Standardisierung von Agenten zur Leistungserfassung in Klinikinformationssystemen

MAS Software Engineering 2023 - 2025
Projektteam: Jvan Fadda, Guillaume Fricker, Benjamin Thormann
Hauptbetreuung: Tobias Büchel
Betreuung: Manuel Bauer, Tobias Büchel, Martina Lux

Inhaltsverzeichnis

Glossar 6

1 Einleitung 7

1.1 Problemstellung 7

1.1.1 Technische Herausforderungen 7

1.1.2 Probleme im operativen Betrieb 8

1.1.3 Organisatorische Defizite 8

1.2 Zielsetzung 9

2 Grundlagen 9

2.1 Klinisches Informationssystem 9

2.2 Leistung 10

2.2.1 TARMED 10

2.2.2 TARDOC 10

2.2.3 Tarife im KIS-Kontext 11

2.3 Agent im klinischen Kontext 11

2.3.1 Zur Relevanz von Agenten im schweizerischen Gesundheitswesen 11

2.4 Domänenspezifische und Systemabgrenzung 12

3 Methodik 13

3.1 Projektorganisation und Vorgehen 13

3.1.1 Aufwandsschätzung 15

3.1.2 Entwicklungsprozess 15

3.2 Risikoanalyse 16

3.3 Stakeholder-Analyse 16

3.3.1 Rollen 17

3.3.2 Stakeholder 18

3.3.3 Stakeholder-Matrix 18

3.4 Architekturentscheidungen 19

3.5 Testing Strategie 19

3.6 DevOps-Strategie und Qualitätssicherung 20

3.6.1 Entwicklungsphase 20

3.6.2 CI/CD Pipeline 21

3.6.3 Betrieb 21
3.6.4 Metriken 21
3.6.5 Systemverteilung und Deployment 21
4 Problemanalyse 22
4.1 Ist-Zustand / Ausgangslage 22
4.1.1 Strukturelle Eigenschaften 22
4.1.2 Persistenzschicht 22
4.1.3 Deployment, Monitoring und Betrieb 23
4.1.4 Kommunikation 23
4.2 Use-Cases und Anforderungen 23
4.2.1 Quellen für Stakeholder- und Anforderungsanalyse 23
4.2.2 Akteure 23
4.2.3 Use-Cases 24
4.3 Anforderungskriterien 26
4.4 Randbedingungen 29
4.5 Domain Model 29
4.6 Risikoanalyse 30
4.6.1 Initial definierte Risiken 31
4.6.2 Technische Risiken 31
4.6.3 Organisatorische Risiken 32
4.6.4 Finale Risikoanalyse 34
4.7 Architektur 35
4.7.1 Auswertung möglicher Technologiestacks 35
4.7.2 Walking Skeleton 36
4.7.3 Architekturentscheide / Architecture Decision Records (ADR's) 37
4.8 Systemverteilung und Deployment 38
5 Ergebnisse 38
5.1 Projektorganisation 39
5.1.1 Zusammenarbeit mit der CISTEC AG 39
5.1.2 Zusammenarbeit im Projektteam 39
5.1.3 Arbeitsaufwände und Eigenverantwortung 39
5.1.4 Risikoanalyse 40
5.1.5 Domäne und Abnahmekriterien (ADR0007) 40

5.1.6 Domänenspezifischer vs. generischer Framework-Prototyp 40
5.1.7 Technischer Grossausfall 41
5.1.8 Auswertung Risikomatrix 41
5.1.9 Lessons Learned 42
5.2 Architektur 42
5.2.1 Architekturbeschreibung 42
5.2.2 Microservices als Architekturansatz 42
5.2.3 Kommunikationsmuster 43
5.2.4 Evolution der Architekturentscheidungen 43
5.2.5 Architekturprinzipien und Patterns 43
5.2.6 Trade-Offs 44
5.2.7 Systemüberblick 44
5.2.8 Tech-Stack 47
5.3 Systemverteilung und Deployment 47
5.3.2 GitOps & Fleet 50
5.4 Monitoring und Logging 50
5.4.1 Datenfluss und umgesetzte Architektur 50
5.4.2 Verwendung 51
5.4.3 Bekannte Grenzen 51
5.5 Qualitätssicherung 51
5.5.1 Testinfrastruktur und Hilfsmittel 51
5.5.2 Metriken 52
6 Diskussion 53
6.1 Zielerreichung 53
6.2 Challenges 53
6.2.1 Domänenspezifischer vs. generischer Framework-Prototyp 53
6.2.2 Cyberangriff der CISTEC AG 54
6.2.3 Enterprise-Infrastruktur 54
6.2.4 Kommunikation und Zusammenarbeit 54
6.2.5 Fazit 55
6.3 Weiterentwicklung 55
6.3.1 Message Queue 55
6.3.2 Transaction Logging 55

6.3.3 Nachgenerierung 55
6.3.4 Monitoring und Alerting 56
6.3.5 NATS Queue-Groups und Jetstream 56
6.3.6 Verteilung, Deployment, DevOps, Security und Skalierung 56
7 Fazit 57
8 Abbildungsverzeichnis 58
9 Tabellenverzeichnis 58
10 Literaturverzeichnis 59
11 Anhang 60
11.1 Deklaration zu genutzten (KI)-Tools 60
11.2 Zeiterfassung und Zeitauswertung 61
11.3 ADR's 63
11.4 Use Cases 79
11.5 Produkt-Backlog 92
11.6 Code of Conduct 94
11.7 Auszug aus Build-Server und CI/CD 96
11.8 Technische Diagramme 97
12 Selbstständigkeitserklärung 101

Glossar

ADR: Abkürzung für Architecture Decision Record, also für eine dokumentierte Architekturentscheidung.
Agent: Eine Software-Komponente, welche im Kontext dieser Arbeit aus bestehenden klinischen Daten mögliche Leistungen ableitet und diese in einem regelmässigen Zeitintervall aggregiert, verarbeitet und in eine Datenbank schreibt.
Agent-Core: Der Projektname und das Endprodukt unserer Arbeit.
Agenten-Framework: Generische Bezeichnung für das Agent-Core Projekt, welches grundlegende Strukturen und gebündelte Funktionalitäten für die Entwicklung von Agenten anbietet.
Agentenlauf: Gesamter Ablauf eines Agenten, also der Lebenszyklus von Start bis Stopp des Prozesses eines Agenten.
KISIM: Krankenhausinformationssystem der CISTEC AG.
KIS / Krankenhausinformationssystem: Zentrale Dokumentationssoftware im klinischen Bereich.
Kurve: Patient:innendaten, welche in einer gesundheitlichen Institution während des Aufenthalts laufend erfasst und konsultiert werden.
Leistung: Eine erbrachte medizinische Tätigkeit, ein Medikament, oder ein medizinisches Material.
Leistungserbringer:in: Medizinische Fachperson, die eine Leistung ausführt - z. B. Ärzteschaft, Pflegefachperson, Anästhesist:in, Psychiater:in.
Leistungserfassung: Allgemeiner Begriff für die Dokumentation einer erbrachten medizinischen Leistung in einem KIS, bezogen auf Patient:in und Fall.
Leistungsgenerierung: Das Gleiche wie die Leistungserfassung, spezifisch auf Agenten bezogen.
Nachgenerierung: Erneute Erstellung oder Generierung von Leistungen basierend auf die aktuelle Datenlage (z. B. bei Problemen im ERP oder Bugs im Agenten).
PO: Product Owner.
PVC: Persistent-Volume-Claim (Kubernetes-Ressource).
Shift Left: Arbeitsweise, damit die Verifikation eines Prozesses so früh wie möglich geschieht.
Sofortexport: Im Bereich der Leistungserfassung als “sofortige Rechnungsstellung” zu verstehen, z. B. wenn Patient:in ein Medikament erhält und dies gleich vor Ort im Spital bezahlt, um anschliessend gleich wieder nach Hause zu gehen.
TARDOC: Leistungskatalog im Schweizer Gesundheitswesen ab 2016 (siehe Kapitel 2.2.2).
TARMED: Leistungskatalog im Schweizer Gesundheitswesen seit 2004 (siehe Kapitel 2.2.1).

1 Einleitung

Die CISTEC AG entwickelt und betreibt das Klinikinformationssystem KISIM. KISIM unterstützt die tägliche Arbeit von medizinischen Fachkräften (Ärzt:innen, Pflegende, weitere Spezialist:innen) im Schweizer Gesundheitswesen mit einem Fokus auf Akutspitäler und Psychiatrien.
Die Leistungserfassung in Spitälern dient zur Erfassung und Abrechnung von medizinischen Leistungen bei Krankenkassen oder Patient:innen und ist ein komplexer Prozess. Entsprechend wird die Leistungserfassung durch verschiedene Tarifsysteme geregelt – im Schweizer Gesundheitswesen sind das TARMED oder TARDOC.
Das KISIM bietet für diesen Anwendungsfall eine integrierte Leistungserfassung an, welche Fachkräfte dabei unterstützt, abrechnungsrelevante Leistungen zu erfassen. Ziel der Leistungserfassung ist es, abrechnungsrelevante Leistungen (Eingriffe, Konsultationen, Medikamente, Materialien etc.) möglichst vollständig zu dokumentieren, damit die erbrachten Leistungen eines Spitals oder einer Klinik korrekt in Rechnung gestellt werden können.
Um die Leistungserfassung teilweise automatisieren zu können, führte die CISTEC AG im Rahmen des KISIM sogenannte Agenten ein, welche nach spezifizierter Logik zusätzliche Leistungsdaten generieren können. Dies spart den Anwender:innen von KISIM viel manuellen Erfassungsaufwand ein. Diese Agenten des KISIM Moduls Leistungserfassung unterstützen den Prozess der Leistungserfassung, indem sie automatisch vorhandene Leistungen, Berichte und Termindaten durchsuchen, um daraus beispielsweise Folgeleistungen, ganze Leistungsblöcke oder ergänzende Leistungen für die Abrechnung bereitzustellen. Diese Automatisierung entlastet die Fachkräfte im Bereich der Dokumentation und reduziert den manuellen Erfassungsaufwand erheblich.

1.1 Problemstellung

Die CISTEC AG betreibt im klinischen Alltag ungefähr ein Dutzend produktiver Agenten zur automatisierten Leistungserfassung auf zahlreichen Kundensystemen. Diese Agenten laufen als nächtliche Cron-Jobs und generieren Leistungen auf Basis fachlicher Regeln wie Terminstatus, laufende Massnahmen oder Dokumentationsdaten. Obwohl sie funktional ihren Zweck erfüllen, zeigen sich im Betrieb eine Reihe technischer und organisatorischer Probleme: Der aktuelle Zustand der Agentenlandschaft bei der CISTEC AG ist von heterogenen Implementationen, einem hohem Wartungsaufwand und eingeschränkter Transparenz geprägt. Die technische Architektur führt zu erheblichen Einschränkungen in Skalierbarkeit und Robustheit. Für die Mitarbeitenden bedeutet dies eine erhöhte Komplexität bei Nachgenerierungen, unklare Fehleranalysen und einen hohen Abstimmungsaufwand zwischen Fach- und Entwicklungsteam. Diese Ausgangslage unterstreicht die Notwendigkeit einer standardisierten, modularen und zukunftsfähigen Framework-Lösung, welche die aktuellen Pain Points adressiert.

1.1.1 Technische Herausforderungen

Die bestehenden Agenten wurden jeweils individuell implementiert, also ohne ein gemeinsames Framework oder eine wiederverwendbare Bibliothek. Daraus ergeben sich

folgende Probleme:

Hoher Wartungsaufwand: Jeder Agent bringt eigene Abhängigkeiten, Treiber und Bibliotheken mit. Updates müssen für jeden einzelnen Agenten durchgeführt werden, wodurch redundanter Aufwand entsteht. Copy-Paste-Code führt zudem zu Inkonsistenz und erhöhter Fehleranfälligkeit.

Datenbankabfragen: Agenten greifen direkt auf die relationale Datenbank zu und formulieren eigene SQL-Abfragen. Dadurch lassen sich Performance-Probleme nicht zentral optimieren. Hinzu kommt, dass Abfragen in der Regel sequenziell ausgeführt werden und bei wachsendem Datenvolumen kaum skalieren.

Deployment-Struktur: Jeder Agent wird als eigenständiges Deployment in Kubernetes betrieben. Dabei müssen Deployment-Konfigurationen für jeden Agenten erneut erstellt und gepflegt werden.

Keine Statusupdates der Records: Scheitert ein Agentenlauf (z. B. wegen Datenbankausfällen über Nacht), werden die verpassten Leistungen nicht beim nächsten Lauf automatisch aufgegriffen. Eine explizite Wiederanlauf- oder Recovery-Logik fehlt. Bei solchen Fällen kann mindestens ein ganzer Tag an generierten Leistungen komplett verloren gehen. Sollte der Ausfall über mehrere Tage unbemerkt bleiben, kann dies auch zu finanziellen Verlusten bei den Spitälern führen.

Kein Monitoring: Es existieren weder ein konsolidiertes Dashboard über alle Agenten noch ein Alerting System. Fehlerhafte Läufe oder Stillstände bleiben oft unbemerkt, bis Fachanwender:innen auf fehlende Leistungen hinweisen.

1.1.2 Probleme im operativen Betrieb

Neben den technischen Defiziten ergeben sich im täglichen Betrieb weitere Pain Points:

Aufwändige Nachgenerierungen: Wenn Leistungen nachträglich exportiert werden müssen (z. B. bei einem ERP-Fehler), setzen Entwickler:innen manuell Start- und Enddatum im Deployment-Chart. Aufgrund schwacher Skalierung müssen Zeiträume in kleine Pakete zerlegt werden. Eine Nachgenerierung von drei Monaten kann dadurch dutzende manuelle Iterationen erfordern und mehrere Stunden Arbeit in Anspruch nehmen.

Fehleranalyse und Debugging: Ohne einheitliche Logs und Monitoring ist die Ursachenanalyse auf Produktion zeitintensiv. Entwickler:innen müssen tief in die jeweilige Agentenlogik einsteigen, was durch fehlende Standards stark erschwert wird.

1.1.3 Organisatorische Defizite

Die fehlende Standardisierung schlägt sich auch in der Organisation nieder:

Keine klaren Guidelines: Es existiert keine Definition, was einen korrekt implementierten und sich korrekt verhaltenden Agenten ausmacht. Best Practices sind nicht dokumentiert und neue Entwickler:innen müssen sich in jeden Agenten separat einarbeiten. Das verlängert Einarbeitungszeiten und Projektlaufzeiten erheblich.

Unklare Verantwortlichkeiten: Ohne exaktes Monitoring ist nicht klar, wer die korrekte Ausführung überwachen soll. Entwickler:innen werden regelmässig ad hoc mit manuellen Nachgenerierungen betraut. Fachanwender:innen haben keine Möglichkeit, Logs einzusehen

oder Nachgenerierungen selbst anzustossen.

1.2 Zielsetzung

Die CISTEC AG hat sich aus den gerade aufgeführten Gründen dazu entschieden, den Prototypen eines entsprechenden Agenten-Frameworks in Form dieser Masterarbeit in Auftrag zu geben. Es soll ein standardisiertes technisches Grundgerüst erschaffen werden, welches die spezifischen Pain-Points der CISTEC AG adressiert, um so die Arbeit mit bestehenden Agenten massgeblich zu vereinfachen und die Umsetzung zukünftiger Agenten effizient, wartbar und skalierbar zu gestalten. Das Framework soll zentrale Funktionen bündeln, die Wiederverwendbarkeit von Logiken ermöglichen, sowie die Effizienz und Qualität der Entwicklung steigern. So soll eine Grundlage für Agenten geschaffen werden, welche es ermöglicht, nur noch die spezifischen Businesslogiken auf Agentenebene zu implementieren. Andere Basisfunktionen wie zum Beispiel Authentifizierung, Datenbankverbindung sowie Bausteine für die Datenverarbeitung werden zentral bereitgestellt.

2 Grundlagen

Um den Kontext dieser Arbeit nachvollziehen zu können, werden nun die wesentlichen Konzepte aus dem Fachbereich der Spitalinformatik erläutert, welche für das Verständnis essenziell sind. Die folgenden Kapitel legen als Wissensbausteine das benötigte Fachwissen dar, um den groben Grundriss der fachlichen Domäne zu erfassen: den klinischen Kontext der Leistungserfassung im Zusammenhang mit der CISTEC AG.

2.1 Klinisches Informationssystem

Ein Klinikinformationssystem (KIS) bildet die zentrale digitale Infrastruktur eines Spitals.
Es umfasst alle IT-Anwendungen, welche die medizinischen, pflegerischen und administrativen Prozesse unterstützen und miteinander verbinden. Während einzelne Abteilungssysteme (z. B. Labor, Radiologie) jeweils spezifische Aufgaben abdecken, verfolgt ein KIS den Anspruch, die Vielzahl dieser Systeme zu integrieren und zentral eine einheitliche, patientenorientierte Datenbasis bereitzustellen.
In der Medizininformatik existieren zahlreiche Begriffe, welche im Zusammenhang mit elektronischen Informationssystemen verwendet werden. Prokosch (2001) weist darauf hin, dass darunter häufig eine gewisse “Begriffsverwirrung” besteht (S. 1-2). So werden neben KIS auch Bezeichnungen wie Klinisches Arbeitsplatzsystem (KAS), Elektronische Krankenakte (EKA), Elektronische Patientenakte (EPA) oder Elektronische Gesundheitsakte (EGA) genutzt.
Während KAS typischerweise einzelne klinische Arbeitsplätze adressiert, beschreibt das KIS die Gesamtheit aller Systeme, die ein Spital in seinen Abläufen unterstützen (Prokosch, 2001, S. 1-2). Wichtig ist dabei, dass ein KIS nicht als einzelnes Produkt verstanden werden kann. Vielmehr handelt es sich um einen konzeptionellen Rahmen, innerhalb dessen sich die verschiedenen Anwendungen eines Spitals entwickeln und über Schnittstellen integriert werden. Ziel ist die Schaffung einer kohärenten und durchgängigen Informationslandschaft.
Im Schweizer Kontext wird der Begriff Klinikinformationssystem (KIS) bevorzugt, da er die

Gesamtversorgung innerhalb eines Spitals adressiert und nicht nur die Verwaltung betont. Ein prominentes Beispiel ist das KISIM, welches von der CISTEC AG entwickelt und unterhalten wird und als Grundlage für diese Arbeit dient. KISIM wird insbesondere in Akutspitälern und psychiatrischen Kliniken eingesetzt. Es unterstützt die Dokumentation und Steuerung klinischer Prozesse, die Leistungserfassung sowie administrative Abläufe. Durch die modulare Architektur lassen sich medizinische Kernprozesse mit abrechnungsrelevanten Anforderungen im Rahmen von TARMED oder TARDOC verbinden. Damit entspricht KISIM dem in der Literatur beschriebenen Verständnis eines KIS als integrierte Plattform, die verschiedene Anwendungen bündelt und eine durchgängige Nutzung von Patientendaten ermöglicht. Dieser monolithische Datenfluss von Patienten- und Falldaten v KISIM wurde auch als Grundlage für das Verständnis in der Konzeptphase dieser Arbeit herangezogen. Wir beziehen uns im Verlauf dieser Arbeit wie im deutschsprachigen Raum gewöhnlich auf die Begrifflichkeit “KIS”.

2.2 Leistung

Die Leistungserfassung ist ein zentraler Bestandteil der Spitalorganisation und der Abrechnung medizinischer Dienstleistungen. Unter diesem Begriff versteht man die strukturierte Erfassung und Dokumentation aller von Leistungserbringenden (z. B. Ärzt:innen, Pflegefachpersonen, Therapeut:innen) erbrachten medizinischen Leistungen, um diese gegenüber Kostenträgern (insbesondere Krankenversicherungen) abrechnen zu können.
In der Schweiz erfolgt die Abrechnung ambulanter ärztlicher Leistungen im Wesentlichen nach einheitlichen Tarifsystemen, die von weiteren gesundheitlichen Akteuren (Krankenkassen, Ärzteschaft, Spitälern und dem Bundesamt für Gesundheit) ausgehandelt und vom Bundesrat genehmigt werden.

2.2.1 TARMED

TARMED ist seit 2004 der schweizweit gültige Einzelleistungstarif für ambulante Leistungen. Jede medizinische Handlung (zum Beispiel eine Konsultation, Untersuchung oder Intervention) ist in Form einer Tarifposition abgebildet und mit einem bestimmten Punktewert versehen. Dieser Punktewert wird mit einem regional unterschiedlichen Taxpunktwert multipliziert, woraus sich der Preis der Leistung ergibt. TARMED deckt das gesamte Spektrum ambulanter ärztlicher Leistungen ab und ist für Spitäler ebenso verbindlich wie für niedergelassene Ärzt:innen (FMH Swiss Medical Association, 2025).

2.2.2 TARDOC

TARDOC ist als Nachfolgetarif zu TARMED entwickelt worden. Ziel ist es, den veralteten und vielfach kritisierten TARMED-Tarif abzulösen und eine zeitgemässe, medizinisch wie ökonomisch korrekte Abbildung ärztlicher Leistungen zu schaffen. TARDOC berücksichtigt unter anderem den tatsächlichen Ressourcenverbrauch (Zeitaufwand, Infrastruktur, Qualifikation) und soll eine transparentere und differenzierte Abrechnung ermöglichen. Die Einführung des neuen Tarifs ist für den 01.01.2026 geplant (FMH Swiss Medical Association, 2025).

2.2.3 Tarife im KIS-Kontext

Für Spitäler bedeutet die Leistungserfassung nach TARMED / TARDOC einen hohen organisatorischen und administrativen Aufwand. Leistungen müssen vollständig, korrekt und zeitnah erfasst werden, um einerseits eine faire Vergütung zu erhalten und andererseits Transparenz gegenüber Versicherern zu gewährleisten. KISIM unterstützt diesen Prozess durch die integrierte Leistungserfassung mit der Dokumentation von Leistungen. Zusätzlich (und der Hauptfokus in dieser Arbeit) findet auch die automatisierte Generierung von Zusatzleistungen durch hinterlegte Logiken oder sogenannte Agenten statt, welche erbrachte, aber noch nicht dokumentierte Leistungen identifizieren und diese automatisch erfassen.

2.3 Agent im klinischen Kontext

In der Domänensprache der CISTEC AG ist ein Agent eine Softwarekomponente, die automatisch und wiederkehrend spezifische Aufgaben zur Datenverarbeitung und Datengenerierung übernimmt. Ein Agent ist Teil des KISIM-Moduls Leistungserfassung.
In der CISTEC AG werden diese Agenten verwendet, um in diversen Use-Cases abrechenbare Leistungen für die Leistungserfassung zu generieren.
Diese Agenten laufen auf Kubernetes-Clustern als üblicherweise einmal nächtlich ausgeführte CRON-Jobs, um so die notwendigen Daten zu verarbeiten und zu generieren. Diese Daten werden anschliessend ausserhalb des KISIM-Moduls Leistungserfassung und / oder ausserhalb von KISIM weiterverwendet – z. B. durch die Spitäler für das Controlling und in der Rechnungserstellung.
Um die notwendigen Informationen für die Leistungserfassung zu generieren, fragen Agenten die benötigten Daten aus einer klinischen Datenbank ab (z. B. Leistungen, Termine, Kurvenmassnahmen). Dies geschieht aktuell über individuell erstellte und in jedem Agenten hart-codierte SQL-Statements oder über vorhandene GraphQL-Schnittstellen, welche den Backend-Service aus der Leistungserfassung ansprechen.

2.3.1 Zur Relevanz von Agenten im schweizerischen Gesundheitswesen

Die Agenten verkörpern für die CISTEC AG ein wichtiges Verkaufsargument, denn viele Kund:innen zögern bei der Neueinführung eines KIS, ob die Projektkosten für die Einführung der KISIM Leistungserfassung als Modul in der jeweiligen gesundheitlichen Institution die Mehrkosten für sie berechtigt. Konkurrenzprodukte bieten oftmals auch eine von KISIM losgelöste Leistungserfassung an, welche die gesetzlichen Vorgaben des Bundes erfüllt.
Durch das Angebot der Leistungsagenten wird die Leistungserfassung entsprechend erweitert und kann den Erfassungsaufwand in verschiedenen Fachbereichen mit automatisierten Erfassungen von Zusatzleistungen signifikant entlasten und den Kostendruck der Abteilungen dämpfen. Die Agenten heben die Leistungserfassung der CISTEC AG von anderen Produkten ab, wodurch sich die Leistungserfassung mit KISIM mittlerweile als beliebte Alternative im schweizerischen Gesundheitswesen etabliert hat.

2.4 Domänenspezifische und Systemabgrenzung

Softwareentwicklung im Schweizer Gesundheitswesen erfordert ein tiefgreifendes Verständnis über Gesetzgebungen, Prozesse, kontextbezogene Spezialanforderungen sowie ein fachliches und medizinisches Verständnis. Oftmals sind im Gesundheitswesen Legacy-Systeme im Einsatz, was dazu führt, dass viele Lösungen nicht optimal umgesetzt sind. Dies kann auch die Einarbeitung in eine Fachdomäne und das Verständnis für die Implementation teils stark erschweren.

Aus diesen Gründen wurde im Rahmen dieser Arbeit bewusst darauf verzichtet, die aktuellen Agenten, welche bei der CISTEC AG aktuell produktiv im Einsatz sind, nachzubilden.

Die bestehenden Agenten wurden jedoch tiefgreifend für die Problemanalyse und zur Bestimmung von technischen Anforderungen verwendet. In der Umsetzung wurde versucht, diese nachzubilden, was sich als zu aufwändig und wenig zielführend für den Rahmen dieses Projekts herausstellte. Daher wurden für die Implementation eigene Szenarien entwickelt, welche sich von bestehenden Agenten inspirieren liessen, ohne dabei das Fachwissen abzubilden. Zudem bearbeiten die tatsächlich eingesetzten Agenten der CISTEC AG unter anderem personenbezogene und andere sensible Daten. Diese Sicherheitsaspekte wurden im Rahmen dieser Arbeit abgegrenzt. Darunter fallen die Authentifizierung im Nachrichtenaustausch zwischen Diensten, das Loggen bestimmter Daten sowie der Zugriff auf Log-Ausgaben, sichere Passwörter sowie deren Verschlüsselung.

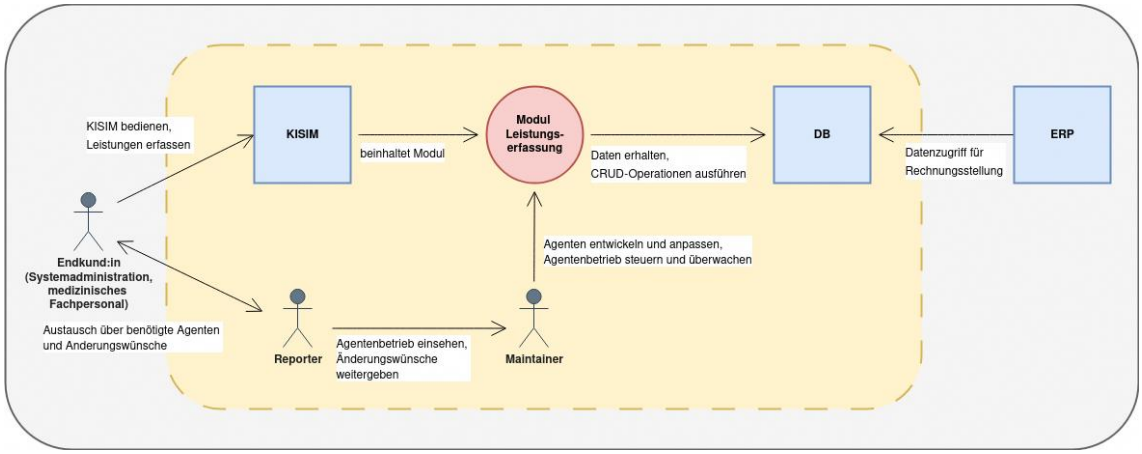


Abb. 1: Systemkontextdiagramm

Unser Augenmerk liegt auf dem Modul Leistungserfassung, welches unser System abbildet (siehe Abb. 1: Systemkontextdiagramm, rot eingefärbt). Im zugehörigen Kontext (gelb) steht links die KISIM-Software, die unter anderem das Leistungserfassungs-Modul als Baustein sowie bestehende TARMED/TARDOC-Tarifregelwerke beherbergt. Auf der rechten Seite befindet sich die Datenbank, welche erfasste sowie generierte Leistungen persistiert. Der Reporter ist am Zustand des Systems interessiert, während der Maintainer treibender Akteur unserer Use-Cases ist. Der Maintainer nutzt, wartet, pflegt und entwickelt das System weiter.

Folgende Beziehungen sind kontext-agnostisch zu betrachten: Die Beziehungen zwischen dem System und dem Endkunden sowie die Beziehung zum ERP-System, welches unter anderem auf Grundlage der Leistungen Rechnungen erstellt. Der Endkunde wiederum erfasst über KISIM Termine, Berichte und Dokumente.

3 Methodik

3.1 Projektorganisation und Vorgehen

Das Projektteam setzte sich aus den Studierenden zusammen, weitere Projektbeteiligte waren die Betreuer der OST und Ansprechpersonen der Auftraggeberin CISTEC AG (vgl. Tab. 1 Übersicht über Projektbeteiligte und Projektrollen).

Projektrolle	**Organisation**	**Person**	**Aufgabe**
Studierende / Projektteam	OST	Jvan Fadda, Guillaume Fricker, Benjamin Thormann	Konzeption, Implementation und Dokumentation des Agenten-Frameworks
Hauptbetreuer	OST	Tobias Büchel	Methodische Begleitung
Betreuer	OST	Tobias Büchel, Manuel Bauer	Formale Abnahme der Masterarbeit, Zwischenreview
Auftraggeberin	CISTEC AG	Martina Lux	Bereitstellung von Use Cases und Feedback zur korrekten Umsetzung der Anforderungen
Auftraggeberin	CISTEC AG	David Gaudliz	Fachliche Unterstützung bei technischen Fragestellungen zu Anforderungen im Bereich DevOps und Developer Experience.

Tab. 1: Übersicht über Projektbeteiligte und Projektrollen

Zur Umsetzung wählten wir ein hybrides Vorgehensmodell: Auf höchster Ebene definierten wir Phasen, ein klassisches Element aus der Wasserfall-Planung (vgl. Abb. 2: Projektplan zu Projektbeginn).

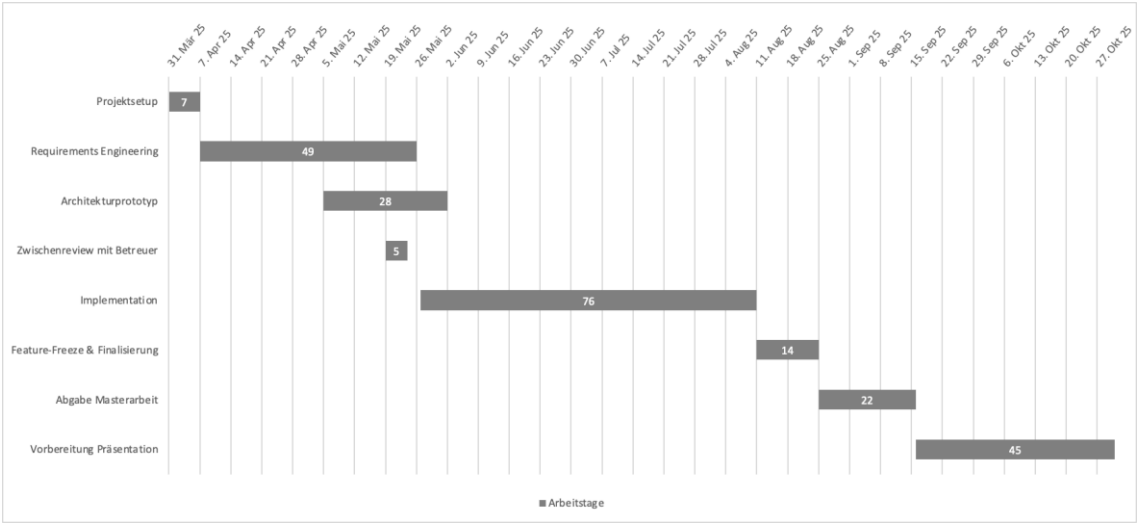


Abb. 2: Projektplan zu Projektbeginn

Jede Phase hat ein Start- und Enddatum. Arbeitspakete werden als Tickets bzw. Issues erfasst und einer Phase zugewiesen. Innerhalb der Phasen wird agil gearbeitet, und zwar mit wöchentlichen Iterationen im Projektteam (Weekly) und in zusätzlichen Feedbackschleifen mit den weiteren Projektbeteiligten (siehe Tab. 1: Übersicht über Projektbeteiligte und Projektrollen). Im Weekly bespricht das Projektteam mithilfe eines gemeinsamen Issue Boards Priorität und Status der anstehenden und aktuellen Tasks sowie aktuelle Probleme (vgl. Abb. 3: Kollaboration und Kommunikationsablauf im Entwicklungsprozess).
Die Feedbackschleifen bestehen aus einem initialen Kick-off und einem mindestens vierteljährlichen Austausch mit der Auftraggeberin sowie aus zweiwöchentlichen Meetings mit dem Hauptbetreuer. Mit diesem Vorgehen soll die gebrauchte Flexibilität und Kommunikation zwischen allen Projektbeteiligten sichergestellt und realistisch gestaltet werden.

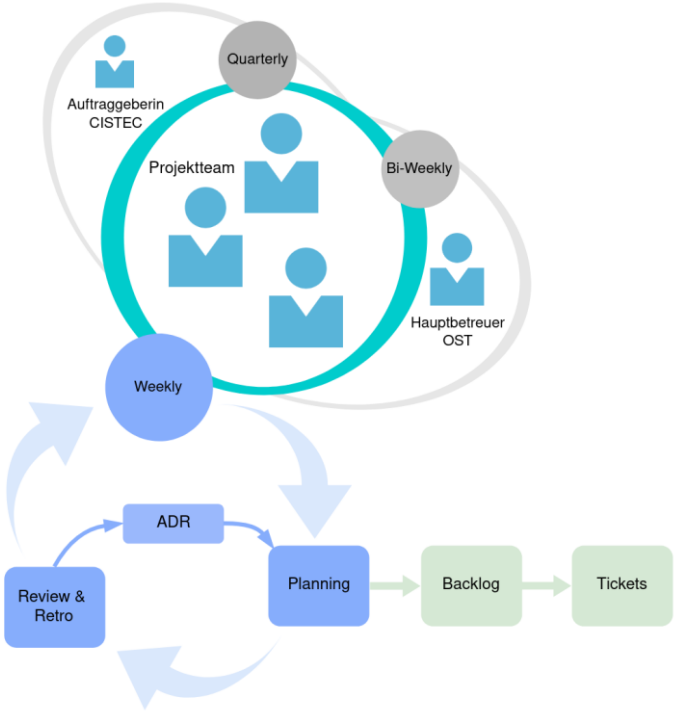


Abb. 3: Kollaboration und Kommunikationsablauf im Entwicklungsprozess

Als Tool zur Projektorganisation wurde GitLab verwendet, sodass wir das Repository, die genutzten Build Pipelines, unsere Meilensteine sowie Arbeitspakete in Form von Issues direkt mit einbeziehen und bei Bedarf untereinander verlinken konnten.
Im Rahmen der wöchentlichen Iterationen organisierten wir unsere Projektarbeit mittels Issue Board Spalten (Open, WIP, Hold, Closed) und Labels.
Letztere nutzten wir zur Aufwandschätzung, dem Markieren von besonderen Issue Typen und zum Erweitern der Issue Board Spalten wie folgt (Tab. 2: Nutzung und Erklärung der genutzten Labels):

Label-Nutzung	Label	Label-Erklärung
Issue Typ	discussion	Issue mit Unklarheiten. Zur Markierung von Besprechungsbedarf für das Weekly des Projektteams.
	documentation	Dokumentations-Tasks.
Issue Status	hold	Blockiertes Issues, z. B. durch ausstehendes Review. Grundlage für Issue Board Spalte.
	wip	Work in progress, also aktuell bearbeitetes Issue. Grundlage für Issue Board Spalte.
	bug	Issue mit Bugs oder zur Behebung von Bugs.
		Totenkopf: Aktuell tot. Zu gross für den Projekt-Scope oder tiefere Abklärungen benötigt.
Aufwandsschätzung		Erdbeere: Kleines Arbeitspaket (1 Tag).
		Apfel: Mittleres Arbeitspaket (2 bis 3 Tage).
		Ananas: Grosses Arbeitspaket (4 bis 6 Tage).

Tab. 2: Nutzung und Erklärung der genutzten Labels

3.1.1 Aufwandsschätzung

Um abschätzen zu können, wie viele Arbeitspakete wir in einer Iteration tatsächlich umsetzen und damit wir unser Projektmanagement entsprechend verbessern können, bewerteten wir während unserer Weeklys den Aufwand unserer Arbeitspakete. Den geschätzten und tatsächlichen Aufwand hielten wir auf Issue-Ebene mit Labels fest (vgl. Tab. 2: Nutzung und Erklärung der genutzten Labels). Zur Grösseneinteilung der Arbeitspakete haben wir einen Tag als die Anzahl Stunden definiert, die wir berufsbegleitend für das Studium leisten können (ca. 2 bis 6 Stunden). Zur weiteren Unterscheidung führten wir blaue Labels zur Aufwandschätzung und violette Labels für den tatsächlichen Aufwand ein.

3.1.2 Entwicklungsprozess

Der festgelegte Arbeitsablauf lässt sich anhand von Abb. 4: Branching sowie Build- und Deploymentprozess im Entwicklungsprozess nachvollziehen: Arbeitspakete werden als Tickets bzw. GitLab Issues im Backlog erfasst. Das Backlog besteht aus allen Issues, welche sich in keiner Issue Board Spalte befinden. Während der Weeklys werden Issues zugewiesen und auf einem eigenen Branch implementiert. Vor der Zusammenführung mit dem Hauptentwicklungszweig müssen zuerst ein erfolgreich abgenommenes Code Review und ein erfolgreicher Gitlab CI/CD Pipeline Durchlauf erfolgt sein. Stellt sich bei einem dieser Schritte

heraus, dass die Implementation angepasst werden muss, wird dies durch die implementierende Person erledigt und der Prozess wird wieder beim Code Review oder dem Gitlab CI/CD Pipeline Durchlauf weitergeführt. Zuletzt entsteht ein Artefakt, das bezogen und deployed werden kann.

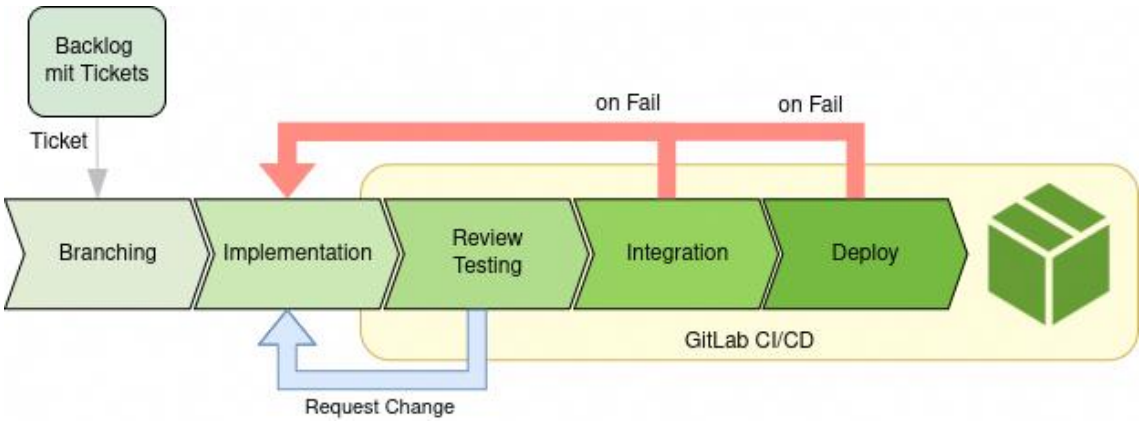


Abb. 4: Branching sowie Build- und Deploymentprozess im Entwicklungsprozess

3.2 Risikoanalyse

Zur systematischen Bewertung möglicher Unsicherheiten im Projekt wurde eine Risikoanalyse in drei Etappen durchgeführt. Dieses Vorgehen erlaubte es, sowohl frühzeitig identifizierte Risiken aus der Planungsphase als auch neu auftretende Risiken während der Umsetzung zu berücksichtigen und auf Probleme zu reagieren. Die Risikoeinschätzung erfolgte mit der folgenden Formel: *Risiko = Wahrscheinlichkeit × Einfluss.*
Wir berücksichtigten folgende Messwerte: niedrig (1–2), mittel (3–4), hoch (6–9).
Zur grafischen Darstellung wurden pro Etappe Risikomatrizen erstellt.

3.3 Stakeholder-Analyse

In diesem Abschnitt werden die für das Projekt relevanten Stakeholder festgehalten, ihre Rollen und Interessen beschrieben, sowie ihre Bedeutung für den Projekterfolg eingeschätzt. Die Analyse dient als Grundlage, um potenzielle Konflikte frühzeitig zu erkennen, Prioritäten in der Zusammenarbeit zu setzen und sicherzustellen, dass die Projektziele im Einklang mit den Anforderungen der Stakeholder erreicht werden können.
Aufgrund der Cyberattacke auf die CISTEC AG war der Projektstart in der wichtigen Kickoff-Phase stark beeinträchtigt (Jochum, 2025). Daher wurde die Stakeholder-Analyse bewusst minimal gehalten, um so auf die initialen Unsicherheiten eingehen und reagieren zu können.

3.3.1 Rollen

Für die Projektumsetzung wurden drei zentrale Rollen definiert:

Rolle	Beschreibung	Interessen und Ziele
Product Owner	Verantwortlich für die fachliche Ausrichtung des Projekts. Definiert die erwarteten Features, priorisiert diese und fungiert als zentrale Schnittstelle zwischen der CISTEC AG und dem Projektteam. Der Product Owner hat ein grosses Interesse an einer erfolgreichen Zusammenarbeit mit dem Projektteam und an der Erfüllung der gestellten Anforderungen aus dem Projekt. Definiert und priorisiert Anforderungen, gibt Feedback zu Use-Cases und übernimmt die Endabnahme.	Interessen der Endkund:innen abholen und stellvertreten. Bedürfnisse der Endkund:innen realisieren. Maintenance-Aufwand der Maintainer verringern. Neu- und Weiterentwicklungen schnell umgesetzt bekommen.
Reporter	Zuständig für die Analyse des Monitorings und die Einsicht der Logs. In einem Business-Alltag wäre der Reporter auch die erste Anlaufstelle für Endkund:innen, falls diese Auskunft zu den Agenten haben möchten. Der Reporter hat ein hohes Interesse daran, über eine grafische Benutzeroberfläche rasch an die relevanten Logs zugreifen zu können.	Einzelne Agenten in einem Monitoring einsehen. Endkund:innen die Möglichkeit bieten, Agenten selbst zu bedienen.
Maintainer	Übernimmt die technische Verantwortung für die langfristige Wartbarkeit und Stabilität der Lösung. Sorgt in einem Business-Alltag für operativen Betrieb und ist interessiert an technisch durchdachten Lösungen und kann bei der Developer Experience mitreden.	Simple, reproduzierbare Konfiguration von Agenten. Angenehme und effiziente Developer Experience. Maintenance-Aufwand verringern und möglichst automatisieren. Abläufe und Architektur optimieren, um Zeit einzusparen und einfacher verständliche Agenten.

Tab. 3: Rollendefinition

3.3.2 Stakeholder

Die folgende Matrix fasst die relevanten Stakeholder mit ihren Rollen, Interessen und Einflussmöglichkeiten zusammen:

Name	Martina Lux	David Gaudliz → Aufgrund des erhöhten Workloads durch die erfolgte Cyberattacke eingeschränkt (substituiert durch Guillaume Fricker)
Rollen	Product Owner, Reporter	Maintainer
Einfluss	Hoch	Niedrig
Interesse	Hoch	Mittel
Kontaktkanäle	Bei Bedarf, direkt per E-Mail oder vor Ort bei CISTEC AG	Eingeschränkt verfügbar, Kommunikation über Guillaume Fricker oder Gruppentreffen
Verfügbarkeit	Lange Ferien vorausgeplant → Mitte Juni bis Mitte August	Keine Kapazität, für Engineering-Inputs nur für kurze Austausche kontaktieren.
Fachwissen	Hohes fachliches Know-How.	Hohes technisches Know-How.

Tab. 4: Stakeholder-Liste

3.3.3 Stakeholder-Matrix

Die Stakeholderanalyse zeigt deutlich auf, dass Martina Lux als Hauptansprechperson sehr wichtig für den Erfolg des Projekts ist. David Gaudliz sollte beim Projekt möglichst entlastet werden, aber es kann auf spezifische Bedürfnisse des Maintainers eingegangen werden.

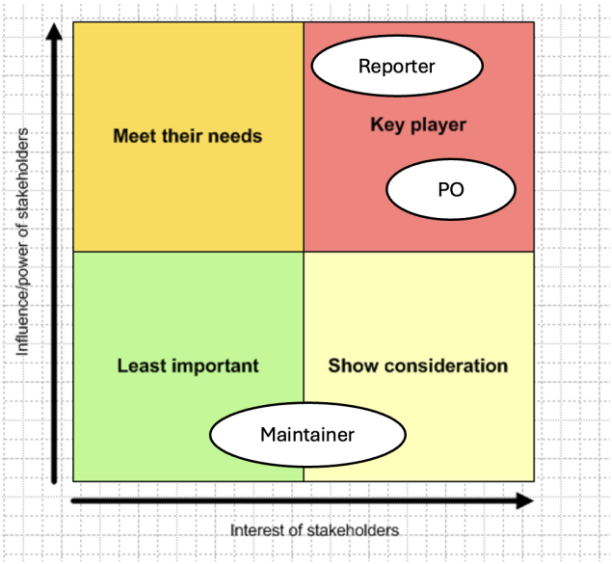


Abb. 5: Stakeholder-Matrix

3.4 Architekturentscheidungen

In der Entwicklung entpuppen sich falsche Annahmen und Unwissenheit meist während der Umsetzung. Dies kann je nach Problem zu einer grösseren oder kleineren Auswirkung führen. Wir möchten mittels ADR's festhalten, welche Entscheide wir als Team getroffen haben und deren Auswirkungen aufzeigen, so dass solche Entscheidungen im Nachgang chronologisch nachverfolgbar und nachvollziehbar sind, auch für CISTEC AG.
Ein ADR (Architectural Decision Record) ist die Kurzbeschreibung einer einzelnen getroffenen Architekturentscheidung: Sie beschreibt den Kontext, vor dem die Entscheidung getroffen wurde, die eigentliche Entscheidung und deren Folgen. Ausserdem führt ein ADR den Status der Entscheidung. Gerade in einem iterativen Vorgehen ist der Status wichtig, da sich eine Entscheidung ändern kann, wenn die Situation es verlangt. Für das Vorhaben der Architekturentscheidungen folgen wir dem Architectural Decision Records Schema nach Nygard (Nygard, 2011).
Unsere ADR-Schablone ist wie folgt:
Titel: Bestehend aus Laufnummer und einer sehr knappen Zusammenfassung der Architekturentscheidung.
Status: Status der Entscheidung (vorgeschlagen, angenommen, abgelehnt, veraltet, ersetzt).
Kontext: Welches Problem sehen wir, das uns zu dieser Veränderung oder Entscheidung motiviert?
Entscheidung: Welche Veränderung schlagen wir vor und / oder setzen wir um?
Folgen: Was wird aufgrund dieser Veränderung einfacher? Was wird aufgrund dieser Veränderung schwieriger?

3.5 Testing Strategie

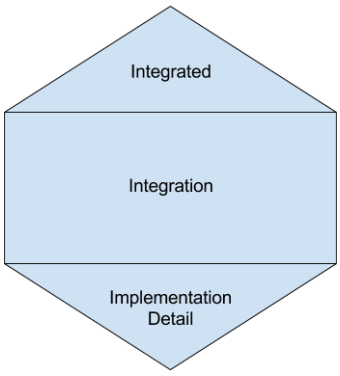


Abb. 6: Microservices Testing-Strategie (Schaffer, 2018)

Unsere Testing Strategie lässt sich wie folgt zusammenfassen:

1. Alle Tests sind durchgängig nach dem "Arrange-Act-Assert"-Muster erstellt.
2. Mittels Unit-Tests prüfen wir die reine Logik mit minimalen Abhängigkeiten.
3. Die E2E-Tests haben eine hohe Priorität, weil unser System mit verteilten Services und Messaging arbeitet, wo potenziell viele Fehler entstehen.
4. Nicht-funktionale Tests (z. B. ein gezielter Stresstest) bleiben unterstützend, sind aber nicht im Vordergrund.

Damit folgt unsere Teststrategie am ehesten dem Testing-Honeycomb-Prinzip (vgl. Abb. 6: Microservices Testing-Strategie), welches speziell für Microservice-Architekturen sinnvoll ist. Der Grund ist, dass Microservices eher klein sind und deren Komplexität nicht in der Business-Logik liegt, sondern vielmehr in der Interaktion mit anderen Services. Daher sollte gemäss dem Testing-Honeycomb-Prinzip der Fokus auf der Interaktion mit anderen Services liegen.
Ein weiterer Aspekt ist, dass der Schwerpunkt dieser Arbeit nicht darauf lag, die fachliche Logik der Agenten so weit wie möglich abzudecken – die Agenten dienen lediglich als Beispiele.

Unser Augenmerk richtete sich bewusst darauf, schnelle und solide E2E-Tests schreiben zu können und somit die Entwicklungszeit auch beim Schreiben von Tests zu reduzieren.
Für E2E-Tests haben wir die erforderlichen Infrastrukturkomponenten wie Datenbank, Message Broker, Agent-Core und optionales Logging mit Testcontainern bereitgestellt. Diese konnten wir bequem über Factories in die Tests integrieren, wodurch wir mit geringem Aufwand eine realitätsnahe und isolierte Umgebung mit klarer Struktur schaffen konnten.

3.6 DevOps-Strategie und Qualitätssicherung

Durch die Kombination von automatisierten Prüfungen und definierten Prozessen entstand ein mehrstufiges Qualitätssicherungskonzept. Dieses adressiert sowohl technische als auch organisatorische Aspekte und stellt sicher, dass die entwickelte Lösung robust, wartbar und nachvollziehbar bleibt. Die Qualitätssicherung wurde als durchgängiger Prozess verstanden, der sich von der lokalen Entwicklung über die Integration bis hin zum Betrieb erstreckte.
Dabei wurde das Prinzip von *Shift Left* verfolgt, um möglichst schnelle Feedback-Zyklen zu realisieren und deren Kosten gering zu halten. Der gesamte Entwicklungs- und Betriebsprozess wurde durch geeignete technische und organisatorische Massnahmen abgesichert.

3.6.1 Entwicklungsphase

Bereits in der lokalen Entwicklungsumgebung kamen verschiedene Massnahmen zur Anwendung, um die Codequalität sicherzustellen (vgl. Schritte Entwicklung in Abb. 7 Entwicklungsphasen):

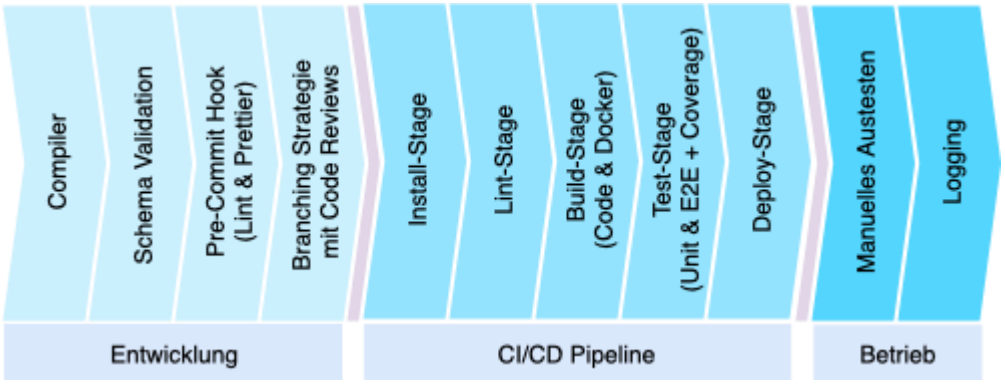


Abb. 7: Entwicklungsphasen

Die **Compiler-Prüfung** dient als erste Instanz, um syntaktische und typisierte Fehler frühzeitig zu erkennen.
Die **Schema-Validierung** stellt sicher, dass Datenstrukturen konsistent und robust gegen fehlerhafte Daten sind.
Der **Pre-Commit Hook** verhindert mittels Linter, dass fehlerhafter Code eingecheckt wird. Mit definierten Regeln wird verhindert, dass unerlaubte Code-Segmente wie `console.log` in das Repository gelangen und spezifizierte Formatierungen mit Prettier eingehalten werden.
Eine definierte **Branching-Strategie** in Kombination mit **Code Reviews** sorgt dafür, dass Änderungen nachvollziehbar diskutiert und überprüft werden können, bevor diese in den Hauptentwicklungszweig integriert werden.

3.6.2 CI/CD Pipeline

Die automatisierte GitLab Pipeline übernimmt die kontinuierliche Prüfung der Code-Base und der Bereitstellung der Docker-Images (vgl. Schritte CI/CD Pipeline in Abb. 7: Entwicklungsphase). Diese besteht aus mehreren, sequenziell durchzulaufenden Stufen. Dieser Aufbau trägt dazu bei, das Risiko auf fehlerhafte Software und die damit verbundenen Entwicklungskosten möglichst klein zu halten.
Die **Install-Stage** installiert alle Abhängigkeiten.
Die **Build-Stage** stellt sicher, dass sowohl der Applikations-Code als auch die Container fehlerfrei gebaut werden können.
Die **Lint-Stage** stellt sicher, dass definierte Regeln im Code mit ESLint eingehalten werden (derselbe Lint-Prozess wie beim Pre-Commit Hook)
Die **Test-Stage** führt Unit- und End-to-End-Tests sowie eine Coverage-Messung aus, um somit die Funktionalität und Testabdeckung zu prüfen.
Die **Deploy-Stage** ist nur auf dem Hauptzweig (main) aktiviert und dient der automatisierten Bereitstellung der Docker-Images in der GitLab-Container-Registry.

3.6.3 Betrieb

Nach der Bereitstellung (Deployment) wird die Qualitätssicherung ad hoc durch manuelles Austesten sichergestellt, beispielsweise die Einsehbarkeit der Agenten-Logs auf Grafana (vgl. Schritte Betrieb in Abb. 7: Entwicklungsphase).

3.6.4 Metriken

Zur Ergänzung der automatisierten Qualitätssicherung wurden während der Entwicklung verschiedene Metriken erhoben. Diese dienten als objektive Indikatoren zur Codequalität und Testabdeckung, welche wir mittels Coverage-Reports zur Ermittlung der Testabdeckung, der Anzahl der implementierten Tests sowie einer SonarQube-Analyse zur Prüfung auf potenzielle Sicherheitslücken und Verstösse gegen Clean-Code Prinzipien erhoben haben. Die Auswertung dieser Metriken wird im Kapitel 5.5.2 dargestellt.

3.6.5 Systemverteilung und Deployment

Für die Umsetzung wurde ein technologiegestützter Ansatz gewählt, der sich an den Anforderungen der Domäne der CISTEC AG orientiert, aber bewusst in Umfang und Komplexität reduziert ist. Die Kombination aus GitLab CI/CD, Docker, Kubernetes und Fleet erlaubt ein kostengünstiges und gut realisierbares Vorgehen, welches für den Projektrahmen (Prototyp und Machbarkeit) angemessen ist.

3.6.5.1 Continuous Integration und Continuous Deployment (CI/CD)

Die Builds und Deployments wurden über GitLab realisiert. Die GitLab Community Edition genügt als Grundlage, da es die wesentlichen CI/CD-Funktionalitäten kostenfrei anbietet und für die schulische Projektumgebung praktikabel ist.

3.6.5.2 Containerisierung (Docker)

Alle Komponenten – Core, Agents sowie Infrastruktur – wurden in Container verpackt.
Dies gewährleistet eine reproduzierbare Umgebung und entspricht modernen Standards in der Softwarebereitstellung.

3.6.5.3 Orchestrierung (Kubernetes)

Kubernetes dient als Orchestrierung-Plattform. Damit können die Services in separaten Namespaces isoliert betrieben und durch standardisierte Deployment-Artefakte verwaltet werden.

3.6.5.4 GitOps mit Fleet

Anstelle einer komplexen Deployment-Infrastruktur, wie die CISTEC AG in der Produktion nutzt (ArgoCD, Kundenclusters, mandantenfähige Rollouts), wurden simpel gehaltene Fleet-Charts eingesetzt. Fleet ermöglicht es, Deployments aus Git heraus zu steuern, benötigt jedoch keine zusätzliche Infrastruktur ausser des GitLab-Repositories. Damit blieb der methodische Ansatz einfach und realistisch, ohne den Rahmen eines Prototypen zu sprengen.

4 Problemanalyse

Ziel dieses Abschnitts ist es, die fachlichen und technischen Herausforderungen im aktuellen Ist-Zustand zu analysieren und daraus eine belastbare Problemdefinition für die Framework-Entwicklung abzuleiten. Die Problemanalyse bildet die Grundlage für die funktionalen Anforderungen, die Gestaltung der Use-Cases sowie das resultierende Domain Model.

4.1 Ist-Zustand / Ausgangslage

Die bestehenden Agenten wurden über die Jahre heterogen entwickelt und folgen keinem einheitlichen Entwicklungsstandard. Typische Merkmale des Stacks sind:

4.1.1 Strukturelle Eigenschaften

Aktuell ist kein gemeinsames Framework und kein geteilter Code vorhanden. Validierungen erfolgen ad hoc im Code mit einzeln implementierten Schema-Definitionen. Datenbankzugriffe werden häufig direkt über SQL-Statements im Code abgewickelt.

4.1.2 Persistenzschicht

Alle Agenten greifen direkt auf die relationale Datenbank (Oracle) zu. Es existiert kein abstrahierter Code, welcher Datenzugriffe kapselt, sondern dieser wird für jeden Agenten einzeln implementiert. Queries existierten für jeden Agent individuell und sind oft nicht optimiert. Joins über grosse Tabellen ohne Primary Key führen regelmässig zu Performanceproblemen.

4.1.3 Deployment, Monitoring und Betrieb

Jeder Agent wird als separates Deployment in Kubernetes betrieben. Eine zentrale Middleware oder ein zentrales Gateway existieren nicht. Die Agenten interagieren direkt mit der Datenbank. Logs werden primär in das Container-Stdout geschrieben und in Kubernetes gesammelt. Auswertungen erfolgen reaktiv, meist durch Entwickler:innen im Fehlerfall. Alerting ist nicht vorhanden. Ein Agentenstillstand bleibt oft unbemerkt, bis fehlende Leistungen gemeldet werden.

4.1.4 Kommunikation

Externe Anstösse (z. B. manuelle Nachgenerierungen) erfolgen über Änderungen in den Deployment-Charts, nicht über API's oder Message Queues. Die Agenten wissen zu keinem Zeitpunkt, welche Records sie bearbeitet haben. Wenn ein Agent für einen Tag nicht ausgeführt werden kann oder einen Absturz erleidet, gehen diese Daten für immer verloren.

4.2 Use-Cases und Anforderungen

Nach dem Erfassen des Ist-Zustands erarbeiteten wir die Soll-Szenarien, also die benötigten Features der konkreten Endbenutzer:innen des Frameworks für einen gelungenen Prototypen. Hierzu haben wir, beginnend im Stakeholder-Austausch, Quellen gesammelt, diese Quellen und die Stakeholder zum Ableiten von Use-Cases analysiert und schliesslich verfeinernde Anforderungen erarbeitet.

4.2.1 Quellen für Stakeholder- und Anforderungsanalyse

Zur Identifikation der gebrauchten Prototyp-Features und deren Abbildung in Form von Use-Cases und Anforderungen, verwendeten wir initial folgende Quellen:
Personen: Austausch mit Stakeholder Martina Lux (Rollen PO und Reporter)
Dokumente: Initiale Anforderungsliste des Stakeholders Martina Lux
Im Laufe des Projekts gewannen wir folgende Quellen hinzu und werteten diese aus, sodass sich der Fokus weiter auf die technisch wesentlichen Aspekte des Frameworks einengte:
Personen: Austausch mit Stakeholder Guillaume Fricker (Rolle Maintainer).
Dokumente: Produktdokumentationen des Moduls Leistungserfassung und einzelner Agenten:

- Ablauf Ausleitung von Anästhesieleistungen
- Leistungsgenerierung
- NoShow Agent
- Anästhesie Agent
- OAT Agent

Daten: Leistungserfassungs-Testdaten aus KISIM Datenbank (vgl. Abb. 1: Systemkontextdiagramm)

4.2.2 Akteure

Die Stakeholder beeinflussen als Interessenseigner die Ziele des Prototyps (Tab. 4: Stakeholder-Liste), sind jedoch noch nicht die tatsächlichen Akteure, welche die zu

entstehenden Features nutzen und vom Framework-Prototypen direkt profitieren werden.
Als zentrale Akteure leiteten wir **Maintainer** und **Reporter** ab, welche beide seitens CISTEC AG operieren und die Features des Frameworks nutzen und direkt profitieren.
Mögliche Akteure ausserhalb unseres Scopes haben wir verworfen (vgl. Abb. 1: Systemkontextdiagramm: Systemadministration bei Endkundschaft im Spital, welches den KISIM-Betrieb indirekt überwacht sowie medizinisches Fachpersonal, welches das KISIM als Endanwender:in bedient und Leistungen erfasst).
Ein **Maintainer** besitzt technische Expertise und entwickelt Agenten, wartet diese und nutzt das Framework. Aufgrund mangelnder Ressourcen durch einen Ransomware-Angriff bei der CISTEC AG wurde der Akteur indirekt vertreten: Initial durch den Stakeholder Martina Lux in der Rolle als Product Owner und im weiteren Projektverlauf durch den Stakeholder-Stellvertreter Guillaume Fricker in der Rolle als Maintainer.
Ein **Reporter** besitzt fachliche Expertise und vermittelt zwischen Endkundschaft und Maintainer: Ein Reporter fordert die Konfiguration und Informationen aus der Überwachung von Agenten beim Maintainer an. Dieser Akteur wurde direkt und indirekt durch den Stakeholder Martina Lux in den Rollen als Reporter und als Product Owner vertreten.

4.2.3 Use-Cases

Durch die Analyse unserer Quellen und Stakeholder konnten wir Use-Cases ableiten. Die detaillierte Use-Case-Beschreibungen sind in tabellarischer Form mit Normal- und Alternativabläufen im Anhang aufgeführt (siehe Kapitel 11.5).
Das Schema zur Beschreibung der Use-Cases orientiert sich an RUP (Rational Unified Process) bzw. an dem von Cockburn beschriebenen RUP Style (Cockburn, 2011, S. 123 f.).
Im Use-Case Diagramm (Abb. 8: Use-Case Diagramm) wird der Zusammenhang zwischen den Akteuren und ihren Interaktionen mit dem Framework sowie die Beziehungen zwischen den Use Cases klar.

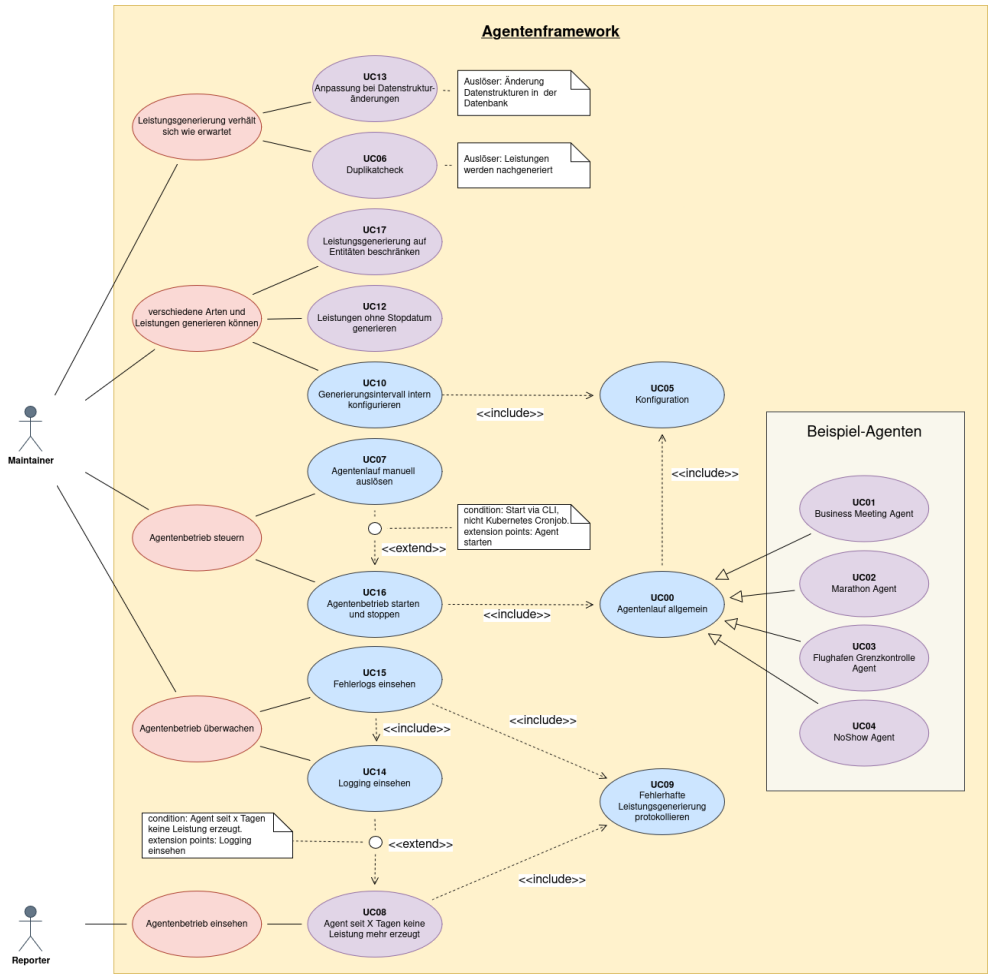


Abb. 8: Use-Case Diagramm

Um die Relevanz der Use-Cases aus Akteur-Sicht zu verdeutlichen, wurden zusätzlich die übergeordneten User Goals (Cockburn, 2011, S. 68) dargestellt. Diesem Flow folgend (das Diagramm von links nach rechts lesend), werden die Use-Cases immer feingranularer. Ausserdem lassen sich die Use-Cases in zwei Sichtweisen aufteilen, welche jeweils einen anderen Fokus setzen (vgl. Tab. 5: Use-Case Unterteilung):
Auf die **Domäne** fokussierte Use-Cases, welche sich mit fachlich orientierten Features der Leistungsgenerierung beschäftigen (violett eingefärbt)
Auf das **Agenten-Framework** fokussierte Use-Cases, welche sich mit technisch orientierten Features rund um Betrieb, Infrastruktur und Entwicklung beschäftigen (blau eingefärbt)

Use Cases mit Fokus auf gewünschte technische Features	**Use Cases mit Fokus auf gewünschte fachliche Features**
UC00 Agentenlauf allgemein	UC01 Business-Agent
UC05 Konfiguration	UC02 Marathon-Agent
UC07 Agentenlauf manuell auslösen	UC03 Flughafen-Grenzkontrolle-Agent
UC09 Fehlerhafte Leistungsgenerierung protokollieren	UC04 NoShow-Agent

UC10 Generierungsintervall intern konfigurieren	UC06 Duplikatcheck
UC14 Logging einsehen	UC08 Agent seit X Tagen keine Leistung mehr erzeugt
UC15 Fehlerlogs einsehen	UC12 Leistung ohne Stopdatum generieren
UC16 Agentenbetrieb starten und stoppen	UC13 Anpassung bei Datenstrukturen
	UC17 Leistungsgenerierung auf Entitäten beschränken

Tab. 5: Use-Case Unterteilung

Als die fundamentalsten Use-Cases, welche einen Durchstich ermöglichen, priorisierten wir UC00 und UC05. In Verhandlung mit den Stakeholdern wird zum Zwecke der Prototypenentwicklung versucht, vor allem jene Use-Cases zu priorisieren, welche sich weniger auf die Businesslogik fokussieren: Erstens ist diese Logik bei jedem Agenten einzigartig, also schwierig zu automatisieren. Zweitens ist das primäre Ziel die Entwicklung eines Framework-Prototypen für Agenten.
In diesem Sinne wurden der Use-Case 11 und die damit zusammenhängenden Akteure (Endkund:innen können den Agentenbetrieb mittels eines GUI selbst steuern) bereits sehr frühzeitig verworfen, sodass wir hierzu keine detaillierten Beschreibungen aufführen und sich diese Elemente auch nicht im Use-Case Diagramm wiederfinden.
Die Use-Cases 1 bis 4 sind Trade-Offs zwischen erstens den fachlichen und technischen Interessen der Stakeholder an der Prototypenentwicklung sowie zweitens den Rahmenbedingungen dieser Abschlussarbeit und dem benötigten Aufwand, um die CISTEC AG Infrastruktur in diesem Projektrahmen abzubilden. Diese Beispiel-Agenten sind für die Entwicklung und das Testing nötig, während sie gleichzeitig die Umsetzung von domänenspezifischen Use-Cases und Anforderungen demonstrieren.

4.3 Anforderungskriterien

Die von den Akteuren erwarteten Funktionen an den Prototypen des Agentenframeworks wurden bereits durch die Use-Cases definiert (vgl. Abb. 8: Use-Case Diagramm). Um das erwartete Ergebnis oder Verhalten der gewünschten Features verfeinert zu spezifizieren, wurden funktionale und Qualitätsanforderungen aufgestellt (vgl. Tab. 6: Funktionale und Qualitätsanforderungen). Jede Anforderung wurde priorisiert (muss, soll, oder kann) und aufgeteilt nach funktionalen und Qualitätsanforderungen gewichtet (Pro Anforderungskategorie: Durchnummerierung aller Anforderungen, wobei jede Zahl nur einmal vorkommen darf und die kleinste Zahl die höchste Priorität bedeutet), um eine Priorisierung zu erzielen. Ähnlich wie der UC11 wurden die Anforderungen FA5, FA9 und NFA8 ebenfalls frühzeitig verworfen.
Im Rahmen der Framework-Entwicklung ist es zu erwarten, dass bereits einige funktionale Anforderungen aspektorientiert sind, um z. B. Modularität zu gewährleisten. Dies ist auch in

diesem Projekt bei den funktionalen Anforderungen rund um Logging und Konfiguration (FA_REQ3, FA_REQ6) sowie bei einem von allen Beispiel-Agenten genutzten Feature (FA_REQ12) der Fall.

Funktionale Anforderung	**Qualitätsanforderung**
FA_REQ1 (muss): Der Agent muss eine vorgängige Datenselektion durchführen, damit er effizient mit grossen Datenmengen umgehen kann. Grosse Datenmengen sind 1500 - 2000 Entitäten, also ca. 2 MB.	**NFA_REQ1 (muss):** Die Anforderungen und Konfigurationen des Agenten-Frameworks müssen eindeutig, vollständig und verständlich dokumentiert werden. ANMERKUNG: Es soll auch klar sein, ob und welche Konfigurationen Agent-Spezifisch oder Global vorgenommen werden können.
FA_REQ2 (kann): Der Agent muss einen Duplikats-Check durchführen, wenn es sich um eine Nachgenerierung handelt.	**NFA_REQ2 (muss):** Das Agenten-Framework muss in TypeScript entwickelt werden, um Kompatibilität zur bestehenden Web-Infrastruktur sicherzustellen.
FA_REQ3 (muss): Das System muss ein Log führen, in dem nicht erfolgreiche Leistungsgenerierungen dokumentiert werden.	**NFA_REQ3 (muss):** Das System muss in einer Container-basierten Umgebung (Docker) betrieben werden können.
FA_REQ4 (kann): Der Agent muss konfigurierbar sein, um entweder einen 'Sofortexport' oder einen 'Export mit Verzögerung' zu unterstützen.	**NFA_REQ4 (muss):** Es muss ein automatisiertes CI/CD-Pipeline-Setup existieren, um frühzeitiges Feedback durch Tests sicherzustellen.
FA_REQ6 (soll): Der Maintainer soll das Generierungsintervall eines Agenten über eine Konfigurationsdatei oder interne Systemeinstellungen konfigurieren können.	**NFA_REQ5 (muss):** Änderungen und Events im Agentenbetrieb müssen mit einem Logging- und Monitoring-Tool nachverfolgt werden können.
FA_REQ7 (soll): Der Agent soll Leistungen auch dann erzeugen können, wenn für die Massnahme, das Rennen o. ä. noch kein Stopdatum gesetzt wurde. Ein Beispiel aus Domäne zum Verständnis: Laufende Kurvenmassnahme beim OAT-Agenten.	**NFA_REQ6 (muss):** Das Framework muss modular und erweiterbar aufgebaut sein, sodass neue Agenten effizient entwickelt werden können.
FA_REQ8 (kann): Das System soll bei ausbleibender Leistungsgenerierung einen Alert auslösen.	**NFA_REQ7 (muss):** Die entwickelte Lösung muss unter einer Open-Source-Lizenz (MIT, Apache 2.0 oder BSD) veröffentlicht werden.
FA_REQ10 (kann): Der Benutzer soll auf einem Log der nicht erfolgreichen Leistungsgenerierungen zugreifen können.	**NFA_REQ9 (soll):** Das Agenten-Framework soll die Implementierung eines neuen Agenten innerhalb von max. zehn Arbeitstagen ermöglichen. Bemerkung: Die aktuelle Implementation dauert ca. zwei - drei Arbeitswochen.

FA_REQ11 (kann): Der Benutzer soll konfigurieren können, ob der Agent läuft oder gestoppt ist (Aktivierung und Deaktivierung des Agenten).	**NFA_REQ10 (soll):** Der Agent muss bei Änderungen an der zugrundeliegenden Datenstruktur – insbesondere beim Hinzufügen von Feldern, der Änderung von Feldtypen oder einer Umstellung zwischen optional und required – durch Anpassung des zentralen Schema-Files weiterhin korrekt funktionieren, ohne dass die Agentenlogik manuell angepasst werden muss.
FA_REQ12 (kann): Das System soll die Generierung auf bestimmte Entitäten (über ID) einschränkbar machen.	

Tab. 6: Funktionale und Qualitätsanforderungen

In Bezug auf die CISTEC AG sollen die Qualitätsanforderungen (NFA's) eine moderne, kompatible Infrastruktur (NFA_REQ3, NFA_REQ4) und eine erfolgreiche Übernahme des Framework-Prototypen sicherstellen (NFA_REQ1, NFA_REQ2, NFA_REQ7).
Ausserdem sollen sie die Senkung des Implementations- und Betriebsaufwands direkt (NFA_REQ9) oder indirekt durch moderne und nachhaltig gestaltete Architektur (NFA_REQ6, NFA_REQ10) und Logging (NFA_REQ5) unterstützen.
Die Relationen zwischen Use-Cases und funktionalen Anforderungen (FA's) sind in Tab 7: Traceability Matrix nochmals aufgezeigt.

	FA_REQ1	**FA_REQ2**	**FA_REQ3**	**FA_REQ4**	**FA_REQ6**	**FA_REQ7**	**FA_REQ8**	**FA_REQ10**	**FA_REQ11**	**FA_REQ12**
UC00 Agentenlauf allgemein	X		X							
UC05 Konfiguration			X	X	X					
UC07 Agentenlauf manuell auslösen			X							
UC09 Fehlerhafte Leistungsgenerierung protokollieren			X							
UC10 Generierungsintervall intern konfigurieren					X					
UC14 Logging einsehen			X							
UC15 Fehlerlogs einsehen								X		
UC16 Agentenbetrieb starten und stoppen									X	
UC01 Business-Agent										X

UC02 Marathon-Agent										X
UC03 Flughafen-Grenzkontrolle-Agent										X
UC04 NoShow-Agent										X
UC06 Duplikatcheck		X								
UC08 Agent seit X Tagen keine Leistung mehr erzeugt							X			
UC12 Leistung ohne Stopdatum generieren						X				
UC13 Anpassung bei Datenstrukturen										
UC17 Leistungsgenerierung auf Entitäten beschränken										X

Tab. 7: Traceability Matrix

4.4 Randbedingungen

Die wesentlichen Rahmenbedingungen wurden durch die CISTEC AG vorgegeben:

1. TypeScript als bevorzugte Programmiersprache
2. Node.js als bevorzugte Laufzeitumgebung
3. Datenhaltung über relationale Datenbanken (PostgreSQL und Oracle werden bei CISTEC AG bereits eingesetzt)
4. Keine Lizenzierungskosten: Der Betrieb und die Übergabe des Frameworks darf keine zusätzlichen Kosten für die CISTEC AG generieren
5. Permissive Open Source Lizenz, damit die CISTEC AG das Framework bei sich intern frei nutzen und erweitern darf

4.5 Domain Model

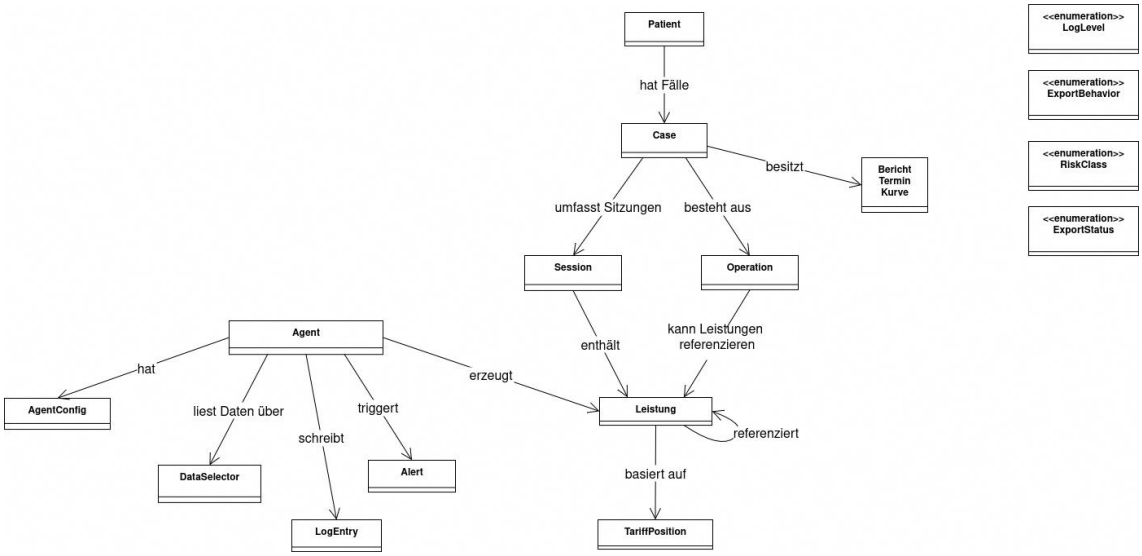


Abb. 9: Domain Model

Aus den Use Cases wurde ein Domänenmodell abgeleitet, welches die wichtigsten Komponenten für die automatisierte Leistungserfassung mittels Agenten veranschaulicht. Das Domain Model umfasst unter anderem folgende zentrale Entitäten:

Agent ist die zentrale Logik-Einheit. Er verarbeitet Falldaten, generiert Leistungen und exportiert diese in eine Datenbank. Jeder Agent:

- hat seine eigene Konfiguration
- nutzt ***Datenselektor***, um auf Daten (Berichte, Termine etc.) und die Datenbank zuzugreifen.
- generiert die zu erstellenden Leistungen aufgrund von TARMED und TARDOC (bestehendes Regelwerk)
- schreibt Logs und kann Alerts auslösen

Konfiguration enthält spezifische Einstellungen des Agenten:

- Generierungs-Intervalle
- Zeithorizont der Generierung
- Exportverhalten (sofort oder verzögert für mögliche Nachbearbeitung)
- Log-Level
- Aktivitäts-Status, ob der Agent laufen soll oder nicht

Datenselektor definiert die Kriterien, die bestimmen, welche Daten ein Agent vor der Leistungsgenerierung verarbeitet.

Leistung ist das von einem Agenten zu erzeugende Ergebnis. Sie wird nach bestimmten Regeln und gemäss Konfiguration exportiert. Eine Leistung kann sensible Daten enthalten, da sie im Zusammenhang mit Patient:innen- und Falldaten steht.

Logeintrag dokumentiert Fehler oder fehlgeschlagene Generierungen, um die Nachvollziehbarkeit zu gewährleisten. Es gibt unterschiedliche Levels wie z. B. DEBUG, INFO, ERROR.

Alerting löst eine Warnung aus, wenn ein Agent über einen definierten Zeitraum keine Leistungen generiert. Ein Agent kann auch selbst ein Alerting auslösen, z. B. bei kritischen Fehlern.

Nutzer (CISTEC AG Entwickler:innen, also Maintainer) kann Agenten starten/stoppen, Konfigurationen einsehen und Logs prüfen.

Datenbank ist nicht als Entität aufgeführt: Sie existiert ausserhalb des Systems und ist damit out of Scope, da sie in diesem Kontext domänengerecht als Bericht, Termine, Kurve modelliert wird.

4.6 Risikoanalyse

Aufbauend auf der beschriebenen Methodik werden in diesem Abschnitt die im Projekt identifizierten Risiken vorgestellt. Im Vordergrund stehen dabei die während der Implementierung neu erkannten technische (Kapitel 4.6.2) sowie organisatorischen Risiken (Kapitel 4.6.3). Jedes Risiko wurde hinsichtlich seiner Eintrittswahrscheinlichkeit und

Auswirkung bewertet und mit möglichen Gegenmassnahmen versehen. Durch diese strukturierte Analyse konnten sowohl ein Überblick über den aktuellen Risikostatus gewonnen werden als auch die Grundlage für spätere Vergleiche mit tatsächlich eingetretenen Ereignissen geschaffen werden.
Die nachfolgenden Tabellen und Matrizen bauen auf die initial definierten Risiken (Kapitel 4.6.1) auf, teilen diese besser auf und erweitern die Gesamteinschätzung, dokumentieren diese und bilden die Grundlage für die spätere Auswertung im Ergebnisteil (Kapitel 5.1.8). Die nachfolgenden Kapitel dokumentieren diese Einschätzungen:

4.6.1 Initial definierte Risiken

Zum Projektstart wurde eine initiale Risikoanalyse betrieben, welche sich mit den grundlegenden möglichen Problemen befasst

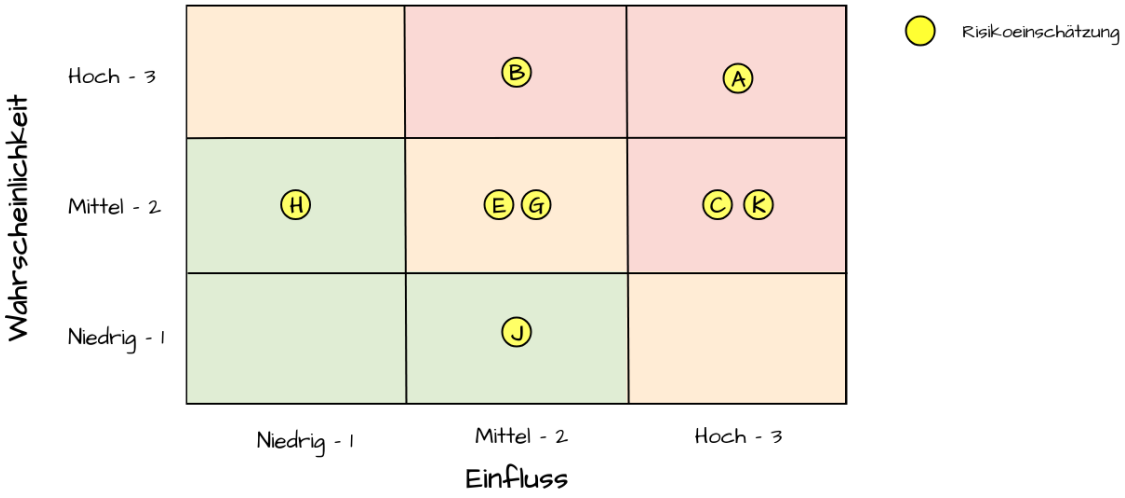


Abb. 10: Risikomatrix der initial definierten Risiken

Diese Risikoanalyse war noch stark an die domänenspezifischen Anforderungen der CISTEC AG gekoppelt und erwies sich zum Projektstart als hilfreich, musste aber beim Start der Umsetzung sehr rasch neu evaluiert werden. Dieser Teil wird deshalb nur zur Nachvollziehbarkeit aufgeführt und nicht weiter behandelt.

4.6.2 Technische Risiken

Im Verlauf der Implementierung traten verschiedene potenzielle technische Risiken hervor, die in der initialen Planungsphase noch nicht vollständig absehbar waren. Dazu zählen insbesondere Fragestellungen rund um die Architekturentscheidungen, die Komplexität der Fachdomäne und der einzelnen Komponenten sowie die Handhabbarkeit der Entwicklungsumgebung.
Um diese Aspekte systematisch angehen zu können, wurden die Risiken in einer Risikomatrix erfasst und mit geeigneten Massnahmen ergänzt. Die folgende Analyse zeigt die identifizierten technischen Risiken, ihre Bewertung sowie die vorgeschlagenen Gegenmassnahmen:

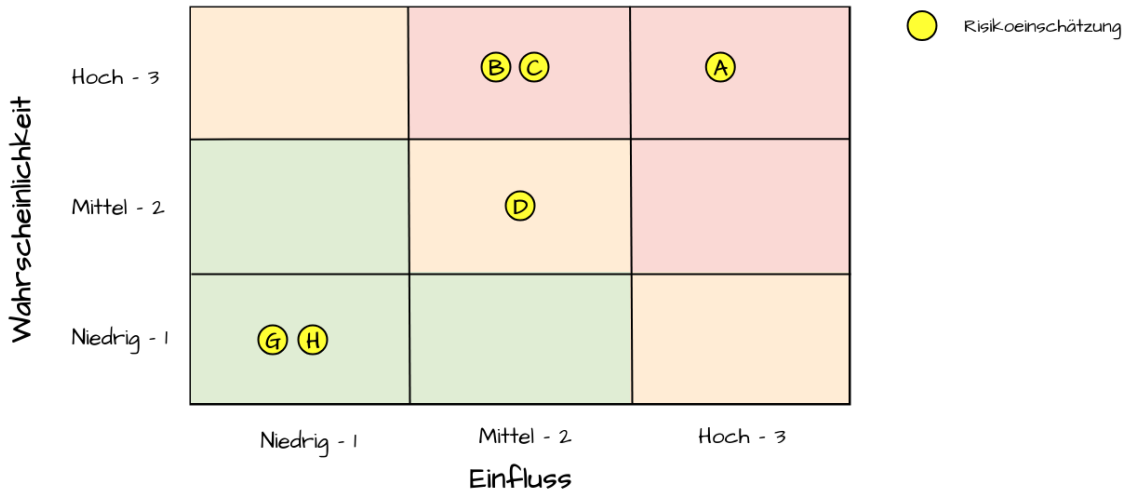


Abb. 11: Risikomatrix der technischen Risiken

Kürzel	Einschätzung	Beschreibung	Massnahme
A	9	Performanceprobleme bei Re-Exports oder grossen Datenmengen	Architektur skalierbar entwerfen, DB-Queries limitieren, Paging einbauen
B	6	Komplexität der Agentenlogik wird unterschätzt	Use-Cases einzeln verifizieren, generischere Use-Cases priorisieren
C	6	WebUI-Komponenten zu aufwändig (Monitoring, Trigger, Nachgenerierung)	Keine UI-Umsetzung, sondern auf CLI oder Fleet fokussieren (User sind Entwickler:innen, keine Endkund:innen)
D	4	Kubernetes-Deployment zu komplex für lokale Umgebung	Einfache Fleet-Charts und Templates bereitstellen
G	1	Agentenlogik zu stark auf alte Struktur der CISTEC AG zugeschnitten	Abstrakte Use Cases verwenden, Fachtransfer absichern
H	1	Uneinheitliche Verwendung von Zod und class-validator, keine konsequente Einhaltung von DTOs	Validierungsstrategie in ADR dokumentieren und durchsetzen, den Einsatz von Shared Types abwägen

Tab. 8: Erwartete technische Risiken

4.6.3 Organisatorische Risiken

Neben den technischen Herausforderungen müssen auch organisatorische Risiken berücksichtigt werden, die sich insbesondere mit dem Zeitmanagement, Teamkoordination und die Zusammenarbeit mit den Stakeholdern auseinandersetzen. Diese Faktoren sind für den Projekterfolg ebenso entscheidend wie die technische Umsetzung, da sie die Effizienz und Qualität der Arbeit massgeblich beeinflussen. Die nachfolgende Analyse fasst die ermittelten organisatorischen Risiken zusammen, bewertet sie nach Eintrittswahrscheinlichkeit und Auswirkung und stellt anschliessend die vorgesehenen Gegenmassnahmen dar:

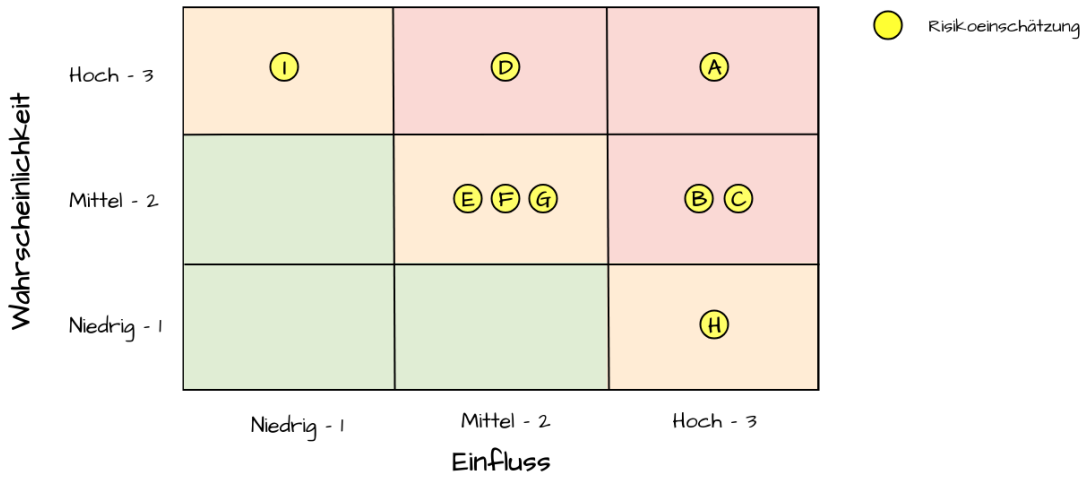


Abb. 12: Risikomatrix der organisatorischen Risiken

Kürzel	Einschätzung	Beschreibung	Massnahme
A	9	Zeitrahmen unterschätzt: Parallele Entwicklung, Thesis, Review-Phasen kollidieren	Umfang frühzeitig abgrenzen, Projektplan mit Puffer kommunizieren
B	6	Stakeholderwechsel oder plötzliche Umdisposition durch CISTEC AG	Projektentscheidungen und Anforderungen schriftlich in ADR's dokumentieren
C	6	ADR's nicht einheitlich oder widersprüchlich dokumentiert	ADR als Teil vom Reviewprozess einschleusen
D	6	Unklare Abnahme- oder Erfolgskriterien	Kriterien für erfolgreichen Prototyp schriftlich fixieren, Review-Termine einplanen
E	4	Team zieht nicht an einem Strang (z. B. Rollen unklar, Ownership fehlt)	Klare Verantwortlichkeiten im Team festlegen, Weekly Syncs zur Abstimmung einführen
F	4	Dokumentation wird zu spät oder unstrukturiert erstellt	Dokumentationsstruktur früh definieren, regelmässig aktualisieren
G	4	Projektmanagement-Tool wird nicht effektiv genutzt	Einheitliches Toolset festlegen und aktiv nutzen
H	3	Projektfokus verschiebt sich durch externe Anforderungen (z. B. Bugs oder Unklarheiten bei CISTEC AG)	Feature-Umfang definieren, Feature-Freeze ab gewissem Zeitpunkt einführen und einhalten
I	3	Kommunikation mit CISTEC AG unregelmässig (z. B. PO nicht erreichbar, in den Ferien)	Fragen frühzeitig stellen, Antworten dokumentieren, technische Feinheiten mit Maintainer und Stv. klären

Tab. 9: Erwartete organisatorische Risiken

4.6.4 Finale Risikoanalyse

In der abschliessenden Etappe der Risikoanalyse wurden teilweise sowohl bestehende Risiken zusammengeführt als auch neue Risiken berücksichtigt, welche sich während der Implementation herauskristallisierten. Das Ziel war hier, die Risikoanalyse für das Projekt weiter zu festigen und systematisch auszuwerten. Dabei wurden sowohl technische als auch organisatorische Aspekte berücksichtigt, um eine vollständige Übersicht über das Risikoprofil des Projekts zu erhalten. Diese wird in den Ergebnissen im Kapitel 5.1.8 dann im Detail ausgewertet.

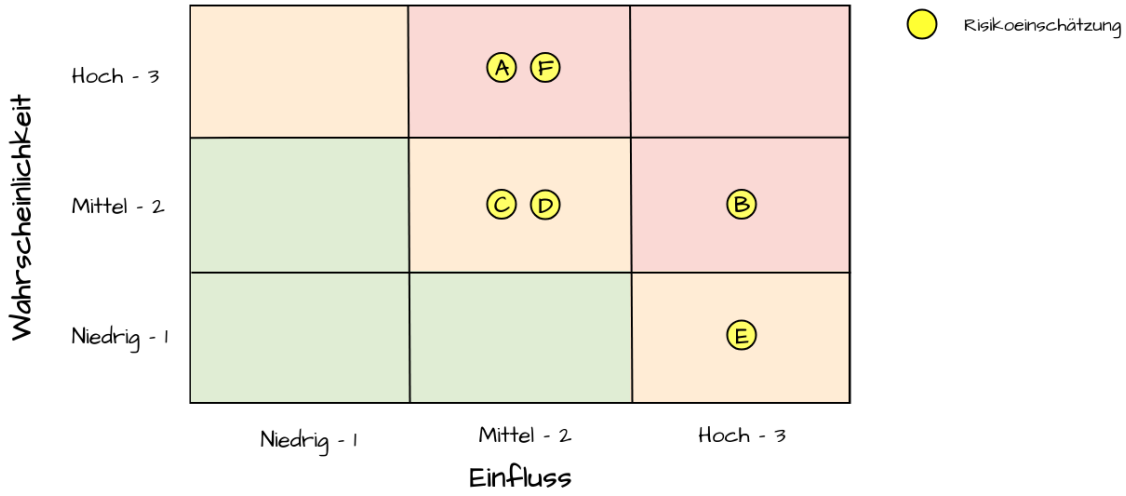


Abb. 13: Risikomatrix der finalen Risiken

Kürzel	Einschätzung	Beschreibung	Massnahme
A	6	Bereitstellung der Testdaten durch Stakeholder verzögert sich nach Cyberattacke	Frühzeitige Bereitstellung von und Validierung mit Stakeholdern
B	6	Testdaten aus bestehendem KISIM sind für Entwicklungszwecke qualitativ nicht ausreichend nutzbar	Eigene Domäne definieren, um so die Bedürfnisse der Stakeholder abzudecken
C	4	Eingeschränkter Stakeholderkontakt, Hauptstakeholder längere Zeit abwesend, kein regelmässiger Austausch möglich	Gezielte Abnahmen nach Ferienabwesenheit, grösserer Fokus auf Selbsteinschätzung
D	4	Entwicklungsprozess gerät ins Stocken, Prozesse nicht systematisch gelebt, Dokumentation und Abnahmen lassen teils mit Verzögerung auf sich warten	Definierte Prozesse, zeitnahe Dokumentation von wichtigen Entscheidungen mittels ADR's
E	3	Ungleichmässige Arbeitslastverteilung und temporäre Abwesenheiten im Team, die den Fortschritt verzögern könnten	Rollenverteilung, regelmässige Abstimmungen, Code Sessions
F	6	Fachspezifische Domäne statt einem	Abstraktion auf Framework-

		generischen Framework, Stakeholder-Vorgaben zu spezifisch und / oder nicht auf Framework übertragbar	orientierte Use-Cases und eigene Definitionen von Agenten, die den Funktionsumfang von CISTEC AG -spezifischen Agenten abdecken

Tab. 10: Erwartete finale Risiken

4.7 Architektur

Die Architektur- und Technologieauswahl wurde in enger Koppelung mit den Stakeholder-Interessen, der Risikobetrachtung und den zuvor definierten Qualitätszielen getroffen.
Es galt, eine langfristig wartbare, skalierbare und erweiterbare Lösung zu entwerfen, welche die Betriebskosten nicht unnötig erhöht. Es ergaben sich drei zentrale Qualitätsziele:
Wartbarkeit: Klare Strukturen, Minimierung von Redundanz, gute Testbarkeit.
Skalierbarkeit: Agenten sollen unabhängig voneinander deploy- und versionierbar sein.
Transparenz: Logs, Monitoring und Fehlermeldungen müssen konsolidiert und einfach einsehbar sein.
Erweiterbarkeit: Neue Agenten müssen sich mit minimalem Aufwand in die bestehende Infrastruktur einfügen.

4.7.1 Auswertung möglicher Technologiestacks

Es wurden drei Frameworks für die Auswahl des Technologiestacks untersucht:

Framework	Eigenschaften
AdonisJS	● Laravel-ähnliches JavaScript-Framework ● Kleine Community, rein Community-getrieben, wenige Lernressourcen ● Mögliches Risiko hinsichtlich langfristiger Wartbarkeit
Express	● Maximale Kontrolle und Flexibilität ● Gefahr von chaotischem Code, da wenig Struktur vorgegeben wird ● Wartbarkeit hängt stark von der Disziplin der Entwickler:innen ab ● Keine integrierte Microservice-Unterstützung
NestJS	● Wird bereits in der CISTEC AG verwendet ● Wird von einer Firma maintained, breite Community und Lernressourcen ● OOP-Ansätze, Modularität, Dependency Injection, Decorators ● Integrierte Microservice-Schnittstellen, einfache Integration in Kubernetes

Tab. 11: Auswertung Technologiestack

Die Entscheidung fiel aufgrund der vielen genannten Vorteile eindeutig auf NestJS.
Diese erfolgte dabei stets im Abgleich mit den Stakeholder-Bedürfnissen (z. B. Wartbarkeit und Transparenz für Entwickler:innen, Monitoring für Projektleiter:innen), den Qualitätszielen (Skalierbarkeit, Robustheit) und der Risikoanalyse (Minimierung von Bottlenecks und Legacy-Abhängigkeiten). Zum weiteren Technologiestack hinzu kam die Entscheidungen für PostgreSQL als Datenbank, TypeORM als Library für das Daten-Mapping, Jest zwecks Testing sowie Grafana zwecks Monitorings.

4.7.2 Walking Skeleton

Im Rahmen der Arbeit wurde ein Walking Skeleton entwickelt, der als technischer Prototyp diente und den architektonischen Rahmen für die Umsetzung der Agenten vorgab. Das Ziel war es, frühzeitig einen minimal funktionsfähigen Durchstich durch alle relevanten Systemkomponenten zu realisieren, um technische Risiken zu adressieren und zentrale Architekturentscheidungen zu validieren.
Zu Beginn wurden zwei alternative Ansätze für das Walking Skeleton skizziert:
Version 1: Ein schlankes Monolith-Design, bei dem Core- und Agentenlogik in einem Prozess zusammengeführt werden. Vorteil: schnelle Umsetzung; Nachteil: keine saubere Trennung der Verantwortlichkeiten.
Version 2: Agenten als eigenständige Deployments mit direkter Kommunikation zu einem zentralen Core-Service über API oder Message Queue. Vorteil: zentrale Wartung und klare Schnittstellen; Nachteil: höherer initialer Setup-Aufwand.
Bei der Abwägung dieser Varianten fiel die Wahl schliesslich auf einen serviceorientierten Ansatz. Damit konnte sowohl die Skalierbarkeit als auch die künftige Versionierung von Agenten und Core berücksichtigt werden. Die umgesetzte Version des Walking Skeletons war Gegenstand des Zwischenreviews am 26.05.2025 und entspricht dem Commit bec4b20bcc6967ce359a7685e2ad7f297b927876 im Git-Repository.
Das Walking Skeleton umfasste in der Endausführung folgende Merkmale:

- Eine lauffähige NestJS-Anwendung mit Basisfunktionalität für Core-Module
- Eine erste Agenten-Implementation als Cron-Job, welche den definierten Use Case in seinen Grundzügen abbildet
- Eine funktionale Anbindung an die Datenbank mittels TypeORM
- Initiale API-Endpunkte inklusive automatisierter Tests
- Ein rudimentäres Deployment auf Kubernetes (lokal über Minikube)

Damit wurde ein vollständiger Durchstich über alle Layers erreicht: Datenhaltung, Businesslogik, API-Schnittstelle und Deployment waren im Kern erprobt. Das Walking Skeleton konnte also sicherstellen, dass die technische Basis stabil und erweiterbar integriert wurde. Damit konnte das Team die verbleibende Entwicklungszeit gezielt auf die Umsetzung der Agentenlogik und der Erweiterungen am Framework fokussieren.

4.7.3 Architekturentscheide / Architecture Decision Records (ADR's)

Während des Projektverlaufs sind wir auf einige Herausforderungen gestossen und mussten bezüglich der Architektur Entscheidungen treffen. Alle ADR's sind im Anhang aufgeführt (siehe Kapitel 11.4). Die wichtigsten ADR's werden nun erläutert.

4.7.3.1 0003 Architektur-Prototyp Agentenframework

Zu Beginn der Arbeit wurde der Walking Skeleton implementiert, um einen vollständigen Durchstich durch alle Layers des Systems zu ermöglichen. Das Walking Skeleton stellte sicher, dass grundlegende Konzepte wie Schema-Validierung, Persistenz mit TypeORM und CI-Tests auf Agentenebene (also für jeden einzelnen Agenten) von Beginn an berücksichtigt wurden. Für das Zwischenreview konnte ein konsistenter Minimal-Durchstich bis zur Datenbank präsentiert werden, welcher die Basis für die weitere Arbeit bildete. Allerdings war der Walking Skeleton nur als Zwischenstand gedacht und wurde durch spezifische Entscheidungen weiterentwickelt und verfeinert. Damit erfüllte ADR0003 seine Rolle als Architekturprototyp, wurde aber schrittweise durch die nachfolgenden ADR's erweitert oder abgelöst. Es diente als Fundament, auf dem alle späteren Entscheidungen aufbauen konnten.

4.7.3.2 0007 Fachliche Kriterien

Der Kontext, die Umsysteme und die fachliche Agentenlogik der CISTEC AG überstiegen die Ressourcen des Projektrahmens. Die Komplexität war im Rahmen dieser Arbeit nicht machbar. Deshalb haben wir uns in Absprache mit Martina Lux (Stakeholder und PO seitens CISTEC AG) aus dem klinischen, also fachlichen Kontext gelöst, ohne den ursprünglichen Funktionsumfang einzuschränken. Diese Entscheidung erwies sich als sehr gelungen, kam aber leider etwas spät.

4.7.3.3 0009 Microservices

Im Verlauf der Umsetzung stieg die Komplexität des Projekts stetig an. Um die Kopplung zwischen den Diensten (Agent-Core und Agenten) zu minimieren, einen modernen Ansatz zu verfolgen und die Basisfunktionalität weiter herauszutrennen, beschlossen wir, ein verteiltes System mit Microservices umzusetzen. In der Folge konnten wir die Agentenlogik sauber kapseln sowie den Datenzugriff über den Agent-Core reduzieren. Die ermöglichte Skalierung und unterschiedliche Release-Zyklen sind weitere Vorteile. Zu den Nachteilen gehören eine komplexere Architektur, erschwerte Bedingungen bei E2E-Tests sowie die Schnittstellenpflege.

4.7.3.4 0010 Kommunikation via NATS

Anfänglich hatten wir REST als Kommunikationsprotokoll zwischen den Diensten angedacht. Da das eingesetzte NestJS-Framework eine Microservice-Unterstützung anbietet, versuchten wir, möglichst viel damit zu lösen. REST gehörte nicht zu den unterstützten Protokollen. Damit wir auf einen selbst entwickelten Kommunikationsbus mit Idempotenz verzichten konnten, hatten wir uns für die unterstützte Transportschicht mit dem Messaging Service NATS entschieden. Diese bietet mehrere Kommunikationsmuster, zum Beispiel Fire-and-Forget, Request/Reply sowie Publish/Subscribe. Darüber hinaus stellt NATS optionale Features wie Queue Groups zur Lastverteilung und über JetStream zusätzlich Persistenz & Replay für einen

„at least once“ Zustellungsmodus bereit. Damit haben wir für viele Ausbaumöglichkeiten vorgesorgt, nehmen dafür jedoch einen zusätzlichen Dienst in der Architektur in Kauf.

4.7.3.5 0012 Standardisierung der Agenten-Pipeline

In den bestehenden Agenten waren die fachliche Logik und die Persistenzschicht stark miteinander vermischt. Änderungen wurden direkt in der process-Methode geschrieben, was zu einem unklaren Ablauf führte und somit die Testbarkeit und Wartbarkeit erschwerte.
Im Rahmen der Arbeit wurde deshalb ein Standardprozess des Agentenlaufs eingeführt, welcher die Verantwortlichkeiten klar trennt: Die processItem-Methode kümmert sich ausschliesslich um die Berechnung und liefert ein Ergebnis-DTO zurück, während die update- und create-Methoden für die Persistierung verantwortlich sind. Dies führte zu konsistenten Agenten und verbesserte die Trennung von Logik und Datenzugriff. Tests können gezielt auf der reinen Processing-Ebene ohne Seiteneffekte ausgeführt werden.

4.8 Systemverteilung und Deployment

Die Ausgangslage der CISTEC AG ist eine komplexe Agentenlandschaft, die mit selbst entwickelten Deployment-Prozessen auf Basis von Argo-CD und internen Tools betrieben wird. Dies stellt die Maintainer vor Herausforderungen: Es existiert eine Vielzahl von Agenten und Services mit sehr unterschiedlichen Anforderungen, die Komplexität der Build- und Deployment-Prozesse ist hoch und die Integration in die bestehende Infrastruktur tief und schwer nachvollziehbar. Daher soll die Agentenlandschaft mithilfe des Agenten-Frameworks auf mehrere, lose gekoppelte Dienste verteilt werden. Dabei sollen die Deployments vereinheitlicht und Abhängigkeiten explizit gemacht werden. Auch eine zukünftige horizontale Skalierung soll möglich sein.
Im Rahmen der Prototypentwicklung konnte (Zeit- und Budgetbegrenzung) und sollte (Produktfokus bewahren) diese vollständige Landschaft nicht nachgebaut werden. Stattdessen wurde ein vereinfachtes, aber funktionales Setup gewählt: Core, Agenten und zentrale Infrastruktur (NATS, PostgreSQL, Monitoring), orchestriert über Fleet / GitOps.
Auch auf Features wie Secrets-Management, High Availability, Persistenz für Loki oder ausgereifte Skalierungs-Strategien wurde bewusst verzichtet: Der Fokus des Projekts lag darauf, die Grundbedürfnisse und Basisanforderungen an eine Agentenarchitektur in einem Prototypen abzubilden – nicht die gesamte Produktionsumgebung der CISTEC AG.

5 Ergebnisse

In diesem Kapitel zeigen wir auf, was wir erreicht haben und wie wir dorthin gekommen sind. Wir geben einen Überblick, wie der Prototyp aufgebaut ist, welche Use-Cases und Anforderungen abgedeckt wurden und welche nicht. Mit dem methodischen Vorgehen konnten wir die Ausgangslage grösstenteils gut lenken und konnten teilweise auch Risiken früh erkennen. Durch unsere Problemanalyse haben wir die fachlichen Anforderungen von den technischen unterschieden und konnten eine transparente Architektur auf die Beine stellen. Bei dieser wird schnell klar, was der technische Basis-Funktionsumfang ist und was fachlich zugeordnet wird.

5.1 Projektorganisation

Die Projektarbeit startete mit einem offiziellen Kick-off Meeting mit der CISTEC AG am 09.04.2025. Dabei wurden die Rahmenbedingungen geklärt und ein Systemüberblick geschaffen. Erste gewünschte Anforderungen konnten bereits dort benannt werden. Auf Seiten der CISTEC AG stand Martina Lux (Product Owner, Reporter) als primäre Ansprechpartnerin zur Verfügung.

5.1.1 Zusammenarbeit mit der CISTEC AG

Der Kontakt zur CISTEC AG war weniger technisch geprägt, sondern erfolgte primär über die fachliche Rolle des Product Owners. Ein direkter Austausch mit Entwickler:innen der CISTEC AG war zwar vorgesehen, konnte aber aufgrund der hohen Arbeitslast nach dem Cyberangriff kaum in Anspruch genommen werden. Darüber hinaus führte die längere Abwesenheit des PO (zwei Monate Ferien) dazu, dass die Kapazitäten für regelmässige Reviews auf Seiten der CISTEC AG eingeschränkt waren.
Eine Übergabe von Artefakten (vgl. Abb. 4: Branching sowie Build- und Deploymentprozess im Entwicklungsprozess) an den PO erfolgte daher nicht. Dies war insofern unproblematisch, da es sich um einen Framework-Prototypen handelt, dessen Integration in die produktive CISTEC-Infrastruktur ohnehin nicht Teil des Projektumfangs war. Der Nachbau der komplexen Infrastruktur der CISTEC AG wurde bereits zu Beginn der Masterarbeit als out of scope identifiziert und nicht als Ziel aufgenommen.

5.1.2 Zusammenarbeit im Projektteam

Zur internen Zusammenarbeit wurde von Beginn an ein wöchentliches Projekttreffen etabliert. Im Verlauf des Projekts zeigte sich jedoch, dass zur gemeinsamen Bearbeitung technischer Fragestellungen ein zusätzlicher Austausch notwendig war. Ab Projektwoche 13 (03.07.2025) wurde daher ein zweites wöchentliches Zeitfenster eingeführt, das gezielt für Pair Programming und enge Abstimmungen im Entwicklerteam genutzt wurde. Diese Anpassung stellte eine Abweichung von der ursprünglich in Kapitel 3.1 beschriebenen Methodik dar (vgl. Abb. 3: Kollaboration und Kommunikationsablauf im Entwicklungsprozess), erwies sich jedoch als sinnvoll, um die Effizienz zu steigern und technische Herausforderungen im Team besser zu bewältigen.

5.1.3 Arbeitsaufwände und Eigenverantwortung

Die Arbeitsaufwände wurden pro Teammitglied kontinuierlich erfasst:

Name	**Stunden**
Jvan Faddah	452
Guillaume Fricker	416
Benjamin Thormann	483

Tab. 12: Arbeitsaufwände (zusammengefasst)

Das Team arbeitete oft zusammen und konnte die Arbeitslast gut aufteilen. Aufgrund der eingeschränkten Möglichkeiten zur fachlichen Abnahme seitens CISTEC AG arbeitete das Projektteam in hohem Masse eigenverantwortlich. Regelmässige Abstimmungen und Reviews fanden vor allem intern statt, während externe Reviews auf das notwendige Minimum reduziert wurden. Im Laufe des Projekts musste bei den ersten groben Zeitengpässen gezielt die Zusammenarbeit intensiviert werden. Essenziell war auch der reduzierte Fokus auf eine externe Abnahme, bedingt durch die organisatorischen und infrastrukturellen Rahmenbedingungen der CISTEC AG. Dadurch entwickelte sich ein Arbeitsmodus, der stark auf Eigenverantwortung und Teamkoordination setzte und dennoch eine kontinuierliche Zielerreichung sicherstellen konnte.

5.1.4 Risikoanalyse

Im Rahmen der Ergebnisse wird aufgezeigt, welche der erwarteten Risiken tatsächlich eingetreten sind, wie sie sich auf den Projektverlauf ausgewirkt haben und in welchem Mass die geplanten Gegenmassnahmen wirksam waren. Dabei zeigt sich, dass sowohl technische als auch organisatorische Risiken in unterschiedlicher Stärke realisiert wurden. Darüber hinaus traten auch unvorhergesehene Risiken auf, die in der ursprünglichen Planung nicht berücksichtigt wurden. Die nachfolgenden Kapitel dokumentieren diese Befunde und leiten daraus zentrale Learnings aus dem Projekt ab.

5.1.5 Domäne und Abnahmekriterien (ADR0007)

Ein zentrales Risiko bestand darin, dass der Erfolg des Projekts anfangs stark an klinische Fachkriterien gekoppelt war. Diese Vorgabe führte zu einer Einschränkung der Entwicklung, da die Agentenlogik zu nah an bestehenden KISIM-Strukturen und klinischen Use-Cases orientiert war.

Mit der Entscheidung ADR0007 wurde dieses Risiko entsprechend adressiert. Die Vorgabe, klinische Agenten nachzubauen, wurde aufgehoben und durch abstrahierte, alltagsnahe Use-Cases ersetzt (z. B. Marathon-Agent anstelle eines OAT-Agenten). Dadurch ergaben sich mehrere Vorteile:

- Flexibilität bei der technischen Umsetzung und beim Datenmodell
- Nachvollziehbarkeit für Entwickler:innen ohne klinischen Hintergrund
- Raum für Innovation und kreative Lösungsansätze anstelle von Aufrechterhaltung von Legacy-Strukturen

In Zukunft muss mit einem erhöhten Aufwand bei der späteren Integration auf echte klinische Systeme gerechnet werden, da diese nicht 1:1 im Prototyp abgebildet wurden. Insgesamt war dieser Schritt jedoch entscheidend, um das Risiko einer auch lediglich fachliche Aspekte begrenzte Abnahme zu vermeiden und den Erfolg des Projekts auf technischer Ebene zu sichern, was wesentlich zum Erfolg des Projekts beitrug.

5.1.6 Domänenspezifischer vs. generischer Framework-Prototyp

Ein weiteres Risiko betraf die Balance zwischen domänenspezifischer und generischer Ausrichtung des Frameworks. Während ein enger klinischer Fokus zwar eine hohe Relevanz

und Attraktivität für die CISTEC AG darstellt, führte dieser zu hohen Abhängigkeiten zu bestehenden Legacy-Datenstrukturen und erschwerte die Übertragbarkeit von wichtigen Anforderungen. Die Lösung bestand darin, eine eigene Entwicklungsdomäne zu definieren, die als Basis für generische Agenten diente. So konnten Risiken wie mangelnde Testdatenqualität oder zu starke Kopplung an bestehende KISIM-Module umgangen werden. Gleichzeitig blieb durch die konsequente Einhaltung der technischen Requirements die Möglichkeit erhalten, spätere klinische Erweiterungen zu integrieren.

5.1.7 Technischer Grossausfall

Ein unerwartetes Risiko entstand durch eine Cyberattacke auf die Systeme der CISTEC AG. Diese Attacke verzögerte die Bereitstellung von Testdaten und die Flexibilität des Stakeholders erheblich. Dieses Risiko konnte nicht durch das Projektteam beeinflusst werden, hatte aber einen hohen Einfluss auf den Projektplan.
Das Team reagierte, indem es einen eigenen Testrahmen definierte, eigene Use Cases und deren Entwicklung unabhängig von Stakeholder-Daten vorantrieb.
Diese Massnahmen ermöglichten es, trotz des Ausfalls Fortschritte zu erzielen und die Entwicklungsziele nach bestem Gewissen einzuhalten.

5.1.8 Auswertung Risikomatrix

In diesem Abschnitt wird die Auswertung der Risikoanalyse zusammengefasst. Die Tabelle fasst die während des Projektverlaufs relevanten Risiken, ihre Bewertung, die geplanten Gegenmassnahmen sowie die tatsächlichen Auswirkungen kurz zusammen.

Kürzel	Einschätzung	Beschreibung und Massnahme	Reflexion
A	6	Testdaten verzögert, Cyberattacke → Frühzeitige Bereitstellung und Validierung	Eingetreten, externe Ursache, hoher Einfluss
B	6	KIS-Testdaten unbrauchbar → Eigene Domäne definieren	Eingetreten, Legacy-System unbrauchbar, hoher Zusatzaufwand
C	4	Stakeholderkontakt eingeschränkt → Gezielte Abnahmen, Fokus auf Selbsteinschätzung	Teilweise eingeschränkt, ausreichend durch gezielten Austausch
D	4	Entwicklung gerät ins Stocken → Definierte Prozesse und ADR-Dokumentation	Teilweise eingetreten, ADRs nachgezogen, Prozess iterativ erweitert
E	3	Ungleichmässige Arbeitslast → Rollenverteilung, Abstimmungen, Code Sessions	Eingetreten, nicht systematisch adressiert, moderater Einfluss
F	6	Fachspezifische Domäne → Abstraktion auf Framework, eigene Agenten-Definitionen	Eingetreten, aber erfolgreich entschärft, Massnahme wirksam

Tab. 13: Auswertung finaler Risiken (zusammengefasst)

5.1.9 Lessons Learned

Die Risikoanalyse hat verdeutlicht, dass die grössten Risiken nicht wie zuerst angenommen hauptsächlich in der technischen Natur lagen, sondern auch in Organisation, Abhängigkeiten und Scope-Definition. Die wichtigsten Erkenntnisse sind:

Abstraktion statt Fachlogik: Durch die Scope-Definition in ADR0007 wurde die Entwicklung von klinisch engstirnigen Abnahmekriterien gelöst, was Innovationsspielraum eröffnete.

Eigene Domäne als Basis: Der Aufbau einer unabhängigen Domäne erwies sich als effektive Massnahme, um externe Risiken wie Datenqualität und die Cyberattacke abzufedern.

Resilienz durch Flexibilität: Unerwartete Ausfälle und Stakeholder-Abwesenheiten konnten durch pragmatische Workarounds (regelmässige Syncs, selbstdefinierte Agenten und deren Use-Cases, Feature-Freeze) abgefangen werden. Die Auswertung zeigt somit, dass das Projektergebnis weniger von einer risikofreien Umsetzung abhing, sondern von der Fähigkeit, auf Unsicherheiten adaptiv zu reagieren und zentrale Entscheidungen transparent zu dokumentieren.

5.2 Architektur

5.2.1 Architekturbeschreibung

Das entwickelte Framework basiert auf einem modularen Monorepo, in dem alle Komponenten (Agent-Core, zentrale Library und einzelne Agenten) in einer konsistenten Projektstruktur zusammengeführt werden. Dieses Vorgehen ermöglichte eine einheitliche Verwaltung des Datenflusses zwischen Core und Agenten sowie eine gemeinsame Nutzung der zentralen Datenbank. Darüber hinaus vereinfacht es Refactorings und erlaubt eine flexible Weiterentwicklung, wie sie im Kontext der CISTEC AG erforderlich ist.

Die Entscheidung für ein Monorepo ist zugleich eine strukturierte Zusammenfassung der gesamten Agentenlandschaft, um so die Developer Experience signifikant zu steigern, da so alle Änderungen an Agenten an einem zentralen Ort vorgenommen werden können.

5.2.2 Microservices als Architekturansatz

Die Struktur der Agenten wird als Microservices betrieben. Der Agent-Core stellt eine zentrale Schnittstelle zur Datenbank bereit, während die Agenten jeweils eigenständige Workloads (Microservices als CronJobs) darstellen. Zwischen den Agenten existieren keine direkten Abhängigkeiten und die gesamte Kommunikation läuft ausschliesslich über den Core. Die Trennung der Agenten in Microservices adressiert mehrere Ziele:

Modularität und Kohäsion: Jeder Agent kapselt seine eigene Business-Logik, ohne die Logik oder Kommunikation anderer Agenten zu kennen.

Lose Kopplung: Änderungen im Core, in der zentralen Library, oder in einem Agenten können unabhängig voneinander erfolgen.

Skalierbarkeit: Core und Agenten können horizontal skaliert werden, je nach Bedarf und Ressourcenauslastung.

Resilienz: Ein Ausfall eines Agenten beeinträchtigt nicht die Funktionsfähigkeit der restlichen Plattform.

Damit ist auch die nichtfunktionale Anforderung NFA_REQ6 nach einer skalierbaren und

modularen Architektur erfüllt.

5.2.3 Kommunikationsmuster

Die Kommunikation zwischen Core und Agenten folgt einem RPC-Pattern auf Basis von NATS, welches über ein publish-subscribe Modell kommuniziert (NATS.io, 2025). Agenten senden ihre Anfragen an den Core, der diese beantwortet. Der Core selbst initiiert keine Kommunikation mit Agenten. Weitere Kommunikationspfade sind wie folgt definiert:

Core und Datenbank (PostgreSQL): Klassischer Read/Write-Zugriff über TypeORM.

Logging: Promtail sammelt Container-Logs und pusht sie an Loki. Grafana greift auf Loki als Datasource zu (siehe Kapitel 5.4).

Obwohl eine Event-Driven-Architektur naheliegt, wurde bewusst auf eine vollständige Message Queue verzichtet. Die aktuelle Lösung deckt die benötigten Muster ab und reduziert Komplexität. Sollte sich beim Stakeholder später der Wunsch nach einer Kommunikation mittels Message Queue äussern, kann dies nahtlos in NestJS und NATS integriert werden, da NATS Message Queues unterstützt. Gleichzeitig ermöglicht die Wahl von NATS eine spätere Erweiterung, beispielsweise durch JetStream für Persistenz oder Replay (NATS.io, 2025).

5.2.4 Evolution der Architekturentscheidungen

Die Architektur entwickelte sich iterativ weiter. Ursprünglich war ein REST-basiertes Kommunikationsmodell vorgesehen, das durch Versionierung abgesichert werden sollte. Diese Variante erwies sich allerdings als unzureichend, da das NestJS-Framework keine native Unterstützung für REST im Microservice-Modul bietet.

Im Zuge der Evaluation wurde daher der Wechsel zu NATS vollzogen. Dieser Entscheid überholte einige der ursprünglichen Annahmen, insbesondere in Bezug auf Idempotenz und API-Versionierung. Dennoch blieben die grundlegenden Vorteile der Microservice-Architektur, etwa klare Verantwortlichkeitstrennung und unabhängige Skalierbarkeit, bestehen.

Im Zwischenreview wurde mit dem Walking Skeleton (ADR0003) ein erster lauffähiger Prototyp präsentiert. Die späteren Architekturentscheidungen (ADR0009 Microservices, ADR0010 Kommunikation via NATS, ADR0011 Monitoring & Logging, ADR0012 Standardisierung der Generierungsprozesses) bauten darauf auf und erweiterten das Konzept kontinuierlich weiter.

5.2.5 Architekturprinzipien und Patterns

Die Architektur folgt etablierten Prinzipien und Patterns:

Trennung der Verantwortlichkeiten: Der Core übernimmt Datenzugriffe, Validierung und Metriken und dient als zentraler Pool für Abfragen. Agenten implementieren ausschliesslich fachliche Logik und sind somit kleine, isolierte Tasks, die unabhängig voneinander skalierbar und austauschbar sind.

Wiederverwendbarkeit: Gemeinsame Bausteine (DTOs, Message-Pattern, NATS-Client, Logging) werden als Bausteine bereitgestellt. Dadurch wird vermieden, dass der Core zum unübersichtlichen Monolithen anwächst. Kommunikation über NATS entkoppelt Agents und Core zeitlich und organisatorisch.

Erweiterbarkeit: Neue Agenten können integriert werden, ohne bestehende Komponenten

anzupassen.
Resilienz: Fällt ein Agent aus, bleiben die anderen lauffähig. Der Core bleibt weiterhin stabil. Bei einem Absturz des Cores warten die Agenten auf ihre nächste Ausführung.
Ein zentrales Pattern ist das Repository Pattern, das durch NestJS und TypeORM vorgegeben wird. Es trennt Datenzugriffe von der Geschäftslogik und stellt über DTOs und Zod-Schemas sicher, dass nur gültige Daten verarbeitet werden.

5.2.6 Trade-Offs

Die gewählte Architektur bringt klare Vorteile, erfordert aber auch bewusste Kompromisse:

Vorteile	● Hohe Skalierbarkeit und Flexibilität. ● Saubere Trennung von Logik, Messaging und Persistenz. ● Verbesserte Wartbarkeit durch modulare Services. ● Vereinfachtes projektweites Refactoring (z.B. Umstrukturierung oder Dependency-Updates)
Nachteile	● Zusätzliche Komplexität im Betrieb (was jedoch durch die hohe Skalierbarkeit ausgehebelt wird) ● Strikte Regeln für Entwickler:innen und (z.B. Modulgrenzen, Schema-Validierung, fixer Prozess in der Datenverarbeitung) erhöht die Lernkurve marginal. ● Die Codebasis hat sehr klare Vorstellungen darüber, wie Dinge gemacht werden sollten. Dies kann bei neuen Anforderungen zu Refactorings führen.

Tab. 14: Architektur Vor- und Nachteile

Trotz der Nachteile ist die Architektur ein nachhaltiger Entscheid, der sowohl die kurzfristigen Projektziele als auch die langfristige Erweiterbarkeit sicherstellt.
Das Framework ermöglicht durch die Modulare Struktur auch Integrationen mit anderen Systemen, wie zum Beispiel der KISIM-Datenbank, die ins TypeORM integriert werden kann. Dadurch holt Framework Ist-Zustand CISTEC AG ab und setzt keine Schranken, damit CISTEC AG auch andere Systeme ausprobieren, migrieren sowie andere KIS-Anbieterinnen das Framework nutzen können.

5.2.7 Systemüberblick

Das entwickelte System besteht aus einem zentralen Agent-Core und mehreren Agenten, die jeweils eine abgegrenzte fachliche Aufgabe erfüllen (z. B. NoShow, Marathon, Grenzkontrolle). Der Agent-Core abstrahiert die Datenbank und fungiert als zentraler Service, welcher von allen Agenten benötigt wird. Die Kommunikation erfolgt über NATS, wodurch die Agenten entkoppelt und unabhängig voneinander betrieben werden können.

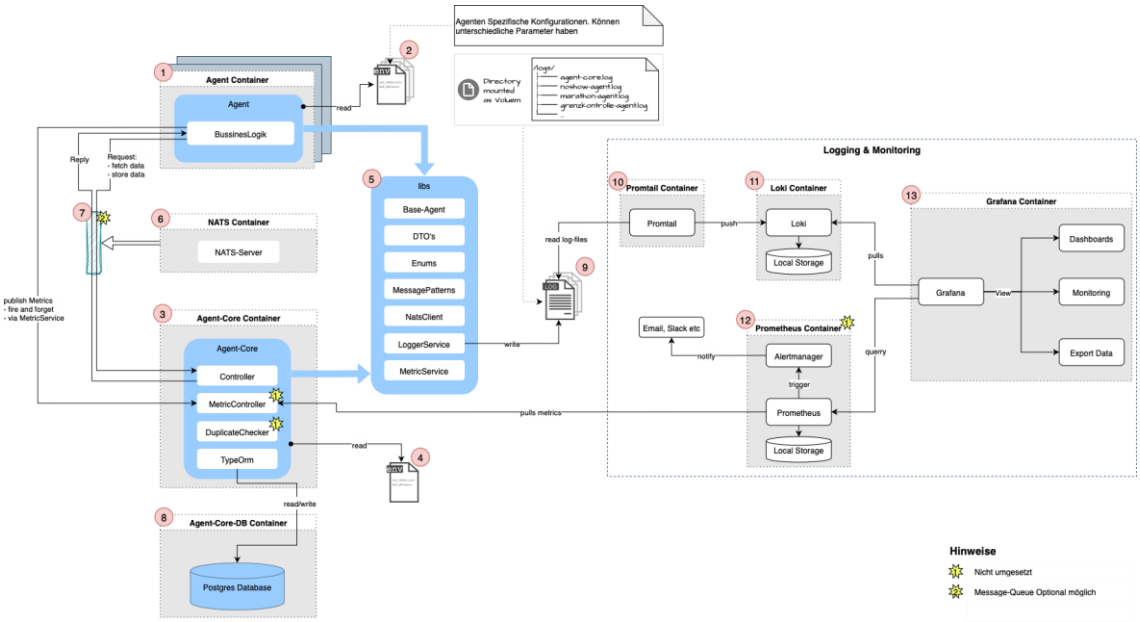


Abb. 14: Systemüberblick

Um den Systemüberblick im Detail nachzuvollziehen, wird nun auf die nummerierten Stationen des Ablaufs im System Agent-Framework sowie dem darin enthaltenen Subsystem Logging & Monitoring (Schritte 10 bis 13) eingegangen (siehe Abb. 14: Systemüberblick):

1. Der **Agent Container** führt die Businesslogik aus. Dementsprechend gibt es für jede Fachdomäne (Noshow, Marathon, Grenzkontrolle) einen eigenständigen Service. Damit sind die Agenten fachlich sauber voneinander getrennt. Da die Änderungen an einem Agenten nur diesen und keine weiteren Agenten betreffen, können Agent unabhängig voneinander skaliert werden. Die Anfragen an den Core stellt der Agent im Request-Reply-Muster über NATS.

2. Die **Environment-Variablen jedes einzelnen Agenten** beinhalten die Konfigurationswerte zur Parametrisierung des Laufzeitverhaltens des jeweiligen Agenten. Dadurch werden Umkonfigurationen pro Umgebungen vereinfacht und ein einheitliches, reproduzierbares Deployment ermöglicht.

3. Der **Agent-Core Container** ist der zentrale Backend-Service, welcher als Single Source of Truth die Datenhaltung und die Datenbank abstrahiert: Er antwortet auf Anfragen der Agenten und führt die Datenzugriffe aus. Durch diese saubere Entkopplung bleiben die eigentlichen Agenten schlank und müssen keine Datenbank-Details kennen.

4. Die **Environment-Variablen des Agent-Core beinhalten** die Konfigurationswerte des Agent-Core (Datenbank-Zugriff, NATS-Einstellungen). Dadurch werden Umkonfigurationen pro Umgebungen vereinfacht und ein einheitliches, reproduzierbares Deployment ermöglicht.

5. Die **Libraries** sind wiederverwendbare, gemeinsame Code-Bausteine für Core und Agenten und dienen zur Vereinheitlichung von Typen und DTOs, Messaging-Patterns, Infrastruktur Zugriffen und des Loggings. Dadurch wird die benötigte Boilerplate erheblich gesenkt, eine konsistente Struktur in allen Agenten erreicht und potentielle Fehlerquellen reduziert.

6. Der **NATS Container** ist ein leichtgewichtiges, schnelles Messaging-System und fungiert als Transportschicht zwischen Agenten und Core: NATS unterstützt Subjects/Topics, Request-Reply und optionale Queue-Groups, hat eine sehr geringe Latenz, ist einfach zu skalieren und ist lose gekoppelt. NATS ist also für die Kommunikation zwischen Microservices und damit für

damit für den Agenten-Framework sehr gut geeignet

7. Die **NATS Queue-Groups** ermöglichen die Nachrichtenverteilung als Lastverteilung in Pub/Sub-Szenarien: hierbei hören mehrere Worker auf das gleiche Subject, jedoch verarbeitet nur ein Worker aus der Gruppe die Nachricht. Das System ist auf die Nutzung der optionalen Erweiterung **JetStream** vorbereitet, sodass in Zukunft eine horizontale Skalierung und erweiterte Anforderungen an die Persistenz und die garantierte Zustellung leicht umgesetzt werden können. JetStream ermöglicht die Nutzung folgender Features: Persistente Nachrichten, eine garantierte Zustellung (at-least-once oder exactly-once), eine historische Wiedergabe (replay) sowie ein verteilter Schlüssel/Wertspeicher (z. B. zur Konfiguration oder für Zustände).

8. Der **Datenbank Container (Postgres)** ist die relationale Persistenzschicht, auf welche ausschliesslich der Agenten-Core via TypeORM zugreift. Es ist also eine klare Zugriffskontrolle vorhanden. Die Datenbank selbst bildet die zentrale, konsistente Datenhaltung für fachliche Daten bzw. Leistungen.

9. Die **Applikations-Logdateien** sind die Grundlage für die Fehlersuche und Datenanalyse. Sie existieren für den Core und jeden einzelnen Agenten. Es werden Betriebszustände, Fehler sowie fachliche Ereignisse protokolliert.

10. Beginn Subsystem Logging & Monitoring: Der **Promtail Container** liest und sammelt die Logs aus den Log-Dateien, versieht sie mit Labels und sendet diese an den Loki-Container. Das Ergebnis ist eine einheitliche Log-Pipeline mit minimalem Konfigurationsaufwand pro Service.

11. Der **Loki Container** speichert und indexiert die Logs über Labels, um so eine Log-Aufbewahrung mit schneller Suche und Filterung zu realisieren.

12. Der **Prometheus Container (inkl. Alert Manager)** kann Metriken vom Core-Service ziehen, Regeln auswerten und Alerts versenden (E-Mail, Slack etc.). Dies wurde nicht umgesetzt, würde jedoch die Sichtbarkeit von und Benachrichtigung bei Fehlern und Ereignissen verbessern.

13. Ende Subsystem Logging & Monitoring: Der **Grafana Container** wird als zentrale Visualisierungs- und Dashboard-Plattform für das Binden von Prometheus (Metriken) und Loki (Logs) genutzt. Die konfigurierbare, einheitliche Monitoring-Oberfläche (Metriken / Logs) vereinfacht die Datenanalyse sowie den Datenexport.

5.2.7.1 Bausteinsicht

Im Paketdiagramm ist die modulare Architektur des Framework-Prototypen gut (Abb. 15 Paketdiagramm) zu erkennen: Die Agenten (agent) und der Agent-Core (core) sind voneinander getrennt und es besteht keine direkte Abhängigkeit. Beide greifen auf eine gemeinsame Bibliothek (libs) zu: Durch diese Wiederverwendung von Code-Bausteinen wird die Entwicklung und Anpassung von Agenten nochmals stark vereinfacht. Zum Beispiel wird ein abstrakter Basis-Agent bereitgestellt (libs/agent in Abb. 15 Paketdiagramm), welcher als Grundlage für neue Agenten dient.

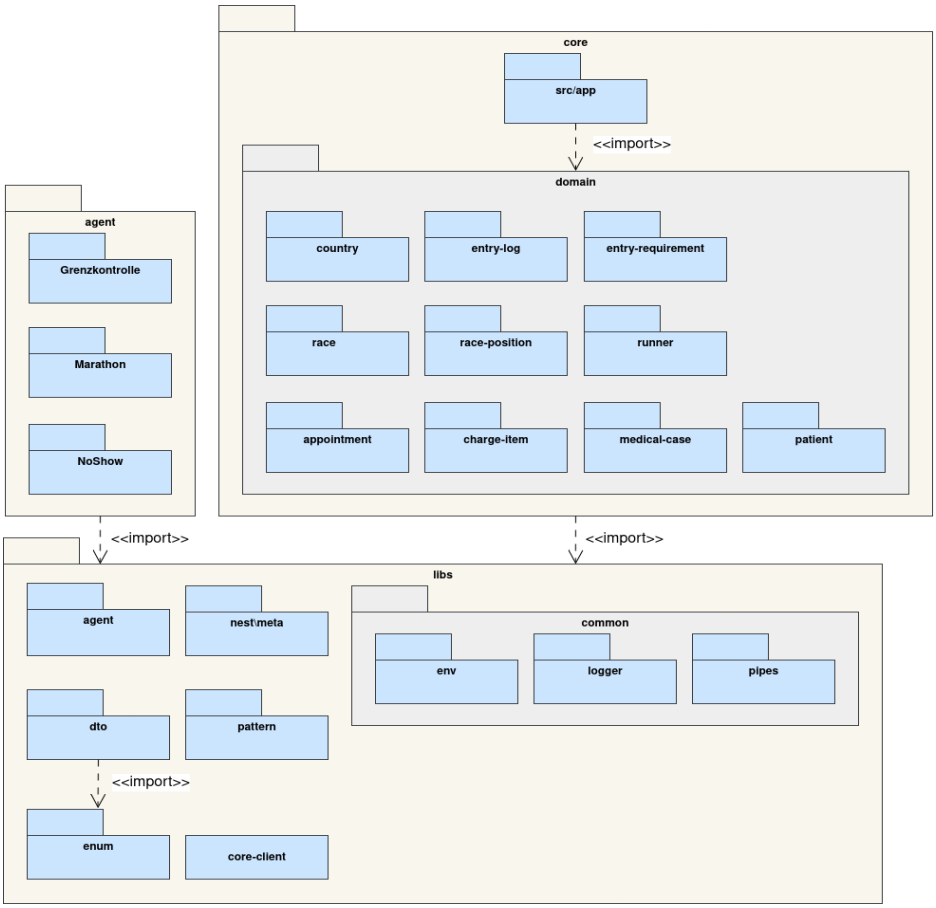


Abb. 15: Paketdiagramm

5.2.8 Tech-Stack

Der Tech-Stack wurde wie in Kapitel 4.7.1 umgesetzt und hat sich als zufriedenstellend und zielführend herausgestellt. Durch die Wahl von TypeScript ist NFA_REQ2 umgesetzt. Ausserdem wurde durch den Einsatz von pnpm workspaces ein kohäsives monorepo mit unabhängigen Modulen, wie in ADR0013 beschrieben, erreicht. Dies hat die Struktur des Frameworks weiter gefestigt und aufgewertet. Desweiteren ist der Tech-Stack zusammengestellt aus:

- **TypeScript**: primäre Programmiersprache (moderner CISTEC-Standard)
- **NestJS**: Basis-Framework (modular, strukturiert, skalierbar, enterprise-ready)
- **TypeORM**: Entity-Mapping (Datenbankabstraktion, gut lesbar)
- **Zod**: Schema-Validierung (Typensicherheit zur Laufzeit, sicher und deklarativ).
- **Winston**: Logging (Konfigurierbarer Logger, super für CI/CD)
- **Grafana + Promtail**: Monitoring (Standard in der Webentwicklung)
- **pnpm**: Package-Manager mit Monorepo Workspaces (schnell, platzsparend, konsistent, einfach, flexibel)

5.3 Systemverteilung und Deployment

Die entwickelte Lösung basiert auf einer GitOps-gesteuerten, verteilten Architektur und addressiert die Use-Cases UC05 (Kofiguration), UC07 (Agentenlauf manuell auslösen), UC10 (Generierungsintervall konfigurieren) und somit die Anforderungen FA_REQ3, FA_REQ4 und NFA_REQ6.

Alle relevanten Komponenten (Core, Agents, Infrastruktur wie NATS/Postgres sowie Monitoring) werden zentral im Git-Repository versioniert und über Fleet automatisch in das Kubernetes-Cluster ausgerollt. Die Aufteilung im Cluster durch die Namespaces sorgt dafür, dass man schnell erkennt, was läuft und was nicht.
Im Prototyp wurde die Verteilung der Systemkomponenten vereinheitlicht und auf zwei zentralen Helm-Charts gebaut: eines für den Agent-**Core**, eines für alle **Agenten**. Der Agent-Core wird über sein Chart als Deployment ausgerollt, während das Agenten-Chart CronJobs erzeugt, bei dem der Intervall angegeben werden kann pro Job. Ein neuer Agent wird lediglich über einen zusätzlichen Block der Datei “gitops/agent/chart/values.yaml” eingebunden – das Chart selbst bleibt unverändert.
Die Infrastrukturkomponenten sind als eigene Fleet-Pakete implementiert. PostgreSQL, NATS sowie die Monitoring-Dienste laufen in separaten Namespaces und werden über `dependsOn`-Beziehungen in Fleet orchestriert, um die Reihenfolge der Inbetriebnahme zu steuern.
Für die Installation ist ein Installationsscript im Anhang angefügt sowie für das manuelle Starten der Cronjobs ausserhalb des konfigurierten Intervalls.
Für die lokale Entwicklung steht ein Docker-Compose-Setup bereit, das das Cluster-Setup in vereinfachter Form spiegelt. Entwickler:innen können so End-to-End-Tests mit Core, Agents, Datenbank und Logging-Pipeline auch ohne Kubernetes durchführen.

5.3.1 Verteilungssicht

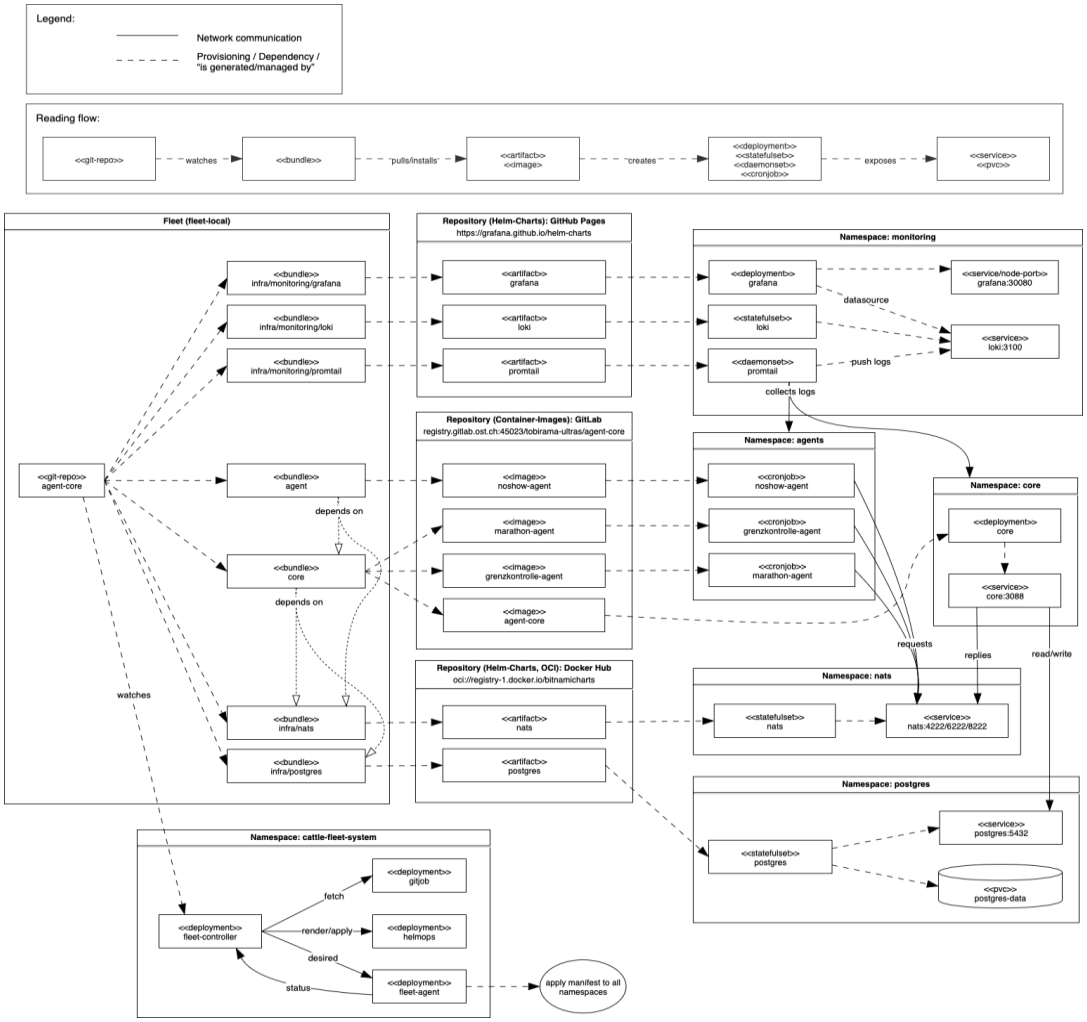


Abb. 16: Verteilungsdiagramm

Abbildung 16: Verteilungsdiagramm stellt die Verteilung dar: Von links nach rechts ist der Ablauf zu erkennen: ausgehend von den Git-Repo-Definitionen (z. B. fleet-yaml) enstehen Bundles, welche die Charts aus den Repositories installieren oder Container-Images von der GitLab-Registry gezogen werden. Daraus resultieren in den Ziel-Namespaces die ausgerollten Dienste als Deployment, Cronjob, Service etc.
Im unteren linken Bereich ist die Fleet-Control-Plane (Manager) u. a. für Aufgaben wie Überwachung des Git-Repos und automatisches Ausrollen. Es gibt folgende Runtime-Namespaces:
Core: beherbergt den zentralen Core-Service. Als Deployment dauerhaft verfügbar; persistiert in Postgres und kommuniziert über NATS.
Agents: enthält die spezialisierten Agents (z. B. `noshow`, `marathon`, `grenzkontrolle`), die periodisch als CronJobs laufen und nach dem Job wieder terminieren.
Nats: stellt den Message Broker bereit.
Postgres: verwaltet den Datenbank-State; persistent über PVC.
Monitoring: stellt Observability über Loki, Promtail und Grafana bereit.

5.3.2 GitOps & Fleet

GitOps gibt uns die Möglichkeit, Änderungen an Verteilung, Konfiguration und Charts, bequem und direkt im Git-Repo des Quellcodes zu handhaben. Dies ermöglicht es, jeden ausgerollten Zustand zu reproduzieren. Abhängigkeiten zwischen Workloads lassen sich bequem steuern, etwa um den Agent-Core erst zu starten, wenn Postgres und NATS verfügbar sind. Das automatische Ausrollen bei neuen Commits erhöht die Developer Experience und hilft, stabile und nachvollziehbare Deployments über verschiedenen Systemen auszurollen.
Bei Fleet unterscheiden wir zwei technische Aspekte: Fleet-Control-Plane (Manager) und Runtime. Die Runtime-Services laufen in eigenen Namespace gemäss der Definition wie core, agents, nats, postgres und monitoring. Diese sind von uns erstellte Namespaces sowie die darin laufenden Dienste. Die Control-Plane wird installiert und übernimmt die Git-Überwachung sowie das automatische Ausrollen. Fleet setzt dafür mehrere Dienste ein, die in der Control-Plane laufen: Der *gitjob-Controller* überwacht das Git-Repo und reagiert auf neue Commits. Der *fleet-/ & helmops-Controller* rendert die Manifeste zu Bundles und berücksichtigt Abhängigkeiten. Der fleet-Agent übernimmt die Aufgabe, Manifeste auf Ziel-Namespaces anzuwenden. Über Pfadangaben wird gesteuert, welche Artefakte gebaut und ausgerollt werden (vgl. Abb. 16: Verteilungsdiagramm).

5.4 Monitoring und Logging

Um Fehler schnell zu finden und Abläufe nachzuverfolgen, benötigt man einen zentralen Blick auf alle Logs der Dienste – vom Agent-Core bis zu den einzelnen Agenten. Unsere Lösung ist bewusst leichtgewichtig gehalten und funktioniert auch dann zuverlässig, wenn ein Agent nur kurz als Cronjob läuft. Das Ergebnis: Alle relevanten Ereignisse landen an einem Ort, sind einfach durchsuchbar und lassen sich einem konkreten Lauf zuordnen.
Unsere Monitoring- und Logging-Lösung adressiert die technischen Use-Cases UC09 (fehlerhafte Leistungsgenerierung protokollieren), UC14 (Logging einsehen) und UC15 (Fehlerlogs einsehen) und stützt damit die Anforderungen FA_REQ3, FA_REQ10 sowie NFA_REQ5. Weiter wird der fachlich Use-Case UC08 (Agent seit X Tagen keine Leistung mehr erzeugt) mit FA_REQ8 behandelt.

5.4.1 Datenfluss und umgesetzte Architektur

Jeder Dienst schreibt Meldungen mit dem Logger in eine eigene Datei (z. B. agent-core.log oder noshow-agent.log). Diese Einträge sind als JSON strukturiert und werden mit einem Hilfsprogramm namens Promtail fortlaufend mit Zusatzangaben versehen. Unter anderem der Service-Name und das Log-Level. Anschliessend werden die Daten an Loki geschickt. Loki ist ein Dienst für die zentrale Ablage der Logs. Dort werden sie gespeichert und so indexiert, dass man sie später wiederfinden kann. Die Auswertung erfolgt mithilfe von Grafana, das eine bequeme und schnelle Durchsuchung der Logs ermöglicht („Zeige mir alle Fehler des Agent-Core der letzten zwei Stunden.“) (UC09, UC15). Darüber hinaus können einfache Auswertungen direkt aus den Logdaten gebildet werden (z. B. „Wie viele Warnungen gab es pro Stunde?“) – siehe Abb. 14: Systemüberblick.

5.4.2 Verwendung

Sowohl im Agent-Core als auch in den Agenten wird das Logger-Modul importiert und per Dependency Injection in den Konstruktor übergeben. Danach kann direkt geloggt werden, beispielsweise mit logger.info(...), logger.warn(...) oder logger.error(...). In Grafana erscheinen die Logs der Dienste unmittelbar und können direkt durchsucht werden, ohne manuelle Konfiguration – alles ist bereits bereitgestellt (UC14).

5.4.3 Bekannte Grenzen

Aus Zeitgründen wurde eine automatische Überwachung über Metriken und eingerichtete Alarmmeldungen nicht umgesetzt. Falls etwas aus dem Ruder läuft, wird dies nur über eine Auswertung der Logs erkennbar – nicht über Metrik-Dashboards oder Benachrichtigungen (UC08).

5.5 Qualitätssicherung

Das Shift Left Prinzip wurde im Projektverlauf über eine durchgehende, automatische CI/CD Pipeline umgesetzt. Um die E2E Test sowie eine sinnvolle Verteilung sicherzustellen, wurde Docker eingesetzt. Die Test-Stage sichert die Qualität noch weiter über einen Pre-Commit-Hook und Unit-Tests. Zur Laufzeit wird über Zod sichergestellt, dass die Daten typsicher sind. Die Qualitätsanforderungen NFA_REQ3, NFA_REQ4 wurden somit erfolgreich umgesetzt. Die Schema Validierung trägt zudem zur Realisierung von NFA_REQ9 bei.

5.5.1 Testinfrastruktur und Hilfsmittel

Für die Testinfrastruktur wurden verschiedene Hilfsmittel als Bausteine realisiert:
Infra-Builder für E2E-Tests: Startet Postgres, NATS und den Agent-Core in einem isolierten Netzwerk als Testcontainer. Per Builder-Pattern kann man frei wählen, welche Dienste benötigt werden, und kann leicht erweitert werden. Bei Bedarf lassen sich zusätzlich die Container-Logs aktivieren – das hilft, die Ursache bei fehlgeschlagenen Tests schnell zu finden.
DataSource-Factory für E2E-Tests: Initialisiert eine TypeORM-Datenquelle mit den übergebenen Domänenentitäten und stellt eine Reset-Funktion bereit, um die Datenbank in Test-Hooks (Setup/Teardown) sauber zurückzusetzen. Damit lassen sich Arrange- und Assert-Phasen realistisch und flexibel aufsetzen, und die Tests bleiben sauber isoliert.
AgentTest-Factory für E2E-Tests: Erstellt einen Agenten als Nest-App und vereint den Infra-Builder und DataSource-Factory in einem. Damit lässt sich mit sehr wenig Zeilen Code ein gesamter E2E-Test aufziehen.
TestLogger-Factory für Unit- und E2E-Tests: Stellt einen Winston-Logger bereit, mit dem sich Logs der Ablauflogik gezielt “ausspionieren“ oder für die Konsole aktivieren bzw. deaktivieren lassen. Dies erleichtert die Verfolgung von Prozessen beim Schreiben von Tests sowie bei der Fehlersuche erheblich – insbesondere, weil mehrere Dienste gleichzeitig ausgeführt werden.
Diese Bausteine senken den Setup-Aufwand, vereinheitlichen den Testaufbau ohne Flexibilität zu verlieren – so werden auch komplexere E2E-Tests gut handhabbar.

5.5.2 Metriken

Wir haben folgende Metriken protokolliert:

- **Anzahl Tests**: 43
- **Testabdeckung (Coverage)**: 85.86% (Stmts), 68.9% (Branch)
- **SonarQube Quality-Gate**: 18 Open issues, 6 Accepted issues, 12 Security Hotspots (dies liegt daran, dass wir die Docker-Images nicht Infra-spezifisch abgesichert haben)
- **Umsetzungszeit Template Agent:** 2 Stunden (Bei CISTEC AG normalerweise 2 Tage für komplettes Setup)
- **Lines of Code für einen simplen Agenten:** 48 Zeilen für den Service, 283 Zeilen für den gesamten Agenten inkl. Domain, DTOs und Entities.

Die Interpretation von Metriken ist keine exakte Wissenschaft, jedoch ist hier klar sichtbar, dass das Framework eine hohe Testabdeckung erzielt hat und den Arbeitsaufwand sowie die Umsetzungszeit für einen simplen Agenten massiv verringert hat. Das SonarQube Quality-Gate schlägt 24 bestätigte Issues vor (grösstenteils TODO oder FIXME Kommentare) und fand 12 Sicherheits-Vulnerabilitäten. Diese liegen ausschliesslich an den Docker-Images, welche die Dependencies direkt in das Image kopieren, sowie nicht ausreichend gegen Role Based Access Control (RBAC) geschützt sind. Dies ist in unserem definierten Scope und in einem nicht produktiven Setting vertretbar und kann bei der Integration in eine richtige Firmeninfrastruktur gepatcht werden.

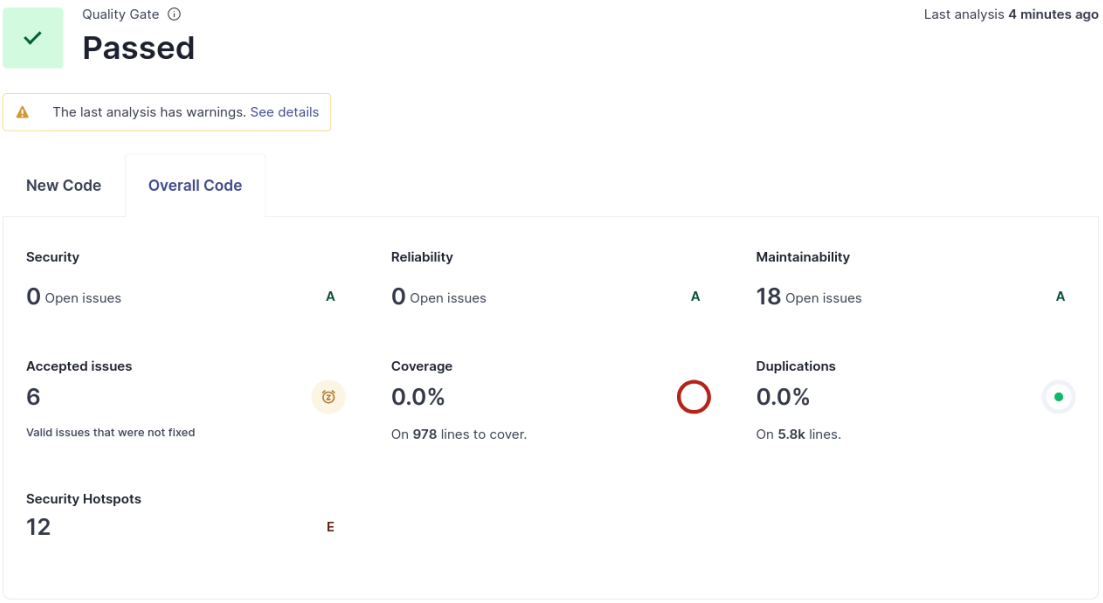


Abb. 17: SonarQube Quality Gate

6 Diskussion

6.1 Zielerreichung

Ausgehend von der formulierten Problemstellung im Kapitel 1.1 und 4.1 – einer fragmentierten, schwer wartbaren und kaum skalierbaren Agentenlandschaft – konnte im Rahmen dieser Arbeit ein standardisiertes, modulares Framework entwickelt werden. Der rote Faden von der Analyse des Ist-Zustands über die Definition von Anforderungen bis hin zu Architekturentscheidungen und Prototyping wurde nach bestem Wissen und Gewissen verfolgt.

Die wesentlichen Pain Points aus Kapitel 1.1 wurden adressiert:

Standardisierung und Wiederverwendbarkeit: Klare Guidelines, konsistente Implementationsmuster (Prozess mit processItem, update, create), und ADR-Dokumentation schaffen eine belastbare Grundlage für künftige Agenten. Erweiterbare Codebausteine (libs) bieten eine Basis, welche von allen Agenten verwendet und erweitert werden können.

Wartbarkeit: Durch die Trennung von Core und Agenten, Zod-Schemata und Monitoring-Infrastruktur (Prometheus, Grafana, Loki) ist der Betrieb nachvollziehbarer und Fehler lassen sich schneller eingrenzen. Reporter können auf ein vorkonfiguriertes Dashboard zugreifen.

Skalierbarkeit: Die Microservice-Architektur mit Kubernetes-Deployments erlaubt horizontale Skalierung und Deployments pro Agent, welche enterprise-ready sind.

Flexibilität: Mit NATS als Kommunikationsschicht ist das Framework offen für neue Use Cases und zukünftige Erweiterungen.

Für die CISTEC AG entsteht ein klarer Mehrwert durch reduzierten Wartungsaufwand, höhere Wiederverwendbarkeit, gesteigerte Flexibilität und konsequente Standardisierung. Besonders deutlich wird dies an der Entwicklungsgeschwindigkeit neuer Agenten: Während bislang mehrere Tage für die Umsetzung benötigt wurden, reduziert das Framework diesen Aufwand auf wenige Stunden, da zentrale Infrastruktur und Basiskomponenten bereits vorhanden sind. Die gewählte Architektur stellt zugleich sicher, dass das System sowohl Batch-Verarbeitungen als auch eventgetriebene Szenarien unterstützt und damit nachhaltig auf unterschiedliche Einsatzkontexte vorbereitet ist. Damit erfüllt das Framework nicht nur die inhaltlichen Ziele der Masterarbeit, sondern liefert der CISTEC AG eine tragfähige Grundlage, um ihre Agentenlandschaft langfristig resilienter, effizienter und zukunftssicher zu gestalten.

6.2 Challenges

Die Projektergebnisse zeigen, dass der Erfolg dieser Arbeit weniger von einer idealisierten Planung als vielmehr von der Fähigkeit abhing, auf Unsicherheiten adaptiv zu reagieren und zentrale Entscheidungen transparent zu dokumentieren. Die im Kapitel der Risikoanalyse dargestellten Entwicklungen verdeutlichen, dass sowohl fachliche als auch technische und organisatorische Faktoren den Projektverlauf massgeblich beeinflussten.

6.2.1 Domänenspezifischer vs. generischer Framework-Prototyp

Ein zentrales Spannungsfeld lag in der Abgrenzung zwischen einem domänenspezifischem und generischen Framework-Ansatz. Ursprünglich bestand die Erwartung, klinische Agenten aus

KISIM nachzubilden. Diese Ausrichtung erwies sich jedoch als zu komplex und zu eng mit spezifischen Strukturen der CISTEC AG verknüpft. Mit der Entscheidung ADR0007 wurde dieser Pfad bewusst verlassen und durch abstrahierte, alltagsnahe Use-Cases ersetzt. Diese Abkehr von fachlicher Detailtreue ermöglichte eine generische Lösung, die auch für Entwickler:innen ohne klinischen Hintergrund nachvollziehbar blieb. Gleichzeitig wurden gute Grundlagen für Innovation und langfristige Erweiterbarkeit bereitet. Allerdings führte die späte Abgrenzung zu Verzögerungen, was die Bedeutung eines frühzeitigen Scope-Managements unterstreicht.

6.2.2 Cyberangriff der CISTEC AG

Neben der fachlichen Dimension prägten externe Ereignisse den Projektverlauf erheblich. Der Ransomware-Angriff auf die CISTEC AG verzögerte die Bereitstellung von Testdaten und reduzierte die Möglichkeiten der Auftraggeberin und des Maintainers, aktiv am Projekt mitzuwirken. Als die Daten verfügbar wurden, erwiesen sie sich zudem als ungeeignet für Entwicklungszwecke. Die Konsequenz war die Definition einer eigenen Domäne, die es erlaubte, unabhängigen Fortschritt zu erzielen. Dieses Vorgehen erwies sich als entscheidend für den Projekterfolg, zeigte jedoch auch, dass externe Abhängigkeiten frühzeitig durch eigenständige Initiativen abgefedert werden müssen.

6.2.3 Enterprise-Infrastruktur

Technisch stellte die Intransparenz der bestehenden Build- und Deployment-Pipelines eine weitere Herausforderung dar. Da die internen Prozesse der CISTEC AG aufgrund ihrer Komplexität und Sicherheitsvorgaben nicht zugänglich waren, musste das Projektteam eine eigenständige, schlanke CI/CD-Pipeline aufbauen. Mit Fleet und GitOps konnte eine funktionale Lösung geschaffen werden, die die Kernanforderungen abdeckte, sich jedoch nicht nahtlos in die Verteilung der CISTEC AG mit ArgoCD integrieren lässt. Abstriche bei Persistenz, Skalierung und Security waren dabei unvermeidlich, etwa durch fehlende Log-Historien oder fehlendes Secret-Management. Dennoch zeigte sich, dass der Fokus auf Deployment, Logging und Modularität ausreichend war, um einen praxistauglichen Prototyp zu liefern.

6.2.4 Kommunikation und Zusammenarbeit

Organisatorisch erwiesen sich eingeschränkte Stakeholder-Präsenz und fehlende technische Rollen auf Seiten der CISTEC AG als Hemmfaktoren. Der Product Owner war über längere Zeit abwesend, wodurch regelmässige Abnahmen nicht stattfinden konnten. Entscheidungen mussten daher eigenverantwortlich getroffen und in Architecture Decision Records (ADR’s) dokumentiert werden. Auch innerhalb des Projektteams führten ungleichmässige Arbeitslast und temporäre Abwesenheiten zu zusätzlichem Koordinationsaufwand. Diese Herausforderungen konnten durch zusätzliche Syncs, Pair Programming und eine konsequente Dokumentation abgemildert, jedoch nicht vollständig gelöst werden.

6.2.5 Fazit

Zusammenfassend lässt sich feststellen, dass die grössten Herausforderungen weniger in der technischen Umsetzung als vielmehr in der Domänenabgrenzung, der Abhängigkeit von externen Faktoren und der organisatorischen Koordination lagen. Die Entscheidung für ein generisches Framework, die Einführung einer eigenen Testdomäne und der Aufbau einer unabhängigen Deployment-Pipeline erwiesen sich als zentrale Weichenstellungen, um das Projekt erfolgreich abzuschliessen. Gleichzeitig verdeutlicht der Projektverlauf, dass Resilienz in der Softwareentwicklung nicht durch Risikofreiheit entsteht, sondern durch die Fähigkeit, auf Unsicherheiten flexibel, offen und transparent zu reagieren.

6.3 Weiterentwicklung

Der im Rahmen dieser Arbeit entwickelte Prototyp bildet eine robuste Grundlage für den Betrieb von Agenten im klinischen Kontext. Gleichwohl konnten nicht alle potenziellen Funktionen im Prototypen umgesetzt werden. Für eine zukünftige Weiterentwicklung sind insbesondere folgende funktionale Erweiterungen denkbar:

6.3.1 Message Queue

Die aktuelle Kommunikation zwischen Core und Agenten basiert auf einem Request-Response-Muster via NATS. Für den produktiven Einsatz könnte dieser Mechanismus um eine Message Queue ergänzt werden. Eine solche Warteschlange erlaubt es, Agenten nach dem Pull-Prinzip Aufträge abarbeiten zu lassen. Jeder Agent kann so eigenständig neue Jobs aus der Queue entnehmen und abarbeiten. Dieses Verfahren erhöht die Robustheit, da Aufträge im Falle eines Absturzes nicht verloren gehen. Zusätzlich sind Retries durch den Broker unmittelbar umsetzbar. NestJS bietet hierfür bereits integrierte Adapter (z. B. für NATS JetStream oder Redis Streams), sodass die Architektur flexibel erweiterbar bleibt.

6.3.2 Transaction Logging

Für Nachvollziehbarkeit und Revisionssicherheit soll künftig ein zentrales Transaction Logging etabliert werden. Jede erzeugte oder aktualisierte Leistung wird dabei in einer separaten Entität der Datenbank persistiert. Durch dieses Vorgehen entsteht ein detailliertes Journal aller durchgeführten Operationen, das sowohl für Audits als auch für technische Analysen genutzt werden kann. Darüber hinaus bildet es die Basis für Nachgenerierungen, da Transaktionen für eine bestimmte Zeitspanne erneut ausgerollt werden können.

6.3.3 Nachgenerierung

Im aktuellen Prototyp existiert noch kein Mechanismus zur Nachgenerierung von Leistungen. Dieser ist jedoch essenziell, wenn Daten nachträglich korrigiert werden müssen (z. B. bei ERP-Störungen oder fehlerhaften Abrechnungsdaten). Eine mögliche Erweiterung besteht darin, dass Agenten auf Basis des Transaction Logs oder Message Queues mithilfe eines Von- und Bis-Datums erneut ausgeführt werden können. Auf diese Weise lassen sich vergangene Leistungen vollständig oder partiell neu berechnen, ohne die Integrität der

bestehenden Daten zu gefährden.
Eine praxisnahe Ergänzung wäre ebenfalls die Einführung eines Betriebsmodus-Flags, mit dem Agenten zwischen Normalbetrieb und Nachgenerierung umgeschaltet werden können. Im Normalbetrieb arbeiten die Agenten wie gewohnt im kontinuierlichen Modus und verarbeiten aktuelle Daten. Im Nachgenerierungsmodus hingegen wird eine spezifische Zeitspanne berücksichtigt, die Agenten arbeiten alle relevanten Datensätze innerhalb dieses Bereichs erneut ab. Dieses Vorgehen trennt die beiden Anwendungsfälle klar und erleichtert die Nachvollziehbarkeit bei Bugs.

6.3.4 Monitoring und Alerting

Im Bereich Monitoring ist bislang nur Logging umgesetzt, das Systemereignisse in Grafana sichtbar und bequem zugänglich macht. Für die nächste Ausbaustufe bietet es sich an, das Setup um Prometheus und den Alertmanager zu erweitern (siehe Abb. 14: Systemüberblick). Prometheus sammelt Metriken wie CPU- und RAM-Auslastung, Durchsatz und Fehlerraten und macht Trends sichtbar. Der Alert Manager löst bei definierten Schwellwerten automatisch Benachrichtigungen (z. B. per Mail oder Slack) aus. So lässt sich gezielt und frühzeitig eingreifen und würde den UC08 “Agent seit X Tagen keine Leistung mehr erzeugt” abdecken.
Die Erweiterung von Prometheus für das Monitoring gilt es auch in der Verteilung nachzuziehen und zu integrieren. Metriken sollten im Agent-Core aggregiert werden, bis sie von Prometheus abgerufen werden. Ausserdem läuft Loki derzeit ohne Persistenz. Für eine produktive Umgebung sollten Persistent-Volume-Claims mit ausreichender Retention eingeplant werden.

6.3.5 NATS Queue-Groups und Jetstream

NATS bietet verschiedene Kommunikationsmuster an – unter anderem Queues-Groups. Wir empfehlen, das System um Queues-Groups zu erweitern, um eine robustere und skalierbare Lösung zu erreichen. Diese Erweiterung ermöglicht die Nachrichtenverteilung als Lastverteilung in Pub/Sub-Szenarien: hierbei hören mehrere Worker auf das gleiche Subject, jedoch verarbeitet nur ein Worker aus der Gruppe die Nachricht.
Für persistente Nachrichten und grantierte Zustellung, lässt sich NATS optional mit Jetstream erweitern, sodass Nachrichten Ausfälle überdauern und später automatisch nachgeholt werden können. Lastspitzen werden gepuffert und Verbraucher entkoppelt und stellen einen schlanken Key/Value-Store für Zustände oder Konfigurationen bereit. In Kombination mit NATS und Queue-Groups ergibt sich eine robuste Basis, die sich leicht horizontal skalieren lässt.

6.3.6 Verteilung, Deployment, DevOps, Security und Skalierung

Die heutige Fleet/GitOps-Struktur funktioniert gut, ist aber von automatischen Tests ausgeschlossen. Für den produktiven Betrieb sollte die automatische Verteilung um automatisierte Tests ergänzt werden. Beim Thema Sicherheit haben wir im Prototyp bewusst Abstriche gemacht. In einem produktiven Betrieb gehören Passwörter und Zugangsdaten nicht ins Repo, sondern in ein Secret-Management. Ebenso ist die Kommunikation zwischen den Diensten über NATS zu verschlüsseln.

7 Fazit

Ziel dieser Arbeit war es, ein Framework zu entwickeln, welches die Agenten-basierte Leistungserfassung in Klinikinformationssystemen standardisiert, modularisiert und langfristig wartbar macht. Ausgangspunkt bildete die Analyse der bestehenden, Agentenlandschaft bei der CISTEC AG, die durch individuelle Implementierungen, hohen Wartungsaufwand und eingeschränkte Skalierbarkeit geprägt war.
Die Entscheidung, die bestehenden Agenten nicht 1:1 nachzubilden, sondern eine eigene Domäne aus nicht-medizinischen Szenarien zu verwenden, reduzierte die fachliche Komplexität erheblich und führte zu einer klareren, robusteren Architektur. Auf dieser Basis wurden exemplarische Use Cases umgesetzt (Agent-NoShow, Agent-Marathon, Agent-Grenzkontrolle), welche die Modularisierung und Wiederverwendbarkeit von Komponenten demonstrieren.
Die Arbeit zeigt, dass mit einer serviceorientierten Architektur auf Basis von NestJS, NATS, TypeORM, Schema-Validierung in zod, und Kubernetes ein tragfähiges Fundament für Agenten geschaffen werden kann.
Ergänzt durch einen einheitlichen Prozess, welcher als Grundlage für alle Agenten gültig ist, werden Doppelspurigkeiten in der Business-Logik signifikant verringert. Mittels gebündelten Logging-Ressourcen können die Agenten mühelos in Grafana eingesehen werden, um bei allfälligen Problemen im klinischen Bereich schneller auf Schlussfolgerungen kommen zu können. Damit wurde ein System entworfen, das sowohl robust als auch flexibel ist und bei fachlichen Anforderungen im klinischen Bereich einfach erweitert werden kann. Durch die zentrale Rolle von ADR’s konnte die Transparenz im Entwicklungsprozess sichergestellt und zukünftige Weiterentwicklungen methodisch abgesichert werden.
Die Evaluierung anhand von Muss- und Kann-Kriterien sowie einer Risikomatrix verdeutlicht, dass das Framework die identifizierten Pain Points adressiert: Wiederverwendbarkeit, Standardisierung und Wartbarkeit wurden deutlich verbessert. Gleichzeitig wurde durch die modulare Struktur die Grundlage für Erweiterbarkeit geschaffen.
Für die Zukunft eröffnen sich Weiterentwicklungsmöglichkeiten wie Message Queues, erweitertes Transaction-Logging, Mechanismen für Nachgenerierungen, oder die Erweiterung um kundenindividuelle Konfigurationen. Diese Arbeit bildet somit die Basis für einen nachhaltigen Transformationsprozess in der Leistungserfassung von einer fragmentierten, schwer wartbaren Landschaft hin zu einem standardisierten, transparenten und zukunftsfähigen System.

8 Abbildungsverzeichnis

Abb. 1: Systemkontextdiagramm 12
Abb. 2: Projektplan zu Projektbeginn 13
Abb. 3: Kollaboration und Kommunikationsablauf im Entwicklungsprozess 14
Abb. 4: Branching sowie Build- und Deploymentprozess im Entwicklungsprozess 16
Abb. 5: Stakeholder-Matrix 18
Abb. 6: Microservices Testing-Strategie (Schaffer, 2018) 19
Abb. 7: Entwicklungsphasen 20
Abb. 8: Use-Case Diagramm 25
Abb. 9: Domain Model 29
Abb. 10: Risikomatrix der initial definierten Risiken 31
Abb. 11: Risikomatrix der technischen Risiken 32
Abb. 12: Risikomatrix der organisatorischen Risiken 33
Abb. 13: Risikomatrix der finalen Risiken 34
Abb. 14: Systemüberblick 45
Abb. 15: Paketdiagramm 47
Abb. 16: Verteilungsdiagramm 49
Abb. 17: SonarQube Quality Gate 52
Abb. 18: Projektplan Soll-Ist-Vergleich 61
Abb. 19: Aufwandsverlauf (kumuliert) 61
Abb. 20: Aufwandsintensität (exponentiell geglättet) 62

9 Tabellenverzeichnis

Tab. 1: Übersicht über Projektbeteiligte und Projektrollen 13
Tab. 2: Nutzung und Erklärung der genutzten Labels 15
Tab. 3: Rollendefinition 17
Tab. 4: Stakeholder-Liste 18
Tab. 5: Use-Case Unterteilung 26
Tab. 6: Funktionale und Qualitätsanforderungen 28
Tab. 7: Traceability Matrix 29
Tab. 8: Erwartete technische Risiken 32
Tab. 9: Erwartete organisatorische Risiken 33
Tab. 10: Erwartete finale Risiken 35
Tab. 11: Auswertung Technologiestack 35
Tab. 12: Arbeitsaufwände (zusammengefasst) 39
Tab. 13: Auswertung finaler Risiken (zusammengefasst) 41
Tab. 14: Architektur Vor- und Nachteile 44
Tab. 15: Deklaration genutzter (KI)-Tools 60

10 Literaturverzeichnis

Cockburn, A. (2011). Writing effective use cases (23rd printing). Addison-Wesley.

FMH Swiss Medical Association: TARMED: Online-Tarifbrowser. Publikationsdatum unbekannt. Zuletzt abgerufen am 14.09.2025 von https://www.fmh.ch/themen/ambulante-tarife/tarmed-tarifbrowser-datenbank.cfm

FMH Swiss Medical Association: Wichtige Tarif-Info | TARDOC. Publiziert am 04.09.2025. Zuletzt abgerufen am 14.09.2025 von https://tarifeambulant.fmh.ch/wichtige-tarif-info.cfm

Jochum, K. Ransomware-Angriff auf KIS-Anbieter Cistec. Publiziert am 28.02.2025. Zuletzt abgerufen am 13.09.2025 von https://www.inside-it.ch/ransomware-angriff-auf-kis-anbieter-cistec-20250228

NATS.io. The official NATS documentation. Publiziert am 01.08.2025. Zuletzt abgerufen am 15.09.2025 von https://docs.nats.io/

NATS.io. Queue Groups. Publiziert am 06.12.2024. Zuletzt abgerufen am 15.09.2025 von https://docs.nats.io/nats-concepts/core-nats/queue

Nygard, M. Documenting Architecture Decisions. Publiziert am 15.11.2011. Zuletzt abgerufen am 13.09.2025 von https://www.cognitect.com/blog/2011/11/15/documenting-architecture-decisions

Prokosch, H.-U. (2001) KAS KIS EKA EPA EGA E-Health: Ein Plädoyer gegen die babylonische Begriffsverwirrung in der Medizinischen Informatik. Münster: Westfälische-Wilhelms-Universität Münster.

Schaffer, A. Testing of Microservices. Publiziert am 11.01.2018. Zuletzt abgerufen am 12.09.2025 von https://engineering.atspotify.com/2018/01/testing-of-microservices

11 Anhang

11.1 Deklaration zu genutzten (KI)-Tools

Tool-Kategorie	Tools	Verwendungszwecke im Projekt
Plattform zur Versionsverwaltung	● GitLab (self-hosted durch OST)	● Code Reviews ● Issue Board ● Technische Dokumentation ● Versionsverwaltung
Kommunikation	● Discord ● Microsoft Teams ● WhatsApp ● Zoom	● Teamkommunikation, ● Abstimmungen intern und extern ● Pair-Programming Sessions
IDE	● Visual Studio Code ● WebStorm	● Lokale Entwicklungsumgebung ● Pair-Programming Sessions
Ergänzungen zur Lokalen Entwicklungsumgebung	● Docker ● Helm ● kubectl	● Lokale Entwicklungsumgebung
AI	● ChatGPT ● Codepilot	● Hilfsmittel für das Generieren von Code-Bausteinen, Dokumentation, sowie Fehlerbehandlung.
Darstellungen	● Draw.io ● Excalidraw	● Erstellung von Diagrammen und Visualisierungen
Zeiterfassung	● Excel ● Timecamp	● Nachvollziehbarkeit der aufgewendeten Arbeit
Dokumentenablage	● Google Drive	● Datenablage ● Kollaborativer Zugriff auf Protokolle und Abschlussarbeit

Tab. 15: Deklaration genutzter (KI)-Tools

11.2 Zeiterfassung und Zeitauswertung

Soll-Ist-Vergleich:

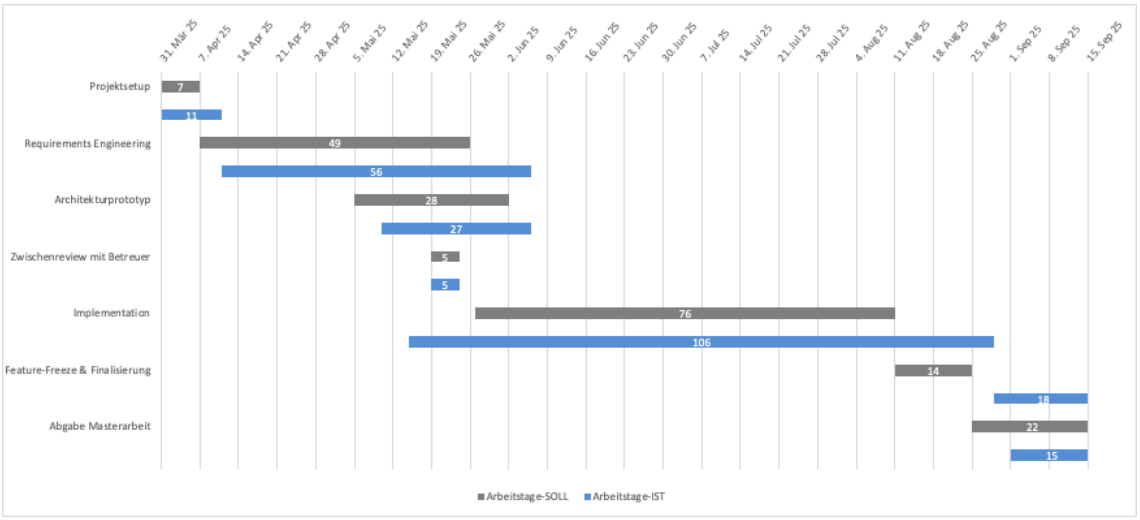


Abb. 18: Projektplan Soll-Ist-Vergleich

Die Abb. 18 zeigt Soll (grau) und Ist (blau). Das Setup dauerte etwas länger, das Requirements Engineering zog sich bis in den Mai. Beim Architekturprototyp konnten wir leicht aufholen. Die Implementierung startete teils vorzeitig, erwies sich jedoch als komplexer als erwartet. Zur gleichen Zeit entschieden wir Mitte Juni, von den fachlichen Kriterien Abstand zu nehmen, was weitere Anpassungen nötig machte. Dadurch verschob sich der Feature-Freeze leicht, und die Abgabephase fiel entsprechend kompakter aus.

Aufwandsverlauf (kumuliert):

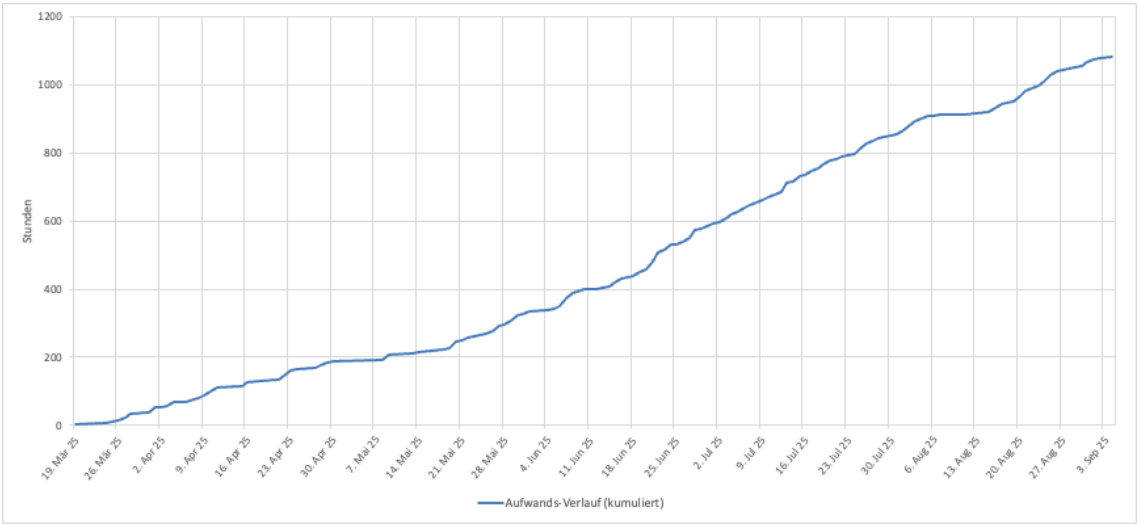


Abb. 19: Aufwandsverlauf (kumuliert)

Der kumulierte Aufwand steigt über die gesamte Laufzeit stetig an und erreicht zum Schluss gut über tausend Stunden. Ab Ende Mai zieht die Kurve sichtbar an – die steilste Phase liegt zwischen Mitte Juni und Ende Juli.

Hinweis: Darstellung endet am 3. September, da der Export aufwändig war.

Aufwandsintensität (exponentiell geglättet):

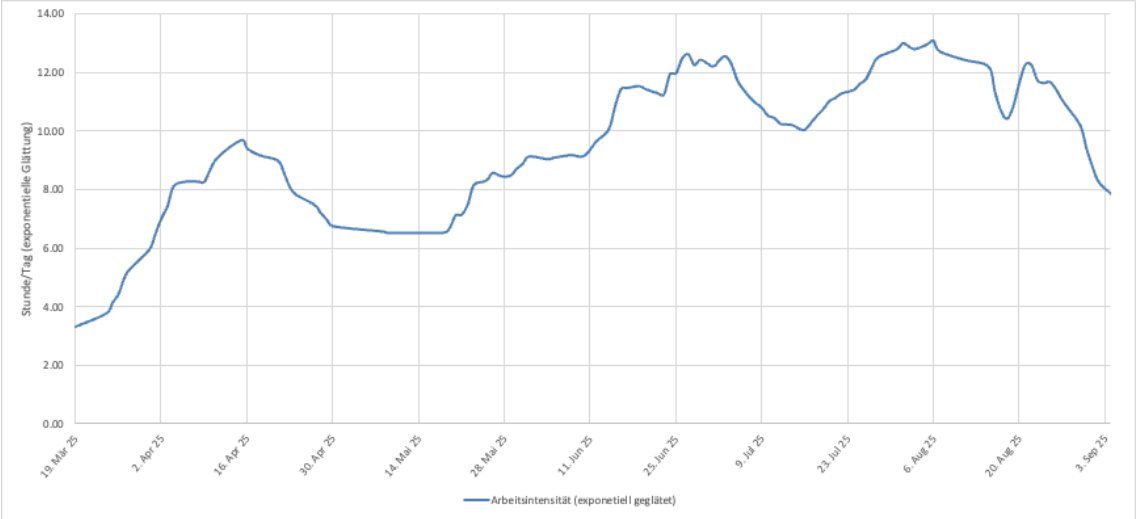


Abb. 20: Aufwandsintensität (exponentiell geglättet)

Die tägliche Intensität startet moderat, erreicht Mitte April erste Spitzen, fällt kurz ab und steigt ab Juni erneut deutlich an. Im Juli/August halten längere Hochphasen an. In den letzten Wochen nimmt die Intensität spürbar ab.

Hinweis: Darstellung endet am 3. September, da der Export aufwändig war.

11.3 ADR's

0001 TypeScript Framework NestJS

Entschieden am: 12.04.2025

Status

- [ ] vorgeschlagen
- [x] angenommen
- [ ] abgelehnt
- [ ] veraltet
- [ ] ersetzt

Kontext

- Unser grobes Ziel ist die Entwicklung eines Frameworks zur Generierung von "Agenten" Modulen in der Sprache TypeScript, um die Flexibilität dieser Module zu erhöhen und deren Wartungs- und Entwicklungsaufwand zu verringern
- Um ein möglichst nachhaltiges und einfaches Framework zu erstellen, bietet es sich an, ein Framewokr zu nutzen und das Rad nicht neu zu erfinden

Entscheidung

- Auswahlkriterien für das Framework:
  - TypeScript benutzbar
  - Für Cistec Entwickler:innen gut benutzbar
  - ein beständiges und sicheres Framework
- Untersuchte Frameworks:
  - ExpressJS
  - NestJS
  - AdonisJS

Framework	Pro	Contra
ExpressJS	Absolute Kontrolle	Gefahr für Spaghetti Code, Wildwuchs
		Wartbarkeit hängt sehr fest von der Disziplin der Devs ab
		Loose Kopplung ist schwieriger erreichbar
AdonisJS	Leicht zu erlernen	Community getriebenes Framework
	Ähnlichkeit zu PHP Framework Laravel vereinfacht Arbeit für entsprechend erfahrene Devs	Kleine Community
		Cistec kann vom Laravel-Vorteil nicht profitieren
NestJS	Unternehmen hinter dem Framework	hohe Lernkurve
	Grosse Community	braucht mehr boilerplate
	Gibt durch Modularität Struktur vor	
	decorator-orientiert	
	microservice-affin	
	Cistec nutzt bereits NestJS	

Folgen

Das wird aufgrund dieser Veränderung einfacher:

- siehe NestJS pros

Das wird aufgrund dieser Veränderung schwieriger:

- siehe NestJS cons

0002 Library Zod

Status

Wie ist der Status der Entscheidung?

- [ ] vorgeschlagen
- [x] angenommen
- [ ] abgelehnt
- [ ] veraltet
- [ ] ersetzt

Kontext

Das Agentenframework basiert auf TypeScript, das zur Kompilierzeit bereits sehr gute Typsicherheit bietet. Allerdings bietet TypeScript keine Laufzeit-Typenvalidierung. Sobald z.B. Daten von einer externen Quelle (Datenbank, HTTP, Message Queue) eingehen, existiert kein Schutz mehr zur Laufzeit.

Um die Agenten robust, fehlertolerant und erweiterbar zu gestalten, benötigen wir:

- Schema-Validierung zur Laufzeit
- Typ-Inferenz zur Entwicklungszeit
- Erkennbarkeit und Dokumentation der erwarteten Datenstruktur als Code

Entscheidung

Wir setzen die Library zod ein, um:

- Laufzeit-Validierung von Daten und Konfigurationsobjekten vorzunehmen
- Zentrale Datenstrukturen (z.B. OP-Bericht, Patienten, Agent-Konfigurationen) als Schemas zu definieren
- Gleichzeitig Typen für TypeScript direkt abzuleiten (z.infer)

Zod ersetzt in unserem Projekt die sonst im NestJS-Ökosystem übliche Kombination aus:

- class-validator
- class-transformer

Wir haben uns aus folgenden Gründen bewusst gegen class-validator und class-transformer entschieden:

- Stark auf Klassen und Decorators ausgelegt
- Keine Schema-zentrierte Entwicklung wie bei zod

Folgen

Das wird aufgrund dieser Veränderung einfacher:

- Einheitliche und typensichere Datenvalidierung im gesamten System
- Einfache Generierung von DTO-Types, Konfigurationen, Agenten-Datenstrukturen
- Reduzierte Fehler durch explizite Schema-Definitionen
- Bessere Developer-Experience durch Autocompletion und Fail-Fast Approach bei Inkompatibilität

Das wird aufgrund dieser Veränderung schwieriger:

- Integration in NestJS benötigt zusätzliche Pipe (z.B. zod-validation-pipe) für Validierung
- Entwickler müssen mit funktionaler Validierung statt OOP/Decorators arbeiten, Geschmackssache
- Keine Fehlertoleranz bei sich rasch verändernder Datenstruktur

0003 Architektur-Prototyp Agentenframework

Status

Wie ist der Status der Entscheidung?

- [x] vorgeschlagen
- [ ] angenommen
- [ ] abgelehnt
- [ ] veraltet
- [ ] ersetzt

Kontext

Für das Agentenframework wurde ein Prototyp mit NestJS, zod und TypeORM erstellt. Dieser soll die Basis für eine skalierbare, wartbare und konfigurierbare Plattform bieten, um automatisiert medizinische Leistungen durch verschiedenen Informationen aus einer Datenbank eines KIS (Krankenhausinformationssystem) zu generieren.

Ziel ist ein mandantenfähiges Framework, das sowohl geplante Agentenprozesse via CRON-Jobs als auch manuell getriggerte Prozesse (z.B. manuelle Ausführung eines regulären Agentenjobs oder Nachgenerierung) ermöglicht. Dabei soll der agent-core als zentrale technische Steuerungseinheit fungieren, während Agenten als eigenständige Worker die entsprechende Businesslogik implementieren.

Entscheidung

Wir haben den Agenten-Prototypen auf Basis folgender Technologien und Konzepte erstellt:

- NestJS für Grundstruktur für Backend-Framework (siehe 0001 TypeScript Framework NestJS)
- zod für Laufzeitvalidierung und Typen-Synchronisation (siehe 0002 Library Zod)
- TypeORM für Relationale Datenbankanbindung (PostgreSQL)

Folgende Grunzprinzipien wurden für den Prototypen eingehalten

- agent-core als Package-Dependency pro Agent, um Inkompatibilitäten zwischen verschiedenen Agenten zu vermeiden
- *-agent als CRON-Job
- Zentrale API zur Steuerung mit Beispielabfragen in Bruno
- Zod-Schemas zur DTO-Validierung in allen Komponenten (agent & core)
- PostgreSQL als zentrale relationale Datenbank

Alternativen

Monolithisches Design

- Agent und Core im selben Prozess
- Nachteile: entspricht stark der problematischen IST-Lösung schwer skalierbar, kein unabhängiges Error-Handling, nicht Kubernetes-optimiert

Serviceorientierte Architektur

- Core als Singleton-Service, Agenten kommunizieren via API oder Queue
- Vorteile: zentrale Wartung, einheitliche Logging & Alerting-Logik, Hotfix-fähig
- Nachteile: Netzwerkabhängigkeit, API-Versionierung notwendig, komplexeres Setup

Package-Dependency (Core mehrfach in Agenten gebündelt)

- Vorteile: Agenten enthalten automatisch kompatible Core-Version, keine zusätzliche Core-Infrastruktur erforderlich (z.B. Point-of-Entry), gute Wartbarkeit bei hoher Testabdeckung, API-Versionierung kein MUSS (aber dennoch sehr nützlich)
- Nachteile: Update-Aufwand bei gröberen Core-Änderung, Redundanz, Versionsinkonsistenz bei Deployments
- Randnotiz: Diese Variante wurde zur Umsetzung gewählt siehe 0005 Version Fallback sicherstellen

Folgen

Das wird aufgrund dieser Veränderung einfacher:

- Skalierbares Agentenframework (pro Mandant, pro Agent)
- Jeder Agent enthält die richtige agent-core-Version automatisch → keine zentrale Abhängigkeit
- CI-Tests pro Agent stellen sicher, dass die integrierte agent-core-Version funktioniert (API-spezifische Tests möglich)
- Einfache lokale Entwicklung & Tests (kein externer Service nötig)
- Dynamische Erweiterbarkeit um neue Agenten
- Wiederverwendbare zentrale Logik in agent-core (Export, Validierung, Konfiguration)

- Kein zusätzliches Deployment für agent-core notwendig

Das wird aufgrund dieser Veränderung schwieriger:

- Änderungen an agent-core erfordern situative Abwägung von Abwärtskompatibilität veralteter Funktionalität bei allen Agenten
- Konsistenz muss über CI-Tests und Versionsmanagement der API gewährleistet sein
- Höherer Speicherverbrauch in Artefakten (agent-core mehrfach enthalten)

Fazit

Der Prototyp mit NestJS bietet eine tragfähige und moderne Grundlage für das Agentenframework. Durch die klare Trennung zwischen Core und Agenten, moderne Schemavalidierung mit zod und erste geklärte strukturelle Fragestellungen für zukünftige Deployments ist das System langfristig wartbar, mandantenfähig und performant.

Diese Architektur wird schrittweise weiterentwickelt und modularisiert, um zusätzliche Anforderungen schnell und kontinuierlich integrieren zu können. Damit ist die Grundlage für den Prototypen gelegt.

0004 Package Struktur

Status

Wie ist der Status der Entscheidung?

- [ ] vorgeschlagen
- [x] angenommen
- [ ] abgelehnt
- [ ] veraltet
- [ ] ersetzt

Kontext

- Welches Problem sehen wir, das uns zu dieser Veränderung oder Entscheidung motiviert?
- Der Agent Core code ist aktuell wie folgt in packages organisiert: das package users/dto enthält dto und entity code
- die aktuelle Struktur entspricht also dem Ansatz package by feature; sie entspricht also damit nicht der Onion Architektur
- Wollen wir uns jetzt für eine Strategie, wie Code in Paketen strukturieren, entscheiden? Wenn ja, zu welcher und worauf müssen wir hierbei besoners Rücksicht nehmen?

Entscheidung

Vorschläge:

1. Onion Architektur
2. Layered Architektur
3. Entscheidung abwarten und Learning by Doing: Was geben unsere bisherigen Architekturentscheidungen vor? Und was brauchen wir, um unsere Anforderungen gut umsetzen zu können?

Entscheidung: 3.

Begründung:

- NestJS ist modular aufgebaut
- das heisst, bereits mit dem Anwenden des CLI-Commands zum Generieren von Dateien ist ein örtliches Auftrennen nach einer anderen Logik nicht möglich

Folgen

Das wird aufgrund dieser Veränderung einfacher:

- Entscheidungsfindung: Flexibilität und Erfahrung nutzen, um informiert und lösungsgerecht eine bewusste Entscheidung treffen zu können.

Das wird aufgrund dieser Veränderung schwieriger:

- Einschränkung durch Entscheid für NestJS und folglich durch die Architektur von NestJS.

0005 Version Fallback sicherstellen

Status

Wie ist der Status der Entscheidung?

- [ ] vorgeschlagen
- [x] angenommen
- [ ] abgelehnt
- [ ] veraltet
- [ ] ersetzt

Kontext

- Annahme: Es wird im Laufe der Zeit konsequenterweise laufend weitere Versionen des Frameworks geben
- Problem: Wie stellen wir sicher, dass Agenten stets mit einer jeweils kompatiblen Version des Frameworks / Agent Core arbeiten können?

Entscheidung

Strategie

- mittels Git Tags und Release Branches tragen wir dafür Sorge, dass die Frameworkversionen referenzierbar und sichtbar werden
- im main repository befindet sich immer der neueste Stand
- die Agenten verbinden sich via REST call auf die jeweils von ihnen benötigte Version des Agent Core

Folgen

Das wird aufgrund dieser Veränderung einfacher:

- Versionierung und Version Fallback sind so sichergestellt

Das wird aufgrund dieser Veränderung schwieriger:

- ...

0006 Domainsprache Deutsch

Status

Wie ist der Status der Entscheidung?

- [x] vorgeschlagen
- [ ] angenommen
- [ ] abgelehnt
- [ ] veraltet
- [ ] ersetzt

Kontext

Das Agentenframework bewegt sich im Kontext des schweizer Gesundheitswesens, welches stark durch die Fachsprache geprägt ist. Fachbegriffe wie "Patient", "Fall" oder "Leistung" sind nicht nur lokal etabliert, sondern auch fester Bestandteil regulatorischer und betrieblicher Prozesse.

Im bisherigen Code und im Proof of Concept des NoShow-Agents wurden einige domänenspezifische Begriffe ins Englische übersetzt (z.B. medicalCaseId, costCenter, ...). Dies erschwert die fachliche Lesbarkeit und die Kommunikation zwischen technischen und fachlichen Stakeholdern. Weiter wurde hier ein Unterbruch zwischen Code und Domäne festgestellt.

Entscheidung

Wir entscheiden uns, domänenspezifische Begriffe konsequent in deutscher Fachsprache abzubilden. Dies bildet fachliche Konzepte wie Fall, Kostenstelle, Leistung, Diagnosen, sowie Entities wie Patient, Termin, Leistungserbringer ab. Strukturelemente mit Domainbezug wie fallId, kostenstelle, behandlungsart werden ebenfalls ausgedeutscht.

Die Englische Sprache soll wie gewohnt beibehalten werden für technisch generische Konzepte wie id, date, isProcessed, status, createdAt, dto, service, controller etc.

Folgen

Das wird aufgrund dieser Veränderung einfacher:

- Klarere Trennung zwischen fachlichen und technischen Begriffen
- Verbesserte Verständlichkeit für nicht-technische Stakeholder
- Weniger Missverständnisse bei Reviews, Testfällen und Business Rules
- Näher am Domain Model

Das wird aufgrund dieser Veränderung schwieriger:

- Bisherigen Code für den NoShow-Agenten auf Deutsche Fachsprache ummünzen

0007 Keine klinische Fachkriterien für eine erfolgreiche Abnahme

Status

Wie ist der Status der Entscheidung?

- [ ] vorgeschlagen
- [x] angenommen
- [ ] abgelehnt
- [ ] veraltet
- [ ] ersetzt

Hinweis: Diese Entscheidung wurde mit dem Product Owner der Leistungserfassung bei CISTEC AG (Martina Lux) besprochen und angenommen.

Kontext

Die CISTEC AG ist die Auftraggeberin für dieses Projekt und hat ein starkes Interesse daran, ein Endprodukt zu erhalten, das einen konkreten Nutzen im Alltag der medizinischen Leistungserfassung bringt. Zu Projektbeginn wurde deshalb vereinbart, dass drei klinisch geprägte Agenten (NoShow, OAT, Anästhesie) möglichst nahe am IST-Zustand des Systems nachgebaut werden sollten.

Im Laufe der Umsetzung stellte sich jedoch heraus, dass diese Vorgabe unnötig einschränkend wirkt. Man neigte stark dazu, sich zu fest an bestehenden Strukturen zu orientieren, anstatt mit kreativen und erneuernden Ideen zu experimentieren.

Ebenfalls wurde klar, dass es für die Entwickler:innen ohne Bezug zur CISTEC AG schwierig war, die klinischen Use Cases intuitiv zu verstehen, ohne dabei sich strikte an eine Vorgabe zu halten.

Zusammen mit der CISTEC AG wurde deshalb geprüft, ob der ursprüngliche Scope angepasst werden kann. Ziel war es, Raum für technische Innovation und fachliche Abstraktion zu schaffen, ohne dabei den realen Nutzen aus den Augen zu verlieren.

Entscheidung

Die Vorgabe, bestehende Agenten der CISTEC AG nachzuimplementieren, wurde **aufgehoben**. Stattdessen sollen Use Cases und Agenten konzipiert werden, welche Analogien zum nichtmedizinischen Alltag besitzen. Diese sind fachlich nachvollziehbar, technisch abbildbar und erleichtern die Kommunikation zwischen Entwicklung und Stakeholdern.

Die daraus resultierenden Agenten dürfen kreativ modelliert werden, solange sie:

- Zentrale Anforderungen der ursprünglichen Agenten weiterhin abdecken
- Den fachlichen Workflow in abstrahierter Form widerspiegeln
- In ein modulares und konfigurierbares Framework integrierbar bleiben

Die Formulierung der Agenten würde wie folgt abweichen:

Agent-Name (vorher)	Anforderungsbeschreibung (vorher)	Agent Name (nachher)	Anforderungsbeschreibung (nachher)
NoShow-Agent	Der Agent sucht nach Terminen mit dem Sprechstundenstatus `PROC1=NOSHOW` und dem Terminstatus `DONE`	Business-Meeting-Agent	Der Agent überprüft, ob Business-Personen bei einem geplanten Meeting gefehlt haben. Fehlende Teilnahme wird protokolliert und kann intern über das Monitoring ausgewertet werden.
OAT-Agent	Der Agent generiert laufend Leistungen für die Laufzeit einer Intensivpauschalen-Massnahme bis ein Tag vor STOPDAT	Marathon-Leaderboard-Agent	Der Agent bewertet regelmässig aktive Läufer:innen während eines Marathons. Wer bereits im Ziel ist, wird nicht neu bewertet.
Anästhesie-Agent	Der Agent prüft, ob zu einem Termin exportierte Eingriffs-/Untersuchungsleistungen nach TARMED vorhanden sind und ein Anästhesie-Nachweis angelegt ist.	Flughafen-Grenzkontrolle-Agent	Der Agent prüft an einer Grenzkontrolle, ob alle Reisedokumente vollständig sind. Der Agent kann zwischen Reisepass, ID und Visum unterscheiden. Nur vollständige Fälle werden zugelassen. Fehlerdaten werden ebenfalls protokolliert und zurückgewiesen.

Konkrete Use Cases aus bestehenden Agenten werden entsprechend abgeleitet. Anforderungen werden so generischer und haben mehr Freiraum bei technischen Details wie z.B. Ablauf, Datenmodell. Die dadurch entstehende einfache Sprache lässt konkrete Anforderungen klarer wirken.

Folgen

Das wird aufgrund dieser Veränderung einfacher:

- Mehr kreativer Freiraum für die Umsetzung von Anforderungen

- Mehr Möglichkeiten für neue Ideen
- Leichteres Verständnis für Entwickler:innen ohne klinischen Hintergrund
- Flexibleres Datenmodell und Entkopplung von konkreten KIS-Legacy-Strukturen

Das wird aufgrund dieser Veränderung schwieriger:

- Integration bestehender Agenten bei CISTEC AG müssen nach Abnahme nachgezogen werden
- Fachspezifische Feinlogik muss bei Bedarf erneut detailliert aufgenommen werden
- Validierung der Agenten durch Fachexperten erfordert stärkere Kontextvermittlung

Fazit

Die Entscheidung, klinische Detailvorgaben zugunsten verständlicher, abstrahierter Use Cases aufzugeben, ist zentral für die technische und gestalterische Freiheit des Frameworks. Sie schafft ein gemeinsames Verständnis zwischen Entwicklung und Stakeholdern, ermöglicht Innovation und kann später bei Bedarf durch fachliche Präzisierung ergänzt werden.

Diese Entscheidung erlaubt eine MVP-orientierte, explorative Umsetzung der Agentenlogik.

0008 E2E Tests mit Testcontainers

Status

Wie ist der Status der Entscheidung?

- [ ] vorgeschlagen
- [x] angenommen
- [ ] abgelehnt
- [ ] veraltet
- [ ] ersetzt

Kontext

Mit der Umstellung von einem Monolithen auf eine Microservice-Architektur (Core + Agenten) war es notwendig, die E2E-Tests so zu gestalten, dass sie die reale Ausführungsumgebung möglichst gut abbilden. Kernpunkte dabei:

- **Abgrenzung der Tests**: Jeder Testlauf soll eine saubere, isolierte Umgebung erhalten.
- **Realistische Infrastruktur**: Core, PostgreSQL und NATS müssen wie im echten Betrieb laufen, damit die Agenten getestet werden können.
- **Einheitliche Seed-Daten**: Für jeden Test werden Entitäten gezielt erstellt, um Abhängigkeiten zu vermeiden.

Diskutierte Varianten:

1. **Programmgesteuert mit Testcontainers** *(gewählt)*

- Dynamisches Starten von Core, Postgres, NATS pro Testlauf
- Isolierte Netzwerk- und Containerumgebung
- Einfach zu kontrollieren, wann und wie Services gestartet/gestoppt werden

2. **SQL-Seed im Image / entrypoint**

- Vordefinierte Daten werden beim Containerstart eingespielt
- Weniger flexibel, Gefahr von veralteten Seeds bei Änderungen
- Erfordert häufiges Neubauen der Images

3. **TypeORM Migrations + Seed-Runner**

- Datenbankstruktur + Seed-Skripte beim Start ausführen
- Nützlich für initiale Demo-/Dev-Umgebungen
- Weniger geeignet für isolierte Tests, da Seeds global wirken

Entscheidung

Wir verwenden **Testcontainers** für den E2E-Testaufbau:

- Core, PostgreSQL und NATS werden programmgesteuert in einer eigenen Test-Netzwerkumgebung als Container gestartet
- Für jeden Testlauf wird die Datenbank neu erstellt (Drop + Synchronize)
- Seed-Daten werden im Test über Repository, statt ein globales Seed-Skript erzeugt

Begründung

- Hohe Realitätsnähe, da die Komponenten wie Core, Datenbank und NATS in Containern laufen
- Jeder Test ist vollständig isoliert und kann beliebige Testdaten selbst erstellen
- Kein Abhängigkeitschaos durch globale Seed-Skripte oder Migrations-Overhead
- Änderungen an Entities brechen die Tests frühzeitig (Left-Shift-Ansatz)
    - Da die Tests direkt mit echten Entity-Klassen arbeiten, schlagen Änderungen an deren Struktur bereits beim Kompilieren oder in statischer Codeanalyse fehl, nicht erst zur Laufzeit

Folgen

Das wird aufgrund dieser Veränderung einfacher::

- Isolierte, reproduzierbare Testumgebung – Tests sind unabhängig und beeinträchtigen sich nicht gegenseitig

- Realitätsnahe Tests (Container statt Mocks)
- Kein globaler Seed-State, sondern gezielte Arrange-Phase pro Test
- Fehler durch Schema-Änderungen werden früh erkannt (Left-Shift: Compilerfehler statt Runtime-Fehler)

Das wird aufgrund dieser Veränderung schwieriger::

- Testausführung dauert länger (Containerstart)
- Docker muss lokal oder in CI verfügbar sein
- Erfordert Image-Build vor Teststart
- Mehr Code beim Anlegen von Testdaten pro Test

0009 Microservices

Status

Wie ist der Status der Entscheidung?

- [ ] vorgeschlagen
- [x] angenommen
- [ ] abgelehnt
- [ ] veraltet
- [ ] ersetzt

Kontext

- Die Lösung besteht aus **Agent-Core** und mehreren **Agenten**.
- Core übernimmt API, und Persistenz; die Agenten enthalten die jeweilige Fachlogik.
- Ziel: lose Kopplung, klare Verantwortlichkeiten, zentrale Observability, unabhängige Releasezyklen.

Entscheidung

- Wir setzen auf eine **Microservice-Architektur** mit klarer Aufgabentrennung:
  - **Agent-Core** als zentraler, stateless Service:
    - stellt REST-API bereit,
    - abstrahiert die Datenbank (Persistenz, Duplikat-Checks, Konsistenz),
    - übernimmt Validierung (DTO/Schemas), Health/Metriken.
  - **Agenten** als eigene Workloads (Deployment oder CronJob):
    - kapseln fachliche Regeln je Domäne,
    - interagieren über den Core (kein direkter DB-Zugriff),
    - können unabhängig versioniert, ausgerollt und skaliert werden.

Folgen

Das wird aufgrund dieser Veränderung **einfacher**:

- **Domänenzuschnitt:** Jede fachliche Logik kapselt ihre Regeln und Datenzugriffe sauber.
- **Skalierung:** Agent-Core kann als stateless API-/Router-Schicht horizontal skaliert werden; Agenten je nach Bedarf separat.
- **Robustheit & Isolation:** Fehler/Lastspitzen in einem Agenten beeinflussen andere nicht.
- **Wartbarkeit:** Klare Schnittstelle und getrennte Deployments; Unterschiedliche Release-Zyklen; schnellere Iteration pro Agent.

Das wird aufgrund dieser Veränderung **schwieriger**:

- **Betrieb:** Mehr Services (Core + mehrere Agenten) erhöhen CI/CD- und Kubernetes-Aufwand.
- **Schnittstellenpflege:** Verträge zwischen Core und Agenten (Versionierung, Rückwärtskompatibilität) müssen aktiv gemanagt werden.
- **Konsistenz über Grenzen:** Verteilte Abläufe erfordern Idempotenz, sauberes Fehler-/Retry-Handling und E2E-Tracing.
- **End-to-End-Tests & Tracing:** Höherer Aufwand für Integrationstests.

0010 Kommunikation via NATS

Status

Wie ist der Status der Entscheidung?

- [ ] vorgeschlagen
- [x] angenommen
- [ ] abgelehnt
- [ ] veraltet
- [ ] ersetzt

Kontext

- Wir brauchen intern sowohl **Request/Reply** (z. B. Abfragen, Trigger) als auch **Fire-and-Forget** (z. B. Telemetrie).
- Das NestJS-Microservices-Modul stellt **kein HTTP/REST** als Transport bereit, dafür Adapter wie NATS, Redis, MQTT, gRPC.
- Ziel: vorhandene Transport-Adapter nutzen, keine eigene Bus-Schicht bauen.

Entscheidung

NATS als interner Transport zwischen Core und Agenten:

- Muster: **Request/Reply**, **Pub/Sub**, **Fire-and-Forget**; **Queue Groups** für Lastverteilung.
- Betrieb: Start mit **NATS Core**; **JetStream** optional für Persistenz/Replay.
- Extern bleibt **REST** am Core; intern NATS-Subjects.

Folgen

Das wird aufgrund dieser Veränderung **einfacher**:

- NATS is leichtgewichtig, portabel und schnell startklar sowohl lokal als auch in der CI/CD.
- Mehrere Kommunikationsmuster ohne Zusatzframework.
- Optionale Persistenz & Replay: Mit JetStream können wir „at-least-once" und Replays aktivieren (schrittweise einführbar).
- Entkopplung und horizontale Skalierung pro Agent.

Das wird aufgrund dieser Veränderung **schwieriger**:

- Betrieb eines zusätzlichen Dienstes bzw Message-Brokers.
- Idempotenz/Retry/Timeouts müssen sauber definiert werden (sofern JetStream aktiv).
- Tracing/Monitoring wird wichtiger, weil mehr asynchron passiert.

0011 Monitoring & Logging mit Promtail, Loki, Prometheus und Grafana)

Status

Wie ist der Status der Entscheidung?

- [ ] vorgeschlagen
- [x] angenommen
- [ ] abgelehnt
- [ ] veraltet
- [ ] ersetzt

Kontext

- Core und Agenten laufen als separate Workloads. Für Support brauchen wir zentrale Sichten auf **Metriken**, **Logs** und **Alerts**.
- Lokale Container-Logs reichen nicht aus, um Vorfälle über mehrere Dienste hinweg zu analysieren.

Entscheidung

- **Metriken/Alerting**: Prometheus (Scrape) + Alertmanager; **Grafana** als UI.
- **Logging**: **Loki** als Log-Backend, **Promtail** als Collector; Abfragen in Grafana.
- Jeder Dienst liefert Basis-Metriken (Erfolg/Fehler, Dauer, erzeugte Leistungen). Alerts z. B. bei ausbleibenden Läufen oder ungewöhnlicher Fehlerquote.

Folgen

Das wird aufgrund dieser Entscheidung **einfacher**:

- Einheitliche Dashboards für Core und Agenten.
- Schnellere Fehlersuche durch kombinierte Metrik-/Log-Sichten.
- Wiederverwendbare Monitoring-Standards für neue Agenten.
- Alerting bei Ausfällen oder Anomalien über verschiedene Kommunikationswege (Email, Slack etc).

Das wird aufgrund dieser Entscheidung **schwieriger**:

- Zusätzliche Infrastruktur (Ressourcen, Updates, Backups, Retention, Pflege/Wartung).
- Sorgfalt bei Log-Inhalten (PII) und Label-Strategie (Cardinality).

0012 Standardisierung der Agenten-Pipeline

Status

Wie ist der Status der Entscheidung?

- [ ] vorgeschlagen
- [x] angenommen
- [ ] abgelehnt
- [ ] veraltet
- [ ] ersetzt

Kontext

- Bisher implementierten Agenten wie `maraton` oder `noshow` persistierten Änderungen direkt mit der `process`-Methode und hatten keine klare Aufteilung von prozessierung, update und create.
- Dadurch war die Logik nicht klar von der Persistenz getrennt, was Testing, Nachvollziehbarkeit und Wiederverwendbarkeit längerfristig erschweren würde.

Entscheidung

Jeder Agent folgt einen einheitlichen Prozess:

- `processItem`: berechnet ausschliesslich den neuen Zustand oder ein Ergebnis-DTO, ohne Persistenz
- `update`: übernimmt die Persistierung und Seiteneffekte auf Basis des Ergebnisses
- `create`: wird aufgerufen, wenn ein neuer Eintrag erzeugt werden muss

Damit sind Berechnung und Persistenz strikt getrennt und das Framework unterstützt dabei einen klaren Denkprozess, ohne zu viele Implementationsdetails zu forcieren.

Folgen

Das wird aufgrund dieser Veränderung einfacher:

- Agenten sind konsistent implementiert
- Tests können die processing-Logik (aus `processItem`) isoliert prüfen, ohne Seiteneffekte
- Persistenz- und Kommunikationsfehler lassen sich klarer lokalisieren

Das wird aufgrund dieser Veränderung schwieriger:

- Höherer Initialaufwand bei neuen Agenten, da immer mehrere Methoden (`processItem`, `update` und `create`) befüllt werden müssen
- Entwickler:innen müssen sich an die strikte Trennung halten und ggf. bestehende Agenten aus dem Legacy-System refaktorisieren
- Neue Agenten-Anforderungen müssen im vorgegebenen Rahmen des Frameworks umsetzbar gestaltet werden

0013 Verwendung von pnpm workspaces und eslint-plugin-boundaries

Status

Wie ist der Status der Entscheidung?

- [ ] vorgeschlagen
- [x] angenommen
- [ ] abgelehnt
- [ ] veraltet
- [ ] ersetzt

Kontext

Das Projekt wächst und umfasst verschiedene Projekte, `core`, `libs` und mehrere `agents`. Diese Projekte werden aktuell alle zentral im root `package.json` verwaltet, was mittelfristig zu Skalierungsproblemen führen wird.

Ohne klare Struktur besteht die Gefahr von unkontrollierten Abhängigkeiten zwischen Modulen (z. B. agentA zu agentB oder libs zum core etc.). Dies soll mit entsprechenden compiler-seitigen Einschränkungen verwaltet werden.

Entscheidung

- Einführung von pnpm workspaces für `core`, `libs` und `agents/*`.
- Verwendung von eslint-plugin-boundaries, um Importregeln festzulegen:
  - libs dürfen nur libs importieren
  - core darf libs + core importieren
  - agents dürfen core + libs + sich selbst importieren (aber keine anderen Agents)
  - niemand darf agents importieren

Folgen

Das wird aufgrund dieser Veränderung einfacher:

- Konsistente, skalierbare Projektarchitektur
- Frühzeitige Validierung von Architekturregeln durch ESLint
- Gemeinsame build-, lint- und test-scripts über alle packages hinweg
- Bessere Wartbarkeit und Erweiterbarkeit, wenn weitere Agents hinzukommen

Das wird aufgrund dieser Veränderung schwieriger:

- Entwickler:innen müssen sich an die Boundaries-Regeln halten
- Leichte Einstiegshürde für neue Teammitglieder:innen (workspaces + boundaries-Konzept verstehen)
- evtl. Anpassungen bestehender Strukturen bei zukünftigen Refactorings notwendig, wenn Regeln verletzt werden

11.4 Use Cases

Name	UC00: Agentenlauf allgemein
Ziel	Das Framework stellt die nötigen Ablaufbausteine bereit, damit ein beliebiger Agent konsistent gestartet, ausgeführt und sauber beendet werden kann.
Betroffene Anforderungen	- FA_REQ1 - FA_REQ3
Vorbedingung	- Agent kann gestartet werden. - Agent-Core ist erreichbar. - Agent-spezifische Konfiguration ist vorhanden/valide. - Der konkrete Agent ist eine Ableitung des `BaseAgent` (oder ähnlich).
Erfolgs-Endbedingung	Für alle **verarbeitbaren** Items wurden die definierten Geschäftsaktionen ausgeführt (**process -> create -> update**). Der Lauf liefert eine zusammengefasste Rückmeldung (*message, success, failed*).
Fehler-Endbedingung	**Kritischer Fehler** (z. B. bereits beim Laden der Items) führt zu Abbruch des Laufs; Fehler wird geloggt und propagiert. **Item-bezogene Fehler** werden gezählt, geloggt und beeinträchtigen die Verarbeitung der übrigen Items nicht.
Akteure	Maintainer
Auslöser	Periodischer Trigger (Cron/Kubernetes) **oder** manuelles Starten. **Ein Prozesslauf pro Trigger**.
Normaler Ablauf	**1) Start:** Agent initialisiert und loggt Start (Kontext, Service-Name). **2) FETCH:** lädt Items aus dem Agent-Core (über Messaging). **3) FILTER:** kann eine Filterlogik anwenden. **4) PROCESS:** Für jedes gefilterte Item wird: - die fachliche Aktion ausgeführt (z. B. erzeugen/ändern) - Fehler pro Item protokolliert und mitzählt; übrige Einträge weiterverarbeitet. **5) RESULT:** Der Agent fasst zusammen: *Items processed, success, failed*.
Alternative Abläufe	- **Keine Items** nach Filter -> „No active items to process" (oder ähnlich). - **Kritischer Fehler** in FETCH -> Lauf bricht mit Fehler ab; Fehler wird geloggt (kritische Meldung). - **Fehler in create/update** je Item -> Item wird als *failed* gezählt, Rest läuft weiter. - **Domänenspezifische Skips** erfolgen in der FILTER-Phase.
Erweiterungen	- **Dry-Run** (nur fetch/filter/process ohne create/update). - Idempotenz/Duplikatschutz (Token/Keys je Item/Batch). - **Metriken/Tracing** (z. B. Zähler für success/failed, Dauer je Phase). - **Konfigurierbare Defaults** (Performer, Kostenstellen, Fenster).
Verwandte Informationen	- UC01: Business Meeting Agent - UC02: Marathon Agent - UC03: Flughafen Grenzkontrolle Agent - UC04: Noshow Agent - UC05: Konfiguration - UC16: Agentenbetrieb starten und stoppen

Comments

Benjamin Thormann @benjamin.thormann · 1 month ago

Owner

Bei Vereinheitlichung der Use Case Nummern und Namen wurde folgende Use Cases mit UC00: Agentenlauf allgemein vereint:

- ehemaliger UC1 Datenselektion durchführen wurde UC00: Agentenlauf allgemein zugeordnet

Ehemalige Use Cases:

ID	Use-Case	Akteur	Ziel
UC1	Datenselektion durchführen	Agent	Der Agent filtert relevante Datensätze vor der Generierung.

Name	UC01: Business Meeting Agent
Ziel	Zwei Executives fehlen bei einem strategischen Business-Meeting. Der Agent erkennt das anhand der Terminstatus und erstellt ein Protokoll für HR oder Geschäftsleitung zur Nachverfolgung.
Betroffene Anforderungen	- FA_REQ12
Vorbedingung	*Agent-Core* ist erreichbar. Ein Business-Meeting mit Pflichtteilnehmenden ist geplant und dokumentiert.
Erfolgs-Endbedingung	Alle abwesenden Teilnehmenden wurden korrekt anhand von ihrem Terminstatus identifiziert und als Terminleistung protokolliert.
Fehler-Endbedingung	- Termin-Stati fehlen oder sind nicht eindeutig - Meeting (Haupttermin) und Teilnahme (Untertermin) können nicht zugeordnet werden
Akteure	Maintainer
Auslöser	Periodischer Lauf (z. B. Cron/Kubernetes Job) **oder** manuelles Auslösen (CLI). Pro Trigger **ein** Prozesslauf.
Normaler Ablauf	1. Meeting-Daten abrufen 2. Erwartete Teilnehmer:innen mit tatsächlicher Teilnahme abgleichen 3. Fehlende Teilnehmer protokollieren 4. Ergebnis an Geschäftsstelle über Alerting weiterleiten
Alternative Abläufe	- Teilnehmer hat abgesagt, wird als abgesagt vermerkt - Teilnehmer hat Terminstatus "unklar" = als nicht teilgenommen gewertet
Erweiterungen	- Eskalationsmeldung bei wiederholtem Fehlen - Auswertung der Fehlleistungen (z.B. Dr. Oetker hat 5x nicht teilgenommen)
Verwandte Informationen	- UC00: Agentenlauf allgemein - Leistung: z.B. `MEET004` - Aktion: z.B. `PARTICIPATED`, `NOSHOW`, `CANCELLED` - Häufigkeit: Periodisch (z.B. wöchentlich, monatlich) - Kann an Abrechnungssysteme und HR-Workflows gekoppelt werden. Optional täglicher Report zur Meeting-Disziplin.

Comments

Benjamin Thormann @benjamin.thormann · 3 months ago — Owner

- Habe eine Frage
  - Use Case: Flughafen Grenzkontrolle Agent. Erweiterung Zweitprüfung bei manuellem Override
  - Entweder bräuchte es noch den Alternativen Ablauf "VIP Reisende:r darf einfach so rein, hat Aufseher der Grenzkontrolle beschlossen"
  - Oder der manuelle Override und Zweitprüfung wären ein weiter womöglicher Use Case, der uns im Moment nicht interessiert.
  - Ich vermute letzteres
- Rest verstehe ich, bin dabei
- Habe ein paar Sätze, so wie ich die Use Cases verstehe, umformuliert:
  - Use Case: Business Meeting Agent
  - "Auslöser Das Meeting findet statt (Trigger über Terminstatus oder geplanter Tageslauf" von allen Pflichtteilnehmenden ")"
  - Use Case: Marathon Agent
  - "Fehlerendbedingung Falls das Rennen offiziell beendet wird (z.B. Unwetter, Verletzungen etc.), bleiben alle" Zwischen"Bewertungen final und es entstehen keine neue Bewertung"

Name	UC02: Marathon-Agent
Ziel	Zwischenstände aktiver Läufer:innen aus neuesten Positionsdaten fortschreiben, Zielerreichung erkennen und Läufer:innen bzw. beendete Rennen finalisieren.
Betroffene Anforderungen	- FA_REQ12
Vorbedingung	*Agent-Core* ist erreichbar. Es existieren Läufer:innen in **laufenden** Rennen. Für einzelne Läufer:innen liegen ggf. aktuelle Positionsangaben (neueste Messung) vor. Verletzte oder abgemeldete Teilnehmende werden im Prozesslauf nicht berücksichtigt.
Erfolgs-Endbedingung	Für **alle aktiven** Läufer:innen wird – falls die neueste gemessene Distanz grösser als die gespeicherte ist – die zurückgelegte Strecke (in km) aktualisiert; bei Erreichen/Überschreiten der Renndistanz wird der Teilnahmestatus auf **beendet** gesetzt. Nach dem Prozesslauf, werden in bereits **beendeten** Rennen alle noch **aktiven** Läufer:innen als **beendet** markiert (Finalisierung).
Fehler-Endbedingung	Ungültige Positionsdaten (z. B. Distanz < 0) führen zu einem Fehler für die betroffene Person; diese wird übersprungen, die Verarbeitung der übrigen Läufer:innen läuft weiter. Fehlen Positionsdaten, erfolgt **keine** Änderung.
Akteure	Maintainer
Auslöser	Periodischer Lauf (z. B. Cron/Kubernetes Job) **oder** manuelles Auslösen (CLI). Pro Trigger **ein** Prozesslauf.
Normaler Ablauf	1) **Aktive** Läufer:innen in **laufenden** Rennen ermitteln. 2) Neueste Position je Person abrufen. 3) **Keine** oder **nicht gestiegene** Distanz -> keine Aktualisierung. 4) **Gestiegene** Distanz -> zurückgelegte Strecke aktualisieren – falls >= Renndistanz zusätzlich Status auf **beendet** setzen. 5) **Finalisierung**: beendete Rennen laden und darin **aktive** Läufer:innen auf **beendet** setzen.
Alternative Abläufe	– Keine aktiven Läufer:innen -> „nichts zu tun". – Läufer:in ohne Positionsmessung -> überspringen (keine Änderung). – Ungültige Positionsangabe (Distanz < 0) -> Fehler loggen, Läufer:in bleibt unverändert, übrige weiterverarbeiten. – Rennen nicht laufend -> Läufer:in im Hauptdurchlauf überspringen (Finalisierung greift nur bei **beendeten** Rennen).
Erweiterungen	-
Verwandte Informationen	UC00: Agentenlauf allgemein

Szenario: Der Agent wird (z. B. alle X Minuten oder manuell) gestartet. Er lädt alle aktiven Läufer:innen. Für jede Person in einem laufenden Rennen holt er die neueste Position:

- Ist die Distanz grösser als die bisher gespeicherte, aktualisiert er die Distanz; bei Zielerreichung setzt er den Status auf beendet.
- Fehlt eine Position oder ist die Distanz nicht gestiegen, passiert nichts.
- Ungültige Positionsdaten werden pro Läufer:in als Fehler geloggt; die übrigen Läufer:innen werden trotzdem verarbeitet. Nach dem Durchlauf finalisiert der Agent beendete Rennen, indem er dort alle noch aktiven Läufer:innen als beendet markiert.

Name	UC03: Flughafen-Grenzkontrolle
Ziel	READY-Einträge gegen aktuelle Einreiseanforderungen des Ziellands evaluieren und den Einreisestatus (GRANTED/REJECTED) mit Begründung setzen.
Betroffene Anforderungen	- FA_REQ12
Vorbedingung	*Agent-Core* ist erreichbar. Es liegen Einträge in der **Einreiseliste** vor, die den Status **READY** haben. Für das Zielland sind **aktuelle Einreiseanforderungen** hinterlegt.
Erfolgs-Endbedingung	Alle READY-Einträge werden **bewertet** und erhalten einen aktualisierten Status (**GRANTED** oder **REJECTED**) samt Begründung.
Fehler-Endbedingung	Ungültige/inkonsistente Daten führen zu einer **Zurückweisung** des betreffenden Eintrags mit **Begründung**; die Verarbeitung der übrigen Einträge läuft weiter.
Akteure	Maintainer
Auslöser	Periodischer Lauf (z. B. Cron/Kubernetes-Job) **oder** manuelles Auslösen. Pro Trigger **ein Prozesslauf**.
Normaler Ablauf	1) **READY**-Einträge abrufen. 2) **Aktuelle Anforderungen** zum Ziel-Land laden (z. B. Visumspflicht, Passpflicht). 3) **Bewerten**: Dokumentgültigkeit prüfen; falls **Passpflicht** -> Pass erforderlich; falls **Visumspflicht** -> Visum erforderlich. 4) **Status setzen**: GRANTED, wenn alle Bedingungen erfüllt; sonst REJECTED inkl. Grund. 5) Ergebnis **im EntryLog** speichern.
Alternative Abläufe	– **Nicht-READY**-Einträge werden **übersprungen** (keine Änderung). – Ungültiges Dokument -> **REJECTED** mit Grund „Document is invalid". – Passpflicht, aber kein Pass -> **REJECTED** („Passport is required but not presented"). – Visumspflicht ohne Visum -> **REJECTED** („Visa is required but not presented").
Erweiterungen	Erweiterte Regeln pro Herkunftsland (z. B. Sonderabkommen), **manueller Override** (führt zu Nicht-READY und damit Ausschluss aus der Automatik), nachgelagerte **Stichprobenprüfung**.
Verwandte Informationen	- UC00: Agentenlauf allgemein - **Entitäten**: Country, EntryRequirement, EntryLog. **EntryLog-Status (technisch)**: READY -> wird bewertet; GRANTED/REJECTED -> Ergebnis; OVERRIDDEN -> manuell, wird übersprungen. **Anforderungen**: `passportRequired`, `visaRequired` (Hinweis: `idCardAccepted` ist zulässig, sofern **keine Passpflicht** besteht).

Szenario: Der Agent startet, lädt alle READY-Einträge und bewertet jeden Eintrag gegen die aktuellen Anforderungen des Ziellands:

- Dokument ungültig -> REJECTED („Document is invalid").
- Passpflicht & kein Pass -> REJECTED.
- Visumspflicht & kein Visum -> REJECTED.

Andernfalls wird GRANTED gesetzt. Nicht-READY-Einträge (z. B. OVERRIDDEN) werden nicht angefasst. Bei Zurückweisungen wird ein Warn-Log geschrieben, die restlichen Einträge werden unabhängig weiterverarbeitet.

Name	UC04: NoShow-Agent
Ziel	Für alle nicht verarbeiteten NOSHOW/NOSHOW_PAY/DONE-Termine je eine Abrechnungsposition erzeugen und den Termin als verarbeitet markieren; SCHEDULED-Termine ignorieren.
Betroffene Anforderungen	- FA_REQ12
Vorbedingung	Agent-Core erreichbar; es existieren Termine mit gültiger Fallreferenz.
Erfolgs-Endbedingung	Für **alle nicht verarbeiteten** Termine im Status **NOSHOW**, **NOSHOW_PAY** oder **DONE** wird **genau eine** Abrechnungsposition erzeugt und der Termin als **verarbeitet** markiert. Termine im Status **SCHEDULED** bleiben unberührt.
Fehler-Endbedingung	Kann für einen Termin **keine** gültige Position abgeleitet/erstellt werden (z. B. unbekannter Status, fehlende Fallnummer o. ä.), wird dieser Termin **übersprungen**, protokolliert und die Verarbeitung der übrigen Einträge **fortgesetzt**.
Akteure	Maintainer
Auslöser	**Periodisch** (Cron/Kubernetes Job) **oder manuell** Container-Start. Pro Trigger **ein Prozesslauf**.
Normaler Ablauf	1) Alle Termine laden. 2) **Filtern**: geplante Termine (in der Zukunft/SCHEDULED), bereits **verarbeitete** oder ohne Fallreferenz werden übersprungen. 3) **Mapping** nach Terminstatus: - `NOSHOW` -> Chargeltem `code = "NOSHOW"`, *kostenlos*. - `NOSHOW_PAY` -> `code = "NOSHOW_FEE"`, *kostenpflichtig* (mit Kostenstelle). - `DONE` -> `code = "REGULAR_VISIT"` (regulär, mit Kostenstelle). 4) **Erstellen** der Abrechnungsposition pro Termin. 5) Termin updaten als verarbeitet markieren.
Alternative Abläufe	– **Keine verarbeitbaren** Termine -> Meldung „nichts zu tun". – **Unbekannter** Terminstatus -> Warn-Log, Termin bleibt unverändert (zählt als *failed*), Rest wird weiterverarbeitet. – **Fehlende Fallreferenz** -> Warn-Log, Termin übersprungen. – **Erstellung fehlgeschlagen** -> Fehler geloggt, Verarbeitung der übrigen Einträge geht weiter.
Erweiterungen	(geplant/optional) **Zeitfenster** für Selektion (z. B. T-n…T-1), **Idempotenz/Duplikat-Vermeidung** (z. B. Schlüssel je Termin), **Chunking** grosser Datenmengen, **konfigurierbare Standardwerte** (Performer/Kostenstelle), **Dry-Run**.
Verwandte Informationen	- UC00: Agentenlauf allgemein - Terminstatus (Eingang): `NOSHOW`, `NOSHOW_PAY`, `DONE`, `SCHEDULED` (wird übersprungen). Erzeugte Codes (Ausgang): `NOSHOW`, `NOSHOW_FEE`, `REGULAR_VISIT`. Nach Verarbeitung: Terminflag `isProcessed = true`.

Szenario: Gestern existierten drei Termine: A = NOSHOW, B = NOSHOW_PAY, C = DONE; morgen ein Termin D = SCHEDULED. Beim Lauf erzeugt der Agent drei Abrechnungspositionen (für A/B/C) und markiert diese Termine als verarbeitet; D bleibt unverändert.

- Ohne medicalCaseId wird ein Termin nicht verarbeitet (Warn-Log).
- Bei unbekanntem Status wird gewarnt und nur dieser Eintrag verworfen; der Rest läuft weiter.

Name	UC05: Konfiguration
Ziel	Dienste (Agent-Core und Agenten) lesen ihre Laufzeit-Konfiguration einheitlich, sicher und nachvollziehbar ein. Fehlkonfigurationen werden früh erkannt.
Betroffene Anforderungen	- FA_REQ3 - FA_REQ4 - FA_REQ6
Vorbedingung	Konfigurationswerte sind bereitgestellt (z. B. Env-Variablen / .env-Dateien je Modul).
Erfolgs-Endbedingung	Der Dienst startet mit **validierter** Konfiguration; alle benötigten Werte sind vorhanden oder durch definierte Defaults abgedeckt.
Fehler-Endbedingung	Bei **ungültiger** oder **inkonsistenter** Konfiguration bricht der Start **deterministisch** ab; die Fehlursache ist im Log eindeutig ersichtlich. Teilstarts mit „halb gültiger" Konfiguration finden nicht statt.
Akteure	Maintainer
Auslöser	Dienststart (z. B. Cron/Kubernetes, manuell, CI/CD-Pipe)
Normaler Ablauf	1) Konfiguration laden. 2) **Validieren** gegen ein **Schema** (Typen, Pflichtfelder, erlaubte Wertebereiche).
Alternative Abläufe	– **Konfigurationsdatei fehlt** -> Abbruch mit Hinweis. – **Pflichtwert fehlt/Typfehler** -> Validierung schlägt fehl -> **Abbruch**; Log enthält konkrete Variable und erwarteten Typ/Bereich. – **Unbekannte Variablen** -> ignorieren oder warnen, ohne Start zu verhindern. – **Testbetrieb** -> Validierungsumfang kann bewusst reduziert werden.
Erweiterungen	– **Dynamische Neu-Ladung** bei Änderung. – **Konfigurations-Telemetry** (z. B. Metrik „config_loaded_ok", „config_validation_failed").
Verwandte Informationen	- UC00: Agentenlauf allgemein - UC10: Generierungsintervall intern konfigurieren **Konfigurationsdomänen**: - **Allgemein** (z. B. Umgebung, Log-Level, Service-Name, Broker-URL) - **Agent-Core-spezifisch** (z. B. DB-Zugriff) - **Agent-spezifisch** (z. B. Filter/Zeitfenster, Defaults).

Comments

Benjamin Thormann @benjamin.thormann · 1 month ago — Owner

Bei Vereinheitlichung der Use Case Nummern und Namen wurde folgende Use Cases mit UC05: Konfiguration vereint:

- ehemaliger UC04 Bootstrap ist gleich UC05: Konfiguration
- ehemaliger UC4 Export-Strategie konfigurieren wurde UC05: Konfiguration zugeordnet und damit aufgelöst
- ehemaliger UC5 Standard-Logiken konfigurieren wurde UC05: Konfiguration zugeordnet und aufgrund von 0007 Fachliche Kriterien gestrichen

Ehemalige Use Cases:

ID	Use-Case	Akteur	Ziel
UC04	Bootstrap		
UC4	Export-Strategie konfigurieren	Systemadministrator / Fachpersonal	Der Administrator wählt zwischen Sofortexport oder Export nach ALIS-Logik.

UC5	Standard-Logiken konfigurieren	@all wer ist hier der Akteur?	@all bestimmen! **Kontext:** *«Logiken für Standard-Leistungsinformationen müssen einrichtbar sein, z.B. zur Ermittlung eines Leistungserbringer, einer leistungserbringenden Kostenstelle» à Was ist hier fachlich gemeint? Das hatten wir leider nicht ganz verstanden. Beziehungsweise, wie soll das in einem CISTEC-Agnostischen Framework implementiert werden?* **Martina**: *Mir fehlt es etwas schwer, die Leistungsbrille abzulegen... Vielleicht wird es so verständlicher: Wichtige Attribute für eine Leistungssitzung müssen unabhängig von der Logik pro Kunde individuell einstellbar sein. Beispiel erbringende Kostenstelle beim Medi-Leistungs-Agenten. Hier soll der Kunde definieren können, ob die erbringende Kostenstelle aus dem Patientenfall ermittelt wird oder aus der erbringenden Organisationseinheit wo der dokumentierende User zugehörig ist.*

Edited 1 month ago by Benjamin Thormann

Name	UC06: Duplikatcheck
Ziel	Durch Idempotentes Persistieren wird sichergestellt, dass der Agent Core beim mehrfachen Empfang eines Agentenergebnisses diese als ein Ergebnis verarbeitet.
Betroffene Anforderungen	FA_REQ2
Vorbedingung	- Agent hat eine Batch-Payload erzeugt - Jeder Eintrag enthält einen Idempotenz-Key - dedizierte Tabelle für processed-keys
Erfolgs-Endbedingung	- Für alle Einträge, deren Idempotenz-Key noch nicht existieren, wurden Geschäftsobjekte erzeugt und in der DB gespeichert. - Bereits vorhandene Keys wurden übersprungen; Agent erhält Liste welche übersprungen wurden.
Fehler-Endbedingung	- Doppelte Einträge in der DB
Akteure	Maintainer
Auslöser	Agent sendet POST /results an Agent-Core
Normaler Ablauf	1. Für jedes result prüft Core auf Tabelle processed_keys: - Key vorhanden -> skippedKey.add - Key neu -> speichert Entität und fügt key in tb-processed_keys 2. Core Antwortet mit HTTP-201 und mit liste 'created' und 'skipped' 3. Agent loggt und fährt fort
Alternative Abläufe	- Kein proccessed_key mitgesendet - Anfrage wird abgewiesen Http-4xx
Erweiterungen	
Verwandte Informationen	Idempotency: https://osamadev.medium.com/idempotency-in-apis-making-sure-api-requests-are-safe-and-reliable-7d5cb51520fe

Comments

Benjamin Thormann @benjamin.thormann · 1 month ago

Owner

Bei Vereinheitlichung der Use Case Nummern und Namen wurden folgende Use Cases mit UC06: Duplikatcheck vereint:

- ehemaliger UC2 Duplikate erkennen und ausschliessen wurde UC02: Marathon Agentzugeordnet

Ehemalige Use Cases:

ID	Use-Case	Akteur	Ziel
UC2	Duplikate erkennen und ausschliessen	Agent-Core	Der Agent prüft, ob Leistungen bereits existieren und vermeidet Dopplungen.

Name	**UC07: Agentenlauf manuell auslösen**
Ziel	Maintainer möchte einen Agent gezielt manuell starten und nicht auf das scheduling des Cronjobs warten.
Betroffene Anforderungen	- FA_REQ3
Vorbedingung	*Kubernetes-Cluster verfügbar*, Fleet-Deployment angewendet (CronJobs existieren: `noshow-agent`, `marathon-agent`, `grenzkontrolle-agent`), Agent-Core + NATS + DB laufen, `kubectl`-Zugriff vorhanden, Agent-Images in Registry
Erfolgs-Endbedingung	Job pro ausgewähltem Agent wurde **einmalig** gestartet und **erfolgreich abgeschlossen**; Agent hat verarbeitet, Ergebnisse sind via Agent-Core in der DB persistiert; Logs des Laufs sind abrufbar.
Fehler-Endbedingung	Job/POD startet nicht oder endet **nicht erfolgreich**; es wurden **keine** Ergebnisse persistiert; Fehlermeldungen sind im Pod-/Job-Log ersichtlich; Skript beendet sich mit Fehler.
Akteure	Maintainer
Auslöser	Maintainer stösst den Lauf manuell an, über bereitgestellte Skript.
Normaler Ablauf	1. Maintainer startet `scripts/fleet/start-cronjob.sh`. 2. Das Skript erstellt je Agent einen Job aus dem bestehenden CronJob. 3. Normaler Agent-Lauf wie im UC00: Agentenlauf allgemein beschrieben 4. Agent-Core persistiert Ergebnisse; Skript zeigt Live-Logs des Jobs. 5. Job erreicht Complete; Skript beendet sich erfolgreich.
Alternative Abläufe	- **Timeout**: Job wird nicht complete innerhalb der Frist -> Skript endet mit Fehler. - **Fehlstart**: CronJob/Job-Name existiert nicht (falscher Namespace/Agent) -> kubectl-Fehler, Skript bricht ab. - **Abhängigkeit down**: Agent-Core/NATS/DB nicht erreichbar -> Agent meldet Fehler, Job endet failed. - **Agent-Fehler**: Domänenseitige Fehler führen zu failed-Zählern pro Item; der Job kann trotzdem overall erfolgreich sein.
Erweiterungen	**Docker-only-Run**: Alternativ lokal per docker run (One-Shot), wenn kein Cluster benötigt wird.
Verwandte Informationen	UC16: Agentenbetrieb starten und stoppen

Name	UC08: Agent seit X Tagen keine Leistung mehr erzeugt
Ziel	Reporter möchte benachrichtigt werden, sobald ein Agent nach X Tagen keine Leistung erzeugt hat.
Betroffene Anforderungen	FA_REQ8
Vorbedingung	- jeder Agent publiziert nach jedem Lauf eine Metrik - Alert-Regel und Benachrichtigungskanal sind eingerichtet
Erfolgs-Endbedingung	- Alert wird ausgelöst - Man kann die Ursache im Log nachvollziehen
Fehler-Endbedingung	- Alert wird nicht ausgelöst - Ursache nicht nachvollziehen - keine Logs erstellt
Akteure	Reporter
Auslöser	Schwellwert von X Tage überschritten
Normaler Ablauf	1. Monitoring System wertet Regel einmal täglich aus 2. Bedingung `> x Tage` trifft zu 3. Monitoring-System löst Alert aus- zB Email an Subscriber mit Meldung und Log 4. Reporter erhält Meldung mit betroffenen Agent und Log 5. Nach Fix erzeugt nächster Agent-Lauf wieder Leistungen (-> Metrik wird aktualisiert) 6. Monitoring-System löst kein erneutes Alert aus
Alternative Abläufe	1a. Monitoring System erkennt (bewusste) inaktivität des Agenten - Alert wird nicht ausgelöst 1b. Monitoring-System ausser Betrieb - wird nicht behandelt 3. keine Subscriber vorhanden - Subscriber werden nicht validiert (zb keine gültige Email) - wird nicht behandelt bzw. Alert wird ignoriert
Erweiterungen	
Verwandte Informationen	- UC09: Fehlerhafte Leistungsgenerierung protokollieren - UC14: Logging einsehen

Comments

Benjamin Thormann @benjamin.thormann · 1 month ago — Owner

Bei Vereinheitlichung der Use Case Nummern und Namen wurde folgende Use Cases mit UC08: Agent seit X Tagen keine Leistung mehr erzeugt vereint:

- ehemaliger UC10 Alerting bei fehlender Leistungserzeugung wurde UC05: Konfiguration zugeordnet

Ehemalige Use Cases:

ID	Use-Case	Akteure	Ziel
UC10	Alerting bei fehlender Leistungserzeugung	Monitoring-System	Das System meldet automatisch, wenn über einen Zeitraum keine Leistungen mehr erzeugt werden.

Name	UC09: Fehlerhafte Leistungsgenerierung protokollieren
Ziel	Der Maintainer oder Reporter möchte fehlerhafte Leistungen protokolliert haben, um diese Analysieren zu können
Betroffene Anforderungen	FA_REQ3
Akteure	Maintainer, Reporter
Verwandte Informationen	- UC08: Agent seit X Tagen keine Leistung mehr erzeugt - UC15: Fehlerlogs einsehen

Name	UC10: Generierungsintervall intern konfigurieren
Ziel	Der Maintainer setzt das Generierungsintervall über Konfigurationsdateien.
Betroffene Anforderungen	FA_REQ6
Akteure	Maintainer
Verwandte Informationen	UC05: Konfiguration

Name	UC12: Leistungen ohne Stopdatum generieren
Ziel	Der Agent generiert Leistungen auch bei fehlendem Enddatum der Massnahme / des Marathon-Rennens
Betroffene Anforderungen	FA_REQ7
Akteure	Maintainer

Name	UC13: Anpassung bei Datenstrukturänderungen
Ziel	Der Agent verarbeitet Daten weiterhin korrekt bei abwärtskompatiblen Schemaänderungen (z. B. zusätzliche optionale Felder, neue Spalten, geänderte Reihenfolge).
Betroffene Anforderungen	NFA_REQ10
Akteure	Maintainer

Comments

Jvan Fadda @jvan.fadda · just now Author Owner

hier haben wir lange **"Der Agent bleibt bei moderaten Datenstrukturänderungen funktionstüchtig."** als Ziel stehen gehabt.

Da das Ziel unklar und **nicht** Messbar ist, und bis heute keine genauere Spezifikation seitens Cistec gekommen ist, haben wir dieses Ziel definiert, was implizit erfüllt ist bzw. nicht weiter darauf eingegangen worden ist.

Name	UC14: Logging einsehen
Ziel	Der Maintainer kann Logs einsehen, um Systemzustände und Fehler zu analysieren.
Betroffene Anforderungen	- FA_REQ3 - NFA_REQ5
Vorbedingung	Logger eingebunden; Notwendige Env gesetzt (z.B. `SERVICE_NAME`, `LOG_LEVEL`, `NODE_ENV`)
Erfolgs-Endbedingung	- Während der Laufzeit entstehen farbige, menschlich lesbare **Console-Logs** und **JSON-File-Logs** unter `./logs/<SERVICE_NAME>.log` (mit Timestamp, Stacktraces, strukturierte Felder). - Fehler werden mit Stacktrace geloggt. - In test entstehen keine Logfiles. - Messages enthalten Service-Kontext (zb `noshow`, `marathon`).
Fehler-Endbedingung	Fehlende Pflicht-Env (z. B. `SERVICE_NAME`, `LOG_LEVEL`) -> Start bricht ab.
Akteure	Maintainer
Auslöser	Service/Agent startet; Code schreibt Logs (info/warn/error/debug...).
Normaler Ablauf	1) Logger injizieren. 2) Fachlich korrekt loggen: `info` für Business-Events, `warn` für abweichende, aber erwartbare Fälle, `error` mit Stack-Trace, `debug` für detailierte Ablaufinfos.
Alternative Abläufe	`NODE_ENV=test` -> nur **Console-Transport** (kein File).
Erweiterungen	-
Verwandte Informationen	- UC15: Fehlerlogs einsehen - UC08: Agent seit X Tagen keine Leistung mehr erzeugt - Console: Nest-like Format (farben, pretty). File: JSON + Rotation(max-size ~20MB, max-Files 10). Dateipfad: `./logs/<SERVICE_NAME>.log`. Standardisierte, konsistentes, strukturiertes Logging, das plug-and-play für neue Services/Agenten genutzt werden kann.

Name	UC15: Fehlerlogs einsehen
Ziel	Der Maintainer kann zentral Logs einsehen und durchsuchen, um Fehler der Agentenleistung zu analysieren (z.B.: bei nicht erfolgreicher Leistungsgenerierung)
Betroffene Anforderungen	- FA_REQ10 - NFA_REQ5
Vorbedingung	Stack aktiv (Promtail, Loki, Grafana): **promtail** liest `./logs`, sendet an **Loki**; **Grafana** ist provisioniert (automatisch vorkonfiguriert). Services (Agent-Core/Agents) schreiben nach `./logs`.
Erfolgs-Endbedingung	- Logs erscheinen in Grafana innerhalb einer Minute. - Logs sind in Grafana filterbar nach Service-Name & Level (z. B. nach `service=<noshow>`) (mindestens wenn mit `docker-compose` gestartet) - Rotation verhindert Dateien ohne Grössen-/Anzahlbegrenzung. - Level-Änderungen greifen nach Neustart.
Fehler-Endbedingung	Keine Logs auf Grafana sichtbar
Akteure	Maintainer
Auslöser	Betriebsstart
Normaler Ablauf	1) Env prüfen: `SERVICE_NAME`, `LOG_LEVEL`. 2) Files erscheinen unter `./logs/*.log` (ausser wenn `test`). 3) promtail versendet Logs nach Loki und erscheinen auf Grafana 4) Bei Bedarf **Log-Level** per Env ändern und neu starten.
Alternative Abläufe	– Umgebung `test`: **keine File-Logs**. – Rechte/Owner-Probleme im `./logs`-Volume -> promtail kann nicht lesen -> fixen und neu laden. – `LOG_LEVEL=error` -> geringe Sichtbarkeit –> Level erhöhen.
Erweiterungen	Alerts aus Logs (Loki-Rules), Dashboards pro Service/Use-Case, Log-basierte Metriken (Error-Rate), mandantenfähige Labels
Verwandte Informationen	- UC09: Fehlerhafte Leistungsgenerierung protokollieren - UC14: Logging einsehen - docker-compose: **loki**, **promtail**, **grafana**; Agents/Agent-Core mounten `./logs`. Rotation lokal durch Winston

Name	UC16: Agentenbetrieb starten und stoppen
Ziel	Der Maintainer kann Agenten starten und stoppen (aktivieren/deaktivieren).
Betroffene Anforderungen	FA_REQ11
Akteure	Maintainer
Verwandte Informationen	- UC00: Agentenlauf allgemein - UC07: Agentenlauf manuell auslösen

Name	UC17: Leistungsgenerierung auf Entitäten beschränken
Ziel	Der Maintainer kann die Agentenleistung auf spezifische Entitäten einschränken (über PID/FID).
Betroffene Anforderungen	FA_REQ12
Akteure	Maintainer

11.5 Produkt-Backlog

Title	Issue ID	State	Author	Assignee	Milestone	Labels
Gitlab aufsetzen	1	Closed	Jvan Fadda	Guillaume Fricker	Projektsetup	
Typescript Framework evaluieren	2	Closed	Jvan Fadda	Benjamin Thormann	Requirements Engineering	
Domain model	3	Closed	Jvan Fadda	Jvan Fadda	Requirements Engineering	documentation
Ziele(MUSS, KANN, SOLL) vereinbaren	4	Closed	Jvan Fadda	Jvan Fadda	Requirements Engineering	documentation
IST-Situation analysieren	5	Closed	Jvan Fadda	Guillaume Fricker	Requirements Engineering	
Systemerfassung und Abgrenzung definieren	6	Closed	Jvan Fadda	Benjamin Thormann	Requirements Engineering	
Terminplan erstellen	7	Closed	Jvan Fadda	Jvan Fadda	Requirements Engineering	
CI/CD	8	Closed	Jvan Fadda	Guillaume Fricker	Projektsetup	
Initiales Backend-Gerüst mit Datenbankintegration	9	Closed	Jvan Fadda	Guillaume Fricker	Architekturprototyp	
Durchstich: DB-Anbindung mit ORM (TypeORM + PostgreSQL) testen	11	Closed	Jvan Fadda	Jvan Fadda	Architekturprototyp	
Architektur skizzieren	12	Closed	Jvan Fadda	Benjamin Thormann	Architekturprototyp	documentation
Use Case erfassen	13	Closed	Jvan Fadda	Jvan Fadda	Requirements Engineering	documentation
Code Of Coduct erstellen	14	Closed	Jvan Fadda	Jvan Fadda	Projektsetup	documentation
Dokumente "Anforderungen Agenten" durcharbeiten	16	Closed	Benjamin Thormann	Benjamin Thormann	Requirements Engineering	documentation
ADR erstellen	17	Closed	Benjamin Thormann	Benjamin Thormann	Projektsetup	documentation
ADR-Eintrag Typescript Framework	18	Closed	Benjamin Thormann	Benjamin Thormann	Requirements Engineering	documentation
Demo-Daten/Testmandant zur Verfügung stellen	19	Closed	Jvan Fadda	Guillaume Fricker	Architekturprototyp	
Risikoanalyse erstellen	20	Closed	Benjamin Thormann	Guillaume Fricker	Requirements Engineering	
estimate abmachen	21	Closed	Benjamin Thormann	Benjamin Thormann	Projektsetup	
Architekturdokumentation Packages	22	Closed	Benjamin Thormann	Benjamin Thormann	Architekturprototyp	documentation
Architekturdokumentation Version Fallback	23	Closed	Benjamin Thormann	Benjamin Thormann	Requirements Engineering	documentation
Architekturdokumentation Framework-Prototyp	24	Closed	Benjamin Thormann	Benjamin Thormann	Architekturprototyp	documentation
Architekturdokumentation Zod	25	Closed	Benjamin Thormann	Guillaume Fricker		
Simple NoShow Agent	26	Closed	Jvan Fadda	Jvan Fadda	Architekturprototyp	
Logging-Konfiguration zentral steuerbar machen	27	Closed	Jvan Fadda	Jvan Fadda	Implementation	
E2E-Tests: Automatischer DB-Rollback pro Test	28	Closed	Jvan Fadda		Implementation	
CI/CD extended	29	Closed	Jvan Fadda	Guillaume Fricker	Implementation	
Agent-Core API Versionierung	30	Closed	Jvan Fadda		Implementation	
Core Messaging Queue Integration	31	Closed	Jvan Fadda		Implementation	
Core: Duplikatsprüfung auf Dokumentationsbasis	32	Open	Jvan Fadda	Benjamin Thormann	Implementation	
Agent: OAT-Agent Logik	33	Closed	Jvan Fadda		Implementation	
Agent: Anästhesie-Nachweis-Parsing	34	Closed	Jvan Fadda		Implementation	
API: Healthcheck-Endpoint für Agentenlauf	35	Closed	Jvan Fadda	Jvan Fadda	Implementation	
Config: Konfigurationsmodul für Agenten	36	Closed	Jvan Fadda	Jvan Fadda	Implementation	
Config: Toggle für Agent-Läufe	37	Open	Jvan Fadda		Implementation	
CORE: Zod-Schema verbessern durch Generics	38	Open	Jvan Fadda		Architekturprototyp	
Domänenspezifische Begriffe eindeutschen	39	Closed	Jvan Fadda		Architekturprototyp	
Monitoring: Logging-System einführen	40	Closed	Jvan Fadda	Jvan Fadda	Implementation	
Monitoring: Zentralisiertes Logging mit Loki & Grafana	41	Closed	Jvan Fadda	Jvan Fadda	Implementation	
Agent-Core und NoShow-Agent auf Docker bringen 🐳	42	Closed	Guillaume Fricker	Guillaume Fricker	Implementation	
Agent-Core mit NoShow Agent als Microservices implementieren	44	Closed	Benjamin Thormann	Jvan Fadda	Architekturprototyp	
Agent Marathonläufer	45	Closed	Benjamin Thormann	Jvan Fadda	Implementation	
Agent Flughafengrenzkontrolle	47	Closed	Benjamin Thormann	Guillaume Fricker		
Agent Businessmeeting	48	Closed	Benjamin Thormann	Benjamin Thormann	Implementation	
Agent für Use Case Nachgenerierung	49	Closed	Benjamin Thormann		Implementation	
Use Case Nachgenerierung schreiben	50	Closed	Benjamin Thormann	Guillaume Fricker		
Linter aufsetzen für statische code analyse	51	Closed	Guillaume Fricker	Guillaume Fricker		
Follow-up from "Refactor: Auftrennung in Microservices – Core & Noshow-Agent via NATS"	52	Closed	Jvan Fadda	Guillaume Fricker		
Follow-up from "Refactor: Auftrennung in Microservices – Core & Noshow-Agent via NATS"	53	Closed	Jvan Fadda	Jvan Fadda	Implementation	
Follow-up from "Refactor: Auftrennung in Microservices – Core & Noshow-Agent via NATS"	54	Closed	Jvan Fadda			
Follow-up from "Refactor: Auftrennung in Microservices – Core & Noshow-Agent via NATS"	55	Closed	Jvan Fadda	Jvan Fadda		
Follow-up from "Refactor: Auftrennung in Microservices – Core & Noshow-Agent via NATS"	56	Closed	Jvan Fadda	Jvan Fadda		
Follow-up from "Refactor: Auftrennung in Microservices – Core & Noshow-Agent via NATS"	57	Closed	Jvan Fadda	Guillaume Fricker		
Follow-up from "Refactor: Auftrennung in Microservices – Core & Noshow-Agent via NATS"	58	Closed	Jvan Fadda			
Config: Environments bereinigen und vereinheitlichen	59	Closed	Jvan Fadda	Jvan Fadda	Implementation	
Agent Flughafengrenzkontrolle als Microservice implementieren	60	Closed	Benjamin Thormann	Benjamin Thormann	Implementation	
Implementation Use Case Duplikatcheck	61	Closed	Benjamin Thormann	Benjamin Thormann	Implementation	
Follow-up from "Refactor: Auftrennung in Microservices – Core & Noshow-Agent via NATS"	62	Closed	Jvan Fadda	Guillaume Fricker		
Follow-up from "Refactor: Auftrennung in Microservices – Core & Noshow-Agent via NATS"	63	Closed	Jvan Fadda	Jvan Fadda	Implementation	

Follow-up from "Refactor: Auftrennung in Microservices – Core & Noshow-Agent via NATS"	64	Closed	Jvan Fadda	Jvan Fadda		💀
Follow-up from "Refactor: Auftrennung in Microservices – Core & Noshow-Agent via NATS"	65	Closed	Jvan Fadda			💀
Follow-up from "Refactor: Auftrennung in Microservices – Core & Noshow-Agent via NATS"	66	Closed	Jvan Fadda	Jvan Fadda		💀
Follow-up from "Refactor: Auftrennung in Microservices – Core & Noshow-Agent via NATS"	67	Closed	Jvan Fadda	Jvan Fadda	Implementation	🍓,🍓.
linter: pre-commit hook	68	Closed	Jvan Fadda	Guillaume Fricker		🍓,🍓.
Follow-up from "agent-marathon" Simulation in Test-Suite bringen	69	Closed	Jvan Fadda	Jvan Fadda	Implementation	🍓,🍓.
Follow-up from "agent-marathon" DTO/ Schema Naming	70	Closed	Jvan Fadda	Jvan Fadda		💀
Follow-up from "agent-marathon" DTO in der Persistenzschicht?	71	Closed	Jvan Fadda	Jvan Fadda		💀
Anforderungen (NFAs und FAs) überarbeiten	72	Closed	Guillaume Fricker	Benjamin Thormann	Requirements Engineering	🍓,🍾.
E2E-Tests stabilisieren nach Umstellung von Monolith auf Microservices	73	Closed	Jvan Fadda	Jvan Fadda	Implementation	🍏,🍾.
skipped test E2E Grezkontrolle	74	Closed	Jvan Fadda	Guillaume Fricker	Implementation	bug,💀
Core E2E test adaptieren mit TestContainer	75	Closed	Jvan Fadda	Jvan Fadda	Implementation	bug,🍓,🍓.
Follow-up from "E2E für Microservice-Architektur" Generische E2E Test für Agenten	76	Closed	Jvan Fadda	Benjamin Thormann	Implementation	💀
Anforderungen und Use Cases verbessern	77	Closed	Benjamin Thormann	Benjamin Thormann	Zwischenreview mit Betreu	🍓,🍾.
Stakeholder-Analyse	78	Closed	Benjamin Thormann	Benjamin Thormann	Requirements Engineering	🍓,🍾.
Prozess und Persistierung sauber trennen	79	Closed	Jvan Fadda	Guillaume Fricker	Implementation	🍾,🍾.
Fleet Chart ausbauen	80	Closed	Benjamin Thormann	Benjamin Thormann	Implementation	💀
`.env.test` einführen und `env.validation.ts` verfeinern	81	Closed	Benjamin Thormann			💀
Fleet-Config Modul für zentralisierte Agenten-Konfiguration	82	Closed	Jvan Fadda	Jvan Fadda	Implementation	🍾,🍾.
Fleet-Config Gitops Reviewen	83	Closed	Jvan Fadda	Benjamin Thormann	Implementation	🍏,🍏.
Setup Readme	84	Closed	Benjamin Thormann	Guillaume Fricker	Feature-Freeze & Finalisieru	🍓,🍏.
Noshow verarbeitet Termine mit Status SCHEDULED	85	Closed	Jvan Fadda	Jvan Fadda	Feature-Freeze & Finalisieru	bug,🍓,🍓.
Zentrales, konsistentes Logging im gesamten Monorepo	86	Closed	Jvan Fadda	Jvan Fadda	Feature-Freeze & Finalisieru	🍏,🍾.
Core-client provider clean up	87	Closed	Jvan Fadda	Jvan Fadda	Feature-Freeze & Finalisieru	🍓,🍓.
Skalierbare Projektarchitektur mit pnpm workspaces und import-Regelung	88	Closed	Guillaume Fricker	Guillaume Fricker		🍏,🍏.
Clean-up: diverses	89	Closed	Jvan Fadda	Jvan Fadda	Feature-Freeze & Finalisieru	🍓,🍓.
TypeOrm in eigenes DatabaseModule auslagern	90	Closed	Jvan Fadda	Jvan Fadda	Feature-Freeze & Finalisieru	🍓,🍓.
Environment Schema Validierung für die Agenten.	91	Closed	Jvan Fadda		Feature-Freeze & Finalisieru	🍓,🍓.

11.6 Code of Conduct

Unser Entwicklungs-Team arbeitet gemäss folgenden Verhaltensregeln, die von allen Entwicklern einzuhalten sind, um die Zusammenarbeit möglichst reibungslos sicherzustellen:

Allgemeine Grundsätze und ethisches Verhalten:

1. Wir behandeln einander mit Respekt und kommunizieren höflich und konstruktiv.
2. Jedes Teammitglied erfüllt seine Aufgaben termingerecht, sorgfältig und informiert das Team bei Verzögerungen.
3. Wir fördern eine Kultur des offenen Austauschs von Feedback und Ideen.
4. Wir nutzen die Stärken jedes Einzelnen, treffen Entscheidungen gemeinsam und unterstützen uns gegenseitig.
5. Wir legen Wert auf lesbare, wartbare sowie einheitliche Programmierung.
6. Codebeiträge werden vor der Integration gründlich getestet und einem Review unterzogen - "Qualität geht vor Quantität".
7. Wir nehmen Code Reviews ernst und geben konstruktives Feedback.
8. Wir teilen Wissen und Erfahrung innerhalb des Teams.
9. Alle Entwicklungsarbeiten werden ausführlich dokumentiert, aktuell gehalten, und ist für alle Teammitglieder zugänglich.

Workflow und Branching-Strategie Grundsätze:

1. Wir synchronisieren vor jedem Arbeitsbeginn unseren Arbeitsbereich mit dem Repository.
2. Wir committen und pushen regelmässig in möglichst kurzen Abständen implementierte Arbeitseinheiten, die erfolgreich kompilieren und getestet sind.
3. Die Behebung eines defekten Builds hat oberste Priorität. Nicht funktionsfähige Änderungen werden rückgängig gemacht.
4. Entwickler, die den Build brechen, sind verantwortlich für dessen Reparatur. Dabei unterstützen die Teammitglieder sich gegenseitig.
5. Ein gebrochener Build muss analysiert, den Grund ermitteln und Massnahmen ergriffen werden, die sicherstellen, dass dieser Fehler in Zukunft vermieden wird.
6. Zuständigkeiten für Bugfixes werden innerhalb des Teams kommuniziert.
7. Branching-Konventionen und Strategie:
    - Wir verwenden Trunk-Based Development: Unser Hauptzweig heisst main und bildet direkt den Entwicklungszweig. Es gibt keinen eigenen develop-Branch, keine Hotfix-Branches und keine Release-Branches. Änderungen werden direkt über Feature- oder Bugfix-Branches in main integriert.
    - Für die Benennung der Branches folgen wir einer klaren Konvention: feat/ für neue Features und fix/ für Bugfixes. Release-Branches werden nicht benötigt. Versionen werden direkt über Tags auf dem main-Branch markiert.
    - Für Änderungen, die ausschliesslich die Dokumentation betreffen, verwenden wir den Prefix 'doc/', gefolgt von einem spezifischen Thema, z.B. 'doc/code-of-conduct', um die CI-Infrastruktur nicht unnötig zu belasten.
    - Für jeden neuen Feature, Bugfix oder Release wird ein eigener Branch erstellt, deren Namen beschreibend sind und den Zweck klar kommunizieren (z.B. 'feat/add-user-login', fix/login-issue').
    - Branch-Namen sind in Kleinbuchstaben zu halten und Wörter durch Bindestriche zu verbinden.
    - Commit-Nachrichten sollten klar und präzise sein und den Zweck des Commits in kurzer Form erklären.
    - Wir wenden kein Rebase oder ähnliches an, welche die Historie der Commits verändert.
    - Wir streben nach kurzen Lebenszyklen für Feature-Branches.
    - Grosse Features werden in kleinere Features unterteilt und iterative - möglichst oft - in den Develop-Branch integriert.
    - Ein Feature wird als abgeschlossen betrachtet, wenn:
        - das Projekt ohne Fehler gebuildet (Clean-Build) werden kann und alle Funktionalitäten mit Unit-Tests geprüft wurden.
        - die dazugehörige Dokumentation angepasst oder erstellt wurde, einschliesslich aller notwendigen technischen Spezifikationen und Benutzeranleitungen.
        - der Code keine auskommentierten Teile enthält und alle *TODO*-Kommentare ein Issue-Ticket auf Gitlab referenzieren.
        - die Funktionalität im Client manuell getestet wurde und die Ergebnisse den erwarteten Anforderungen entsprechen.
        - vor dem Mergen alle Tests fehlerfrei durchlaufen, ein Peer-Review erfolgt ist und alle Phasen der CI/CD-Pipeline fehlerfrei absolviert wurden.
8. Für die Integration von Änderungen in den Develop-Branch verwenden wir Pull/Merge Requests.
9. Wir folgen Semantic Versioning 2.0. Das bedeutet, Versionen werden nach dem Schema MAJOR.MINOR.PATCH benannt, wobei inkompatible Änderungen die MAJOR-Version erhöhen, rückwärts kompatible Funktionshinzufügungen die MINOR-Version und rückwärts kompatible Bugfixes die PATCH-Version erhöhen.
10. Für die Code Reviews nutzen wir die integrierte Review und Kommentar-Funktionalität in Git-Lab. Nach Abschluss des Reviews wird ein zusammenfassender Kommentar im 'Summary-Comment' hinterlassen, der eine kurze Gesamtbeurteilung der vorgenommenen Änderungen enthält.
11. Jedes Teammitglied wird einer Disziplin zugewiesen, um im Konfliktfall Entscheidungen sowie Massnahmen zu fällen. Die Aufteilung und Zuständigkeit der Disziplinen sind auf der Seite Rollenverteilung & Verantwortlichkeiten beschrieben.

Testing Grundsätze:

Dieser Leitfaden soll das wichtigste für das schreiben von guten Unit Tests beschreiben. Eine genauere Beschreibung ist unter How to Write Good Unit Tests zu finden.

1. Wir testen kleine Codeabschnitte isoliert und ohne Abhängigkeiten zu externen Elementen.
2. Wir folgen dem Arrange, Act, Assert (AAA) Schema in unseren Tests.
3. Wir halten unsere Tests kurz und auf das Wesentliche beschränkt.
4. Wir decken zuerst den Happy Path - Idealfall mit optimalen Bedingungen - ab, bevor wir uns komplexeren Szenarien zuwenden.
5. Wir testen Edge Cases, um die Robustheit unserer Anwendung zu gewährleisten.

6. wir schreiben Tests, bevor wir Bugs beheben, um eine effektive Fehlerbehebung zu gewährleisten.
7. Wir schreiben deterministische Tests, die unter gleichen Bedingungen stets dasselbe Ergebnis liefern.
8. Wir verwenden beschreibende Namen für unsere Tests, um deren Zweck klar zu kommunizieren.
9. Wir bevorzugen präzise Assertions, um spezifische und informative Fehlermeldungen zu erhalten.
10. Wir führen unsere Tests automatisch als Teil des Build-Prozesses aus.

Tooling und Weitere:

1. Jedes Tool muss von der Mehrheit des Teams akzeptiert werden.
2. Projektnahe, technische Dokumentation (z.B. Anforderungen, Architektur, Use-Cases, Diagramme) bevorzugen wir direkt im Projekt-Wiki zu pflegen.
   - Für Diagramme verwenden wir die integrierte draw.io Anbindung im Wiki, Mermaid oder PlantUML.
   - Ziel ist eine nachvollziehbare, versionierte und code-nahe Dokumentation.
3. Kollaborative Inhalte (z.B. Besprechungsnotizen, Skizzen, Fotos, Ideensammlungen) werden flexibel entweder im Wiki oder in Google Docs erfasst, je nachdem, was situativ besser passt.
4. Für Besprechungen, Austausch und als allgemeiner Kommunikationsweg werden die Programme Microsoft Teams, WhatsApp oder der Email-Verkehr genutzt.
5. Alle Dateien werden, sofern text-basiert und/oder sinnvoll, in das Git-Repository des Projekts eingecheckt, als Wiki-Page oder in Google-Drive abgelegt, das von jedem Teammitglied zugänglich ist.
6. 🔴 ~~Wir verwenden Deutsch als Dokumentationssprache und Englisch für Commit-Nachrichten sowie für die Benennung und Kommentierung von Codeelementen.~~

Comments

Jvan Fadda @jvan.fadda · 4 months ago

Owner

Zusammen besprochen:

Branching-Strategie

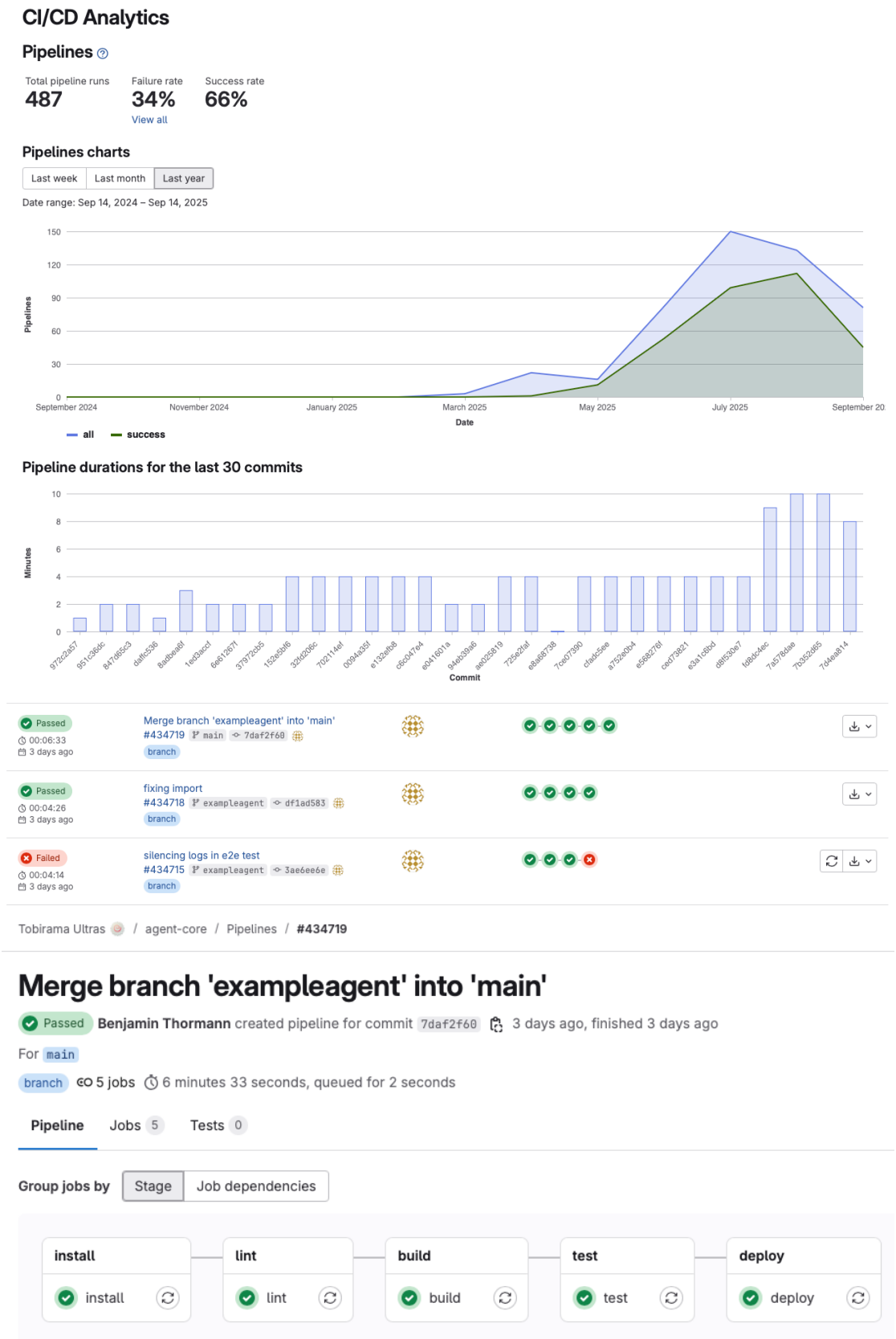
- Kein Git Flow nur Trunk-Based
- Branch-Naming:
  - `main` Haupt-Entwicklungszweig -> `trunk`
  - `fix` -> für alle fixes
  - `feat` -> für alle Features
  - Kein `release` -Branch und keine Release-Vorbereitung notwendig, da nicht mehrere Versionen gewartet werden.
    - Releases bzw. Versionen kennzeichnen wir lediglich mit `Tags`

Tooling

- wo woegmlich mit Wiki-Pages und integriertem draw-IO für Grafiken arbeiten.
  - Anforderungen, Use-Cases, Diagramme, Mermaid, PlantUml
- Google-Docs als alternative für PDFs, Dokumente der Cistec AG und kollaborative Dokumente wie Notizen, Skizzen, Fotos, Protokolle bzw. alles was nicht direkt mit dem code zu tun hat

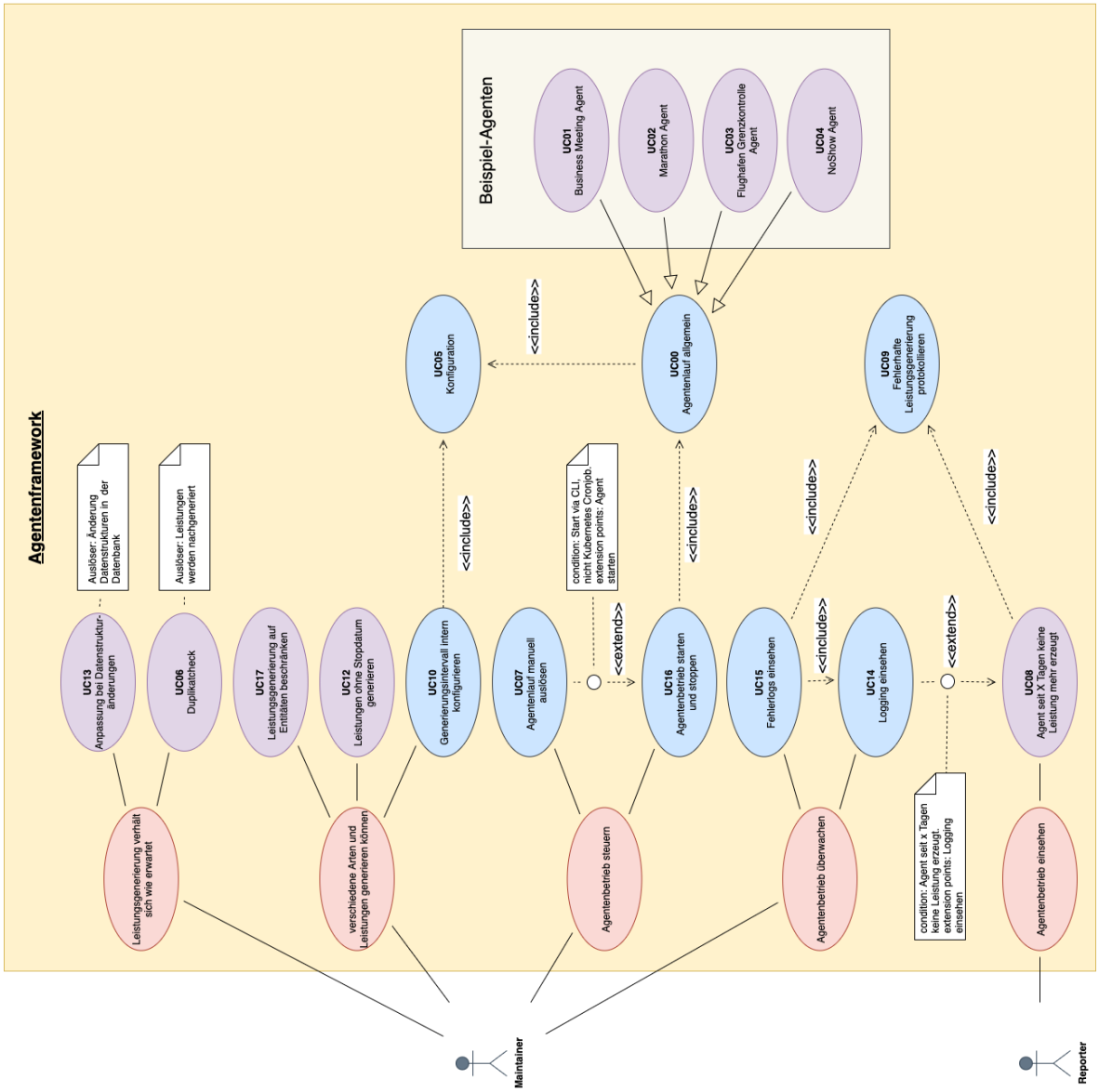
Edited 4 months ago by Jvan Fadda

11.7 Auszug aus Build-Server und CI/CD

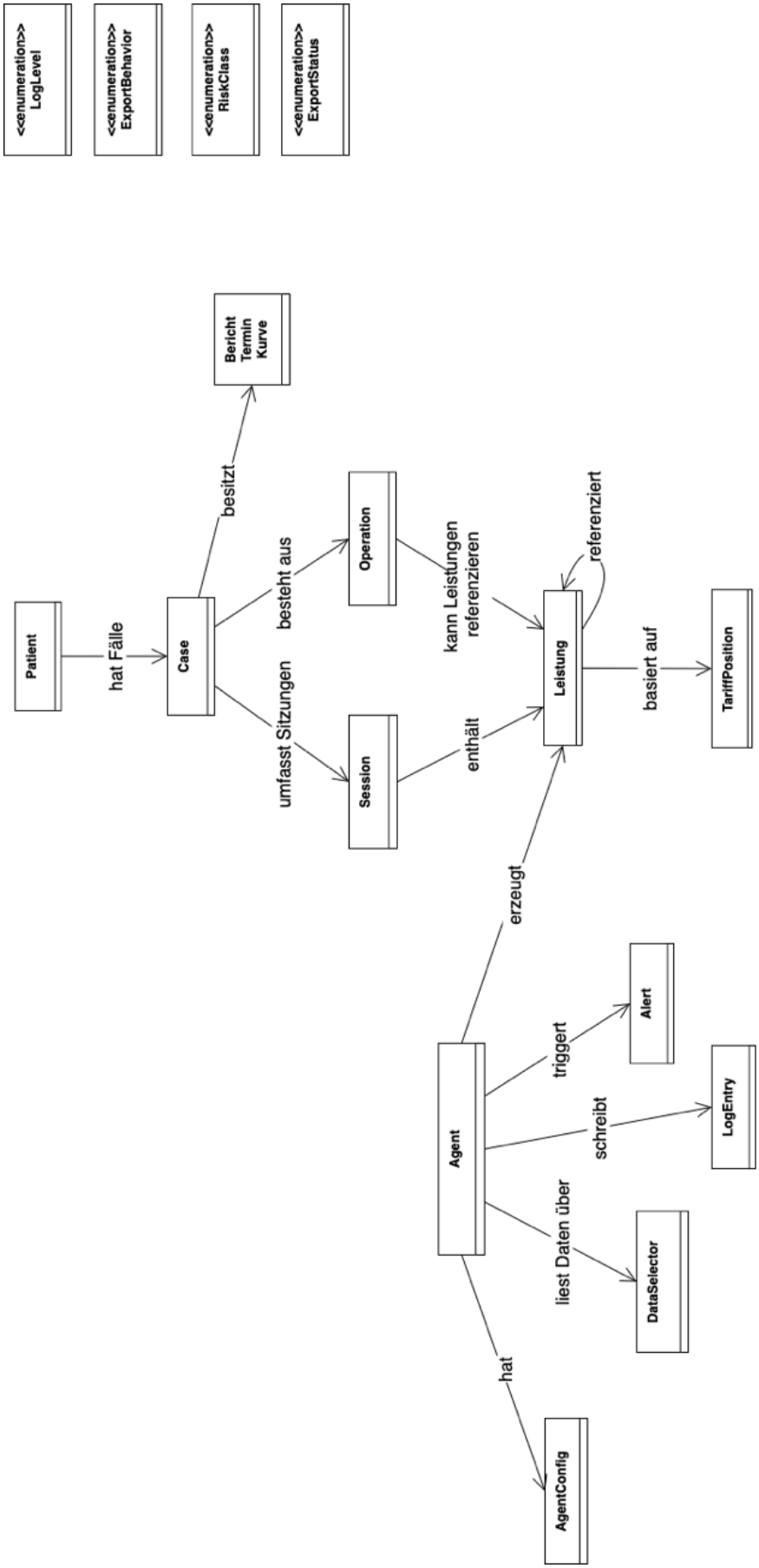


11.8 Technische Diagramme

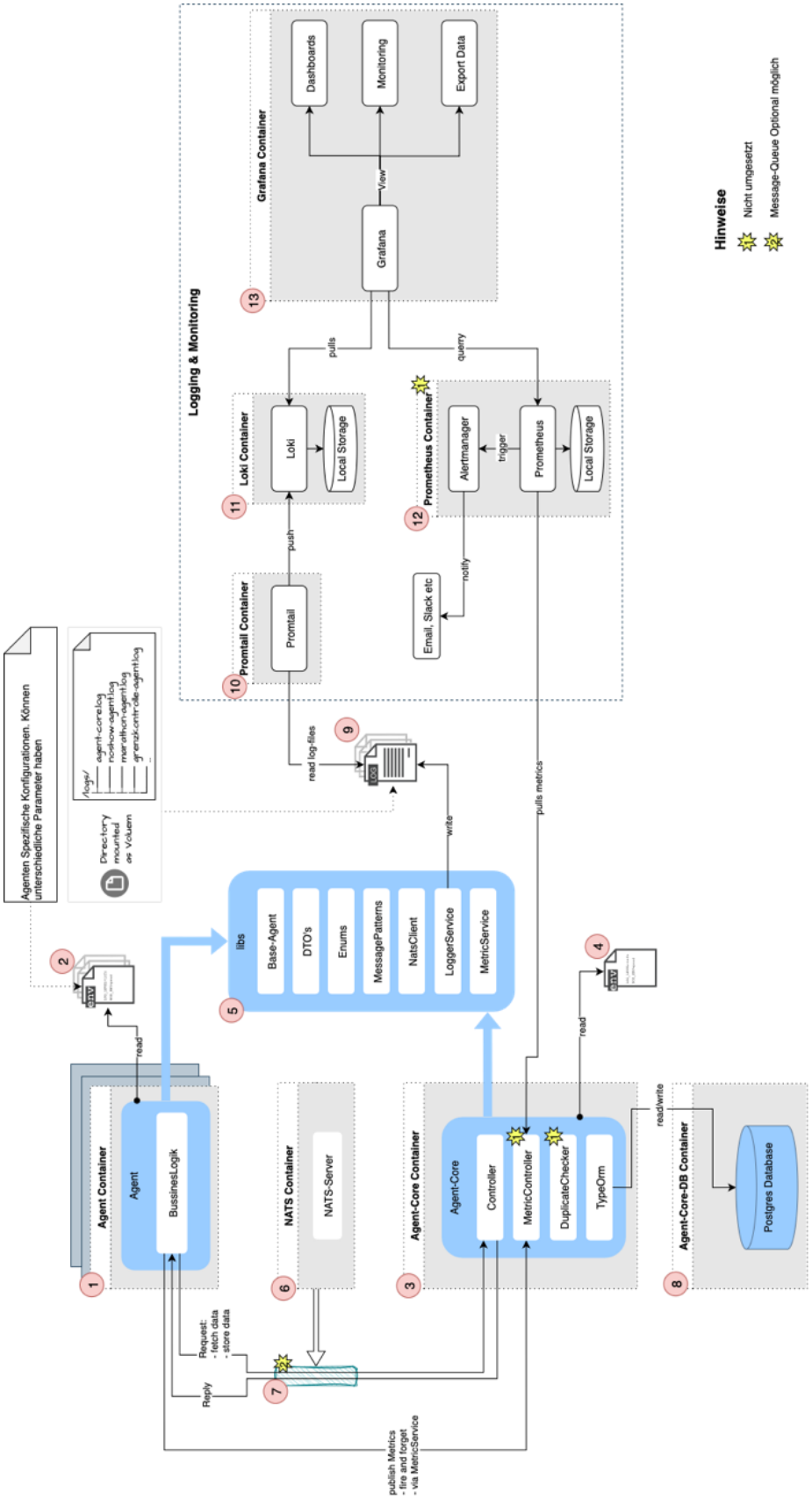
Use-Case Diagramm



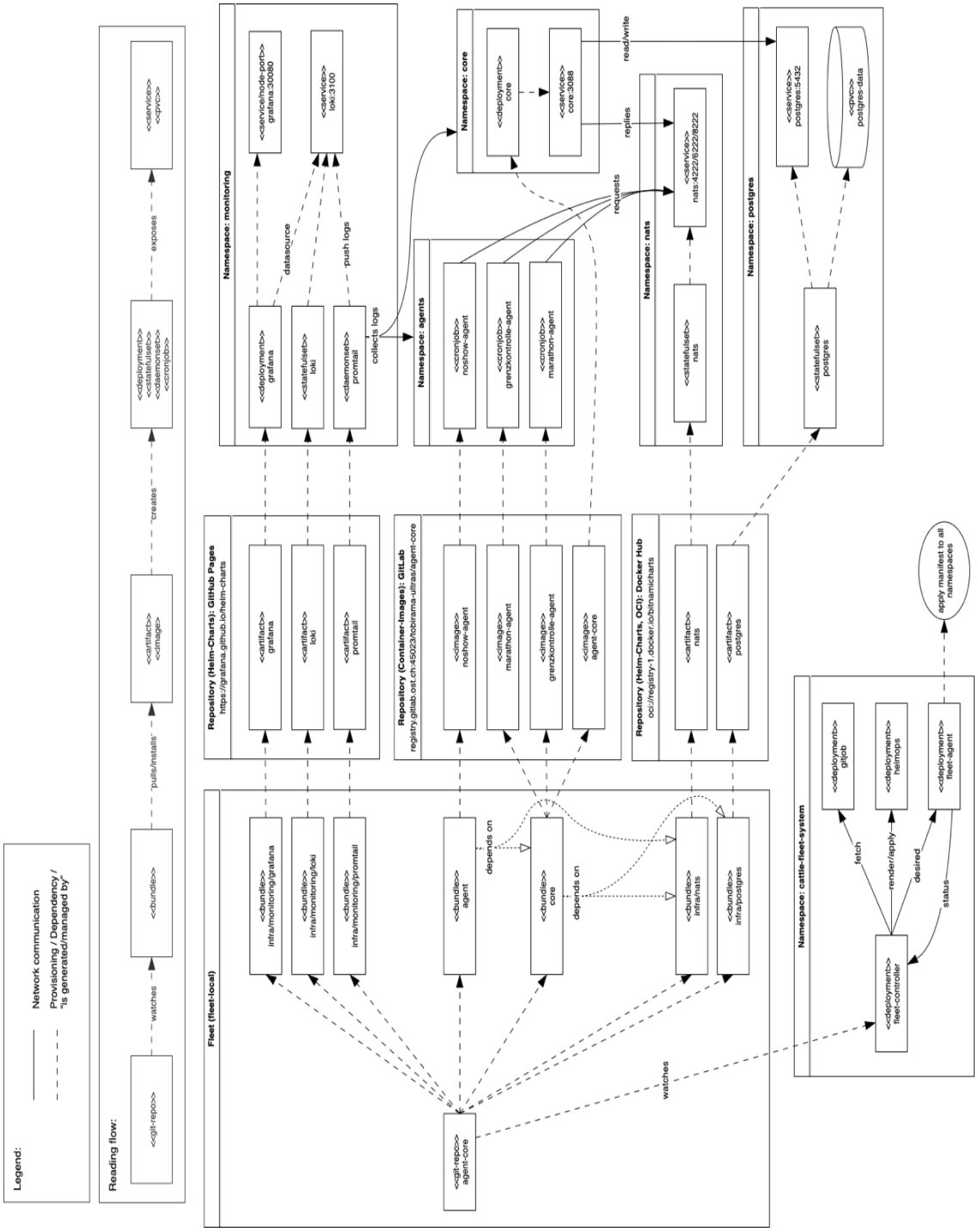
Domain Model



Systemüberblick



Verteilungsdiagramm



12 Selbstständigkeitserklärung

Selbstständigkeitserklärung

Hiermit erklären wir, dass wir die vorliegende Masterarbeit im MAS Software Engineering mit dem Titel «*Standardisierung von Agenten zur Leistungserfassung in Klinikinformationssystemen*» selbstständig und ohne unerlaubte fremde Hilfe angefertigt, keine anderen als die angegebenen Quellen und Hilfsmittel verwendet und die den verwendeten Quellen und Hilfsmitteln wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht haben. Weiterhin erklären wir, dass wir keine durch Copyright geschützten Materialien (z.B. Bilder) in dieser Arbeit in unerlaubter Weise verwendet haben und in dieser Arbeit keine Adressen, Telefonnummern und andere persönliche Daten von Personen, die nicht zum Kernteam gehören, publizieren.

Ort, Datum Campus Rapperswil-Jona, 14.09.2025

Name, Unterschrift: Fricker Guillaume,

Name, Unterschrift: Thormann Benjamin,

Name, Unterschrift: Fadda Jvan,