI zum Client in derselben Response gestreamt [53].

Bei der Entscheidung für eine Rendering-Strategie sollten folgende Faktoren berücksichtigt werden: Wie oft ändert sich dieser Content? SSG ist gut für statische Inhalte, ISR ist hervorragend für periodisch wechselnde Inhalte, und SSR oder CSR ist am besten für Echtzeit-Daten [69]. Next.js ermöglicht Entwicklern, verschiedene Rendering-Methoden innerhalb einer einzelnen Anwendung zu nutzen, je nach Bedarf, auf Seiten-Basis [69].

## 3.3 UI/UX Frameworks & Design System

Die systematische Entwicklung von Benutzeroberflächen erfordert strukturierte Ansätze zur Gewährleistung von Konsistenz, Wartbarkeit und Skalierbarkeit. Moderne Web-Anwendungen stehen vor der Herausforderung, Interfaces für eine Vielzahl von Geräten, Bildschirmgrößen und Nutzungskontexten bereitzustellen. Design Systems und UI-Frameworks bieten methodische Lösungen für diese Komplexität, wobei sich in den letzten Jahren fundamentale Paradigmenwechsel vollzogen haben.

### 3.3.1 Grundlagen modularer Design-Systeme

Die theoretische Grundlage moderner Design-Systeme bildet Brad Frosts Atomic Design Methodology, die 2013 erstmals als Blogpost vorgestellt und 2016 in Buchform veröffentlicht wurde [27]. Inspiriert von chemischen Konzepten, postuliert Atomic Design, dass komplexe UI systematisch in kleinere, wiederverwendbare Komponenten zerlegt werden können.

**Hierarchische Komponentenstruktur:** Die Methodologie definiert fünf hierarchische Ebenen [27]:

- **Atoms:** Fundamentale HTML-Elemente wie Buttons, Input-Felder oder Labels, die nicht weiter zerlegt werden können ohne ihre Funktionalität zu verlieren.
- **Molecules:** Gruppen von Atoms, die zu funktionalen Einheiten kombiniert werden. Beispiel: Label, Input und Button bilden ein Search-Form-Molecule.
- **Organisms:** Komplexere Komponenten aus Molecules und Atoms, die eigenständige Interface-Sektionen bilden wie Navigation, Header oder Formulare.
- **Templates:** Layout-Strukturen, die Organisms arrangieren und die Content-Struktur ohne spezifischen Inhalt definieren.
- **Pages:** Konkrete Template-Instanzen mit realem Content, die für Usability-Testing und Design-Validation dienen.

**Vorteile modularer Komponentenarchitektur:** Der atomare Ansatz bietet mehrere fundamentale Vorteile [27]:

- **Konsistenz:** Wiederverwendbare Komponenten garantieren visuell und funktional einheitliche Interfaces.
- **Skalierbarkeit:** Die modulare Struktur vereinfacht langfristige Wartung und ermöglicht reibungslose Erweiterungen.
- **Effizienz:** Designer und Entwickler können Komponenten schnell kombinieren statt jedes Element neu zu erstellen.
- **Wartbarkeit:** Änderungen an einem Atom propagieren automatisch durch alle abhängigen Molecules und Organisms.
- **Shared Vocabulary:** Die hierarchische Nomenklatur schafft eine gemeinsame Sprache zwischen Design und Development.

**Traversierung zwischen Abstraktion und Konkretion:** Ein zentraler Vorteil von Atomic Design liegt in der Fähigkeit, simultan zwischen abstrakten Elementen und konkreten Interfaces zu wechseln [27]. Designer können sowohl isolierte Atoms betrachten als auch deren Komposition in finalen Pages analysieren, was iterative Verfeinerung auf allen Hierarchieebenen ermöglicht.

### 3.3.2 Utility-First CSS: Paradigmenwechsel im Styling

Traditionelle Cascading Style Sheets (CSS)-Methodologien wie Block Element Modifier (BEM) und Object Oriented CSS (OOCSS) organisierten Styles primär um semantische Komponenten-Klassen. Der Utility-First-Ansatz vollzieht einen fundamentalen Paradigmenwechsel, indem Styles als komposierbare Utility-Klassen bereitgestellt werden, die direkte CSS-Properties repräsentieren [62].

**Tailwind CSS als Referenzimplementierung:** Tailwind CSS etablierte sich als führendes Utility-First Framework [83]. Im Gegensatz zu traditionellen Frameworks wie Bootstrap oder Foundation liefert Tailwind keine vorgefertigten Komponenten, sondern ein umfassendes Set von Utility-Klassen für Layout, Spacing, Typography, Colors und weitere CSS-Properties [62].

Die Philosophie manifestiert sich in einem einfachen Beispiel:

```
<!-- Traditionelles \ac{CSS} mit semantischen Klassen -->
<div class="message-warning">
  <p class="message-text">Warnung!</p>
</div>

<!-- Utility-First mit Tailwind -->
<div class="bg-yellow-300 font-bold p-4 rounded">
  <p class="text-gray-900">Warnung!</p>
</div>
```

**Constraints-basiertes Design:** Ein fundamentaler Vorteil von Utility-First liegt im constraints-basierten Design-Ansatz [62]. Statt arbitrary „Magic Numbers“ in Custom CSS zu verwenden, wählen Entwickler aus einem vordefinierten Design System mit konsistenten Spacing-Skalen, Farbpaletten und Typographie-Hierarchien. Dies fördert visuell konsistente Interfaces und erleichtert Design Token Management.

**Technische Architektur und Evolution:** Tailwind CSS durchlief signifikante architektonische Entwicklungen [83]:

- **Version 1-2:** Vollständige Stylesheet-Generierung mit nachgelagertem PurgeCSS Tree-Shaking. Nachteil: Lange Build-Zeiten und massive CSS-Dateien vor Purging.
- **Version 3+ (Just-In-Time (JIT)-Mode):** JIT Compiler analysiert HTML/JSX/Vue-Files und generiert nur tatsächlich verwendete Utility-Klassen. Resultat: Drastisch reduzierte Build-Zeiten und Bundle-Größen.
- **Version 4 (aktuell):** Native CSS Layer System (`@layer base, components, utilities`) eliminiert Specificity-Konflikte und ermöglicht klare Style-Hierarchien.

**Performance-Charakteristika:** Production Builds mit Tailwind CSS generieren typischerweise CSS-Bundles unter 10KB durch aggressive Unused-CSS-Elimination [62]. Diese Bundle-Größe ist signifikant kleiner als vollständige CSS-Frameworks wie Bootstrap, was zu verbesserten First Contentful Paint (FCP) Metriken führt.

**State Management und Responsiveness:** Tailwind bietet integrierte Variant-Systeme für States und Responsive Design [62]:

```
<button class="bg-blue-500 hover:bg-blue-700
               md:w-auto sm:w-full
               dark:bg-blue-900">
  Responsive Button
</button>
```

Diese Variants ermöglichen State-Handling (hover, focus, active) und Media Queries direkt im Markup, ohne separate CSS-Dateien.

**Kritische Betrachtung:** Trotz der Vorteile existieren valide Kritikpunkte am Utility-First-Ansatz: Die Utility-First-Denkweise weicht fundamental von traditionellem CSS ab und erfordert Einarbeitungszeit. HTML-Elemente können mit vielen Utility-Klassen überladen werden, was die Lesbarkeit beeinträchtigt. Die Developer Experience wird jedoch durch moderne Tooling wie VSCode-Extensions mit vollständiger IntelliSense signifikant verbessert.

### 3.3.3 Evolution der Komponenten-Bibliotheken

Die Landschaft der UI-Komponenten-Bibliotheken durchlief mehrere Evolutionsstufen, die unterschiedliche Trade-offs zwischen Convenience und Control reflektieren.

#### Generation 1: Opinionated Component Libraries

Traditionelle UI-Frameworks wie Bootstrap und Material-UI (MUI) liefern vollständige Design-Systeme mit vorgefertigten, gestylten Komponenten [42]. Diese Libraries folgen etablierten Design-Prinzipien – Bootstrap mit eigener Design-Language, Material-UI mit Googles Material Design Guidelines [37].

**Strukturelle Vorteile:**

- **Rapid Prototyping:** Out-of-the-box Komponenten ermöglichen schnelle UI-Entwicklung ohne Style-Implementation.
- **Design-Konsistenz:** Kohärente visuelle Sprache über alle Komponenten garantiert.
- **Große Communities:** Etablierte Frameworks bieten umfangreiche Dokumentation, Beispiele und Community-Support.
- **Accessibility Built-In:** Accessibility-Features wie Accessible Rich Internet Applications (ARIA)-Attributes und Keyboard-Navigation standardmäßig implementiert.

**Strukturelle Limitierungen:** Die fundamentalen Nachteile opinionated Libraries manifestieren sich in mehreren Dimensionen [42]:

- **Design-Uniformität:** Viele Anwendungen teilen identische „Bootstrap-Ästhetik“ oder „Material Design Look“, was Brand-Differenzierung erschwert.
- **Override-Komplexität:** Abweichungen vom Default-Design erfordern tiefgreifende CSS-Override-Kaskaden und Theme-System-Manipulation. Bei MUI bedeutet dies häufig Kämpfe gegen Emotion's CSS-in-JS Engine.
- **Bundle-Größe:** Comprehensive Libraries wie MUI liefern signifikante JavaScript-Bundles inklusive Runtime-Styling-Engines, was FCP und TTI Metriken beeinträchtigt.
- **Vendor Lock-In:** Migration zu alternativen Design-Systemen erfordert Rewrite großer Codebases.
- **Specificity Wars:** Tief verschachtelte CSS-Selektoren erschweren selektive Style-Overrides.

### Generation 2: Headless UI Components

Als Reaktion auf die Limitierungen opinionated Libraries entstand das Headless UI Pattern: Komponenten liefern Funktionalität, Accessibility und State Management, aber keine visuellen Styles [37].

**Separation of Concerns:** Headless Components implementieren strikte Trennung zwischen [40]:

- **Behavior Layer:** State Management, Event Handling, Keyboard Navigation
- **Accessibility Layer:** ARIA Attributes, Screen Reader Support, Focus Management
- **Presentation Layer:** Vollständig Developer-kontrolliert, keine Default-Styles

**Prominente Implementierungen:** Radix UI bietet unstyled Primitives für komplexe Patterns wie Dialogs, Dropdowns, Popovers und Tooltips mit strengem Fokus auf Web Accessibility Initiative (WAI)-ARIA Compliance [84]. Headless UI von Tailwind Labs ist optimiert für Integration mit Tailwind CSS. React Aria von Adobe fokussiert auf Internationalization und Platform-Agnostic Patterns [40].

**Vorteile des Headless-Patterns:**

- **Design-Freiheit:** Keine Style-Overrides nötig, vollständige visuelle Kontrolle.
- **Framework-Agnostic:** Primitives integrieren mit beliebigen Styling-Solutions (Tailwind, Emotion, Vanilla CSS).
- **Minimale Bundle-Größe:** Keine CSS-Bundles oder Styling-Runtime.
- **Accessibility Garantiert:** Professionell implementierte ARIA-Patterns ohne Developer-Overhead.

**Trade-offs:** Jede Component erfordert vollständiges Custom-Styling und kleinere Communities bedeuten weniger Community-Resources als etablierte UI-Libraries [37].

MUI erkannte diese Entwicklung und führte Base UI als eigene headless Component-Suite ein, die die Accessibility-Features von Material-UI ohne visuelle Opinions bereitstellt [42].

### Generation 3: Hybrid-Ansatz Shadcn/ui

Shadcn/ui synthetisiert Headless Primitives mit production-ready Styling und etabliert ein innovatives Distribution-Modell [45].

**Code Ownership Model:** Im Gegensatz zu traditionellen Node Package Manager (NPM)-Dependencies kopiert Shadcn/ui Component-Code direkt ins Projekt via Command Line Interface (CLI) [45]:

```
npx shadcn-ui@latest init
npx shadcn-ui@latest add button dialog form
```

Resultat: Components existieren als modifizierbare Source-Files im Projekt, keine Black-Box-Dependency.

**Technische Foundation:**

- **Radix UI Primitives:** Accessibility und Behavior Layer [84]
- **Tailwind CSS:** Utility-First Styling
- **Class Variance Authority (CVA):** Type-safe Variant Management
- **TypeScript:** Vollständige Type-Safety

**Vorteile des Copy-Paste-Modells:**

- **100% Kontrolle:** Code im Projekt modifizierbar ohne Library-Constraints
- **Kein Vendor Lock-In:** Keine Runtime-Dependency auf Shadcn Package
- **Versionskontrolle:** Components unter eigener Git-History
- **Selective Adoption:** Nur benötigte Components im Projekt

**Trade-offs:** Updates müssen manuell integriert werden und das Modell erfordert Tailwind CSS Expertise sowie aufwändigere initiale Konfiguration als NPM-Install [45].

### 3.3.4 Graph-Visualisierung und automatische Layouts

Die Visualisierung komplexer relationaler Datenstrukturen wie Quest-Dependencies oder Ressourcen-Hierarchien erfordert spezialisierte Bibliotheken für interaktives Graph-Rendering.

**React Flow als State-of-the-Art:** React Flow (seit 2024 rebrandend als XyFlow) etablierte sich als führende deklarative Bibliothek für node-basierte Editoren und interaktive Diagramme [85]. Die Library bietet:

- **Deklarative API:** Nodes und Edges als React-Komponenten mit Props-basierter Konfiguration
- **Built-In Interaktivität:** Drag-and-Drop, Zoom, Pan ohne Custom-Implementation
- **Custom Node Types:** Vollständig anpassbare Node-Komponenten (z.B. Quest-Cards mit Thumbnails)
- **Performance-Optimierung:** Virtualisierung für Graphen mit tausenden Nodes

**Layout-Algorithmen:** Manuelle Node-Positionierung ist für komplexe Graphen impraktikabel. Automatische Layout-Algorithmen berechnen optimale Node-Platzierung basierend auf Graph-Struktur [85].

*Dagre – Hierarchical Directed Graphs:* Der Dagre-Algorithmus optimiert für Directed Acyclic Graph (DAG) mit klarer Hierarchie [85]:

Dagre minimiert Edge-Crossings und optimiert Layer-Spacing. [85].

**Limitierungen:** Dagre setzt uniform Node-Dimensionen voraus. Custom Node-Sizes erfordern Post-Layout Adjustments. Dynamisches Re-Layout bei Graph-Änderungen muss manuell getriggert werden [85].

*Alternativen für komplexere Anforderungen:* ELK.js bietet extensive Konfiguration für diverse Graph-Typen (layered, force-directed, radial), wobei die Komplexität jedoch Support erschwert. D3-Force implementiert Physik-basierte Layouts für nicht-hierarchische Graphen mit Spring-Force-Simulation für organische Node-Distribution. D3-Hierarchy ist optimiert für Tree-Strukturen mit single Root Node [85].

**Semantic-Specific Custom Algorithms:** Generische Layout-Algorithmen können semantische Bedeutung von Graphen nicht berücksichtigen [80]. Für domain-spezifische Visualisierungen (z.B. Business Process Flows, Quest-Chains) können Custom-Algorithmen überlegen sein. Beispiel-Anforderungen: Sequentielle Quest-Chains sollten linear visualisiert werden, Parallel-Quests gruppieren sich visuell, Critical-Path-Highlighting erfordert spezifische Node-Platzierung. Die Entwicklung von Custom-Algorithmen folgt iterativen Refinement-Prozessen mit kontinuierlichem Visual-Feedback [80].

## 3.4 State Management & Data Fetching

Die systematische Verwaltung von Anwendungszustand (State) stellt eine fundamentale Herausforderung in der Entwicklung moderner Webanwendungen dar. Die Evolution von State Management Lösungen im React-Ökosystem reflektiert eine kontinuierliche Auseinandersetzung mit den Trade-offs zwischen Komplexität, Developer Experience und Skalierbarkeit. Für die Arc Raiders Companion App ergibt sich die Notwendigkeit einer hybriden State Management Architektur, die sowohl lokalen UI-State als auch Server-synchronisierte Daten effizient verwaltet.

### 3.4.1 Evolution der State Management Paradigmen

Die historische Entwicklung von State Management Libraries in React verdeutlicht einen Paradigmenwechsel von monolithischen, opinionated Lösungen hin zu spezialisierten, komposi-

tionellen Architekturen.

**Die Flux-Architektur als konzeptuelle Grundlage:** Facebooks Einführung der Flux-Architektur um 2014 etablierte das Konzept unidirektionalen Datenflusses als Antwort auf die Probleme bidirektionaler Data Bindings [26]. Die zentrale Innovation bestand in der Separation von State-Updates durch ein strukturiertes Muster: Actions beschreiben *was* passiert ist, während Reducer definieren *wie* der State sich ändert [50]. Diese konzeptuelle Trennung adressierte die Unvorhersagbarkeit von Backbone-artigen Model-View Architekturen, bei denen Template-Updates zu unkontrollierbaren Kaskaden von State-Mutationen führen konnten [50].

**Redux als De-facto-Standard (2015–2019):** Dan Abramovs Redux, 2015 veröffentlicht, synthetisierte Flux-Prinzipien mit funktionaler Programmierung und wurde durch die Demonstration von Time-Travel Debugging bekannt [50]. Bis 2016 etablierte sich die Konvention „If you're using React, you must be using Redux too" [50], was jedoch zu übermäßiger Adoption selbst in Kontexten führte, wo Redux nicht erforderlich war. Die zentrale Stärke von Redux manifestierte sich in der Vorhersagbarkeit: Ein zentraler Store, reine Reducer-Funktionen und immutable Updates ermöglichten deterministisches State Management [50].

**Das Boilerplate-Problem:** Die fundamentale Kritik an Redux fokussierte sich auf den exzessiven Boilerplate-Code [26]. Die Implementation einer einzelnen State-Property erforderte die Erstellung von Action Types, Action Creators, Reducer Cases und Selectors. Für asynchrone Operationen wie API-Requests mussten zusätzlich Middleware-Patterns (Redux Thunk, Redux Saga) mit separaten Actions für Loading-, Success- und Error-States implementiert werden [50]. Diese Code-Proliferation führte zu signifikanter Entwicklungslatenz und erschwerte die Wartbarkeit, insbesondere in schnell iterierenden Projekten.

**Die Context API Revolution (2019):** Die Einführung der modernisierten React Context API mit korrekter Value-Propagation und die Verfügbarkeit von React Hooks veränderten die State Management Landschaft fundamental [50]. Die ursprüngliche Context API war „essentially broken" und konnte updated Values nicht korrekt propagieren, was Redux als faktisch einzige Option für Application-Wide State etablierte [50]. Mit der neuen API wurde `useState` mit `useContext` zu einer viablen Alternative für einfachere Anwendungsfälle, was die „You don't need Redux at all" Bewegung initiierte.

**Spezialisierung und Komposition (2019–heute):** Die Erkenntnis, dass nicht alle State-Kategorien identische Management-Patterns erfordern, führte zu spezialisierten Lösungen [50]. Moderne Architekturen trennen explizit zwischen Client State Management (Zustand, Jotai, Recoil) und Server State Management (TanStack Query, Stale-While-Revalidate (SWR), Apollo) [26]. Diese Spezialisierung reduziert Komplexität durch Single Responsibility: Bibliotheken fokussieren sich auf spezifische Problemdomänen statt universeller Lösungen [15].

### 3.4.2 Die Client-Server State Dichotomie

Die fundamentale Unterscheidung zwischen Client State und Server State bildet die theoretische Grundlage moderner State Management Architekturen.

**Client State – Kurzlebige UI-Logik:** Client State umfasst Daten, die vollständig im Browser existieren und keine Server-Persistenz erfordern [78]. Charakteristisch sind:

- **Lokale UI-Zustände:** Modal-Visibility, Dropdown-States, Form-Input-Werte, Tab-Selection
- **Temporäre Daten:** Multi-Step-Form Progress, Undo-Redo History, Draft-Content
- **View-Layer Optimierungen:** Scroll-Position, Pagination-Cursor, Filter-Kriterien

Die Management-Strategie für Client State priorisiert Entwicklerergonomie und React-Integration, da dieser State keine Synchronisation mit Backend-Systemen erfordert [78].

**Server State – Asynchrone Remote-Daten:** Server State repräsentiert Daten, deren autoritative Quelle serverseitig liegt und die asynchron fetched, gecacht und synchronisiert werden müssen [64]. Die inhärenten Charakteristika unterscheiden sich fundamental von Client State [78]:

- **Asynchronität:** Fetching erfolgt über Network Requests mit unvorhersagbarer Latenz
- **Shared Ownership:** Andere Clients können dieselben Daten parallel modifizieren
- **Staleness:** Gecachte Daten werden potenziell obsolet und erfordern Revalidierung
- **Persistenz:** Daten persistieren über Sessions und Devices hinweg

Die herkömmliche Verwaltung von Server State in globalen Client State Managern (Redux, MobX) führt zu suboptimalen Patterns [64]. Entwickler müssen manuell Loading States, Error Handling, Cache Invalidation, Request Deduplication und Background Refetching implementieren – Komplexität, die spezialisierte Libraries wie TanStack Query abstrahieren [15].

### 3.4.3 Zustand als minimalistischer Client State Manager

Zustand etablierte sich als moderne Alternative zu Redux durch radikale Vereinfachung bei Beibehaltung essentieller State Management Patterns [51].

**Philosophische Grundprinzipien:** Zustand verfolgt eine „barebones“ Philosophie: minimale API-Surface, keine Opinions über State-Struktur, und vollständige Hook-basierte Integration [47]. Im Gegensatz zu Redux erfordert Zustand keine Provider-Wrapping der Application, keine Action Types, keine Reducer Boilerplate [51]. Der gesamte Store wird durch eine einzelne `create`-Function definiert [51]:

```
import { create } from 'zustand'

const useQuestStore = create((set, get) => ({
  // State
  activeFilter: 'all',
  selectedQuest: null,

  // Actions
  setFilter: (filter) => set({ activeFilter: filter }),
  selectQuest: (quest) => set({ selectedQuest: quest }),

  // Computed/Derived State via get()
  getFilteredQuests: () => {
    const filter = get().activeFilter
    // Filtering logic...
  }
}))
```

**Selektive Reactivity und Performance:** Zustand implementiert feingradiges Dependency Tracking durch Proxy-basierte Subscriptions [51]. Components subscriben nur zu spezifischen State-Slices via Selector-Functions, was unnötige Re-Renders minimiert [47]:

```
// Component re-rendert nur bei activeFilter Changes
const filter = useQuestStore(state => state.activeFilter)

// Component re-rendert nur bei selectedQuest Changes
const quest = useQuestStore(state => state.selectedQuest)
```

Diese automatische Optimization eliminiert die Notwendigkeit manueller Memoization via `React.memo` oder `useMemo` [51].

**DevTools und Persistence:** Zustand bietet Middleware für Redux DevTools Integration und LocalStorage Persistence [47]:

```
import { devtools, persist } from 'zustand/middleware'

const useStore = create(
  devtools(
    persist(
      (set) => ({ /* store definition */ }),
      { name: 'quest-filter-storage' }
    )
  )
)
```

Die Persist-Middleware ermöglicht automatisches Speichern und Laden von State aus LocalStorage, essentiell für Features wie Quest-Filter-Präferenzen, die über Sessions persistieren sollen [51].

**Trade-offs und Limitierungen:** Zustand's Minimalismus ist gleichzeitig Stärke und Limitation. Die Library bietet keine eingebauten Patterns für asynchrone Operations, keine strukturierten Side-Effect-Modelle wie Redux Saga, und ein kleineres Ecosystem als Redux [51]. Für komplexe State Machines oder stark strukturierte Business Logic kann Redux mit Redux Toolkit weiterhin überlegen sein.

### 3.4.4 TanStack Query als Server State Spezialist

TanStack Query (vormals React Query) revolutionierte Server State Management durch Abstraktion der inhärenten Komplexität asynchroner Datenoperationen [64].

**Deklaratives Data Fetching:** TanStack Query verschiebt Data Fetching von imperativem `useEffect`-Code zu deklarativen Query-Definitionen [15]:

```
import { useQuery } from '@tanstack/react-query'

function QuestList() {
  // Deklarative Query-Definition
  const { data, isLoading, error } = useQuery({
    queryKey: ['quests'],
    queryFn: () => fetch('/api/quests').then(r => r.json())
  })

  if (isLoading) return <LoadingSpinner />
  if (error) return <ErrorMessage error={error} />

  return <div>{data.map(quest => <QuestCard quest={quest} />)}</div>
}
```

Diese Deklaration eliminiert manuelle State-Variablen für `loading`, `error` und `data`, die traditionell in Redux oder lokalem State verwaltet werden mussten [64].

**Intelligent Caching und Query Keys:** Das Cache-System basiert auf Query Keys als eindeutigen Identifiern [19]. Beim Query-Execution prüft TanStack Query zunächst den Cache:

1. Existiert gecachte Data für den Query Key?
2. Ist die gecachte Data „fresh“ basierend auf `staleTime`?
3. Falls fresh: Sofortige Rückgabe aus Cache

4. Falls stale: Background Refetch während gecachte Data angezeigt wird [15]

Die Parameter `staleTime` und `gcTime` (vormals `cacheTime`) kontrollieren dieses Verhalten [19]:

- **staleTime:** Duration, für die Data als fresh gilt und nicht refetched wird (Default: 0ms)
- **gcTime:** Duration, für die unused Cache-Entries in Memory verbleiben (Default: 5min) [19]

**Request Deduplication:** Wenn multiple Components simultan denselben Query requesten, konsolidiert TanStack Query diese zu einem einzelnen Network Request [15]. Alle subscribenden Components erhalten dieselbe Response, was Redundanz eliminiert:

```
// Component A
const { data } = useQuery({ queryKey: ['quests'], queryFn: fetchQuests })

// Component B (mountet simultan)
const { data } = useQuery({ queryKey: ['quests'], queryFn: fetchQuests })

// Resultat: Nur 1 HTTP Request, beide Components erhalten Response
```

**Automatic Background Refetching:** TanStack Query implementiert intelligente Refetch-Strategien [64]:

- **Window Focus Refetching:** Queries refetchen, wenn User zur Tab zurückkehrt
- **Network Reconnection:** Automatic Refetch nach Netzwerk-Wiederverbindung
- **Polling:** Konfigurierbare Interval-basierte Refetches

Diese Features garantieren Data Freshness ohne manuelle Orchestration [15].

**Mutations und Optimistic Updates:** Für Data-Modification bietet TanStack Query `useMutation` mit Optimistic Update Support [63]:

```
const updateQuestMutation = useMutation({
  mutationFn: (updates) => fetch(`/api/quests/${updates.id}`, {
    method: 'PATCH',
    body: JSON.stringify(updates)
  }),

  // Optimistic Update
  onMutate: async (updates) => {
    // Cancel outgoing refetches
    await queryClient.cancelQueries({ queryKey: ['quests', updates.id] })

```

```
    // Snapshot previous value
    const previous = queryClient.getQueryData(['quests', updates.id])

    // Optimistically update cache
    queryClient.setQueryData(['quests', updates.id], updates)

    // Return context for rollback
    return { previous }
  },

  // Rollback on error
  onError: (err, updates, context) => {
    queryClient.setQueryData(['quests', updates.id], context.previous)
  },

  // Refetch after success/error
  onSettled: (data, error, updates) => {
    queryClient.invalidateQueries({ queryKey: ['quests', updates.id] })
  }
})
```

Optimistic Updates ermöglichen instant UI-Feedback: Der Cache wird sofort mit der antizipierten Response aktualisiert, bevor die Server-Response eintrifft [63]. Bei Fehlern erfolgt automatisches Rollback zur vorherigen Cache-Version.

**Cache Invalidation Strategien:** TanStack Query bietet granulare Cache Invalidation [64]:

```
// Invalidate specific query
queryClient.invalidateQueries({ queryKey: ['quests', questId] })

// Invalidate all quest queries
queryClient.invalidateQueries({ queryKey: ['quests'] })

// Invalidate with predicate
queryClient.invalidateQueries({
  predicate: (query) => query.queryKey[0] === 'quests' &&
   query.state.data?.status === 'active'
})
```

**Integration mit Server-Side Rendering:** TanStack Query unterstützt SSR durch Dehydration/Hydration Patterns [64]. Queries können serverseitig prefetched, dehydriert und im Client rehydriert werden, was FCP optimiert:

```
// Next.js getServerSideProps
```

```
2 export async function getServerSideProps() {
3   const queryClient = new QueryClient()
4
5   await queryClient.prefetchQuery({
6     queryKey: ['quests'],
7     queryFn: fetchQuests
8   })
9
10  return {
11    props: {
12      dehydratedState: dehydrate(queryClient)
13    }
14  }
15 }
```

## 3.5 Datenbank und Backend Design

Die Entwicklung moderner Webanwendungen erfordert fundierte Entscheidungen bezüglich der Backend-Architektur und Datenbankanbindung. Für die Arc Raiders Companion App stellt sich die Frage, welche Infrastruktur den optimalen Trade-off zwischen Entwicklungsgeschwindigkeit, Skalierbarkeit und langfristiger Wartbarkeit bietet. Die folgenden Abschnitte analysieren den Stand der Technik im Bereich Backend-as-a-Service-Plattformen und Object-Relational Mapping Lösungen.

### 3.5.1 Backend-as-a-Service: Paradigmenwechsel in der Backend-Entwicklung

Backend as a Service (BaaS) repräsentiert ein Cloud-Service-Modell, bei dem Entwickler alle serverseitigen Aspekte einer Web- oder Mobile-Anwendung auslagern können, um sich ausschließlich auf die Frontend-Entwicklung zu konzentrieren [13]. BaaS-Anbieter stellen vorgefertigte Software für serverseitige Aktivitäten bereit, darunter Benutzerauthentifizierung, Datenbankverwaltung, Remote-Updates und Push-Benachrichtigungen sowie Cloud-Speicher und Hosting [13].

**Konzeptuelle Grundlagen:** Das BaaS-Paradigma abstrahiert die Komplexität der Backend-Infrastrukturverwaltung, einschließlich Serverwartung, Datenbankadministration und Skalierungsherausforderungen. Die modulare Architektur kombiniert einfach zu verwaltende Funktionalitäten, um komplexe Backend-Operationen für Entwickler zu vereinfachen. Typische BaaS-Plattformen bieten Representational State Transfer (REST) APIs, die die Verwaltung

und Konfiguration von Architekturmodulen vereinfachen, einschließlich Authentifizierung und Datenbankänderungen.

**Strukturelle Vorteile und Limitierungen:** Mit vorgefertigten Backend-Services können Anwendungen schnell prototypisiert, entwickelt und deployed werden. Bei plötzlichem Nutzerwachstum skaliert die Infrastruktur im Hintergrund, was ein reibungsloses Benutzererlebnis gewährleistet. BaaS eliminiert die Notwendigkeit großer Anfangsinvestitionen in Backend-Infrastruktur und ermöglicht Pay-as-you-use-Modelle [13]. Als strukturelle Limitierung gilt jedoch das Risiko des Vendor Lock-In: Wenn eine BaaS-Plattform genutzt wird, ist das Backend der Anwendung tief mit deren Infrastruktur und Services integriert, was einen späteren Anbieterwechsel erschwert. Während BaaS Komfort und Effizienz bietet, kann das Niveau an Customization und Kontrolle geringer sein als bei maßgeschneiderter Entwicklung.

**Abgrenzung zu Serverless und Function as a Service (FaaS):** Es existiert eine gewisse Überschneidung zwischen BaaS und Serverless Computing, da bei beiden der Entwickler nur den Anwendungscode schreiben muss [13]. Die Backends von Serverless-Anwendungen sind jedoch in Funktionen aufgeteilt, die jeweils auf spezifische Events reagieren und nur eine Aktion ausführen. BaaS-serverseitige Funktionalitäten sind dagegen nach Belieben des Anbieters konstruiert, und Entwickler müssen sich nur um das Frontend kümmern [13]. BaaS adressiert Backend-Funktionalität als Ganzes, während FaaS nur Microservices in Anwendungen bedient, die auf auftretende Events reagieren.

### 3.5.2 Supabase als Open-Source Backend-Plattform

Supabase positioniert sich als Open-Source-Alternative zu Firebase und bietet eine Suite von Tools für sichere, skalierbare Web- und Mobile-Anwendungen [60]. Im Gegensatz zu Firebase, das primär NoSQL (Firestore) verwendet, basiert Supabase auf PostgreSQL und bringt ein standardbasiertes, relationales Datenmodell mit, während es die intuitive Entwicklererfahrung beibehält, die Firebase-Nutzer gewohnt sind.

**Architektonische Grundlagen:** Supabase kombiniert bewährte Open-Source-Tools wie PostgreSQL, Realtime, PostgREST, GoTrue, pg_graphql und Kong zu einer modernen Entwicklungsplattform [60]. Die Philosophie lautet: Wenn Werkzeuge und Communities mit MIT-, Apache-2- oder äquivalenter Open-License existieren, werden diese genutzt und unterstützt. Wenn ein Werkzeug nicht existiert, wird es selbst entwickelt und als Open Source veröffentlicht [60].

**PostgreSQL als Fundament:** PostgreSQL ist ein objektrelationales Datenbanksystem mit über 30 Jahren aktiver Entwicklung, das sich einen starken Ruf für Zuverlässigkeit, Feature-Robustheit und Performance erarbeitet hat [60]. Die Nutzung von PostgreSQL ermöglicht

erweiterte SQL-Abfragen, Row-Level Security (RLS), Transaktionen und strukturierte Datenschemata mit relationalen Constraints. Für viele Enterprise-Entwickler und erfahrene Ingenieure ist SQL essentiell, weshalb Supabase den optimalen Punkt zwischen einfachem Start und der Möglichkeit zur Skalierung trifft.

### Row Level Security: Datenbankseitige Zugriffskontrolle

RLS ist ein PostgreSQL-Sicherheitsfeature, das Datenbankadministratoren ermöglicht, Policies zu definieren, die kontrollieren, welche Zeilen Benutzer sehen oder modifizieren können [48]. RLS ist im Wesentlichen ein zusätzlicher Filter, der auf eine PostgreSQL-Datenbanktabelle angewendet wird. Wenn ein Benutzer versucht, eine Aktion auf einer Tabelle auszuführen, wird dieser Filter vor den Abfragekriterien angewendet, und die Daten werden gemäß der Sicherheitsrichtlinie eingeschränkt oder abgelehnt.

**Funktionsprinzip:** In PostgreSQL arbeiten RLS-Policies mit Rollen und Bedingungen [48]. Der allgemeine Prozess umfasst: Aktivierung von RLS auf Tabellenebene mit `ALTER TABLE ... ENABLE ROW LEVEL SECURITY`, Definition von Policies für spezifische Befehle (SELECT, INSERT, UPDATE, DELETE), Verwendung von Boolean-Ausdrücken zur Bestimmung sichtbarer/modifizierbarer Zeilen. Mehrere Policies können kombiniert werden: permissive Policies mit OR, restriktive Policies mit AND. Standardmäßig unterliegen Tabelleneigentümer nicht der Row-Level-Security, es sei denn, die Tabelle wird mit der Option `FORCE ROW LEVEL SECURITY` modifiziert [48].

**Relevanz für Multi-Tenancy:** Für SaaS-Plattformen ist RLS als Grundlage für Multi-Tenancy absolut erwägenswert. Anstatt sich darauf zu verlassen, dass jeder Entwickler Tenant-Bedingungen in Code hinzufügt, garantiert RLS auf Datenbankebene, dass mandantenübergreifender Datenzugriff nicht passieren kann. Dies ermöglicht Defense in Depth: Selbst wenn der Anwendungscode einen Bug enthält, wird die Datenbank keine Daten außerhalb des Tenant-Bereichs zurückgeben oder modifizieren.

**Supabase-Integration:** Wenn granulare Autorisierungsregeln benötigt werden, ist PostgreSQL's Row Level Security unübertroffen [59]. Supabase ermöglicht bequemen und sicheren Datenzugriff vom Browser aus, solange RLS aktiviert ist. RLS muss immer auf Tabellen aktiviert sein, die in einem exponierten Schema gespeichert sind [59]. RLS ist unglaublich leistungsfähig und flexibel und ermöglicht das Schreiben komplexer SQL-Regeln, die einzigartigen Geschäftsanforderungen entsprechen. In Kombination mit Supabase Auth bietet es End-to-End-Benutzersicherheit vom Browser bis zur Datenbank [59].

**Performance-Implikationen:** Jedes Autorisierungssystem hat Auswirkungen auf die Performance [59]. Dies gilt besonders für Abfragen, die jede Zeile einer Tabelle scannen, einschließlich vieler Select-Operationen mit Limit, Offset und Ordering. RLS-Policies werden für jede einzelne Zeile während der Abfrageausführung evaluiert. Best Practices umfassen: Hinzufügen von

Indizes auf Spalten, die in Policies verwendet werden, Verwendung von `SECURITY DEFINER`-Funktionen zur Vermeidung verketteter RLS-Policies, und strategische Denormalisierung für konsistente RLS-Performance.

### Realtime: Echtzeit-Datensynchronisation

Supabase Realtime ist ein Server, der mit Elixir unter Verwendung des Phoenix Frameworks entwickelt wurde und ermöglicht, auf Änderungen in der PostgreSQL-Datenbank via Logical Replication zu lauschen und diese dann über WebSockets zu broadcasten [58].

**Architektur:** Supabase Realtime ist ein global verteilter Elixir-Cluster [58]. Clients können sich über WebSockets mit jedem Node im Cluster verbinden und Nachrichten an jeden anderen verbundenen Client senden. Phoenix ist schnell und kann Millionen gleichzeitiger Verbindungen handhaben, da Elixir leichtgewichtige Prozesse (keine OS-Prozesse) bereitstellt. Channels werden mit Phoenix Channels implementiert, die Phoenix.PubSub mit dem Standard-PG2-Adapter nutzen. Der PG2-Adapter verwendet Erlang-Prozessgruppen zur Implementierung des PubSub-Modells [58].

**Funktionale Komponenten:** Das Realtime-System bietet drei Hauptfunktionen [58]:

- **Broadcast:** Senden ephemerer Nachrichten von Client zu Clients mit niedriger Latenz
- **Presence:** Tracking und Synchronisation von geteiltem State zwischen Clients
- **Postgres Changes:** Lauschen auf PostgreSQL-Datenbankänderungen und Senden an autorisierte Clients

**Technische Implementation:** Bei der Kernfunktionalität pollt Supabase Realtime die integrierte Replikationsfunktionalität von PostgreSQL auf Datenbankänderungen, konvertiert Änderungen zu JSON und broadcastet das JSON über WebSockets an autorisierte Clients [60]. Diese Architektur ermöglicht Event-basiertes Lauschen auf INSERT, UPDATE, DELETE oder alle Events, Schema- und Tabellen-Targeting sowie granulares Filtering.

### Edge Functions: Serverless Computing am Netzwerkrand

Supabase Edge Functions sind serverseitige TypeScript-Funktionen, die on-demand ausgeführt werden und global verteilt sind, was niedrige Latenzzeiten nahe der Nutzer gewährleistet [57]. Sie werden mit Deno entwickelt, einem Open-Source-JavaScript-Runtime, das maximale Leistung und Flexibilität gewährleistet.

**Technische Grundlagen:** Edge Functions nutzen das Supabase Edge Runtime (ein Deno-kompatibles Runtime) mit TypeScript-First-Ansatz [57]. Funktionen sind einfache .ts-Dateien,

die einen Handler exportieren. Der Workflow umfasst: Bundling der Funktion mit Abhängigkeiten in ein ESZip-File, Upload zum Supabase-Backend, Generierung einer eindeutigen URL für globalen Zugriff. Selbst initiale Ausführungen sind schnell (Millisekunden) dank des kompakten ESZip-Formats und des minimalen Deno-Runtime-Overheads. Isolates können für einen Zeitraum aktiv bleiben, um nachfolgende Requests ohne Neustart zu handhaben, und mehrere Isolates können gleichzeitig am selben Edge-Standort laufen [57].

**Vorteile des Edge-Computing-Modells:** Die Nähe zu Nutzern reduziert Round-Trip-Zeiten erheblich. Das System handhabt variable Lasten ohne Server-Provisionierung automatisch. Native TypeScript-Unterstützung erhöht die Entwicklererfahrung, und die Open-Source-Natur ermöglicht Ausführung auf jeder Deno-kompatiblen Plattform [57].

### 3.5.3 Object-Relational Mapping: Brücke zwischen Objekten und Relationen

Object-Relational Mapping (ORM) ist eine Programmiertechnik zur Konvertierung von Daten zwischen einer relationalen Datenbank und dem Speicher einer objektorientierten Programmiersprache [81]. Dies erzeugt effektiv eine virtuelle Objektdatenbank, die innerhalb des Programms verwendet werden kann.

**Das Object-Relational Impedance Mismatch:** Eine Vielzahl von Schwierigkeiten entsteht bei der Überlegung, wie ein Objektsystem mit einer relationalen Datenbank abgeglichen werden kann [81]. Diese Schwierigkeiten werden als Object-Relational Impedance Mismatch bezeichnet. Während Objekte Daten und Methoden enthalten, können relationale Datenbanken nur Daten speichern. Objekte sind oft aus anderen Objekten zusammengesetzt und Klassen erben Strukturen von anderen Klassen, aber relationalen Modellen fehlt ein eingebauter hierarchischer Mechanismus. Sie flachen die natürliche hierarchische Struktur eines Objekts in separate Tabellen mit atomaren Werten und eindeutigen Zeilen ab.

**Mapping-Patterns:** ORMs verwenden primär zwei Patterns zur Abbildung von Anwendungsobjekten auf Datenbankstrukturen: Das *Active Record Pattern* verknüpft eine Anwendung mit einem Datenbankschema, setzt Tabellen mit Klassen, Zeilen mit Objekten und Spalten mit Attributen gleich. Jede Klasse stellt die grundlegenden Methoden für CRUD-Operationen bereit. Das *Data Mapper Pattern* versucht hingegen, die Geschäftslogik in den Objekten von der Datenbank zu entkoppeln, was den Wechsel von Datenbanken erleichtert und die Wiederverwendung derselben Programmierlogik ermöglicht.

**Vorteile von ORM-Werkzeugen:** Verglichen mit traditionellen Techniken des Austauschs zwischen objektorientierter Sprache und relationaler Datenbank reduziert ORM oft die Menge an Code, die geschrieben werden muss [81]. ORM verbirgt und kapselt Änderungen in der

Datenquelle, sodass wenn sich Datenquellen oder ihre APIs ändern, nur ORM geändert werden muss. ORM-Tools sind darauf ausgelegt, die Möglichkeit von SQL-Injection-Angriffen zu eliminieren, da Queries prepared und sanitized werden.

**Nachteile von ORM-Werkzeugen:** Nachteile von ORM-Tools resultieren generell aus dem hohen Abstraktionsgrad, der verbirgt, was tatsächlich im Implementierungscode passiert [81]. Aufgrund der Abstraktionsschicht kann Software mit ORM langsamer laufen als mit rohem SQL, besonders bei komplexen Queries gegen große Datensätze. Ein bekanntes Problem ist das N+1-Problem: Wenn über eine Collection iteriert und für jeden Eintrag dynamisch verwandte Zeilen gefetcht werden, queried das ORM jede einzelne Zeile separat.

### 3.5.4 Drizzle ORM: TypeScript-native Datenbankabstraktion

Drizzle ORM ist ein modernes TypeScript-First-Werkzeug, das die Arbeit mit Datenbanken einfacher und weniger fehleranfällig macht [18]. Es positioniert sich komfortabel zwischen rohem SQL und TypeScripts Typsystem: Entwickler schreiben ihr Schema nach eigenem Stil, und Drizzle generiert automatisch Typen, sodass der Code sicher, sauber und konsistent bleibt.

**Philosophische Grundprinzipien:** Die Hauptphilosophie hinter Drizzle lautet: "If you know SQL, you know Drizzle" [18]. Im Gegensatz zur Abstraktion von SQL spiegelt Drizzle SQL in seiner API wider und bietet gleichzeitig einen höheren Abstraktionsgrad für Relational Queries. Drizzle nimmt einen anderen Ansatz als traditionelle ORMs: Es bietet einen typisierten SQL-Builder anstatt SQL hinter Abstraktionsschichten zu verstecken. Das Schema wird in TypeScript definiert, Queries geschrieben, die wie SQL aussehen (aber vollständig typsicher sind), und die Notwendigkeit für Runtime-Clients oder CLI-Generatoren entfällt.

**Architektonische Komponenten:** Statt eines monolithischen Designs folgt Drizzle einem dezentralisierten, composition-first Ansatz, bei dem jeder Teil eine klare, einzelne Verantwortung hat und unabhängig verwendet werden kann [18]:

- **Schema Definition:** Eine deklarative, TypeScript-basierte API zur Definition der Datenbankstruktur
- **SQL Driver Adapters:** Leichtgewichtige Wrapper um Datenbanktreiber, die standardisieren, wie Drizzle mit ihnen kommuniziert
- **Query Builder:** Ein Set komposierbarer Funktionen, die SQL generieren und dabei vollständige Type-Safety bewahren
- **Type Inference:** Ein leistungsstarkes System, das TypeScript-Typen direkt aus dem Schema extrahiert

- **Migration Tools:** Bereitgestellt durch drizzle-kit, helfen diese bei der Generierung und Ausführung von Datenbankmigrationen

**Type Safety und Developer Experience:** Eines der herausragenden Merkmale von Drizzle ORM ist Type Safety [2]. TypeScripts statische Typprüfung stellt sicher, dass die Daten, die aus der Datenbank abgerufen werden, und die geschriebenen Queries konsistent mit dem definierten Datenbankschema sind. Dies hilft, Fehler früh im Entwicklungsprozess statt zur Laufzeit zu erkennen [2]. Die Dinge, die Entwickler an Drizzle besonders schätzen, umfassen: Type Safety mit sehr starkem Typsystem, Performance-Optimierung, reichhaltige SQL-Dialekte, zero Dependencies, kein Code-Generation-Erfordernis, und Edge-Readiness.

**Code-First-Methodologie:** Drizzle ermutigt zu einer Code-First-Methodologie, bei der das TypeScript-Schema als Single Source of Truth dient. Das bedeutet, dass das Datenbankschema direkt im Codebase definiert wird, was Konsistenz und Versionskontrolle gewährleistet. Da Drizzle keinen Generate-Step wie Prisma nach jeder Schema-Änderung erfordert, funktionieren die Typen live und sind nicht von der Ausführung eines Befehls abhängig.

## 3.6 Deployment und DevOps

Die Bereitstellung moderner Webanwendungen erfordert ein fundiertes Verständnis der zugrundeliegenden Deployment-Strategien und DevOps-Praktiken. Die Arc Raiders Companion App steht vor der Herausforderung, einen effizienten, zuverlässigen und skalierbaren Deployment-Prozess zu etablieren. Die folgenden Abschnitte analysieren die theoretischen Grundlagen und den Stand der Technik im Bereich DevOps und modernes Web-Deployment.

### 3.6.1 DevOps: Theoretische Fundierung und Kernprinzipien

DevOps beschreibt die Integration und Automatisierung von Softwareentwicklung und IT-Operations mit dem Ziel, den Entwicklungszyklus zu verkürzen und eine kontinuierliche Auslieferung hochwertiger Software zu ermöglichen [38]. Obwohl keine einheitliche Definition existiert, wird DevOps allgemein als Bewegung verstanden, die darauf abzielt, die traditionelle Kluft zwischen Entwicklung (Dev) und Betrieb (Ops) zu überbrücken und schnellere, agilere Software-Deployments zu ermöglichen.

**Die Three Ways als theoretisches Fundament:** Kim et al. identifizieren in "The DevOps Handbook" drei fundamentale Prinzipien, aus denen sich alle DevOps-Patterns ableiten lassen [38]. Der *First Way* (Flow) betont die Performance des gesamten Systems und fokussiert auf den schnellen Links-nach-Rechts-Fluss der Arbeit von Development über Operations zum Kunden. Dies beinhaltet, niemals bekannte Defekte an nachgelagerte Arbeitszentren weiterzugeben und stets ein tiefes Systemverständnis anzustreben. Der *Second Way* (Feedback)

etabliert Rechts-nach-Links-Feedback-Loops mit dem Ziel, diese zu verkürzen und zu verstärken, sodass notwendige Korrekturen kontinuierlich vorgenommen werden können. Der *Third Way* (Continual Learning and Experimentation) kultiviert kontinuierliches Experimentieren, Risikobereitschaft und Lernen aus Fehlern sowie das Verständnis, dass Wiederholung und Übung die Voraussetzung für Meisterschaft sind [38].

**Continuous Delivery als Enabler:** Humble und Farley definieren Continuous Delivery als die Fähigkeit, Änderungen aller Art – einschließlich neuer Features, Konfigurationsänderungen, Bugfixes und Experimente – sicher, schnell und nachhaltig in die Produktion zu bringen [33]. Die Autoren betonen, dass Wiederholbarkeit und Zuverlässigkeit aus zwei Prinzipien resultieren: der Automatisierung nahezu aller Prozesse und der Versionskontrolle aller Artefakte, die für Build, Deploy, Test und Release benötigt werden. Continuous Delivery unterscheidet sich von Continuous Deployment dadurch, dass bei ersterem die Automatisierung vor der Produktionsfreigabe pausiert und eine manuelle Freigabe erfordert, während letzteres den gesamten Release-Prozess vollständig automatisiert.

### 3.6.2 DORA-Metriken: Empirische Messung von Software Delivery Performance

Die wissenschaftliche Fundierung von DevOps-Praktiken wurde maßgeblich durch die Forschung von Forsgren, Humble und Kim vorangetrieben, deren vierjährige Studie in "Accelerate" die Grundlage für die heute weitverbreiteten DORA-Metriken bildet [25]. Die Autoren entwickelten mittels rigoroser statistischer Methoden vier Schlüsselmetriken zur Messung der Software Delivery Performance.

**Throughput-Metriken:** Die *Deployment Frequency* misst, wie häufig eine Organisation Code erfolgreich in Produktion deployt. Elite-Performer erreichen On-Demand-Deployments, oft mehrmals täglich, während Low-Performer monatlich oder seltener deployen [25]. Die *Lead Time for Changes* erfasst die durchschnittliche Zeit von einem Code-Commit bis zur produktiven Bereitstellung. Diese Metrik reflektiert die Effizienz des gesamten Software-Delivery-Prozesses und die Fähigkeit eines Teams, auf neue Anforderungen zu reagieren.

**Stability-Metriken:** Die *Change Failure Rate* misst den Prozentsatz der Deployments, die zu Produktionsfehlern führen und Hotfixes oder Rollbacks erfordern. Eine niedrigere Rate indiziert einen zuverlässigeren Delivery-Prozess. Die *Failed Deployment Recovery Time* (früher Mean Time to Recovery) erfasst die Zeit, die benötigt wird, um sich von einem fehlgeschlagenen Deployment zu erholen. Kürzere Recovery-Zeiten indizieren resilientere und reaktionsfähigere Systeme [17].

**Zentrale Forschungsergebnisse:** Die DORA-Forschung belegt empirisch, dass Geschwindigkeit und Stabilität keine Trade-offs darstellen. Elite-Performer erreichen sowohl höhere

Deployment-Frequenzen als auch niedrigere Fehlerraten [25]. Organisationen, die Elite-Status in den DORA-Metriken erreichen, haben eine doppelt so hohe Wahrscheinlichkeit, ihre organisatorischen Leistungsziele zu erreichen. Die Studie identifiziert zudem 24 technische und kulturelle Capabilities, die höhere Performance treiben, darunter Versionskontrolle, automatisiertes Testing, Trunk-based Development und eine Kultur des kontinuierlichen Lernens.

### 3.6.3 CI/CD-Pipelines: Automatisierung des Software-Lebenszyklus

Continuous Integration (CI)/Continuous Deployment (CD) bildet das technische Rückgrat moderner DevOps-Praktiken und automatisiert den gesamten Prozess von der Code-Integration bis zum Deployment. Continuous Integration bezeichnet die Praxis, neue Code-Änderungen automatisch zu bauen, zu testen und in ein Repository zu integrieren. Continuous Delivery bzw. Deployment automatisiert den nachfolgenden Bereitstellungsprozess [30].

**Strukturelle Komponenten:** Eine typische CI/CD-Pipeline umfasst mehrere Phasen: In der *Build-Phase* werden Code und Abhängigkeiten zu einem ausführbaren Artefakt kompiliert. Die *Test-Phase* validiert durch automatisierte Tests, dass der Code erwartungsgemäß funktioniert. In der *Staging-Phase* wird die Anwendung in einer produktionsähnlichen Umgebung ausgeführt. Die *Deployment-Phase* stellt die Anwendung automatisch den Endnutzern bereit [30].

**GitHub Actions als Implementierungsbeispiel:** GitHub Actions ist ein in GitHub integriertes Automatisierungswerkzeug, das Entwicklern ermöglicht, Workflows direkt in ihren Git-Repositories zu definieren. Workflows werden in YAML-Dateien spezifiziert und können durch verschiedene Events wie Push, Pull Request oder Schedule getriggert werden. Die Plattform unterstützt Linux, macOS, Windows und Container-Umgebungen und bietet Zugang zu einem umfangreichen Marketplace vorgefertigter Actions [30]. Der State of DevOps Report zeigt, dass Organisationen mit ausgereiften CI/CD-Praktiken 208-mal häufiger deployen und eine 106-mal kürzere Lead Time aufweisen als der Rest [25].

**Vorteile automatisierter Pipelines:** CI/CD-Pipelines ermöglichen erhöhte Entwicklungsgeschwindigkeit durch schnelleres Feedback und häufigere, kleinere Commits. Die Stabilität und Zuverlässigkeit steigt, da automatisiertes Testing sicherstellt, dass Codebases jederzeit release-ready bleiben. Die Integration von Security-Checks (DevSecOps) ermöglicht frühzeitige Erkennung von Sicherheitsproblemen. Skalierbarkeit wird gewährleistet, da robuste CI/CD-Setups mühelos mit wachsenden Teams und Projektkomplexität expandieren können [30].

### 3.6.4 Frontend-Deployment-Plattformen: Vercel als Framework-aware Infrastructure

Moderne Frontend-Deployment-Plattformen wie Vercel repräsentieren einen Paradigmenwechsel in der Art, wie Webanwendungen bereitgestellt und betrieben werden. Vercel bietet eine auf Frontend-Entwicklung optimierte Cloud-Infrastruktur, die insbesondere für Next.js-Anwendungen entwickelt wurde [72].

**Framework-Defined Infrastructure:** Vercel implementiert das Konzept der Framework-Defined Infrastructure, bei dem die Infrastruktur automatisch basierend auf dem gewählten Framework konfiguriert wird. Bei einem Deploy analysiert Vercel die Anwendung, erkennt das Framework und richtet automatisch die optimalen Tools und Optimierungen ein [72]. Für Next.js umfasst dies die automatische Konfiguration von Server-Side Rendering, Static Site Generation, Incremental Static Regeneration und Edge Functions.

**Serverless Functions und Edge Computing:** Vercel Functions ermöglichen die Ausführung serverseitigen Codes ohne Serververwaltung. Die Funktionen skalieren automatisch basierend auf der Nutzernachfrage. Im Gegensatz zu traditionellen Serverless Functions, die in spezifischen Regionen laufen, werden Edge Functions global an Edge-Standorten ausgeführt und reagieren signifikant schneller durch die Nähe zu den Nutzern [72]. Edge Functions nutzen ein leichtgewichtiges Runtime basierend auf der V8-Engine und können bis zu 40% schneller als herkömmliche Serverless Functions bei einem Bruchteil der Kosten sein.

**Fluid Compute:** Vercel's Fluid Compute-Technologie ermöglicht die gleichzeitige Ausführung mehrerer Requests innerhalb derselben Serverless-Instanz. Durch die Nutzung von Leerlaufzeiten für Compute werden Cold Starts reduziert, die Latenz gesenkt und Compute-Kosten gespart. Dies verhindert die Notwendigkeit, multiple isolierte Instanzen hochzufahren, wenn Tasks die meiste Zeit auf externe Operationen warten [72].

### 3.6.5 Preview Deployments: Kollaborative Entwicklung durch Branch-basierte Umgebungen

Preview Deployments repräsentieren einen fundamentalen Fortschritt in der kollaborativen Webentwicklung. Sie ermöglichen die automatische Erstellung einer Live-URL für jeden Branch oder Pull Request, sodass Änderungen in Isolation getestet und reviewed werden können, bevor sie in den Hauptbranch gemergt werden [74].

**Funktionsprinzip:** Standardmäßig erstellt Vercel ein Preview Deployment, wenn ein Commit zu einem Branch gepusht wird, der nicht der Production-Branch ist. Jedes Deployment erhält eine automatisch generierte URL, und Links erscheinen typischerweise in den PR-Kommentaren des Git-Providers. Dies ermöglicht instantanes visuelles Feedback für Reviewer,

die Features ohne technisches Setup betrachten können, sichere Testumgebungen als einzigartige Sandboxes für jeden PR und schnellere, bessere Code-Reviews durch Einbeziehung von Design-Teams, QA und nicht-technischen Stakeholdern [74].

**Kollaborationsvorteile:** Preview Deployments reduzieren das "It worked on my machine"-Problem und ermöglichen Teams, mit höherer Konfidenz zu shippen. Bugs oder Design-Probleme werden früher und im Kontext bemerkt, da jeder PR isoliert ist. Die Separation von Entwicklungs-, Staging- und Produktionsumgebungen verbessert die Zusammenarbeit und minimiert Risiken für Endnutzer während des gesamten Software Development Lifecycles [74].

### 3.6.6 Theoretische Synthese für die Arc Raiders Companion App, (falsches kapitel?)

Die Integration der beschriebenen Konzepte ermöglicht ein fundiertes theoretisches Framework für die Deployment-Strategie der Arc Raiders Companion App. Die Three Ways nach Kim et al. bilden das kulturelle und prozessuale Fundament, während die DORA-Metriken empirisch validierte Erfolgsindikatoren bereitstellen. Die Kombination von Vercel als Framework-aware Platform mit GitHub Actions für CI/CD implementiert die technischen Prinzipien von Continuous Delivery.

Für eine studentische Companion-App-Entwicklung bietet dieser Stack mehrere Vorteile: Die automatische Infrastrukturkonfiguration durch Framework-Defined Infrastructure eliminiert komplexe DevOps-Aufgaben. Preview Deployments ermöglichen kollaboratives Feedback ohne separate Staging-Infrastruktur. Die serverlose Architektur skaliert automatisch und minimiert Betriebskosten bei variablen Nutzungsmustern. Die Git-Integration gewährleistet Traceability und ermöglicht die Messung von Lead Time und Deployment Frequency als zentrale DORA-Metriken.

## 3.7 Testing Frameworks & Strategien

### 3.7.1 Vitest

Unit und Integration Tests

fast, modern, built for TS

### 3.7.2 Cypress

End-to-End Testing

real browser testing

# 4 Durchführung

## 4.1 Anforderungserhebung

Die Anforderungserhebung bildet den Ausgangspunkt der Entwicklung und folgt dem in Abschnitt 2.2.1 definierten Phasenmodell. Ziel ist die systematische Transformation impliziter Nutzerbedürfnisse in explizite, implementierbare Anforderungen. Dabei kommen drei komplementäre Erhebungsmethoden zum Einsatz, deren Ergebnisse im Folgenden dargestellt werden.

### 4.1.1 Methodische Grundlagen der Erhebung

Die Anforderungserhebung stützt sich auf eine methodische Triangulation, die unterschiedliche Perspektiven auf die Problemdomäne vereint. Tabelle 4.1 gibt einen Überblick über die eingesetzten Methoden und deren spezifischen Beitrag zum Anforderungskatalog.

Tabelle 4.1: Übersicht der Anforderungsquellen und deren Beitrag

| **Methode** | **Durchführung** | **Identifizierte Anforderungen** |
|---|---|---|
| Empirische Datenerhebung | Technischer Spieltest mit systematischer Protokollierung | Quest-Tracking, Item-Datenbank, Material-Kalkulation, Workstation-Planung |
| Stakeholder-Interviews | Strukturierte Gespräche mit Spielern verschiedener Erfahrungsstufen | Recycling-Kalkulator (nachträglich priorisiert) |
| Comparative Analysis | Heuristische Evaluation etablierter Companion Apps | Kanban-Board, Flow-Chart- und weitere Visualisierungen |

Die empirische Datenerhebung während des Spieltests ermöglichte die Identifikation konkreter Pain Points aus der Nutzerperspektive. Dabei kristallisierten sich drei zentrale Problemfelder heraus, die bereits in Abschnitt 1.2 beschrieben wurden: die mangelnde Übersicht über Quest-Abhängigkeiten, die Komplexität der Ressourcenplanung sowie fehlende Unterstützung bei der Squad-Koordination.

Die Stakeholder-Interviews ergänzten diese Erkenntnisse um Anforderungen, die durch reine Selbstbeobachtung nicht erfasst werden konnten. Insbesondere das Recycling-Feature wurde

erst durch ein Interview mit einem erfahrenen Spieler als kritischer Bedarf identifiziert, wie in Abschnitt 4.1.4 detailliert dargestellt wird.

Die Comparative Analysis etablierter Companion Apps lieferte Best Practices für UI/User Experience (UX)-Patterns bei der Darstellung komplexer Spielinformationen. Hieraus wurden insbesondere das Kanban-Board-Konzept für die Quest-Übersicht sowie die Flow-Chart-Visualisierung für Abhängigkeitsdarstellungen abgeleitet.

### 4.1.2 Transformation in User Stories

Die identifizierten Nutzerbedürfnisse werden nach dem Independent, Negotiable, Valuable, Estimable, Small, Testable (INVEST)-Schema in User Stories transformiert [79]. Jede User Story folgt dem etablierten Format:

> *„Als [Rolle] möchte ich [Funktion], um [Nutzen] zu erreichen."*

Das INVEST-Akronym definiert dabei die Qualitätskriterien für gut formulierte User Stories: **I**ndependent (unabhängig voneinander umsetzbar), **N**egotiable (verhandelbar im Scope), **V**aluable (Wertvoll für den Nutzer), **E**stimable (schätzbar im Aufwand), **S**mall (klein genug für eine Iteration) und **T**estable (durch Akzeptanzkriterien validierbar).

Exemplarisch werden im Folgenden drei User Stories aus unterschiedlichen Feature-Bereichen vorgestellt, die die Bandbreite der funktionalen Anforderungen repräsentieren.

#### US-QM-01: Quest-Übersicht als Kanban-Board

**Quelle:** Issue ARC-24 **Phase:** P1-Visualization **Priorität:** Must-Have

> *„Als Spieler möchte ich alle Quests in einem Kanban-Board mit den Spalten „Active", „Locked" und „Completed" sehen, um meinen aktuellen Fortschritt auf einen Blick zu erfassen."*

Diese User Story adressiert den in der Problemstellung identifizierten Mangel an Übersichtlichkeit bei der Quest-Verwaltung. Das Kanban-Board-Pattern wurde aus der Comparative Analysis als bewährtes Konzept für die Statusvisualisierung übernommen und auf den Gaming-Kontext adaptiert.

**US-RC-02: Reverse-Engineering von Recycling-Pfaden**

**Quelle:** Stakeholder-Interview **Phase:** P1-Visualization (repriorisiert) **Priorität:** Must-Have

> *„Als Spieler möchte ich für ein Zielmaterial alle möglichen Recycling-Pfade sehen, um zu verstehen, welche Items ich recyceln kann, um das benötigte Material zu erhalten."*

Diese User Story entstand aus einem Stakeholder-Interview und wurde aufgrund ihres hohen Nutzwerts nachträglich in die erste Entwicklungsphase aufgenommen. Die Entstehungsgeschichte wird in Abschnitt 4.1.4 als Fallbeispiel für adaptives Anforderungsmanagement dokumentiert.

**US-MC-01: Material-Aggregation für Planung**

**Quelle:** Issue ARC-34 **Phase:** P1-Visualization **Priorität:** Must-Have

> *„Als Spieler möchte ich basierend auf ausgewählten Quests und Workstation-Upgrades die Gesamtmenge benötigter Materialien berechnen, um mein begrenztes Inventar optimal zu nutzen."*

Diese User Story adressiert direkt das Problem der ineffizienten Ressourcenplanung. Die Aggregationsfunktion ermöglicht eine vorausschauende Planung, die im Spielkontext durch das begrenzte Inventar besonders relevant ist.

### 4.1.3 Definition von Akzeptanzkriterien

Jede User Story wird durch messbare Akzeptanzkriterien konkretisiert, die eine objektive Validierung der Implementierung ermöglichen. Die Kriterien folgen dem Specific, Measurable, Achievable, Relevant, Time-bound (SMART)-Schema, dessen Dimensionen in Abbildung 4.1 visualisiert sind.

Die SMART-Kriterien stellen sicher, dass Akzeptanzkriterien **S**pezifisch (eindeutig und abgegrenzt), **M**essbar (quantifizierbar), **A**chievable (technisch umsetzbar), **R**elevant (beitragend zum Nutzwert) und **T**ime-bound (zeitlich einordenbar) formuliert sind.

Exemplarisch zeigt Tabelle 4.2 die Akzeptanzkriterien für die User Story US-QM-01 (Kanban-Board für Quest-Übersicht).

Die Akzeptanzkriterien definieren präzise Metriken (Spaltenanzahl, Latenz in Millisekunden) und den Zeitpunkt der Validierung (bei Seitenladung, bei Interaktion). Diese Präzision ermöglicht eine eindeutige Überprüfung im Rahmen der Testphase.

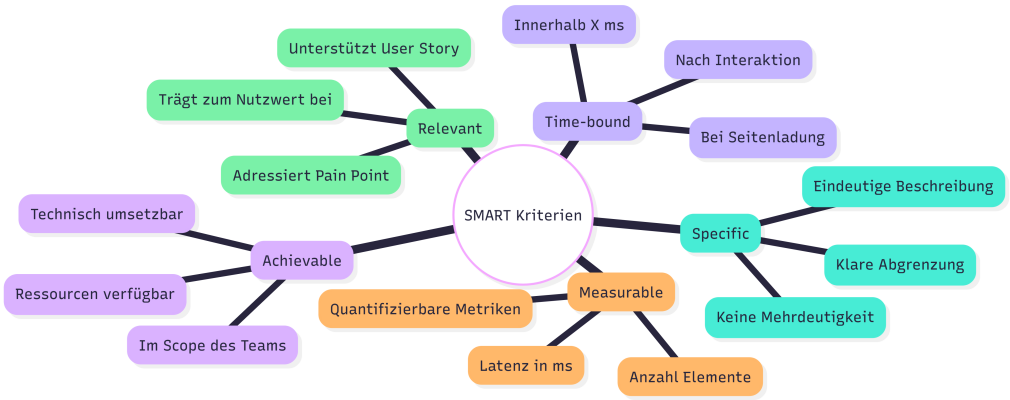


Abbildung 4.1: Dimensionen der SMART-Kriterien für Akzeptanzkriterien

Tabelle 4.2: Akzeptanzkriterien für US-QM-01 (Quest-Kanban-Board)

| ID | Kriterium | Messbar | Zeitpunkt |
|---|---|---|---|
| AC-QM-01.1 | Alle Quests in drei Spalten kategorisiert | Spaltenanzahl = 3 | Bei Seitenladung |
| AC-QM-01.2 | Echtzeit-Filterung nach Quest-Name | Latenz $<$ 100ms | Bei Tastatureingabe |
| AC-QM-01.3 | Numerische Quest-Anzahl pro Spalte | Counter sichtbar | Permanent |

### 4.1.4 Fallbeispiel: Adaptives Anforderungsmanagement

Das Recycling-Feature demonstriert exemplarisch die Anwendung agiler Prinzipien im Anforderungsmanagement. Es illustriert, wie durch kontinuierliche Stakeholder-Einbindung Features mit hohem Nutzwert identifiziert werden können, die durch reine Selbstbeobachtung nicht erkannt wurden.

#### Ausgangssituation

Das initiale Backlog, abgeleitet aus dem Spieltest und der Comparative Analysis, sah die Interaktiven Karten (Issue ARC-37) als nächstes Feature der Phase P1-Visualization vor. Diese Priorisierung basierte auf der Annahme, dass Kartenvisualisierungen einen hohen Nutzwert für die Squad-Koordination bieten würden.

#### Identifikation durch Stakeholder-Interview

Während eines strukturierten Interviews in der zweiten Entwicklungsiteration artikulierte ein erfahrener Arc Raiders-Spieler folgenden Pain Point:

> *„Ich brauche immer ewig um die richtigen Gegenstände zum recyclen zu finden, da man bei jedem Gegenstand einzeln das Menü öffnen muss, um zu erfahren in welche Materialien er zerlegt werden kann.“*

Diese Aussage offenbarte ein fundamentales Problem der Ressourcenverwaltung, das im Spieltest nicht als solches erkannt wurde. Das fehlende Verständnis der Recycling-Mechaniken führt zu suboptimalen Entscheidungen und damit zu ineffizienter Nutzung des ohnehin begrenzten Inventarplatzes.

### Analyse und Entscheidungsfindung

Die Bewertung des neu identifizierten Bedarfs erfolgte anhand von vier Kriterien, die in Tabelle 4.3 den ursprünglich geplanten Interaktiven Karten gegenübergestellt werden.

Tabelle 4.3: Entscheidungsmatrix zur Repriorisierung

| **Kriterium** | **Recycling-Kalkulator** | **Interaktive Karten** |
|---|---|---|
| Identifizierter Nutzerbedarf | Hoch (validiert durch Interview) | Mittel (Annahme) |
| Technische Komplexität | Mittel | Hoch (externe Kartendaten) |
| Geschätzte Entwicklungszeit | $\sim$40 Stunden | $\sim$60 Stunden |
| Wissenschaftlicher Mehrwert | Hoch (Graph-Algorithmen) | Mittel |

Der Recycling-Kalkulator überzeugte durch einen validierten Nutzerbedarf, geringere technische Komplexität und kürzere Entwicklungszeit. Zusätzlich bot die Implementierung von Graph-Algorithmen für das Reverse-Engineering der Recycling-Pfade einen höheren wissenschaftlichen Mehrwert im Kontext dieser Arbeit.

### Resultierende Repriorisierung

Basierend auf dieser Analyse wurde der Recycling-Kalkulator über die Interaktiven Karten priorisiert. Diese Entscheidung folgt dem agilen Grundprinzip: *„Reagieren auf Veränderung ist wertvoller als das Befolgen eines Plans“* [7].

Das Recycling-Feature wurde in drei User Stories unterteilt:

- **US-RC-01:** Recycling-Datenbank mit Effizienzberechnung
- **US-RC-02:** Reverse-Engineering von Recycling-Pfaden
- **US-RC-03:** Visualisierung der Recycling-Ketten als Graph

Die Implementierung erfolgte in der ersten Entwicklungsphase (P1-Visualization), während die Interaktiven Karten auf eine spätere Iteration verschoben wurden.

#### Erkenntnisse für den Entwicklungsprozess

Das Fallbeispiel verdeutlicht drei zentrale Aspekte agilen Anforderungsmanagements:

Erstens kann kontinuierliche Stakeholder-Einbindung Features mit hohem Nutzwert identifizieren, die durch isolierte Analyse nicht erkannt werden. Die Recycling-Problematik war während des eigenen Spieltests nicht als kritisch wahrgenommen worden, da sie erst bei fortgeschrittenem Spielfortschritt relevant wird.

Zweitens ermöglicht flexible Priorisierung die Reaktion auf neue Erkenntnisse, ohne den Gesamtplan zu gefährden. Die Verschiebung der Interaktiven Karten hatte keine negativen Auswirkungen auf das MVP, da keine anderen Features von ihnen abhängig waren.

Drittens erhöht die Dokumentation der Entscheidungsgrundlage die Nachvollziehbarkeit im wissenschaftlichen Kontext. Die explizite Gegenüberstellung der Optionen legitimiert die Reprioritisierung und macht den agilen Entscheidungsprozess transparent.

### 4.1.5 Priorisierung nach MoSCoW

Die Must-Have, Should-Have, Could-Have, Won't-Have (MoSCoW)-Methode ermöglicht eine systematische Kategorisierung der Anforderungen nach ihrer Relevanz für das MVP [11]. Die Priorisierung berücksichtigt sowohl den erwarteten Nutzwert als auch den geschätzten Implementierungsaufwand.

Abbildung 4.2 visualisiert die Einordnung der identifizierten Features in einem Quadranten-Diagramm mit den Achsen „Nutzwert“ und „Aufwand“. Features im oberen linken Quadranten (hoher Wert, niedriger Aufwand) werden als Must-Have klassifiziert, während Features im unteren rechten Quadranten (niedriger Wert, hoher Aufwand) auf spätere Releases verschoben werden.

Tabelle 4.4 fasst die resultierende Kategorisierung zusammen und begründet die Zuordnung der einzelnen Feature-Gruppen.

Die Must-Have-Features bilden das MVP und adressieren direkt die drei Kernprobleme aus der Problemstellung: Informationsasymmetrie bei Quests (Kanban, Flow-Chart, Details), ineffizientes Ressourcenmanagement (Material-Calculator, Recycling-Features) und fehlende Planungsgrundlage (Item-Katalog, Workstation-Übersicht).

Die Should-Have-Features erweitern die Kernfunktionalität um Komfortfunktionen wie die Persistierung des Spielerfortschritts und aggregierte Statistiken. Diese Features erhöhen den Nutzwert erheblich, sind jedoch für die grundlegende Funktionsfähigkeit der Anwendung nicht zwingend erforderlich.

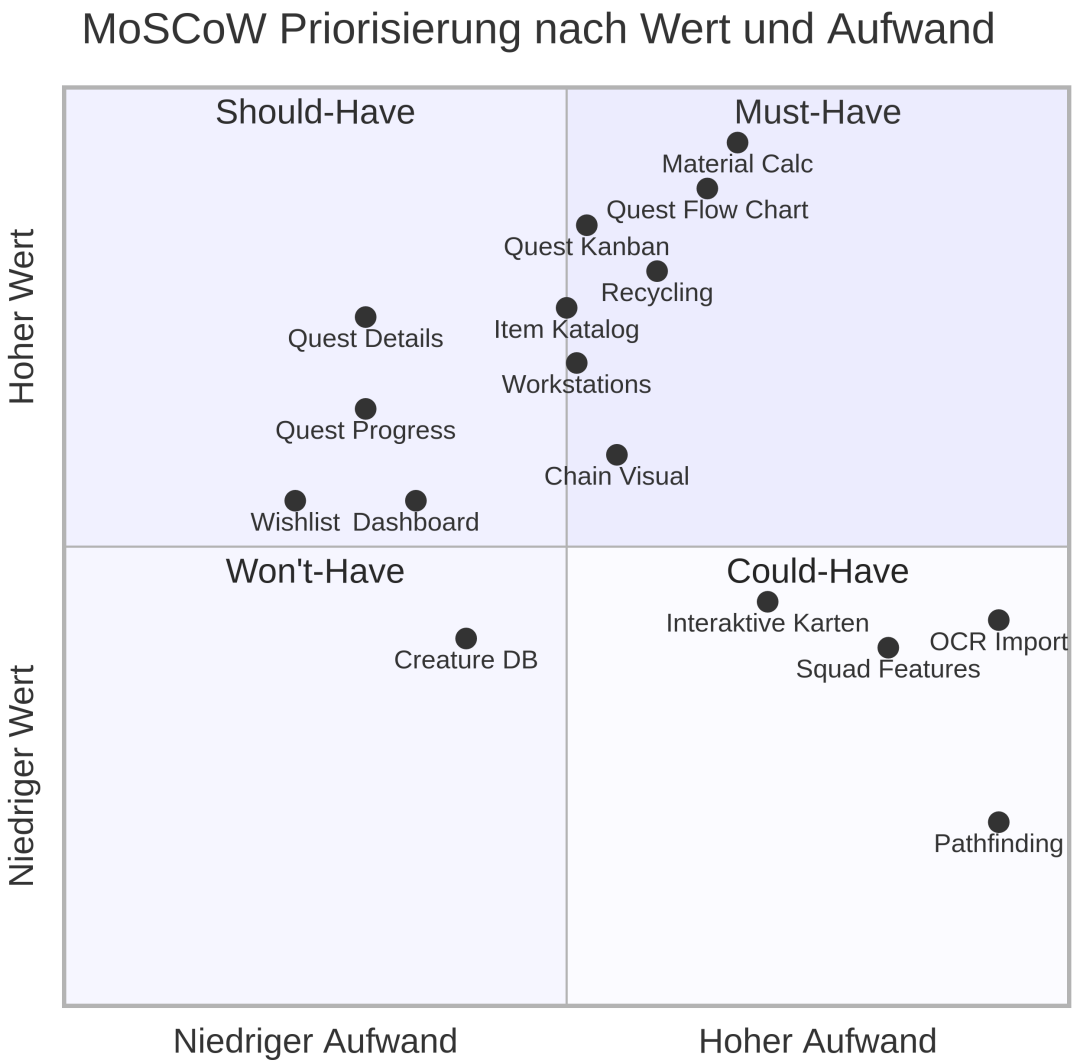


Abbildung 4.2: MoSCoW-Priorisierung der Features nach Nutzwert und Aufwand

Tabelle 4.4: MoSCoW-Kategorisierung der Anforderungen

| Kategorie | Features | Begründung |
|---|---|---|
| Must-Have | Quest-Kanban, Flow-Chart, Quest-Details, Item-Katalog, Workstation-Übersicht, Material-Calculator, Recycling | Adressieren die in Abschnitt 1.2 definierten Kernprobleme; bilden funktionales MVP |
| Should-Have | Quest-Progress und -Details, Dashboard-Statistiken, Wishlist-Integration | Erhöhen den Nutzwert signifikant; Anwendung funktional auch ohne diese Features |
| Could-Have | Squad-Features, Pathfinding-Algorithmen, OCR-basierter Screenshot-Import, Interaktive Karten | Hoher Aufwand; für spätere Releases nach Validierung des MVP geplant |
| Won't-Have | Creature-Datenbank | Zusatzfeatures mit geringerer Dringlichkeit |

Die Could-Have- und Won't-Have-Features wurden bewusst zurückgestellt, um den Fokus auf die Kernfunktionalität zu wahren. Insbesondere die Squad-Features und Pathfinding-Algorithmen erfordern erheblichen Entwicklungsaufwand und sind erst nach erfolgreicher Validierung des MVP sinnvoll zu implementieren.

## 4.2 Technologieentscheidungen

Die Auswahl geeigneter Technologien bildet das Fundament für eine erfolgreiche Implementierung. Anstelle einer ad-hoc-Entscheidung wird ein systematischer Evaluationsansatz mittels MCDA angewandt, der die Nachvollziehbarkeit und Objektivität der Technologiewahl gewährleistet. Dabei werden bewusst unterschiedliche Architekturansätze verglichen: von vollständig verwalteten Platform as a Service (PaaS)-Lösungen über klassische Infrastructure as a Service (IaaS)-Infrastruktur bis hin zu Self-Hosting-Optionen.

### 4.2.1 Methodisches Vorgehen: Multi-Criteria Decision Analysis

Die MCDA ist ein strukturiertes Verfahren zur Entscheidungsfindung bei mehreren, teilweise konkurrierenden Kriterien [8]. Das Vorgehen umfasst vier Schritte:

1. **Kriteriendefinition:** Identifikation relevanter Bewertungskriterien basierend auf den Projektanforderungen

2. **Gewichtung:** Zuweisung von Gewichtungsfaktoren entsprechend der relativen Bedeutung (Summe = 100%)
3. **Bewertung:** Qualitative Einschätzung jeder Alternative pro Kriterium auf einer Skala von 1 (ungenügend) bis 5 (exzellent)
4. **Aggregation:** Berechnung gewichteter Gesamtpunktzahlen zur Identifikation der optimalen Lösung

### 4.2.2 Frontend-Framework-Evaluation

Die Wahl des Frontend-Frameworks bestimmt maßgeblich die Entwicklungseffizienz, Performance und Wartbarkeit der Anwendung. Die Evaluation vergleicht vier fundamental unterschiedliche Ansätze: ein Full-Stack-Framework mit Server-Side Rendering, eine klassische SPA-Architektur, einen Multi-Page-Application-Ansatz mit minimaler JavaScript-Abhängigkeit sowie ein alternatives Ökosystem.

#### Evaluierte Alternativen

**Next.js 14 (App Router):** Full-Stack React-Framework mit Server-Side Rendering, Static Site Generation und dem App Router mit React Server Components. Optimiert für PaaS-Deployment auf Vercel, aber auch self-hostbar [71].

**Vite + React (SPA):** Klassische Single-Page-Application-Architektur mit modernem Build-Tool. Framework-agnostisch und auf beliebiger Infrastruktur deploybar – von einem einfachen nginx-Server bis zu Content Delivery Network (CDN)-Hosting [22].

**Astro + React Islands:** Multi-Page-Application-Framework mit Island Architecture. Generiert primär statisches HTML und hydriert nur interaktive Komponenten („Islands"). Minimaler JavaScript-Footprint bei voller React-Kompatibilität [5].

**Angular 17:** Googles opinioniertes Full-Stack-Framework mit eigenem Ökosystem. Bietet SSR via Angular Universal, strikte Architekturvorgaben und umfassende Enterprise-Features [31].

#### Kriteriendefinition und Gewichtung

Die Kriterien und deren Gewichtung ergeben sich aus den projektspezifischen Anforderungen:

- **SSR/SEO-Fähigkeit (20%):** Für eine öffentlich zugängliche Companion App ist Suchmaschinenoptimierung relevant, um organischen Traffic zu generieren.

- **Developer Experience (20%):** Die Entwicklungseffizienz ist bei begrenzter Projektlaufzeit kritisch. Hierzu zählen Hot Module Replacement, Debugging-Tools und Dokumentationsqualität.
- **Ecosystem & Bibliotheken (15%):** Verfügbarkeit von Bibliotheken für Graph-Visualisierung (React Flow, D3.js) und Charting.
- **Performance (15%):** Ladezeiten, Bundle-Größe und Runtime-Performance beeinflussen die User Experience direkt.
- **TypeScript-Integration (10%):** Durchgängige Typsicherheit reduziert Fehler und verbessert die Wartbarkeit.
- **Hosting-Flexibilität (10%):** Unabhängigkeit von spezifischen Hosting-Plattformen; Möglichkeit zum Self-Hosting.
- **Lernkurve (10%):** Einarbeitungsaufwand bei vorhandenen JavaScript/TypeScript-Kenntnissen.

### Entscheidungsmatrix Frontend-Framework

Tabelle 4.5 zeigt die Bewertung der Alternativen.

Tabelle 4.5: MCDA-Entscheidungsmatrix: Frontend-Framework

| Kriterium | Gew. | Next.js | Vite+React | Astro | Angular | Max |
|---|---|---|---|---|---|---|
| SSR/SEO-Fähigkeit | 20% | 5 | 2 | 5 | 4 | 5 |
| Developer Experience | 20% | 5 | 5 | 4 | 3 | 5 |
| Ecosystem & Bibliotheken | 15% | 5 | 5 | 4 | 3 | 5 |
| Performance | 15% | 4 | 4 | 5 | 3 | 5 |
| TypeScript-Integration | 10% | 5 | 4 | 4 | 5 | 5 |
| Hosting-Flexibilität | 10% | 3 | 5 | 5 | 4 | 5 |
| Lernkurve | 10% | 4 | 5 | 4 | 2 | 5 |
| **Gewichtete Summe** | | **4,50** | **4,05** | **4,35** | **3,30** | 5,00 |
| **Rang** | | **1** | 3 | 2 | 4 | |

### Begründung der Bewertungen

**Next.js 14** erreicht die höchste Gesamtpunktzahl (4,50) und wird als Frontend-Framework ausgewählt. Die Stärken liegen in der exzellenten SSR-Unterstützung durch den App Router, dem umfangreichen React-Ökosystem mit direkter Unterstützung für React Flow und Recharts sowie der nahtlosen TypeScript-Integration. Die eingeschränkte Hosting-Flexibilität (Bewertung 3) reflektiert die Tatsache, dass Next.js zwar self-hostbar ist, jedoch ohne Vercel auf einige Optimierungen verzichtet werden muss.

**Astro** (4,35) erreicht den zweiten Rang und wäre eine valide Alternative. Die Island Architecture liefert exzellente Performance durch minimalen JavaScript-Footprint. Für eine datenintensive Companion App mit komplexen interaktiven Visualisierungen würde jedoch ein Großteil der Seite aus „Islands" bestehen, was den architektonischen Vorteil relativiert.

**Vite + React** (4,05) exzelliert bei Developer Experience und Hosting-Flexibilität – die resultierende SPA kann auf jedem statischen Webserver deployed werden. Die fehlende native SSR-Unterstützung ist jedoch für eine SEO-relevante Companion App ein signifikanter Nachteil, der zusätzliche Infrastruktur (z.B. Prerendering-Service) erfordern würde.

**Angular 17** (3,30) bietet ein vollständiges Enterprise-Framework mit strikten Architekturvorgaben. Die steile Lernkurve, das separate Ökosystem (keine direkte Nutzung von React-Bibliotheken) und die vergleichsweise schwergewichtige Runtime machen es für ein Einzelentwickler-Projekt weniger geeignet.

### 4.2.3 Backend- und Datenbank-Evaluation

Die Backend-Architektur muss die Speicherung von Spieldaten, Benutzerfortschritt und perspektivisch User-Generated Content unterstützen. Die Evaluation vergleicht vier fundamental unterschiedliche Ansätze: eine BaaS-Plattform, eine selbst gehostete Lösung auf einem Virtual Private Server (VPS), eine klassische IaaS-Architektur auf AWS sowie eine containerisierte Self-Hosting-Lösung.

#### Evaluierte Alternativen

**Supabase :** Open-Source Plattform mit PostgreSQL als Fundament, Realtime-Subscriptions, Row Level Security und Edge Functions. Managed Hosting mit großzügigem Free Tier [61].

**Self-Hosted PostgreSQL + Node.js API (VPS):** Klassische Drei-Schichten-Architektur auf einem Virtual Private Server (z.B. Hetzner, DigitalOcean). PostgreSQL-Datenbank, Express/Fastify-API und nginx als Reverse Proxy. Vollständige Kontrolle bei moderatem Administrationsaufwand.

**AWS (RDS + Lambda + API Gateway):** Enterprise-Grade IaaS-Lösung mit managed PostgreSQL (RDS), serverless Compute (Lambda) und API Gateway. Hochskalierbar mit Pay-per-Use-Modell, jedoch komplexe Konfiguration und potenziell hohe Kosten.

**Self-Hosted mit Coolify/Dokku (PaaS-on-VPS):** Selbst gehostete PaaS-Lösung auf eigenem Server. Coolify oder Dokku abstrahieren die Infrastruktur ähnlich wie Heroku, laufen aber auf eigener Hardware. Kombiniert Kontrolle mit reduziertem Ops-Aufwand.

### Kriteriendefinition und Gewichtung

- **Relationales Datenmodell (20%):** Die komplexen Beziehungen zwischen Quests, Items, Workstations und deren Abhängigkeiten erfordern ein relationales Schema mit Foreign Keys und Joins.
- **Kosteneffizienz (20%):** Als Hobby-Projekt ohne Monetarisierung ist ein niedriges Kostenmodell essentiell.
- **Administrationsaufwand (20%):** Zeit für Setup, Wartung, Backups und Security-Updates reduziert die verfügbare Entwicklungszeit.
- **Realtime-Fähigkeit (10%):** Für zukünftige Squad-Features ist Echtzeit-Synchronisation relevant.
- **TypeScript/ORM-Integration (10%):** Kompatibilität mit Type-Safe ORMs wie Drizzle oder Prisma.
- **Skalierbarkeit (10%):** Fähigkeit zur Bewältigung von Nutzerwachstum ohne Architekturänderung.
- **Kontrolle & Flexibilität (10%):** Anpassbarkeit, Zugriff auf Konfiguration, Unabhängigkeit von Anbieter-Entscheidungen.

### Entscheidungsmatrix Backend/Datenbank

Tabelle 4.6: MCDA-Entscheidungsmatrix: Backend und Datenbank

| Kriterium | Gew. | Supabase | VPS | AWS | Coolify | Max |
|---|---|---|---|---|---|---|
| Relationales Datenmodell | 20% | 5 | 5 | 5 | 5 | 5 |
| Kosteneffizienz | 20% | 5 | 4 | 2 | 4 | 5 |
| Administrationsaufwand | 20% | 5 | 2 | 3 | 4 | 5 |
| Realtime-Fähigkeit | 10% | 5 | 3 | 3 | 3 | 5 |
| TypeScript/ORM-Integration | 10% | 5 | 5 | 4 | 5 | 5 |
| Skalierbarkeit | 10% | 4 | 2 | 5 | 3 | 5 |
| Kontrolle & Flexibilität | 10% | 3 | 5 | 4 | 5 | 5 |
| **Gewichtete Summe** | | **4,60** | **3,50** | **3,50** | **4,10** | 5,00 |
| **Rang** | | **1** | 3 | 3 | 2 | |

### Begründung der Bewertungen

**Supabase** erreicht mit 4,60 die höchste Punktzahl und wird als Backend-Lösung ausgewählt. Die Kombination aus vollwertigem PostgreSQL, integrierter Realtime-Funktionalität und minimalem Administrationsaufwand ermöglicht die Fokussierung auf die Frontend-Entwicklung.

Das großzügige Free Tier (500 MB Datenbank, 1 GB Dateispeicher, 50.000 monatliche API-Requests) deckt den Bedarf eines Hobby-Projekts ab. Die reduzierte Bewertung bei Kontrolle (3) reflektiert die Abhängigkeit von Supabase-spezifischen Features wie Row Level Security Policies.

**Coolify/Dokku** (4,10) erreicht den zweiten Rang und stellt eine attraktive Mittelweg-Lösung dar. Die selbst gehostete PaaS auf einem günstigen VPS (ab ca. 5€/Monat bei Hetzner) kombiniert die Einfachheit eines managed Service mit voller Datenkontrolle. Der initiale Setup-Aufwand ist jedoch höher als bei Supabase.

**Self-Hosted VPS** (3,50) bietet maximale Kontrolle und Flexibilität, erfordert jedoch erheblichen Administrationsaufwand für PostgreSQL-Konfiguration, Backups, SSL-Zertifikate, Firewall-Regeln und Security-Updates. Für einen Einzelentwickler mit Fokus auf Frontend-Features ist dieser Overhead signifikant.

**AWS** (3,50) bietet Enterprise-Grade-Infrastruktur mit exzellenter Skalierbarkeit, jedoch zu Lasten der Kosteneffizienz und Komplexität. Selbst bei minimaler Nutzung entstehen Kosten für RDS, Lambda, API Gateway, CloudWatch und Netzwerk-Traffic. Die Lernkurve für die AWS-Konsole und Identity and Access Management (IAM)-Konfiguration ist beträchtlich.

### 4.2.4 Deployment-Plattform-Evaluation

Die Deployment-Plattform muss eine reibungslose CI/CD-Integration, Preview Deployments für iterative Entwicklung und kosteneffizientes Hosting ermöglichen. Die Evaluation vergleicht eine spezialisierte PaaS-Lösung, eine klassische IaaS-Architektur auf AWS, ein Self-Hosting-Setup auf einem VPS sowie eine containerisierte Lösung.

#### Evaluierte Alternativen

**Vercel:** Framework-aware Deployment-Plattform mit nativer Next.js-Optimierung, Edge Functions, globalem CDN und automatischen Preview Deployments. Spezialisiert auf Frontend-Frameworks [76].

**AWS (S3 + CloudFront + EC2):** Klassische IaaS-Architektur mit S3 für statische Assets, CloudFront als CDN, EC2 für SSR-Funktionen und Route53 für DNS. Maximale Kontrolle bei hoher Komplexität.

**Self-Hosted VPS (nginx + PM2):** Traditionelles Deployment auf einem Virtual Private Server mit nginx als Reverse Proxy und PM2 als Node.js Process Manager. Günstig und vollständig kontrollierbar.

**Docker + Kubernetes (Self-Managed):** Container-basiertes Deployment mit Docker-Images und Kubernetes-Orchestrierung (z.B. k3s auf eigenem Server oder managed K8s). Maximale Portabilität und Skalierbarkeit.

### Kriteriendefinition und Gewichtung

- **Next.js-Kompatibilität (25%):** Unterstützung für App Router, Server Components, ISR und Edge Runtime.
- **Kosteneffizienz (25%):** Gesamtkosten für Hosting eines Hobby-Projekts mit moderatem Traffic.
- **Administrationsaufwand (20%):** Zeit für Setup, Wartung und Troubleshooting.
- **CI/CD & Preview Deployments (15%):** Automatisierte Builds und Branch-basierte Preview-Umgebungen.
- **Kontrolle & Portabilität (15%):** Unabhängigkeit von Plattform-Lock-In, Möglichkeit zur Migration.

### Entscheidungsmatrix Deployment

Tabelle 4.7: MCDA-Entscheidungsmatrix: Deployment-Plattform

| Kriterium | Gew. | Vercel | AWS | VPS | K8s | Max |
|---|---|---|---|---|---|---|
| Next.js-Kompatibilität | 25% | 5 | 3 | 3 | 4 | 5 |
| Kosteneffizienz | 25% | 4 | 2 | 5 | 3 | 5 |
| Administrationsaufwand | 20% | 5 | 2 | 2 | 1 | 5 |
| CI/CD & Preview | 15% | 5 | 3 | 2 | 4 | 5 |
| Kontrolle & Portabilität | 15% | 2 | 4 | 5 | 5 | 5 |
| **Gewichtete Summe** | | **4,25** | **2,75** | **3,45** | **3,30** | 5,00 |
| **Rang** | | **1** | 4 | 2 | 3 | |

### Begründung der Bewertungen

**Vercel** erreicht mit 4,25 die höchste Punktzahl und wird als Deployment-Plattform ausgewählt. Als Entwickler von Next.js bietet Vercel die beste Framework-Integration mit automatischer Erkennung und Optimierung aller Next.js-Features. Die Preview-Deployment-Funktionalität unterstützt den agilen Entwicklungsprozess durch automatische Deployments für jeden Pull Request. Die niedrige Bewertung bei Kontrolle (2) reflektiert das Vendor Lock-In: Eine Migration weg von Vercel würde den Verlust von Optimierungen wie Edge Functions und ISR bedeuten.

**Self-Hosted VPS** (3,45) erreicht den zweiten Rang und wäre eine kosteneffiziente Alternative. Ein Hetzner Cloud Server (CX11, ca. 4€/Monat) kann eine Next.js-Anwendung mit moderatem Traffic problemlos hosten. Der manuelle Setup von nginx, SSL-Zertifikaten (Let's Encrypt), CI/CD (GitHub Actions + SSH Deploy) und Monitoring erfordert jedoch Initial- und Wartungsaufwand.

**Docker + Kubernetes** (3,30) bietet maximale Portabilität – das Docker-Image kann auf beliebiger Infrastruktur deployed werden. Für ein Einzelentwickler-Projekt ist der Overhead von Kubernetes (selbst k3s) jedoch unverhältnismäßig. Die Komplexität von Ingress-Konfiguration, Persistent Volumes und Cluster-Maintenance übersteigt den Nutzen.

**AWS** (2,75) bietet Enterprise-Grade-Features, jedoch mit erheblicher Komplexität und Kosten. Die manuelle Konfiguration von S3-Buckets, CloudFront-Distributions, EC2-Instanzen, Load Balancern und IAM-Policies erfordert signifikantes AWS-Expertise. Für Next.js-SSR müssten zusätzlich Lambda@Edge oder EC2-Instanzen konfiguriert werden.

### 4.2.5 Resultierender Technologie-Stack

Die MCDA-Evaluation identifiziert folgenden optimalen Technologie-Stack für die Arc Raiders Companion App:

Tabelle 4.8: Resultierender Technologie-Stack

| Kategorie | Technologie | Score | Kernargument |
|---|---|---|---|
| Frontend-Framework | Next.js 14 (App Router) | 4,50 | SSR + React-Ecosystem |
| Backend/Datenbank | Supabase + PostgreSQL | 4,60 | Minimaler Ops-Aufwand + Realtime |
| Deployment | Vercel | 4,25 | Native Next.js-Optimierung |
| ORM | Drizzle | – | TypeScript-native, performant |
| State Management | Zustand | – | Minimalistisch, localStorage-Sync |
| UI-Framework | Radix UI + Tailwind CSS | – | Accessible + Utility-First |
| Visualisierung | React Flow + Recharts | – | Graph-Visualisierung + Charts |

Die Entscheidung für den „Managed Stack“ (Next.js + Supabase + Vercel) priorisiert Entwicklungseffizienz und minimalen Administrationsaufwand gegenüber maximaler Kontrolle. Diese Priorisierung ist projektspezifisch begründet: Als Einzelentwickler-Projekt mit begrenzter Laufzeit und Fokus auf Frontend-Features überwiegt der Nutzen reduzierter Infrastruktur-Komplexität die Nachteile des Vendor Lock-In.

Für Projekte mit anderen Rahmenbedingungen – etwa einem dedizierten DevOps-Team, höheren Anforderungen an Datensouveränität oder langfristiger Kostensensitivität – könnte Beispielsweise die Kombination aus **Astro + Self-Hosted PostgreSQL + VPS** eine valide Alternative darstellen.

### 4.2.6 Kritische Würdigung der Methodik, in Evaluation und Ergebnisse verschieben?

Die angewandte MCDA-Methodik unterliegt inhärenten Limitierungen. Die Gewichtung der Kriterien erfolgt subjektiv und projektspezifisch; andere Projekte könnten bei abweichenden Anforderungen zu unterschiedlichen Ergebnissen gelangen. Insbesondere die hohe Gewichtung von „Administrationsaufwand“ (20% bei Backend, 20% bei Deployment) reflektiert die Einzelentwickler-Situation und würde in einem Team-Kontext anders ausfallen.

Die qualitativen Bewertungen basieren auf der Evaluation zum Zeitpunkt der Entscheidungsfindung (Oktober 2025) und können sich durch Technologie-Updates ändern. Beispielsweise könnte eine zukünftige Verbesserung der Self-Hosting-Dokumentation von Next.js die Hosting-Flexibilität-Bewertung erhöhen.

Dennoch bietet die strukturierte Evaluation gegenüber einer Ad-hoc-Entscheidung mehrere Vorteile: Die explizite Kriteriendefinition zwingt zur Reflexion der tatsächlichen Anforderungen, die dokumentierte Bewertung ermöglicht Nachvollziehbarkeit, und die Gegenüberstellung unterschiedlicher Architekturansätze (Managed vs. Self-Hosted vs. IaaS) macht Trade-offs explizit.

## 4.3 Systemarchitektur und -design

Dieses Kapitel beschreibt die Architektur der Arc Raiders Companion App aus verschiedenen Perspektiven. Ausgehend von einer Kontextabgrenzung auf Systemebene wird das Datenbankdesign erläutert, bevor die Komponentenarchitektur und der Datenfluss am Beispiel des Quest-Management-Systems detailliert werden.

### 4.3.1 Systemkontext und externe Schnittstellen

Das Kontextdiagramm in Abbildung 4.3 visualisiert die Systemgrenzen der ArcDéx-Anwendung sowie deren Interaktion mit externen Akteuren und Diensten. Die Darstellung folgt dem arc42-Template, das eine standardisierte Dokumentation von Softwarearchitekturen ermöglicht [55].

#### Externe Akteure und Systeme

**Player (Primärer Akteur):** Der Spieler interagiert über einen Webbrowser via HTTPS mit der Anwendung. Die Kommunikation erfolgt über das globale CDN von Vercel, das statische Assets cached und dynamische Anfragen an Serverless Functions weiterleitet.

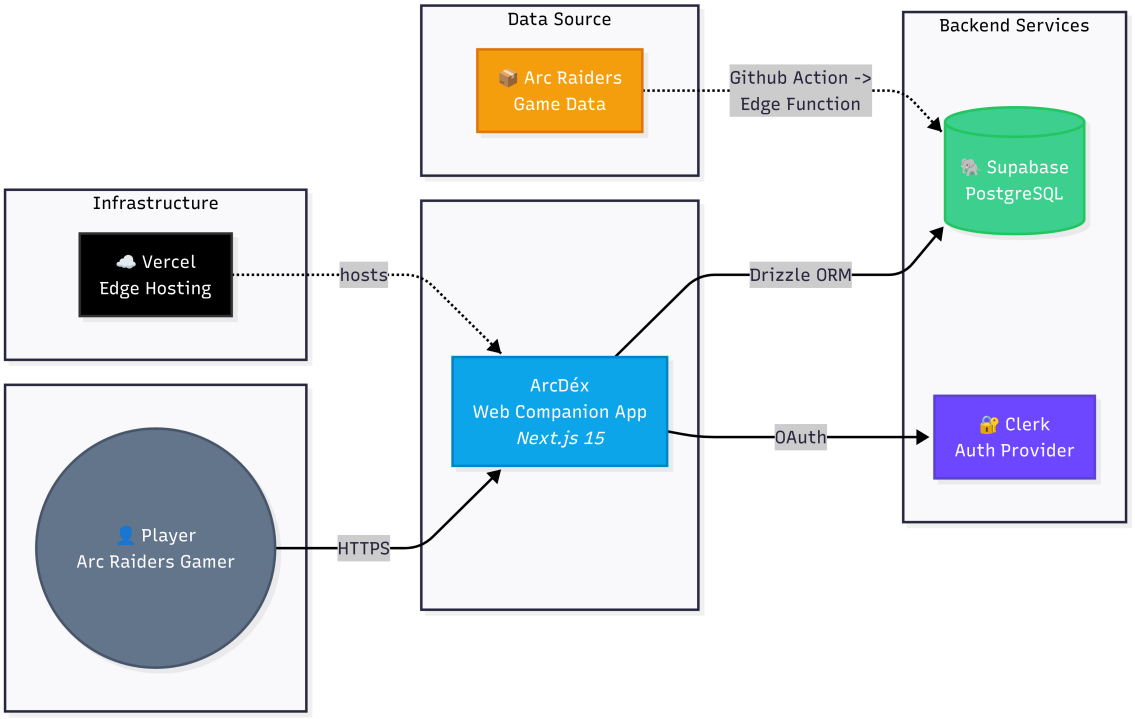


Abbildung 4.3: Kontextdiagramm der ArcDéx Companion App

**Vercel (Hosting-Infrastruktur):** Die Next.js-Anwendung wird auf Vercels Edge-Netzwerk gehostet. Vercel übernimmt das automatische Deployment, SSL-Terminierung, CDN-Distribution und die Ausführung von Serverless Functions für SSR und Server Actions.

**Supabase (Backend-as-a-Service):** Die PostgreSQL-Datenbank wird von Supabase gehostet und verwaltet. Die Kommunikation erfolgt über Drizzle ORM, das type-safe Datenbankabfragen in TypeScript ermöglicht. Supabase stellt zusätzlich Realtime-Subscriptions für zukünftige Squad-Features bereit.

**Clerk (Authentication Provider):** Die Benutzerauthentifizierung wird an Clerk ausgelagert, einen spezialisierten Auth-Provider mit OAuth-Integration. Dies externalisiert die sicherheitskritische Authentifizierungslogik und ermöglicht Social Login via Discord, Google und anderen Providern.

**Game Data (Datenquelle):** Die Spieldaten von Arc Raiders werden über einen automatisierten Pipeline-Prozess synchronisiert. Eine GitHub Action triggert periodisch eine Supabase Edge Function, die aktualisierte Spieldaten aus der Community-Datenquelle importiert und in die PostgreSQL-Datenbank schreibt.

### Architektonische Entscheidungen auf Kontextebene

Die Architektur folgt dem „Managed Stack“-Ansatz, der in Abschnitt 4.2 evaluiert wurde. Die Externalisierung von Authentifizierung (Clerk), Datenhaltung (Supabase) und Hosting

(Vercel) reduziert den operativen Overhead und ermöglicht die Fokussierung auf die Anwendungslogik.

Die lose Kopplung zwischen den externen Diensten wird durch standardisierte Schnittstellen erreicht: OAuth 2.0 für Authentifizierung, REST/PostgreSQL-Protokoll für Datenbankzugriffe und HTTPS für alle Client-Server-Kommunikation.

### 4.3.2 Datenbankdesign

Das relationale Datenbankschema bildet die Domänenobjekte des Spiels Arc Raiders ab und unterstützt die in Abschnitt 4.1 definierten funktionalen Anforderungen. Abbildung 4.4 zeigt das ER-Diagramm der PostgreSQL-Datenbank.

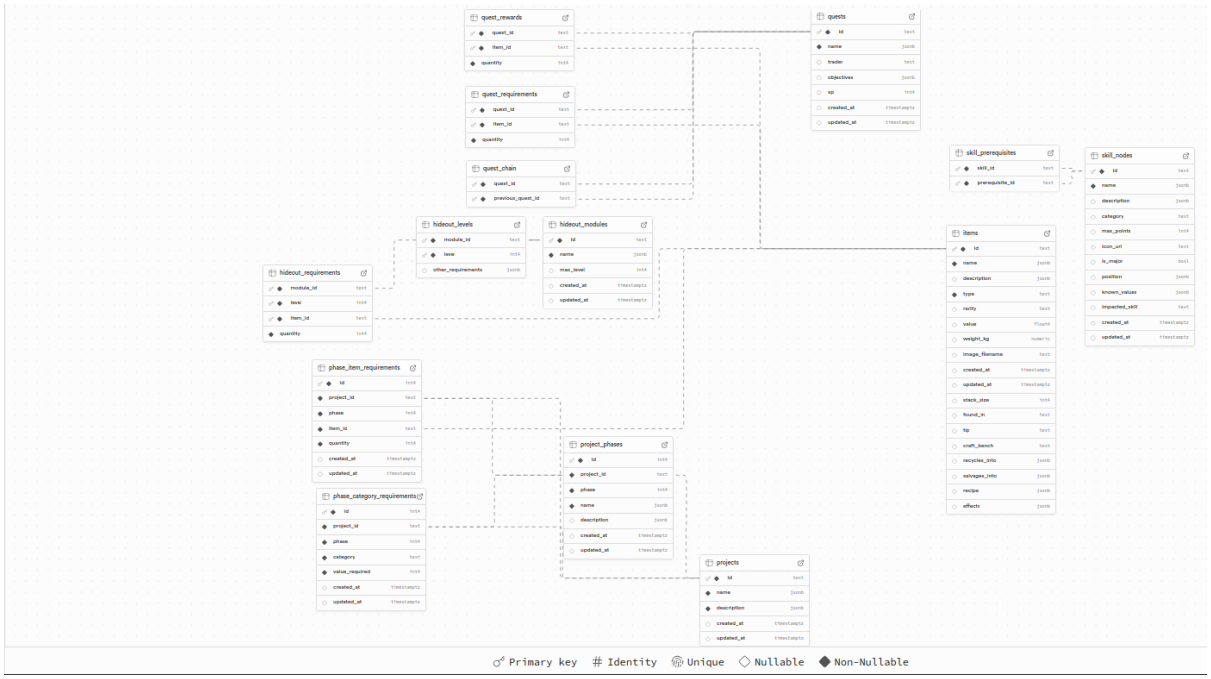


Abbildung 4.4: Entity-Relationship-Diagramm der ArcDéx-Datenbank

#### Domänenmodell und Entitäten

Das Schema gliedert sich in vier funktionale Bereiche, die den Feature-Bereichen der Anwendung entsprechen:

**Quest-Management:** Die Entität `quests` speichert die Grunddaten einer Quest (Name, Händler, Ziele, Erfahrungspunkte). Quest-Abhängigkeiten werden über die Assoziationstabelle `quest_chain` als gerichteter Graph modelliert, wobei `previous_quest_id` auf die Voraussetzungs-Quest verweist. Die Tabellen `quest_requirements` und `quest_rewards` verknüpfen Quests mit den benötigten bzw. erhaltenen Items.

**Item-Datenbank:** Die zentrale Entität `items` enthält alle Spielgegenstände mit Attributen wie Typ, Seltenheit, Wert und Gewicht. Komplexe, variable Datenstrukturen wie Effekte (`effects`) werden als `JSONB`-Spalten gespeichert, was flexible Schemata innerhalb des relationalen Modells ermöglicht.

**Hideout-System:** Das Workstation-Upgrade-System wird durch drei Tabellen abgebildet: `hideout_modules` definiert die verfügbaren Workstations mit maximalem Level, `hideout_levels` spezifiziert die einzelnen Upgrade-Stufen mit sonstigen Voraussetzungen, und `hideout_requirements` verknüpft jedes Level mit den benötigten Items.

**Skill-System:** Die Entität `skill_nodes` repräsentiert die Fähigkeiten im Skill-Baum mit Attributen wie Kategorie, maximale Punkte und Position im Baum. Die Tabelle `skill_prerequisites` modelliert die Abhängigkeiten zwischen Skills als gerichteten azyklischen Graphen.

### Mehrsprachigkeit durch JSONB

Ein zentrales Designmerkmal ist die Unterstützung von Mehrsprachigkeit ohne zusätzliche Übersetzungstabellen. Textfelder wie `name` und `description` werden als `JSONB`-Objekte gespeichert:

```
{
  "de": "Kameraobjektiv",
  "en": "Camera Lens",
  "hr": "Objektiv Kamere",
  "it": "Obiettivo fotografico",
}
```

Quelltext 4.1: Mehrsprachige Textspeicherung in JSONB

Dieser Ansatz ermöglicht die dynamische Erweiterung um zusätzliche Sprachen ohne Schemaänderung und effiziente Abfragen über PostgreSQLs native JSONB-Operatoren. Die Anwendung extrahiert zur Laufzeit den Text für die aktuelle Benutzersprache.

### Graph-Strukturen für Abhängigkeiten

Sowohl Quest-Ketten (`quest_chain`) als auch Skill-Voraussetzungen (`skill_prerequisites`) werden als Adjazenzlisten modelliert. Diese Struktur ermöglicht:

- Effiziente Abfragen direkter Vorgänger/Nachfolger via Foreign Key Join
- Rekursive Pfadberechnung mit PostgreSQLs `WITH RECURSIVE` Common Table Expression (CTE)
- Zykluserkennung zur Sicherstellung der Datenintegrität

Die Wahl der Adjazenzliste gegenüber alternativen Ansätzen (Nested Sets, Materialized Paths) basiert auf dem Anwendungsprofil: Die Graphen sind relativ flach (typisch 3–5 Ebenen), Schreiboperationen sind selten (nur bei Datenimport), und die primären Lesezugriffe benötigen direkte Nachbarn, nicht vollständige Teilbäume.

### 4.3.3 Komponentenarchitektur

Die Komponentenarchitektur folgt dem Schichtenmodell moderner React-Anwendungen mit klarer Trennung von Präsentation, Zustandsverwaltung und Datenzugriff. Abbildung 4.5 illustriert die Architektur am Beispiel der Quest-Seite.

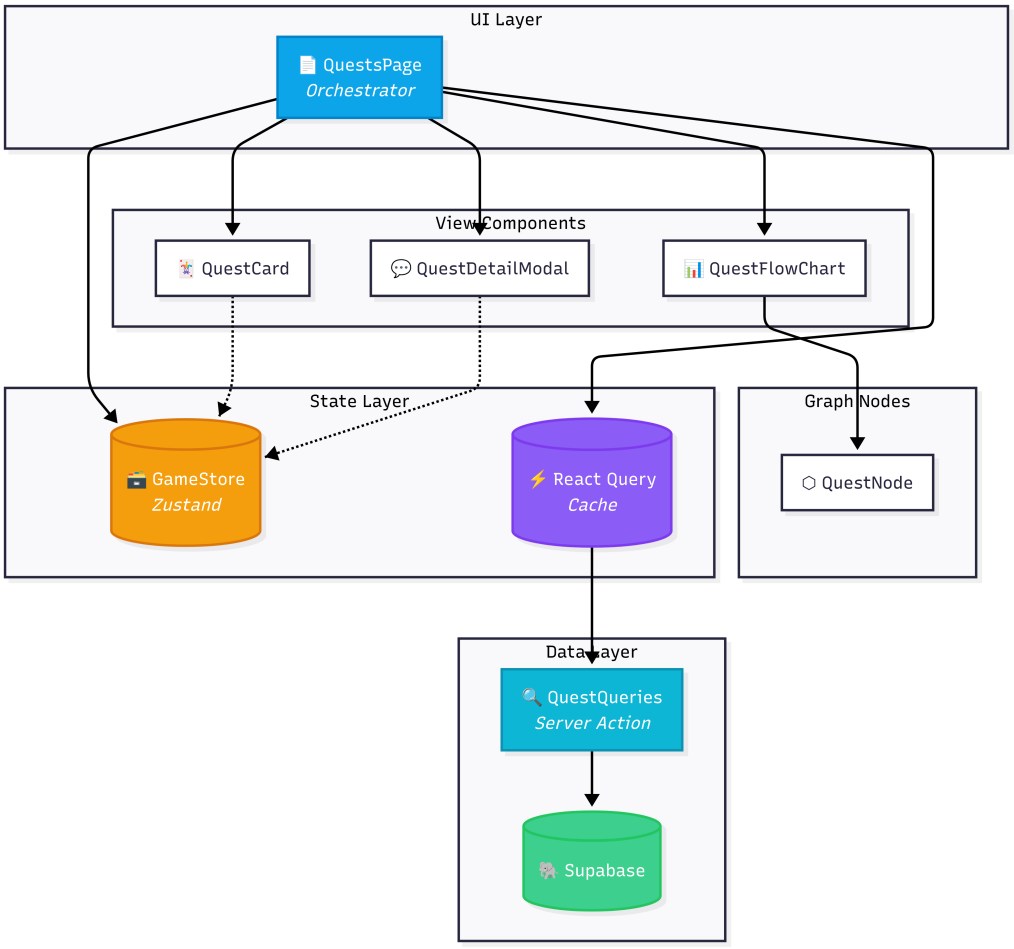


Abbildung 4.5: Komponentendiagramm der Quest-Seite

#### UI Layer

Die `QuestsPage`-Komponente fungiert als Orchestrator und koordiniert die untergeordneten View-Komponenten. Sie ist verantwortlich für das Layout, die Initialisierung des Datenabrufs und die Weiterleitung von Benutzerinteraktionen an die State-Schicht.

### View Components

Drei spezialisierte View-Komponenten realisieren die unterschiedlichen Darstellungsformen der Quest-Daten:

**QuestCard:** Kartenbasierte Darstellung einer einzelnen Quest mit Status-Indikator, Händler-Badge und Kurzinformationen. Die Komponente liest den Completion-Status aus dem Game-Store und visualisiert ihn durch Farbcodierung.

**QuestFlowChart:** Interaktive Graph-Visualisierung der Quest-Abhängigkeiten basierend auf React Flow. Die Komponente transformiert die Quest-Chain-Daten in Nodes und Edges und wendet den Dagre-Layout-Algorithmus für automatische Positionierung an. Jede Quest wird durch eine `QuestNode`-Komponente gerendert.

**QuestDetailModal:** Modale Detailansicht mit vollständigen Quest-Informationen, Requirements, Rewards und Navigation zu verknüpften Quests. Die Komponente ermöglicht das Markieren einer Quest als abgeschlossen.

### State Layer

Die Zustandsverwaltung implementiert die in Abschnitt 3.4.2 beschriebene Client-Server-State-Dichotomie:

**GameStore (Zustand):** Verwaltet den Client State, insbesondere die vom Benutzer markierten abgeschlossenen Quests (`completedQuests`). Der Store persistiert seine Daten automatisch im `localStorage` des Browsers, sodass der Fortschritt über Sessions hinweg erhalten bleibt.

**React Query Cache:** Verwaltet den Server State, d.h. die von der Datenbank geladenen Quest-Daten. Der Cache implementiert Stale-While-Revalidate-Semantik: Gecachte Daten werden sofort angezeigt, während im Hintergrund eine Aktualisierung geprüft wird.

### Data Layer

**QuestQueries (Server Actions):** Next.js Server Actions kapseln die Datenbankabfragen und werden auf dem Server ausgeführt. Sie nutzen Drizzle ORM für type-safe Queries und geben typisierte Ergebnisse an den Client zurück. Die Server Actions sind nicht direkt von Client-Komponenten aufrufbar, sondern werden über React Query orchestriert.

**Supabase (PostgreSQL):** Die Datenbank liefert die persistenten Daten. Der Zugriff erfolgt ausschließlich über die Server Actions, nie direkt vom Client, was die Sicherheit durch Kapselung gewährleistet.

### 4.3.4 Datenfluss und State Management

Der Datenfluss in der Anwendung folgt einem unidirektionalen Muster, das die Nachvollziehbarkeit von Zustandsänderungen gewährleistet. Abbildung 4.6 zeigt den vereinfachten Datenfluss zwischen Client, Edge und Datenbank.

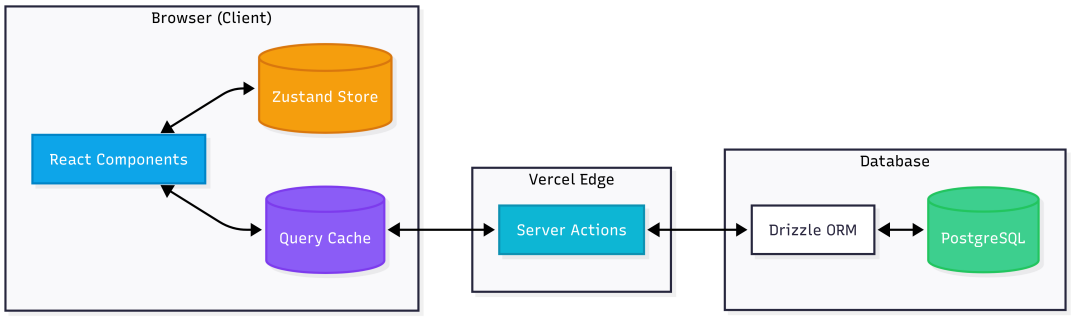


Abbildung 4.6: Vereinfachtes Datenflussdiagramm

#### Client-seitiger Datenfluss

React-Komponenten interagieren mit zwei Datenquellen: dem Zustand Store für lokale Benutzerdaten und dem Query Cache für Serverdaten. Die Komponenten sind „dumme" Präsentationskomponenten, die ihren Zustand nicht selbst verwalten, sondern aus den Stores beziehen. Zustandsänderungen erfolgen ausschließlich über definierte Actions, was das Debugging durch zentrale Logging-Punkte vereinfacht.

#### Server-seitiger Datenfluss

Datenbankabfragen werden durch React Query initiiert, das Server Actions auf Vercels Edge-Infrastruktur aufruft. Die Server Actions nutzen Drizzle ORM für typsichere Queries gegen die Supabase-PostgreSQL-Datenbank. Die Ergebnisse werden serialisiert, an den Client übertragen und im Query Cache gespeichert.

#### State-Derivation am Beispiel Quest-Status

Abbildung 4.7 illustriert, wie der Quest-Status aus der Kombination von Server State (Quest-Daten) und Client State (Completion-Markierungen) abgeleitet wird.

Der Quest-Status (`active`, `locked`, `completed`) ist ein abgeleiteter Zustand (Derived State), der aus zwei Quellen berechnet wird:

1. **Quest-Daten (React Query):** Enthält die Quest-Chain-Informationen, d.h. welche Quests Voraussetzung für andere sind.

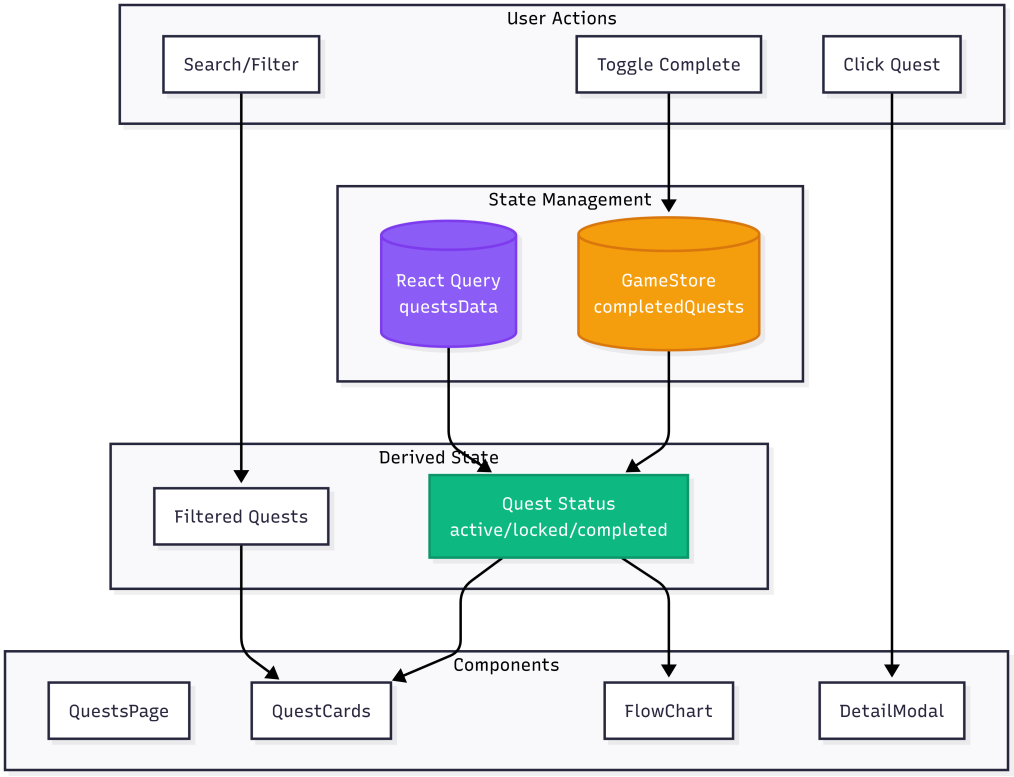


Abbildung 4.7: State-Flow-Diagramm der Quest-Seite

2. **Completed Quests (Zustand Store):** Enthält die IDs der vom Benutzer als abgeschlossen markierten Quests.

Die Statusberechnung erfolgt bei jedem Render durch eine Selektorfunktion:

```
const getQuestStatus = (quest, completedQuests, questChain) => {
  if (completedQuests.includes(quest.id)) return 'completed';

  const prerequisites = questChain
    .filter(chain => chain.questId === quest.id)
    .map(chain => chain.previousQuestId);

  const allPrerequisitesMet = prerequisites
    .every(preId => completedQuests.includes(preId));

  return allPrerequisitesMet ? 'active' : 'locked';
};
```

Quelltext 4.2: Quest-Status-Berechnung als Derived State

Dieser Ansatz vermeidet redundante Datenhaltung: Der Status wird nicht gespeichert, sondern zur Laufzeit berechnet. Änderungen am `completedQuests`-Array propagieren automatisch zu allen abhängigen Komponenten durch Reacts Reaktivitätssystem.

### 4.3.5 Zusammenfassung der Architekturentscheidungen

Die Systemarchitektur der ArcDéx Companion App basiert auf folgenden zentralen Entscheidungen:

Tabelle 4.9: Zusammenfassung der Architekturentscheidungen

| Aspekt | Entscheidung | Begründung |
|---|---|---|
| Systemgrenzen | Managed Services für Auth, DB, Hosting | Reduzierter Ops-Aufwand für Einzelentwickler |
| Datenmodell | Relationales Schema mit JSONB für flexible Strukturen | Starke Konsistenz + Flexibilität für Mehrsprachigkeit |
| Graph-Strukturen | Adjazenzlisten für Quest-Chains und Skill-Trees | Optimiert für flache Graphen mit seltenen Schreiboperationen |
| State Management | Client/Server-State-Trennung (Zustand + React Query) | Klare Verantwortlichkeiten, optimierte Caching-Strategien |
| Derived State | Berechnung zur Laufzeit statt Speicherung | Vermeidung von Inkonsistenzen, Single Source of Truth |

Die Architektur priorisiert Wartbarkeit und Entwicklungseffizienz über maximale Performance. Für eine Companion App mit primär lesenden Zugriffen und moderatem Traffic ist dieser Trade-off angemessen. Bei signifikantem Nutzerwachstum könnten Optimierungen wie Denormalisierung häufig abgefragter Daten oder serverseitige Status-Berechnung evaluiert werden.

## 4.4 Evaluation und Testdurchführung

Dieses Kapitel beschreibt die Durchführung der Evaluationsphase gemäß dem in Abschnitt 2.2 definierten Phasenmodell. Die Phase 4 gliedert sich in eine quantitative Evaluation der funktionalen Anforderungen und Performance-Metriken sowie eine qualitative Evaluation der User Experience. Die Ergebnisse dieser Evaluation werden in Kapitel 5 präsentiert und diskutiert.

### 4.4.1 Quantitative Evaluation

Die quantitative Evaluation verfolgt zwei Ziele: die Validierung der in Abschnitt 4.1.3 definierten Akzeptanzkriterien sowie die Messung von Performance-Metriken unter realistischen Bedingungen.

### Methodische Differenzierung der Testansätze

Nicht alle Akzeptanzkriterien erfordern automatisierte Tests. Die Evaluation unterscheidet daher zwischen zwei Kategorien:

**Automatisierte E2E-Tests** werden für Akzeptanzkriterien mit expliziten Performance-Vorgaben eingesetzt. Diese Kriterien enthalten messbare Zeitangaben (z.B. „Latenz $<$ 100ms“) oder quantifizierbare Schwellwerte, deren Einhaltung nur durch instrumentierte Tests objektiv nachweisbar ist. Tabelle 4.10 zeigt die für automatisierte Tests selektierten Kriterien.

Tabelle 4.10: Akzeptanzkriterien mit automatisierten E2E-Tests

| ID | Metrik | Zielwert |
|---|---|---|
| AC-QM-01.2 | Swimlane-Toggle-Reaktionszeit | $<$ 200ms |
| AC-QM-01.3 | Filter-Latenz bei Texteingabe | $<$ 100ms |
| AC-QM-02.3 | Flow-Chart-Render-Zeit | $<$ 2000ms |
| AC-RC-02.2 | Rekursive Pfadberechnung | Tiefe $\geq$ 3, $<$ 2000ms |
| AC-MC-01.3 | Material-Aggregation | $<$ 500ms |
| – | TTI aller Hauptseiten | $<$ 3000ms |

**Manuelle Evaluation** erfolgt für funktionale Akzeptanzkriterien ohne Performance-Vorgaben. Diese Kriterien beschreiben Funktionalitäten, deren Erfüllung durch Inspektion der implementierten Features feststellbar ist (z.B. „Kanban-Board zeigt drei Spalten“). Die Bewertung erfolgt durch den Entwickler anhand einer dreistufigen Skala: *vollständig erfüllt*, *teilweise erfüllt* oder *nicht erfüllt*, jeweils mit Begründung.

Diese Differenzierung reduziert den Testaufwand auf die Kriterien, bei denen automatisierte Messung tatsächlichen Mehrwert liefert, ohne die Validierungsabdeckung zu kompromittieren.

### Testumgebung und -werkzeuge

Die automatisierten Tests werden mit Cypress durchgeführt, einem E2E-Testing-Framework für Webanwendungen. Cypress ermöglicht die Simulation realer Nutzerinteraktionen und bietet integrierte Möglichkeiten zur Performance-Messung über die Browser Performance API.

```
1 cy.get('[data-testid="quest-search"]')
2   .then(() => { startTime = performance.now(); })
3   .type(searchTerm);
4
5 cy.get('[data-testid="quest-card"]')
6   .should('contain', searchTerm)
7   .then(() => {
```

```
8       const duration = performance.now() - startTime;
9       expect(duration).to.be.lessThan(100);
10    });
```

Quelltext 4.3: Performance-Messung in Cypress

Die Tests werden gegen die Produktivumgebung auf Vercel ausgeführt, um realistische Netzwerkbedingungen und Edge-Function-Latenzen abzubilden. Jeder Test wird mehrfach ausgeführt, um Varianz durch Netzwerkschwankungen zu reduzieren.

#### Erfassungsschema für Akzeptanzkriterien

Die Gesamtheit der Akzeptanzkriterien wird in einer strukturierten Matrix erfasst (Tabelle 4.11). Für automatisiert getestete Kriterien werden die gemessenen Werte dokumentiert; für manuell evaluierte Kriterien erfolgt eine begründete Einordnung.

Tabelle 4.11: Erfassungsschema für Akzeptanzkriterien

| ID | Testmethode | Status | Evidenz/Begründung |
|---|---|---|---|
| AC-QM-01.1 | Manuell | ✓ / ∼ / × | [Beschreibung] |
| AC-QM-01.2 | E2E (Cypress) | ✓ / × | Gemessen: –ms |
| ... | ... | ... | ... |

### 4.4.2 Qualitative Evaluation

Die qualitative Evaluation erhebt die subjektive Nutzererfahrung und bewertet, inwieweit die in Abschnitt 1.2 identifizierten Probleme aus Nutzersicht gelöst wurden. Da die Anwendung zum Zeitpunkt der Evaluation nicht öffentlich verfügbar ist, entfallen nutzungsbasierte Metriken wie Daily Active Users (DAU). Stattdessen erfolgt die Evaluation durch einen strukturierten Online-Fragebogen.

#### Fragebogendesign

Der Fragebogen wurde als Google Form implementiert und gliedert sich in drei Teile, die unterschiedliche Dimensionen der User Experience erfassen:

**Teil B: System Usability Scale (SUS)** umfasst die zehn standardisierten Items nach Brooke [10]. Der System Usability Scale (SUS) ist ein etabliertes Instrument zur Messung der wahrgenommenen Usability und ermöglicht durch seinen standardisierten Score (0–100) den Vergleich mit Benchmark-Werten. Die Fragen werden auf einer fünfstufigen Likert-Skala beantwortet.

**Teil C: Problemlösungs-Bewertung** evaluiert direkt die in der Problemstellung identifizierten Kernprobleme sowie das im nachinein enstandene Recycling-Feature. Dieser Abschnitt ist in drei Dimensionen unterteilt:

1. Quest-Übersicht und -Planung (Fragen C1–C3)
2. Ressourcenmanagement (Fragen C4–C6)
3. Recycling-Verständnis (Fragen C7–C9)

Jede Dimension wird durch drei Fragen operationalisiert, die auf einer fünfstufigen Skala von „überhaupt nicht gelöst“ (1) bis „vollständig gelöst“ (5) beantwortet werden. Diese Struktur ermöglicht eine direkte Rückkopplung zur ursprünglichen Problemdefinition.

**Teil D: Feature-Bewertung** erfasst die Bewertung der zwölf implementierten Hauptfeatures auf einer fünfstufigen Skala. Zusätzlich steht die Option „nicht genutzt“ zur Verfügung, um Features zu identifizieren, die von Nutzern möglicherweise nicht entdeckt wurden.

### Stichprobe und Durchführung

Die Zielgruppe des Fragebogens sind Arc Raiders-Spieler, die die Anwendung vor dem Ausfüllen mindestens 30 Minuten genutzt haben. Diese Mindestnutzungsdauer stellt sicher, dass die Teilnehmer mit den Kernfeatures vertraut sind und eine fundierte Bewertung abgeben können.

Der Fragebogen wird asynchron über einen Zeitraum von zwei Wochen erhoben. Die Teilnehmer erhalten Zugang zur Anwendung sowie einen Link zum Fragebogen. Die Teilnahme ist freiwillig und anonym; es werden keine personenbezogenen Daten erfasst.

### Auswertungsmethodik

Die quantitativen Fragebogendaten werden deskriptiv ausgewertet:

- **SUS-Score:** Berechnung nach der standardisierten Formel (Summe der adjustierten Item-Scores multipliziert mit 2,5). Der resultierende Score wird anhand der Adjektivskala nach Bangor et al. [6] interpretiert.
- **Problemlösungs-Scores:** Berechnung des arithmetischen Mittels pro Problemdimension sowie über alle neun Items.
- **Feature-Scores:** Berechnung des arithmetischen Mittels pro Feature, Ranking nach Bewertung, Analyse der N/A-Raten zur Identifikation von Discovery-Problemen.

### 4.4.3 Traceability und Ergebniszuordnung

Die Evaluation schließt den Entwicklungszyklus: Problemstellung → User Stories → Akzeptanzkriterien → Implementierung → Test/Fragebogen → Ergebnisse. Diese Nachverfolgbarkeit ermöglicht in Kapitel 5 eine direkte Gegenüberstellung von technischer Erfüllung (AC) und wahrgenommener Problemlösung (UX-Score).

- Erfüllungsgrad der Akzeptanzkriterien (quantitativ, entwicklerseitig)
- Problemlösungs-Bewertung (qualitativ, nutzerseitig)

Diese duale Perspektive ermöglicht eine differenzierte Bewertung: Ein Feature kann technisch alle Akzeptanzkriterien erfüllen, aber dennoch aus Nutzersicht das zugrundeliegende Problem nur teilweise lösen – oder umgekehrt. Die kritische Reflexion dieser Diskrepanzen ist Gegenstand von Abschnitt 5.2.

# 5 Ergebnisse und Diskussion

Metriken definieren: Wie misst du Erfolg? (Performance, Usability, Code-Qualität)

## 5.1 Objektivierung der Ergebnisse

(Tabelle, Aufgabe, erledigt?, verifiziert?)

## 5.2 Kritische Reflektion

Stellungsnahme zu den Ergebnisse, aufzeigen von Alternativen

# 6 Fazit und Ausblick

Was kann man damit anfangen

Wie kann man es erweitern

# Literatur


[1] A. Abukhalaf et al. „Automated requirements engineering framework for agile model-driven development“. In: *Frontiers in Computer Science* 7 (2025). DOI: 10.3389/fcomp.2025.1537100.

[2] Ikeh Akinyemi. *Drizzle ORM adoption guide: Overview, examples, and alternatives*. Comprehensive technical analysis of Drizzle ORM. 2024. URL: https://blog.logrocket.com/drizzle-orm-adoption-guide/ (besucht am 28. 12. 2024).

[3] Scott W. Ambler. *Agile Requirements Modeling: Strategies for Agile Teams*. Agile Modeling. 2023. URL: http://agilemodeling.com/essays/agileRequirements.htm (besucht am 27. 01. 2025).

[4] Kulsum Ansari. „Next.js App Router: Routing“. In: (2024). Published: April 19, 2024. URL: https://medium.com/@kulsumansari4/next-js-app-router-routing-8d795dbe324c (besucht am 18. 11. 2024).

[5] Astro Technology Company. *Astro Documentation - Build faster websites*. Island Architecture für optimale Performance. 2024. URL: https://docs.astro.build/ (besucht am 15. 10. 2025).

[6] Aaron Bangor, Philip Kortum und James Miller. „Determining What Individual SUS Scores Mean: Adding an Adjective Rating Scale“. In: *Journal of Usability Studies* 4.3 (2009), S. 114–123.

[7] Beck, Kent and Beedle, Mike and van Bennekum, Arie and Cockburn, Alistair and Cunningham, Ward and Fowler, Martin and others. *Manifesto for Agile Software Development*. 2001. URL: https://agilemanifesto.org/ (besucht am 01. 10. 2025).

[8] Valerie Belton und Theodor Stewart. *Multiple Criteria Decision Analysis: An Integrated Approach*. Springer Science & Business Media, 2002. ISBN: 978-0792375050.

[9] Rohit Bhatu. „Why Choose TypeScript in 2024: The Evolution of JavaScript Development“. In: (2024). Published: March 8, 2024. URL: https://medium.com/@bhaturohit/why-choose-typescript-in-2024-the-evolution-of-javascript-development-fdf77dd56a62 (besucht am 18. 11. 2024).

[10] John Brooke. „SUS: A ‘Quick and Dirty’ Usability Scale“. In: *Usability Evaluation in Industry* (1996). Standardisierte Methodik zur Usability-Messung, S. 189–194.

[11] Dai Clegg und Richard Barker. *Case Method Fast-Track: A RAD Approach*. Ursprung der MoSCoW-Priorisierungsmethode. Addison-Wesley, 1994. ISBN: 978-0201624328.

[12] J. Clement. *Number of gaming console and storefront companion app downloads worldwide from 2020 to 2024 YTD (in millions)*. Published: August 19, 2024. Statista. 2024. URL: https://www.statista.com/statistics/1266736/game-console-companion-app-downloads/ (besucht am 18. 11. 2024).

[13] Cloudflare. *What is BaaS? | Backend-as-a-Service vs. serverless*. Authoritative industry definition of Backend-as-a-Service concepts. 2024. URL: https://www.cloudflare.com/learning/serverless/glossary/backend-as-a-service-baas/ (besucht am 28. 12. 2024).

[14] Matthew Cochran. „Best Companion Apps In Video Games“. In: (2023).

[15] Codemancers. *Managing Server State in React Application using TanStack React Query*. Praktischer Guide zu Server State Management mit TanStack Query. 2024. URL: https://www.codemancers.com/blog/2024-managing-server-state-in-react-app-using-react-query (besucht am 18. 11. 2025).

[16] Hassan Djirdeh. „Mastering TypeScript: Benefits and Best Practices“. In: *Telerik Blog* (2024). Published: November 01, 2024. URL: https://www.telerik.com/blogs/mastering-typescript-benefits-best-practices (besucht am 18. 11. 2024).

[17] DORA Team. *DORA's software delivery metrics: the four keys*. Official DORA metrics documentation from Google. 2024. URL: https://dora.dev/guides/dora-metrics-four-keys/ (besucht am 28. 12. 2024).

[18] Drizzle Team. *Drizzle ORM - next gen TypeScript ORM*. Official Drizzle ORM documentation. 2024. URL: https://orm.drizzle.team/ (besucht am 28. 12. 2024).

[19] Filip Džebo. *Asynchronous State Management with TanStack Query*. Technische Erklärung von TanStack Query Caching-Mechanismen und Cache-Parametern. Aug. 2024. URL: https://www.atlantbh.com/asynchronous-state-management-with-tanstack-query/ (besucht am 18. 11. 2025).

[20] G. Ebirim et al. „Advancements and innovations in requirements elicitation“. In: *World Journal of Advanced Research and Reviews* 22.01 (2024), S. 1209–1220.

[21] Embark Studios. *ARC Raiders on Steam*. Valve Corporation. 2025. URL: https://store.steampowered.com/app/1808500/ARC_Raiders/ (besucht am 18. 11. 2024).

[22] Evan You. *Vite - Next Generation Frontend Tooling*. 2024. URL: https://vitejs.dev/ (besucht am 15. 10. 2025).

[23] Expedite Informatics. *TypeScript in 2024: Trends, Standards, Benefits, Challenges, and Commitments*. Expedite Informatics. 2024. URL: https://expediteinformatics.com/typescript-in-2024-trends-standards-benefits-challenges-and-commitments/ (besucht am 18. 11. 2024).

[24] Facebook Open Source. *Composition vs Inheritance*. React Documentation. 2024. URL: https://legacy.reactjs.org/docs/composition-vs-inheritance.html (besucht am 18. 11. 2024).

[25] Nicole Forsgren, Jez Humble und Gene Kim. *Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations*. Shingo Publication Award winner, establishes DORA metrics through rigorous research. IT Revolution Press, 2018. ISBN: 978-1942788331.

[26] Frontend Undefined. *A brief history of state management libraries for React*. Historische Analyse der Evolution von React State Management Libraries von Flux bis moderne Lösungen. 2024. URL: https://www.frontendundefined.com/posts/monthly/react-state-management-libraries-history/ (besucht am 18. 11. 2025).

[27] Brad Frost. *Atomic Design*. Brad Frost, 2016. ISBN: 978-0-9982966-0-9. URL: https://atomicdesign.bradfrost.com/ (besucht am 18. 11. 2024).

[28] GeeksforGeeks. *React Component Based Architecture*. Published: July 23, 2025. GeeksforGeeks. 2025. URL: https://www.geeksforgeeks.org/reactjs/react-component-based-architecture/ (besucht am 18. 11. 2024).

[29] Olga Giersza. „Companion Apps are Taking Video Games to the Next Level“. In: (2024).

[30] GitHub. *GitHub Actions: Automate your workflow*. Official GitHub Actions documentation for CI/CD automation. 2024. URL: https://github.com/features/actions (besucht am 28. 12. 2024).

[31] Google. *Angular Documentation*. 2024. URL: https://angular.dev/ (besucht am 15. 10. 2025).

[32] Austin Hodak. *The Hideout: Tarkov Sidekick - Mobile Companion App for Escape from Tarkov*. Google Play Store. 2024. URL: https://play.google.com/store/apps/details?id=com.austinhodak.thehideout (besucht am 18. 11. 2024).

[33] Jez Humble und David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Jolt Award winner, seminal work on continuous delivery practices. Addison-Wesley Professional, 2010. ISBN: 978-0321601919.

[34] IconEra Community. *What will the Extraction Shooter market look like in two years? (market size)*. Published: November 20, 2024. IconEra Forums. 2024. URL: https://icon-era.com/threads/what-will-the-extraction-shooter-market-look-like-in-two-years-market-size.14825/ (besucht am 18. 11. 2024).

[35] Insider Gaming. „Love Them or Hate Them, Extraction Shooters Are Running Wild Right Now". In: (2025). Published: 1 week ago from retrieval date. URL: https://insider-gaming.com/love-or-hate-extraction-shooters-wild/ (besucht am 18. 11. 2024).

[36] Invictarasolutions. *Why TypeScript is Becoming the Go-To for Large-Scale JavaScript Projects*. Published: October 27, 2024. Medium. 2024. URL: https://medium.com/@invictarasolutions/why-typescript-is-becoming-the-go-to-for-large-scale-javascript-projects-5439aabf4bb7 (besucht am 18. 11. 2024).

[37] Mohammad Khaled. „Headless vs. Traditional UI Libraries: When and Why to Choose Each". In: *DEV Community* (2025). Published: April 11, 2025. URL: https://dev.to/mohammad_kh4441/headless-vs-traditional-ui-libraries-when-and-why-to-choose-each-1c5b (besucht am 18. 11. 2024).

[38] Gene Kim et al. *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. Foundational DevOps reference establishing The Three Ways framework. IT Revolution Press, 2016. ISBN: 978-1942788003.

[39] Axel van Lamsweerde. „Requirements Engineering in the Year 00: A Research Perspective". In: *Proceedings of the 22nd International Conference on Software Engineering (ICSE'00)*. IEEE Press, 2000, S. 5–19. DOI: 10.1145/337180.337184.

[40] LogRocket. „Exploring the shift from CSS-in-JS to headless UI libraries". In: (2024). Published: June 4, 2024. URL: https://blog.logrocket.com/exploring-shift-css-in-js-headless-ui-libraries/ (besucht am 18. 11. 2024).

[41] Mohi Mishra. „Composition vs Inheritance in React: Simplifying Component Reusability". In: (2024). Published: April 2, 2024. URL: https://medium.com/@mohimishra016/composition-vs-inheritance-in-react-simplifying-component-reusability-f978363bacf6 (besucht am 18. 11. 2024).

[42] MUI. *Introducing MUI Base: the headless alternative to Material UI*. Published: September 7, 2022. MUI Blog. 2022. URL: https://mui.com/blog/introducing-base-ui/ (besucht am 18. 11. 2024).

[43] NehalemX. *Handbook for EFT - Escape from Tarkov Companion App*. Google Play Store. 2024. URL: https://play.google.com/store/apps/details?id=com.nehalemx.tarkovwiki2 (besucht am 18. 11. 2024).

[44] Oakcode. *Game Maps: Tarkov & More - Interactive Gaming Maps Desktop App*. Overwolf. 2024. URL: https://www.overwolf.com/app/W4rGo-Game_Maps_Escape_from_Tarkov (besucht am 18. 11. 2024).

[45] OpenReplay. „Why Developers Are Switching to shadcn/ui in React Projects“. In: (2024). URL: https://blog.openreplay.com/developers-switching-shadcn-ui-react/ (besucht am 18. 11. 2024).

[46] Thiraphat Phutson. „Mastering Next.js App Router: Best Practices for Structuring Your Application“. In: (2024). Published: September 21, 2024. URL: https://thiraphat-ps-dev.medium.com/mastering-next-js-app-router-best-practices-for-structuring-your-application-3f8cf0c76580 (besucht am 18. 11. 2024).

[47] Poimandres. *Zustand – Bear necessities for state management in React*. Offizielle Zustand GitHub Repository mit Dokumentation und Beispielen. 2024. URL: https://github.com/pmndrs/zustand (besucht am 18. 11. 2025).

[48] PostgreSQL Global Development Group. *PostgreSQL Documentation: Row Security Policies*. Official PostgreSQL documentation for Row Level Security. 2024. URL: https://www.postgresql.org/docs/current/ddl-rowsecurity.html (besucht am 28. 12. 2024).

[49] ProNextJS. *File-Based Routing with App Router*. ProNextJS Workshops. 2024. URL: https://www.pronextjs.dev/workshops/next-js-foundations-for-professional-web-development~lxb18/file-based-routing-with-app-router~dtnrx (besucht am 18. 11. 2024).

[50] Redux Team. *The History of Redux*. Offizielle Dokumentation zur Entstehungsgeschichte und Design-Entscheidungen von Redux. 2024. URL: https://redux.js.org/understanding/history-and-design/history-of-redux (besucht am 18. 11. 2025).

[51] Refine. *How to use Zustand*. Umfassender Guide zu Zustand State Management mit Best Practices. Juli 2024. URL: https://refine.dev/blog/zustand-react-state/ (besucht am 18. 11. 2025).

[52] Grand View Research. *Video Game Market Size, Share And Growth Report*. 2024.

[53] Nikhil Snayak. „Dissecting Partial Pre Rendering“. In: (2024). Published: July 6, 2024. URL: https://www.nikhilsnayak.dev/blogs/dissecting-partial-pre-rendering (besucht am 18. 11. 2024).

[54] Ritesh Srivastava. „Next JS: Rendering Strategies — SSG, SSR, CSR and PPR“. In: (2025). Published: February 10, 2025. URL: https://medium.com/@ritsrivastava/next-js-rendering-strategies-ssg-ssr-csr-and-ppr-d6243ec0ce72 (besucht am 18. 11. 2024).

[55] Gernot Starke und Peter Hruschka. *arc42 - Ressourcen für Software-Architekten*. Template für Softwarearchitektur-Dokumentation. 2024. URL: https://arc42.org/ (besucht am 20. 10. 2025).

[56] StudyRaid. *Composition vs Inheritance - React - The Complete Guide*. StudyRaid. 2024. URL: https://app.studyraid.com/en/read/1665/22475/composition-vs-inheritance (besucht am 18. 11. 2024).

[57] Supabase. *Edge Functions | Supabase Docs*. Official Supabase Edge Functions documentation. 2024. URL: https://supabase.com/docs/guides/functions (besucht am 28. 12. 2024).

[58] Supabase. *Realtime: Broadcast, Presence, and Postgres Changes via WebSockets*. Official Supabase Realtime server repository. 2024. URL: https://github.com/supabase/realtime (besucht am 28. 12. 2024).

[59] Supabase. *Row Level Security | Supabase Docs*. Official Supabase RLS integration documentation. 2024. URL: https://supabase.com/docs/guides/database/postgres/row-level-security (besucht am 28. 12. 2024).

[60] Supabase. *Supabase: The Postgres development platform*. Official Supabase repository with architecture documentation. 2024. URL: https://github.com/supabase/supabase (besucht am 28. 12. 2024).

[61] Supabase Inc. *Supabase - The Open Source Firebase Alternative*. 2024. URL: https://supabase.com/docs (besucht am 15. 10. 2025).

[62] Tailwind Labs. *Utility-First Fundamentals*. Tailwind CSS Documentation. 2024. URL: https://tailwindcss.com/docs/utility-first (besucht am 18. 11. 2024).

[63] TanStack. *Optimistic Updates | TanStack Query React Docs*. Offizielle Dokumentation zu Optimistic Update Patterns in TanStack Query. 2024. URL: https://tanstack.com/query/latest/docs/framework/react/guides/optimistic-updates (besucht am 18. 11. 2025).

[64] TanStack. *Overview | TanStack Query React Docs*. Offizielle Dokumentation zu TanStack Query Konzepten und Architektur. 2024. URL: https://tanstack.com/query/latest/docs/framework/react/overview (besucht am 18. 11. 2025).

[65] Tarkov.dev Community. *Tarkov.dev - A free, community made, and open source ecosystem of Escape from Tarkov tools and guides*. 2024. URL: https://tarkov.dev/ (besucht am 18. 11. 2024).

[66] Toxigon. *TypeScript Trends 2024: The Rise of Static Typing*. Published: March 21, 2025. Toxigon Blog. 2025. URL: https://toxigon.com/typescript-trend-2024 (besucht am 18. 11. 2024).

[67] TreeStn. *Tarkov Companion - Desktop App on Overwolf*. Overwolf. 2024. URL: https://www.overwolf.com/app/treestn-tarkov_companion (besucht am 18. 11. 2024).

[68] Emmanuel Udeji. „Mastering Next.js 15: A Guide to App Router Rendering Strategies for Modern Web Applications“. In: (2024). Published: November 4, 2024. URL: https://medium.com/@emmaudeji/mastering-next-js-15-a-guide-to-app-router-rendering-strategies-for-modern-web-applications-8a6683d8c348 (besucht am 18. 11. 2024).

[69] Vercel. *How to choose the best rendering strategy for your app*. Vercel Blog. 2024. URL: https://vercel.com/blog/how-to-choose-the-best-rendering-strategy-for-your-app (besucht am 18. 11. 2024).

[70] Vercel. *Next.js Docs: App Router*. Next.js Documentation. 2024. URL: https://nextjs.org/docs/app (besucht am 18. 11. 2024).

[71] Vercel. *Next.js Documentation*. 2024. URL: https://nextjs.org/docs (besucht am 15. 10. 2025).

[72] Vercel. *Next.js on Vercel*. Official Vercel documentation for Next.js deployment. 2024. URL: https://vercel.com/docs/frameworks/nextjs (besucht am 28. 12. 2024).

[73] Vercel. *Partial prerendering: Building towards a new default rendering model for web applications*. Vercel Blog. 2024. URL: https://vercel.com/blog/partial-prerendering-with-next-js-creating-a-new-default-rendering-model (besucht am 18. 11. 2024).

[74] Vercel. *Preview Deployments*. Official Vercel documentation on preview deployment features. 2024. URL: https://vercel.com/docs/deployments/preview-deployments (besucht am 28. 12. 2024).

[75] Vercel. *Routing: Defining Routes*. Next.js Documentation. 2024. URL: https://nextjs.org/docs/14/app/building-your-application/routing/defining-routes (besucht am 18. 11. 2024).

[76] Vercel Inc. *Vercel Documentation*. 2024. URL: https://vercel.com/docs (besucht am 15. 10. 2025).

[77] Visure Solutions. *Requirements Gathering Techniques in Agile Software Engineering*. 2025. URL: https://visuresolutions.com/alm-guide/requirements-gathering-techniques-for-agile (besucht am 27. 01. 2025).

[78] Jeet Vora. *Server State vs Client State in React for Beginners*. Einführung in die Unterscheidung zwischen Server State und Client State mit praktischen Beispielen. Juni 2023. URL: https://dev.to/jeetvora331/server-state-vs-client-state-in-react-for-beginners-3pl6 (besucht am 18. 11. 2025).

[79] Bill Wake. *INVEST in Good Stories, and SMART Tasks*. Ursprüngliche Definition des INVEST-Akronyms für User Stories. 2003. URL: https://xp123.com/articles/invest-in-good-stories-and-smart-tasks/ (besucht am 01. 12. 2025).

[80] Horst Werner und Simon Fishel. „Lessons Learned from Creating a Custom Graph Visualization in React“. In: *Medium - Splunk Engineering* (2022). Published: May 10, 2022. URL: https://medium.com/splunk-engineering/lessons-learned-from-creating-a-custom-graph-visualization-in-react-9a667ba799d1 (besucht am 18. 11. 2024).

[81] Wikipedia. *Object–relational mapping*. Comprehensive overview of ORM concepts and patterns. 2024. URL: https://en.wikipedia.org/wiki/Object-relational_mapping (besucht am 28. 12. 2024).

[82] Wikipedia contributors. *ARC Raiders*. Wikipedia, The Free Encyclopedia. Last edited: 3 hours ago from retrieval date. 2025. URL: https://en.wikipedia.org/wiki/ARC_Raiders (besucht am 18. 11. 2024).

[83] Wikipedia contributors. *Tailwind CSS*. Wikipedia, The Free Encyclopedia. Last edited: 4 weeks ago from retrieval date. 2025. URL: https://en.wikipedia.org/wiki/Tailwind_CSS (besucht am 18. 11. 2024).

[84] WorkOS. *What is the difference between Radix and shadcn-ui?* Published: February 20, 2025. WorkOS Blog. 2025. URL: https://workos.com/blog/what-is-the-difference-between-radix-and-shadcn-ui (besucht am 18. 11. 2024).

[85] xyflow. *Layouting - React Flow*. React Flow Documentation. 2024. URL: https://reactflow.dev/learn/layouting/layouting (besucht am 18. 11. 2024).