

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	v
Quelltextverzeichnis	vi
Abkürzungsverzeichnis	vii
Abstract	ix
Zusammenfassung	x
1 Einleitung	1
1.1 Motivation	1
1.2 Problemstellung	2
2 Aufgabenstellung und Entwicklungsmethodik	3
2.1 Operationalisierung der Problemstellung	3
2.2 Entwicklungsmethodik und Phasenmodell	4
2.2.1 Phase 1: Requirements Engineering	4
2.2.2 Phase 2: Architektur und Design	6
2.2.3 Phase 3: Implementierung in Iterationen	7
2.2.4 Phase 4: Evaluation und Reflexion	7
2.3 Methoden und Verfahren	9
2.3.1 Verwendete Tools und Technologien	9
3 Grundlagen	10
3.1 Arc Raiders & Gaming Tools	10
3.1.1 Companion Applications im digitalen Ökosystem	10
3.1.2 Gaming Companion Apps: Funktionale Kategorisierung	10
3.1.3 Extraction Shooter: Genre-spezifische Anforderungen	11
3.1.4 Arc Raiders: Technische Charakteristika und Systeme	11
3.1.5 Referenzimplementierungen: Tarkov Companion Ecosystem	12
3.2 Moderne Web-Technologien	12
3.2.1 TypeScript	13
3.2.2 React	14
3.2.3 Full Stack React Frameworks	15
3.3 UI/UX Frameworks & Design System	17
3.3.1 Grundlagen modularer Design-Systeme	17
3.3.2 Utility-First CSS: Paradigmenwechsel im Styling	18
3.3.3 Evolution der Komponenten-Bibliotheken	20
3.3.4 Graph-Visualisierung und automatische Layouts	22
3.4 State Management & Data Fetching	23
3.4.1 Evolution der State Management Paradigmen	23

3.4.2	Die Client-Server State Dichotomie	25
3.4.3	Zustand als minimalistischer Client State Manager	25
3.4.4	TanStack Query als Server State Spezialist	27
3.5	Datenbank und Backend Design	30
3.5.1	Supabase	30
3.5.2	Orm - Drizzle	30
3.6	Deployment & DevOps	30
3.6.1	Git basierter Workflow	30
3.6.2	Vercel	30
3.7	Testing Frameworks & Strategien	31
3.7.1	Vitest	31
3.7.2	Cypress	31
4	Durchführung	32
4.1	Anforderungsanalyse	32
4.2	Vorbereitung	32
4.3	Implementierung	32
4.4	Testen	32
5	Ergebnisse und Diskussion	33
5.1	Objektivierung der Ergebnisse	33
5.2	Diskussion?	33
5.3	Marktanalyse, vorhandene Tools/Anwendungen	33
6	Fazit und Ausblick	34
Anhang		
Literatur		35

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

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 [35]. 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 [17] [6].

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 [23].

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 [50]. 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 [2]. 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 Anforderungvalidierung

- **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 Architekturentwurf:** Definition der Systemgrenzen und Komponenten (Frontend, Backend, Datenbank, externe Datenquellen) sowie Auswahl geeigneter Architekturmuster unter Berücksichtigung der Komplexität. Die Dokumentation erfolgt durch Architekturdigramme nach dem C4-Modell (Context, Container, Component, Code), um verschiedene Abstraktionsebenen abzubilden und Stakeholdern unterschiedliche Detailgrade zu ermöglichen.
- **2.1.2 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.
- **2.1.3 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 Modellierungsdigrammen 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 [10].

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.

- **4.1.3 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 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.
- **4.3.2 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.
- **4.3.3 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 [5]. 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 [27].

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 [18].

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 [18].

Karten- und Navigationshilfen: Interactive Maps-Apps bieten detaillierte, community-erstellte Karten mit Loot-Spots, Extraktionspunkten, Key-Locations und Quest-Details [28].

Build-Planer und Kalkulatoren: Weapon Builder und Damage Calculator („Tarkov'd Simulator“) ermöglichen theoretisches Durchspielen von Szenarien vor der Implementierung im Spiel [18].

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 dopaminingesteuerte Befriedigung freisetzt [20]. 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 [19]. Arc Raiders hat durch seine Betonung von Teamsynergien und intuitiveren Spielerführungssystemen einen zugänglicheren Einstiegspunkt geschaffen [20].

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 [19]. 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 [53]. Bis zum 11. November 2025 hatte das Spiel weltweit über vier Millionen Exemplare verkauft [53], 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 [11]. 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 [11]. Bei begrenztem Inventarplatz ist strategisches Ressourcen-Management essentiell.

Skill-Progression: Der ARC Raiders Skill-Tree verzweigt sich in drei Pfade: Survival, Mobility und Conditioning [11], 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 [11].

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 [44]. 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, Bartern, Maps, Loot-Tiers, Hideout-Profiten und einer freien API [42]. 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 [21].

Type Safety in großen Projekten: Type Safety ist eines der herausragenden Merkmale von TypeScript und adressiert eine kritische Limitierung des dynamischen Typsystems von JavaScript [21]. In größeren Anwendungen, wo mehrere Entwickler am selben Codebase arbeiten, können unterschiedliche Annahmen über Datentypen zu unerwartetem Verhalten und schwer auffindbaren Bugs führen [21]. TypeScript ermöglicht es Entwicklern, Typen explizit für Variablen, Funktionsparameter und Rückgabewerte zu definieren, wodurch potenzielle Fehler zur Compile-Zeit statt zur Laufzeit erkannt werden [21].

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 [12]. Der strukturierte Ansatz von TypeScript hilft Teams, komplexe Codebases effizienter zu verwalten [12]. In größeren Teams verbessert Type Safety die Zusammenarbeit durch Reduzierung der kognitiven Last und Förderung klarerer Kommunikation [21].

Developer Experience: TypeScript verbessert nicht nur die Code-Qualität, sondern steigert auch signifikant die Developer Experience durch überlegenes Tooling und Error Reporting [21]. Die Integration mit modernen IDEs bietet erweiterte Features wie intelligente Code-Vervollständigung, automatisches Refactoring und Code-Navigation [8].

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 [43]. 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 [12].

Durch die Nutzung des statischen Typsystems von TypeScript können Entwickler sichereren und wartbareren Code schreiben und die Gesamtqualität und Zuverlässigkeit ihrer Anwendungen verbessern [8]. 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 [4].

3.2.2 React

React hat sich als führende JavaScript-Bibliothek für den Aufbau von Benutzeroberflächen etabliert, primär für Single Page Application (SPA) [16]. 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 [16]. Komponenten ermöglichen es, eine Anwendung in kleinere, unabhängige Teile zu zerlegen, die effizient verwaltet und wiederverwendet werden können [16]. 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 [16].

Jede React-Anwendung besteht aus einem Baum von Komponenten, wobei jede Komponente ihre eigene Logik, ihren State und ihre UI-Repräsentation hat [16]. 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 [16]

Komposition over Inheritance: React verfügt über ein leistungsstarkes Kompositionsmodell, und die Verwendung von Komposition anstelle von Vererbung wird empfohlen, um Code zwischen Komponenten wiederzuverwenden [13]. 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 [13]. Props und Komposition bieten die gesamte Flexibilität, die benötigt wird, um das Aussehen und Verhalten einer Komponente auf explizite und sichere Weise anzupassen [13].

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 [38]. 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 [38]

- **Einfachheit:** Sie fördert einfachere Hierarchien mit Komponenten, die isoliert verstanden werden können, ohne eine Vererbungskette kennen zu müssen [38]
- **Wiederverwendbarkeit:** Für Komposition entworfene Komponenten sind oft einfacher wiederzuverwenden, da sie keine Annahmen über den Kontext treffen, in dem sie verwendet werden [38]
- **Vermeidung von Tight Coupling:** Vererbung kann zu enger Kopplung zwischen Komponenten führen, was die Codebasis fragil und schwer zu refaktorisieren macht [38]

React fördert Komposition gegenüber Vererbung, weil sie größere Flexibilität und Trennung von Belangen ermöglicht [25]. Vererbung kann manchmal zu enger Kopplung zwischen Komponenten führen, was es schwieriger macht, Anwendungen zu modifizieren oder zu skalieren [25]. Durch die Übernahme von Komposition anstelle von Vererbung können Entwickler die Wiederverwendbarkeit von Komponenten vereinfachen und ihren Code flexibler und wartbarer gestalten [25].

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 [37]. Der App Router ist ein dateibasierter Router, der Reacts neueste Features wie Server Components, Suspense und Server Functions nutzt [47]. Next.js verwendet ein dateisystembasiertes Routing, bei dem Ordner zur Definition von Routen verwendet werden [49]. Jeder Ordner repräsentiert ein Routen-Segment, das einem URL-Segment zugeordnet wird [49].

File-Based Routing: Mit dem App Router ermutigt Next.js zu einem dateibasierten Routing-System, bei dem die Verzeichnisstruktur die URL-Struktur widerspiegelt [30]. 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 [32].

Next.js folgt weiterhin dem dateibasierten Routing, aber mit der Einführung des App Routers haben Dateien und Ordner nun strikt definierte Rollen [3]:

- **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 [3]
- **Dateien:** Dateien erstellen die UI für ein Routen-Segment [3]

Eine spezielle `page.ts`-Datei wird verwendet, um Routen-Segmente öffentlich zugänglich zu machen [49]. Dynamic Route Segments können durch Umschließen des Ordernamens mit eckigen Klammern erstellt werden, wie `[productId]` [3].

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) [46]. Diese wurden entwickelt, um Performance, SEO und User Experience in verschiedenen Situationen zu optimieren [46].

SSG: Bei SSG wird die initiale Hypertext Markup Language (HTML) zur Build-Zeit generiert. Ultra-schnellen Ladezeiten führt [37]. SSG ist ideal für Inhalte, die sich nicht häufig ändern, wie Blog-Posts, Dokumentation oder Marketing-Seiten [37]. Im App Router-Modell rendert SSG automatisch jede Komponente statisch, die keine server-spezifischen Funktionen nutzt [45].

SSR: SSR generiert HTML auf dem Server für jede Anfrage [37]. Diese Strategie ist optimal für echtzeitbezogene, nutzerspezifische Inhalte wie personalisierte Dashboards, Account-Profile oder News-Feeds [45]. SSR garantiert, dass jeder Besucher bei jedem Laden der Seite die neuesten Daten sieht [45].

ISR: ISR baut auf SSG auf und fügt die Fähigkeit hinzu, Inhalte zu aktualisieren, ohne einen vollständigen Rebuild der Site zu erfordern [37]. 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 [37]. ISR ist geeignet für Inhalte, die sich periodisch ändern, wie Blog-Posts oder Produkt-Listings [45].

PPR: PPR ist eine der neuesten Ergänzungen zu Next.js und befindet sich derzeit im experimentellen Status [45]. PPR ermöglicht es, dass eine Seite teilweise mit statischen und dynamischen Segmenten kombiniert pre-gerendert wird [45]. Dies ist besonders nützlich für Seiten mit Sektionen, die progressiv laden können, während kritischer Content sofort erscheint [45].

Partial Prerendering kombiniert ultra-schnelle statische Edge-Delivery mit vollständig dynamischen Fähigkeiten und hat das Potenzial, das Standard-Rendering-Modell für Webanwendungen zu werden [48]. PPR bietet ein vereinheitlichtes Modell, das die Zuverlässigkeit und Geschwindigkeit von ISR mit den dynamischen Fähigkeiten von SSR verbindet [48].

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 [36]. 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 [36]. Sobald die suspendierten Children auflösen, wird die UI zum Client in derselben Response gestreamt [36].

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 [46]. Next.js ermöglicht Entwicklern, verschiedene Rendering-Methoden innerhalb einer einzelnen Anwendung zu nutzen, je nach Bedarf, auf Seiten-Basis [46].

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 [15]. 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 [15]:

- **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 [15]:

- **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 [15]. 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 [39].

Tailwind CSS als Referenzimplementierung: Tailwind CSS etablierte sich als führendes Utility-First Framework [54]. 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 [39].

Die Philosophie manifestiert sich in einem einfachen Beispiel:

```

1 <!-- Traditionelles \ac{CSS} mit semantischen Klassen -->
2 <div class="message-warning">
3   <p class="message-text">Warnung!</p>
4 </div>
5
6 <!-- Utility-First mit Tailwind -->
7 <div class="bg-yellow-300 font-bold p-4 rounded">
8   <p class="text-gray-900">Warnung!</p>
9 </div>

```

Constraints-basiertes Design: Ein fundamentaler Vorteil von Utility-First liegt im constraints-basierten Design-Ansatz [39]. 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 [54]:

- **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 [39]. 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 [39]:

```

1 <button class="bg-blue-500 hover:bg-blue-700
2           md:w-auto sm:w-full
3           dark:bg-blue-900">
4   Responsive Button
5 </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 [26]. Diese Libraries folgen etablierten Design-Prinzipien – Bootstrap mit eigener Design-Language, Material-UI mit Googles Material Design Guidelines [22].

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 [26]:

- **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 [22].

Separation of Concerns: Headless Components implementieren strikte Trennung zwischen [24]:

- **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 [55]. Headless UI von Tailwind Labs ist optimiert für Integration mit Tailwind CSS. React Aria von Adobe fokussiert auf Internationalization und Platform-Agnostic Patterns [24].

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 [22].

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 [26].

Generation 3: Hybrid-Ansatz Shadcn/ui

Shadcn/ui synthetisiert Headless Primitives mit production-ready Styling und etabliert ein innovatives Distribution-Modell [29].

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) [29]:

```
1 npx shadcn-ui@latest init
2 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 [55]
- **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 [29].

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 [56]. 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 [56].

Dagre – Hierarchical Directed Graphs: Der Dagre-Algorithmus optimiert für Directed Acyclic Graph (DAG) mit klarer Hierarchie [56]:

Dagre minimiert Edge-Crossings und optimiert Layer-Spacing. [56].

Limitierungen: Dagre setzt uniform Node-Dimensionen voraus. Custom Node-Sizes erfordern Post-Layout Adjustments. Dynamisches Re-Layout bei Graph-Änderungen muss manuell getriggert werden [56].

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 [56].

Semantic-Specific Custom Algorithms: Generische Layout-Algorithmen können semantische Bedeutung von Graphen nicht berücksichtigen [52]. 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 [52].

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 [14]. 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 [33]. Diese konzeptuelle Trennung adressierte die Unvorhersagbarkeit von Backbone-artigen Model-View Architekturen, bei denen Template-Updates zu unkontrollierbaren Kaskaden von State-Mutationen führen konnten [33].

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 [33]. Bis 2016 etablierte sich die Konvention „If you’re using React, you must be using Redux too“ [33], 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 [33].

Das Boilerplate-Problem: Die fundamentale Kritik an Redux fokussierte sich auf den exzessiven Boilerplate-Code [14]. 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 [33]. 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 [33]. 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 [33]. 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 [33]. Moderne Architekturen trennen explizit zwischen Client State Management (Zustand, Jotai, Recoil) und Server State Management (TanStack Query, Stale-While-Revalidate (SWR), Apollo) [14]. Diese Spezialisierung reduziert Komplexität durch Single Responsibility: Bibliotheken fokussieren sich auf spezifische Problemdomänen statt universeller Lösungen [7].

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 [51]. 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 [51].

Server State – Asynchrone Remote-Daten: Server State repräsentiert Daten, deren autoritative Quelle serverseitig liegt und die asynchron fetched, gecacht und synchronisiert werden müssen [41]. Die inhärenten Charakteristika unterscheiden sich fundamental von Client State [51]:

- **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 [41]. Entwickler müssen manuell Loading States, Error Handling, Cache Invalidation, Request Deduplication und Background Refetching implementieren – Komplexität, die spezialisierte Libraries wie TanStack Query abstrahieren [7].

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 [34].

Philosophische Grundprinzipien: Zustand verfolgt eine „barebones“ Philosophie: minimale API-Surface, keine Opinions über State-Struktur, und vollständige Hook-basierte Integration [31]. Im Gegensatz zu Redux erfordert Zustand keine Provider-Wrapping der Application, keine Action Types, keine Reducer Boilerplate [34]. Der gesamte Store wird durch eine einzelne create-Function definiert [34]:

```

1 import { create } from 'zustand'
2
3 const useQuestStore = create((set, get) => ({
4   // State
5   activeFilter: 'all',
6   selectedQuest: null,
7
8   // Actions
9   setFilter: (filter) => set({ activeFilter: filter }),
10  selectQuest: (quest) => set({ selectedQuest: quest }),
11
12  // Computed/Derived State via get()
13  getFilteredQuests: () => {
14    const filter = get().activeFilter
15    // Filtering logic...
16  }
17 })))

```

Selektive Reactivity und Performance: Zustand implementiert feingradiges Dependency Tracking durch Proxy-basierte Subscriptions [34]. Components subscriben nur zu spezifischen State-Slices via Selector-Functions, was unnötige Re-Renders minimiert [31]:

```

1 // Component re-rendert nur bei activeFilter Changes
2 const filter = useQuestStore(state => state.activeFilter)
3
4 // Component re-rendert nur bei selectedQuest Changes
5 const quest = useQuestStore(state => state.selectedQuest)

```

Diese automatische Optimization eliminiert die Notwendigkeit manueller Memoization via `React.memo` oder `useMemo` [34].

DevTools und Persistence: Zustand bietet Middleware für Redux DevTools Integration und LocalStorage Persistence [31]:

```

1 import { devtools, persist } from 'zustand/middleware'
2
3 const useStore = create(
4   devtools(
5     persist(
6       (set) => ({ /* store definition */ }),
7       { name: 'quest-filter-storage' }
8     )
9   )
10 )

```

Die Persist-Middleware ermöglicht automatisches Speichern und Laden von State aus Local-Storage, essentiell für Features wie Quest-Filter-Präferenzen, die über Sessions persistieren sollen [34].

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 [34]. 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 [41].

Deklaratives Data Fetching: TanStack Query verschiebt Data Fetching von imperativem `useEffect`-Code zu deklarativen Query-Definitionen [7]:

```

1 import { useQuery } from '@tanstack/react-query'
2
3 function QuestList() {
4   const { data, isLoading, error } = useQuery({
5     queryKey: ['quests'],
6     queryFn: () => fetch('/api/quests').then(r => r.json())
7   })
8
9   if (isLoading) return <LoadingSpinner />
10  if (error) return <ErrorMessage error={error} />
11
12  return <div>{data.map(quest => <QuestCard quest={quest} />)}</div>
13 }

```

Diese Deklaration eliminiert manuelle State-Variablen für `loading`, `error` und `data`, die traditionell in Redux oder lokalem State verwaltet werden mussten [41].

Intelligent Caching und Query Keys: Das Cache-System basiert auf Query Keys als eindeutigen Identifiern [9]. 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 [7]

Die Parameter `staleTime` und `gcTime` (vormals `cacheTime`) kontrollieren dieses Verhalten [9]:

- **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) [9]

Request Deduplication: Wenn multiple Components simultan denselben Query requesten, konsolidiert TanStack Query diese zu einem einzelnen Network Request [7]. Alle subscribenden Components erhalten dieselbe Response, was Redundanz eliminiert:

```

1 // Component A
2 const { data } = useQuery({ queryKey: ['quests'], queryFn: fetchQuests })
3
4 // Component B (mountet simultan)
5 const { data } = useQuery({ queryKey: ['quests'], queryFn: fetchQuests })
6
7 // Resultat: Nur 1 HTTP Request, beide Components erhalten Response

```

Automatic Background Refetching: TanStack Query implementiert intelligente Refetch-Strategien [41]:

- **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 [7].

Mutations und Optimistic Updates: Für Data-Modification bietet TanStack Query `useMutation` mit Optimistic Update Support [40]:

```

1 const updateQuestMutation = useMutation({
2   mutationFn: (updates) => fetch(`/api/quests/${updates.id}`, {
3     method: 'PATCH',
4     body: JSON.stringify(updates)
5   }),
6
7   // Optimistic Update
8   onMutate: async (updates) => {
9     // Cancel outgoing refetches
10    await queryClient.cancelQueries({ queryKey: ['quests', updates.id] })
11
12    // Snapshot previous value
13    const previous = queryClient.getQueryData(['quests', updates.id])

```

```

14
15     // Optimistically update cache
16     queryClient.setQueryData(['quests', updates.id], updates)
17
18     // Return context for rollback
19     return { previous }
20 },
21
22 // Rollback on error
23 onError: (err, updates, context) => {
24     queryClient.setQueryData(['quests', updates.id], context.previous)
25 },
26
27 // Refetch after success/error
28 onSettled: (data, error, updates) => {
29     queryClient.invalidateQueries({ queryKey: ['quests', updates.id] })
30 }
31 })

```

Optimistic Updates ermöglichen instant UI-Feedback: Der Cache wird sofort mit der antizipierten Response aktualisiert, bevor die Server-Response eintrifft [40]. Bei Fehlern erfolgt automatisches Rollback zur vorherigen Cache-Version.

Cache Invalidation Strategien: TanStack Query bietet granulare Cache Invalidation [41]:

```

1 // Invalidate specific query
2 queryClient.invalidateQueries({ queryKey: ['quests', questId] })
3
4 // Invalidate all quest queries
5 queryClient.invalidateQueries({ queryKey: ['quests'] })
6
7 // Invalidate with predicate
8 queryClient.invalidateQueries({
9     predicate: (query) => query.queryKey[0] === 'quests' && query.state.data?.status === 'pending'
10 })

```

Integration mit Server-Side Rendering: TanStack Query unterstützt SSR durch Dehydration/Hydration Patterns [41]. Queries können serverseitig prefetched, dehydriert und im Client rehydriert werden, was FCP optimiert:

```

1 // 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

3.5.1 Supabase

PostgreSQL-basiert

Real-time Capabilities

Row Level Security

edge functions (Serverless Functions)

3.5.2 Orm - Drizzle

ORM's

Type-Safety

code-first approach

3.6 Deployment & DevOps

3.6.1 Git basierter Workflow

3.6.2 Vercel

Deployment von Nextjs Applikationen

weitere features

CI/CD Pipelines

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

an T2000 orientieren? SZeige, wie du von der Problemstellung zu Requirements kommst
Traceability Matrix: Verknüpfe Anforderungen → Issues → Tests → Code"

4.1 Anforderungsanalyse

- Zielgruppenanalyse
- Funktionale Anforderungen
- Nicht-funktionale Anforderungen

4.2 Vorbereitung

- Modellierung
- Technologieentscheidungen (kurz)

4.3 Implementierung

- Architektur
- Beispiel Datenbankmodell, edge function
- Beispiel API Endpunkte
- Beispiel Frontend Komponenten

4.4 Testen

- Teststrategie/-umgebung
- Beispiel Test
- Ergebnisse

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 Diskussion?

Stellungnahme zu den Ergebnisse, aufzeigen von Alternativen

5.3 Marktanalyse, vorhandene Tools/Anwendungen

(Parallelen zu Tools aus anderen Spielen wie Tarkov?)

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] 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).
- [3] 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).
- [4] 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).
- [5] 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).
- [6] Matthew Cochran. „Best Companion Apps In Video Games“. In: (2023).
- [7] 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).
- [8] 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).
- [9] 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).
- [10] G. Ebirim et al. „Advancements and innovations in requirements elicitation“. In: *World Journal of Advanced Research and Reviews* 22.01 (2024), S. 1209–1220.
- [11] Embark Studios. *ARC Raiders on Steam*. Valve Corporation. 2025. URL: https://store.steampowered.com/app/1808500/ARC_Raiders/ (besucht am 18.11.2024).
- [12] 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).
- [13] Facebook Open Source. *Composition vs Inheritance*. React Documentation. 2024. URL: <https://legacy.reactjs.org/docs/composition-vs-inheritance.html> (besucht am 18.11.2024).

- [14] 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).
- [15] Brad Frost. *Atomic Design*. Brad Frost, 2016. ISBN: 978-0-9982966-0-9. URL: <https://atomicdesign.bradfrost.com/> (besucht am 18.11.2024).
- [16] 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).
- [17] Olga Giersza. „Companion Apps are Taking Video Games to the Next Level“. In: (2024).
- [18] 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).
- [19] 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).
- [20] 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).
- [21] 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).
- [22] 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).
- [23] 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.
- [24] 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).
- [25] 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).
- [26] 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).
- [27] 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).

- [28] 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).
- [29] 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).
- [30] 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).
- [31] 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).
- [32] 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).
- [33] 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).
- [34] 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).
- [35] Grand View Research. *Video Game Market Size, Share And Growth Report*. 2024.
- [36] 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).
- [37] Ritesh Srivastava. „Next JS: Rendering Strategies — SSG, SSR, CSR and PPR“. In: (2025). Published: February 10, 2025. URL: <https://medium.com/@ritsriavastava/next-js-rendering-strategies-ssg-ssr-csr-and-ppr-d6243ec0ce72> (besucht am 18.11.2024).
- [38] 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).
- [39] Tailwind Labs. *Utility-First Fundamentals*. Tailwind CSS Documentation. 2024. URL: <https://tailwindcss.com/docs/utility-first> (besucht am 18.11.2024).
- [40] 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).
- [41] 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).

- [42] Tarkov.dev Community. *Tarkov.dev - A free, community made, and open source eco-system of Escape from Tarkov tools and guides*. 2024. URL: <https://tarkov.dev/> (besucht am 18.11.2024).
- [43] 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).
- [44] TreeStn. *Tarkov Companion - Desktop App on Overwolf*. Overwolf. 2024. URL: https://www.overwolf.com/app/treestn-tarkov_companion (besucht am 18.11.2024).
- [45] 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).
- [46] 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).
- [47] Vercel. *Next.js Docs: App Router*. Next.js Documentation. 2024. URL: <https://nextjs.org/docs/app> (besucht am 18.11.2024).
- [48] 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).
- [49] 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).
- [50] 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).
- [51] 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).
- [52] 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).
- [53] 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).
- [54] 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).

- [55] 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).
- [56] xyflow. *Layouting - React Flow*. React Flow Documentation. 2024. URL: <https://reactflow.dev/learn/layouting/layouting> (besucht am 18.11.2024).