Rechercheberichte

Erstellt von:	Überprüft von:
Miguel Friesen, Tom-Liam Müller, Maurice Nowotni, Noah Schwenk	Noah Schwenk

Version	Effektiv ab	Beschreibung / Änderungen
1.0	14.03.2026	Erster Entwurf
1.1	16.03.2026	Überarbeitung
1.2	17.03.2026	Anpassungen
1.3	18.03.2026	Kleine Änderungen
1.4	19.03.2026	Komplette Überarbeitung
1.5	20.03.2026	Änderungen und Überprüfung
2.0	21.03.2026	Korrektur lesen

Inhaltsverzeichnis

1. Allgemeines, Einsatzumfeld und Problemstellung

Die Problematik der manuellen Aufgabenplanung in der modernen Wissensarbeit besteht fundamental darin, dass Abhängigkeiten zwischen Aufgaben in der Praxis häufig übersehen werden und rein persönliche, subjektive Dringlichkeitseinschätzungen bei Beschäftigten systematisch zur kognitiven Überlastung führen. Die psychologische Fehlannahme klassischer Projektplanung liegt in der Illusion, dass menschliche Konzentration eine beliebig skalierbare und lineare Ressource sei.

Aus biologischer und neurologischer Sicht ist die Kapazität für hochkonzentrierte, tiefe Wissensarbeit auf ein striktes Limit von etwa 2 bis 4 Stunden pro Tag begrenzt. Dass diese Kapazität keine bloße Präferenz, sondern eine harte Leistungsgrenze darstellt, belegt bereits die fundamentale Studie von Ericsson et al. (1993), die das Limit für maximale Konzentration bei etwa vier Stunden verortet. Klassische Planungswerkzeuge ignorieren diese unumstößliche Tatsache vollständig. Sie verleiten Anwender zu einer unrealistischen Aneinanderreihung von kognitiv stark beanspruchenden Tätigkeiten. Dies führt unweigerlich zur Erschöpfung, wodurch kurzfristige Umplanungen zur Regel werden und die tatsächlichen Entscheidungsgrundlagen für die Arbeitsstrukturierung völlig intransparent bleiben.

Aus dieser scharfen Problemdefinition leitet sich die zwingende Systemlogik unserer architektonischen Lösung ab: Das zu entwickelnde System dient im absoluten Gegensatz zu etablierten Marktführern *ausschließlich* dem individuellen kognitiven Selbstmanagement. Ein zentrales Merkmal der Lösung ist die automatisierte Erstellung ausgewogener Arbeitspläne, die sowohl die Effizienz als auch die individuelle Belastbarkeit mathematisch-algorithmisch ausbalancieren.

Daraus folgt die strikte, architektonische Maxime für dieses Projekt: **Team-Features, Supervisor-Dashboards zur Mitarbeiterüberwachung oder aggregierte Ressourcenplanungen sind konzeptionell streng verboten.** Solche Makro-Management-Ansätze würden den systemischen Schutzmechanismus des Individuums sofort untergraben, da sie externe Deadlines und Fremdbestimmung über die neurologische Leistungsgrenze des Einzelnen stellen. Das System erstellt stattdessen algorithmisch generierte, nachhaltige Arbeitspläne völlig unabhängig von

subjektiven Fehleinschätzungen und rein auf Basis der physischen und mentalen Realität des Nutzenden.

Terminologie (Vorarbeit für das Glossar)

Um architektonische Missverständnisse in der anstehenden Definitions- und Entwurfsphase auszuschließen, sind im Vorfeld des in der Anforderungsanalyse geforderten Glossars folgende Begriffe zwingend und unumstößlich zu definieren:

- **Kognitive Kapazität:** Die neurologisch begrenzte, endliche Ressource eines Menschen für tiefe und konzentrierte Wissensarbeit. Sie ist auf ein biologisches Maximum von 2 bis 4 Stunden pro Tag limitiert. Das System nutzt diese Metrik als harte systemische Grenze, um bei der automatisierten Planung kognitive Überlastungen algorithmisch zu verhindern.
- **Flow-Schutz:** Ein systemischer Planungs- und Abwehrmechanismus der algorithmischen Engine. Er stellt sicher, dass tiefgreifende Fokusphasen (innerhalb der kognitiven Kapazität) nicht durch triviale Aufgaben, externe Störfaktoren oder algorithmisch falsch terminierte Pausen fragmentiert werden.
- **Energiebedarf (Ressourcenbedarf):** Ein quantifizierbarer Parameter, der bei der Konfiguration einer Aufgabe zwingend erfasst wird. Er definiert nicht die zeitliche Dauer, sondern den mentalen Ressourcenaufwand einer Aufgabe. Dieser Wert fließt direkt in die Arbeitsplangenerierung ein, um zu verhindern, dass Aufgaben mit hohem Energiebedarf gebündelt ohne ausreichende Regenerationsphasen eingeplant werden.
- **Individuelles Arbeitsprofil:** Die zentrale Konfigurationseinheit eines jeden Nutzenden. Hier werden die absoluten Rahmenbedingungen für die Algorithmen festgelegt, darunter die verfügbare Arbeitszeit pro Tag oder Woche, die maximale Dauer zusammenhängender Arbeitsphasen sowie die strikte Anzahl und Dauer von zwingenden Pausen. Jede Änderung an diesem Profil zwingt das System zu einer Neuberechnung zukünftiger Arbeitspläne.
- **Task-Switching:** Der psychologische und zeitliche Reibungsverlust, der durch den ständigen Wechsel zwischen Aufgaben unterschiedlicher Kontexte und Energiebedarfe entsteht. Unser Planungssystem minimiert diesen Overhead algorithmisch, indem es logische Abhängigkeiten und den Energiebedarf so anordnet, dass inhaltliche Brüche und der damit einhergehende kognitive Verschleiß drastisch reduziert werden.

Als Synthese der Recherche lassen sich folgende Leistungsmerkmale einer prototypischen persönlichen Aufgabenplanungsanwendung ableiten:

- Erfassung von Aufgaben mit Deadline, Priorität, geschätzter Dauer und Mindest-Zeiteinheit
- Unterstützung mehrerer Organisationen mit separaten Verfügbarkeitsfenstern pro Nutzer

- Automatische Arbeitsplangenerierung unter Berücksichtigung von Deadlines, Prioritäten und Aufgabenabhängigkeiten
- Verwaltung von Blockern und festen Aufgaben als unveränderliche Planungsgrenzen
- Proaktive Deadline-Warnung bei rechnerisch nicht mehr einplanbaren Aufgaben
- Automatisches Rescheduling bei Änderungen der Nutzerverfügbarkeit

2. Übersicht über themenrelevante Applikationen

Gemäß den Projektvorgaben existiert eine Vielzahl von Aufgaben-Planungssystemen, die als inspirativer Ausgangspunkt dienen. Eine rigorose architektonische Analyse der Zuweisungs-Algorithmen dieser Systeme offenbart jedoch, dass der Markt die biologischen und kognitiven Limitierungen der menschlichen Leistungsfähigkeit nach wie vor gravierend vernachlässigt.

2.1 Motion (Usemotion)

Motion ist eine KI-gestützte Planungsanwendung, die Aufgaben, Meetings und Deadlines in einer einzigen Oberfläche zusammenführt. Nutzer legen Aufgaben mit Priorität, Deadline und geschätzter Dauer an und der Algorithmus verteilt sie vollautomatisch in freie Kalenderslots. Ändert sich ein Termin, plant Motion alle betroffenen Aufgaben sofort in die nächsten verfügbaren Zeitfenster um.

- **Verwendeter Algorithmus:** Modifizierter *Constraint-Satisfaction-Algorithmus* (Bedingungserfüllung) gepaart mit einem *Greedy-Ansatz* (gierige Lückenfüllung).
- **Architektonische Kritik:** Der Algorithmus betrachtet den Kalender als Array von verfügbaren Zeitblöcken und sortiert Aufgaben primär nach Fristen. Er optimiert den Kalender auf 100 % Auslastung (wie bei einem Server-Cluster) und plant komplexe Denkaufgaben direkt hintereinander, solange der Kalender leer ist. Das fundamentale Konzept der begrenzten "kognitiven Kapazität" (2-4 Stunden Limit) sowie ein echter Flow-Schutz fehlen vollständig.
- **Tech-Stack:** React und TypeScript im Frontend; Node.js und Temporal für die asynchrone Backend-Orchestrierung; PostgreSQL (relationale Datenbank); gehostet auf der Google Cloud Platform (GCP).

2.2 Timehero

Timehero ist eine Planungsplattform mit einem klaren Fokus auf ressourcenübergreifendes Projektmanagement. Das System fokussiert sich auf die dynamische Platzierung von sogenannten „Fluid Time“-Blöcken rund um statische

Kalender-Events, wobei anstehende Aufgaben bedarfsgerecht fragmentiert und automatisch in freie Zeitlücken zwischen Meetings eingepasst werden.

- **Verwendete Algorithmen:** Timehero operiert mit einem reaktiven, eventbasierten Scheduling-Algorithmus, der Aufgaben bedingungsgesteuert in verfügbare Zeitfragmente aufteilt.

- **Architektonische Kritik:** Die Architektur optimiert primär auf mathematische Zeiteffizienz, Termintreue (Deadlines) und Durchsatz. Psychologische Belastungsmetriken wie die „Löffel-Theorie“ (Spoon Theory) werden nicht als algorithmische Constraints modelliert und somit völlig ignoriert. Die aggressive Aufgabenfragmentierung ("Zerstückeln") erzwingt bei diesem System zudem ein hohes Maß an "Task-Switching", was den kognitiven Flow der Nutzer empirisch massiv beeinträchtigt.
- **Tech-Stack:** PHP (Laravel-Framework) im Backend; MySQL zur relationalen Datenhaltung; Angular und Ionic im Frontend; Cloud-SaaS-Infrastruktur.

2.3 Sunsama

Sunsama ist eine tagesorientierte Planungsanwendung, die Aufgaben aus verschiedenen Quellen wie Trello, Asana oder GitHub in einer zentralen Tagesansicht bündelt. Der Nutzer zieht Aufgaben manuell per Drag-and-Drop in Zeitblöcke und gibt dabei eine realistische Zeitschätzung an. Das System zeigt kontinuierlich die Summe der verplanten Stunden an und warnt aktiv, sobald der Tag rechnerisch überplant ist.

- **Verwendeter Algorithmus:** *Manuelle Kapazitäts-Heuristik*. Das System summiert die geplante Dauer der Aufgaben und blockiert ab einem bestimmten Punkt das Hinzufügen weiterer Aufgaben.
- **Architektonische Kritik:** Sunsama modelliert das menschliche Limit (die kognitive Kapazität) hervorragend und schützt vor Überarbeitung. Es versagt jedoch auf architektonischer Ebene bei der Automatisierung: Wenn der Kalender kollabiert, fehlt ein intelligentes Backend (Auto-Rescheduling), das die Umplanung vollautomatisch durchführt.
- **Tech-Stack:** React-basiertes Frontend; Node.js im Backend; MongoDB Atlas als dokumentenorientierte NoSQL-Datenbank für eine strikte Workspace-Isolation; bereitgestellt über die AWS-Infrastruktur (us-east-1).

2.4 Morgen

Morgen ist eine Kalender- und Aufgaben-App, die Aufgaben und Termine in einer gemeinsamen Tagesansicht zusammenführt. Der Nutzer definiert thematische Zeitfenster, sogenannte Frames, für verschiedene Arbeitstypen wie Fokusarbeit oder administrative Tätigkeiten. Die KI schlägt anschließend passende Aufgaben für jeden Frame vor. Die finale Einplanung bestätigt der Nutzer manuell, wodurch das System stets eine bewusste Entscheidung des Nutzers erfordert.

- **Verwendeter Algorithmus:** *Frame-Matching-Algorithmus (Regelbasierte Klassifizierung)*. Der Nutzer definiert Zeitfenster (Frames) für verschiedene Arbeitstypen. Die KI schlägt Aufgaben vor, die in diese Frames passen (Human-in-the-Loop).

- **Architektonische Kritik:** Die Zuweisung ist sehr starr. Unser System muss dynamischer agieren und den individuellen Biorhythmus bzw. Chronotypen (Hochphasen am Morgen vs. Abend) algorithmisch auswerten, um Arbeitspläne zu generieren, die den energetischen Peak des Nutzers mathematisch abbilden.
- **Tech-Stack:** Node.js und Express.js für die serverseitige Logik; Electron (für die Cross-Platform-Desktop-App) und React im Frontend; MongoDB Atlas (NoSQL) als Primärdatenbank; gehostet auf der Google Cloud Platform (GCP).

2.5 Microsoft To Do

Microsoft To Do ist eine persönliche Aufgabenverwaltungsanwendung von Microsoft, die seit 2017 als direkter Nachfolger von Wunderlist verfügbar ist. Nutzer erfassen Aufgaben mit Fälligkeitsdatum, Priorität, Wiederholungen und Unteraufgaben und organisieren sie in thematischen Listen. Eine automatische zeitliche Einplanung in Verfügbarkeitsfenster existiert nicht. Die Anwendung unterstützt ausschließlich bei der Erfassung und Strukturierung, die zeitliche Koordination verbleibt vollständig beim Nutzer.

- **Verwendeter Algorithmus:** Keiner. Microsoft To Do ist ein reines Erfassungs- und Strukturierungssystem ohne jegliche Planungslogik.
- **Architektonische Kritik:** Das Fehlen eines Planungsalgorithmus ist kein technisches Versagen, sondern eine bewusste Produktentscheidung: Microsoft To Do löst das Problem der Aufgabenerfassung, nicht das der Aufgabenverteilung. Für unser System markiert das genau die Lücke, die wir schließen, ein vollständiges Aufgabenmodell mit Deadline und Priorität ist notwendige Voraussetzung, reicht aber alleine nicht aus. Aus technischer Perspektive ist Microsoft To Do für dieses Projekt besonders relevant: Die Anwendung wird vollständig auf Microsoft Azure betrieben, das Backend basiert auf C# und

dem .NET-Framework, das Frontend ist in React implementiert. Die Wahl desselben Technologie-Stacks durch einen produktionsreifen Microsoft-Dienst mit Millionen von Nutzern bestätigt die Eignung dieser Technologien für skalierbare Web-Applikationen und validiert unsere eigene Stack-Entscheidung (siehe Abschnitt 3.2).

- **Tech-Stack:** C# .NET-Framework im Backend; React im Frontend; Microsoft SQL Server (Azure SQL) als relationale Datenbasis; nativ gehostet auf der Microsoft Azure Cloud (Enterprise-Skalierung).

3. Übersicht über IT-Spezifika und Technologie-Evaluation

Die Analyse der in Abschnitt 2 betrachteten Systeme zeigt, dass persönliche Aufgabenplanungsanwendungen trotz unterschiedlicher Schwerpunkte eine Reihe gemeinsamer technischer Muster aufweisen. Diese Gemeinsamkeiten ergeben sich direkt aus den funktionalen Anforderungen solcher Systeme: Aufgaben müssen plattformübergreifend verfügbar sein, Planungsänderungen müssen unmittelbar sichtbar werden, und die Planungslogik muss performant auf sich ändernde Nutzereingaben reagieren. Im Folgenden werden die technischen Kernmuster beschrieben, die für die Entwicklung des eigenen Systems relevant sind, und daraus die Technologie-Entscheidungen abgeleitet.

3.1 Technologische Erkenntnisse aus der Marktanalyse

Cloud-basierte SaaS-Architektur

Alle analysierten Systeme, Motion, Timehero, Sunsama, Morgen und Microsoft To Do, werden ausschließlich als Cloud-basierte Software-as-a-Service-Lösungen betrieben. Eine lokale Installation existiert bei keinem dieser Systeme als primäres Modell. Dies hat einen klaren technischen Grund: Aufgaben und Pläne müssen geräteübergreifend synchron sein. Ändert der Nutzer auf dem Smartphone eine Deadline, muss diese Änderung sofort im Browser sichtbar sein. Cloud-Infrastrukturen mit verwalteten Diensten erlauben dabei eine automatische Skalierung ohne manuelle

Serveradministration, was besonders für kleine Entwicklungsteams mit begrenzter Projektlaufzeit entscheidend ist.

Frontend-Technologien und Interaktionsmodelle

Bei den Frontend-Technologien zeigt die Marktanalyse eine klare Dominanz komponentenbasierter JavaScript-Frameworks. Microsoft To Do setzt im Frontend auf React, Motion auf einen TypeScript-basierten Stack. Die Gemeinsamkeit liegt im Interaktionsmodell: Aufgabenplanungsanwendungen erfordern hochreaktive Oberflächen, bei denen Änderungen, wie das Verschieben einer Aufgabe per Drag-and-Drop oder die sofortige Visualisierung einer neu berechneten Planung, unmittelbar reflektiert werden, ohne die gesamte Seite neu zu laden. Klassisch server-seitig gerenderte Anwendungen sind hierfür ungeeignet; ein Virtual-DOM-basierter Ansatz wie React ermöglicht isolierte, performante UI-Updates.

Datenhaltung: Relationale und dokumentenorientierte Modelle

Die analysierten Systeme verwenden unterschiedliche Datenbankstrategien abhängig von ihrem Datenmodell. „Systeme mit klar strukturierten, tabellarischen Abhängigkeiten, wie Timehero, dessen Backend auf PHP (Laravel) mit einer relationalen MySQL-Datenbank basiert, setzen auf klassische relationale Modelle. Planungsanwendungen mit dynamischen, nutzerspezifischen Strukturen wie individuellen Verfügbarkeitsfenstern, organisationsbezogenen Konfigurationen und flexiblen Aufgabenattributen tendieren dagegen zu dokumentenorientierten NoSQL-Datenbanken, da diese schema-flexibel auf sich ändernde Anforderungen reagieren können. Für ein persönliches Planungssystem, bei dem jeder Nutzer ein individuelles Profil mit eigenen Verfügbarkeiten und Aufgabenpools pro Organisation besitzt, ist ein dokumentenorientierter Ansatz besonders geeignet.

Planungslogik und Echtzeit-Reaktion

Das technisch anspruchsvollste Merkmal von Systemen wie Motion ist die Fähigkeit zur Echtzeit-Neuplanung: Ändert sich ein Kalendereintrag oder eine Deadline, berechnet das Backend den Arbeitsplan umgehend neu und überträgt das Ergebnis sofort an das Frontend. Dies setzt zwei Dinge voraus: erstens eine Backend-Logik, die Planungsberechnungen schnell und zuverlässig ausführen kann, und zweitens eine

Infrastruktur, die Ergebnis-Updates in Echtzeit an den Client pushen kann – etwa über WebSockets oder Echtzeit-Datenbankabonnements. Der Kontrast zu Sunsama macht dies deutlich: Sunsama verzichtet bewusst auf ein automatisches Rescheduling-Backend und überträgt diese Last manuell an den Nutzer – mit dem Ergebnis, dass Planungskonflikte zwar erkannt, aber nicht aufgelöst werden. Die Wahl der Infrastruktur hat damit direkten Einfluss auf die Implementierbarkeit der Planungslogik.

3.2 Technologie-Evaluation und Stack-Entscheidung

Eine anspruchsvolle, fachlich strikt eingegrenzte Software (rein auf das kognitive Selbstmanagement des Individuums fokussiert) verlangt nach einem kompromisslosen, hochprofessionellen architektonischen Fundament. Unser System ist keine triviale Datenverwaltungs-Applikation (CRUD), sondern eine reaktionsschnelle, mathematische Planungsmaschine. Gemäß der Anforderung /F40/ muss das System in der Lage sein, Arbeitspläne unter strikter Einbeziehung von kognitiver Kapazität, Energiebedarf, Prioritäten und komplexen Aufgabenabhängigkeiten vollautomatisch zu berechnen.

Um diese rechenintensive Logik performant und fehlerfrei auszuführen, haben wir uns nach einer rigorosen Evaluation für den folgenden Technologie-Stack entschieden, der exakt auf unsere Projektanforderungen und die zeitlichen Restriktionen (fünf Wochen) zugeschnitten ist. Die Wahl der Systemarchitektur und die Technologie-Evaluation folgen dabei strikt den etablierten Methoden des Software Engineerings.

Backend-Architektur: C# .NET für die algorithmische Engine

Die automatisierte Generierung von Arbeitsplänen ist das funktionale und mathematische Herzstück unseres Systems. Der Verteilungs-Algorithmus muss hunderte Parameter (Deadlines, kognitive Limits von 2-4 Stunden, Pausenrhythmen und Energiebedarfe) simultan verarbeiten und Abhängigkeitsgraphen auflösen. Für diese massiv rechenintensive Logik ist eine Skriptsprache unzureichend.

Wir setzen hier zwingend auf **C# in Kombination mit dem .NET-Framework**. Die Entscheidung begründet sich architektonisch wie folgt:

- **Massive Rechenleistung und Typsicherheit:** C# bietet durch seine starke, statische Typisierung eine herausragende Laufzeitsicherheit und Performance. Dies eliminiert eine Vielzahl potenzieller Laufzeitfehler bei der Berechnung komplexer Aufgabenabhängigkeiten bereits zur Kompilierzeit.
- **Stabilität für Geschäftslogik:** Die Validierung durch etablierte Marktakteure bestätigt die Eignung dieses Stacks: Microsoft To Do, ein produktionsreifer Dienst mit Millionen aktiver Nutzer, setzt ebenfalls auf C# .NET im Backend und React im

Frontend und beweist damit, dass dieser Technologie-Stack sowohl für die skalierbare Datenhaltung als auch für hochreaktive Planungsoberflächen geeignet ist.

- **Asynchrone Programmierung:** Die tiefe Integration von async/await-Mustern in ASP.NET Core erlaubt eine hochgradig effiziente, blockierungsfreie Verarbeitung paralleler API-Anfragen.

Frontend-Architektur: React.js für dynamische Interaktionsmodelle

Die Planungsoberfläche unseres Systems muss für den Nutzenden unmittelbar und reibungslos reagieren. Um den Flow-Schutz zu gewährleisten, darf die Bedienung des Tools selbst keine zusätzliche kognitive Last erzeugen.

Wir evaluieren **React.js** als exklusives Frontend-Framework und verwerfen Alternativen wie Angular.

- **Virtual DOM und Rendering-Performance:** Die Benutzeroberfläche erfordert eine hochgradig interaktive Kalender- und Listenansicht, bei der Aufgaben per Drag-and-Drop verschoben werden können, woraufhin die Engine sofortige Re-Kalkulationen visualisieren muss. Der Virtual DOM von React.js garantiert hierbei flüssige, isolierte UI-Updates, ohne den Browser durch vollständige Seiten-Reloads auszubremsen.
- **Deklaratives Programmiermodell:** React zwingt uns zu einer sauberen, komponenten-basierten Architektur. Dies fördert die Wiederverwendbarkeit von UI-Elementen und ermöglicht dem Team – im direkten Vergleich zur steilen Lernkurve und den starren Vorgaben ("The Angular Way") von Angular – eine drastisch effizientere Implementierungsphase.

Infrastruktur und Datenhaltung: Google Cloud Platform (GCP)

Wir verfolgen eine radikale **Cloud-First-Strategie**, um jeglichen DevOps- und Administrations-Overhead zu eliminieren und uns rein auf die Systemlogik konzentrieren zu können. Die Google Cloud Platform (GCP) übertrifft AWS und Azure in diesem speziellen Projektkontext durch ihre Best-in-Class PaaS- (Platform as a Service) und Serverless-Dienste.

- **Zustandslose Skalierbarkeit mit Cloud Run:** Das rechenintensive C#-Backend wird in Docker-Containern isoliert und über *Google Cloud Run* gehostet. Cloud Run ist eine vollständig verwaltete Serverless-Umgebung. Sie skaliert unsere C#-Container dynamisch bei jeder Algorithmus-Anfrage hoch und fährt bei Inaktivität sofort auf null Ressourcen herunter ("Scale-to-Zero"). Dies garantiert maximale

Kosteneffizienz und sofortige Verfügbarkeit der Rechenleistung für die Planungsengine.

- **Schema-flexible Datenspeicherung mit Cloud Firestore:** Klassische relationale Datenbanken sind für die hochdynamischen, tief verschachtelten Profil- und Aufgabenparameter unseres Systems zu starr. Wir nutzen die NoSQL-Datenbank *Cloud Firestore*. Sie ist dokumentenorientiert und erlaubt uns, relationale Planungslogiken agil auf eine schema-flexible Struktur abzubilden. Firestore liefert zudem Out-of-the-Box Echtzeit-Synchronisation, was zwingend notwendig ist, sobald das Backend einen Arbeitsplan neu generiert und das React-Frontend diese Updates verzögerungsfrei an den Nutzenden pushen muss.
- **Identity Management (Firebase Auth):** Das System lagert die hochkomplexe und sicherheitskritische Benutzerauthentifizierung vollständig an Firebase Authentication aus. Dies schützt das Individuum durch kryptografische Standards (Bearer Tokens) vor Fremdzugriffen und senkt den serverseitigen Implementierungsaufwand massiv.

3.3 Evaluierung der Team-Kompetenzen und Qualifizierungsbedarf

- Gemäß den Vorgaben der Projektplanung muss die architektonische Entscheidung gegen die bestehende Kompetenzmatrix der Projektgruppe gespiegelt werden. Der gewählte Technologie-Stack aus C# .NET, React.js und der Google Cloud Platform ist hochprofessionell, erfordert jedoch ein zielgerichtetes Upskilling des Teams innerhalb der Vorprojektphase ("Sprint 0"):
- **Backend & Algorithmik (C# .NET):** Die Projektgruppe verfügt über belastbare Grundkenntnisse in der objektorientierten Programmierung. Die Implementierung des hochkomplexen, mathematischen Verteilungs-Algorithmus (inklusive Flow-Schutz und der harten 2-4h-Kapazitätsgrenze) verlangt jedoch zwingend die zügige Einarbeitung in asynchrone Programmiermuster (async/await) innerhalb von ASP.NET Core. Nur so kann eine blockierungsfreie und performante Berechnung der Arbeitspläne sichergestellt werden.
- **Frontend (React.js):** Aktuell existieren im Team primär Basiskenntnisse in der generischen Webentwicklung. Das deklarative Paradigma von React (Virtual DOM, Komponenten-Lebenszyklen, State-Management via Hooks) muss zwingend erlernt werden, um die interaktive, flüssige Drag-and-Drop-Planungsoberfläche kognitiv reibungslos und absturzfrei umzusetzen.
- **Infrastruktur & Datenhaltung (GCP/Firestore):** Erfahrungswerte im professionellen Cloud-Deployment (Docker-Containerisierung, Cloud Run) sind im Team derzeit marginal vorhanden. Besonders das Designschema für NoSQL-Datenbanken (Cloud Firestore) stellt ein kritisches Lernfeld dar. Das Team muss lernen, wie relationale Denkmuster (z.B. für verschachtelte Aufgabenabhängigkeiten und Profilparameter) effizient auf

dokumentenorientierte, schema-flexible NoSQL-Strukturen migriert werden können.

4. Genauere Beschreibung einer konkreten Applikation: Usemotion (Motion)

4.1 Verbale Beschreibung der Leistungsmerkmale

Um die algorithmische Mechanik einer modernen Planungsoberfläche zu evaluieren und die Parameter für unsere eigene C#-Backend-Engine exakt zu definieren, betrachten wir die Applikation *Motion* auf Systemebene. Motion agiert als prädiktiver KI-Task-Manager, der den Arbeitstag durch einen Autopiloten vollautomatisch plant.

Das architektonische Herzstück von Motion ist das "Predictive Auto-Scheduling". Im Gegensatz zu trivialen To-Do-Listen löst Motions Algorithmus das Planungsproblem nicht linear, sondern bewertet Aufgaben anhand eines komplexen Regelwerks, das konstant gegen die Kalenderverfügbarkeit (Availability) des Nutzenden validiert wird.

Die innere Logik arbeitet nach einer strikten, deterministischen Hierarchie, um die Zuweisung in freie Kalender-Slots (Token-Zuweisung) zu steuern:

1. **ASAP-Status (Override):** Dieser Parameter überschreibt alle anderen Signale. Die Aufgabe wird zwingend vor allen anderen eingeplant.
2. **Hard Deadlines (Harte Fristen):** Das System garantiert die Einhaltung und plant Aufgaben notfalls auch außerhalb der regulären Arbeitszeiten ein, um die Deadline zu retten.
3. **Soft Deadlines:** Reguläre Fälligkeitsdaten.
4. **Priority (Priorität):** Sortierung nach High, Medium, Low.
5. **Duration & Chunking (Dauer & Zerstückelung):** Kann der Algorithmus keinen ausreichend großen Zeitslot finden, greift die "Chunking"-Funktion. Die Aufgabe wird in kleinere Blöcke (z. B. 30 Minuten) zerteilt, bis sie in den Kalender passt.
6. **Recurrence (Wiederholung):** Wiederkehrende Aufgaben werden zur Wahrung von Routinen vor einmaligen Aufgaben mittlerer Priorität geplant.

Kommt es durch externe Störfaktoren (z. B. ein neues Meeting) zu Terminüberschneidungen, greift das **Auto-Rescheduling**: Der Algorithmus erkennt den Konflikt im gerichteten Graphen und iteriert asynchron über alle betroffenen Knoten, um diese neu zu verteilen.

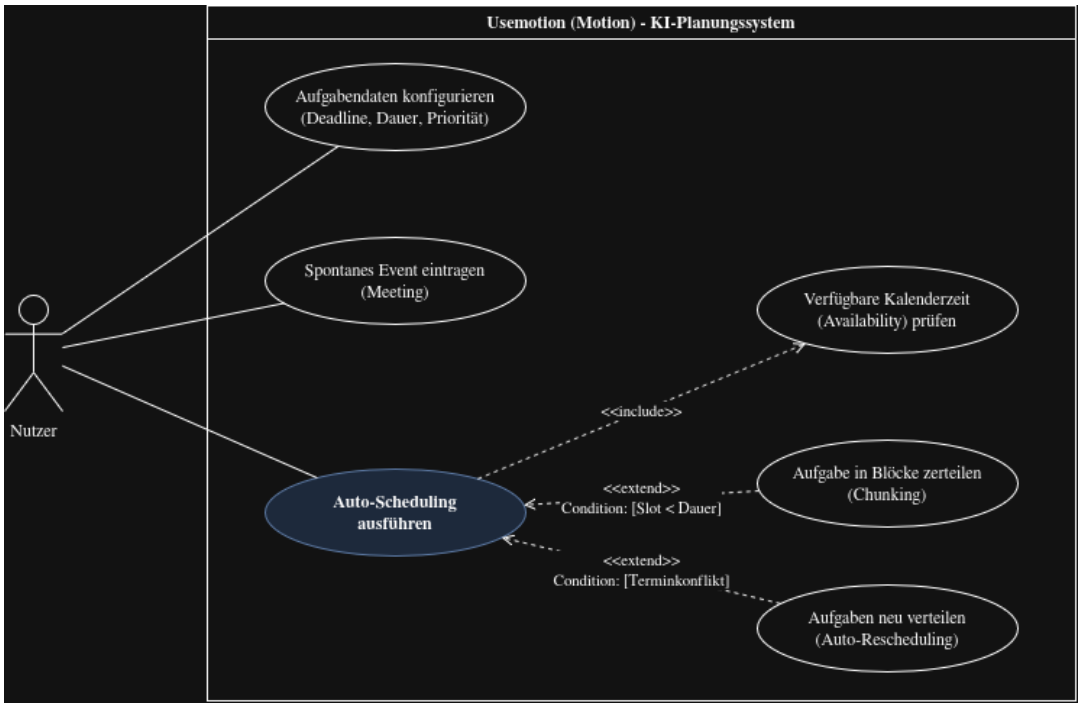
4.2 Beschreibung von Anwendungsfällen (Use Cases)

Use Case 1: Arbeitsplan durch KI generieren lassen
Ziel: Vollautomatische Verteilung von Aufgaben in freie Kalender-Slots nach Priorität und Deadline
Kategorie: primär
Vorbedingung: Der Nutzende hat Aufgaben mit Deadline, Priorität und Dauer im System hinterlegt.

Nachbedingung Erfolg: Alle Aufgaben sind konfliktfrei im Kalender platziert.
Akteure: Nutzender
Auslösendes Ereignis: Eine neue Aufgabe wird angelegt oder der Kalender ändert sich.
Beschreibung: 1. System liest alle offenen Aufgaben und Kalendereinträge ein. 2. System bewertet Aufgaben nach ASAP-Status, Hard Deadline, Priorität und geschätzter Dauer. 3. System platziert Aufgaben der Reihe nach in freie Kalender-Slots, ist ein Slot zu klein, wird die Aufgabe per Chunking in kleinere Blöcke aufgeteilt.

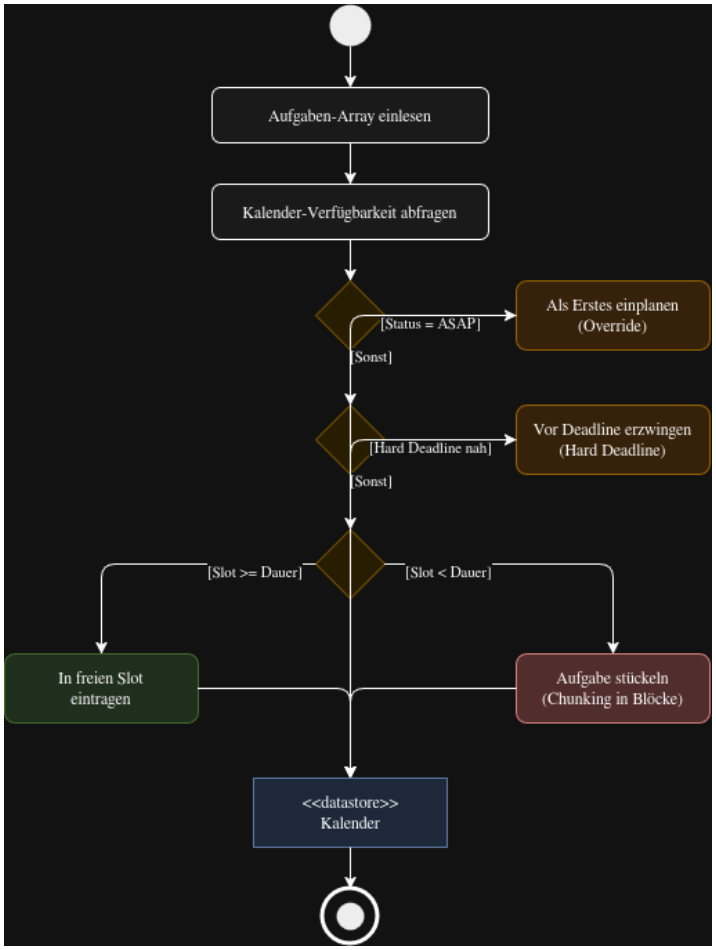
Use Case 2: Individuelles Arbeitsprofil konfigurieren
Ziel: Definition der kognitiven Limits zur Steuerung der Engine.
Kategorie: primär
Vorbedingung: Nutzender ist registriert.
Nachbedingung Erfolg: Das Profil ist in der Datenbank gespeichert.
Akteure: Nutzender
Auslösendes Ereignis: Nutzender ruft Profil-Einstellungen auf.
Beschreibung: 1. System zeigt Konfigurationsmaske. 2. Nutzender definiert kognitive Kapazität (2-4h) und den maximalen Energiebedarf pro Tag. 3. System speichert das Profil.

4.3 Darstellung der inneren Logik (UML-Diagramme)



Das Use-Case-Diagramm (UML 2.5) modelliert die exakte funktionale Abdeckung von Motion. Der primäre Anwendungsfall, Auto-Scheduling auszuführen, zwingt das System über eine <<include>>-Beziehung dazu, die Verfügbarkeit (Availability) auszuwerten. Die von Motion verwendete Konfliktauflösung wird durch präzise <<extend>>-Beziehungen dargestellt: Ist ein freier Zeitslot kleiner als die geschätzte Aufgabendauer, greift die

Erweiterung Aufgabe stückeln (Chunking). Entstehen durch neue Termine Konflikte, greift das Auto-Rescheduling.



Analyse des Kontrollflusses (Token-Semantik): Das Aktivitätsdiagramm seziert die deterministische Hierarchie des Motion-Algorithmus. Nach der Abfrage der Kalenderverfügbarkeit durchläuft das Token mehrere Entscheidungsknoten (Rauten). Zunächst wird der ASAP-Status (höchste Priorität) und anschließend die Hard Deadline geprüft. Der entscheidende Architektur-Fehler klassischer Tools wird am letzten Entscheidungsknoten sichtbar: Die Engine prüft lediglich die zeitliche Passung ([Slot >= Dauer]). Schlägt dies fehl, nutzt Motion das "Chunking" , um die Aufgabe gnadenlos in verbleibende 30-Minuten-Lücken zu pressen, bis der Kalender zu 100 % ausgelastet ist (Motion Help Center, 2026).

Fazit für unsere eigene Systemarchitektur: Das Diagramm beweist visuell den fundamentalen Architektur-Fehler klassischer Tools: Die Prüfung der "kognitiven Kapazität" (Spoon-Theorie) existiert im Kontrollfluss der Konkurrenz nicht. Unsere eigene

C#-Applikation wird diesen Ablauf in der Basis adaptieren, muss jedoch zwingend eine zusätzliche Entscheidungsschleife (Guard: [Kognitives Tageslimit überschritten?]) vor der finalen Kalendereintragung implementieren, um kognitive Überlastung zu verhindern.

5. Literaturverzeichnis

- Balzert, Helmut; Liggesmeyer, Peter (2011): Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb. 3. Aufl. Heidelberg: Spektrum Akademischer Verlag.
- BIG direkt gesund (2026): Löffel-Theorie: Sichtbarkeit und Energiemanagement für Menschen mit chronischen Erkrankungen. Online unter: https://www.big-direkt.de (Zugriff: 20.03.2026).
- Morgen AG (2026): 5 beste KI-Task-Manager-Software | Getestet & aktualisiert für 2026. Online unter: https://www.morgen.so (Zugriff: 20.03.2026).
- Timeflow (2026): Timeflow Aufgaben-Planung. Online unter: https://www.timeflow.site/ (Zugriff: 20.03.2026).
- Usemotion (2026): Motion – AI Task Manager and Calendar. Online unter: https://www.usemotion.com/ (Zugriff: 20.03.2026).
- Motion Help Center (2026): How the Automatic Scheduler works. Online unter: https://www.usemotion.com/help/time-management/auto-scheduling (Zugriff: 21.03.2026).
- Friedman, Benny (2024): Unmasking Motion AI Assistant: Under the Hood of AI Scheduling. Online unter: https://medium.com/@benny-friedman2/unmasking-motion-ai-assistant-under-the-hood-ai-scheduling-ac86fff1e26e (Zugriff: 21.03.2026).
- ERICSSON, K. A., KRAMPE, R. T. und TESCH-RÖMER, C., 1993. The role of deliberate practice in the acquisition of expert performance. Psychological Review. Bd. 100, Nr. 3, S. 363-406
- Microsoft Corporation (2026): Microsoft To Do – Lists, Tasks & Reminders. Online unter: https://to-do.microsoft.com (Zugriff: 20.03.2026)