Fachhochschule Nordwestschweiz FHNW

Individuelle praktische Arbeit Informatiker/in EFZ
Applikationsentwicklung

Kalenderbasierte Auftragssteuerung für einen autonomen Assistenzroboter

Autor:	Keanu Maleeq Koelewijn
Institut:	Institut für Mobile und Verteilte Systeme (IMVS)
Verantwortliche Fachkraft:	Andri Wild
Haupt-/Nebenexperten:	Daniel Ramser, Roland Michelberger
Startdatum:	30.03.2026
Abgabedatum:	14.04.2026

Eigenständigkeitserklärung

Hiermit erkläre ich, Keanu Maleeq Koelewijn, dass ich die vorliegende IPA-Arbeit selbstständig und ohne fremde Hilfe verfasst habe. Sämtliche verwendeten Quellen und Hilfsmittel sind im Bericht vollständig und korrekt angegeben. Ich bestätige zudem, dass alle KI-Anfragen, die während der Erstellung dieser Arbeit gemacht wurden, im Bericht dokumentiert sind.

Ort, Datum: _____

Unterschrift: _____

Inhaltsverzeichnis

1 Umfeld und Ablauf

1.1 Aufgabenstellung

1.1.1 Ausgangslage

Das Institut für Mobile und Verteilte Systeme (IMVS) der FHNW betreibt einen autonomen Assistenzroboter. Dabei handelt es sich um eine komplexe mobile Plattform, die eigenständig durch Räumlichkeiten navigiert und mit Personen interagiert. Um dies zu ermöglichen, ist der Roboter mit modernster Technik ausgestattet, die ihm eine eigenständige Fortbewegung auf Basis einer digitalen Gebäudekarte erlaubt.

Innerhalb des Instituts dient der Roboter primär als Forschungsplattform für Studierendenprojekte. Da die Bedeutung autonomer Roboter stetig zunimmt und wir künftig immer häufiger Berührungspunkte mit solchen Systemen haben werden, sollen auch Institutsmitarbeitende den Roboter über einen öffentlichen Kalender buchen können. Ziel ist es, Berührungspunkte für Personen zu schaffen, die im Alltag nicht mit autonomen Systemen arbeiten. Zu diesem Zweck befasst sich das Projekt mit der Entwicklung einer Task-Steuerung, über die Aufgaben für den Roboter direkt via Kalender erfasst werden können.

Als technologische Basis dient das Framework ROS (Robot Operating System). Dabei handelt es sich nicht um ein Betriebssystem, sondern vielmehr um ein Framework, das auf einem Publish-Subscribe-System basiert. ROS bietet eine Vielzahl vordefinierter Applikationen und Libraries, die den Aufbau komplexer Lösungen im Bereich der Robotik erheblich vereinfachen. Die in diesem Projekt erarbeitete Lösung wird in das bestehende ROS-System des Roboters integriert, wobei die Entwicklung in Form von ROS-Nodes und RViz Panel umgesetzt wird. Ein ROS-Node ist ein eigenständiger Prozess, der über Topics mit anderen ROS-Nodes kommunizieren kann. RViz ist ein Visualisierungs-Tool des ROS-Ökosystem zur Überwachung und Beobachtung des Systems, wie zum Beispiel die Kommunikation zwischen Nodes.

1.1.2 Detaillierte Aufgabenstellung

Die Arbeit umfasst im Wesentlichen die Konzeption und Entwicklung einer Applikation, welche Aufträge aus einem Kalender extrahiert, validiert und an den Roboter übermittelt. Die Umsetzung erfolgt mittels ROS-Nodes, die in Python und C++ geschrieben werden. Zu den Kernfunktionen gehören das Abrufen der Kalenderdaten, eine Machbarkeitsprüfung der Aufgaben, Interaktion mit der Roboter-Schnittstelle sowie eine Visualisierung der Aufgaben in einem RViz-Panel.

Datenerfassung und Schnittstellen

Die Erfassung der Aufgaben erfolgt über einen spezifischen Outlook-Kalender der FHNW (IMVS_Robotik). Der Kalender wird als Ressource zur Verfügung gestellt, sodass diese bei Terminerstellungen als «Raum» hinzugefügt werden kann. Dieser Mechanismus ermöglicht das programmatische Akzeptieren oder Ablehnen von Terminen via Microsoft Graph API. Da die Termine auf diese Weise personenbezogen sind, werden Rückmeldungen und Status-Updates an den Erfasser ermöglicht. Dies ist ein wichtiger Aspekt für das Feedback innerhalb des Aufgaben-Managements.

Die Kalendertermine basieren auf Freitextfeldern (Titel und Beschreibung). Hierfür definiert der Kandidat ein Format, um ein Parsing der Daten zu ermöglichen. Benutzer sollen Übermittlungs- und Transportaufträge erfassen können. Die konkrete Art und Weise dieser Erfassung sind Teil der zu entwickelnden Lösung. Um den Prozess für die Erfasser zu vereinfachen, sollen lediglich die Mitteilung und die Zielperson (bzw. bei Transporten zwei Personen) angegeben werden. Der Termin muss keine Informationen über den Standort der Personen beinhalten.

Datenhaltung und Validierung

Auf dem Roboter ist eine PostgreSQL-Datenbank in Betrieb, die Informationen wie Personenlisten, Arbeitsplätze und benannte Räumlichkeiten des Instituts enthält. Zu den Räumlichkeiten hinterlegte Koordinaten ermöglicht dem Roboter, das autonome fahren zu unterschiedlichen Orten im Institut. Dazu wird die Navigierungs- und Lokalisierungslösung von ROS2 verwendet. Für die Interaktion mit der Datenbank existiert ein Layer, der als API für Datenbankzugriffe fungiert. Dieser soll mit den für das Auftrags-Management nötigen Abfragen erweitert werden

Nach der Interpretation der Aufgaben aus dem Kalender soll die Applikation eine Validierung durchführen. Dabei wird nach einer vom Kandidaten entwickelten Strategie entschieden, welche Aufträge ausführbar sind und welche abgelehnt werden müssen. Eine Optimierung (z. B. auf maximale Aufgabenerfüllung) ist nicht gefordert.

Überwachung und Benutzeroberfläche

Zur Analyse und Überwachung des Systems wird ein RViz-Panel implementiert. Dieses ist in C++ unter Verwendung von Qt zu realisieren und lässt sich in das bestehende ROS2 RViz Tool als Erweiterung hinzufügen. Da RViz als eigenständiger ROS-Node ausgeführt wird, erfolgt der Datenaustausch über entsprechende ROS-Topics. Der Kandidat definiert hierfür die Schnittstellen und entwirft ein User Interface zur Anzeige der Tagesaufgaben. Über dieses Interface müssen Aufgaben manuell abgelehnt werden können. Zudem soll der aktuelle Status zu jeder Aufgabe ersichtlich sein.

Kommunikation mit dem Roboter

Um eine hardwareunabhängige und effiziente Entwicklung zu gewährleisten, entfällt die direkte Kommunikation mit dem physischen Roboter. Dem Kandidaten wird stattdessen eine Mock-Komponente zur Verfügung gestellt, welche das reale Robotersystem simuliert und die validierten Aufträge entgegennimmt. Der Kandidat entwickelt und testet seine Lösung gegen diese vordefinierte Schnittstelle. Dieser Ansatz ermöglicht ein vereinfachtes, isoliertes Monitoring der Funktionalität und verhindert Entwicklungsverzögerungen oder Wartezeiten, die durch reale Fahr- und Navigationszeiten des physischen Roboters entstehen würden.

Erwartete Resultate und prüfbare/ messbare Ziele

- Ein funktionsfähiger ROS-Node in Python implementiert ruft Termine via Graph API ab und extrahiert Entitäten (Art des Auftrags, Personen, Mitteilungen) aus Freitexten.
- Aufträge werden nach einer klar nachvollziehbaren und dokumentierten Strategie akzeptiert oder abgelehnt.
- Ausführbare Aufträge werden im Kalender akzeptiert, nicht ausführbare abgelehnt.
- Aufträge werden gegen die Stammdaten in der Datenbank validiert, sodass sie vom Roboter ausgeführt werden können.
- Erstellte Aufträge im Kalender werden bei Annahme innerhalb einer Minute im RViz-Panel angezeigt.
- Der Auftraggeber erhält bei Annahme/Ablehnung automatisiert eine E-Mail.
- Der Auftraggeber erhält bei erfolgreicher oder fehlgeschlagener Ausführung des Auftrags automatisiert eine E-Mail.
- Angenommene Aufträge werden an den Roboter (Mock) gesendet und dessen Feedback verarbeitet.
- Ein C++/Qt-basiertes RViz-Panel zeigt alle Tagesaufträge inklusive Status (pending, active, completed, failed) an.
- Über das RViz-Panel können Aufträge manuell storniert werden.
- Die Kommunikation zwischen den ROS-Nodes erfolgt mit sinnvoll gewählten Quality-of-Service-Attributen von ROS.

Tests / Qualitätssicherung

- Unit-Test für die Applikationslogik
- Integrationstest: Validierung des gesamten Datenflusses vom Outlook-Eintrag bis zur Anzeige im RViz-Panel
- Szenario-Test: Durchführung eines End-to-End Test ohne physischen Roboter

Dokumentationsanforderungen

- Anforderungsanalyse
- Beschreibung der Softwarearchitektur
- ROS-Architektur (Nodes, Topics, Services, QoS)
- Beschreibung der Parsing-Strategie und der definierten Validierung-Regeln
- Testprotokolle
- Installations- und Bedienungsanleitung für das System

1.2 Projektorganisation

Die IPA wird im Rahmen eines einjährigen Praktikums am Institut für Mobile und Verteilte Systeme (IMVS) der FHNW durchgeführt. Sie findet vor Ort im Büro in Windisch statt. Dies wird damit begründet, dass der Zugriff auf das Robotersystem ausschliesslich innerhalb des FHNW-Netzwerks möglich ist. Als Berufsbildnerin wird Frau Ursula Nohl (Hauptverantwortliche IMS, Kantonsschule Baden) eingesetzt.

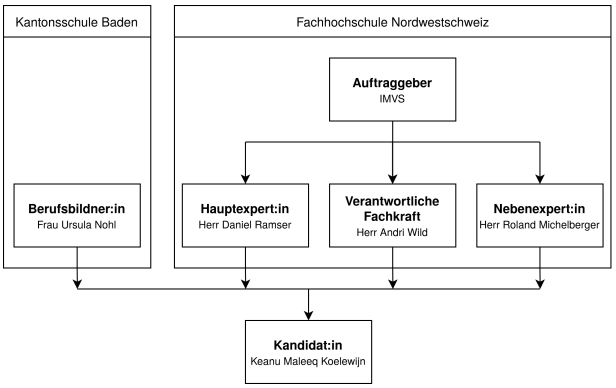


Abbildung 1: Organisationsstruktur der IPA

Als verantwortliche Fachkraft wird Herr Andri Wild bestimmt, durch den die fachliche Betreuung der Arbeit als Praxisbetreuer im Praktikum übernommen wird. Als Auftraggeber wird das IMVS in seiner Funktion als Betreiber des Roboterprojekts festgelegt. Für die korrekte Durchführung der IPA wird Herr Daniel Ramser als Hauptexperte eingesetzt, zudem wird durch ihn die Rolle des Prüfers übernommen. Als Nebenexperte wurde Herr Roland Michelberger zur Unterstützung und Mitbeurteilung hinzugezogen.

Rolle	**Person**	**Aufgaben**
Kandidat:in	Keanu Maleeq Koelewijn	Planung, Umsetzung, Tests und Dokumentation der IPA werden durchgeführt.
Berufsbildner:in	Ursula Nohl	Die übergeordnete Betreuung der Ausbildung sowie die organisatorische Unterstützung werden sichergestellt.
Hauptexpert:in	Daniel Ramser	Die Bewertung der IPA gemäss Vorgaben sowie die Durchführung von Expertenbesuchen werden übernommen.
Nebenexpert:in	Roland Michelberger	Die Unterstützung des Hauptexperten bei der Bewertung wird sichergestellt.
Verantwortliche Fachkraft	Andri Wild	Die Bewertung der IPA gemäss Vorgaben, die fachliche Betreuung, die Rückmeldung sowie die Klärung technischer Fragen werden übernommen.
Auftraggeber:in	IMVS	Die Definition der Anforderungen und Zielsetzung sowie die Bereitstellung der Infrastruktur werden vorgenommen.

Die Kommunikation zwischen dem Kandidaten, der verantwortlichen Fachkraft und dem Auftraggeber wird primär vor Ort sowie bei Bedarf oder bei Abwesenheit über Microsoft Teams oder per E-Mail geführt. Zur Gewährleistung der Nachvollziehbarkeit werden relevante Abmachungen und Entscheidungen mit Angabe von Datum, Beteiligten, Entscheidung und Konsequenz schriftlich im Arbeitsjournal festgehalten.

Für die IPA sind folgende fixen Termine definiert.

Termin	Datum	Beschreibung
Start der IPA	30.03.2026	Beginn der individuellen praktischen Arbeit
Expertenbesuch 1	30.03.2026	Besprechung der IPA, Besichtigung des Arbeitsplatzs und Klärung offener Fragen
Expertenbesuch 2	10.04.2026	Stand der Arbeit besichtigen
Abgabe der IPA	14.04.2026	Einreichung der Abschlussdokumentation und aller Arbeitsergebnisse
Abschlusspräsentation	21.04.2026	Präsentation der Ergebnisse

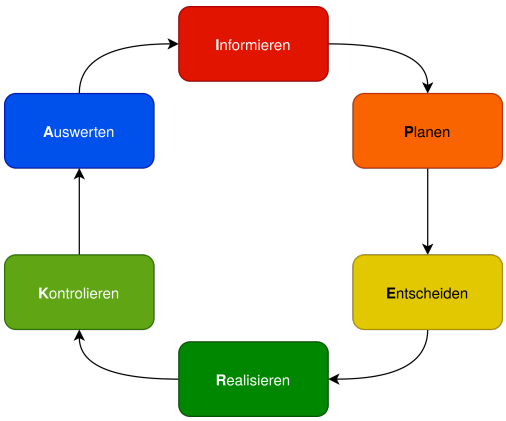


Abbildung 2: Visualisierung IPERKA

Für die Durchführung der IPA wurde die Projektmethode IPERKA gewählt. Da die Projektanforderungen sowie das Zielbild zu Beginn klar definiert sind, wird ein strukturiertes und phasenbasiertes Vorgehen ermöglicht. Mit IPERKA wird ein systematisches Vorgehen von der Analyse über die Planung und Entscheidungsfindung bis hin zur Umsetzung und Überprüfung gewährleistet. Zudem wird die individuelle praktische Arbeit innerhalb eines klar begrenzten Zeitrahmens durchgeführt. Ein lineares Vorgehensmodell wie IPERKA unterstützt eine realistische Zeitplanung und trägt zur Reduktion unnötiger Iterationen bei.

Das Projekt umfasst mehrere aufeinander aufbauende Komponenten, darunter Backend, ROS-Schnittstellen und eine Benutzeroberfläche. Mithilfe von IPERKA werden die bestehenden Abhängigkeiten strukturiert analysiert, sinnvoll geplant und schrittweise umgesetzt. Dadurch wird eine nachvollziehbare Arbeitsweise sichergestellt. Die Anwendung von IPERKA wird im weiteren Verlauf der Dokumentation durch die entsprechende Struktur der Kapitel sichtbar gemacht.

1.3 Deklaration Vorkenntnisse

Der Kandidat verfügt über Grundkenntnisse im ROS2-Framework, die er sich in den letzten Monaten angeeignet hat. Dazu gehören das Verständnis der Node-Kommunikation (Publisher/ Subscriber), der Umgang mit Services und Actions sowie die Nutzung der ROS2-CLI-Tools. Grundlegende Erfahrungen in Python und C++ sind vorhanden. Die Handhabung von Linux-Systemen (Ubuntu) und Git ist Bestandteil seiner täglichen Arbeitsroutine.

1.4 Deklaration Vorarbeiten

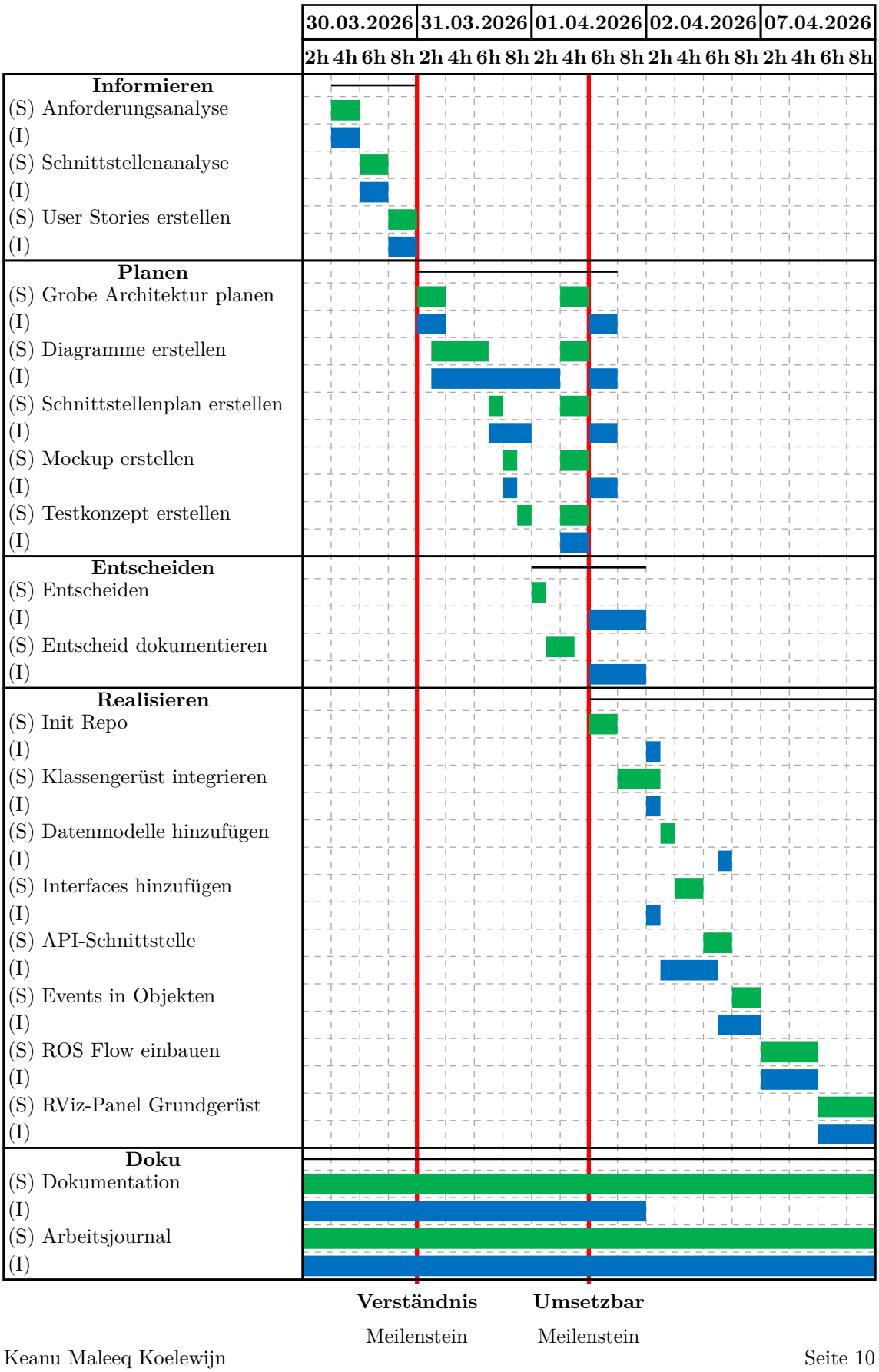
Damit der Gesamtumfang der Arbeit nicht zu gross wird, erhält der Kandidat einen vorgefertigten ROS-Node, der als Action Server fungiert und den Roboter-Mock bereitstellt.

1.5 Deklaration Firmenstandards

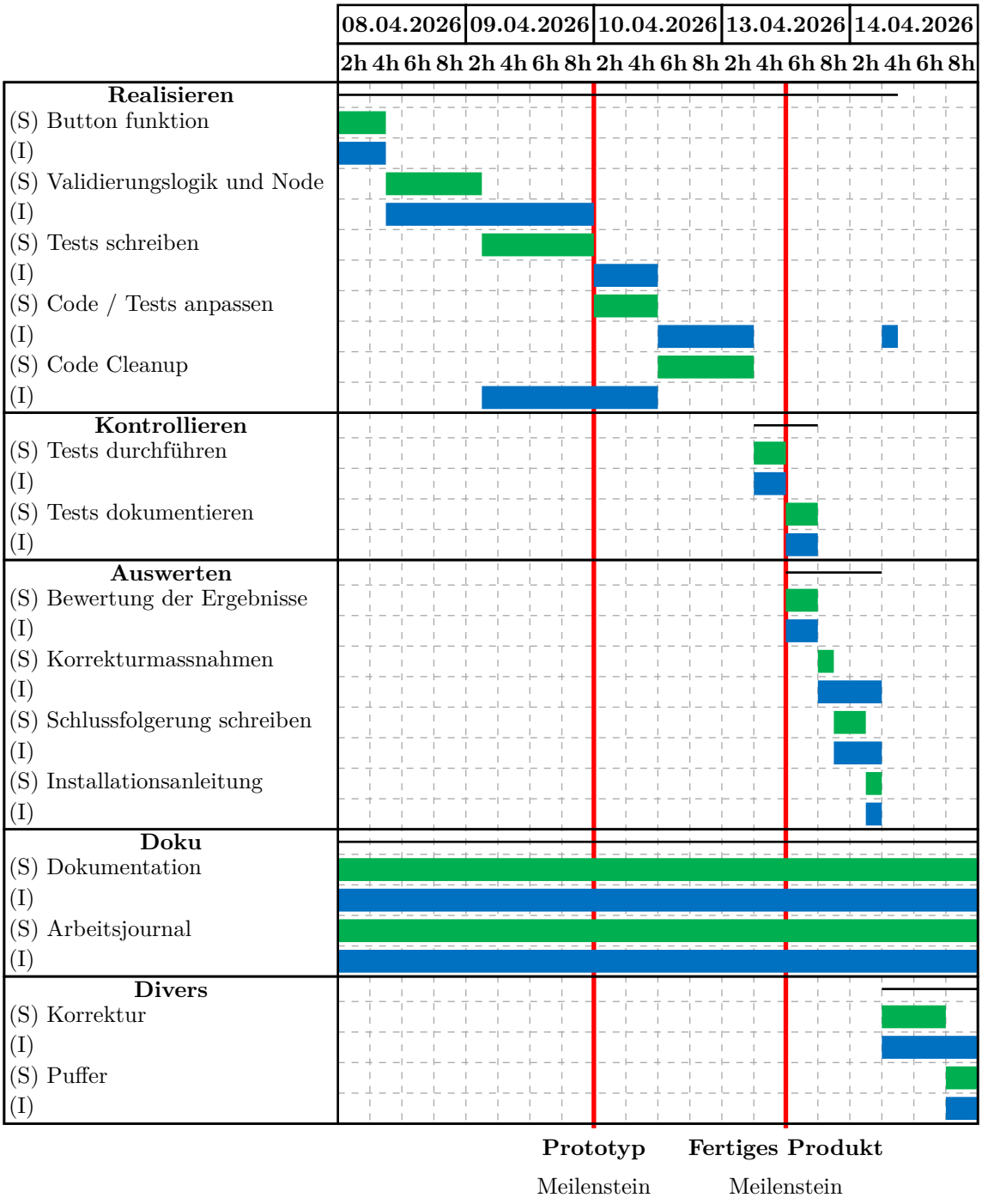
Die Coding Standards sollen gemäss dem bereitgestellten Dokument ‹Coderichtlinien› berücksichtigt werden, das in Anhang 2 zu finden ist.

1.6 Zeitplan

1.6.1 Zeitplan Tag 1-5



1.6.2 Zeitplan Tag 5-10



- Grün = Soll
- Blau = Ist
- Rot = Meilenstein

1.7 Arbeitsjournal

Arbeitsjournal 1

Datum:	30.03.2026
IPA-Tag:	1
Ort:	Windisch
Geplante Zeit:	08:00 - 12:00, 12:30 - 16:30
Effektiv:	08:25 - 11:45, 12:30 - 14:00, 15:00 - 18:10
Wochenendarbeit:	Nein

Tätigkeiten (Soll → Ist)

Geplant gemäss Zeitplan	Erledigt (Ist)	Ergebnis / Artefakt
Anforderungsanalyse	Anforderungsanalyse	«Informieren» der Dokumentation
Schnittstellenanalyse	Schnittstellenanalyse	«Informieren» der Dokumentation
User Stories erstellen	User Stories erstellen	«Informieren» der Dokumentation

Probleme, Risiken & Lösungen

Problem / Risiko	Ursache	Lösung / Massnahme	Status
Unklare Anforderungen zur Produktabgabe	Unsicherheit bezüglich der korrekten Abgabe	Nachfrage beim Hauptexperten	gelöst

Hilfestellungen, Abmachungen & Entscheide

Datum	Beteiligte	Hilfestellung / Abmachung / Entscheid	Konsequenz / Nutzen
30.03.2026	Ich, Hauptexperte	Termine für den zweiten und dritten Expertenbesuch grob abgestimmt	Klarheit über die weiteren Expertentermine
30.03.2026	Ich, Hauptexperte	Versand des Arbeitsjournals und des aktualisierten Zeitplans für den Folgetag vereinbart	Transparenz über Arbeitsstand und Planung
30.03.2026	Ich, Hauptexperte	Vorgehen für die ersten IPA-Tage abgestimmt	Klarheit über die nächsten Schritte
30.03.2026	Ich, Internet	Grundlagen zur Schnittstellenanalyse sowie zu Microsoft Graph, ROS2 Topics/ Services/QoS und RViz recherchiert	Fachliche Grundlage für die Anforderungs- und Schnittstellenanalyse geschaffen

Soll/Ist-Vergleich & Stundenübersicht

Block (2h Raster)	Soll (h)	Ist (h)	Abweichung	Grund
08:00 - 10:00	2:00	1:35	−0:25	Arbeitsbeginn erst um 08:25
10:00 - 12:00	2:00	1:45	−0:15	Arbeitsende vor 12:00
12:30 - 14:30	2:00	1:30	−0:30	Expertenbesuch von 14:00 bis 15:00
14:30 - 16:30	2:00	1:30	−0:30	Expertenbesuch von 14:00 bis 15:00
16:30 - 18:10	0:00	1:40	+1:40	Nachbearbeitung ausserhalb der geplanten Zeit
Total	8:00	8:00	0:00	Ich habe die Arbeitszeit durch Nachbearbeitung kompensiert

Reflexion

Was lief gut?	Ich habe die geplanten Tätigkeiten umgesetzt, offene Fragen mit dem Hauptexperten geklärt und den Meilenstein «Verständnis» erreicht.
Was lief nicht gut?	Ich musste zuerst eine geeignete Vorlage für die User Stories erstellen. Dadurch entstand zusätzlicher Aufwand.
Was ändere ich morgen?	Ich verwende eine feste Vorlage für die User Stories und sende den aktualisierten Zeitplan sowie das Arbeitsjournal frühzeitig.

Pendenzen

Ich sende morgen früh das Arbeitsjournal und den aktualisierten Zeitplan.

Arbeitsjournal 2

Datum:	31.03.2026
IPA-Tag:	2
Ort:	Windisch
Geplante Zeit:	09:00 - 12:00, 12:30 - 17:30
Effektiv:	09:00 - 12:00, 12:30 - 18:00
Wochenendarbeit:	Nein

Tätigkeiten (Soll → Ist)

Geplant gemäss Zeitplan	Erledigt (Ist)	Ergebnis / Artefakt
Grobe Architektur planen	Teilweise erledigt	«Planen» der Dokumentation
Diagramme erstellen	Teilweise erledigt	«Planen» der Dokumentation
Schnittstellenplan erstellen	Schnittstellenplan erstellen	«Planen» der Dokumentation

Probleme, Risiken & Lösungen

Problem / Risiko	Ursache	Lösung / Massnahme	Status
Höherer Planungsaufwand als erwartet	Der Aufwand wurde zu tief eingeschätzt	Planung priorisieren und verbleibende Punkte als Pendenzen übernehmen	teilweise gelöst

Hilfestellungen, Abmachungen & Entscheide

Datum	Beteiligte	Hilfestellung / Abmachung / Entscheid	Konsequenz / Nutzen
31.03.2026	Ich, verantwortliche Fachkraft	Feedback zur Grobarchitektur eingeholt	Eine zusätzliche Perspektive erhalten und neue Optionen erkannt
31.03.2026	Ich, Mitarbeitende	Feedback zum Mockup eingeholt	Hinweise zu Verbesserungsmöglichkeiten sowie Bestätigung gelungener Aspekte erhalten
31.03.2026	Ich, verantwortliche Fachkraft	Frage zum Klassendiagramm geklärt	Entscheid getroffen, statt eines klassischen Klassendiagramms eine ROS2-gerechte Darstellung zu verwenden

Soll/Ist-Vergleich & Stundenübersicht

Block (2h Raster)	Soll (h)	Ist (h)	Abweichung	Grund
09:00 - 11:00	2:00	2:00	0:00	-
11:00 - 12:00	1:00	1:00	0:00	-
12:30 - 14:30	2:00	2:00	0:00	-
14:30 - 16:30	2:00	2:00	0:00	-
16:30 - 17:30	1:00	1:00	0:00	-
17:30 - 18:00	0:00	0:30	+0:30	Nachbearbeitung aufgrund höheren Planungsaufwands
Total	8:00	8:30	+0:30	Zusätzlicher Aufwand für die Planung

Reflexion

Was lief gut?	Ich habe trotz hohem Planungsanteil die zentralen Planungsgrundlagen erstellt.
Was lief nicht gut?	Die durchgehende Planungsarbeit war kognitiv anspruchsvoll, wodurch meine Konzentration gegen Ende des Tages abnahm.
Was ändere ich morgen?	Wenn meine Konzentration sinkt, mache ich eine kurze aktive Pause und setze die Arbeit danach strukturiert fort.

Pendenzen

Klassendiagramm fertig planen
Mockup dokumentieren
Testkonzept und Testfälle schreiben

Arbeitsjournal 3

Datum:	01.04.2026
IPA-Tag:	3
Ort:	Windisch
Geplante Zeit:	08:00 - 12:00, 12:30 - 16:30
Effektiv:	08:00 - 11:30, 12:30 - 18:00
Wochenendarbeit:	Nein

Tätigkeiten (Soll → Ist)

Geplant gemäss Zeitplan	Erledigt (Ist)	Ergebnis / Artefakt
Grobe Architektur planen	Grobe Architektur planen	«Planen» der Dokumentation
Diagramme erstellen	Diagramme erstellen	«Planen» der Dokumentation
Mockup erstellen	Mockup erstellen	«Planen» der Dokumentation
Testkonzept erstellen	Testkonzept erstellen	«Planen» der Dokumentation
Entscheiden	Entscheiden	«Entscheiden» der Dokumentation
Entscheide dokumentieren	Entscheide dokumentieren	«Entscheiden» der Dokumentation
Repository initialisieren	Nicht erledigt	Pendenz für Tag 4
Klassengerüst integrieren	Nicht erledigt	Pendenz für Tag 4

Probleme, Risiken & Lösungen

Problem / Risiko	Ursache	Lösung / Massnahme	Status
Höherer Planungsaufwand als erwartet	Der Aufwand wurde zu tief eingeschätzt	Planung priorisieren und verbleibende Punkte als Pendenzen übernehmen	teilweise gelöst

Hilfestellungen, Abmachungen & Entscheide

Datum	Beteiligte	Hilfestellung / Abmachung / Entscheid	Konsequenz / Nutzen
01.04.2026	Ich, Auftraggeber	Feedback zu den User Stories eingeholt	Zusätzliche Perspektive erhalten und Hinweise zur weiteren Ausarbeitung gewonnen
01.04.2026	Ich, Auftraggeber	Umsetzungsvarianten abgestimmt	Bewertung der Varianten fachlich abgesichert
01.04.2026	Ich, Auftraggeber	Feedback zum Mockup eingeholt	Hinweise zu Verbesserungen sowie Bestätigung gelungener Aspekte erhalten
01.04.2026	Ich, Internet	Testkonzept-Grundlagen recherchiert	Testkonzept und Testfälle methodisch strukturiert
01.04.2026	Ich, Internet	Methode Präferenzmatrix nachgeschlagen	Nutzwertanalyse methodisch begründet

Soll/Ist-Vergleich & Stundenübersicht

Block (2h Raster)	Soll (h)	Ist (h)	Abweichung	Grund
08:00 - 10:00	2:00	2:00	0:00	-
10:00 - 12:00	2:00	1:30	−0:30	Arbeitsende vor 12:00
12:30 - 14:30	2:00	2:00	0:00	-
14:30 - 16:30	2:00	2:00	0:00	-
16:30 - 18:00	0:00	1:30	+1:30	Nachbearbeitung am Tagesende
Total	8:00	9:00	+1:00	Ich habe zusätzliche Nachbearbeitung geleistet

Reflexion

Was lief gut?	Ich habe einen weiteren Meilenstein erreicht und den Übergang in die Realisierungsphase vorbereitet. Die Anpassung vom Vortag hat meine Konzentration verbessert.
Was lief nicht gut?	Nach dem Feedback des Auftraggebers musste ich mehrere Dokumentationspunkte überarbeiten. Dadurch entstand zusätzlicher Zeitaufwand.
Was ändere ich morgen?	Ich behalte kurze Pausen bei, lege den Fokus auf die Implementierung und nutze freie Zeitfenster für den Feinschliff der Dokumentation.

Pendenzen

Repository initialisieren
Klassengerüst integrieren

Arbeitsjournal 4

Datum:	02.04.2026
IPA-Tag:	4
Ort:	Windisch
Geplante Zeit:	09:00 - 12:00, 12:30 - 17:30
Effektiv:	09:00 - 11:30, 12:30 - 18:00
Wochenendarbeit:	Nein

Tätigkeiten (Soll → Ist)

Geplant gemäss Zeitplan	Erledigt (Ist)	Ergebnis / Artefakt
Repository initialisieren	Repository initialisieren	Anhang 1
Klassengerüst integrieren	Klassengerüst integrieren	Anhang 1
Datenmodelle hinzufügen	Datenmodelle hinzufügen	Anhang 1
Interfaces hinzufügen	Interfaces hinzufügen	Anhang 1
Events in Objekten	Events in Objekten	Anhang 1

Probleme, Risiken & Lösungen

Problem / Risiko	Ursache	Lösung / Massnahme	Status
Keine	-	-	-

Hilfestellungen, Abmachungen & Entscheide

Datum	Beteiligte	Hilfestellung / Abmachung / Entscheid	Konsequenz / Nutzen
02.04.2026	Ich, verantwortliche Fachkraft	Frage zu den Codingrichtlinien für Docstrings geklärt	Ich ergänze Docstrings nur bei wichtigen oder nicht selbsterklärenden Funktionen und Klassen
02.04.2026	Ich, Internet	Microsoft Graph SDK-Authentifizierung und calendarView-Aufruf nachgeschlagen	GraphClientNode fachlich korrekt umgesetzt
02.04.2026	Ich, Internet	Python-Dataclasses und Parsing-Ansatz recherchiert	Task-Modell und Parser robuster aufgebaut

Soll/Ist-Vergleich & Stundenübersicht

Block (2h Raster)	Soll (h)	Ist (h)	Abweichung	Grund
09:00 - 11:00	2:00	2:00	0:00	-
11:00 - 12:00	1:00	0:30	−0:30	Arbeitsende vor 12:00
12:30 - 14:30	2:00	2:00	0:00	-
14:30 - 16:30	2:00	2:00	0:00	-
16:30 - 17:30	1:00	1:00	0:00	-
17:30 - 18:00	0:00	0:30	+0:30	Nachbearbeitung am Tagesende
Total	8:00	8:00	0:00	Ich habe die Abweichung vom Vormittag am Tagesende kompensiert

Reflexion

Was lief gut?	Ich habe den Rückstand im Zeitplan aufgeholt und durch kurze Konzentrationspausen einen stabilen Arbeitsfluss gehalten.
Was lief nicht gut?	Ich habe die Dokumentation nicht parallel zur Implementierung nachgeführt.
Was ändere ich morgen?	Ich plane feste Zeitblöcke für die Dokumentation ein und halte den Fokus weiterhin auf der Implementierung.

Pendenzen

Keine

Arbeitsjournal 5

Datum:	07.04.2026
IPA-Tag:	5
Ort:	Windisch
Geplante Zeit:	09:00 - 12:00, 12:30 - 17:30
Effektiv:	09:00 - 12:00, 12:15 - 18:00
Wochenendarbeit:	Nein

Tätigkeiten (Soll → Ist)

Geplant gemäss Zeitplan	Erledigt (Ist)	Ergebnis / Artefakt
ROS2-Flow einbauen	ROS2-Flow einbauen	Anhang 1
RViz-Panel-Grundgerüst	RViz-Panel-Grundgerüst	Anhang 1

Probleme, Risiken & Lösungen

Problem / Risiko	Ursache	Lösung / Massnahme	Status
Keine	-	-	-

Hilfestellungen, Abmachungen & Entscheide

Datum	Beteiligte	Hilfestellung / Abmachung / Entscheid	Konsequenz / Nutzen
07.04.2026	Ich, Internet	CMake-Konfiguration aus dem RViz-Custom-Panel-Tutorial geprüft	Relevante Punkte übernommen und trotz begrenzter C++/RViz-Erfahrung sauber umgesetzt
07.04.2026	Ich, Internet	Beispielcode `dial_panel.cpp` zur RViz-Pluginstruktur analysiert	Aufbau des eigenen Panels und Signalfluss klar umgesetzt
07.04.2026	Ich, Internet	C++ Header/Interface-Konzept nachgeschlagen	HPP/CPP-Trennung sauber strukturiert

Soll/Ist-Vergleich & Stundenübersicht

Block (2h Raster)	Soll (h)	Ist (h)	Abweichung	Grund
09:00 - 11:00	2:00	2:00	0:00	-
11:00 - 12:00	1:00	1:00	0:00	-
12:15 - 14:30	2:00	2:15	+0:15	Ich habe die Mittagspause verkürzt
14:30 - 16:30	2:00	2:00	0:00	-
16:30 - 17:30	1:00	1:00	0:00	-
17:30 - 18:00	0:00	0:30	+0:30	Begonnene Implementierung abgeschlossen
Total	8:00	8:45	+0:45	Ich habe einen begonnenen Implementierungsteil abgeschlossen

Reflexion

Was lief gut?	Ich habe den vorgesehenen Funktionsumfang umgesetzt und die Kernarbeiten termingerecht abgeschlossen.
Was lief nicht gut?	Ich habe weniger dokumentiert als geplant, weil die Fehlersuche am RViz-Panel mehr Zeit benötigt hat.
Was ändere ich morgen?	Ich dokumentiere morgen die umgesetzten Punkte direkt im Anschluss, damit ich den Realisieren-Teil später effizient verfeinern kann.

Pendenzen

Keine

Arbeitsjournal 6

Datum:	08.04.2026
IPA-Tag:	6
Ort:	Windisch
Geplante Zeit:	09:00 - 12:00, 12:30 - 17:30
Effektiv:	09:00 - 12:00, 12:30 - 17:30
Wochenendarbeit:	Nein

Tätigkeiten (Soll → Ist)

Geplant gemäss Zeitplan	Erledigt (Ist)	Ergebnis / Artefakt
Button-Funktion	Button-Funktion	Anhang 1
Validierungslogik und Node	Validierungslogik und Node	Anhang 1

Probleme, Risiken & Lösungen

Problem / Risiko	Ursache	Lösung / Massnahme	Status
Keine	-	-	-

Hilfestellungen, Abmachungen & Entscheide

Datum	Beteiligte	Hilfestellung / Abmachung / Entscheid	Konsequenz / Nutzen
08.04.2026	Ich, Internet	Python-`set`-Verhalten für Event-ID-Tracking nachgeschlagen	Duplikate in der Verarbeitung sicher verhindert
08.04.2026	Ich, Internet	`len()`-Prüfung für Quaternion-Länge nachgeschlagen	Ungültige Zieldaten zuverlässig abgefangen

Soll/Ist-Vergleich & Stundenübersicht

Block (2h Raster)	Soll (h)	Ist (h)	Abweichung	Grund
09:00 - 11:00	2:00	2:00	0:00	-
11:00 - 12:00	1:00	1:00	0:00	-
12:30 - 14:30	2:00	2:00	0:00	-
14:30 - 16:30	2:00	2:00	0:00	-
16:30 - 17:30	1:00	1:00	0:00	-
Total	8:00	8:00	0:00	Keine Abweichung

Reflexion

Was lief gut?	Ich habe heute fokussiert gearbeitet und die geplanten Implementierungsschritte umgesetzt.
Was lief nicht gut?	Ein zusätzlicher Node war im ursprünglichen Zeitplan nicht berücksichtigt. Ausserdem habe ich die Implementierung noch nicht ausreichend dokumentiert.
Was ändere ich morgen?	Ich schliesse den letzten Node ab, plane feste Zeit für die Dokumentation ein und erstelle Unit-Tests für zwei Nodes.

Pendenzen

Dokumentation des Realisieren-Teils ergänzen

Arbeitsjournal 7

Datum:	09.04.2026
IPA-Tag:	7
Ort:	Windisch
Geplante Zeit:	07:00 - 12:00, 12:30 - 15:30
Effektiv:	07:00 - 09:00, 09:30 - 12:00, 12:30 - 16:00
Wochenendarbeit:	Nein

Tätigkeiten (Soll → Ist)

Geplant gemäss Zeitplan	Erledigt (Ist)	Ergebnis / Artefakt
Validierungslogik und Node	Validierungslogik und Node	Anhang 1
Tests schreiben	Nicht erledigt	Pendenz für Tag 8

Probleme, Risiken & Lösungen

Problem / Risiko	Ursache	Lösung / Massnahme	Status
Task-Ausführungskomponente aufwändiger als erwartet	Im Zeitplan nicht explizit berücksichtigt	Ich plane Code-Cleanup parallel zur Umsetzung ein	offen

Hilfestellungen, Abmachungen & Entscheide

Datum	Beteiligte	Hilfestellung / Abmachung / Entscheid	Konsequenz / Nutzen
09.04.2026	Ich, verantwortliche Fachkraft	Designentscheidung im ValidationNode besprochen	Zusätzliche fachliche Perspektive erhalten und in der Umsetzung berücksichtigt
09.04.2026	Ich, Hauptexperte	Frage zum Dokumentumfang geklärt	Mit Begründung sind zusätzliche Seiten zulässig
09.04.2026	Ich, Hauptexperte	Ich habe mit dem Hauptexperten abgesprochen, dass die Dokumentation etwas länger ausfällt	Die erweiterte Dokumentation ist abgestimmt und wurde als in Ordnung bestätigt
09.04.2026	Ich, Internet	Microsoft Graph Endpunkte für accept/decline/delete geprüft	Statusänderungen und Stornierung korrekt an Outlook angebunden
09.04.2026	Ich, Internet	ROS2 Executors und Intervall-Überschneidungsprüfung nachgeschlagen	Asynchrone Rückmeldungen und Konflikterkennung stabilisiert

Soll/Ist-Vergleich & Stundenübersicht

Block (2h Raster)	Soll (h)	Ist (h)	Abweichung	Grund
07:00 - 09:00	2:00	2:00	0:00	-
09:00 - 11:00	2:00	1:30	−0:30	Arbeitsbeginn um 09:30
11:00 - 12:00	1:00	1:00	0:00	-
12:30 - 14:30	2:00	2:00	0:00	-
14:30 - 15:30	1:00	1:00	0:00	-
15:30 - 16:00	0:00	0:30	+0:30	Zusätzliche Nachbearbeitung
Total	8:00	8:00	0:00	Zeitabweichungen innerhalb des Tages ausgeglichen

Reflexion

Was lief gut?	Ich habe den letzten Node abgeschlossen und die Dokumentation weiter ergänzt.
Was lief nicht gut?	Ich liege weiterhin leicht hinter dem Zeitplan. Der sichtbare Fortschritt war geringer als geplant.
Was ändere ich morgen?	Ich setze die Dokumentation priorisiert fort und arbeite die offenen Punkte strukturiert ab, um den Rückstand zu reduzieren.

Pendenzen

Tests schreiben

Arbeitsjournal 8

Datum:	10.04.2026
IPA-Tag:	8
Ort:	Windisch
Geplante Zeit:	09:00 - 12:00, 12:30 - 17:30
Effektiv:	10:00 - 12:00, 12:30 - 19:00
Wochenendarbeit:	Nein

Tätigkeiten (Soll → Ist)

Geplant gemäss Zeitplan	Erledigt (Ist)	Ergebnis / Artefakt
Tests schreiben	Tests schreiben	Anhang 1
Code / Tests anpassen	Code / Tests anpassen	Anhang 1
Code-Cleanup	Nicht erledigt	Pendenz für Tag 9

Probleme, Risiken & Lösungen

Problem / Risiko	Ursache	Lösung / Massnahme	Status
Task-Ausführungskomponente aufwändiger als erwartet	Im Zeitplan nicht explizit berücksichtigt	Ich habe Code-Cleanup parallel zur Umsetzung durchgeführt	teilweise gelöst

Hilfestellungen, Abmachungen & Entscheide

Datum	Beteiligte	Hilfestellung / Abmachung / Entscheid	Konsequenz / Nutzen
10.04.2026	Ich, Internet	Microsoft Graph Endpunkte `event-update` und `user-sendMail` geprüft	Abschlussstatus und E-Mail-Rückmeldung zuverlässig umgesetzt

Soll/Ist-Vergleich & Stundenübersicht

Block (2h Raster)	Soll (h)	Ist (h)	Abweichung	Grund
09:00 - 11:00	2:00	1:00	−1:00	Späterer Arbeitsbeginn
11:00 - 13:00	1:30	1:30	0:00	-
13:00 - 15:00	2:00	2:00	0:00	-
15:00 - 17:00	2:00	2:00	0:00	-
17:00 - 19:00	0:30	2:00	+1:30	Zusätzliche Umsetzungs- und Dokumentationsarbeiten
Total	8:00	8:30	+0:30	Ich habe zusätzliche Umsetzungs- und Dokumentationsarbeiten ausgeführt

Reflexion

Was lief gut?	Ich habe den Code im Wesentlichen fertiggestellt und zusätzliche Tests ergänzt.
Was lief nicht gut?	Der Code-Cleanup ist noch nicht abgeschlossen. Zudem muss ich die Realisierung noch ausführlicher dokumentieren.
Was ändere ich morgen?	Ich erstelle morgen eine priorisierte Aufgabenliste, prüfe die offenen Punkte im Code und arbeite den Kriterienkatalog systematisch ab.

Pendenzen

Code-Cleanup abschliessen
Realisierung detaillierter dokumentieren

Arbeitsjournal 9

Datum:	13.04.2026
IPA-Tag:	9
Ort:	Windisch
Geplante Zeit:	08:00 - 12:00, 12:30 - 16:30
Effektiv:	08:00 - 12:00, 13:00 - 17:00
Wochenendarbeit:	Nein

Tätigkeiten (Soll → Ist)

Geplant gemäss Zeitplan	**Erledigt (Ist)**	**Ergebnis / Artefakt**
Code / Tests anpassen	Code / Tests anpassen	Anhang 1
Tests durchführen	Tests durchführen	Abschnitt Kontrollieren
Tests dokumentieren	Tests dokumentieren	Abschnitt Kontrollieren
Bewertung der Ergebnisse	Bewertung der Ergebnisse	Abschnitt Auswerten
Korrekturmassnahmen	Nicht erledigt	Pendenz für Tag 10
Schlussfolgerung schreiben	Nicht erledigt	Pendenz für Tag 10

Probleme, Risiken & Lösungen

Problem / Risiko	**Ursache**	**Lösung / Massnahme**	**Status**
Keine	-	-	-

Hilfestellungen, Abmachungen & Entscheide

Datum	**Beteiligte**	**Hilfestellung / Abmachung / Entscheid**	**Konsequenz / Nutzen**
13.04.2026	Ich	Ich habe keine Hilfestellungen benötigt und keine neuen Abmachungen getroffen	Kein zusätzlicher Abstimmungsbedarf

Soll/Ist-Vergleich & Stundenübersicht

Block (2h Raster)	**Soll (h)**	**Ist (h)**	**Abweichung**	**Grund**
08:00 - 10:00	2:00	2:00	0:00	-
10:00 - 12:00	2:00	2:00	0:00	-
12:30 - 14:30	2:00	1:30	−0:30	Längere Mittagspause
14:30 - 16:30	2:00	2:30	+0:30	Kompensation der Arbeitszeit
Total	8:00	8:00	0:00	Keine Abweichung

Reflexion

Was lief gut?	Ich bin gut vorangekommen und habe den Projektstand weitgehend abgeschlossen. Offen sind nur noch Abschlussprüfungen und Korrekturen.
Was lief nicht gut?	Ich habe Textpassagen gekürzt, um die Dokumentation zu straffen. Die inhaltliche Prüfung war dabei zeitaufwändig.
Was ändere ich morgen?	Ich arbeite die verbleibenden Punkte Schritt für Schritt ab und schliesse die Dokumentation kontrolliert ab.

Pendenzen

Korrekturmassnahmen
Schlussfolgerung schreiben

Arbeitsjournal 10

Datum:	14.04.2026
IPA-Tag:	10
Ort:	Windisch
Geplante Zeit:	09:30 - 12:00, 12:30 - 18:00
Effektiv:	06:00 - 12:00, 13:00 - 18:00
Wochenendarbeit:	Nein

Tätigkeiten (Soll → Ist)

Geplant gemäss Zeitplan	Erledigt (Ist)	Ergebnis / Artefakt
Korrekturmassnahmen	Korrekturmassnahmen	Abschnitt Auswerten
Code / Tests anpassen	Code / Tests anpassen	Anhang 1
Schlussfolgerung schreiben	Schlussfolgerung schreiben	Abschnitt Auswerten
Installationsanleitung	Installationsanleitung	Abschnitt Installationsanleitung

Probleme, Risiken & Lösungen

Problem / Risiko	Ursache	Lösung / Massnahme	Status
Keine	-	-	-

Hilfestellungen, Abmachungen & Entscheide

Datum	Beteiligte	Hilfestellung / Abmachung / Entscheid	Konsequenz / Nutzen
14.04.2026	Ich, Auftraggeber	Entscheid bestätigt, auf ein klassisches Klassendiagramm zu verzichten, da es im ROS2-Ökosystem wenig Mehrwert bietet	Die Dokumentation bleibt auf die ROS2-Komponenten und Schnittstellen fokussiert
14.04.2026	Ich	Ich habe keine zusätzlichen Hilfestellungen benötigt	Kein weiterer Abstimmungsbedarf

Soll/Ist-Vergleich & Stundenübersicht

Block (2h Raster)	Soll (h)	Ist (h)	Abweichung	Grund
06:00 - 08:00	0:00	2:00	+2:00	Ich habe früher begonnen, um zusätzliche Korrekturen umzusetzen
08:00 - 10:00	2:00	2:00	0:00	-
10:00 - 12:00	2:00	2:00	0:00	-
12:30 - 14:30	2:00	1:30	−0:30	Längere Mittagspause
14:30 - 16:30	2:00	2:00	0:00	-
16:30 - 18:00	0:00	1:30	+1:30	Zusätzliche Finalisierung vor der Abgabe
Total	8:00	11:00	+3:00	Ich habe zusätzliche Korrekturen und Finalisierungen durchgeführt

Reflexion

Was lief gut?	Ich habe die offenen Punkte abgeschlossen und den Endstand termingerecht finalisiert.
Was lief nicht gut?	Der Korrekturaufwand war höher als erwartet, wodurch zusätzlicher Zeitbedarf entstand.
Was ändere ich morgen?	Für kommende Projekte plane ich feste Korrekturfenster während der Umsetzung ein.

Pendenzen

Keine

1.8 Organisation der Arbeitsergebnisse

Die Arbeitsergebnisse werden in einer klaren und nachvollziehbaren Struktur organisiert. Quellcode, Konfigurationen, Testresultate und Projektdokumentation werden geordnet abgelegt und in dieser Dokumentation beschrieben. Die Codebasis wird mit dem Versionsverwaltungstool GitLab verwaltet und durch regelmässige Commits gesichert. Die Commit-Beschreibungen sind eindeutig formuliert, damit Änderungen jederzeit nachvollzogen werden können. Zusätzlich wird auf eine einheitliche und selbsterklärende Ordner- und Dateibenennung geachtet, um die Wartbarkeit und Übergabe der Arbeitsergebnisse sicherzustellen. Nach der täglichen Arbeit wird die aktuelle Version der Dokumentation auf einer externen SSD gespeichert.

1.9 Persönliches Fazit

Dieses Projekt hat mir sehr gefallen, besonders im Rahmen der IPA. Da ich bereits Vorerfahrungen mit den verschiedenen Technologien hatte, konnte ich mir von Anfang an ein gutes Bild davon machen, was ich wie umsetzen möchte. Das hat mir sehr geholfen. Auch das Vorgehen, die Programmierung Node für Node umzusetzen, hat für mich sehr gut funktioniert, da ich dadurch strukturiert arbeiten sowie einzelne Teile gezielt testen und verbessern konnte. Indem ich Dokumentationen, zum Beispiel zur Graph API, nochmals gelesen habe, konnte ich neue Möglichkeiten im SDK entdecken, die für meinen Anwendungsfall sehr hilfreich waren. Ich wusste vorher nicht, dass ich angeben kann, wie meine Daten sortiert werden sollen oder in welcher Zeitzone ich sie erhalten möchte. Dadurch musste ich weniger eigene Logik implementieren, was die Robustheit meiner Lösung verbessert hat. Zusätzlich habe ich in diesem Projekt den Einsatz von Sets in Python kennengelernt, was ich vorher so noch nicht kannte und was mir sehr geholfen hat. Obwohl ich in der Dokumentation gegen Ende vieles kürzen musste, um näher an die maximale Seitenzahl zu kommen, konnte ich sie dadurch lesefreundlicher machen, auch wenn vielleicht nicht mehr alle Details enthalten sind.

Besonders Schwierigkeiten hatte ich beim `RunTaskNode`, da ich nicht wusste, wie ich den Navigations-Mock-Status in den Auftragsstatus injizieren soll, sodass der Zustand «failed» auch einen konkreten Nutzen hat. Auch beim RViz-Qt-Plugin hatte ich anfangs grosse Schwierigkeiten, da ich mich mit Qt und C++ nicht besonders gut auskannte. Zwar wurde es einfacher, sobald ich mich eingearbeitet hatte, trotzdem fühle ich mich in diesem Bereich noch unsicher. Es gab Stellen, die ich zunächst übernommen habe, ohne sie vollständig erklären zu können, zum Beispiel `explicit TaskPanel(QWidget* parent = 0);`. Beim Dokumentieren habe ich jedoch gelernt, was manche Punkte bedeuten, und konnte mein Verständnis stärken. Bei der Dokumentation habe ich ausserdem gemerkt, wie wichtig es ist, sie immer aktuell zu halten. Da ich Änderungen nicht immer direkt nachgeführt habe, entstand am Ende viel zusätzlicher Aufwand und Druck. Zudem war es schwierig, in der Dokumentation die richtige Balance zwischen Verständlichkeit und Kürze zu finden.

In Zukunft würde ich die Dokumentation jeweils direkt aktualisieren, sobald ich eine Änderung an der Umsetzung vornehme. Das nächste Mal würde ich im Rahmen einer IPA ausserdem mehr Zeit für die Dokumentation einplanen und wenn möglich ein etwas kleineres Projekt wählen. Insgesamt beurteile ich das Projekt als gelungen, auch wenn der Umfang gegen Ende etwas zu gross war und die Dokumentation mehr Aufwand verursachte als erwartet.

2 Projektdokumentation

2.1 Kurzfassung

2.1.1 Ausgangslage

Im Rahmen dieser Arbeit wurde eine Lösung entwickelt, mit der Aufträge für einen autonomen Assistenzroboter direkt über den bestehenden Outlook-Ressourcenkalender erfasst werden können. Ziel war es, die Nutzung des Roboters für Institutsmitarbeitende deutlich zu vereinfachen und den bislang fehlenden Prozess von der Auftragserfassung bis zur Ausführung und Überwachung durchgängig zu automatisieren.

2.1.2 Vorgehen

Die umgesetzte Lösung liest Kalendereinträge über die Microsoft Graph API aus, extrahiert die relevanten Angaben aus dem Freitext, validiert die Aufträge anhand definierter Regeln und vorhandener Stammdaten und übergibt ausführbare Aufträge an das Navigations-Mock des Robotersystems. Gleichzeitig werden Aufträge und Statusänderungen in einem RViz-Panel visualisiert. Zudem können Aufträge dort manuell storniert werden. Rückmeldungen zu Annahme, Ablehnung sowie Ausführung werden automatisiert an die auftragserfassende Person zurückgegeben.

Für die Umsetzung wurde eine modulare ROS2-Architektur mit klar getrennten Verantwortlichkeiten gewählt. Dadurch konnten Schnittstellen, Validierung, Ausführung und Visualisierung sauber voneinander getrennt werden. Für das Parsing wurde bewusst ein deterministischer Schlüssel-Wert-Ansatz eingesetzt, da dieser im gegebenen Projektumfang eine hohe Nachvollziehbarkeit, zuverlässige Fehlerbehandlung und gute Testbarkeit ermöglicht.

2.1.3 Ergebnis

Das Ergebnis ist eine funktionsfähige End-to-End-Lösung vom Outlook-Termin bis zur Anzeige im Betriebssystem des Roboters. Die durchgeführten Unit-, Integrations- und Szenariotests zeigen, dass die zentralen Kernfunktionen im geprüften Umfang erfolgreich umgesetzt wurden. Damit wurde das Projektziel erreicht und eine praxistaugliche Grundlage geschaffen, um den Roboter im Institutsalltag einfacher, transparenter und strukturierter nutzbar zu machen.

2.2 Informieren

Zu Beginn des Projekts werden die fachlichen Anforderungen, technischen Rahmenbedingungen und relevanten Schnittstellen geklärt. Da die Lösung Kalenderdaten verarbeitet, validiert und in ein bestehendes Robotersystem integriert, müssen vor allem Datenfluss, Systemgrenzen und Risiken frühzeitig verstanden werden.

2.2.1 Anforderungsanalyse

Die aus Abschnitt 1.1.2 abgeleiteten Anforderungen werden in funktionale Anforderungen, nichtfunktionale Anforderungen und Randbedingungen gegliedert. Zusätzlich wird ausgewiesen, ob eine Anforderung direkt durch die Softwarelösung umgesetzt wird oder als übergeordnete Vorgabe zu berücksichtigen ist.

Das Mindmap soll den ersten Überblick über die Anforderungen geben. Die Tabelle hält anschliessend Funktionsumfang, Qualitätsmerkmale und Rahmenbedingungen fest.

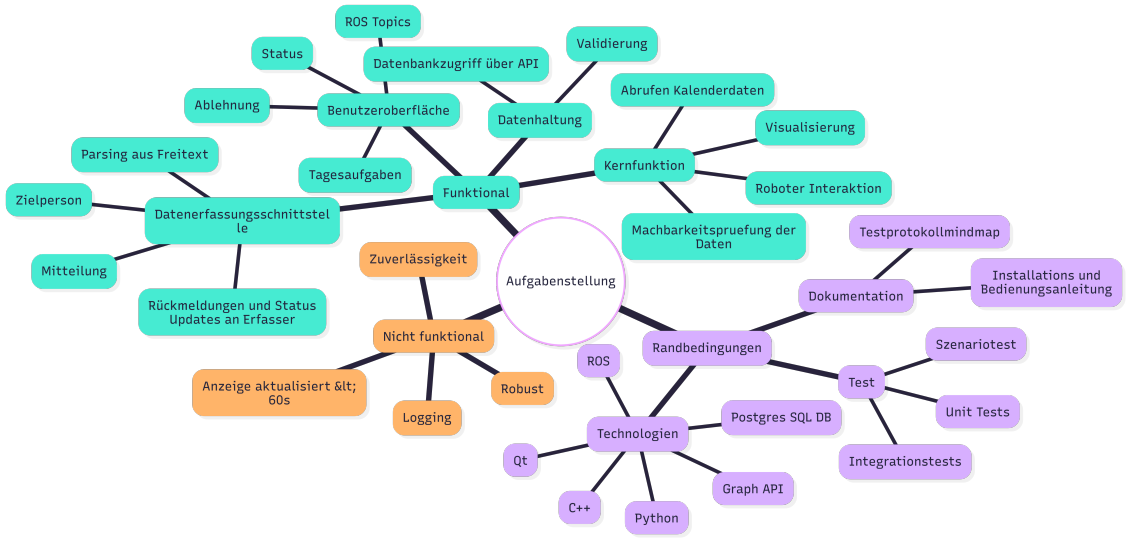


Abbildung 3: Mindmap der Anforderungen

Nr.	Typ	Anforderung	Teil der Lösung
A01	Funktional	Die Lösung muss Kalenderdaten abrufen.	Ja
A02	Funktional	Die Lösung muss abgerufene Daten auf Machbarkeit validieren.	Ja
A03	Funktional	Die Lösung muss validierte Aufträge an die Roboterschnittstelle übermitteln.	Ja
A04	Funktional	Die Lösung muss validierte Aufträge visualisieren.	Ja
A05	Funktional	Die Lösung muss Rückmeldungen und Status-Updates an den Erfasser des Kalendereintrags bereitstellen.	Ja
A06	Funktional	Die Lösung muss Freitext in verarbeitbare Daten parsen. Dabei müssen mindestens eine Mitteilung und eine Zielperson erfasst werden.	Ja

Nr.	Typ	Anforderung	Teil der Lösung
A07	Funktional	Die Lösung muss für Datenbankzugriffe die interne API verwenden.	Ja
A08	Funktional	Die Lösung muss eingegebene Daten gegen die vorhandenen Stammdaten validieren.	Ja
A09	Funktional	Die Lösung muss Tagesaufgaben in der Benutzeroberfläche anzeigen.	Ja
A10	Funktional	Die Lösung muss Status-Updates in der Benutzeroberfläche anzeigen.	Ja
A11	Funktional	Die Lösung muss ausführbare Aufträge im Kalender akzeptieren.	Ja
A12	Funktional	Die Lösung muss nicht ausführbare Aufträge im Kalender ablehnen.	Ja
A13	Funktional	Die Lösung muss den Auftragsersteller bei Annahme oder Ablehnung automatisch benachrichtigen.	Ja
A14	Funktional	Die Lösung muss den Auftragsersteller nach der Ausführung automatisch über Erfolg oder Fehlschlag informieren.	Ja
A15	Funktional	Die Lösung muss das manuelle Stornieren von Aufträgen über das `RViz-Panel` ermöglichen.	Ja
A16	Funktional	Die Lösung muss bei Transportaufträgen zwei Zielpersonen verarbeiten können.	Ja
A17	Nichtfunktional	Die Lösung muss fehlerhafte oder unvollständige Kalendereinträge behandeln, ohne abzustürzen.	Ja
A18	Nichtfunktional	Die Lösung muss relevante Ereignisse und Fehler protokollieren.	Ja
A19	Nichtfunktional	Die Lösung muss eine zuverlässige Verarbeitung und Übertragung von Aufträgen gewährleisten.	Ja
A20	Nichtfunktional	Die Lösung muss neue Kalendereinträge innerhalb von 60 Sekunden verarbeiten und anzeigen.	Ja
A21	Randbedingung	Die Lösung muss mittels Szenariotests validiert werden.	Ja
A22	Randbedingung	Die Lösung muss mittels Integrationstests validiert werden.	Ja
A23	Randbedingung	Die Lösung muss mittels Unit-Tests validiert werden.	Ja
A24	Randbedingung	Die Lösung muss die vorhandene PostgreSQL-Datenbank einbinden.	Ja
A25	Randbedingung	Die Lösung muss ROS2 verwenden.	Ja

Nr.	Typ	Anforderung	Teil der Lösung
A26	Randbedingung	Die Lösung muss in Python umgesetzt werden.	Ja
A27	Randbedingung	Die Benutzeroberfläche muss in C++ mit Qt implementiert werden.	Ja
A28	Randbedingung	Die Lösung muss das bereitgestellte Navigations-Mock zur Simulation der Roboterkomponenten verwenden.	Ja
A29	Randbedingung	Die Lösung muss die Visualisierung im `RViz-Panel` umsetzen.	Ja
A30	Randbedingung	Die Lösung muss die Microsoft Graph API verwenden.	Ja
A31	Randbedingung	Die Kommunikation zwischen den `ROS2-Nodes` muss mit geeigneten QoS-Einstellungen umgesetzt werden.	Ja
A32	Randbedingung	Die Dokumentation muss die Validierungsregeln beschreiben.	Nein
A33	Randbedingung	Die Dokumentation muss eine Anforderungsanalyse enthalten.	Nein
A34	Randbedingung	Die Dokumentation muss eine Beschreibung der Softwarearchitektur enthalten.	Nein
A35	Randbedingung	Die Dokumentation muss eine Beschreibung der ROS2-Architektur enthalten.	Nein
A36	Randbedingung	Die Dokumentation muss eine Installations- und Bedienungsanleitung enthalten.	Nein
A37	Randbedingung	Die Dokumentation muss ein Testprotokoll enthalten.	Nein
A38	Randbedingung	Die Dokumentation muss die Parsing-Strategie der Freitexte beschreiben.	Nein

2.2.2 Schnittstellenanalyse

Die beteiligten Systeme und APIs werden als Mindmap zusammengefasst und anschliessend tabellarisch beschrieben. Erfasst werden Daten, Trigger, technische Umsetzung und Risiken.

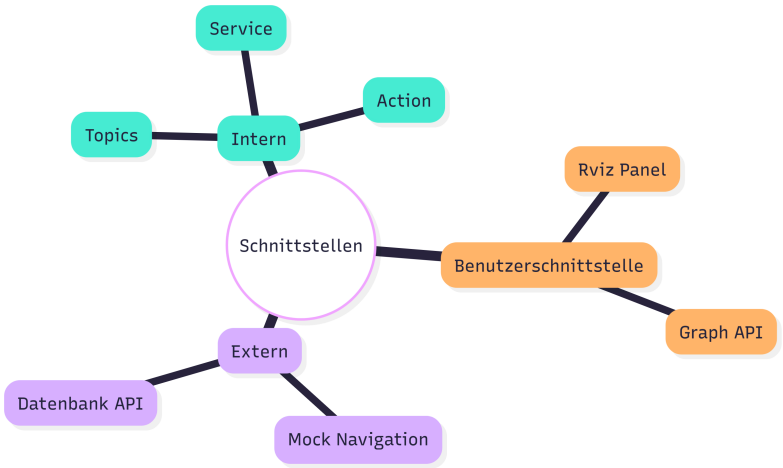


Abbildung 4: Mindmap mit Schnittstellen

Für das Projekt sind insbesondere die Microsoft Graph API, ROS2 und RViz relevant. Die Microsoft Graph API dient als Schnittstelle zu den Microsoft-365-Diensten und ermöglicht in diesem Projekt den Zugriff auf Kalenderdaten sowie die Verarbeitung von Rückmeldungen im Zusammenhang mit Kalendereinträgen[1,2]. ROS2 bildet die technische Grundlage für die Kommunikation zwischen den beteiligten Softwarekomponenten des Robotersystems. Der Datenaustausch erfolgt dabei über Topics und Services. Topics eignen sich für die asynchrone Übertragung von Nachrichten, während Services für direkte Anfrage-Antwort-Kommunikation verwendet werden[3]. Die Quality-of-Service-Einstellungen beeinflussen dabei, wie zuverlässig und unter welchen Bedingungen Nachrichten übertragen werden[4]. Das RViz-Panel dient als Benutzeroberfläche innerhalb des ROS2-Ökosystems und ermöglicht die Darstellung sowie die manuelle Interaktion mit dem System[5].

Die Struktur der Schnittstellentabellen orientiert sich an einer Vorlage zur Schnittstellenanalyse[6].

Aspekt	Beschreibung
Schnittstelle	Navigations-Mock
Beteiligte	Lösung, Navigations-Mock
Art der Daten	Zielkoordinaten (x, y, z), Auftragsstatus, Rückmeldestatus, verbleibende Entfernung
Austauschformat	`ROS2-Action`
Trigger	`Action Request` durch die Lösung
Frequenz	on demand
Risiken	Roboter erreicht das Ziel nicht oder verbleibt in einem Fehlerzustand

Das bereitgestellte Navigations-Mock simuliert die Schnittstelle zum späteren Navigationssystem. Es empfängt Aufträge in Form von `ROS2-Action-Requests` mit den Zielkoordinaten als Anfrage. Als Antwort liefert das Navigations-Mock den aktuellen Bearbeitungsstatus und die verbleibende Entfernung bis zum Ziel zurück.

Im realen Einsatz besteht das Risiko, dass der Roboter das Ziel nicht erreicht und in einem Fehlerzustand bleibt.

Um die benötigten Informationen im System verfügbar zu machen, muss die Datenbank angebunden werden. Aus Sicht der Lösung wird sie als externe Abhängigkeit betrachtet, auch wenn sie auf dem Robotersystem intern betrieben wird. Die Schnittstelle zur Datenbank ist über eine API vorhanden, die von der Auftragsverarbeitung genutzt wird. Bei Bedarf müssen benötigte Punkte selbst hinzugefügt werden. Über diese API müssen Daten nur abgerufen werden.

Diese Schnittstelle ist notwendig, um Personen und Koordinaten für die Verarbeitung bereitzustellen. Risiken bestehen, wenn die Datenbank nicht erreichbar ist oder ungültige beziehungsweise inkonsistente Daten zurückliefert.

Aspekt	Beschreibung
Schnittstelle	Datenbank-API
Beteiligte	Lösung, interne API, Datenbank
Art der Daten	personen, koordinaten
Austauschformat	interne API-Abfrage
Trigger	Anfrage durch die Lösung
Frequenz	on demand
Risiken	Datenbank nicht erreichbar, ungültige oder inkonsistente Daten

Aspekt	Beschreibung
Schnittstelle	Microsoft Graph API
Beteiligte	Lösung, Microsoft Graph API, Microsoft-365-Dienste
Art der Daten	Benutzerdaten, Kalendereignisse, E-Mail-Inhalte
Austauschformat	REST API
Trigger	Anfrage durch die Lösung
Frequenz	on demand
Risiken	Authentifizierungsfehler, ungültige Anfragen, API nicht erreichbar oder unzuverlässig

Eine zentrale externe Schnittstelle ist die Microsoft Graph API. Über sie können Kalendereinträge abgerufen, angenommen, abgelehnt oder storniert werden. Mit den eingerichteten Berechtigungen können ausserdem E-Mails erstellt und versendet werden. Der Zugriff erfolgt über das Microsoft Graph SDK, welches die Kommunikation mit der REST-API abstrahiert. Die entsprechenden Aufrufe werden durch die Auftragsverarbeitung ausgelöst.

Risiken bestehen in Authentifizierungsfehlern, fehlerhaften Anfragen oder in einer eingeschränkten Zuverlässigkeit der API.

Die Benutzeroberfläche im `RViz-Panel` stellt die Schnittstelle zum Nutzer dar. Sie empfängt Auftragsdaten von der Auftragsverarbeitung und stellt diese dar. Zusätzlich können über die Benutzeroberfläche Aufträge manuell storniert werden.

Risiken bestehen darin, dass die Benutzeroberfläche nicht erreichbar ist, fehlerhaft funktioniert oder Informationen aus vorherigen Schnittstellen nicht korrekt übernimmt. In diesem Fall können falsche oder unvollständige Informationen angezeigt werden.

Aspekt	Beschreibung
Schnittstelle	`RViz-Panel`
Beteiligte	`RViz-Panel`, Lösung, Nutzer
Art der Daten	Auftragsdaten, Statusinformationen, manuelle Steuerbefehle
Austauschformat	`ROS2-Nachrichten` als `Topics` und `Service-Aufrufe`
Trigger	Benutzerinteraktion oder Statusänderung
Frequenz	eventbasiert
Risiken	Keine oder ungültige Daten, fehlerhafte Anzeige, Synchronisationsprobleme zwischen `ROS2-Nodes`

2.2.3 User Stories

Mit den User Stories werden die Anforderungen aus Abschnitt 2.2.1 in überprüfbare Anwendungsfälle aus Sicht der Rollen Benutzer und Verwalter überführt. Mit der Benutzerrolle wird die Erfassung von Aufträgen über den Kalender beschrieben, während mit der Verwalterrolle die Überwachung und manuelle Stornierung über das `RViz-Panel` abgedeckt wird.

US-1

Verbindlichkeit:	Muss
Typ:	Funktional
User Story:	Als Benutzer soll ein Übermittlungs- oder Transportauftrag über einen Kalendereintrag erfasst werden können, damit der Roboter einfach aus dem gewohnten Arbeitsablauf heraus beauftragt werden kann.
Akzeptanzkriterien:	• Ein Auftrag kann über einen Kalendereintrag erstellt werden. • Der Kalendereintrag wird vom System erkannt und eingelesen.
Aufwand:	Gross
Notizen:	Technische Umsetzung über Microsoft Graph API. Erfassung ausschliesslich über den Outlook-Ressourcenkalender. Neue Einträge sollen innerhalb von 60 Sekunden erkannt werden.

US-2

Verbindlichkeit:	Muss
Typ:	Funktional
User Story:	Als Benutzer soll ein Übermittlungsauftrag als Freitext eingegeben werden können, damit Zielperson, Auftrag und Mitteilung flexibel formuliert werden können.
Akzeptanzkriterien:	• Der Freitext wird vom System verarbeitet. • Eine Mitteilung und eine Zielpersonen werden erkannt. • Unvollständige oder fehlerhafte Eingaben führen zu einer Ablehnung des Auftrags mit einer Rückmeldung.
Aufwand:	Sehr gross
Notizen:	Das Freitextformat wird im Projekt definiert und in Titel oder Beschreibung des Kalendereintrags verwendet. Fehlerhafte oder unvollständige Eingaben müssen robust behandelt und protokolliert werden. Transportaufträge mit zwei Zielpersonen sind zu unterstützen.

US-3

Verbindlichkeit:	Muss
Typ:	Funktional
User Story:	Als Benutzer soll ein Transportauftrag als Freitext eingegeben werden können, damit Zielperson, Auftrag und Mitteilung flexibel formuliert werden können.
Akzeptanzkriterien:	• Der Freitext wird vom System verarbeitet. • Eine Mitteilung und zwei Zielpersonen werden erkannt. • Unvollständige oder fehlerhafte Eingaben führen zu einer Ablehnung des Auftrags mit einer Rückmeldung.
Aufwand:	Sehr gross
Notizen:	Das Freitextformat wird im Projekt definiert und in Titel oder Beschreibung des Kalendereintrags verwendet. Fehlerhafte oder unvollständige Eingaben müssen robust behandelt und protokolliert werden. Transportaufträge mit zwei Zielpersonen sind zu unterstützen.

US-4

Verbindlichkeit:	Muss
Typ:	Funktional
User Story:	Als Benutzer soll ein Auftrag automatisch auf Korrektheit und Machbarkeit geprüft werden, damit nur gültige Aufträge ausgeführt werden.
Akzeptanzkriterien:	• Eingaben werden gegen vorhandene Daten geprüft. • Die Machbarkeit wird validiert. • Bei Aufträge wird geprüft, ob die erforderlichen Zielpersonen vollständig vorhanden sind. • Ungültige Aufträge werden mit Begründung abgelehnt. • Gültige Aufträge werden zur Ausführung an die Roboterschnittstelle übergeben.
Aufwand:	Sehr gross
Notizen:	Die Validierung erfolgt mithilfe der Daten über die interne API auf Basis der vorhandenen PostgreSQL-Datenbank. Ausführbare Aufträge werden über ROS2 an das Navigations-Mock übergeben. Dabei sind geeignete QoS-Einstellungen zu berücksichtigen.

US-5

Verbindlichkeit:	Muss
Typ:	Funktional
User Story:	Als Benutzer soll bei relevanten Statusänderungen eines Auftrags automatisch informiert werden, damit der aktuelle Stand der Anfrage jederzeit nachvollzogen werden kann.
Akzeptanzkriterien:	• Nach der Prüfung erhält der Auftragsersteller eine Rückmeldung. • Die Rückmeldung enthält den Status angenommen oder abgelehnt. • Bei einer Ablehnung ist für den Benutzer erkennbar, dass der Auftrag nicht weiterverarbeitet wird und eine Begründung. • Bei weiteren relevanten Statusänderungen erhält der Benutzer automatisch eine Rückmeldung. • Nach der Ausführung erhält der Auftragsersteller automatisch eine Rückmeldung über Erfolg oder Fehlschlag. • Die Rückmeldung erfolgt automatisch.
Aufwand:	Sehr gross
Notizen:	Rückmeldungen und Statusänderungen werden automatisiert an den Auftragsersteller übermittelt. Die Verarbeitung muss zuverlässig erfolgen. Annahme, Ablehnung sowie Erfolg oder Fehlschlag der Ausführung sind abzudecken.

US-6

Verbindlichkeit:	Muss
Typ:	Funktional
User Story:	Als Verwalter sollen die heutigen Aufträge und deren Status in der Benutzeroberfläche angezeigt werden, damit alle laufenden und geplanten Aufträge zentral überwacht werden können.
Akzeptanzkriterien:	• Tagesaufgaben werden in der Benutzeroberfläche angezeigt. • Validierte Aufträge werden visualisiert. • Status-Updates zu Aufträgen werden in der Benutzeroberfläche angezeigt. • Der Status eines Auftrags ist für den Verwalter klar erkennbar.
Aufwand:	Gross
Notizen:	Die Benutzeroberfläche wird als `RViz-Panel` in C++ mit Qt umgesetzt. Angezeigt werden Tagesaufträge, validierte Aufträge und deren Status. Neue Einträge sollen innerhalb von 60 Sekunden sichtbar werden.

US-7

Verbindlichkeit:	Muss
Typ:	Funktional
User Story:	Als Verwalter sollen Aufträge bei Bedarf manuell storniert werden können, damit nicht mehr gewünschte Aufträge nicht weiter ausgeführt werden.
Akzeptanzkriterien:	• Bereits erfasste Aufträge können über das `RViz-Panel` manuell storniert werden. • Nach einer Stornierung wird der Auftrag nicht weiter ausgeführt. • Der geänderte Status ist in der Benutzeroberfläche sichtbar.
Aufwand:	Sehr gross
Notizen:	Manuelle Eingriffe im `RViz-Panel` beschränken sich auf Stornierungen. Statusanzeige, weitere Verarbeitung und Rückmeldungen müssen konsistent aktualisiert werden. Relevante Ereignisse und Fehler sind zu protokollieren.

Die User Stories wurden auf Grundlage des Feedbacks des Auftraggebers überarbeitet und anschliessend freigegeben.

Datum	Person	Input/Vorgabe	Umgesetzte Massnahme	Resultat
01.04.2026	Andri Wild in der Rolle des Auftraggebers	Die Auftragsart soll in US-Nr. 1 präzisiert werden.	US-Nr. 1 wurde entsprechend angepasst.	Bearbeitet
01.04.2026	Andri Wild in der Rolle des Auftraggebers	US-Nr. 2 soll nach Auftragsart in zwei User Stories aufgeteilt werden.	US-Nr. 2 wurde in US-Nr. 2 und US-Nr. 3 aufgeteilt. Zusätzlich wurden die Testfälle angepasst.	Bearbeitet
01.04.2026	Andri Wild in der Rolle des Auftraggebers	Die Aufwandsgrösse soll vorzugsweise in Textform statt mit Kleidergrössen angegeben werden.	Die Spalte Aufwand aller User Stories wurde entsprechend überarbeitet.	Bearbeitet

2.3 Planen

In der Planungsphase werden die Anforderungen aus Abschnitt 2.2.1 in Systemarchitektur, Prozesslogik und Bedienoberfläche konkretisiert.

2.3.1 Architekturübersicht

Die Architektur wurde auf Grundlage von Abschnitt 2.2.1 und Abschnitt 2.2.2 so geplant, dass die Komponenten lose gekoppelt sind und jeweils eine klar abgegrenzte Verantwortung übernehmen. Die Struktur folgt dem Prinzip `Separation of Concerns` und dem `Single Responsibility Principle`. Dadurch entsteht eine geringe Kopplung bei hoher Kohäsion.

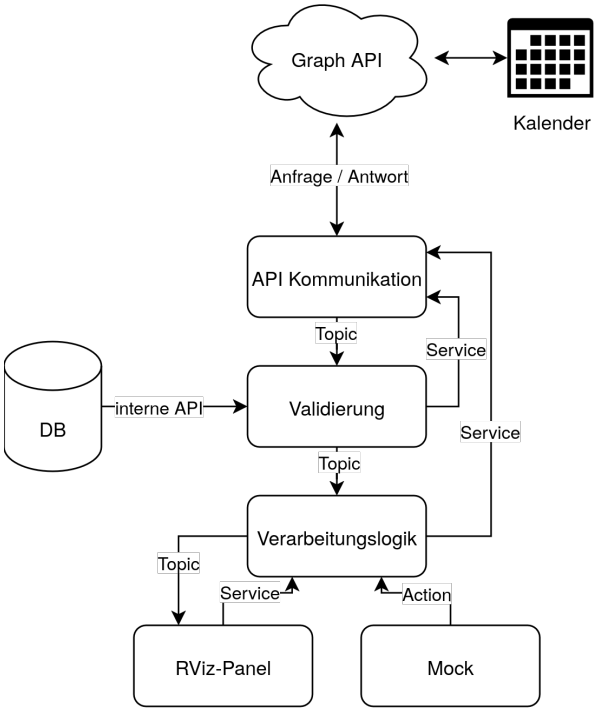


Abbildung 5: Grobarchitektur auf Systemebene.

Der `GraphClientNode` fungiert als Adapter zur Microsoft Graph API und bildet die Schicht gegenüber dem externen System. Eingehende Anfragen werden anschliessend durch die Validierungslogik geprüft. Ungültige oder nicht weiterverarbeitbare Anfragen werden frühzeitig erkannt. Gültige Anfragen werden dort mit den Personendaten aus der Datenbank ergänzt.

Die Verarbeitungslogik sendet den Auftrag abhängig von der Auftragszeit an das Navigations-Mock und stellt den aktuellen Auftragszustand für das `RViz-Panel` bereit. Danach werden die Aufträge im `RViz-Panel` angezeigt. Rückmeldungen aus den beteiligten Komponenten werden über einen zentralen `ROS2-Service` an die Graph API zurückgeleitet.

2.3.2 Use-Case-Diagramm

Die Use-Case-Diagramme übernehmen die Rollen Benutzer und Verwalter aus den User Stories in Abschnitt 2.2.3.

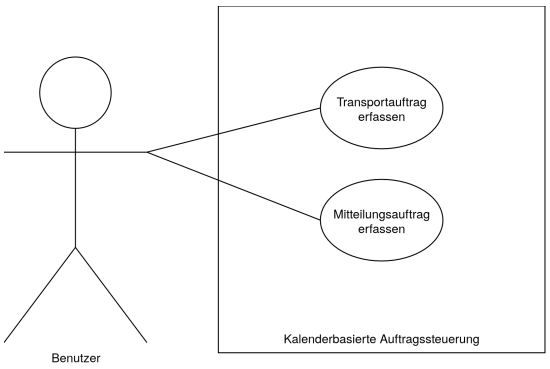


Abbildung 6: Anwendungsfälle des Benutzers

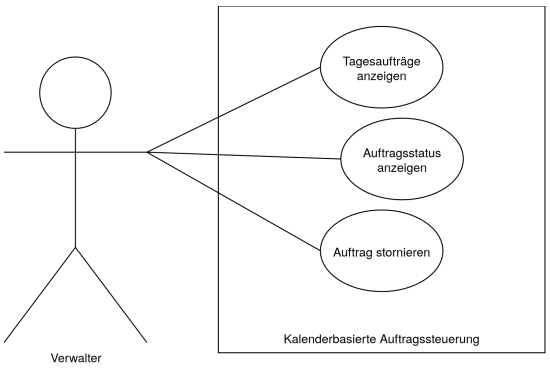


Abbildung 7: Anwendungsfälle des Verwalters

Aus Sicht des Benutzers erfolgt die Interaktion über den Kalender durch das Erfassen eines Transportauftrags oder eines Übermittlungsauftrags. Aus Sicht des Verwalters erfolgt die Interaktion über das `RViz-Panel` durch das Anzeigen der Tagesaufträge, der Statusinformationen und durch das Stornieren von Aufträgen.

2.3.3 Ablaufdiagramm

Die Ablaufdiagramme zeigen die Prozessschritte von der Erfassung bis zum Abschluss oder zur Stornierung eines Auftrags.

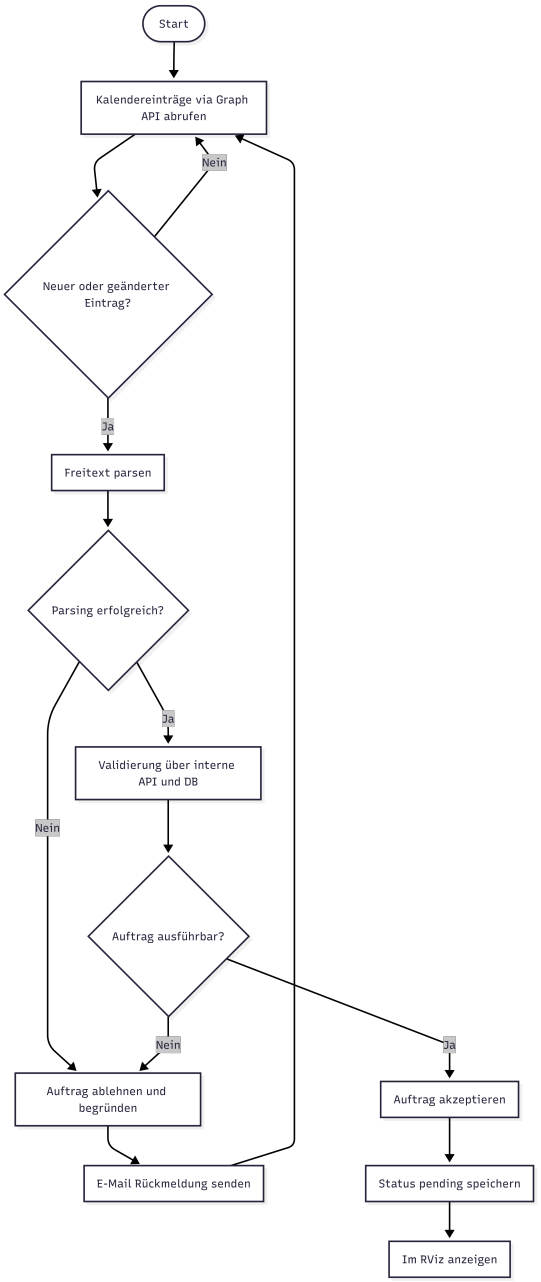


Abbildung 8: Erfassung, Parsing und Validierung eines Kalendereintrags

Nach dem Abruf der Kalendereinträge über die Microsoft Graph API wird der enthaltene Freitext geparst.

Kann der Freitext nicht korrekt interpretiert werden, wird der Auftrag nicht weiterverarbeitet. Bei erfolgreichem Parsing wird über die interne Datenbank geprüft, ob die erforderlichen Daten vorhanden sind und ob der Auftrag ausführbar ist.

Wird der Auftrag als nicht ausführbar bewertet, wird er abgelehnt und der Auftragsersteller per E-Mail mit einer Begründung informiert. Wird der Auftrag als ausführbar bewertet, wird er akzeptiert, mit dem Status `pending` gespeichert und anschliessend im `RViz-Panel` angezeigt.

Das zweite Diagramm beschreibt den Ablauf eines Auftrags im Status `pending`. Bis zur geplanten Startzeit verbleibt der Auftrag im Wartezustand. Zusätzlich wird auf eine manuelle Stornierung über das `RViz-Panel` geprüft.

Wird eine Stornierung ausgelöst, wird diese verarbeitet, der Status aktualisiert und eine entsprechende Rückmeldung versendet. Erfolgt keine Stornierung und ist die Startzeit erreicht, wird der Auftrag an das Navigations-Mock übergeben.

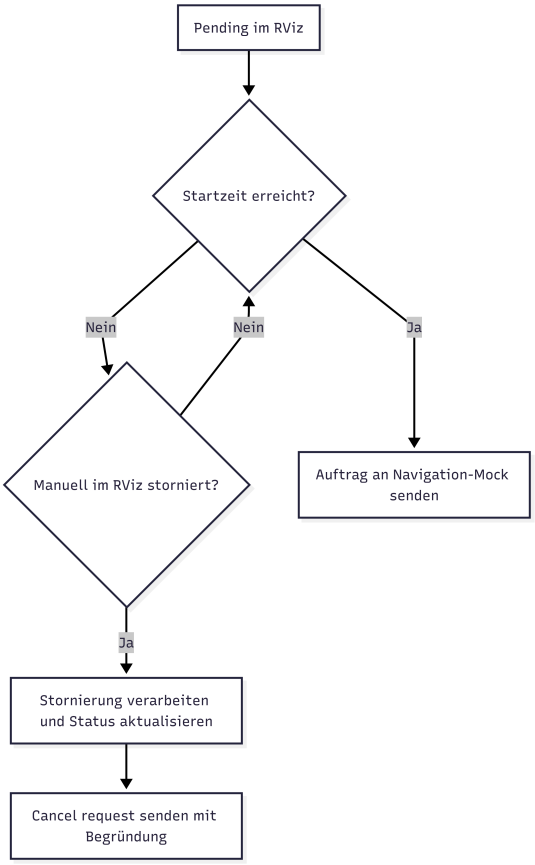


Abbildung 9: Start oder manuelle Stornierung eines Auftrags

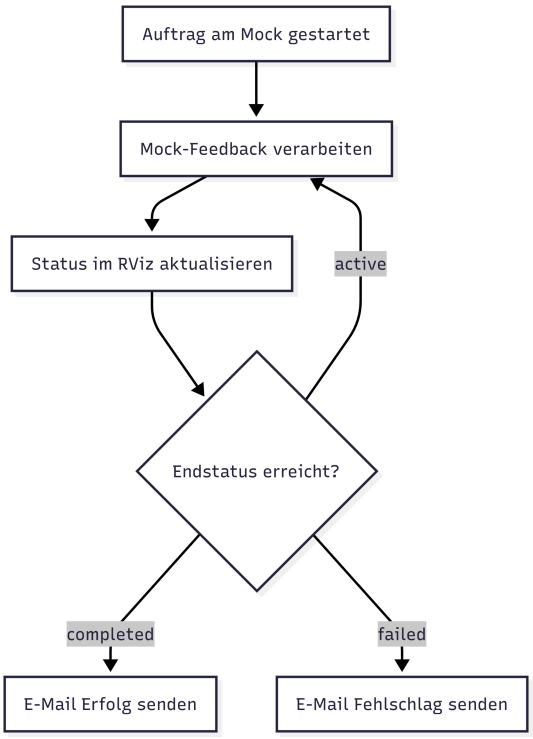


Abbildung 10: Verarbeitung des Feedbacks des Navigations-Mocks und Abschluss eines Auftrags

Das dritte Diagramm zeigt die Verarbeitung eines bereits gestarteten Auftrags. Nach dem Start am Navigations-Mock wird das Feedback verarbeitet und der Status im `RViz-Panel` aktualisiert. Liegt weiterhin der Status `active` vor, wird erneut Feedback verarbeitet. Bei `completed` oder `failed` wird der Auftrag abgeschlossen und eine entsprechende E-Mail versendet.

2.3.4 Schnittstellenplan

Der Schnittstellenplan legt `Topics`, `Services` und den Datenaustausch zwischen den zentralen Komponenten fest.

Die ROS2-Architektur wird nach Verantwortlichkeiten getrennt. Der Kalenderzugriff, die Validierung, die Ausführung und die Visualisierung laufen in separaten `Nodes`. Auftragslisten werden über ein `Publish-Subscribe-Muster` verteilt, weil mehrere Empfänger denselben aktuellen Zustand benötigen. Änderungen am Kalender verwenden `Request-Response-Kommunikation`, weil der aufrufende `Node` eine direkte Rückmeldung über Erfolg oder Fehler benötigt. Die Kommunikation mit dem Navigations-Mock ist als `Action` modelliert.

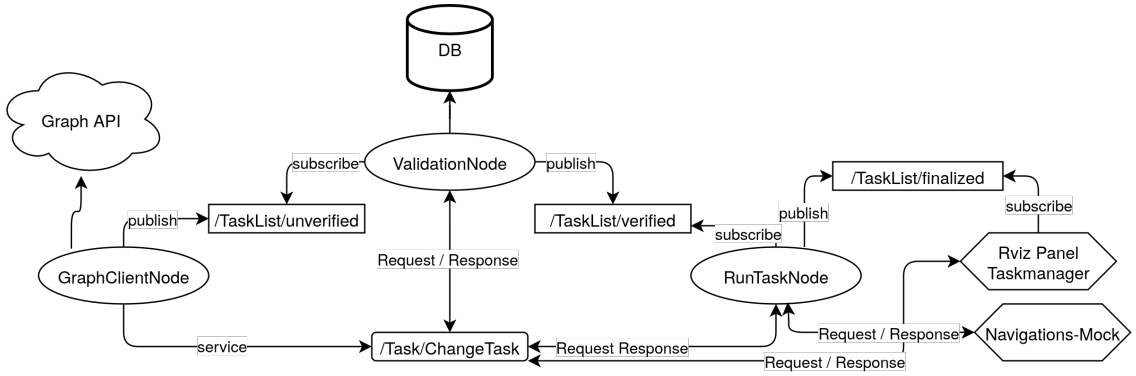


Abbildung 11: Schnittstellenplan der ROS2-Kommunikation

- Rechteck = `Topic`
- Rechteck mit runden Ecken = `Service`
- Ellipse = `Node`
- Hexagon = Externe Systeme

Der `GraphClientNode` publiziert geladene Kalenderdaten auf `/TaskList/unverified`. Der `ValidationNode` ergänzt gültige Aufträge mithilfe der Datenbank und veröffentlicht sie auf `/TaskList/verified`. Der `RunTaskNode` verarbeitet diese Aufträge, publiziert den finalen Status auf `/TaskList/finalized` und versorgt damit das `RViz-Panel`. Änderungen und Rückmeldungen laufen gebündelt über den `Service /Task/ChangeTask` zurück an den `GraphClientNode`.

Für die Auftragslisten ist jeweils der zuletzt bekannte Zustand relevant. Deshalb soll in der Umsetzung ein `QoS-Profil` mit zuverlässiger Zustellung und einem gespeicherten letzten Zustand verwendet. Das ist besonders für das `RViz-Panel` wichtig, da es nach einem Neustart nicht auf die nächste Kalenderabfrage warten soll.

Schnittstelle	**Zweck**	**Wichtigste Daten**
`/TaskList/unverified`	Übermittlung neu geladener, noch nicht validierter Kalendereinträge	Auftragsdaten aus dem Kalender, z. B. Titel, Zeitangaben, Beschreibung, Erfasser
`/TaskList/verified`	Übermittlung validierter und ausführbarer Aufträge	Validierter Auftrag, Status, Zielposition(en), Nachricht
`/TaskList/finalized`	Übermittlung finalisierter oder fehlgeschlagener Aufträge	Validierter Auftrag, Status, Zielposition(en), Nachricht
`/Task/ChangeTask`	Übermittlung von Änderungen und Rückmeldungen an den Kalender- und Benachrichtigungsprozess	Änderungstyp, Auftrags-ID, Nachricht

2.3.5 Parsing-Strategie und Validierungsregeln

Für die Interpretation der Kalendereinträge wurde ein regelbasierter, deterministischer Parser gewählt. Die Eingabe folgt einem schemaorientierten Format der auf Schlüssel-Werte ausgerichtet ist. Der Ansatz priorisiert Determinismus und Testbarkeit gegenüber freier Spracheingabe. Varianten wie ein LLM-Modell oder ein nachgelagertes Formular wurden in betracht gezogen, hätten aber den Umsetzungsaufwand deutlich erhöht.

Kalender: IMVS_Robotik Vertraulichkeit: Normal

Meeting reminder

Teilnehmende	IMVS_Robotik, max.muster@fhnw.ch
Zeit	31.03.2026, 18:00–18:30
Ort / Raum	IMVS_Robotik
Besprechung	Kein Teams-Meeting

Beschreibung

Auftrag: Übermittlung
Nachricht: Erinnerung, wir haben ein Meeting um 19 Uhr
Zielperson: Erika Muster

Der Text muss die Angaben `Auftrag`, `Nachricht`, `Zielperson` und optional, je nach Auftragsart, eine zweite `Zielperson` enthalten. Sind diese Angaben nicht vollständig vorhanden, wird der Auftrag abgelehnt und die Vorlage mitgesendet.

```
Auftrag:
Nachricht:
Zielperson1:
Zielperson2:
```

Dabei werden auch Varianten des Schlüssels `Zielperson` berücksichtigt, beispielsweise nur `Zielperson` oder die Kombination aus `Zielperson1` und `Zielperson2`.

Zusätzlich zum Parsing wurden Validierungsregeln festgelegt. Dabei wird geprüft, ob alle Pflichtfelder vorhanden sind und ob ein Auftrag die Anforderungen erfüllt. Diese Regeln bestimmen, ob ein Auftrag gültig ist und weiterverarbeitet werden kann. Zeitliche Überschneidungen werden als Konflikterkennung behandelt. Die Existenz von Zielpersonen und Räumen ist Teil der Referenzvalidierung gegenüber der Datenbank-API als Datenzugriffsschicht.

- Es dürfen keine zeitlichen Überschneidungen mit anderen Aufträgen bestehen.
- Titel und Beschreibung müssen vorhanden sein.
- Der Erfasser des Kalendereintrags muss vorhanden sein.
- Die Auftragsdauer muss mindestens 5 und höchstens 30 Minuten betragen.
- Der Freitext muss erfolgreich geparst werden können.
- Der Auftragstyp muss als Übermittlungsauftrag oder Transportauftrag erkennbar sein.
- Eine Nachricht muss vorhanden sein.
- Es müssen je nach Auftrag eine oder zwei Zielpersonen vorhanden sein.
- Die Zielperson oder Zielpersonen müssen in der Datenbank vorhanden sein.
- Der Zielperson oder den Zielpersonen muss ein gültiger Raum zugewiesen sein.

Diese Daten können mithilfe des `Graph SDK` abgerufen werden. Die für die weitere Verarbeitung benötigten Informationen werden anschliessend auf einem `Topic` publiziert, sodass dem `ValidationNode` alle erforderlichen Daten zur Validierung eines Auftrags bereitgestellt werden.

2.3.6 Mockup

Im Mockup werden der Aufbau und die Interaktion des `RViz-Panels` vor der Implementierung visualisiert. Für das Mockup wird ein einfaches und übersichtliches Bedienkonzept gewählt, das alle für die Nutzung relevanten Informationen enthält.

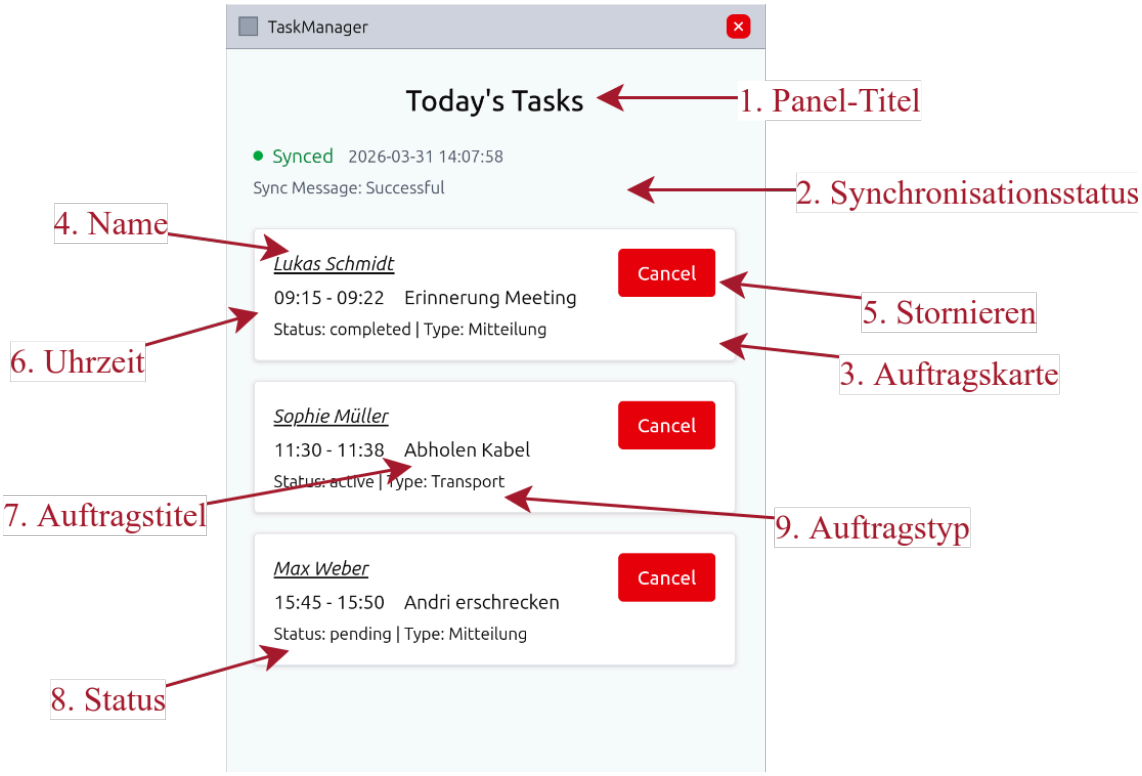


Abbildung 12: Mockup des `RViz-Panels`

1. Panel-Titel: Durch den Panel-Titel wird die Tagesansicht der Aufträge gekennzeichnet.
2. Synchronisationsstatus: Durch den Synchronisationsstatus werden der Verbindungsstatus und der letzte Aktualisierungszeitpunkt angezeigt.
3. Auftragskarte: In der Auftragskarte werden Auftragsdaten und verfügbare Aktionen zusammengefasst.
4. Name: Durch den Namen wird der Erfasser des Auftrags angezeigt.
5. Stornieren: Über diese Funktion wird die manuelle Auftragsstornierung ausgelöst.
6. Uhrzeit: Über die Uhrzeit wird das geplante Ausführungsfenster dargestellt.
7. Auftragstitel: Durch den Auftragstitel der definierter Titel des Erfassers angezeigt.
8. Status: Über den Status wird der aktuelle Auftragszustand angezeigt.
9. Auftragstyp: Durch den Auftragstyp wird die Art des Auftrags gekennzeichnet.

Das Mockup bündelt Auftragstitel, Uhrzeit, Status, Auftragstyp und Erfasser in einzelnen Auftragskarten. Der Synchronisationsstatus zeigt die Aktualität der Daten, und die Stornierungsfunktion ist direkt in der Auftragskarte erreichbar.

Die Benutzeroberfläche bleibt in englischer Sprache, weil die Bedienelemente von RViz2 üblicherweise ebenfalls englisch beschriftet sind.

Es wurden Rückmeldungen von mehreren Mitarbeitenden eingeholt. Dabei wurde überwiegend positives Feedback gegeben. Als Verbesserungen wurden genannt, dass zusätzlich auch das aktuelle Datum hilfreich wäre und das «Generated At» Titel keine zusätzlichen Informationen dem Nutzer bereitstellt. Dieses Feedback wurde berücksichtigt und das Mockup angepasst.

Datum	**Person**	**Input/Vorgabe**	**Umgesetzte Massnahme**	**Resultat**
01.04.2026	Andri Wild in der Rolle des Auftraggebers	Das Mockup ist gut, der Text «Generated At» hat keine zusätzlichen Informationen, weswegen der Titel weggelassen werden kann.	Im Synchronisationsstatus den Variablennamen des Zeitstempels nicht ausgeben.	Bearbeitet
31.03.2026	Simon Freiermuth, Mitarbeiter	Sieht solide aus.	Keine.	Bearbeitet
31.03.2026	Janic Berger, Mitarbeiter	Das Datum könnte noch hilfreich sein.	Datum im Synchronisationsstatus ergänzen und nicht nur den Zeitstempel anzeigen.	Bearbeitet

2.3.7 Testkonzept und Testfälle

Im Testkonzept werden die strategischen Rahmenbedingungen für die geplanten Prüfungen festgelegt. Es wird definiert, welche Teile der Lösung geprüft werden, welche Testarten vorgesehen sind und unter welchen Bedingungen die Tests durchgeführt werden. Die konkreten Testfälle werden in Abschnitt 2.3.7.2 separat beschrieben.

2.3.7.1 Testkonzept

Testziel	Mit den geplanten Prüfungen soll nachgewiesen werden, dass Kalenderaufträge korrekt abgerufen, geparst, validiert, visualisiert und verarbeitet werden. Zusätzlich soll überprüft werden, ob Rückmeldungen und Statusänderungen korrekt an die beteiligten Komponenten übermittelt werden.
Testumfang	Geprüft werden der Kalenderabruf über die Microsoft Graph API, das Parsing der Freitexte, die Validierung über die interne API und die Datenbank, die Übergabe an das Navigations-Mock sowie die Anzeige und Interaktion im `RViz-Panel`.
Nicht Bestandteil	Nicht Bestandteil der Prüfungen ist die direkte Kommunikation mit einem physischen Roboter, da für die Umsetzung und Prüfung ausschliesslich das bereitgestellte Navigations-Mock verwendet wird.

Testart	Geplant	Beschreibung
Unit-Tests	Ja	Prüfung einzelner Komponenten der Applikationslogik, insbesondere von Parsingfunktionen.
Integrationstests	Ja	Prüfung des Zusammenspiels mehrerer Komponenten, insbesondere des Datenflusses vom Kalendereintrag bis zur Anzeige im `RViz-Panel`.
Szenario-Tests	Ja	Prüfung fachlicher Gesamtabläufe unter realitätsnahen Bedingungen, einschliesslich Annahme, Ablehnung, Stornierung und Rückmeldung.

Testumgebung	Die Tests werden in der vorgesehenen Entwicklungsumgebung mit `ROS2`, `RViz`, der Microsoft Graph API, der internen Datenbank-API und dem bereitgestellten Navigations-Mock durchgeführt.
Testdaten	Als Testdaten dienen vorbereitete Kalendereinträge mit gültigen und ungültigen Freitexten, unterschiedlichen Auftragsarten sowie verschiedenen Zeiten.
Testverantwortung	Die Planung, Durchführung und Dokumentation der Tests werden vom Kandidaten vorgenommen.

Eintrittskriterien	Die zu prüfenden Komponenten müssen implementiert, startbar und in der Testumgebung verfügbar sein. Zusätzlich müssen die benötigten Schnittstellen sowie Testdaten bereitstehen.
Abschlusskriterien	Die Tests gelten als abgeschlossen, wenn die definierten Testfälle durchgeführt, dokumentiert und die wesentlichen Anforderungen nachvollziehbar überprüft wurden.
Risiken und Hinweise	Risiken bestehen insbesondere bei der Verfügbarkeit externer Schnittstellen wie der Microsoft Graph API sowie bei fehlerhaften oder unvollständigen Testdaten. Solche Einflüsse sind bei der Auswertung der Testergebnisse zu berücksichtigen.

Die Testfälle wurden aus den User Stories in Abschnitt 2.2.3 abgeleitet.

2.3.7.2 Testfälle

US-Nr.	TC-Nr.	Testart	Voraussetzung	Eingabe	Erwartetes Resultat
1	1.1	Integrationstest	Graph API und Ressourcenkalender sind verfügbar.	Gültigen Kalendereintrag erstellen.	Eintrag wird erkannt und eingelesen.
1	1.2	Integrationstest	Synchronisation ist aktiv.	Bestehenden Kalendereintrag ändern.	Änderung wird erkannt und übernommen.
2	2.1	Integrationstest	Gültiger Kalendereintrag für Übermittlungsauftrag liegt vor.	Freitext mit Auftrag, Nachricht und Zielperson erfassen.	Freitext wird korrekt geparst.
2	2.3	Integrationstest	Kalendereintrag liegt vor.	Unvollständigen Freitext erfassen.	Auftrag wird abgelehnt.
3	3.3	Integrationstest	Gültiger Kalendereintrag für Transportauftrag liegt vor.	Freitext mit Auftrag, Nachricht und zwei Zielpersonen erfassen.	Beide Zielpersonen werden erkannt.
4	4.1	Integrationstest	Interne API und Datenbank sind verfügbar.	Gültigen Auftrag validieren.	Auftrag wird freigegeben.
4	4.2	Integrationstest	Auftrag mit unbekannter Zielperson liegt vor.	Auftrag validieren.	Auftrag wird abgelehnt.

US-Nr.	TC-Nr.	Testart	Voraussetzung	Eingabe	Erwartetes Resultat
4	4.3	Integrationstest	Ein Auftrag im gleichen Zeitraum existiert bereits.	Neuen Auftrag validieren.	Überschneidung wird erkannt und Auftrag abgelehnt.
5	5.1	Integrationstest	Mailversand ist verfügbar. Auftrag ist gültig.	Auftrag annehmen.	Annahme-Rückmeldung wird versendet.
5	5.2	Integrationstest	Mailversand ist verfügbar. Auftrag ist ungültig.	Auftrag ablehnen.	Ablehnungs-Rückmeldung wird versendet.
5	5.3	Integrationstest	Auftrag wurde am Navigations-Mock ausgeführt.	Endstatus completed oder failed empfangen.	Abschluss-Rückmeldung wird versendet.
6	6.1	Integrationstest	Validierte Aufträge liegen vor. `RViz-Panel` ist gestartet.	Keine	Tagesaufträge werden angezeigt.
6	6.2	Integrationstest	Angezeigter Auftrag erhält Statusänderung.	Status-Update empfangen.	Status wird im `RViz` aktualisiert.
7	7.1	Szenario / E2E	Auftrag ist im `RViz` sichtbar. Outlook-Kalender und Mailversand sind verfügbar.	Stornierung im `RViz` auslösen.	Auftrag wird storniert, Status im `RViz` aktualisiert, Termin im Outlook-Kalender abgelehnt und Benachrichtigung versendet.
NF	NF.1	Integrationstest	System läuft. Fehlerhafter Kalendereintrag liegt vor.	Kalendereintrag ohne gültige Pflichtangaben verarbeiten.	System bleibt stabil. Auftrag wird abgelehnt.
NF	NF.2	Integrationstest	Logging ist aktiviert.	Gültigen und ungültigen Auftrag verarbeiten.	Relevante Ereignisse und Fehler werden ausgegeben.
NF	NF.3	Integrationstest	Graph API, `RViz` und Verarbeitung sind aktiv.	Neuen gültigen Kalendereintrag erstellen.	Auftrag wird innerhalb von 60 Sekunden verarbeitet und angezeigt.

US-Nr.	TC-Nr.	Testart	Voraussetzung	Eingabe	Erwartetes Resultat
NF	NF.4	Integrationstest	ROS2-Kommunikation ist aktiv. QoS ist konfiguriert.	Validierten Auftrag über ein `Topic` oder einen `Service` übertragen.	Auftrag wird zuverlässig und vollständig übertragen.
E2E	E2E.1	Szenario / E2E	Graph API, Validierung, Mailversand, `RViz-Panel` und Navigations-Mock sind verfügbar.	Gültigen Kalendereintrag erstellen.	Auftrag wird erkannt, geparst, validiert, angenommen, im `RViz` angezeigt und Annahme-Rückmeldung wird versendet.
E2E	E2E.2	Szenario / E2E	Graph API, Validierung und Mailversand sind verfügbar.	Ungültigen Kalendereintrag erstellen.	Auftrag wird erkannt, verarbeitet, abgelehnt, im Outlook-Kalender entsprechend markiert und Ablehnungs-Rückmeldung wird versendet.
E2E	E2E.3	Szenario / E2E	Gültiger Auftrag wurde angenommen und an das Navigations-Mock übergeben. `RViz-Panel` und Mailversand sind verfügbar.	Endstatus completed oder failed empfangen.	Status wird im `RViz` aktualisiert und Abschluss-Rückmeldung wird versendet.

2.3.8 Planungsänderungen und Verfeinerungen

Im Abschnitt Planungsänderungen und Verfeinerungen werden wesentliche Anpassungen gegenüber der ursprünglichen Planung sowie deren Gründe und Auswirkungen auf die weitere Umsetzung dargestellt.

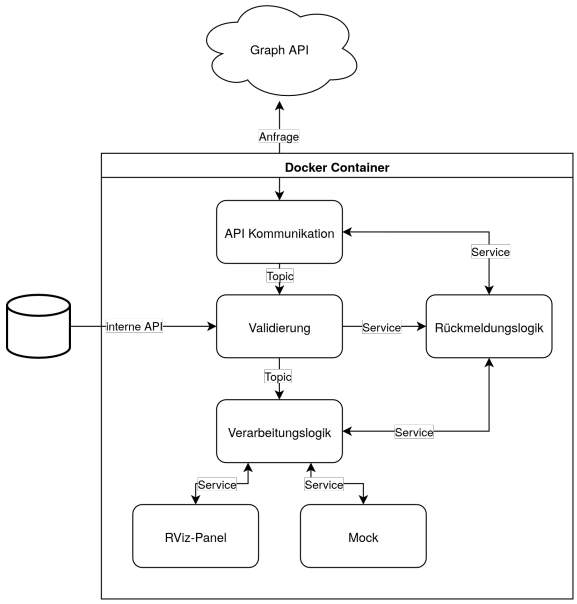


Abbildung 13: Erste Variante der Grobarchitektur

Zu Beginn wurde diese Architekturvariante vorgesehen. Bei der weiteren Ausarbeitung von Abschnitt 2.3.4 wurden die Schnittstellen konkretisiert. Rückmeldungen und Statusänderungen werden zentral über den `Service /Task/ChangeTask` verarbeitet. Für Ablehnung wird die definierte Rückmeldungsvorlage mitgesendet, damit der Nutzer ein eindeutiges Feedback hat.

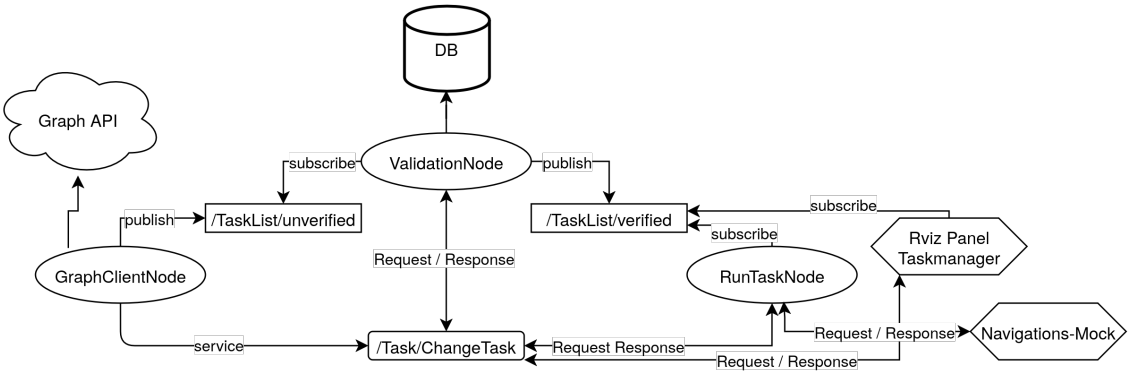


Abbildung 14: Erste Variante des Schnittstellenplans

In dieser ersten Version des Schnittstellenplans wurde zunächst nicht berücksichtigt, dass auch der Status `failed` verarbeitet werden muss. Aus diesem Grund wurde ein zusätzlicher `Service /Task/UpdateStatus` vorgesehen, über den dem `RViz-Panel` mitgeteilt werden sollte, dass ein Auftrag nicht erfolgreich ausgeführt werden konnte und entsprechend als `failed` markiert werden muss.

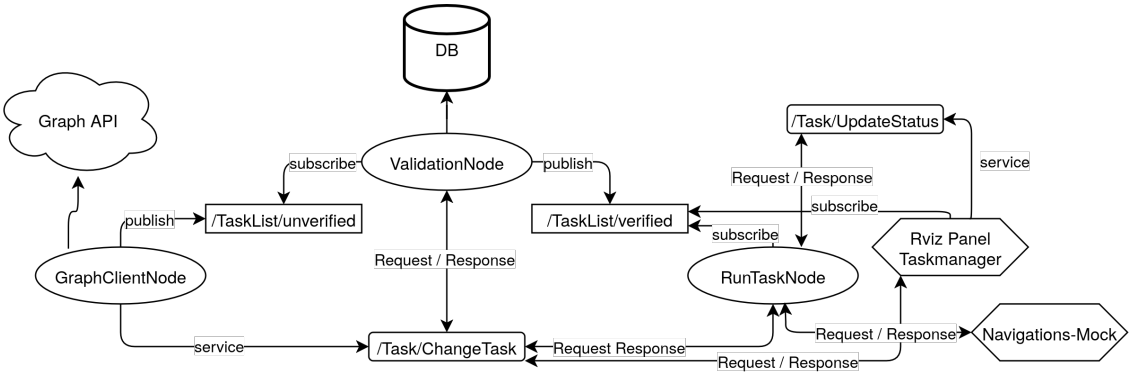


Abbildung 15: Überarbeitete Variante des Schnittstellenplans

Nach dieser Korrektur wurde der Schnittstellenplan weiter vereinfacht. `/Task/UpdateStatus` musste nicht als `Service` umgesetzt werden, weil ein `Topic` für die Statusweitergabe ausreicht. Zusätzlich abonniert das `RViz-Panel` nicht mehr direkt das `Topic` des `ValidationNode`, sondern ein vom `RunTaskNode` publiziertes `Topic`. Der `RunTaskNode` stellt den finalen Status eines Auftrags einschliesslich `failed` direkt bereit, und das `RViz-Panel` bleibt auf Visualisierung und Stornierung beschränkt.

In der Planungsphase war ein Klassendiagramm vorgesehen. Bei der Konkretisierung zeigte sich jedoch, dass die wesentlichen Architekturentscheidungen in diesem Projekt nicht nur durch Klassenbeziehungen, sondern durch die ROS2-Kommunikation über Topics, Services und Actions bestimmt werden. Ein klassisches UML-Klassendiagramm hätte daher nur begrenzten Zusatznutzen geliefert und die entscheidenden Schnittstellen- und Ablaufaspekte nicht ausreichend dargestellt. Aus diesem Grund wurde auf ein separates Klassendiagramm verzichtet und der Fokus auf Schnittstellenplan, Datenfluss- und Ablaufdiagramme gelegt. Die Nachvollziehbarkeit der Lösung bleibt dadurch gewährleistet

2.4 Entscheiden

In diesem Kapitel werden die Entscheidungsgrundlagen für Architektur und Parsing-Strategie auf Basis von Abschnitt 2.3 dokumentiert. Die wichtigsten Varianten werden verglichen, damit die gewählte Lösung nachvollziehbar begründet ist.

2.4.1 Präferenzmatrix

In einer Präferenzmatrix werden Projektziele priorisiert[7]. Dazu wurden die für das Projekt wesentlichen Ziele aus Abschnitt 2.2.1 definiert und anschliessend miteinander verglichen.

Folgende Projektziele wurden identifiziert:

1. Zuverlässige Verarbeitung von Aufträgen

   Es ist wichtig, dass Aufträge korrekt erkannt, validiert und weiterverarbeitet werden, damit die Kernfunktionen der Lösung zuverlässig erfüllt werden können.

2. Geringer Umsetzungsaufwand im IPA-Rahmen

   Die Lösung muss innerhalb des vorgegebenen Zeitrahmens realistisch umsetzbar bleiben.

3. Nachvollziehbarkeit und Wartbarkeit

   Das System muss für Fachpersonen nachvollziehbar aufgebaut sein und in seiner Struktur sowie in seinen Entscheidungen klar begründet werden.

4. Einfache Integration in ROS2 und bestehende Schnittstellen

   Da die Lösung von mehreren Schnittstellen wie ROS2, `RViz`, Microsoft Graph API, Datenbank und Navigations-Mock abhängt, ist eine möglichst einfache und saubere Integration erforderlich.

5. Robuste Behandlung fehlerhafter Eingaben

   Die Fehlerbehandlung ist wichtig, damit Parsing und Validierung auch bei unvollständigen oder fehlerhaften Eingaben korrekt funktionieren.

Mithilfe dieser Ziele wurde eine Präferenzmatrix erstellt, in der jedes Ziel mit den übrigen Zielen verglichen und anschliessend priorisiert wurde.

Welches Ziel ist wichtiger?	**Zuverlässigkeit**	**Umsetzbarkeit**	**Nachvollziehbarkeit**	**Integrationsfähigkeit**	**Fehlerbehandlung**
Zuverlässigkeit	-				
Umsetzbarkeit	Z1	-			
Nachvollziehbarkeit	Z1	Z2	-		
Integrationsfähigkeit	Z1	Z2	Z4	-	
Fehlerbehandlung	Z1	Z2	Z5	Z5	-

Ziel	**Anzahl Präferenzen**	**Rang**	**Abgeleitete Gewichtung**
Zuverlässigkeit	4	1	30%
Umsetzbarkeit	3	2	25%
Nachvollziehbarkeit	0	5	10%
Integrationsfähigkeit	1	4	15%
Fehlerbehandlung	2	3	20%

Die Zuverlässigkeit wurde am höchsten gewichtet, da ein funktionierendes Produkt im Vordergrund steht. Danach folgen die Umsetzbarkeit im IPA-Rahmen und die Fehlerbehandlung. Integrationsfähigkeit sowie Nachvollziehbarkeit und Wartbarkeit bleiben wichtig, wurden jedoch tiefer gewichtet, weil die zuverlässige Kernfunktion zuerst erfüllt werden muss.

Die Anzahl der Präferenzen wird nicht 1:1 als Prozentwert übernommen. Sonst hätte Nachvollziehbarkeit und Wartbarkeit keinen Einfluss auf die Nutzwertanalyse, obwohl dieses Ziel für die Dokumentation und spätere Pflege relevant bleibt. Deshalb werden die Ränge in abgestufte Gewichte übertragen.

2.4.2 Entscheid Architekturaufbau

Für den Architekturaufbau wurde geprüft, ob die Verarbeitung als lose gekoppelte, komponentenbasierte Architektur oder als monolithischer Verarbeitungsknoten umgesetzt werden soll.

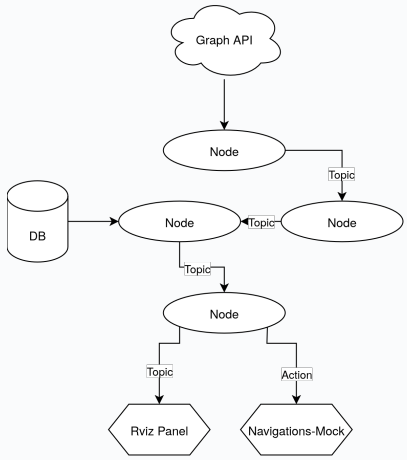
Variante 1: Lose gekoppelte, komponentenbasierte Architektur

Vorteile

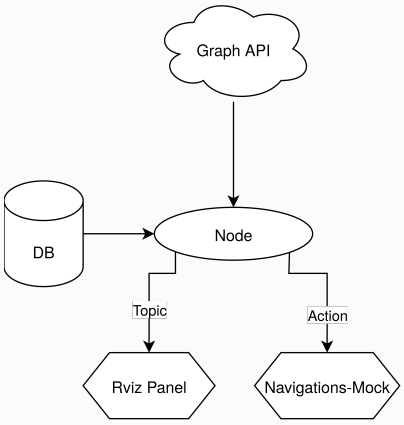
- Klare Verantwortlichkeiten
- Geringe Kopplung bei hoher Kohäsion
- Serviceorientierte Zerlegung
- Gute Rückverfolgbarkeit des Datenflusses
- Hohe Wartbarkeit
- Einfachere Fehlersuche
- Bessere Stabilität durch getrennte Komponenten
- Gute Erweiterbarkeit

Nachteile

- Höherer Implementierungsaufwand
- Zusätzlicher Kommunikationsaufwand
- Komplexere Schnittstellenabstimmung



Variante mit lose gekoppelten Verarbeitungskomponenten



Variante mit monolithischem Verarbeitungsknoten

Variante 2: Monolithischer Verarbeitungsknoten

Vorteile

- Geringerer Implementierungsaufwand
- Einfacherer Datenfluss
- Weniger Kommunikationsaufwand
- Schnellere Umsetzung
- Einfacherer Start und Betrieb

Nachteile

- Hohe Komplexität in einer Komponente
- Geringere Wartbarkeit
- Erschwerte Erweiterbarkeit
- Erschwerte Fehlerlokalisierung

Machbarkeitsprüfung

Die lose gekoppelte, komponentenbasierte Architektur wird als gut machbar eingestuft, da sie sich trotz höherem Umsetzungsaufwand strukturiert in die in Abschnitt 2.3.1 geplante Systemarchitektur integrieren lässt. Der monolithische Architekturansatz wäre technisch ebenfalls machbar, bringt jedoch strukturelle Nachteile mit sich.

Nutzwertanalyse

A = Variante 1: Lose gekoppelte, komponentenbasierte Architektur

B = Variante 2: Monolithischer Verarbeitungsknoten

Die Gewichtung orientiert sich an der Präferenzmatrix aus Abschnitt 2.4.1. Bei der Architekturentscheidung wird die robuste Behandlung fehlerhafter Eingaben als Fehlerisolation und Stabilität der Komponenten bewertet, weil hier nicht die Eingabevalidierung selbst, sondern der Systemaufbau verglichen wird.

Kriterium	Beschreibung	Gew.	Variante A		Variante B		Bem.
			Pkt.	NW	Pkt.	NW	
Zuverlässigkeit	Stabilität und zuverlässige Verarbeitung im Systemaufbau	30%	5	150	3	90	A reduziert Auswirkungen einzelner Fehler.
Umsetzbarkeit	Aufwand für Umsetzung und Erstimplementierung	25%	2	50	5	125	B ist schneller umsetzbar.
Fehlerbehandlung	Fehlerisolation und Eingrenzung von Folgefehlern	20%	5	100	3	60	A unterstützt Fehlerisolation.
Integrationsfähigkeit	Einbindung in ROS2, Schnittstellen und Betrieb	15%	3	45	4	60	B benötigt weniger interne Abstimmung.
Nachvollziehbarkeit	Struktur, Verständlichkeit und Änderbarkeit der Lösung	10%	5	50	2	20	A folgt `Separation of Concerns`.
Total		**100%**		**395**		**355**	**Variante A erreicht den höheren Nutzwert.**

Variante A erzielt den höheren Nutzwert. Ausschlaggebend dafür sind vor allem die höhere Zuverlässigkeit, die bessere Fehlerisolation und die nachvollziehbare Struktur durch lose Kopplung und hohe Kohäsion. Der höhere Implementierungsaufwand wird dadurch ausgeglichen.

2.4.3 Entscheid Parsing-Strategie

Für die Interpretation der Kalendereinträge wurden drei Varianten betrachtet. Bewertet wurden insbesondere zuverlässige Verarbeitung, Umsetzbarkeit im IPA-Rahmen, robuste Fehlerbehandlung, Integrationsfähigkeit, Nachvollziehbarkeit und Benutzerfreundlichkeit.

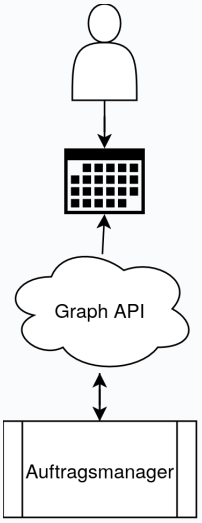
Variante 1: Schlüssel-Wert-Ansatz

Vorteile

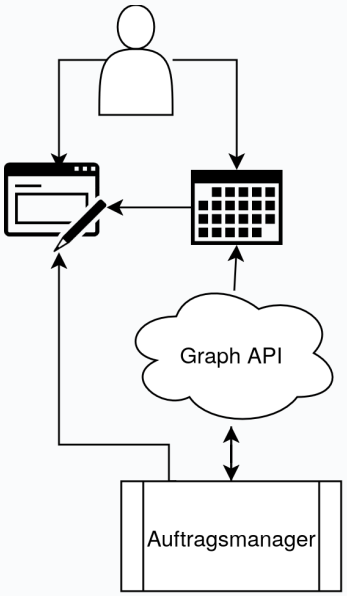
- Geringer Implementierungsaufwand
- Im IPA-Rahmen gut umsetzbar
- Ergebnisse klar nachvollziehbar
- Keine zusätzliche Abhängigkeit
- Einfach testbar und wartbar
- Fehlerursachen leicht erkennbar

Nachteile

- Striktes Eingabeformat erforderlich
- Geringere Flexibilität
- Begrenzte Benutzerfreundlichkeit
- Unvollständige Angaben führen rasch zu Ablehnungen



Variante mit Schlüssel-Wert-Ansatz



Variante mit Formular

Variante 2: Formularbasierte Eingabe

Vorteile

- Strukturierte und vollständige Eingaben
- Geringerer Parsing-Aufwand
- Weniger Fehler bei Pflichtfeldern
- Teilweise Validierung in der Eingabemaske
- Saubere Datenbasis

Nachteile

- Zusätzlicher Implementierungsaufwand
- Schlechtere Einbindung in den Kalenderprozess
- Weniger natürlich für Benutzer
- Medienbruch zwischen Kalender und Formular
- Höherer Aufwand für Rückmeldungen

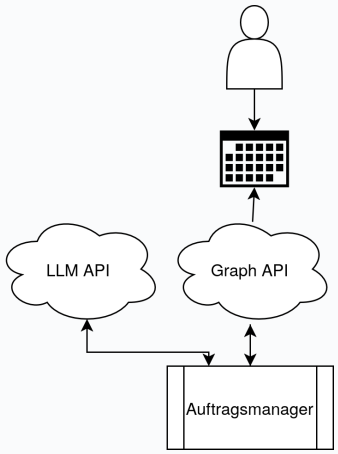
Variante 3: LLM-Modell

Vorteile

- Flexible Verarbeitung natürlicher Sprache
- Freiere Formulierung durch Benutzer
- Auch uneinheitliche Eingaben interpretierbar
- Weniger starre Eingaberegeln

Nachteile

- Hoher Implementierungsaufwand
- Zusätzliche externe Abhängigkeiten
- Schwerer nachvollziehbare Resultate
- Höhere Unsicherheit bei der Extraktion
- Aufwendiger zu testen
- Für den IPA-Rahmen zu komplex
- Zusätzliche Kosten möglich



Variante mit LLM-Modell

Machbarkeitsprüfung

Der Schlüssel-Wert-Ansatz wird als gut machbar eingestuft, da er mit vertretbarem Aufwand umgesetzt und zuverlässig getestet werden kann. Eine formularbasierte Eingabe wäre grundsätzlich machbar, verursacht jedoch zusätzlichen Integrationsaufwand. Der Einsatz eines LLM-Modells wird wegen des technischen und organisatorischen Aufwands als nur eingeschränkt machbar bewertet.

Nutzwertanalyse

A = Variante 1: Schlüssel-Wert-Ansatz

B = Variante 2: Formularbasierte Eingabe

C = Variante 3: LLM-Modell

Die Gewichtung folgt ebenfalls der Präferenzmatrix aus Abschnitt 2.4.1. Die Benutzerfreundlichkeit wird zusätzlich mit 5% aufgenommen, weil sie nur bei der Parsing-Strategie ein eigener Unterscheidungsfaktor ist und in den Projektpräferenzen nicht zu den Hauptzielen gehört.

Kriterium	Beschreibung	Gew.	Variante A		Variante B		Variante C		Bem.
			Pkt.	NW	Pkt.	NW	Pkt.	NW	
Zuverlässigkeit	Zuverlässige und deterministische Verarbeitung der Auftragsdaten	30%	4	120	5	150	2	60	B erzwingt strukturierte Eingaben, A verarbeitet definierte Schlüssel zuverlässig.
Umsetzbarkeit	Machbarkeit innerhalb des IPA-Rahmens	25%	5	125	3	75	1	25	A ist mit vertretbarem Aufwand umsetzbar.
Fehlerbehandlung und Testbarkeit	Umgang mit unvollständigen Eingaben sowie Prüfbarkeit der Resultate	20%	5	100	4	80	2	40	A erlaubt klare Validierungsregeln und erwartbare Ablehnungen.
Integrationsfähigkeit	Einbindung in den bestehenden Kalenderprozess	10%	5	50	2	20	3	30	A bleibt direkt im Kalenderprozess, B erzeugt einen Medienbruch.
Nachvollziehbarkeit	Verständlichkeit, Nachprüfbarkeit und Wartbarkeit der Resultate	10%	5	50	4	40	2	20	A liefert klar interpretierbare Ergebnisse.
Benutzerfreundlichkeit	Flexibilität und Natürlichkeit der Eingabe	5%	2	10	3	15	5	25	C ist am flexibelsten, A am restriktivsten.
Total		**100%**		**455**		**380**		**200**	**Variante A erreicht den höchsten Nutzwert.**

Damit wurde der Schlüssel-Wert-Ansatz als bevorzugte Variante festgelegt, da er innerhalb des IPA-Rahmens die insgesamt zweckmässigste Lösung darstellt. Die formularbasierte Eingabe erreicht bei der zuverlässigen Verarbeitung zwar den höheren Einzelwert, verliert jedoch durch den zusätzlichen Umsetzungsaufwand und die schlechtere Integration in den Kalenderprozess.

2.4.4 Schlussfolgerung

Präferenzmatrix

Auf Grundlage von Abschnitt 2.4.1 wurde festgestellt, dass insbesondere die zuverlässige Verarbeitung, die Umsetzbarkeit im IPA-Rahmen sowie die robuste Behandlung fehlerhafter Eingaben für das Projekt von zentraler Bedeutung sind. Diese Zielgewichtung wurde anschliessend in den beiden Nutzwertanalysen berücksichtigt.

Entscheid Architekturaufbau

Für den Architekturaufbau wurde die lose gekoppelte, komponentenbasierte Architektur ausgewählt. Ausschlaggebend waren die höhere Zuverlässigkeit, die bessere Fehlerisolation und die nachvollziehbare Struktur. Der höhere Implementierungs- und Kommunikationsaufwand wird akzeptiert, weil die Trennung der Verantwortlichkeiten die Erweiterbarkeit unterstützt.

Entscheid Parsing-Strategie

Für die Parsing-Strategie wurde der Schlüssel-Wert-Ansatz ausgewählt. Ausschlaggebend waren die gute Umsetzbarkeit innerhalb des IPA-Rahmens, die direkte Integration in den Kalenderprozess, die klare Fehlerbehandlung und die Nachvollziehbarkeit der Resultate. Die geringere Flexibilität wird in Kauf genommen, da dadurch eine einfache und testbare Verarbeitung möglich bleibt.

Die betrachteten Varianten und getroffenen Entscheidungen wurden mit dem Auftraggeber besprochen, wobei die Bewertung und die daraus abgeleitete Auswahl geteilt wurden.

Datum	Person	Input/Vorgabe	Umgesetzte Massnahme	Resultat
01.04.2026	Andri Wild Auftraggeber	Die Bewertung wird geteilt. Das LLM-Modell wäre zwar benutzerfreundlicher, würde jedoch deutlich mehr Umsetzungsaufwand verursachen.	Variante ohne LLM wurde beibehalten und die Begründung in der Schlussfolgerung präzisiert.	Bestätigt

2.5 Realisieren

In diesem Kapitel werden die Implementierungsschritte entlang der Verarbeitung der Kalendereinträge erläutert.

2.5.1 Architektur und Datenfluss

Die in Abschnitt 2.4.2 gewählte und in Abschnitt 2.3.4 konkretisierte modulare Architektur wurde im ROS2-Ökosystem mit getrennten Komponenten umgesetzt. In diesem Kapitel wird deshalb nicht die Architekturentscheidung erneut begründet, sondern gezeigt, wie die Komponenten technisch realisiert und über konkrete `Topics` und `Services` verbunden wurden. Der `GraphClientNode` stellt Kalenderzugriff und Änderungsservice bereit, der `ValidationNode` validiert neue Aufträge und ergänzt Zielinformationen, der `RunTaskNode` verarbeitet freigegebene Aufträge, und das `RViz-Panel` visualisiert Tagesaufträge mit manueller Stornierung.

Der Datenfluss beginnt mit dem Abruf der Kalendereinträge. Die Einträge werden in ein internes `Task`-Domänenobjekt umgewandelt und auf `/fhnw_bot_task_manager/task_list/unverified` publiziert. Nach der Validierung werden ausführbare Aufträge auf `/fhnw_bot_task_manager/task_list/verified` weitergegeben. Während der Ausführung werden Statusänderungen vom `RunTaskNode` verarbeitet und auf `/fhnw_bot_task_manager/task_list/finalized` veröffentlicht. Automatisierte Annahmen und Ablehnungen, manuelle Stornierungen sowie Abschlussrückmeldungen werden über den `Service /fhnw_bot_task_manager/change_task` an den `GraphClientNode` zurückgeführt.

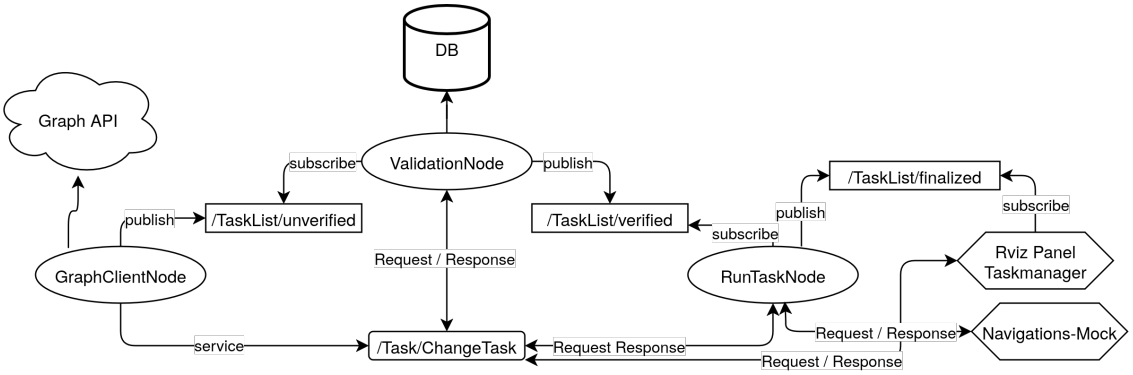


Abbildung 16: Datenfluss der ROS2-Kommunikation

2.5.2 Entgegennahme der Einträge

Der `GraphClientNode` löst das Problem, dass Kalenderzugriff und Rückmeldungen nicht über mehrere Komponenten verteilt werden sollen. Er fungiert als Adapter zur Microsoft Graph API. Deshalb werden alle Interaktionen mit der externen Schnittstelle zentral gekapselt. Für den Abruf der Tagesaufträge wird die Funktion `get_calendar_today` aufgerufen[8]:

`graph_client.users.by_user_id().calendar_view.get()`

Der `graph_client` stellt die zuvor aufgebaute Verbindung zu Microsoft Graph bereit[9]. Als Benutzer beziehungsweise Ressourcenkalender wird `_rrs_WI_5.2B10_Robotik_IMVS@fhnw.ch` im Feld `by_user_id()` verwendet. Pro Abfrage werden alle Kalendereinträge des aktuellen Tages gelesen, nach Startzeit sortiert und mit der Zeitzone `W. Europe Standard Time` (Zürich) über die Konfiguration im `get()`-Aufruf zurückgegeben.

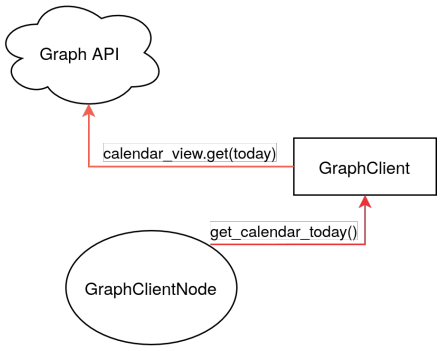


Abbildung 17: Entgegennahme der Einträge

Als vereinfachtes Beispiel liefert die Graph API folgende Antwortstruktur:

graph_response.jsonc · Vereinfacht

```jsonc
{
  "value": [
    {
      "id": "AAMkAGI2...", // Eindeutige ID des Kalendereintrags
      "subject": "Transport Kabel", // Titel des Auftrags
      "start": { // Startzeit inkl. Zeitzone
        "dateTime": "2026-04-10T09:00:00.0000000",
        "timeZone": "W. Europe Standard Time"
      },
      "end": { // Endzeit inkl. Zeitzone
        "dateTime": "2026-04-10T09:30:00.0000000",
        "timeZone": "W. Europe Standard Time"
      },
      "responseStatus": { "response": "accepted" }, // Rueckmeldestatus des Termins
      "organizer": { "emailAddress": { "name": "Max Muster" } }, // Erfasser:in
      "bodyPreview": "Auftrag: Transport Zielperson: Andri Wild..." // Beschreibung vom Erfasser
    }
  ]
}
```

Die entgegengenommene Event-Liste wird anschliessend an den `GraphClientNode` zurückgegeben und in den nachfolgenden Verarbeitungsschritten weiterverwendet. Die Methode wird in einem Intervall von einer Sekunde ausgeführt.

2.5.3 Verarbeitung und Weitergabe der Daten

Die aus Microsoft Graph empfangenen Eventdaten sind für die ROS2-Verarbeitung noch nicht direkt geeignet. Deshalb werden die Einträge in ein internes Auftrags-Objektmodell überführt und um berechnete sowie geparste Felder ergänzt. Das `Task`-Objekt dient dabei als Auftragsobjekt und als `DTO` innerhalb der Lösung[10].

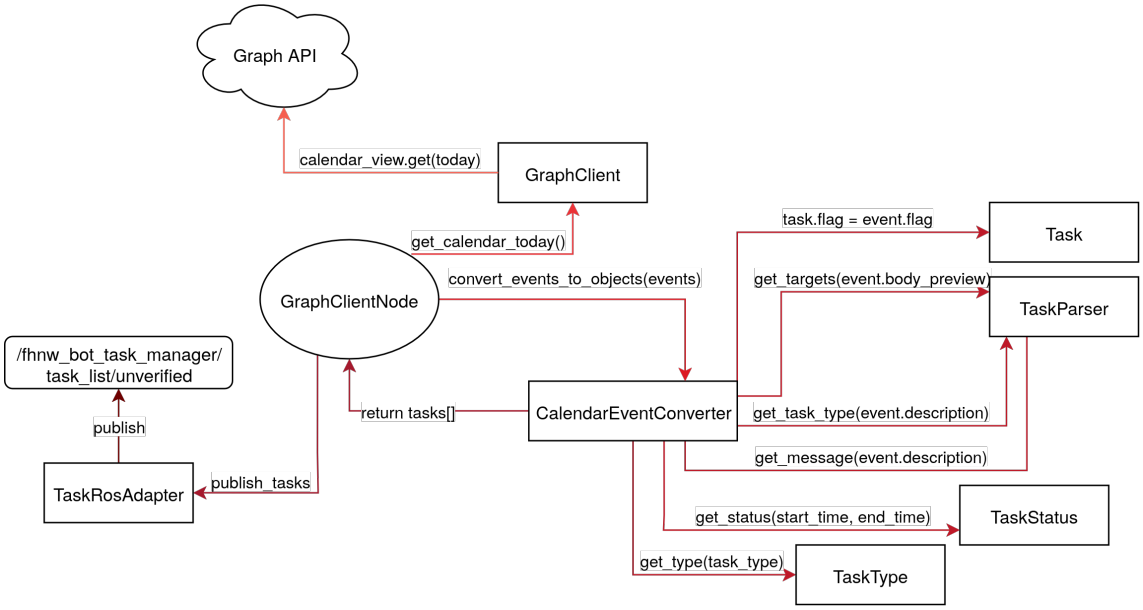


Abbildung 18: Verarbeitung der Einträge

Im `GraphClientNode` werden die Kalendereinträge empfangen und die Konvertierung ausgelöst. Die Klasse `CalendarEventConverter` bildet jedes Event auf das interne `Task`-objekt ab, ergänzt Status- und Parsing-Informationen und übergibt den Auftrag und speichert sie in einer Liste bis alle Events in Aufträge konvertiert wurden und zurückgegeben werden.

Zur Nachvollziehbarkeit wird im Folgenden der Aufbau des internen Auftragsobjekts gezeigt.

Die Event-ID dient als technische Referenz, damit spätere Änderungen am konkreten Auftrag in der Graph API eindeutig zugeordnet werden können. Zusätzlich werden Titel, Start- und Endzeit sowie Antwortstatus und Name der erfassenden Person gespeichert. Diese Felder sind für Statusbestimmung, Validierung und Darstellung im `RViz-Panel` notwendig.

Das Feld `has_description` zeigt an, ob eine Beschreibung vorhanden ist. Aus der Beschreibung werden Auftragstyp, Zielperson(en) und Nachricht extrahiert. Der Zielort bleibt bei der Initialisierung zunächst leer und wird erst im Validierungsschritt durch Datenbankabfragen ergänzt.

task.py · Vereinfacht

```python
@dataclass
class Task:
    """Data model representing a task."""

    event_id: str
    title: str
    start_time: datetime
    end_time: datetime
    status: str
    accepted: bool
    owner_name: str
    parsed_task_type: str
    parsed_targets: list[str]
    parsed_message: str
    destinations: list[Any]
    has_description: bool
```

Bei der Konvertierung wird der Auftragsstatus ermittelt, dieser wird als Teil eines einfachen Statusmodells auf Basis von Start- und Endzeit sowie der lokalen Zeitzone berechnet.

task_status.py · Vereinfacht

```python
def get_status(start_time, end_time, local_timezone):
    now = datetime.now(local_timezone)

    if now < start_time:
        return TaskStatus.PENDING
    if now <= end_time:
        return TaskStatus.ACTIVE
    if now > end_time:
        return TaskStatus.COMPLETED
    return TaskStatus.FAILED
```

Der Status wird zeitabhängig aufgebaut. Liegt die Startzeit in der Zukunft, wird der Auftrag als `PENDING` geführt. Liegt der aktuelle Zeitpunkt zwischen Start- und Endzeit, lautet der Status `ACTIVE`. Nach Ablauf der Endzeit wird der Auftrag als `COMPLETED` markiert.

Das Parsing ist als regelbasierter, deterministischer Parser umgesetzt. Es folgt dem schemaorientierten und template-basierten Schlüssel-Wert-Format aus Abschnitt 2.3.5. Für die Extraktion werden die Wörter des Beschreibungstexts als Grundlage verwendet.

Mit der Funktion `get_task_type` wird der Auftragsmodus anhand der definierten Schlüsselwörter im Beschreibungstext bestimmt. Nach der Normalisierung wird der Text sequenziell durchsucht. Wird das Wort `auftrag` gefunden, entscheidet das nachfolgende Wort über den Typ. Bei `transport` wird ein Transportauftrag erkannt, bei `forward` oder `übermittlung` ein Übermittlungsauftrag. Falls kein passender Marker vorhanden ist, wird `None` zurückgegeben. Die Rückgabe von `None` wird später in der Validierung explizit behandelt.

task_parser.py · Vereinfacht

```python
def get_task_type(self, body):
    try:
        words = self._get_words(body)
        for i, word in enumerate(words):
            if word == "auftrag" and i + 1 < len(words):
                next_word = words[i + 1]
                if next_word == "transport":
                    return "transport"
                if next_word in ("forward", "übermittlung"):
                    return "forward"
                return None
        return None
    except Exception as e:
        self._logger.error(f"Error parsing task type: {str(e)}")
        return None
```

task_parser.py · Vereinfacht

```python
def get_targets(self, body):
    try:
        words = self._get_words(body)
        targets = []
        for i, word in enumerate(words):
            if word not in self._TARGET_MARKERS or i + 1 >=
len(words):
                continue
            target_name = words[i + 1]
            if i + 2 < len(words) and words[i + 2] not in
self._TARGET_MARKERS:
                target_name = f"{target_name} {words[i + 2]}"
            targets.append(target_name)
        return targets
    except Exception as e:
        self._logger.error(f"Error parsing targets: {str(e)}")
        return None
```

Die Zielpersonen werden über die Marker `zielperson`, `zielperson1` und `zielperson2` extrahiert. Nach jedem Marker wird zunächst das folgende Wort als Name übernommen. Ist ein weiterer Name vorhanden und beginnt dort kein neuer Marker, wird dieser als zweiter Namensteil ergänzt. Auf diese Weise entstehen möglichst vollständige Personennamen für die nachfolgende Verarbeitung. Die Logik ist bewusst einfach gehalten und unterstützt die im Projekt definierte Eingabestruktur. Fehlende oder unvollständige Namen führen nicht zu einem Abbruch, sondern werden in der nächsten Stufe validiert.

Mit der Funktion `get_message` wird der freie Nachrichtenteil aus dem Eingabetext gelesen. Ab dem Marker `nachricht` werden alle folgenden Wörter gesammelt, bis ein Zielpersonen-Marker erreicht wird oder der Text endet. Der eigentliche Inhalt der Nachricht bleibt erhalten, während strukturierende Schlüsselwörter ausgeschlossen werden. Dieses Vorgehen stellt sicher, dass die fachlich relevante Mitteilung für den Auftrag vollständig übernommen wird.

task_parser.py · Vereinfacht

```python
def get_message(self, body):
    try:
        words = self._get_words(body)
        message_words = []
        collecting = False
        for word in words:
            if word == "nachricht":
                collecting = True
                continue
            if collecting:
                if word in self._TARGET_MARKERS:
                    break
                message_words.append(word)
        return " ".join(message_words)
    except Exception as e:
        self._logger.error(f"Error parsing message: {str(e)}")
        return None
```

Jeder Kalendereintrag wird im Polling-Intervall in ein internes Auftragsobjekt überführt und auf `/fhnw_bot_task_manager/task_list/unverified` veröffentlicht.

Für die `ROS2-Topics` wird ein gemeinsames `Quality-of-Service-Profil` verwendet. Das Profil legt fest, wie Nachrichten zwischen `Publishern` und `Subscribern` gespeichert und übertragen werden. Für die Auftragslisten sind insbesondere `Reliability`, `History`, `Durability` und `Depth` relevant, da immer der zuletzt bekannte Auftragszustand zuverlässig verfügbar sein muss.

node_utils.py · Vereinfacht

```python
def build_task_list_qos(depth: int = 1) -> QoSProfile:
    return QoSProfile(
        reliability=ReliabilityPolicy.RELIABLE,
        history=HistoryPolicy.KEEP_LAST,
        durability=DurabilityPolicy.TRANSIENT_LOCAL,
        depth=depth,
    )
```

Auf Basis von Abschnitt 2.3.4 wurde für alle Auftragslisten-`Topics` `RELIABLE`, `KEEP_LAST`, `TRANSIENT_LOCAL` und eine `depth` von 1 verwendet. Mit `RELIABLE` wird die Zustellung der Nachrichten priorisiert, da die nachfolgenden `Nodes` auf vollständige Auftragslisten angewiesen sind. Mit `KEEP_LAST` und `depth = 1` wird nur der zuletzt veröffentlichte Zustand gehalten, da ältere Zustände für die Weiterverarbeitung nicht benötigt werden. `TRANSIENT_LOCAL` stellt sicher, dass später startende `Subscriber`, beispielsweise das `RViz-Panel` nach einem Neustart, sofort den letzten veröffentlichten Auftragszustand erhalten und nicht auf die nächste Veröffentlichung warten müssen. Weitere QoS-Parameter wurden auf den Standardwerten belassen, weil sie für diesen Kommunikationspfad keinen zusätzlichen Nutzen bringen.

2.5.4 Validierung der Aufträge

Die Validierung entscheidet, ob ein Kalendereintrag fachlich und technisch ausführbar ist. Ungültige Einträge werden früh gestoppt. Gültige Aufträge werden mit Zielinformationen für die Roboterkomponente angereichert.

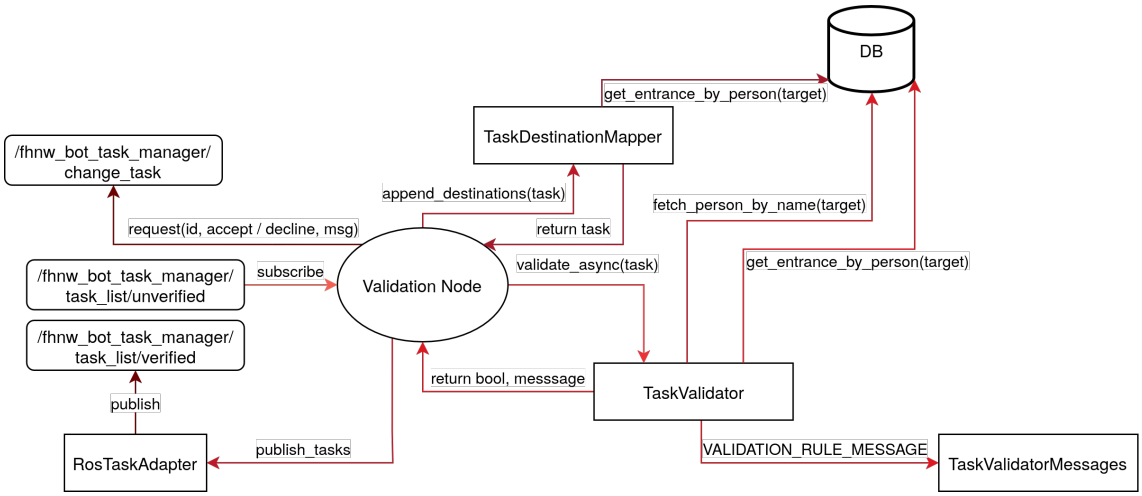


Abbildung 19: Validieren eines Auftrags

Die Validierung ist im `ValidationNode` platziert, weil dort der Zugriff auf die Datenbank-API als Datenzugriffsschicht gekapselt ist. Die Datenbank-API bildet eine Abstraktionsschicht zum `Persistence Layer` für Personen- und Raumdaten. Der `Node` liest Aufträge aus `/fhnw_bot_task_manager/task_list/unverified`, validiert sie mit `TaskValidator` und meldet das Ergebnis über `/fhnw_bot_task_manager/change_task` an den `GraphClientNode`. Für akzeptierte Aufträge ergänzt `TaskDestinationMapper` anschliessend das Feld `destinations`.

Bei der Initialisierung werden dafür die zentralen Komponenten und Schnittstellen erstellt.

validation_node.py · Vereinfacht

```python
class ValidationNode(Node):
  def __init__(self):
    super().__init__("validation_node")

    self._validator = TaskValidator(self.get_logger())
    self._destination_mapper = TaskDestinationMapper(self.get_logger())

    # Sets zum Merken von event_ids
    self._events_in_process = set()  # wird gerade verarbeitet
    self._handled_event_ids = set()  # schon erfolgreich behandelt

    self._unverified_tasks_subscriber = self.create_subscription(
      TaskList,
      TASK_LIST_UNVERIFIED_TOPIC,
      self._unverified_tasks_callback,
      ...
    )

    self._change_task_client = self.create_client(
      ChangeTask,
      CHANGE_TASK_SERVICE
    )
```

Für die spätere Ablaufsteuerung werden die beiden Sets `_events_in_process` und `_handled_event_ids` verwendet. Ein `set` speichert mehrere Werte als ungeordnete Sammlung ohne Indexzugriff. Für die Ablaufsteuerung reicht diese Eigenschaft aus, weil nur geprüft werden muss, ob eine Event-ID bereits vorhanden ist. Damit wird verhindert, dass derselbe Auftrag parallel oder mehrfach verarbeitet wird.

Das `Topic /fhnw_bot_task_manager/task_list/unverified` wird abonniert und mit einem `Callback` verknüpft. Immer wenn neue Daten eintreffen, startet dieser `Callback` den Validierungsablauf.

validation_node.py · Vereinfacht

```python
def _unverified_tasks_callback(self, msg: TaskList):
    # zuerst unverified tasks prüfen
    self._process_unverified_tasks(msg)

    # danach akzeptierte tasks weiterleiten
    verified_tasks = []
    for task in msg.tasks:
        if task.accepted:
            self._destination_mapper.append_destinations(task)
            verified_tasks.append(task)

    TaskRosAdapter.publish_tasks(
        verified_tasks,
        msg.sync_ok,
        msg.sync_message,
        self._verified_tasks_publisher,
        node=self
    )
```

Dieser `Callback` bildet den Einstiegspunkt für neue Aufträge. Zuerst werden die unverifizierten Aufgaben in `_process_unverified_tasks` geprüft. Danach werden nur akzeptierte Aufträge (`accepted == true`) mit Zielorten ergänzt und auf `/fhnw_bot_task_manager/task_list/verified` veröffentlicht.

Die Zielorte werden über die Klasse `TaskDestinationMapper` ergänzt. Dabei wird für jede geparste Zielperson die passende Raumzuordnung aus der Datenbank geladen und in das Nachrichtenformat für ROS2 überführt. Jeder gültige Treffer wird als `RoomEntranceMsg` im Feld `destinations` des Auftrags gespeichert. Ungültige oder unvollständige Einträge, z. B. fehlende Raumdaten oder Quaternionen mit falscher Länge, werden als ungültige Referenz behandelt, übersprungen und geloggt. Die Längenprüfung erfolgt mit `len(...)`. So erhalten die nachfolgenden Komponenten nur vollständig aufgelöste und technisch konsistente Zielinformationen.

task_destination_mapper.py · Vereinfacht

```python
def append_destinations(self, task):
    for target in task.parsed_targets:
        entrance = api.get_entrance_by_person(target)
        if entrance is None:
            self._logger.warning(f'No entrance found for "{target}"')
            continue

        if not entrance.quat or len(entrance.quat) != 4:
            self._logger.warning(f'Invalid quaternion for "{target}"')
            continue

        destination = RoomEntranceMsg()
        destination.id = int(entrance.id)
        destination.coord = Point(x=float(entrance.coord.x), y=float(entrance.coord.y))
        destination.quat = [float(value) for value in entrance.quat]
        task.destinations.append(destination)
```

Diese Zuordnung liegt im `ValidationNode`, weil dort sowohl das validierte Task-Objekt als auch der Zugriff auf die Datenzugriffsschicht vorhanden sind. Die eigentliche Prüfung auf Zielpersonen und Räume erfolgt bereits in der Validierung. Im `TaskDestinationMapper` werden die geprüften Daten nur noch in das ROS2-Nachrichtenformat überführt und technisch abgesichert. So bleiben Datenanreicherung und Datenbankabhängigkeit zentral.

In `_process_unverified_tasks` werden die Aufträge regelbasiert validiert und anschliessend per `Service`-Aufruf akzeptiert oder abgelehnt.

validation_node.py · Vereinfacht

```python
def _process_unverified_tasks(self, msg: TaskList):
    for task in msg.tasks:
        if task.accepted:
            continue
        if task.event_id in self._handled_event_ids:
            continue
        if task.event_id in self._events_in_process:
            continue

        self._events_in_process.add(task.event_id)

        try:
            is_valid, result_messages = self._validator.validate(task)

            request = build_change_task_request(
                event_id=task.event_id,
                change_type="accept" if is_valid else "decline",
                message=result_messages,
            )

            future = self._change_task_client.call_async(request)
            future.add_done_callback(
                self._make_change_task_done_callback(task.event_id)
            )

        except Exception:
            self._events_in_process.discard(task.event_id)
            raise
```

In dieser Funktion wird jeder Auftrag nacheinander verarbeitet. Bereits akzeptierte, laufende oder bereits abgeschlossene Event-IDs werden übersprungen. Für alle übrigen Aufträge ruft der `Node validate(task)` auf und erhält als Ergebnis einen booleschen Status sowie eine Liste von Rückmeldungen.

Die Validierung erfolgt in drei Regelgruppen. Diese Gruppen strukturieren die definierten Anforderungen aus Abschnitt 2.3.5 an einen Auftrag. Zuerst werden Basisregeln geprüft, beispielsweise Constraint-Prüfungen zu Zeitfenster, Owner, Titel und vorhandener Beschreibung. Zeitliche Überschneidungen werden als Konflikterkennung behandelt. Danach folgen Parsing-Regeln wie vollständige Parserdaten, gültiger Auftragstyp und ausreichende Nachrichtenlänge. Abschliessend werden Zielregeln geprüft, insbesondere die Zielanzahl, die Existenz der Zielpersonen in der Datenbank und die vorhandene Raumzuordnung.

Ungültige oder widersprüchliche Daten werden nicht still verworfen. Stattdessen werden die Regelverletzungen gesammelt und als konkrete Rückmeldung an den `Service`-Aufruf übergeben, damit der Erfasser den Ablehnungsgrund nachvollziehen kann. Der `Service`-Aufruf erfolgt asynchron, sodass der `Node` währenddessen weitere Aufträge verarbeiten kann. Das `Future` hält dabei die spätere Antwort, und `add_done_callback` übernimmt die Weiterverarbeitung nach Abschluss[11].

validation_node.py · Vereinfacht

```python
def _make_change_task_done_callback(self, event_id: str):
  def callback(future):
    try:
      response = future.result()

      if response.success:
        self._handled_event_ids.add(event_id)

    finally:
      self._events_in_process.discard(event_id)

  return callback
```

Sobald die `Service`-Antwort eintrifft, wird die Event-ID bei erfolgreicher Verarbeitung in `_handled_event_ids` eingetragen und in jedem Fall aus `_events_in_process` entfernt. Damit bleibt der interne Verarbeitungszustand konsistent und derselbe Auftrag wird nicht erneut parallel bearbeitet.

2.5.5 Ausführen eines Auftrags

Der `RunTaskNode` löst den Zustandsübergang vom validierten Auftrag zur simulierten Navigation. Er wählt fällige Aufträge aus, sendet sie an das Navigations-Mock und leitet aus dessen Antwort den fachlichen Status `completed` oder `failed` ab.

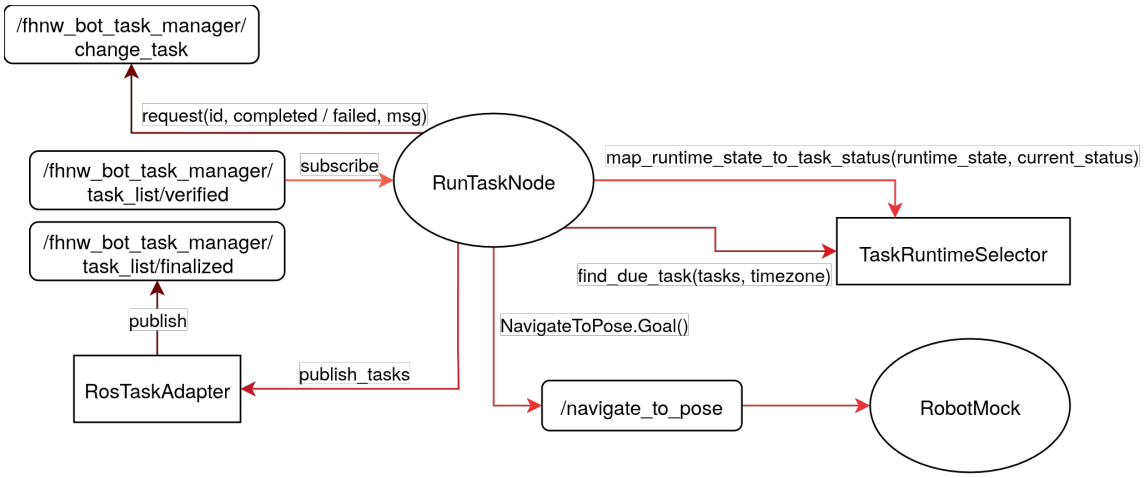


Abbildung 20: Ausführen eines Auftrags

Für diese Aufgabe wurde die Klasse `TaskRuntimeSelector` definiert. Sie überführt interne Runtime-Zustände in sichtbare Auftragsstatus, konvertiert ROS2-Zeitwerte in lokale Datumswerte und wählt fällige Aufträge aus. Damit übernimmt sie einen Teil der Statussteuerung im Auftragslebenszyklus.

run_task_node.py · Vereinfacht

```python
class RunTaskNode(Node):
    def __init__(self):
        super().__init__("run_task_node")

        self._processing = False
        self._started_event_ids = set()  # bereits gestartete Tasks
        self._task_states = {}           # Runtime-State pro event_id
        self._finalized_tasks = []

    def _set_task_state(self, task, state):
        task.status = map_runtime_state_to_task_status(state, task.status)

        if task.event_id not in self._task_states:
            self._task_states[task.event_id] = {}

        self._task_states[task.event_id]["task"] = task
        self._task_states[task.event_id]["state"] = state

        self._update_finalized_task(task)
        self._publish_finalized_tasks()
```

Die Laufzeitdaten verhindern doppelte Ausführungen und halten den sichtbaren Status synchron. `_processing` zeigt an, ob gerade ein Auftrag verarbeitet wird, `_started_event_ids` speichert bereits gestartete Aufträge, und `_task_states` hält pro `event_id` den aktuellen Runtime-State.

In `_set_task_state` wird der fachliche Auftragsstatus aus dem Runtime-State abgeleitet. Dieser Schritt entspricht einem Statuswechsel im Auftragslebenszyklus. `failed` und `rejected` werden auf den Status `failed` abgebildet, `finished` wird zu `completed` und `running` zu `active`. Danach wird der aktualisierte Auftrag in `_finalized_tasks` übernommen und auf `/fhnw_bot_task_manager/task_list/finalized` publiziert.

run_task_node.py · Vereinfacht

```
def _verified_tasks_callback(self, msg):
    self._finalized_tasks = list(msg.tasks)

    for task in self._finalized_tasks:
        runtime_state = self._task_states.get(task.event_id, {}).get("state")
        task.status = map_runtime_state_to_task_status(runtime_state, task.status)

    self._publish_finalized_tasks()

    if self._processing:
        return

    task = find_due_task(msg.tasks, self._local_timezone)
    if task is None:
        return
    if task.event_id in self._started_event_ids:
        return

    self._started_event_ids.add(task.event_id)
    self._processing = True
    self._set_task_state(task, "running")
    self._start_destination(task, 0)
```

Der `Callback _verified_tasks_callback` wird durch neue Daten auf `/fhnw_bot_task_manager/task_list/verified` ausgelöst. Zuerst werden alle Aufträge in `_finalized_tasks` übernommen und mit vorhandenen Runtime-States abgeglichen. Danach werden die Aufträge erneut publiziert.

Falls aktuell kein Auftrag verarbeitet wird, wird mit `find_due_task` der nächste fällige Auftrag bestimmt. Wurde dieser Auftrag noch nicht gestartet, wird seine `event_id` in `_started_event_ids` gespeichert. Danach wird `_processing` auf `True` gesetzt, der Auftrag in den Runtime-State `running` versetzt und das erste Navigationsziel über `_start_destination` gestartet.

run_task_node.py · Vereinfacht

```python
def _start_destination(self, task, index):
    destination = task.destinations[index]
    self._task_states[task.event_id]["destination_index"] = index
    self._send_goal(task, destination)

def _send_goal(self, task, destination):
    goal = NavigateToPose.Goal()
    goal.pose.header.frame_id = self._navigation_frame_id
    goal.pose.pose.position.x = destination.coord.x
    goal.pose.pose.position.y = destination.coord.y
    goal.pose.pose.orientation.x = destination.quat[0]
    goal.pose.pose.orientation.y = destination.quat[1]
    goal.pose.pose.orientation.z = destination.quat[2]
    goal.pose.pose.orientation.w = destination.quat[3]

    future = self._send_goal_action_client.send_goal_async(goal)
    future.add_done_callback(lambda fut: self._goal_response_callback(fut, task))
```

In `_start_destination` wird der übergebene `index` verwendet, um den aktuellen Zielort aus `task.destinations` auszuwählen. Gleichzeitig wird dieser Index in `_task_states` unter der `event_id` des Auftrags gespeichert. So bleibt der Bearbeitungsstand des Auftrags auch über asynchrone Callback-Aufrufe hinweg erhalten. Der gespeicherte Wert dient dazu, das aktuell ausgeführte Ziel eindeutig zu identifizieren und nach dessen Abschluss den nächsten Zielort zu ermitteln. An `_send_goal` wird anschliessend der ausgewählte Zielort übergeben. Dort wird anhand seiner Koordinaten und Quaternionen ein `Action-Goal` erzeugt und an das Navigations-Mock gesendet.

run_task_node.py · Beispiel für _task_states

```python
self._task_states = {
    "AAMkAGI2...AAA=": {
        "task": "<Task event_id=AAMkAGI2...AAA=>",
        "state": "running",
        "destination_index": 1,
    },
    "AAMkAGI2...BBB=": {
        "task": "<Task event_id=AAMkAGI2...BBB=>",
        "state": "finished",
        "destination_index": 0,
    },
}
```

Im ersten Eintrag zeigt `destination_index: 1`, dass gerade der zweite Zielort dieses Auftrags verarbeitet wird.

run_task_node.py · Vereinfacht

```python
def _goal_response_callback(self, future, task):
    goal_handle = future.result()

    if not goal_handle.accepted:
        self._set_task_state(task, "rejected")
        self._send_change_task(
            task,
            finished=False,
            result_messages=TASK_GOAL_REJECTED_MESSAGE,
        )
        self._processing = False
        return

    self._set_task_state(task, "running")
    result_future = goal_handle.get_result_async()
    result_future.add_done_callback(lambda fut: self._result_callback(fut, task))

def _result_callback(self, future, task):
    status = future.result().status
    action_result = future.result().result

    if status == GoalStatus.STATUS_SUCCEEDED:
        self._set_task_state(task, "finished")
        self._send_change_task(
            task,
            finished=True,
            result_messages=TASK_FINISHED_MESSAGE,
        )

    elif status in (GoalStatus.STATUS_ABORTED, GoalStatus.STATUS_CANCELED):
        self._set_task_state(task, "failed")
        self._send_change_task(
            task,
            finished=False,
            result_messages=self._build_navigation_failure_message(action_result),
        )

    self._processing = False
```

Das Navigations-Mock meldet zuerst, ob das `Goal` akzeptiert wurde. Bei Ablehnung wird der Auftrag als `rejected` beziehungsweise `failed` zurückgemeldet. Bei Annahme bleibt er auf `running`, bis `_result_callback` ihn je nach Ergebnis auf `finished` oder `failed` setzt. Die Rückmeldung erfolgt über `/fhnw_bot_task_manager/change_task` an den `GraphClientNode`, aktualisiert das `RViz-Panel` und informiert den Erfasser.

2.5.6 Visualisierung im `RViz-Panel`

Das `RViz-Panel` bildet die Visualisierungsschicht für die Überwachung der Tagesaufträge. Es zeigt den aktuellen Stand an und ermöglicht ausschliesslich die manuelle Stornierung bereits erfasster Aufträge.

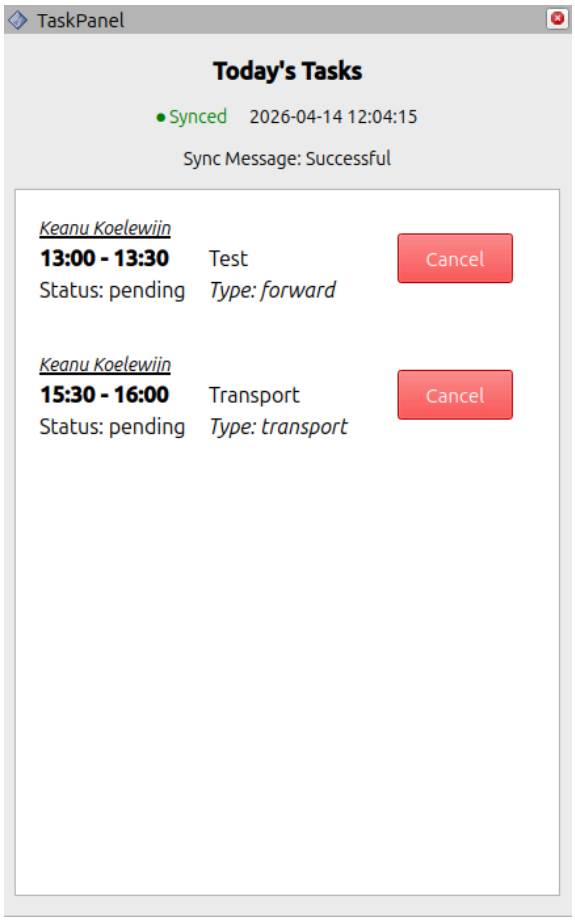


Abbildung 21: Visualisierung im `RViz-Panel`

Die Vorgabe für C++ mit Qt stammt aus Abschnitt 2.2.1, die `RViz-Panel`-Integration aus Abschnitt 2.3.4. Die Struktur orientiert sich an vorhandenen `RViz-Panel`-Beispielen und am Mockup aus Abschnitt 2.3.6[12,13]. Angezeigt werden Erfasser, Zeitfenster, Auftragstitel, Status und Auftragstyp. Zusätzlich wird pro Auftrag eine Stornierungsaktion bereitgestellt, sofern der aktuelle Status diese Interaktion zulässt.

Um Anzeige und Interaktion übersichtlich zu halten, wurde die Visualisierung in das Panel selbst und einzelne Auftragskarten aufgeteilt. Die `.hpp`-Dateien dienen dabei als Interface zur C++-Logik. Sie deklarieren Klassen, öffentliche Methoden, `Signals`, `Slots` und benötigte Attribute. Die eigentliche Implementierung liegt in den zugehörigen `.cpp`-Dateien. Der folgende Auszug zeigt den Aufbau der Auftragskarte.

task_card.hpp · Vereinfacht

```
namespace fhnw_rviz_task_panel {

class TaskCard : public QWidget {
  Q_OBJECT

public:
  explicit TaskCard(QWidget *parent = nullptr);
  void setTask(const fhnw_custom_interfaces::msg::Task &task);

signals:
  void cancelRequest(const QString &event_id);

private slots:
  void cancelClicked();

private:
  void updateButtonBasedOnStatus(const std::string &status);
  std::string formatTime(
    const builtin_interfaces::msg::Time &start_time,
    const builtin_interfaces::msg::Time &end_time
  );

  std::string id;
  QLabel *owner_label, *title_label, *status_label, *task_type_label, *time_label;
  QPushButton *cancel_button;
};

} // namespace fhnw_rviz_task_panel

#endif
```

Die Auftragskarte erbt von `QWidget` und wird als eigenes `Qt-Widget` im `task_panel` verwendet. Mit `Q_OBJECT` werden Qt-Funktionen wie `Signals` und `Slots` aktiviert. Über `setTask` wird eine Karte mit den Daten eines Auftrags befüllt. Das `Signal cancelRequest` wird ausgelöst, wenn der Stornieren-Button betätigt wird. Der `Slot cancelClicked` reagiert auf diese Interaktion und leitet die zugehörige `event_id` weiter.

Im `task_panel.hpp` werden der `Subscriber` für `/fhnw_bot_task_manager/task_list/finalized` und der `Service-Client` für `/fhnw_bot_task_manager/change_task` definiert. Zusätzlich werden die Funktionen `topicCallback`, `updateTaskList`, `updateConnection`, `handleCancelRequest` und `hasTopicChanged` sowie die GUI-Elemente für Titel, Synchronisationsinformationen und Auftragsliste deklariert.

Der Aufbau einer Auftragskarte erfolgt im Konstruktor von `TaskCard`.

task_card.cpp · Vereinfacht

```cpp
TaskCard::TaskCard(QWidget *parent) : QWidget(parent)
{
    title_label = new QLabel(this);
    owner_label = new QLabel(this);
    status_label = new QLabel(this);
    time_label = new QLabel(this);
    cancel_button = new QPushButton("Cancel", this);

    auto layout = new QGridLayout(this);
    layout->addWidget(owner_label, 0, 0);
    layout->addWidget(time_label, 1, 0);
    layout->addWidget(title_label, 1, 1);
    layout->addWidget(status_label, 2, 0);
    layout->addWidget(cancel_button, 0, 2, 3, 1);

    connect(cancel_button, &QPushButton::clicked, this, &TaskCard::cancelClicked);
}
```

Im Konstruktor von `TaskCard` werden die GUI-Elemente erstellt. Die Labels zeigen Auftragsinformationen an, während der `Cancel`-Button die Stornierungsfunktion bereitstellt. Über das Layout werden die Elemente angeordnet. Mit `connect(...)` wird der Button-Klick mit dem Slot `cancelClicked` verbunden.

task_card.cpp · Vereinfacht

```cpp
void TaskCard::setTask(const fhnw_custom_interfaces::msg::Task &task)
{
    owner_label->setText(QString::fromStdString(task.owner_name));
    title_label->setText(QString::fromStdString(task.title));
    status_label->setText("Status: " + QString::fromStdString(task.status));
    time_label->setText(QString::fromStdString(formatTime(task.start_time, task.end_time)));

    updateButtonBasedOnStatus(task.status);
    id = task.event_id;
}
```

Mit dieser Methode werden die Daten eines Auftrags in die Karte übernommen. Die Labels werden mit Erfasser, Titel, Status und Zeitfenster befüllt. Die ROS2-Zeit wird mit `formatTime` in ein lesbares Format umgewandelt, beispielsweise von Startzeit `1775822400` und Endzeit `1775824200` zu `14:00 - 14:30`. Über `updateButtonBasedOnStatus` wird der Button je nach Status angepasst. Abgeschlossene Aufträge werden deaktiviert dargestellt, damit sie nicht storniert werden können.

Abschliessend wird die `event_id` gespeichert, damit die spätere Stornierungsanfrage eindeutig einem Auftrag zugeordnet werden kann.

Diese Funktion wird ausgeführt, wenn der Button gedrückt wird. Dabei wird eine Stornierungsanfrage mit der zugehörigen `event_id` ausgelöst.

task_card.cpp · Vereinfacht

```cpp
void TaskCard::cancelClicked()
{
  if (!id.empty()) {
    cancelRequest(QString::fromStdString(id));
  }
}
```

Im `task_panel.cpp` wird das `RViz-Panel` aufgebaut und mit `ROS2` verbunden. Der Konstruktor erstellt Titel, Synchronisationsstatus, Synchronisationsmeldung, Zeitstempel und die Liste für die späteren Auftragskarten. Danach werden die Elemente so angeordnet, dass oben ein Übersichtsbereich und darunter die Auftragsliste entsteht.

In `onInitialize()` wird das Panel mit ROS2 verbunden. Dafür wird ein `Subscriber` auf `/fhnw_bot_task_manager/task_list/finalized` erstellt und ein `Service-Client` für den `/fhnw_bot_task_manager/change_task` angelegt. Über das `Topic` werden finalisierte Auftragsdaten empfangen. Über den `Service` werden spätere Änderungen wie Stornierungen gesendet.

Bei einer neuen `TaskList` ruft ROS2 `topicCallback()` auf. Die Funktion übernimmt die Nachricht und gibt sie an `updateConnection()` und `updateTaskList()` weiter. Dadurch werden die Synchronisationsinformationen und die sichtbare Auftragsliste aktualisiert.

`updateTaskList()` stellt die Aufträge im Panel dar. Zuerst prüft `hasTopicChanged()`, ob sich die Anzahl der Aufträge oder wichtige Eigenschaften wie `event_id`, `status` oder `accepted` geändert haben. Ohne relevante Änderung bleibt die Anzeige unverändert.

Bei einer Änderung wird die bisherige Liste geleert. Danach wird für jeden Auftrag eine `TaskCard` erstellt, mit Auftragsdaten befüllt und in die Liste eingefügt. Zusätzlich wird jede Karte mit `handleCancelRequest()` verbunden, damit Stornierungsanfragen aus der Benutzeroberfläche weiterverarbeitet werden können.

`updateConnection()` aktualisiert die Synchronisationsinformationen. Dabei werden der Synchronisationserfolg, die Synchronisationsnachricht und der Zeitstempel der empfangenen Liste angezeigt. Der Zeitstempel wird in ein lesbares Datums- und Uhrzeitformat umgewandelt.

Wenn der Stornieren-Button einer Auftragskarte gedrückt wird, ruft die Karte `handleCancelRequest()` auf. Dort wird eine `Service`-Anfrage mit der `event_id` des ausgewählten Auftrags, dem Änderungstyp `cancel` und einer passenden Nachricht vorbereitet. Danach wird geprüft, ob der `Service` erreichbar ist. Ist dies der Fall, wird die Anfrage asynchron gesendet.

2.5.7 Stornieren oder Aktualisieren eines Auftrags

Änderungen an Aufträgen müssen zentral synchronisiert werden, damit Kalender, Rückmeldungen und `RViz`-Anzeige konsistent bleiben. Dafür wird der `Service` `/fhnw_bot_task_manager/change_task` als `Request-Response-Kommunikation` verwendet.

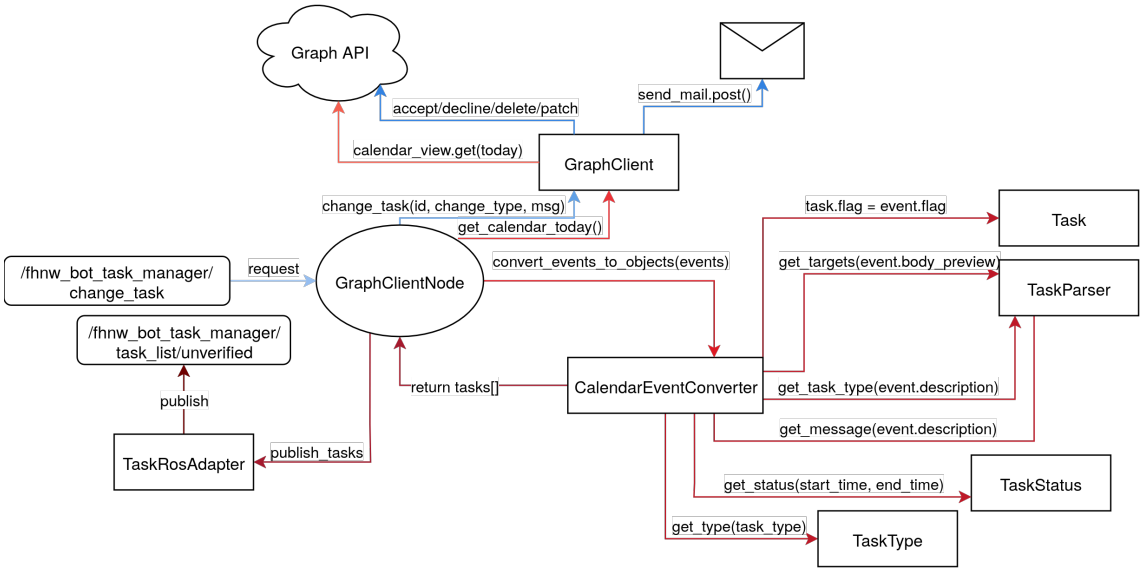


Abbildung 22: Bearbeiten eines Auftrags

Die beteiligten `Nodes` senden je nach Verarbeitungsschritt eine Anfrage mit der `event_id` des Auftrags, dem `change_type` und einer Rückmeldungsnachricht. Über den `change_type` verarbeitet der `GraphClientNode` Validierungsergebnisse, Stornierungen aus dem `RViz-Panel` und Abschlussmeldungen.

graph_client_node.py · Vereinfacht

```python
class GraphClientNode(Node):
  def __init__(self):
    super().__init__("graph_client_node")
    self.create_service(
      ChangeTask, CHANGE_TASK_SERVICE, self._change_task_callback
    )

  def _change_task_callback(self, request, response):
    try:
      response.message = self._graph_client_instance.change_task(
        request.event_id,
        request.change_type,
        request.message
      )
      response.success = True
    except Exception as e:
      response.success = False
      response.message = str(e)
    return response
```

Der `Service` wird im `GraphClientNode` initialisiert und mit dem `Callback` `_change_task_callback` verbunden. Eingehende `Requests` werden an `change_task` weitergegeben. Anschliessend wird eine `Response` mit Erfolgsstatus und Meldung an den anfragenden `Node` zurückgegeben.

graph_client.py · Vereinfacht

```python
def change_task(self, event_id, change_type, message):
    if change_type in ("accept", "decline"):
      self._send_validation(event_id, change_type, message)
    elif change_type == "cancel":
      self._cancel_task(event_id, message)
    elif change_type in ("completed", "failed"):
      self._send_task_result(event_id, change_type, message)
    else:
      self._logger.error(f"Error unknown change type: {change_type}")
```

Je nach `change_type` werden unterschiedliche Aktionen ausgelöst. `accept` und `decline` stammen aus der automatischen Validierung und verwenden die entsprechenden Graph-Operationen[14,15]. `cancel` wird durch die Stornierungsaktion im `RViz-Panel` ausgelöst und storniert den Auftrag mit einer E-Mail an den Erfasser[16]. Bei `completed` oder `failed` wird die Endzeit des Auftrags aktualisiert und ebenfalls eine Rückmeldung versendet[17,18]. Der finale Status wird zusätzlich im `RunTaskNode` geführt und auf `/fhnw_bot_task_manager/task_list/finalized` publiziert, damit das `RViz-Panel` den aktuellen Stand anzeigen kann.

2.6 Kontrollieren

In der Kontrollphase wird geprüft, ob die umgesetzte Lösung die Anforderungen aus Abschnitt 2.2.1 erfüllt. Dazu werden die Testumgebung, die Testergebnisse, erkannte Abweichungen und der Projektfortschritt ausgewertet.

2.6.1 Testumgebung

Die Tests wurden in der vorgesehenen Entwicklungsumgebung durchgeführt. Verwendet wurden `ROS2`, `RViz`, die Microsoft Graph API, die interne Datenbank-API und das bereitgestellte Navigations-Mock. Der physische Roboter wurde nicht direkt eingesetzt, sondern ein Navigations-Mock.

Als Testdaten dienten vorbereitete Kalendereinträge mit gültigen und ungültigen Freitexten. Dabei wurden unterschiedliche Auftragsarten, Zeitfenster, Zielpersonen und Statuszustände geprüft. Zusätzlich wurden Szenarien für manuelle Stornierung, automatische Ablehnung und Rückmeldungen an den Erfasser berücksichtigt.

2.6.2 Automatisierte Testfälle

Die wichtigsten Verarbeitungsschritte wurden zusätzlich mit automatisierten Tests abgesichert. Sie prüfen einzelne Komponenten isoliert und ergänzen die fachlichen Tests im Testprotokoll.

`test_calendar_event_converter.py` prüft, ob Kalendereinträge korrekt in interne `Task`-Objekte umgewandelt werden. Dazu gehören Basisfelder, Annahmestatus, Zeitzonen, Statusberechnung und geparste Werte aus der Beschreibung.

`test_graph_client.py` prüft die Änderungslogik im `GraphClient`. Dabei wird kontrolliert, ob Annahmen, Ablehnungen, Stornierungen und Abschlussstatus an die richtige interne Methode weitergeleitet werden. Zusätzlich werden unbekannte Änderungstypen und der Aufbau der E-Mail-Rückmeldungen geprüft.

`test_task_destination_mapper.py` prüft die Zuordnung von Zielpersonen zu Zielkoordinaten. Gültige Datenbanktreffer werden als Ziele in das `Task`-Objekt übernommen. Fehlende Treffer und ungültige Quaternionen werden übersprungen und protokolliert.

`test_task_parser.py` prüft den regelbasierten Parser für den Freitext. Abgedeckt werden Auftragstyp, Zielpersonen und Nachricht sowie das Verhalten bei unvollständigen Eingaben.

`test_task_runtime_selector.py` prüft die Statuslogik während der Ausführung. Der Test stellt sicher, dass Runtime-Zustände auf fachliche Statuswerte abgebildet werden und dass nur fällige, noch nicht abgeschlossene Aufträge ausgewählt werden.

2.6.3 Testprotokoll

Das Testprotokoll fasst die durchgeführten Testfälle zusammen. Die Detailbeschreibung der Testfälle befindet sich in Abschnitt 2.3.7.2. Bemerkungen werden nur dort ergänzt, wo sie für die Einordnung des Resultats relevant sind.

Nr.	**Resultat (OK/ NOK)**	**Bemerkungen**
1.1	OK	-
1.2	OK	Änderungen werden erkannt und übernommen. Bereits akzeptierte Aufträge werden nicht erneut vollständig validiert.
2.1	OK	-
2.2	OK	-
3.1	OK	-
3.2	OK	-
4.1	OK	-
4.2	OK	-
4.3	OK	Bei der ersten Durchführung wurden bereits akzeptierte Aufträge in der Validierung übersprungen. Nach der Korrektur wurde der Test erneut geprüft.
5.1	OK	-
5.2	OK	-
5.3	OK	-
6.1	OK	-
6.2	OK	-
7.1	OK	-
NF.1	OK	-
NF.2	OK	-
NF.3	OK	-
NF.4	OK	-
E2E.1	OK	-
E2E.2	OK	-
E2E.3	OK	-

Die Ergebnisse zeigen, dass die geprüften Kernfunktionen erfolgreich ausgeführt wurden. Die einzige relevante Abweichung trat beim Testfall `4.3` auf und wurde während der Kontrolle korrigiert.

2.6.4 Abweichungen und Fehleranalyse

Die wichtigste Abweichung betraf den Testfall `4.3` aus Abschnitt 2.3.7.2. Bereits akzeptierte Aufträge wurden im `ValidationNode` übersprungen und dadurch nicht in die Konflikterkennung übernommen. Dadurch konnte eine erneute Reservation eines bereits belegten Zeitfensters fachlich falsch bewertet werden.

Bei Testfall `1.2` aus Abschnitt 2.3.7.2 wurde schwach präzisiert, dass Änderungen erkannt und übernommen werden. Bereits akzeptierte Aufträge werden dabei nicht erneut validiert. Wie formuliert war dieser Test erfolgreich, aber eine neue Validierung wäre hier noch robuster gewesen.

validation_node.py · Ausschnitt

```python
    def _process_unverified_tasks(self, msg: TaskList):
        for task in msg.tasks:
            if (
                task.accepted
                or task.event_id in self._handled_event_ids
                or task.event_id in self._events_in_process
            ):
                self._validator.reserve_time_window(task)
```

Zur Behebung wurde eine öffentliche Methode in der Klasse `TaskValidator` ergänzt. Damit werden bereits akzeptierte Aufträge in die Zeitfensterprüfung übernommen, bevor neue Aufträge validiert werden.

task_validator.py · Ausschnitt

```python
def reserve_time_window(self, task: Task):
    try:
        start = Time.from_msg(task.start_time)
        end = Time.from_msg(task.end_time)
        self._is_not_overlapping(start, end)
    except Exception as e:
        self._logger.error(f"reserve_time_window failed: {e}")
```

Nach der Anpassung wurde der Testfall `4.3` erneut geprüft. Der Konflikt wird nun erkannt und der Test wird als bestanden geführt.

2.6.5 Kontrolle des Projektfortschritts

Der Projektfortschritt wurde mit Abschnitt 2.3 und Abschnitt 1.7 abgeglichen. Die zentralen Arbeitsergebnisse wurden erreicht. Dazu gehören der Kalenderabruf über die Microsoft Graph API, das regelbasierte Parsing, die fachliche Validierung, die Weitergabe an das Navigations-Mock, die Statusverarbeitung und die Anzeige im `RViz-Panel`.

2.6.6 Einsatz von KI-Modellen

KI-Modelle wurden im Projekt gezielt für Formulierungs-, Struktur- und Qualitätsprüfungen eingesetzt. Die Inhalte wurden nie automatisiert übernommen, sondern jeweils fachlich geprüft und manuell entschieden.

Verwendete Modelle	`GPT-5.4 ChatGPT` und `GPT-5.3 Codex`
Einsatzzweck	Formulierungshilfe, Strukturverbesserung und Abgleich mit Kriterien und Coderichtlinien.
Steuerung der Eingaben	Prompts mit klarem Ziel, Kontext und erwarteter Ausgabe formuliert.
Objektive Prüfung	Alle Vorschläge gegen Aufgabenstellung, Implementierungsstand und Testergebnisse geprüft.
Übernahmeprinzip	Die ersten zwei Vorschläge wurden testweise automatisch übernommen und danach geprüft und manuell überarbeitet. Alle weiteren Änderungen wurden erst nach manueller Prüfung übernommen.

Modell	Datum	Prompt-Ziel	Einsatz und Nachweis
`GPT-5.3 Codex`	`09.04.2026`	«Create user friendly error strings for the TaskValidator. The text should be German, and can be used as Email context.»	Die Vorschläge wurden zuerst automatisch übernommen. Danach wurden sie in `task_validator_messages.py` geprüft und manuell überarbeitet.
`GPT-5.3 Codex`	`08.04.2026`	«According to these Coding Requirements, I am not allowed to use magic numbers or strings so transfer them to own variables named on their purpose explaining what they mean to increase readability of the code.»	Die Refactoring-Vorschläge wurden zuerst automatisch übernommen. Danach wurden sie im Code geprüft und manuell überarbeitet.
`GPT-5.3 Codex`	`10.04.2026`	«Compare the code requirements of this file to the project task manager and give me a feedback where improvements are needed to achieve the minimal requirements. Do not edit any code, only give me a feedback.»	Das Qualitätsfeedback zu den Coderichtlinien wurde genutzt. Die nötigen Nacharbeiten wurden umgesetzt.
`GPT-5.4 ChatGPT`	`12.04.2026`	«Use the following Documents and anlyze the which points are achieved and which ones need to be improved, give me a list	Der Kriterienabgleich zeigte offene Punkte in der Dokumentation.

Modell	Datum	Prompt-Ziel	Einsatz und Nachweis
		back with all criterias and give me bullet points what is good, what needs to be improved and what is still needed for achieving all requirements.»	Diese Punkte wurden gezielt ergänzt.
GPT-5.4 ChatGPT	13.04.2026	«Analysiere das folgende Dokument und gebe mir eine Liste zurück mit allen sätzen und wörtern welche grammatikalisch nicht korrekt sind, gebe mir eine korrigierte version, schätze ein welche beschreibungen weniger lesefreundlich sind, welche besser formuliert werden können und gebe an welche massnahme diese verbessern würde.»	Die sprachliche Überprüfung half bei Grammatik und Stil. Die Korrekturen wurden manuell geprüft und übernommen.
GPT-5.4 ChatGPT	12.04.2026	«Analyze the following documents and let me know which sections are explained in too much detail and which ones would still earn full marks even with less detail. Break them down into safe to remove, recommended to remove, critical but can be removed.»	Die Hinweise zur Verdichtung wurden genutzt. Dadurch wurde der Text kürzer und gut lesbar.

2.7 Auswerten

Zum Abschluss werden Zielerreichung, Ergebnisqualität und Grenzen der Lösung bewertet.

2.7.1 Kriterien und Nachweise

Den Bewertungskriterien werden die zugehörigen Nachweise aus Umsetzung, Test und Dokumentation gegenübergestellt.

ID	Beurteilungspunkt	Nachweis in der Dokumentation
A01	Auftrag analysiert; Anforderungen und Methode IPERKA dokumentiert.	Abschnitt 1.1.2, Abschnitt 2.2.1, Abschnitt 1.2
A02	Relevante Informationen gezielt recherchiert und im Projektkontext verwendet.	Abschnitt 2.2.2, Abschnitt 2.5.2, Abschnitt 2.6.6
A03	Informationen klar aufbereitet, visualisiert und im Auftragskontext genutzt.	Abschnitt 2.2.1, Abschnitt 2.2.2, Abschnitt 2.3.2, Abschnitt 2.3.3, Abschnitt 2.3.4
A04	Zeitplan mit IPERKA-Struktur, Datum, Granularität und Soll/Ist-Vergleich.	Abschnitt 1.6, Abschnitt 1.6.1, Abschnitt 1.6.2
A05	Projektfortschritt, Abweichungen und Korrekturmassnahmen nachvollziehbar dokumentiert.	Abschnitt 1.6, Abschnitt 1.7, Abschnitt 2.6.5, Abschnitt 2.3.8, Abschnitt 2.6.4
A06	Projektziele priorisiert, fachlich verfolgt und Ergebnisqualität bewertet.	Abschnitt 2.4.1, Abschnitt 2.4.4, Abschnitt 2.6.5, Abschnitt 2.7.2
A07	Eigenständige Bearbeitung, Problemlösung und Reflexion nachgewiesen.	Abschnitt 1.7, Abschnitt 2.6.4, Abschnitt 2.7.3, Abschnitt 1.9
A08	Fachvokabular konsistent und für externe Fachpersonen verständlich eingesetzt.	Abschnitt 2.2.2, Abschnitt 2.5.1, Abschnitt 2.5.4
A09	Fachkompetenz durch Analyse, Variantenentscheid und Umsetzung angewendet.	Abschnitt 2.4.2, Abschnitt 2.4.3, Abschnitt 2.5.4
A10	Rückmeldungen und Vorgaben aufgenommen, umgesetzt und dokumentiert.	Abschnitt 1.2, Abschnitt 2.2.3, Abschnitt 2.3.6, Abschnitt 2.4.4, Abschnitt 2.3.8, Abschnitt 1.7
A11	Projektrollen beschrieben und Aufbauorganisation grafisch dargestellt.	Abschnitt 1.2
A12	Testumgebung, Testszenarien, Durchführung und Nacharbeit dokumentiert.	Abschnitt 2.3.7, Abschnitt 2.3.7.2, Abschnitt 2.6.1, Abschnitt 2.6.2, Abschnitt 2.6.3, Abschnitt 2.6.4
C11	KI-Modelle zweckmässig eingesetzt, mit präzisen Eingaben gesteuert und Ergebnisse objektiv geprüft.	Abschnitt 2.6.6
G03	Mockup erstellt; Machbarkeit, Benutzerfreundlichkeit und Feedback dokumentiert.	Abschnitt 2.3.6, Abschnitt 2.5.6, Abschnitt 4.2
G08	Umsetzungsvarianten skizziert, bewertet und Auswahl begründet.	Abschnitt 2.4.1, Abschnitt 2.4.2, Abschnitt 2.4.3, Abschnitt 2.4.4
G09	Realisierungskonzept mit Use Cases, Abläufen und Schnittstellen ausgearbeitet.	Abschnitt 2.2.3, Abschnitt 2.3.2, Abschnitt 2.3.3, Abschnitt 2.3.1, Abschnitt 2.3.4, Abschnitt 2.3.8

ID	Beurteilungspunkt	Nachweis in der Dokumentation
I01	Code modular, nachvollziehbar und an Vorgaben ausgerichtet.	Anhang 1
I02	ROS2-Nodes, Topics, Services und Nachrichtenflüsse dokumentiert.	Abschnitt 2.3.1, Abschnitt 2.3.4, Abschnitt 2.5.1
I03	Validierungsstrategie implementiert, mit Datenbankdaten abgeglichen und getestet.	Abschnitt 2.3.5, Abschnitt 2.5.4, Abschnitt 2.6.2, Abschnitt 2.6.3
I04	ROS2-Kommunikation über Topics/ Services und begründete QoS-Profile umgesetzt.	Abschnitt 2.3.4, Abschnitt 2.5.1, Abschnitt 2.5.3
I05	E2E-Datenfluss vom Outlook-Termin bis ins `RViz-Panel` geprüft.	Abschnitt 2.3.7.2, Abschnitt 2.6.3
I06	Manuelle Stornierung über UI mit Kalender- und E-Mail-Rückmeldung umgesetzt.	Abschnitt 2.5.6, Abschnitt 2.5.7, Abschnitt 4.3, Abschnitt 4.4, Abschnitt 2.6.3
Doc01	Bericht klar gegliedert; IPERKA-Struktur in Teil 2 sichtbar.	Chapter 2, Chapter 1, Abschnitt 2.2, Abschnitt 2.3, Abschnitt 2.4, Abschnitt 2.5, Abschnitt 2.6, Abschnitt 2.7
Doc02	Einheitliche Gestaltung, klare Überschriften und lesbare Abstände verwendet.	Chapter 2, Abschnitt 2.2.1, Abschnitt 2.3.1, Chapter 3, Chapter 4
Doc03	Formale Berichtselemente über Vorlage und Dokumentstruktur abgedeckt.	Chapter 2, Abschnitt 2.1, Chapter 3, Chapter 4
Doc04	Text klar, verständlich und fachlich konsistent formuliert.	Chapter 2, Abschnitt 2.1, Abschnitt 2.7.2, Abschnitt 2.7.3
Doc05	Abbildungen, Diagramme und Tabellen mit Beschriftungen und lesbarer Grösse.	Abschnitt 2.2.1, Abschnitt 2.3.2, Abschnitt 2.3.3, Abschnitt 2.3.1, Abschnitt 2.3.4, Abschnitt 2.3.6
Doc06	Kurzfassung mit Ausgangslage, Vorgehen und Ergebnis.	Abschnitt 2.1, Abschnitt 2.1.1, Abschnitt 2.1.2, Abschnitt 2.1.3
Doc07	Arbeitsjournal als Teil 1 mit Tagesberichten und Unterstützungen.	Abschnitt 1.7, Abschnitt 1.6
Doc08	Persönliches Fazit und Reflexion zu Erfolg, Lernen und Verbesserungen.	Abschnitt 2.7.3, Abschnitt 1.9

2.7.2 Analyse und Beurteilung der Ergebnisse

Durch die Ergebnisse wird gezeigt, dass die umgesetzte Lösung die wesentlichen Projektziele im vorgesehenen Umfang erreicht. Der gesamte Ablauf vom Kalendereintrag über das Parsing und die Validierung bis zur Anzeige im RViz-Panel und zur Übergabe an das Navigations-Mock konnte erfolgreich umgesetzt und getestet werden. Die gewählte modulare Architektur konnte im Projekt als sinnvoll bewertet werden. Da die Aufgaben klar aufgeteilt sind, lässt sich das System einfacher warten und bei Fehlern besser analysieren. Auch der Schlüssel-Wert-Ansatz wurde als passend bewertet, weil eine verständliche und zuverlässige Verarbeitung der Eingaben ermöglicht wird.

Die Testergebnisse aus Abschnitt 2.6.3 sind insgesamt positiv zu bewerten. Die zentrale Funktionalität konnte nachgewiesen werden, und eine erkannte Abweichung wurde während der Kontrolle korrigiert. Damit kann die Lösung als fachlich gelungen und im IPA-Rahmen als zweckmässig beurteilt werden. Als Grenze der Arbeit ist festzuhalten, dass die Ausführung nur mit dem Navigations-Mock und nicht mit dem physischen Roboter geprüft wurde. Die leicht erhöhte Länge der Dokumentation wurde mit dem Hauptexperten Daniel Ramser abgesprochen und als in Ordnung bestätigt. Insgesamt wurde das Projektziel jedoch im definierten Umfang erreicht.

2.7.3 Erkenntnisse und Reflexion

Dieses Projekt hat mir sehr gefallen, besonders im Rahmen der IPA. Da ich bereits Vorerfahrungen mit den verschiedenen Technologien hatte, konnte ich mir von Anfang an ein gutes Bild davon machen, was ich wie umsetzen möchte. Das hat mir sehr geholfen. Auch das Vorgehen, die Programmierung Node für Node umzusetzen, hat für mich sehr gut funktioniert, da ich dadurch strukturiert arbeiten sowie einzelne Teile gezielt testen und verbessern konnte. Indem ich Dokumentationen, zum Beispiel zur Graph API, nochmals gelesen habe, konnte ich neue Möglichkeiten im SDK entdecken, die für meinen Anwendungsfall sehr hilfreich waren. Ich wusste vorher nicht, dass ich angeben kann, wie meine Daten sortiert werden sollen oder in welcher Zeitzone ich sie erhalten möchte. Dadurch musste ich weniger eigene Logik implementieren, was die Robustheit meiner Lösung verbessert hat. Zusätzlich habe ich in diesem Projekt den Einsatz von Sets in Python kennengelernt, was ich vorher so noch nicht kannte und was mir sehr geholfen hat. Obwohl ich in der Dokumentation gegen Ende vieles kürzen musste, um näher an die maximale Seitenzahl zu kommen, konnte ich sie dadurch lesefreundlicher machen, auch wenn vielleicht nicht mehr alle Details enthalten sind.

Besonders Schwierigkeiten hatte ich beim `RunTaskNode`, da ich nicht wusste, wie ich den Navigations-Mock-Status in den Auftragsstatus injizieren soll, sodass der Zustand «failed» auch einen konkreten Nutzen hat. Auch beim RViz-Qt-Plugin hatte ich anfangs grosse Schwierigkeiten, da ich mich mit Qt und C++ nicht besonders gut auskannte. Zwar wurde es einfacher, sobald ich mich eingearbeitet hatte, trotzdem fühle ich mich in diesem Bereich noch unsicher. Es gab Stellen, die ich zunächst übernommen habe, ohne sie vollständig erklären zu können, zum Beispiel `explicit TaskPanel(QWidget* parent = 0);`. Beim Dokumentieren habe ich jedoch gelernt, was manche Punkte bedeuten, und konnte mein Verständnis stärken. Bei der Dokumentation habe ich ausserdem gemerkt, wie wichtig es ist, sie immer aktuell zu halten. Da ich Änderungen nicht immer direkt nachgeführt habe, entstand am Ende viel zusätzlicher Aufwand und Druck. Zudem war es schwierig, in der Dokumentation die richtige Balance zwischen Verständlichkeit und Kürze zu finden.

In Zukunft würde ich die Dokumentation jeweils direkt aktualisieren, sobald ich eine Änderung an der Umsetzung vornehme. Das nächste Mal würde ich im Rahmen einer IPA ausserdem mehr Zeit für die Dokumentation einplanen und wenn möglich ein etwas kleineres Projekt wählen. Insgesamt beurteile ich das Projekt als gelungen, auch wenn der Umfang gegen Ende etwas zu gross war und die Dokumentation mehr Aufwand verursachte als erwartet.

3 Installationsanleitung

3.1 Einführung

Diese Installationsanleitung beschreibt die Einrichtung des ROS2-Packages `fhnw_bot_task_manager` in einem FHNW-Roboter-Workspace. Sie richtet sich an Entwicklerinnen, Entwickler und Betreiber, die den Task Manager lokal installieren, konfigurieren und zusammen mit Navigation, Datenbank, Microsoft Graph und RViz-Panel starten.

Installiert und konfiguriert werden das Python-Package `fhnw_bot_task_manager`, die benötigten Custom Interfaces, der Zugriff auf `fhnw_bot_db`, die Microsoft-Graph-Zugangsdaten und optional das RViz-Panel `fhnw_rviz_task_panel`. Der reguläre Start erfolgt über die konfigurierte Launch-Datei `fhnw_bot_task_manager/launch/task_manager.launch.py`.

3.2 Voraussetzungen

Bereich	Voraussetzung
Zielumgebung	Linux mit ROS2 Jazzy.
Workspace	Das Repository muss in einem ROS2-Workspace liegen.
Build-Werkzeuge	`colcon` muss verfügbar sein.
ROS2-Pakete	`rclpy`, `geometry_msgs`, `nav2_msgs`, `rviz_common`, `pluginlib` und die ROS2-Standardwerkzeuge müssen installiert sein.
Workspace-Pakete	`fhnw_custom_interfaces`, `fhnw_bot_db`, `fhnw_bot_task_manager` und für die RViz-Bedienung `fhnw_rviz_task_panel`.
Python-Abhängigkeiten	`python-dotenv`, `azure-identity` und `msgraph-sdk`.
Microsoft Graph	Eine Azure-App mit Zugriff auf den Kalender des konfigurierten Benutzers, Terminaktualisierung und Mailversand. Benötigt werden `TENANT_ID`, `CLIENT_ID`, `CLIENT_SECRET` und `USER_ID`.
Datenbank	`fhnw_bot_db` muss Personen und Room-Entrance-Daten liefern. Für jede Zielperson muss ein Raum-Eingang mit Koordinate und Quaternion mit vier Werten vorhanden sein.
Navigation	Für den Realen betrieb muss ein Nav2-Action-Server für `NavigateToPose` laufen. Für Tests kann der im Package enthaltene `robot_mock` verwendet werden.

3.3 Installation

1. ROS2-Umgebung laden:

   ```
   source /opt/ros/jazzy/setup.bash
   ```

   Danach ist das ROS2-CLI verfügbar.

2. In den Workspace-Root wechseln:

   ```
   cd /fhnw-robot-packages
   ```

   Alle folgenden Befehle werden aus dem Workspace-Root ausgeführt.

3. Python-Abhängigkeiten installieren, falls sie noch nicht vorhanden sind:

   ```
   python3 -m pip install python-dotenv azure-identity msgraph-sdk
   ```

   Diese Pakete werden vom Microsoft-Graph-Client verwendet.

4. Benötigte Workspace-Pakete bauen:

   ```
   colcon build --symlink-install --packages-select fhnw_custom_interfaces fhnw_bot_db
   fhnw_bot_task_manager fhnw_rviz_task_panel
   ```

   Dadurch werden die Custom Messages, die Datenbank-API, der Task Manager und das RViz-Panel gebaut.

5. Workspace-Umgebung laden:

   ```
   source install/setup.bash
   ```

   Danach sind die Executables `graph_client_node`, `validation_node`, `run_task_node` und `robot_mock` verfügbar. Die Launch-Datei liegt im Source-Workspace unter `fhnw_bot_task_manager/launch/task_manager.launch.py`.

6. Installation prüfen:

   ```
   ros2 pkg executables fhnw_bot_task_manager
   ```

   Erwartete Executables:

   ```
   fhnw_bot_task_manager graph_client_node
   fhnw_bot_task_manager robot_mock
   fhnw_bot_task_manager run_task_node
   fhnw_bot_task_manager validation_node
   ```

7. Launch-Datei prüfen:

   ```
   ros2 launch fhnw_bot_task_manager/launch/task_manager.launch.py --show-args
   ```

   Erwartet wird `No arguments.`. Die Launch-Datei startet `graph_client_node`, `validation_node` und `run_task_node`.

3.4 Konfiguration

Die Microsoft-Graph-Konfiguration wird aus Umgebungsvariablen gelesen. Eine `.env`-Datei im Workspace-Root ist für den lokalen Betrieb zweckmässig:

```
TENANT_ID=<azure-tenant-id>
CLIENT_ID=<azure-app-client-id>
CLIENT_SECRET=<azure-app-client-secret>
USER_ID=<calendar-user-id-or-email>
TIMER_INTERVAL=1.0
```

`TENANT_ID`, `CLIENT_ID`, `CLIENT_SECRET` und `USER_ID` sind zwingend erforderlich. `TIMER_INTERVAL` steuert das Polling-Intervall des `graph_client_node`; wenn der Wert fehlt oder ungültig ist, verwendet der Node seinen ROS2-Parameter `default_timer_interval_sec`. Die Default-Zeitzone ist `Europe/Zurich`.

3.5 Inbetriebnahme und Fehlerbehebung

Startreihenfolge für einen lokalen Test mit Mock-Navigation:

1. In jedem Terminal ROS2 und den Workspace laden:

   ```
   source /opt/ros/jazzy/setup.bash
   source /install/setup.bash
   cd /fhnw-robot-packages
   ```

2. Mock-Navigation starten:

   ```
   ros2 run fhnw_bot_task_manager robot_mock --ros-args -p travel_time:=7.0
   ```

   Der Mock stellt den Action-Server `navigate_to_pose` bereit.

3. Task Manager über die Launch-Datei starten:

   ```
   ros2 launch fhnw_bot_task_manager task_manager.launch.py
   ```

   Die Launch-Datei startet `graph_client_node`, `validation_node` und `run_task_node`. Dadurch werden Kalenderereignisse gepollt, unbestätigte Tasks validiert, akzeptierte Tasks ausgeführt und finalisierte Task-Listen publiziert.

4. RViz optional starten und das Panel `fhnw_rviz_task_panel::TaskPanel` hinzufügen:

   ```
   rviz2
   ```

   Das Panel liest die finalisierte Task-Liste und kann Tasks über den Change-Task-Service stornieren.

Für gezieltes Debugging können die Nodes weiterhin einzeln gestartet werden:

```
ros2 run fhnw_bot_task_manager graph_client_node
ros2 run fhnw_bot_task_manager validation_node
ros2 run fhnw_bot_task_manager run_task_node
```

Häufige Fehler:

Fehlerbild	Prüfung oder Massnahme
`Package not found`	ROS2 oder der Workspace wurde nicht gesourced. `source /opt/ros/jazzy/setup.bash` und danach `source install/setup.bash` ausführen.
`Missing env vars. Declare TENANT_ID, CLIENT_ID, CLIENT_SECRET, USER_ID in your .env`	`.env` prüfen oder die Variablen in der Shell setzen.
`Error fetching events from Graph` oder **Sync-Status `Not Synced`**	Graph-Zugangsdaten, Azure-Berechtigungen, Netzwerkzugriff und `USER_ID` prüfen.
`change_task service not available`	Task-Manager-Launch starten oder warten, bis der `graph_client_node` den Service `/fhnw_bot_task_manager/change_task` bereitstellt.
`navigate_to_pose action not available`	Nav2 oder `robot_mock` starten. Falls der Action-Name abweicht, `run_task_node` mit `-p navigate_to_pose_action_name:=<name>` starten.
`ModuleNotFoundError: No module named 'azure'`	Bei `ModuleNotFoundError: No module named "azure"` verwenden Sie folgenden Befehl: `export PYTHONPATH="$HOME/fhnw-robot-packages/.venv/lib/python3.12/site-packages:$PYTHONPATH"`
Zielperson wird nicht gefunden oder hat kein Ziel	Eintrag in `fhnw_bot_db` prüfen. Die Zielperson muss mit dem geparsten Namen gefunden werden und ein Room Entrance mit vier Quaternion-Werten besitzen.

4 Bedienungsanleitung

4.1 Einführung

Diese Bedienungsanleitung richtet sich an Benutzende des RViz-Task-Panels und an Personen, die Aufträge über Kalendertermine erfassen. Der Hauptanwendungsfall ist, heutige Kalendertermine als Roboteraufträge zu übernehmen, zu validieren, zur geplanten Startzeit auszuführen und den Status im RViz-Panel zu verfolgen.

4.2 Benutzeroberfläche und Funktionen

Das RViz-Panel wird nach der Installation aus Abschnitt 3.3 in RViz über `Panels` -> `Add New Panel` hinzugefügt. Die Plugin-Klasse heisst `fhnw_rviz_task_panel::TaskPanel`.

Relevante Bereiche des Panels:

Bereich	Funktion
`Today's Tasks`	Überschrift der Aufgabenliste für den aktuellen Tag.
Sync-Status	Zeigt `Synced` oder `Not Synced`. `Synced` bedeutet, dass die letzte Task-Liste erfolgreich erzeugt wurde. `Not Synced` weist auf ein Problem beim Abruf oder der Verarbeitung hin.
Sync Message	Zeigt die letzte Systemmeldung, zum Beispiel `Successful`, `No events found for today` oder eine Fehlermeldung.
Zeitstempel	Zeigt, wann die angezeigte Task-Liste erzeugt wurde.
Task-Karten	Pro Auftrag werden verantwortliche Person, Zeitfenster, Titel, Status und Auftragstyp angezeigt.
Cancel-Button	Sendet für den Auftrag eine Stornierung an den Task Manager. Der Button ist für `pending` und `active` aktiv und für abgeschlossene oder fehlgeschlagene Aufträge deaktiviert.

4.3 Bedienung

1. System starten.

   Navigation oder Mock-Navigation muss laufen. Danach wird der Task Manager aus dem Workspace-Root mit der Launch-Datei gestartet:

   ```
   ros2 launch fhnw_bot_task_manager task_manager.launch.py
   ```

   Die Launch-Datei startet `graph_client_node`, `validation_node` und `run_task_node`.

2. Kalendertermin erfassen.

   Der Task Manager liest die heutigen Termine des konfigurierten Graph-Benutzers. Der Termin braucht Titel, Organisator, Start- und Endzeit sowie eine strukturierte Beschreibung.

   Beispiel für eine Übermittlung an eine Zielperson:

   ```
   Auftrag übermittlung zielperson Andri Wild nachricht Hallo Welt
   ```

   Beispiel für einen Transport mit zwei Zielpersonen:

   ```
   Auftrag transport zielperson1 Andri Wild zielperson2 Keanu Koelewijn nachricht Bitte den Gegenstand weitergeben
   ```

   `übermittlung` und `forward` werden als Übermittlungsauftrag erkannt. `transport` wird als Transportauftrag erkannt. Zielpersonen werden nach `zielperson`, `zielperson1` oder `zielperson2` gelesen. Der Parser ist auf ein- bis zweiteilige Personennamen ausgelegt. Die Namen müssen zur Datenbank passen.

3. Validierung abwarten.

   Ein neuer oder nicht akzeptierter Termin wird automatisch validiert. Bei Erfolg nimmt der Task Manager den Termin an und sendet die Meldung `Der Auftrag wurde erfolgreich geprüft und kann übernommen werden`. Bei Fehlern lehnt er den Termin mit einer begründeten Nachricht ab.

4. Auftrag im RViz-Panel beobachten.

   Akzeptierte Tasks erscheinen als Karten. Der Status kann `pending`, `active`, `completed` oder `failed` sein. `pending` bedeutet geplant, `active` bedeutet in Ausführung, `completed` bedeutet abgeschlossen, `failed` bedeutet fehlgeschlagen.

5. Automatische Ausführung zur Startzeit.

   Sobald die Startzeit eines validierten Tasks erreicht ist, startet `run_task_node` die Navigation zu den hinterlegten Zielpunkten. Bei mehreren Zielpersonen werden die Ziele nacheinander angefahren.

6. Auftrag stornieren.

   Den `Cancel`-Button auf der Task-Karte klicken. Das Panel sendet eine Stornierung an `/fhnw_bot_task_manager/change_task`. Der Task Manager sendet eine Mail an die organisierende Person und löscht den Kalendertermin.

7. Ergebnis prüfen.

   Nach erfolgreicher Navigation wird der Auftrag als `completed` markiert und eine Abschlussmail gesendet. Bei abgelehntem, abgebrochenem oder gestopptem Navigationsziel wird der Auftrag als `failed` markiert und eine Fehlermail gesendet.

4.4 Meldungen und Hinweise

Wichtige Statusmeldungen:

Meldung	Bedeutung oder Massnahme
`Synced`	Die letzte Task-Liste wurde erfolgreich publiziert.
`Not Synced`	Die letzte Task-Liste konnte nicht korrekt synchronisiert werden. Sync Message und Logs des `graph_client_node` prüfen.
`Successful`	Die Task-Liste wurde erfolgreich erzeugt oder aktualisiert.
`No events found for today`	Für den aktuellen Tag wurden keine Kalendereereignisse gefunden.
`Error fetching events from Graph`	Kalenderdaten konnten nicht aus Microsoft Graph geladen werden. Zugangsdaten, Berechtigungen und Netzwerk prüfen.

Wichtige Validierungsmeldungen:

Meldung	Bedeutung oder Massnahme
`Der Auftrag wurde erfolgreich geprüft und kann übernommen werden.`	Der Auftrag ist gültig und kann ausgeführt werden.
`Der Auftrag konnte nicht übernommen werden. Gründe:`	Mindestens eine Validierungsregel wurde verletzt.
`Zeitüberschneidung mit einem anderen Auftrag.`	Das Zeitfenster überlappt mit einem anderen Auftrag.
`Die Dauer muss zwischen 5 und 30 Minuten liegen.`	Start- und Endzeit des Termins anpassen.
`Kein Verantwortlicher wurde erkannt.` oder **`Kein Titel wurde angegeben.`**	Kalendertermin vervollständigen.
`Keine Beschreibung vorhanden.` oder **`Die Beschreibung konnte nicht vollständig verarbeitet werden.`**	Strukturierte Auftragsbeschreibung ergänzen.
`Ungültiger Auftragstyp. Erlaubt sind "Transport" oder "Übermittlung".`	Auftragstyp in der Beschreibung korrigieren.
`Die Nachricht ist zu kurz oder fehlt.`	Nachricht nach `nachricht` ergänzen.
`Die Anzahl der Zielpersonen passt nicht zum Auftragstyp.`	Für Übermittlung eine Zielperson und für Transport zwei Zielpersonen angeben.
`Eine oder mehrere Zielpersonen wurden im System nicht gefunden.`	Namen so erfassen, wie sie in `fhnw_bot_db` vorhanden sind.
`Für eine oder mehrere Zielpersonen ist kein Raum hinterlegt.`	Datenbankeintrag für Raum oder Eingang ergänzen.

Wichtige Laufzeitmeldungen:

Meldung	Bedeutung oder Massnahme
`Der Auftrag wurde erfolgreich abgeschlossen.`	Alle Ziele wurden erreicht.
`Der Auftrag konnte nicht gestartet werden, da das Navigationsziel abgelehnt wurde.`	Nav2 hat das Ziel nicht akzeptiert. Zielkoordinaten und Navigationsstatus prüfen.
`Der Auftrag konnte nicht abgeschlossen werden, da die Navigation abgebrochen oder gestoppt wurde.`	Navigation wurde abgebrochen oder gestoppt. Roboter- und Nav2-Logs prüfen.
`Ihr Auftrag wurde Storniert.`	Der Auftrag wurde über das RViz-Panel storniert.

Abbildungsverzeichnis

Glossar

Action : Asynchrones ROS2-Muster für längere Operationen mit Ziel, Zwischenfeedback und Ergebnis.
Action Client : Client-Seite einer Action: sendet Ziele und verarbeitet Feedback sowie Abschlussresultate.
Action Server : Server-Seite einer Action: nimmt Ziele an, führt sie aus und liefert Feedback sowie Resultat zurück.
Change-Task-Service : Projektinterner ROS2-Service zur Statusänderung oder Stornierung eines Auftrags.
Launch-Datei : ROS2-Startdatei (meist Python), die mehrere Nodes und Parameter gemeinsam startet.
local_timezone : Konfigurationsparameter zur lokalen Zeitzonenbewertung bei zeitabhängiger Statusberechnung.
Nav2 : Navigation2-Stack von ROS2 für Zielnavigation, Pfadplanung und Bewegungssteuerung.
navigate_to_pose : Name der Navigations-Action für das Senden von Zielpositionen an die Navigationskomponente.
navigation_frame_id : ROS2-Parameter für den Ziel-Koordinatenrahmen, zum Beispiel `map`.
Navigations-Mock : Simuliertes Navigationssystem als Ersatz für die Ausführung auf physischer Roboterhardware.
Publish-Subscribe : Kommunikationsprinzip, bei dem Publisher entkoppelt an Topics senden und Subscriber diese abonnieren.
Publisher : Komponente, die Daten auf ein Topic publiziert.
QoS : Quality of Service: Übertragungsregeln wie Zuverlässigkeit, Historie, Haltbarkeit und Tiefe eines Nachrichtenpuffers.
QoS-Profil : Konkrete Kombination von QoS-Einstellungen für einen bestimmten Kommunikationspfad.
Quaternion : Darstellung einer 3D-Orientierung über vier Werte (`x`, `y`, `z`, `w`).
robot_mock : Im Projekt genutzte Mock-Komponente, die den Navigations-Action-Server des Roboters simuliert.
ROS : Robot Operating System: Framework für Robotik-Anwendungen mit standardisierten Kommunikationsmustern zwischen Komponenten.
ROS-Kommunikation : Gesamtheit der Kommunikationsbeziehungen zwischen ROS2-Nodes und ihren Schnittstellen.
ROS-Nachricht : Typsichere Datenstruktur, die in ROS2 über Topics, Services oder Actions übertragen wird.
ROS-Node : Eigenständiger Prozess im ROS2-System, der über Topics, Services oder Actions mit anderen Nodes kommuniziert.
ROS2 : Aktuelle Generation von ROS mit DDS-basierter Kommunikation und erweiterten QoS-Mechanismen.
RViz : Visualisierungstool im ROS2-Ökosystem zur Anzeige und Beobachtung von Robotikdaten.
RViz-Panel : Erweiterbares Bedienpanel in RViz; in dieser Arbeit zur Anzeige und Stornierung von Tasks verwendet.
Service : Request-Response-Kommunikationsmuster in ROS2 für direkte Anfrage-Antwort-Aufrufe.
sourcen / Workspace sourcen : Laden der ROS2-Umgebung in einem Terminal mittels `source .../setup.zsh`, damit Pakete und Befehle verfügbar sind.
Subscriber : Komponente, die Daten von einem Topic empfängt und verarbeitet.

Sync-Status : Anzeigezustand im Panel, ob die letzte Synchronisation der Task-Daten erfolgreich war.
Task-Liste : Im Projekt publizierte Auftragsliste mit aktuellem Bearbeitungs- und Synchronisationsstand.
TaskPanel : Konkrete Panel-Klasse des RViz-Plugins `fhnw_rviz_task_panel::TaskPanel`.
Topic : Benannter Nachrichtenkanal für asynchrone Datenübertragung im Publish-Subscribe-Modell.
Workspace : ROS2-Arbeitsverzeichnis mit Paketquellen und Build-Artefakten, typischerweise `src`, `build`, `install` und `log`.

Quellenverzeichnis

[1] Microsoft, «Overview of Microsoft Graph». Zugegriffen: 30. März 2026. [Online]. Verfügbar unter: https://learn.microsoft.com/en-us/graph/overview

[2] Microsoft, «Working with calendars and events using the Microsoft Graph API». Zugegriffen: 30. März 2026. [Online]. Verfügbar unter: https://learn.microsoft.com/en-us/graph/api/resources/calendar-overview?view=graph-rest-1.0

[3] Open Robotics, «Topics vs Services vs Actions». Zugegriffen: 30. März 2026. [Online]. Verfügbar unter: https://docs.ros.org/en/jazzy/How-To-Guides/Topics-Services-Actions.html

[4] Open Robotics, «Quality of Service settings». Zugegriffen: 30. März 2026. [Online]. Verfügbar unter: https://docs.ros.org/en/jazzy/Concepts/Intermediate/About-Quality-of-Service-Settings.html

[5] Open Robotics, «RViz User Guide». Zugegriffen: 30. März 2026. [Online]. Verfügbar unter: https://docs.ros.org/en/jazzy/Tutorials/Intermediate/RViz/RViz-User-Guide/RViz-User-Guide.html

[6] BA Tools, «Schnittstellenanalyse». Zugegriffen: 30. März 2026. [Online]. Verfügbar unter: https://ba-tools.de/schnittstellenanalyse/

[7] BWL-Lexikon, «Präferenzmatrix». Zugegriffen: 1. April 2026. [Online]. Verfügbar unter: https://www.bwl-lexikon.de/wiki/praeferenzmatrix/

[8] Microsoft, «List calendarView». Zugegriffen: 2. April 2026. [Online]. Verfügbar unter: https://learn.microsoft.com/en-us/graph/api/calendar-list-calendarview?view=graph-rest-1.0&tabs=python#example

[9] Microsoft, «Choose a Microsoft Graph authentication provider». Zugegriffen: 2. April 2026. [Online]. Verfügbar unter: https://learn.microsoft.com/en-us/graph/sdks/choose-authentication-providers?tabs=python#using-a-client-secret-4

[10] Python Software Foundation, «dataclasses: Data Classes». Zugegriffen: 2. April 2026. [Online]. Verfügbar unter: https://docs.python.org/3/library/dataclasses.html

[11] Open Robotics, «Executors». Zugegriffen: 9. April 2026. [Online]. Verfügbar unter: https://docs.ros.org/en/jazzy/Concepts/Intermediate/About-Executors.html

[12] Open Robotics, «Building a Custom RViz Panel». Zugegriffen: 7. April 2026. [Online]. Verfügbar unter: https://docs.ros.org/en/jazzy/Tutorials/Intermediate/RViz/RViz-Custom-Panel/RViz-Custom-Panel.html

[13] RyodoTanaka, «dial_panel.cpp». Zugegriffen: 7. April 2026. [Online]. Verfügbar unter: https://github.com/RyodoTanaka/rviz_plugin_examples/blob/foxy-devel/src/dial_panel.cpp

[14] Microsoft, «Accept event». Zugegriffen: 9. April 2026. [Online]. Verfügbar unter: https://learn.microsoft.com/en-us/graph/api/event-accept?view=graph-rest-1.0&tabs=python

[15] Microsoft, «Decline event». Zugegriffen: 9. April 2026. [Online]. Verfügbar unter: https://learn.microsoft.com/en-us/graph/api/event-decline?view=graph-rest-1.0&tabs=python

[16] Microsoft, «Delete event». Zugegriffen: 9. April 2026. [Online]. Verfügbar unter: https://learn.microsoft.com/en-us/graph/api/event-delete?view=graph-rest-1.0&tabs=python#example

[17] Microsoft, «Update event». Zugegriffen: 10. April 2026. [Online]. Verfügbar unter: https://learn.microsoft.com/en-us/graph/api/event-update?view=graph-rest-1.0&tabs=python

[18] Microsoft, «user: sendMail». Zugegriffen: 10. April 2026. [Online]. Verfügbar unter: https://learn.microsoft.com/en-us/graph/api/user-sendmail?view=graph-rest-1.0

Anhang 1: Endprodukt

Anhang 2: Zusatz-Information