htlperg
innovativ | kreativ | praxisnah

ClarioAI - Claims Management Tool

DIPLOMARBEIT

Höhere Abteilung für Informatik

01/07/2025 – 26/03/2026

Projektmitglieder: Raphael Becherer
David Romani

Betreuer: Prof. Ing. Patrick Praher, MSc

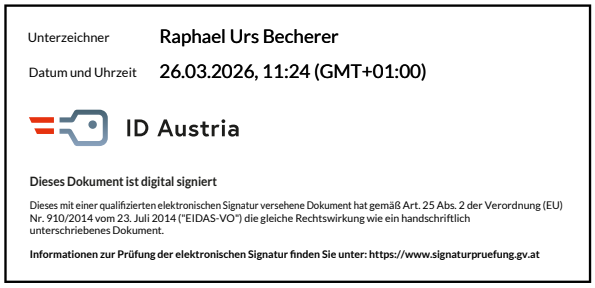
Eidesstattliche Erklärung

Hiermit versichern wir, die vorliegende Arbeit selbständig, ohne fremde Hilfe und ohne Benutzung anderer als der von uns angegebenen Quellen angefertigt zu haben. Alle Stellen, die wörtlich oder sinngemäß aus fremden Quellen direkt oder indirekt übernommen wurden, sind als solche gekennzeichnet.

Bei der Erstellung der Arbeit haben wir die generativen KI-Tools *ChatGPT*, *Gemini* und den Korrekturdienst *Scribbr* zu folgendem Zweck verwendet: zur Unterstützung bei der anfänglichen Ideenfindung, zur Hilfe bei der *LaTeX*-Formatierung, zur sprachlichen Optimierung einzelner Formulierungen sowie zur abschließenden Rechtschreib- und Grammatikprüfung.

Perg, 26.03.2026

Ort, Datum



Unterschrift, Raphael Becherer

Perg, 26.03.2026

Ort, Datum

Unterschrift, David Romani

Inhaltsverzeichnis

0.0.1 Danksagung

Wir bedanken uns sehr herzlich bei der Lamie direkt für die Unterstützung dieses Projekts und für die Möglichkeit, den praktischen Teil dieser Arbeit im Zuge eines Sommerpraktikums zu schreiben. Wir möchten auch besonders Daniel Fröschl und seinem Team danken für das Vertrauen, welches uns geschenkt wurde.

Außerdem möchten wir uns bei Prof. Patrick Praher dafür bedanken, dass er die Rolle als Betreuungslehrer übernommen hat und uns beigestanden ist.

Großer Dank gebührt auch unseren Eltern, die uns immer unterstützen.

1 Einleitung

1.1 Kurzfassung

Die Lamie ist ein Insurance Broker und arbeitet mit Claims, Invoices und den dazugehörigen Invoice Lines. Die Kunden schicken diese Informationen in verschiedenen Dateiformaten den Mitarbeitern der Lamie. Diese Informationen muss ein Mitarbeiter manuell in das interne System eintragen.

Ziel dieser Arbeit ist es, einen Prototyp zu entwickeln, der das Anlegen von Claims, Invoices und Invoice Lines vereinfacht. Dies funktioniert mit Hilfe von Bilderkennung und Künstlicher Intelligenz.

Dazu wurde konkret eine Webapplikation entwickelt, welche mittels Datei-Upload der Formulare der Kunden das Hochladen ermöglicht. Daraus werden dann die nötigen Informationen extrahiert. Dasselbe gilt auch für Invoices, wobei hier zusätzlich eine Künstliche Intelligenz eine Prediction darüber erstellt, ob eine Invoice Line von der Versicherung übernommen wird. Der Benutzer kann diese kontrollieren und kann jegliche Inhalte bearbeiten und verbessern.

Das Ergebnis ist eine Webapplikation, die als Prototyp agiert. Die Software kann Claims und Invoices von diversen Quellen in eine einheitliche, strukturierte Form bringen.

Die Webapplikation ist in keinem Produktiveinsatz. Des Weiteren gibt es keinen quantitativen Nachweis für einen Mehrwert. Jedoch besteht Potenzial für eine Integration in das bestehende System.

1.2 Abstract

Lamie is an insurance broker that handles claims, invoices and the associated invoice lines. Customers send this information to Lamie in various file formats. An employee must then manually enter this information into the internal system.

The aim of this project is to develop a prototype that simplifies the creation of claims, invoices and invoice lines. This is achieved using image recognition and artificial intelligence.

To this end, a web application has been developed which enables customers to submit forms via file upload. The necessary information is then extracted from these. The same applies to invoices, although in this case artificial intelligence also makes a prediction as to whether an

invoice line will be accepted by the insurance company. The user can check this and edit or improve any content.

The result is a web application that acts as a prototype. The software can convert claims and invoices from various sources into a uniform, structured format.

The web application is currently not in productive use. Furthermore, there is no quantitative evidence of added value. However, there is potential for integration into the existing system.

1.3 Motivation

Die Lamie arbeitet als Insurance Broker und ist daher intensiv mit Claims beschäftigt. Zur Verwaltung dieser Claims benutzt die Lamie bisher ein Customer Relationship Management Tool namens „LUIS“ (siehe Abschnitt 2.1.3 LUIS). Obwohl LUIS schon viele Funktionen hat, weist es noch Verbesserungspotenzial auf, da Claims oft noch manuell aufwendig erstellt werden. Im Schadensfall schicken die Kund:innen der Lamie oft handschriftlich ausgefüllte Formulare per E-Mail. Diese können in verschiedenen Dateiformaten vorkommen. Auf Basis dieser, teilweise sehr unschön ausgefüllten Formulare, muss dann ein Mitarbeiter manuell die Daten aus der per E-Mail übermittelten Datei in das LUIS-System eintragen. Ähnlich ist die Situation bei den Invoices, die der Kunde im Zusammenhang mit einem Claim übermittelt. Das sind häufig Handybilder von Rechnungen. Oft kommt es hier vor, dass die Versicherung nur eine Auswahl der Rechnungspositionen übernimmt.

Im Rahmen der Diplomarbeit wurde das Ziel verfolgt, den gesamten Prozess der Claim- und Invoice-Erstellung deutlich zu vereinfachen. Die entwickelte Webapplikation ermöglicht das Hochladen von Formularen und Rechnungen, deren Inhalte automatisch mittels Bilderkennung extrahiert und direkt in die entsprechenden Formulare übernommen werden. Dadurch beschränkt sich die Arbeit der Mitarbeiter lediglich darauf, die Richtigkeit der Daten zu überprüfen und fehlende Felder zu ergänzen.

Zusätzlich erhalten erkannte Invoice Lines mittels der ChatGPT-API eine *Prediction* darüber, ob eine einzelne Invoice Line gemäß dem jeweiligen Versicherungsvertrag übernommen werden würden. Das bedeutet, dass der Mitarbeiter nur mehr kontrollieren und in seltenen Fällen wieder nur vervollständigen und korrigieren muss.

Weil die Idee hinter dem Projekt unter anderem darin besteht mehr Klarheit zu schaffen, durch den Einsatz moderner Technologien, kam das Entwicklungsteam auf den Namen *ClarioAI*.

Das Kernthema dieser Diplomarbeit ist die Entwicklung eines Proof of Concept. Dieser Prototyp könnte dazu dienen, einen Mehrwert zu messen und in Zukunft in das LUIS-System eingebunden zu werden.

1.4 Zielsetzung

Ziel dieser Diplomarbeit ist es, einen Prototyp einer Webapplikation zu entwickeln, welche die Claim- und Invoice-Erstellung bei der Lamie vereinfacht. Diese Webapp wird durch den Einsatz von Bilderkennung und KI-gestützten Verfahren erreicht. Dadurch sollen manuelle Tätigkeiten minimiert werden und somit die Effizienz bei der Verwaltung von Claims gefördert werden.

Ein weiterer Schwerpunkt liegt auf der Integration einer KI-basierten *Prediction* von Invoice Lines, die vorhersagt, ob einzelne Rechnungspositionen gemäß des Versicherungsvertrags des Kunden übernommen werden.

Darüber hinaus wird im Rahmen dieser Arbeit bewertet, ob die Lösung einen potenziellen Mehrwert für den Auftraggeber darstellt. Dabei wird insbesondere analysiert, inwiefern die Integration der entwickelten Features in das bestehende Luis-System technisch bzw. organisatorisch sinnvoll ist.

1.5 Projektinhalt - Überblick

Dieses Kapitel dient dazu, einen Überblick über die elementaren Funktionen zu geben. Dazu werden drei Dinge detaillierter vorgestellt.

1.5.1 Daten aus Claims in der Datenbank speichern

Um Daten aus Claims und zugehörigen Invoices automatisiert zu erfassen, wird Optical Character Recognition (OCR) eingesetzt. Dadurch ist es möglich, auch handschriftliche, schlecht lesbare sowie multilinguale Dokumente zu digitalisieren und strukturiert in der Datenbank abzulegen [1].

Die extrahierten Daten bilden die Grundlage für die weitere Verarbeitung und Auswertung der Invoices. Sie enthalten unter anderem personenbezogene Informationen der Versicherten, wie bestehende Polizzen, Wohnort, persönliche Stammdaten sowie eine detaillierte Beschreibung des Schaden- bzw. Versicherungsfalls.

Als primäre Datenquelle dienen Schadenfälle von Versicherten, bei denen ein Claim zur Beantragung einer Entschädigung eingereicht wird. Diese Claims liegen in unterschiedlichsten Dateiformaten vor. Neben standardisierten PDF-Dokumenten finden sich auch handschriftlich ausgefüllte Formulare, teilweise mit Bleistift und in sehr geringer Schriftqualität. Zusätzlich stellt die multilinguale Natur der Claims eine besondere Herausforderung dar. Aufgrund des überwiegend osteuropäischen Kundenstamms treten Dokumente häufig in Sprachen wie Bulgarisch, Rumänisch, Slowakisch oder Kroatisch auf.

Um eine einheitliche Speicherung der Daten in deutscher Sprache zu gewährleisten und ein korrektes Datenformat sicherzustellen (z. B. bei Datums- und Zeitangaben), wird der rohe OCR-Output nachgelagert verarbeitet. Dazu wird der extrahierte Text mittels eines Azure OpenAI Prompts an ein Sprachmodell übergeben, welches den Text übersetzt, strukturiert und in ein für die Datenbank geeignetes Format überführt. Die Begründung der Modellauswahl erfolgt in Abschnitt 3.2.6.

Die formatierten Daten werden anschließend automatisch in die entsprechenden Eingabefelder eines Formulars im Browser übernommen. Falls einzelne Werte nicht erkannt oder extrahiert werden konnten, bleiben die betroffenen Felder leer und können manuell von den Mitarbeit:innen ergänzt werden. Sämtliche Daten werden vor der finalen Speicherung durch eine Person überprüft, validiert und gegebenenfalls korrigiert, um vollständige und konsistente Datensätze sicherzustellen.

Nach Bestätigung der Daten wird ein entsprechendes Claim-Objekt sowie ein zugehöriges Personenobjekt erstellt. Dadurch können die gespeicherten Daten weiterverarbeitet und zu einem späteren Zeitpunkt zusätzliche Invoices eindeutig dem jeweiligen Claim zugeordnet werden.

1.5.2 Invoices verarbeiten

Zu jedem Claim können nach dessen Erstellung Invoices hinzugefügt werden. Dabei handelt es sich um eine 1:n-Beziehung, da einem Claim mehrere Invoices zugeordnet werden können. Diese Invoices sind häufig kaum oder fast gar nicht lesbar. Neben schwer entzifferbarer Handschrift stellt insbesondere die Qualität der eingereichten Fotos von Rechnungen ein Problem dar. Zusätzlich kommt es vor, dass Rechnungen von Ärzten handschriftlich ausgestellt werden, was eine besondere Herausforderung darstellt. In solchen Fällen stoßen auch KI-Modelle teilweise an ihre Grenzen.

Sofern diese Voraussetzungen erfüllt sind und eine Invoice mittels OCR ausgelesen werden kann, sind für die Auswertung folgende Informationen relevant und werden gespeichert: die einzelnen Rechnungspositionen (Invoice Lines), die Art des Rechnungsausstellers (z.B. Krankenhaus oder Apotheke), Einzelpreise sowie der Gesamtpreis. Diese Daten müssen anschließend von einem Mitarbeiter überprüft werden.

Die Art des ausstellenden Unternehmens wird bestimmt, indem der vollständige Output des OCR-Prozesses erneut in einem GPT-Prompt verarbeitet wird. Anhand der extrahierten Daten wird – sofern auf der Rechnung kein Unternehmen eindeutig angegeben ist – eine Vermutung über den Rechnungsaussteller getroffen. Die Ergebnisse werden vom Prompt in ein korrekt formatiertes JSON-Format überführt und können anschließend in der Datenbank gespeichert werden.

1.5.3 Invoices auswerten

Die verarbeiteten Rechnungen sind jetzt in der Datenbank gespeichert, jeweils mit einer oder mehreren Rechnungspositionen. Diese Rechnungspositionen müssen nun bewertet werden, d. h., es muss geprüft werden, ob diese Position von der vorhandenen Versicherung des Versicherungsnehmers abgedeckt ist. Die Positionen werden dann entsprechend so gespeichert, dass entweder ein Teil, die gesamte Rechnung oder gar nichts davon von der Versicherung abgedeckt ist. Hier ist es auch wieder wichtig, nicht zu vergessen, dass ein Mitarbeiter die KI-Analyse überprüft, um Fehler zu vermeiden.

1.6 Projektumfeld

Dieses Kapitel beschreibt, in welchem Umfeld die Arbeit zustande kam. Es wird erklärt, wer zum Team gehört, wer der Auftraggeber ist und in welchem Ausmaß und in welcher Art die Betreuung passierte.

1.6.1 Projektteam

Das Entwicklerteam besteht aus David Romani und Raphael Becherer. Es handelt sich hierbei um die erste gemeinsame Projektarbeit der beiden Projektmitglieder.

David Romani entwickelt in dieser Arbeit das Frontend sowie dessen Design. Ihm sind die Technologien schon vertraut, da er bereits in Projekten in der Vergangenheit mit denselben Technologien gearbeitet hat.

Raphael Becherer ist in dieser Arbeit für das Backend und die dazugehörende Datenbank zuständig. Die verwendeten Technologien sind ihm größtenteils nur vage bekannt, da er bisher nur kleine Projekte mit C# und noch gar keine Projekte mit Azure-Technologien umgesetzt hat. Dank universeller Verbindungen zu anderen Programmiersprachen und Technologien konnte er sich jedoch relativ gut in die Materie einarbeiten.

1.6.2 Auftraggeber

Die Diplomarbeit wurde in Zusammenarbeit mit der „Lamie direkt“, einem Unternehmen in der Versicherungsbranche, durchgeführt. Als Insurance Broker beschäftigt sich die Lamie direkt mit der Abwicklung von Schadensfällen und arbeitet intensiv mit Claims, Invoices und den dazugehörigen Rechnungspositionen.

Im Rahmen dieser Arbeit übernahm die Lamie direkt die Rolle des Auftraggebers und stellte den thematischen Rahmen sowie mehrere Ansatzpunkte zur Verfügung. Die Themenauswahl geschah eigenständig durch das Entwicklungsteam.

In dieser Arbeit wird die *Lamie direkt* oft verkürzt als *Lamie* bezeichnet.

1.6.3 Betreuung

Es gab keine kontinuierliche Betreuung im klassischen Sinn. Laufende Kontrollen oder Reviews wurden von Seiten des Auftraggebers nicht durchgeführt. Das Entwicklerteam und deren Leistungen wurden nicht kontrolliert.

Der Auftraggeber hat, wie in Abschnitt 1.6.2 beschrieben, den thematischen Rahmen vorgegeben. Innerhalb dieses Rahmens hat das Entwicklerteam viele Freiheiten genossen. Zum Beispiel wurden kein Design oder keine Seiteninhalte vorgegeben. Deswegen war ein hoher Grad an Eigenverantwortung seitens des Entwicklerteams notwendig.

Es gab jedoch punktuelle Unterstützung. So hatte das Entwicklerteam sowohl für das Frontend als auch für das Backend Ansprechpersonen zur Verfügung. Zusätzlich gab es einige informelle fachliche Inputs zu Designfragen für das Frontend. Diese Unterstützungen wurden nur bei Bedarf in Anspruch genommen und waren nicht dauerhaft.

Das Entwicklerteam empfand die Betreuung als angemessen und das Ausmaß der Betreuung hinderte den Projektfortschritt in der Entwicklungsphase in keiner Weise. Das Projekt konnte zielgerichtet umgesetzt werden.

1.7 Projektabgrenzung

Das Produkt dieser Arbeit arbeitet mit keinen produktiven Kundendaten, da es dafür eine rechtliche Prüfung der Datenverarbeitung bräuchte. Deshalb gibt es keine Garantien hinsichtlich der DSGVO-Konformität, der verwendeten externen APIs und KI-Dienste. Die Nutzung ist ausschließlich für Demonstrations- und Prototypzwecke gedacht. Eine reale Nutzung wäre nur nach zusätzlicher Prüfung möglich.

Aus diesen Gründen ist das Produkt weder im Produktionseinsatz noch im Livebetrieb. Außerdem ist diese Software auch kein Ersatz für das LUIS-System der Lamie, vielmehr ist es ein Prototyp einer potenziellen Erweiterung des bestehenden Systems. Das bedeutet, das Produkt der Diplomarbeit steht für sich allein und ist in keinem anderen System integriert oder anderweitig verbunden.

Die *Predictions*, die von der OpenAI-API erstellt werden, sind nicht verbindlich und sind vom Benutzer zu hinterfragen und gegebenenfalls zu korrigieren. Die Bilderkennung sowie die

Predictions von OpenAI sind als Unterstützung zu betrachten. Die Entscheidung und somit die Verantwortung liegen vollständig beim Benutzer der Software.

Diese Arbeit beschäftigt sich nicht mit der Evaluation des Mehrwertes. Es besteht kein quantitativer Nachweis für Effizienzgewinne. Außerdem wurden keine Nutzerstudien oder Zeit-/Kostenmessungen durchgeführt.

2 Theoretische und fachpraktische Grundlagen und Methoden

In den folgenden Unterpunkten werden Begriffe, welche in dieser Arbeit oft vorkommen, erklärt und definiert.

2.1 Grundlegende Fachbegriffe

2.1.1 Claim

Ein Claim beschreibt im Versicherungswesen einen vom Kunden gemeldeten Schadensfall und stellt den formellen Anspruch auf die vertraglich vereinbarten Versicherungsleistungen dar. Ein Claim beinhaltet alle relevanten Informationen zum Schadenereignis (wie Datum, Ort und Beschreibung), zu den beteiligten Personen (Versicherungsnehmer und Leistungsberechtigter) und zum Versicherungsvertrag sowie zum Bearbeitungs- und Statusverlauf.

Der Versicherungsnehmer und die betroffene Person müssen nicht zwingend dieselbe Person sein. Es ist beispielsweise möglich, dass ein Elternteil eine Versicherung abschließt, deren Leistungen im Schadensfall den Kindern zugutekommt.

2.1.2 Invoices und Invoice Line

Eine Invoice beschreibt in dieser Arbeit einen Beleg oder eine Rechnung im betriebswirtschaftlichen Sinne. Invoices sind von den Dateien zu unterscheiden, die der Anwender hochladen kann. Während diese Dateien zwar die ursprünglichen Dokumente darstellen, beschreibt eine Invoice die daraus resultierenden Rechnungsinformationen.

Invoice Lines sind grundlegende Bestandteile jeder Invoice. Man könnte sie mit Rechnungspositionen in der Betriebswirtschaft vergleichen. Invoice Lines sind jedoch nicht gänzlich identisch mit einer klassischen Rechnungsposition. Sie enthalten zusätzliche, kontextuelle Informationen.

2.1.3 LUIS

Das *LUIS-System* oder auch einfach *LUIS* ist ein *Customer Relationship Management Tool* (*CRM*), welches bei der Firma *Lamie* sämtliche Kundendaten verwaltet. Dieses System ist nur intern zugänglich und essenziell für das operative Geschäft des Unternehmens. *LUIS* befindet sich in stetiger Weiterentwicklung.

2.2 Verwendete Technologien

In diesem Kapitel werden alle verwendeten Technologien erklärt. Außerdem wird begründet, warum diese zum Einsatz kamen. Begonnen wird mit den Technologien des Frontends und abgeschlossen wird mit den Technologien, die beim Backend zum Einsatz kamen.

2.2.1 Angular

Angular ist eines der beliebtesten *Frontend-Webframeworks* [2] zur Entwicklung von Webseiten und wird von einem Team von Google-Mitarbeitern gewartet. Die Entwicklung erfolgt unter dem Einsatz von HTML, CSS und TypeScript. Angular-Anwendungen sind klassische clientseitige Anwendungen und gründen auf einem komponentenbasierten Ansatz. Dieses mit Bausteinen vergleichbare System führt dazu, dass sowohl sichtbare Elemente als auch deren Logik klar von anderen Komponenten getrennt sind. Das führt zu einer besseren Wartbarkeit und erleichtert Erweiterungen der Webseite.[3]

Im Gegensatz zu einfachem *JavaScript* bietet *TypeScript* Typsicherheit. Das heißt, Objekte haben Typen, ähnlich wie in objektorientierten Programmiersprachen wie Java, C# oder Python. Dadurch wird die Entwicklung des Frontends vereinfacht und zügiger vorangetrieben. Des Weiteren ist *TypeScript* für die *API-Calls* unverzichtbar. Sie werden in sogenannten *Services* ausgelagert. Die *Services* werden dort, wo sie gebraucht werden, mittels „Dependency Injection“ eingebunden [4].

Für die Realisierung der komplexen Formulare in der *ClarioAI*-Applikation wurden *Reactive Forms* eingesetzt [5]. Hierbei wird die Logik und die Validierung der Eingabefelder direkt im *TypeScript*-Code gesteuert, was eine übersichtliche Kontrolle über die Eingabedaten ermöglicht. Zudem nutzt die Anwendung den *Angular Router*, um als „Single Page Application“ (SPA) zu fungieren. Dies stellt sicher, dass Navigation zwischen den Ansichten ohne vollständiges Neuladen der Seite erfolgen, was die *User-Experience* und die *Performance* verbessert.

Weil viele Prozesse, wie die Kommunikation mit der *OCR-Schnittstelle*, asynchron ablaufen müssen, kommen *RxJS* [6] und „Observables“ [7] zum Einsatz. Diese ermöglichen es, Daten-

ströme effizient zu verarbeiten und die Benutzeroberfläche erst dann zu aktualisieren, wenn die entsprechenden Informationen vom Server vorliegen.

Die Entwickler entschieden sich für Angular als Frontend-Framework, da dieses Framework genau für strukturierte, datengetriebenen Benutzeroberflächen geeignet ist und auch die klare Gliederung durch die Komponenten-Architektur ist gut geeignet für unsere Anwendung. So ermöglicht das Framework, UI-Element auf mehreren Seiten wiederzuverwenden. Zusätzlich hat das Entwicklungsteam schon praktische Erfahrung mit Angular gesammelt, welche eine effiziente Implementierung des Frontends ermöglicht.

2.2.2 Tailwind CSS

Nach Tailwind Labs [8] erfolgt die Nutzung von Tailwind durch den direkten Einsatz der Utility-Klassen in den Markup-Dateien, also in den HTML-Files. Im Entwicklungsprozess muss man daher nie CSS-Files manipulieren. Das Framework baut parallel zum Entwicklungsprozess eigene CSS-Files (build.css) auf. Ermöglicht wird das automatische Generieren des CSS-Files durch die *Just-In-Time-Engine* (kurz: JIT-Engine), welche Tailwind CSS seit Version 3 nutzt. Diesen Ansatz unter Verwendung der beschriebenen Utility-Klassen nennt man *Atomic CSS*, so beschreibt es Polacek [9]. Man verwendet also nicht mehr sprechende, selbstgeschriebene CSS-Klassen, sondern die Tailwind-Klassen. Das hat den Vorteil, dass der Code vorhersehbarer wird, weil jede Klasse genau eine Aufgabe hat.

Hinsichtlich der Performance überzeugt das Framework durch den Einsatz von *PurgeCSS*. Dabei wird beim Erstellen des produktiven *Builds* das gesamte CSS nach ungenutzten Klassen durchsucht und diese entfernt, was zur extremen Minimierung der Dateigrößen führt. Darüber hinaus ermöglicht Tailwind die einfache Handhabung von Benutzerinteraktionen durch *State-Modifier*. Zustände wie „hover“ oder „focus“ können direkt im HTML deklariert werden, was die Komplexität im Stylesheet reduziert.

Tailwind CSS wird in einem Projekt ausschließlich an einer Stelle konfiguriert, nämlich in der *tailwind.config.js*. Dieses File gilt als „Single Source of Truth“. Das bedeutet: Will man eine Änderung an einer Klasse machen, die die ganze App betreffen soll, so geht das nur an dieser Stelle. Also wenn man die Primärfarbe der App ändern will, dann geht man in die erwähnte Datei und ändert die Definition dafür hier um.

Das Entwicklerteam entschied sich für TailwindCSS, weil es dahingehend schon Expertise gesammelt hatte. Dadurch, dass durch die Verwendung das Hin- und Herspringen zwischen HTML- und CSS-File wegfällt, sinkt die Entwicklungszeit massiv. Außerdem steigt durch den *Atomic-CSS*-Ansatz die Wartbarkeit dieser Applikation deutlich an.

2.2.3 ASP.NET Core

ASP.NET Core ist ein Webframework von Microsoft zur Entwicklung von serverseitigen Webanwendungen und insbesondere von Web-APIs [10]. Das Framework ist plattformunabhängig und kann auf Windows, Linux und macOS betrieben werden [11]. Es ist Teil des `.NET`-Ökosystems und arbeitet eng mit C# und `.NET`-Runtime zusammen. Dadurch eignet sich ASP.NET Core gut für Anwendungen, die eine stabile und gut wartbare Backend-Schnittstelle brauchen.

Ein wichtiges Einsatzgebiet von ASP.NET Core ist die Entwicklung von Web-APIs [12]. Dabei werden HTTP-Endpunkte bereitgestellt, die über die `CRUD` Methoden angesprochen werden können [13]. Als Datenformat wird dabei meistens JSON verwendet [14, 15]. Dadurch eignet sich ASP.NET Core sehr gut als Backend für moderne Frontends wie Angular.

Architektonisch unterstützt ASP.NET Core eine klare Trennung von Verantwortlichkeiten. In vielen Projekten wird deshalb ein controllerbasierter Ansatz verwendet. Dabei übernehmen die Controller die HTTP-Kommunikation, während die eigentliche Fachlogik in separaten Service-Klassen umgesetzt wird [16]. Das hat den Vorteil, dass Änderungen an der Business-Logik nicht direkt mit der HTTP-Ebene vermischt sind. Dadurch steigt die Wartbarkeit des Backends.

Ein weiteres wichtiges Merkmal ist die Unterstützung von Middleware. Middleware-Komponenten werden in einer Pipeline angeordnet und verarbeiten Requests und Responses schrittweise [17]. Damit können Themen wie CORS, Authentifizierung, Fehlerbehandlung oder Logging konfiguriert werden. Das hat den Vorteil, dass diese Logik nicht in jedem einzelnen Endpunkt neu implementiert werden muss.

Zusätzlich hat ASP.NET Core standardmäßig einen *Dependency-Injection*-Container [18]. Dadurch können Services, Datenzugriffskomponenten oder externe Clients zentral registriert und danach in andere Klassen injiziert werden. Das erleichtert das Testen der Anwendung.

Die Entwickler entschieden sich für ASP.NET Core, da das Framework gut für strukturierte und datengetriebene Backend-Anwendungen geeignet ist. Besonders die klare Gliederung durch Controller, Services und Middleware passt gut zur Architektur dieser Anwendung. Außerdem ermöglicht das `.NET`-Ökosystem zusammen mit bestehenden Bibliotheken und NuGet-Paketen eine relativ effiziente Implementierung des Backends.

2.2.4 Optical Character Recognition (OCR)

Der Begriff OCR steht für *Optical Character Recognition* und beschreibt Verfahren, mit denen Text aus Bildern, gescannten Dokumenten oder PDF-Dateien extrahiert und in maschinenlesbaren Text umgewandelt werden kann [19]. OCR ist damit eine wichtige Grundlage für die digitale

Dokumentenverarbeitung. Es wird zum Beispiel bei Formularen, Rechnungen oder allgemein bei gescannten Dokumenten eingesetzt.

Frühere OCR-Verfahren arbeiteten oft mit festen Regeln, Mustervorlagen oder merkmalbasierten Ansätzen. Moderne OCR-Systeme gehen darüber hinaus und verwenden Machine Learning und Deep Learning [19]. Dadurch können heute auch komplexere Dokumente, unterschiedliche Schriftarten und handschriftliche Texte deutlich besser verarbeitet werden.

Für ClarioAI ist OCR ein zentraler Bestandteil der Dokumentverarbeitung. In diesem Projekt wird dafür Azure Document Intelligence verwendet. Das sogenannte Read-Modell dient dabei zur eigentlichen Texterkennung und kann gedruckten sowie handschriftlichen Text aus gescannten Dokumenten extrahieren [20]. Zusätzlich erkennt das Modell unter anderem Absätze, Textzeilen, Wörter, Positionen und Sprachen [20, 21].

Der OCR-Prozess kann in 5 Schritte gegliedert werden:

1. **Layoutanalyse**

   Im ersten Schritt wird das Dokument vorbearbeitet, dass die Texterkennung besser funktioniert. Dazu gehören zum Beispiel Kontrastanpassungen, Rauschunterdrückung oder die Entzerrung schiefer Scans. Danach wird analysiert, aus welchen Bereichen das Dokument besteht, also etwa aus Textblöcken, Tabellen oder anderen Elementen. Moderne Systeme berücksichtigen dabei oft auch Lesereihenfolge, Absätze oder weitere Strukturelemente [19].

2. **Segmentierung**

   Danach folgt die Segmentierung. Dabei werden die erkannten Bereiche weiter unterteilt, zum Beispiel in einzelne Textsektionen, Zeilen oder Wörter. Dieser Schritt ist wichtig, weil Fehler an dieser Stelle direkte Auswirkungen auf die spätere Texterkennung haben können [19].

3. **Zeichenerkennung**

   Im nächsten Schritt findet die eigentliche Zeichen- und Texterkennung statt. Während ältere Systeme häufig einzelne Zeichen isoliert erkannt haben, arbeiten moderne Systeme oft auf Wort-, Zeilen- oder Satzebene [19]. Dadurch kann mehr Kontext berücksichtigt werden, was die Erkennung robuster macht. Azure Document Intelligence unterstützt sowohl gedruckte als auch handschriftliche Texte [20].

4. **Nachbearbeitung**

   Nach der Erkennung folgt eine Nachbearbeitung. Dabei können erkannte Inhalte normalisiert, offensichtliche Fehler korrigiert oder an das Zielsystem angepasst werden. Gerade

bei fehlerhaften oder schlecht lesbaren Dokumenten ist dieser Schritt wichtig, weil sonst OCR-Fehler direkt in spätere Verarbeitungsschritte übernommen werden. In ClarioAI wird dieser Schritt zusätzlich durch eine LLM-basierte Strukturierung ergänzt.

5. **Codierung**

   Am Ende werden die Ergebnisse nicht nur als einfacher Text ausgegeben. Moderne Systeme stellen die erkannten Inhalte oft in strukturierter Form zur Verfügung, zum Beispiel als JSON [22, 23]. Das hat den Vorteil, dass die Daten danach leichter automatisiert weiterverarbeitet werden können. Azure Document Intelligence stellt dafür verschiedene Modelltypen bereit. Dazu gehören unter anderem Read-, Layout-, vorgefertigte und benutzerdefinierte Modelle [24, 23].

Die Qualität der OCR-Ergebnisse hängt stark von den Eingabedaten ab. Schlechte Bildqualität, Verzerrungen, ungünstige Beleuchtung oder unübersichtliche Dokumentstrukturen können die Erkennungsgenauigkeit deutlich verschlechtern [19]. Auch moderne Systeme können solche Probleme nicht vollständig ausgleichen. Deshalb ist es vor allem bei kritischen Dokumenten sinnvoll, die Ergebnisse zusätzlich zu validieren.

Zusammenfassend lässt sich sagen, dass OCR heute nicht mehr nur aus reiner Zeichenerkennung besteht. Moderne Systeme verbinden Texterkennung, Layoutanalyse und strukturierte Ausgabe zu einer gemeinsamen Verarbeitungskette. Im Fall von ClarioAI bildet Azure Document Intelligence dafür die technische Grundlage.

2.2.5 Entity Framework Core und SQL Server

Entity Framework Core ist ein Framework von Microsoft für den Datenzugriff in `.NET`-Anwendungen [25]. In dieser Arbeit wird es zusammen mit SQL Server verwendet. SQL Server ist ein relationales Datenbankmanagementsystem von Microsoft und speichert Daten in Zeilen und Spalten [26, 27]. Dadurch eignet sich diese Kombination gut für Anwendungen, bei denen strukturierte Daten langfristig und zuverlässig gespeichert werden müssen.

In diesem Projekt wird mit Code-First-Modellen gearbeitet. Das bedeutet, dass die Modelklassen zuerst im Code definiert werden und daraus später das relationale Schema der Datenbank entsteht [25]. Änderungen am Datenmodell werden mit Migrations nachvollziehbar gemacht. EF Core beschreibt Migrations als einen Mechanismus, mit dem das Datenbankschema schrittweise an Änderungen im Modell angepasst werden kann, ohne bestehende Daten zu verlieren [28]. Das hat den Vorteil, dass sich die Datenbank gemeinsam mit der Anwendung weiterentwickeln kann.

Für die eigentliche Datenstruktur werden relationale Beziehungen zwischen den Entitäten verwendet. Dazu gehören zum Beispiel 1:n-Beziehungen zwischen Schadenfällen und Rech-

nungen oder zwischen Rechnungen und einzelnen Positionen. Diese Art von Datenmodell passt gut zu klassischen CRUD-Abläufen. EF Core unterstützt genau diese Muster sehr gut, da Entitäten über den `DbContext` geladen, geändert und wieder gespeichert werden können [**microsoft_dbcontext**].

Die Entwickler entschieden sich für Entity Framework Core in Kombination mit SQL Server, weil diese Technologien gut in das `.NET`-Ökosystem passen und eine saubere Umsetzung des relationalen Datenmodells ermöglichen. Besonders die Code-First-Modelle und Migrations erleichtern die Weiterentwicklung der Anwendung.

2.2.6 Azure Blob Storage

Azure Blob Storage ist ein Cloud-Speicherdienst von Microsoft zur Ablage unstrukturierter Dateien wie Dokumente, Bilder oder andere Binärdaten [29]. In diesem Projekt wird Blob Storage verwendet, um hochgeladene Dateien nicht direkt in der Datenbank, sondern separat im Storage abzulegen. In der Datenbank werden dann nur die zugehörigen Metadaten wie Dateiname, URL oder Upload-Zeit gespeichert.

Die eigentlichen Dateien werden in Containern organisiert. Diese Container dienen als Struktur innerhalb des Speichers und machen es einfacher, unterschiedliche Dateitypen oder Anwendungsbereiche voneinander zu trennen [29]. Für die Anwendung wichtig sind vor allem die Operationen Hochladen, Anzeigen und Löschen. Die Azure-`BlobClient`-Klasse stellt genau solche Funktionen bereit [30, 31, 32]. Dadurch kann eine Datei hochgeladen, später wieder gelesen und bei Bedarf auch gelöscht werden.

Das hat den Vorteil, dass große Dateien nicht die Datenbank belasten und trotzdem sauber mit den übrigen Daten verknüpft werden können. Die Entwickler entschieden sich deshalb für Azure Blob Storage, weil sich damit eine skalierbare und gut trennbare Dateiverwaltung umsetzen lässt.

2.2.7 Azure OpenAI Integration

Für die nachgelagerte Verarbeitung der OCR-Ergebnisse wird Azure OpenAI verwendet. Dabei wird der von Azure Document Intelligence extrahierte Text an ein Modell übergeben und dort weiterverarbeitet. Azure OpenAI unterstützt dafür unter anderem Structured Outputs. Diese Funktion sorgt dafür, dass das Modell eine vorgegebene JSON-Struktur einhält [33]. Das ist gerade bei der Extraktion von Rechnungs- oder Schadendaten hilfreich, weil die Ergebnisse danach direkt in das Backend übernommen werden können.

Ein wichtiger Unterschied zu OpenAI selbst ist, dass bei Azure OpenAI nicht einfach nur der Modellname verwendet wird. Stattdessen wird immer mit einem Deployment gearbeitet. In den API-Aufrufen muss also der Deployment-Name angegeben werden und nicht nur der Name des zugrunde liegenden Modells [34, 35]. Das ist wichtig, weil dadurch die Modellverwaltung stärker an die Azure-Ressource gebunden ist.

In dieser Arbeit wird Azure OpenAI vor allem dafür genutzt, OCR-Text in eine einheitliche und strukturierte Form zu bringen. Dazu gehören zum Beispiel die Extraktion einzelner Felder, die Normalisierung von Datums- und Zahlenformaten und die Ausgabe als JSON. Auch wenn Structured Outputs die Arbeit deutlich erleichtern, ist in der Praxis oft noch eine zusätzliche Nachbearbeitung der JSON-Antwort nötig, damit das Ergebnis sicher verarbeitet werden kann [33, 36].

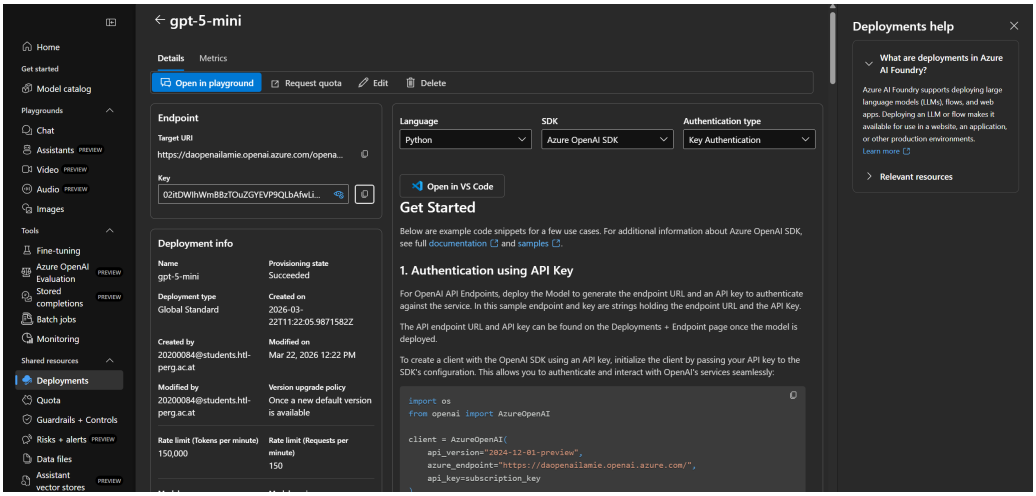


Abbildung 1: Azure AI Foundry

Die Entwickler entschieden sich für Azure OpenAI, weil dadurch die LLM-Verarbeitung direkt in die bestehende Azure-Umgebung integriert werden kann. Das erleichtert die technische Einbindung und passt gut zur restlichen Cloud-Architektur des Systems.

2.2.8 Object Mapping mit Mapster

Mapster ist eine Bibliothek für Object Mapping in `.NET`. Sie wird verwendet, um Objekte automatisiert in andere Objektstrukturen zu bringen, zum Beispiel Entitäten in DTOs oder DTOs zurück in Entitäten [37]. Dadurch muss diese Zuordnung nicht an vielen Stellen von Hand implementiert werden.

Vor allem in Anwendungen mit vielen Requests, Responses und Datenbankobjekten würde sonst viel wiederholter Code entstehen. Durch die zentrale Konfiguration der Mappings lässt sich dieser Boilerplate-Code deutlich reduzieren. Das betrifft vor allem Controller und Service-

Klassen, da dort häufig zwischen internen und externen Datenstrukturen umgewandelt werden muss.

Die Entwickler entschieden sich für Mapster, weil dadurch die Zuordnung zwischen Entities und DTOs kompakter und übersichtlicher umgesetzt werden kann. Das verbessert die Lesbarkeit des Codes und spart an vielen Stellen wiederkehrende Implementierungsarbeit.

2.2.9 Swagger / OpenAPI

Ein wichtiges Einsatzgebiet von ASP.NET Core ist die Entwicklung von Web-APIs [12]. Dabei werden HTTP-Endpunkte bereitgestellt, die über Methoden wie `GET`, `POST`, `PUT` und `DELETE` angesprochen werden können [13]. Als Datenformat wird dabei meistens JSON verwendet [14, 15]. Dadurch eignet sich ASP.NET Core sehr gut als Backend für moderne Frontends wie Angular.

Architektonisch unterstützt ASP.NET Core eine klare Trennung von Verantwortlichkeiten. In vielen Projekten wird deshalb ein controllerbasierter Ansatz verwendet. Dabei übernehmen die Controller die HTTP-Kommunikation, während die eigentliche Fachlogik in separaten Service-Klassen umgesetzt wird [16]. Das hat den Vorteil, dass Änderungen an der Business-Logik nicht direkt mit der HTTP-Ebene vermischt sind. Dadurch steigt die Wartbarkeit des Backends.

3 Implementierung

In diesem Kapitel wird erklärt, wie ClarioAI implementiert wurde und welche Gedanken sich gemacht wurden im Prozess der Entwicklung und davor. Außerdem wird unter anderem die Architektur des Frontends und des Backends erklärt und begründet.

3.1 Frontend

3.1.1 Design und Prototyping

Bevor mit der technischen Implementierung des Frontends in Angular begonnen wurde, wurde ein visueller Entwurf der Benutzeroberfläche (UI) erstellt. Hierfür wurde das Design-Tool **Figma** genutzt. Die Entscheidung fiel auf Figma, weil es als moderner Industriestandard gilt und bereits Vorkenntnisse mit diesem Tool vorhanden waren, was die Designphase beschleunigte.

Die wesentlichen Ziele dieser Phase waren:

- **Strukturierung:**

  Festlegung des Layouts und der Navigationspfade (visualisiert durch Verknüpfungen innerhalb von Figma).

- **Design-System:**

  Definition einer konsistenten Designsprache, um ein einheitliches Erscheinungsbild sicherzustellen. Dabei orientierte man sich an den Farben des Lamie-Logos (siehe 1.6.2 Auftraggeber und 6.0.2 Lamie-Logo) orientiert.

- **Effizienz:**

  Durch die Trennung von Design und Umsetzung konnten gestalterische Fragen vorab geklärt werden. Dies verhinderte langwierige CSS-Anpassungen während der Logik-Implementierung in Angular und reduzierte die Fehleranfälligkeit im Entwicklungsprozess.

3.1.2 UI-Library

Im Rahmen der *Frontend-Implementierung* hat sich das Entwicklungsteam bewusst gegen den Einsatz von etablierten *UI-Libraries* wie *PrimeNG* oder *AngularMaterial* entschieden.

Diese Entscheidung basiert auf der Erfahrung, dass hochgradig vordefinierte Komponenten von *Libraries*, wie die oben genannten *Libraries*, oft zu einer eingeschränkten Flexibilität hinsichtlich des Designs führen. Um ein einzigartiges und konsistentes Design umsetzen zu können, entschied man sich bei ClarioAI deshalb für die eigenständige Entwicklung aller Komponenten mit *TailwindCSS* (siehe 3.1.7 Styling mit TailwindCSS) entschieden.

3.1.3 Architektur

Die Architektur folgt einer Feature-basierten Gruppierung. Ganz nach dem Prinzip von *Separation of Concerns*. Jede Angular-Komponente steht für sich selbst. Das Entwicklungsteam entschied sich jedoch gegen die Standard-Ordnerstruktur von Angular. Da diese bei vielen Komponenten schnell unübersichtlich wird. Deshalb wählten wir auf Empfehlung des Auftraggebers für folgende Struktur. Diese unterscheidet zwischen den Komponenten. Das führt zu schnellerer Navigation im Entwicklungsprozess, besserer Wartbarkeit, besserer Übersicht und besserer Lesbarkeit.

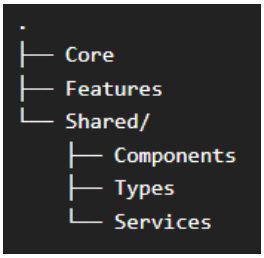


Abbildung 2: Angular Ordnerarchitektur der Komponenten

- **Core**

  Im Core-Ordner sind jene Komponenten enthalten, die, wie der Name schon verrät, zum Kern der Webapp gehören. Diese Komponenten, die auf jeder Seite angezeigt werden und nur einmal instanziiert werden. In der vorliegenden Implementierung enthält der Core-Ordner nur die Navigationsleiste (auch Navbar genannt).

- **Features**

  Hier sind jene Komponenten, die ganze Seiten darstellen. Diese Komponenten dienen also nur dazu, andere Komponenten zu verbinden, indem hier mehrere Komponenten eingebunden sind. Man könnte diese Komponenten auch als Container bezeichnen.

- **Shared**

  Dieser Ordner gilt als Wrapper-Ordner für jene Art von Komponenten, die in der App

mehrere Male instanziiert werden, oder andere Files, die in der App öfters benötigt werden.

- **Components**

  Hier sind jene Komponenten, die innerhalb einer Seite verwendet werden. Dazu gehören zum Beispiel Buttons oder Komponenten zur Anzeige von Daten. Diese Komponenten sind nicht fix an eine Seite gebunden, sondern können überall instanziiert werden.

- **Types**

  In diesem Ordner sind alle Interfaces definiert, die man in der App benötigt, weil Angular durch TypeScript starke Typisierung verlangt. Für jedes Interface gibt es eine eigene Datei. So gibt es zum Beispiel für jeden Fachbegriff, der in 2.1 (Grundlegende Fachbegriffe) erwähnt wird, jeweils eine Typescript-Datei, wo genau ein Interface definiert ist.

- **Services**

  Im Services-Ordner sind alle TypeScript-Files in denen *http-Requests* gesendet werden. Sie implementieren also die Schnittstelle zum Backend. Der Ordner agiert deswegen als *Data Access Layer*.

3.1.4 Seiten

Die Tabelle 1 biete eine Übersicht aller Seiten der Webapp.

Tabelle 1: Übersicht Webapplikation - Routen und Funktionen

Seite	`Route`	**Funktion**
Overview	`/overview`	Claim-Übersicht
Create Claim	`/claim/create`	Erstellung eines neuen Claims
Claim Details	`/claim/{claim_id}/details`	Detailansicht eines spezifischen Claims
Invoice Details	`/claim/{claim_id}/invoice/{invoice_id}`	Detailanzeige einer Invoice

- **Overview**

  Der Sinn dieser Seite ist es, den User:innen einen Überblick über alle Claims zu bieten.

Dazu wurde eine Tabelle entwickelt, die eine aggregierte Darstellung implementiert. Klicken die User:innen auf eine Zeile der Tabelle, so wird auf die Claim Detail-Seite dieses Claims geroutet. Die Aktion erfolgt über imperatives Routing. Im unteren Bereich der Tabelle gibt es auch einen Create Claim-Button, der auf die Create Claim-Seite verweist.

Ein wesentliches Feature dieser Seite ist die Filterlogik über der Tabelle. Diese kann durch kombinatorisches Suchen mit der Verbindung eines logischen AND und durch die Verwendung von HTML5-Input-Typen (Date, Text, Dropdown) die richtigen Claims aus der Datenbank finden. Dazu wird unter anderem der *ClaimService* verwendet.

- **Create Claim**

  Diese Seite ist, wie der Name schon sagt, für die Erstellung eines neuen Claims verantwortlich. Das Herz dieser Seite ist das Analysieren eines hochgeladenen Fotos. Nach der Analyse werden erkannte Textfelder im Formular darunter automatisch ausgefüllt. Die User:innen müssen nur mehr die Werte kontrollieren und am Ende bestätigen und den Claim erstellen.

- **Claim Detail**

  Auf diese Seite wird von der Overview-Seite verlinkt. Diese Seite präsentiert alle Attribute eines Claims und die hochgeladenen Files. Außerdem wird am Ende der Seite eine Übersicht zu den dazugehörigen Invoices angezeigt. Wenn die User:innen auf eine Zeile in dieser Tabelle in der aggregierten Darstellung von Invoices klicken, so werden sie weitergeleitet auf die Invoice Detail-Seite.

- **Invoice Detail**

  Hier werden alle Informationen einer Invoice angezeigt. Also das dazugehörige File, die einzelnen Invoices und Invoice Lines.

3.1.5 Komponenten

In diesem Kapitel werden die wichtigsten und interessantesten Angular-Komponenten genauer beschrieben. Weniger relevante Komponenten werden zugunsten der Übersichtlichkeit nicht explizit erwähnt, sind jedoch in der Architektur-Übersicht (siehe Kap. 3.1.3, Architektur) aufgeführt.

- **Core**
  - **Navbar**

    Diese Komponente stellt die zentrale Navigationsleiste der Applikation dar. Da sie in der `app.component.html` oberhalb des `router-outlet` platziert wurde, bleibt sie bei jedem Routenwechsel persistent sichtbar. Die Navbar ist zweigeteilt: Auf der linken Seite befindet sich das App-Logo (siehe Kap. 6.0.1, ClarioAI-Logo), während rechts die Hauptnavigation untergebracht ist.

- **Features**

  Die funktionale Aufteilung dieser Seiten wurde bereits in Kap. 3.1.4 (Seiten) beschrieben.
  - **Create Claim & Claim Detail**

    Technisch betrachtet teilen sich diese beiden Ansichten dieselbe Komponente. Um die User Experience zu optimieren, werden ähnliche logische Abläufe (Dateneingabe vs. Datenansicht) visuell konsistent dargestellt. Die Unterscheidung der Funktionalität erfolgt dynamisch anhand der aktuellen Route.

    In der `ngOnInit`-Lifecycle-Methode wird der Modus der Komponente ermittelt. Enthält die URL den Parameter "create", wird die Komponente in den Erstellungsmodus versetzt (leere Eingabefelder, aktiver Datei-Upload). Die Steuerung im Template erfolgt über die moderne Angular Control-Flow-Syntax `@if`.

```
<div class="flex w-full flex-col items-center justify-center p-4">
    @if(mode === 'Create') {
        <app-file-upload-component
            #fileUploadComponent
            class="w-full"
            [maxFiles]="10"
            [mode]="'upload'"
            [analysisLoading]="analysisLoading()"
            [analysisResult]="analysisResult()"
            (dataExtracted)="onClaimAnalyzed($event)">
        </app-file-upload-component>
    }
    @if(mode === 'Details') {
        <app-file-upload-component
            class="w-full"
            [maxFiles]="10"
            [mode]="'view'"
            [displayFiles]="displayFiles()">
        </app-file-upload-component>
    }
</div>
```

Listing 1: Dynamische Orchestrierung der Upload-Komponente mittels @if-Steuerstruktur

- **Shared**
  - **FileUploadComponent**

    Die `FileUploadComponent` ist das zentrale Modul zur Dateierfassung. Die Umsetzung basiert auf **Angular Signals**, was eine reaktive und performante UI-Aktualisierung garantiert. Die Komponente unterstützt PDF, PNG und JPEG mittels eines hybriden Rendering-Ansatzes:

    * **Bilddateien** werden direkt über `<img>`-Tags mit generierten *Object-URLs* angezeigt.
    * **PDF-Dokumente** werden in einem **Inline-Frame (iframe)** gekapselt, um die nativen Browser-Funktionen zu nutzen und Rendering-Fehler von der Hauptapplikation zu isolieren.

    Die Kernlogik verarbeitet verschiedene MIME-Types direkt in der Browser-Runtime. Dabei wird zur Laufzeit zwischen *ArrayBuffer* (für PDFs) und *DataURL* (für Bilder) unterschieden. Zur Vermeidung von Sicherheitsrisiken wie Cross-Site-Scripting (XSS) wird der `DomSanitizer` von Angular eingesetzt.

```
1  private handleFiles(files: File[]) {
2      const current = this.uploadedFiles();
3      const spaceLeft = this.maxFiles() - current.length;
4      const toAdd = files.slice(0, spaceLeft);
5
6      toAdd.forEach((file) => {
7          let kind: 'pdf' | 'image' | 'other' = 'other';
8          if (file.type === 'application/pdf') kind = 'pdf';
9          else if (file.type.startsWith('image/')) kind = 'image';
10
11         const reader = new FileReader();
12         reader.onload = () => {
13             const result = reader.result!;
14
15             const url = kind === 'pdf'
16                 ? this.sanitizer.bypassSecurityTrustResourceUrl(
17                     URL.createObjectURL(new Blob([result], { type: file.type
                           })) + '#toolbar=0'
18                   )
19                 : this.sanitizer.bypassSecurityTrustResourceUrl(result as
                       string);
20
21             this.uploadedFiles.update((f) => [...f, file]);
22             this.previewUrls.update((u) => [...u, url]);
23             this.fileTypes.update((t) => [...t, kind]);
24         };
25
26         if (kind === 'pdf') reader.readAsArrayBuffer(file);
27         else reader.readAsDataURL(file);
28     });
29 }
```

Listing 2: Asynchrones Handling von Dateitypen und Generierung sicherer Preview-URLs mittels FileReader

  - **FilterBox**

    Die `FilterBox` aggregiert verschiedene Eingabetypen (Text, Datum, Dropdown). Die Dropdown-Optionen werden mittels `@for` dynamisch generiert. Dieser Aufbau ermöglicht eine zukünftige Anbindung an eine API (z. B. das *Luis-System* der

Lamie), um Filteroptionen ohne Code-Änderungen zu aktualisieren (siehe Kap. 1.4, Zielsetzung).

```
<div class="flex min-w-0 flex-col">
    <label for="event" class="whitespace-normal break-words">Event:</label>
    <select
        name="event"
        id="event"
        [(ngModel)]="filter.eventType"
        class="rounded border-2 border-black">

        <option [ngValue]="null">--All--</option>

        @for(ev of lossEvents; track ev) {
            <option [ngValue]="ev">{{ ev }}</option>
        }
    </select>
</div>
```

Listing 3: Implementierung eines dynamischen Filters mittels Two-Way-Binding und moderner Control-Flow-Syntax

Die Filterkriterien werden gesammelt und per *Output-Signal* an die übergeordnete Komponente delegiert, welche den API-Call initiiert (Details dazu in Kap. 3.1.6). Diese Architektur wurde in Abstimmung mit dem Auftraggeber gewählt, um rechenintensive Filter-Operationen performant im Backend durchzuführen und so die User Experience zu wahren.

3.1.6 Services & API-Calls

In „Services“-Verzeichnis sind alle *Services*. Diese enthalten alle *API-Calls*, siehe auch 3.1.3 (Architektur).

- **Claim Service**

  In diesem Service werden alle *API-Calls* bezüglich *Claims* initiiert, wie im Code-Snippet der FilterBox (siehe Code Snippet in FilterBox 3.1.5 Komponenten), das hier eine Instanz vom `ClaimService` die Funktion `getClaimsWithFilter()` mit dem aggregierten `Filter`-Objekt aufruft. So sieht die Funktion im `Claim-Service` aus:

```
private readonly http = inject(HttpClient);
private readonly filterRequest = signal<Filter | undefined>(undefined);

filteredClaims = rxResource<Claim[], Filter>({
    params: this.filterRequest,
    stream: (filter: Filter) => {
        let params = new HttpParams();
        // Dynamisches Mapping der Filter-Attribute auf HTTP-Query-Parameter
        if (filter.lossDateFrom) params = params.set('lossDateFrom',
            filter.lossDateFrom.toISOString());
        if (filter.lossDateTo) params = params.set('lossDateTo',
            filter.lossDateTo.toISOString());
        if (filter.claimNumber) params = params.set('claimNumber',
            filter.claimNumber);
        if (filter.certificateNumber) params = params.set('certificateNumber',
            filter.certificateNumber);
        if (filter.coverType) params = params.set('coverType', filter.coverType);
        if (filter.eventType) params = params.set('eventType', filter.eventType);
        if (filter.partner) params = params.set('partner', filter.partner);
        if (filter.claimStatus) params = params.set('claimStatus',
            filter.claimStatus);
        if (filter.processingStatus) params = params.set('processingStatus',
            filter.processingStatus);
```

```
18
19          return this.http.get<Claim[]>(`${this.baseUrl}/claims/filtered`, { params });
20      },
21      defaultValue: [],
22  });
```

Listing 4: Reaktive Datenbeschaffung mittels rxResource und dynamischer HttpParams-Konfiguration

Es ist ersichtlich, dass mithilfe des `Filter`-Objekts die Parameter zusammengebaut werden und am Ende ein *http-GET-Request* an das *Backend* gesendet wird mit den zuvor erstellten Parametern. Diese letzten Zeilen sind typisch für den `ClaimService` und generell alle *Services*, weil genau die *API-Calls* mittels injizierter *http-Dependencies* den Kern eines jeden *Services* darstellen.

- **Relative URL**

  Wie im *Code-Snippet* ersichtlich, wird hier `this.baseUrl` verwendet. Diese Variable wurde einheitlich mit `/api"` definiert. Der entscheidende Vorteil ist die *Portabilität*: Die Applikation ist unabhängig von der tatsächlichen Adresse des *Host-Servers* und muss beim Wechsel von der lokalen Entwicklung auf einen Server nicht angepasst werden.

  Während der Entwicklung wird dies über einen *Angular-Entwicklungsproxy* aufgelöst. Im produktiven Betrieb leitet ein sogenannter *Reverse-Proxy* wie *Nginx* oder *Apache* die Anfragen unter dem Pfad `/api` an den *Backend-Server* weiter.

  Dadurch werden Probleme mit *CORS* (*Cross-Origin Resource Sharing*), also der gemeinsamen Nutzung von Ressourcen unterschiedlicher Ursprünge, verhindert.

- **Types**

  Im *Types*-Ordner werden alle *Interfaces* beschrieben, welche benötigt werden. Mehr dazu auch unter 3.1.3 (Architektur).

  - **Filter-Type**

    Der `Filter-Type` ist das *Interface*, welches die von den User:innen ausgewählten Filter-Optionen speichert. Somit steht zum Beispiel von einem *Dropdown* der `ngValue` der ausgewählten `<option>` in diesem Objekt.

```
1  export interface Filter {
2      lossDateFrom: Date | null;
3      lossDateTo: Date | null;
4      claimNumber: string | null;
5      certificateNumber: string | null;
6      coverType: string | null;
7      eventType: string | null;
8      partner: string | null;
9      claimStatus: string | null;
10     processingStatus: string | null;
11 }
```

Listing 5: Definition des Filter-Interfaces zur Gewährleistung der Typsicherheit

Zu jedem *Interface*, welches man in einer extra dafür erstellten Datei definiert, gehört auch das Wort `export`. Nur dadurch kann man das *Interface* an anderen Stellen importieren und verwenden. Wie ersichtlich handelt es sich hier um kein komplexes *Interface*. Es gibt keine Verschachtelungen und der Umfang der Attribute hält sich ebenfalls in Grenzen. Ganz anders sieht das im folgenden Typ aus.

– **Claim Type**

Wir sehen hier den zentralen Typ der App. Hier werden alle *Entitäten* zusammengeführt, ähnlich wie bei 3.3 (Datenbank). Man kann das Konzept auch mit *Single Source of Truth* vergleichen. Wir sehen also, im Vergleich zum vorher erwähnten `Filter-Type` werden nicht nur primitive Datentypen verwendet, sondern ganze *Interfaces*. So ist zum Beispiel `ClaimState` kein einfacher *Boolean*, sondern ein eigenständiges *Interface*. Diese *Kapselung* führt zu mehr Übersicht.

```
export interface Claim {
    certificateNumber: string;
    claimNumber: string;
    assistanceCaseNumber: string | null;
    lossDate: string | null;
    place: string;
    country: string;
    lossEvent: string | null;
    eventDescription: string;
    assistanceCompanyCallDate: string;
    noCallReason: string;
    reimbursableAmount: string | null;
    bankAccountHolderName: string;
    currentBankAccountNumber: string;
    bankName: string;
    claimStatus: ClaimState;
    processingStatus: string | null;
    createdAt: string;
    lastEditedAt: string | null;
    insuredDetail: Insured;
    beneficiaryDetail: Beneficiary;
    contract: Contract;
    trip: Trip;
}
```

Listing 6: Zentrales Claim-Interface zur Abbildung der Schadensfalldaten

3.1.7 Styling mit TailwindCSS

- **ButtonComponent (Shared)**

Diese Komponente dient als generischer, wiederverwendbarer Button innerhalb der gesamten Applikation. Sie nutzt **Lucide-Angular** für eine konsistente Icon-Sprache und **Tailwind CSS** für ein responsives Design. Die Anzeige des Icons erfolgt deklarativ über die neue Angular-Control-Flow-Syntax.

```
    <button
        class="border-primary box-border flex h-full cursor-pointer items-center
            justify-center rounded-md border-2 bg-white p-2">
        <h1 class="text-primary">{{ text() }}</h1>

        @if(icon()) {
            <lucide-angular
                [name]="icon()"
                class="text-primary h-full">
```

```
9                </lucide-angular>
10           }
11       </button>
```

Listing 7: Generischer Button mit optionalem Icon-Rendering und Tailwind-Styling

3.1.8 Responsivität und Adaptive Layouts

Fokus auf *Desktop-First-Umgebungen*

Das *User-Interface* der App wurde mit einem klaren Fokus auf die *Desktop-Nutzung* entwickelt. Das begründet sich primär dadurch, dass die Arbeit ja als *Prototyp* einer Erweiterung eines bestehenden Systems besteht, welches in der *Lamie* ausschließlich auf *Desktop-Geräten* verwendet wird. Das bedeutet, die Analyse und Bearbeitung von komplexen *Claims* erfolgt in der Regel nicht auf Smartphones. Weil das Entwicklerteam auf diese Situation Rücksicht nehmen will, entschied es sich für einen *Desktop-First-Ansatz*.

Zur Realisierung des adaptiven Verhaltens wurde *TailwindCSS* verwendet (siehe 2.2.2 Tailwind CSS). So werden anstatt von hardcodierten Container-Maßen flexible Container verwendet. Dies ermöglicht das angenehme Benutzen der Webseite auf den üblichen *Monitorgrößen*.

- **Responsivität von der `FilterBoxComponent`**
  Wie in 3.1.5 Komponenten schon erwähnt, besteht diese Komponente aus mehreren Eingabefeldern. Damit man bei mehreren Displaygrößen alle Felder gut sehen und bedienen kann, haben die Entwickler die Felder in ein *Grid* eingefügt, welches seine Zeilen- und Spaltenanzahl auf die verfügbare Breite anpasst.

- **Technische Umsetzung**
  In diesem Fall wurde das mit der Verwendung von `repeat(auto-fit, minmax(240px, 1fr))` umgesetzt. Das ist entscheidend für das Verhalten.

  - **Minimale Breite**
    Jedes Feld (zum Beispiel die *Claim Number*) erhält eine minimale Breite von `240px`. Das verhindert, dass das Feld auf kleinen Displays unerkennbar wird.

  - **Automatisches Füllen (*auto-fit*)**
    Der Browser der User:innen der Webseite berechnet eigenständig, wie viele dieser `240px` breiten Spalten nebeneinander im Container Platz finden.

  - **Fluider Platzgewinn (`...1fr`)**
    Der verbleibende Restplatz wird gleichmäßig auf alle Spalten verteilt. Das führt dazu, dass das *Grid* die gesamte Breite des Containers auffüllt.

3.1.9 Frontend Implementierung - Zusammenfassung

Zusammengefasst lässt sich sagen, dass bei der Architektur auf eine klare Modularität geachtet wird. So wird klar zwischen der Darstellung (Angular-Komponenten) und der Business-Logic (Services) unterschieden. Außerdem wird bei den Komponenten auf Shared Components gesetzt, welche die Wiederverwendbarkeit verbessern. Schnelles und konsistentes Umsetzen des Designs wird mit Hilfe des TailwindCSS-Frameworks implementiert. Weiters wurden die Daten aus dem Backend mittels Services beschafft. Hier wird auf eine relative URL gesetzt, die die Portabilität verbessert. Das Ganze wurde in Angular Version 20 entwickelt.

3.2 Backend

3.2.1 Cloud-orientierte Architektur

Die Backend-Architektur dieser Arbeit ist klar cloud-orientiert aufgebaut. Strukturierte Daten werden in einer relationalen Azure SQL-Server-Datenbank gespeichert, Dateien werden in Azure Blob Storage abgelegt, und für die Dokumentanalyse kommen Azure Document Intelligence und Azure OpenAI zum Einsatz [26, 29, 24, 38]. Dadurch wird Dateispeicherung und Datenverarbeitung nicht in einer einzelnen Komponente vermischt, sondern auf spezialisierte Dienste aufgeteilt.

Das hat den Vorteil, dass jede dieser Technologien genau dort eingesetzt wird, wo sie ihre Stärken hat. Die Datenbank eignet sich für strukturierte relationale Daten, Blob Storage für Dateien und die Azure-AI-Dienste für OCR und sprachmodellbasierte Nachbearbeitung. Erst durch diese Kombination entsteht eine komplette Backend-Pipeline, in der ein Dokument hochgeladen, gespeichert, analysiert, strukturiert und schließlich in die Datenbank übernommen werden kann [24, 29, 38].

Die Entwickler entschieden sich für diese Architektur, weil es vom Auftraggeber Lamie so vorgegeben wurde und es auch einige Azure spezialisten in dem Unternehmen gab die uns tatkräftig bei jeglichen Fragen zur Seite standen.

3.2.2 Projektstruktur

Die Backend-Applikation ist als ASP.,NET Core Web API umgesetzt (`.NET 8`, `C# 12`) und folgt dem Prinzip *Separation of Concerns*. Die Verantwortlichkeiten sind dabei auf Controller (HTTP-Schnittstelle), Services (Business-Logik), Datenzugriff (Entity Framework Core) sowie Datenmodelle/DTOs aufgeteilt. Zusätzlich werden Konfiguration, Mapping und Swagger-spezifische Hilfslogik in eigenen Ordnern gekapselt. Dadurch bleibt die Codebasis übersichtlich,

gut wartbar und Erweiterungen (z.,B. neue Endpoints oder neue Extraktionslogik) können ohne starke Seiteneffekte umgesetzt werden.

- **Controllers** In diesem Ordner befinden sich alle API-Endpunkte. Controller kapseln die HTTP-Kommunikation (Routing, Request/Response, Statuscodes) und delegieren die eigentliche Logik an Services. In der vorliegenden Implementierung existieren u.,a. Controller für Claims und Invoices (z.,B. `ClaimsController`, `InvoicesController`) sowie für Dateioperationen (z.,B. `ClaimFileController`, `InvoiceFilesController`) und administrative Vorgänge (z.,B. `ClaimDeletionController`, `ClaimFilterController`).

- **Services**
  Hier ist die Business-Logik zentral gebündelt. Services implementieren die Verarbeitungsschritte (z. B. Analyse/Extraktion/Verarbeitung von Dokumenten) und kapseln die Integration externer Dienste. Die Services werden über Dependency Injection eingebunden (Interfaces wie `IClaimAnalysisService`, `IClaimExtractionService`, `IInvoiceProcessingService`) und in `Program.cs` als *Scoped* registriert, um pro Request eine konsistente Verarbeitung zu gewährleisten.

- **Data**
  Dieser Ordner bildet den Data-Access-Layer über Entity Framework Core. Kernkomponente ist der `ClaimsInvoiceContext` (DbContext), der die Datenbankanbindung und die Entäten verwaltet. Die eigentliche Konfiguration erfolgt in `Program.cs` mit `UseSqlServer(...)` und einer Connection-String-Definition in `appsettings.json`.

- **Models**
  In `Models` liegen die Domänen- und Persistenzobjekte (EF-Core-Entities), z. B. `Claim`, `Invoice`, `InvoiceLineItem`, `Trip`, `Contract` sowie Detailobjekte wie `InsuredPersonDetail` und `BeneficiaryPersonDetail`. Zusätzlich existieren wiederverwendbare Interfaces (z. B. `Models/Interfaces/ICertificateEntity`) zur Vereinheitlichung gemeinsamer Eigenschaften.

- **DTOs**
  DTOs (*Data Transfer Objects*) dienen als stabile ÄPI-Vertragsmodelle"für Requests und Responses und entkoppeln die HTTP-Schnittstelle von den EF-Entities. Beispiele sind Analyse- und Update-Objekte. Dadurch können externe API-Änderungen kontrolliert erfolgen, ohne direkt das Persistenzmodell zu beeinflussen.

- **Mappings**
  Dieser Ordner enthält die Zuordnung zwischen Entities und DTOs mittels Mapster. In

`MappingConfig` werden die Mapping-Regeln zentral definiert und in `Program.cs` initialisiert. Dadurch bleibt Konvertierungslogik konsistent und mehrfach verwendbar.

- **Settings**
  Hier befinden sich typisierte Konfigurationsklassen (z. B. `AzureSettings`), die an die Konfiguration (`appsettings.json`) gebunden werden. So werden Azure-Endpunkte, Keys und weitere Parameter strukturiert und typsicher bereitgestellt.

- **Filters**
  In `Filters` liegt projektspezifische Filter-/Hilfslogik für Swagger/OpenAPI. In der Implementierung wird z. B. ein `FileUploadOperationFilter` verwendet, um File-Upload-Endpunkte korrekt in Swagger zu dokumentieren.

- **Migrations**
  Dieser Ordner enthält die automatisch generierten EF-Core-Migrationen. Die Migrationen dokumentieren Schemaänderungen und ermöglichen reproduzierbare Datenbank-Updates.

- **Root-Konfigurationsdateien**
  `Program.cs` enthält die Startup-/DI-Konfiguration (Controller, CORS, Swagger, JSON-Serialization sowie Registrierung von Azure-Client[0]). `appsettings.json` beinhaltet die Infrastrukturkonfiguration (Azure, Connection Strings). Lokale Startparameter sind in `Properties/launchSettings.json` definiert.

3.2.3 Entitäten, Modelklassen und Data Mapping

Die Persistenzschicht des Backends basiert auf Entity Framework Core und verwendet Code-First-Modelklassen als Grundlage für das relationale Datenbankschema [25]. Die sind im Ordner `Models` definiert und sind die zentralen Fachobjekte des Systems, zum Beispiel `Claim` und `Invoice`. Der Datenzugriff erfolgt über den `DbContext`, der sowohl die `DbSet`-Sammlungen als auch Beziehungen und Indizes verwaltet.

Für die API-Kommunikation werden zusätzlich DTOs verwendet. DTO steht für *Data Transfer Object* und dient dazu, das Persistenzmodell von den Request- und Response-Strukturen zu trennen [39]. Das hat den Vorteil, dass das interne Datenmodell stabil bleiben kann, auch wenn sich die API nach außen weiterentwickelt. Die Umwandlung zwischen Entities und DTOs wird in diesem Projekt zentral mit Mapster umgesetzt [37].

[0] `DocumentAnalysisClient`, `AzureOpenAIClient` und `BlobServiceClient`

DTOs Die DTOs im Ordner `DTOs` definieren die Datenstrukturen, die nach außen über die API sichtbar sind. Dadurch kann das interne Persistenzmodell von der externen API getrennt werden. Das hat den Vorteil, dass man Datums- oder Zahlenformate anpassen oder Payloads reduzieren kann, ohne direkt die Datenbankstruktur ändern zu müssen.

```
1  TypeAdapterConfig<ClaimUpdateDto, Claim>
2      .NewConfig()
3      .Map(dest => dest.LossDate,
4           src => !string.IsNullOrWhiteSpace(src.LossDate) ? ParseDateOnly(src.LossDate) :
                   null)
5      .Map(dest => dest.AssistanceCompanyCallDate,
6           src => !string.IsNullOrWhiteSpace(src.AssistanceCompanyCallDate) ?
                   ParseDateOnly(src.AssistanceCompanyCallDate) : null)
7      .Map(dest => dest.ReimbursableAmount,
8           src => !string.IsNullOrWhiteSpace(src.ReimbursableAmount) ?
                   ParseDouble(src.ReimbursableAmount) : null)
9      .IgnoreNullValues(true);
```

Listing 8: Mapster

Die Mapping-Konfiguration wurde zentral in `Mappings/MappingConfig.cs` umgesetzt und wird beim Start der API in `Program.cs` über `MappingConfig.ConfigureMappings()` initialisiert. In den Controllern erfolgt die Anwendung anschließend mit `dto.Adapt(...)` (z. B. bei `CreateClaim` und `UpdateClaim` im `ClaimsController`).

Für das Update-Mapping von `ClaimUpdateDto` auf `Claim` wurden Typkonvertierungen implementiert, da mehrere Eingabefelder als Strings übertragen werden (z. B. Datum und Betrag), im Datenmodell jedoch als typisierte Werte gespeichert sind (`DateOnly?`, `double?`). Zusätzlich sorgt `IgnoreNullValues(true)` dafür, dass bei Teilupdates nur tatsächlich gesendete Werte übernommen werden und bestehende Daten nicht durch `null` überschrieben werden. Dadurch entsteht ein PATCH-ähnliches Update-Verhalten innerhalb eines `PUT`-Workflows.

3.2.4 API-Endpunkte

Die Backend-Schnittstelle ist als REST-API umgesetzt und wird über ASP. NET Core Controller bereitgestellt. Alle Endpunkte sind unter dem Pfadpräfix `/api` erreichbar. Die Controller sind so aufgebaut, dass sie Aufgaben wie (Routing, Statuscodes, Validierung) übernehmen und die Logik an Services weitergeben. Für File-Uploads werden `multipart/form-data`-Requests verwendet; für strukturierte Daten (z. B. Claim-/Invoice-Erstellung) JSON-Payloads. Die API ist zusätzlich über Swagger/OpenAPI dokumentiert (`/swagger`).

Claim-Endpunkte (`/api/claims`)

- **POST `/api/claims/analyze`**
  Analysiert hochgeladene Claim-Dokumente (PDF/JPG/PNG) und extrahiert relevante

Felder.

Request: `multipart/form-data` mit Feld `files` (Liste von Dateien, bis 100 MB).

Response: extrahierte Claim-Daten (z. B. `CertificateNumber`, `ClaimNumber`); bei Fehlern `400 BadRequest`.

- **POST `/api/claims/create`**

  Legt einen neuen Claim in der Datenbank an (inkl. optionaler Detaildaten wie Versicherte/r, Begünstigte/r, Contract, Trip).

  Request: JSON (`ClaimUpdateDto`).

  Response: `201 Created` inkl. Location auf den neuen Datensatz (`/api/claims/{certificate}`).

- **GET `/api/claims/{certificate}`**

  Liefert einen Claim anhand der Zertifikatsnummer. Zusätzlich werden die zugehörigen Detaildaten (Insured/Beneficiary/Contract/Trip) nachgeladen und in das Ergebnis eingebettet.

  Response: `GetClaimDto` oder `404 NotFound`.

- **GET `/api/claims`**

  Liefert alle Claims, absteigend nach `CreatedAt` sortiert. Pro Claim werden die Detaildaten ergänzend geladen.

  Response: Liste von `GetClaimDto`.

- **PUT `/api/claims/{certificate}`**

  Aktualisiert einen bestehenden Claim (und führt ein Upsert für zugehörige Detail-Entities durch).

  Request: JSON (`ClaimUpdateDto`); Zertifikatsnummer in URL und Body muss übereinstimmen.

  Response: `204 NoContent`, ansonsten `400` oder `404`.

Claim-Filter-Endpunkt (`/api/claims/filtered`)

- **GET `/api/claims/filtered`**

  Unterstützt serverseitige Filterung von Claims über Query-Parameter.

  Query-Parameter: `lossDateFrom`, `lossDateTo`, `coverType`, `eventType`, `partner`, `claimStatus`, `processingStatus`.

  Zusätzlich ist Filterung über Contract-Eigenschaften (`Cover`, `Partner`) implementiert.

  Response: Liste von `GetClaimDto` inkl. geladener Detaildaten.

Claim-Löschung (`/api/claims/{certificate}`)

- **DELETE `/api/claims/{certificate}`**
  Löscht einen Claim und dazugehörige Daten: ClaimFiles, InvoiceFiles, Invoices, LineItems sowie Detail-Entities (Insured/Beneficiary/Contract/Trip). Zusätzlich werden Blob-Dateien im Azure Storage entfernt.
  Response: `204 NoContent` oder `404 NotFound`; bei internen Fehlern `500`.

Claim-Dateien (`/api/claims/{certificate}/files`)

- **POST `/api/claims/{certificate}/files`**
  Upload von Dateien zum Claim in Azure Blob Storage (Container `claim-uploads`).
  Request: `multipart/form-data` mit `files`.
  Response: Metadaten inkl. `downloadEndpoint`.
- **GET `/api/claims/{certificate}/files`**
  Listet gespeicherte Claim-Dateien (inkl. `downloadEndpoint`).
- **GET `/api/claims/{certificate}/files/{id}/view`**
  Liefert die Datei inline (z. B. PDF oder Bild) für die direkte Anzeige im Browser.
- **DELETE `/api/claims/{certificate}/files/{id}`**
  Löscht eine Claim-Datei sowohl aus der DB als auch aus dem Blob Storage.

Invoice-Endpunkte (`/api/invoices/{certificate}`)

- **POST `/api/invoices/{certificate}/analyze`**
  Analysiert eine hochgeladene Rechnung (Dateiformate: PDF/JPG/PNG) und extrahiert Rechnungsdaten/Positionen.
  Request: `multipart/form-data` über `InvoiceAnalyzeRequestDto` (enthält `Files` und Analyseparameter).
  Response: Analyse-Ergebnis oder `404`, wenn der Claim nicht existiert.

- **POST `/api/invoices/{certificate}/create`**
  Legt eine neue Rechnung für einen Claim an.
  Request: JSON (`InvoiceCreateDto`); Zertifikatsnummer in Route und Body muss übereinstimmen.
  Response: `201 Created` mit `InvoiceDto`.

- **PUT `/api/invoices/{certificate}/{invoiceId}`**
  Aktualisiert eine bestehende Rechnung.
  Request: JSON (`InvoiceUpdateDto`); Route-Parameter müssen zu DTO-Werten passen.
  Response: `204 NoContent` oder `404`.

- **DELETE `/api/invoices/{certificate}/{invoiceId}`**
  Löscht eine Rechnung inkl. LineItems sowie zugehörige InvoiceFiles (und entfernt diese im Blob Storage).

- **GET `/api/invoices/{certificate}`**
  Listet alle Rechnungen zu einem Claim (inkl. LineItems), sortiert nach `CreatedAt` absteigend.

- **GET `/api/invoices/{certificate}/{invoiceId}`**
  Liefert eine einzelne Rechnung inkl. LineItems.

Invoice-Dateien (`/api/invoices/{certificate}/{invoiceId}/files`)

- **POST `/api/invoices/{certificate}/{invoiceId}/files`**
  Upload von Rechnungsdateien in Azure Blob Storage (Container `invoice-uploads`), Pfadschema `{certificate}/{invoiceId}/{fileName}`.
  Response: Metadaten inkl. `downloadEndpoint`.

- **GET `/api/invoices/{certificate}/{invoiceId}/files`**
  Listet alle Dateien zu einer Rechnung.

- **GET `/api/invoices/{certificate}/{invoiceId}/files/{id}/view`**
  Liefert die Datei inline (PDF/Bild) zur Anzeige.

- **DELETE `/api/invoices/{certificate}/{invoiceId}/files/{id}`**
  Löscht die Datei in Blob Storage und den DB-Eintrag.

3.2.5 Services und Business-Logik

Die Service-Schicht bildet das Kernstück des Backends. Während Controller primär HTTP-spezifische Aufgaben wie Routing, Request-Binding und Statuscodes übernehmen, wird die eigentliche Verarbeitungslogik in dedizierten Services gemacht. Diese Trennung folgt dem Prinzip *Separation of Concerns* und verbessert Wartbarkeit, Testbarkeit und Erweiterbarkeit.

Im der Solution sind insbesondere folgende Services relevant:

- **ClaimExtractionService**
  Verantwortlich für die Extraktion strukturierter Claim-Daten aus hochgeladenen Dokumenten.

- **InvoiceProcessingService**
  Verantwortlich für die Analyse von Rechnungsdokumenten und die Rückgabe strukturierter Invoice-Daten inklusive Positionen.

- **ClaimAnalysisService**
  Unterstützt die rohe OCR-basierte Auswertung von Dokumentinhalten.

Die Services werden per Dependency Injection eingebunden. Dadurch kann die Implementierung bei Bedarf ausgetauscht werden, ohne Controller-Code anzupassen. Zusätzlich wird so eine klare Kapselung erreicht. Änderungen an Extraktions- oder Analyseprozessen bleiben lokal in der Service-Schicht.

3.2.6 Dokumentanalyse-Pipeline (OCR + LLM)

Die Dokumentanalyse folgt einer mehrstufigen Pipeline. Ziel ist die Umwandlung unstrukturierter Eingabedokumente (z. B. PDF, JPG, PNG) in strukturierte Datenobjekte, die direkt im Frontend weiterverarbeitet werden können.

Die Verarbeitung lässt sich in folgende drei Schritte unterteilen:

1. **OCR-Extraktion**
   Der Textinhalt der hochgeladenen Dokumente wird mit Azure Document Intelligence ausgelesen.

2. **Prompt-basierte Strukturierung**
   Der extrahierte Rohtext wird an ein Sprachmodell (Azure OpenAI) übergeben. Über ein streng definiertes Prompt-Format wird die Ausgabe in ein erwartetes JSON-Schema überführt.

3. **Deserialisierung und Typprüfung**

   Die JSON-Antwort wird serverseitig in DTOs überführt und auf Verarbeitbarkeit geprüft.

Dieser Ansatz kombiniert die Stärken beider Technologien: OCR für robuste Texterfassung und LLM-basierte Auswertung für semantische Strukturierung. Dadurch können auch variierende Dokumentlayouts verarbeitet werden.

Modellauswahl für die OCR-Nachverarbeitung

Für die eigentliche Texterkennung wird in dieser Arbeit Azure Document Intelligence verwendet. Das Document-Modell extrahiert dabei den Text aus den Dokumenten [20]. Danach folgt eine nachgelagerte Verarbeitung des OCR-Texts durch ein Sprachmodell. Dieses Modell übernimmt nicht mehr die optische Zeichenerkennung selbst, sondern die weitere Strukturierung, Normalisierung und Interpretation des bereits extrahierten Textes.

Für diesen Schritt wurden die Modelle o4-mini und GPT-5 mini betrachtet. o4-mini ist ein kompaktes Reasoning-Modell mit einem Kontextfenster von 200 000 Tokens und maximal 100 000 Ausgabetokens [40]. GPT-5 mini besitzt dagegen ein größeres Kontextfenster von 400 000 Tokens und kann bis zu 128 000 Ausgabetokens verarbeiten [41]. Dadurch bietet GPT-5 mini mehr Reserven, wenn OCR-Texte besonders lang, uneinheitlich, fehlerhaft oder mehrsprachig sind.

Auch bei den Kosten gibt es Unterschiede. o4-mini kostet laut OpenAI $1.10 pro 1 Million Input-Tokens und $4.40 pro 1 Million Output-Tokens [40]. GPT-5 mini kostet $0.25 pro 1 Million Input-Tokens und $2.00 pro 1 Million Output-Tokens [41]. Damit ist GPT-5 mini auf Basis der aktuellen API-Preise sowohl beim Input als auch beim Output günstiger. Tabelle 2 zeigt die wichtigsten Unterschiede der beiden Modelle.

Tabelle 2: Vergleich von o4-mini und GPT-5 mini für die OCR-Nachverarbeitung

Kriterium	**o4-mini**	**GPT-5 mini**
Kontextfenster	200 000	400 000
Maximale Ausgabetokens	100 000	128 000
Inputkosten pro 1 Mio. Tokens	$1.10	$0.25
Outputkosten pro 1 Mio. Tokens	$4.40	$2.00

Zur Berechnung der API-Kosten kann für den hier betrachteten Fall folgende Formel verwendet werden:

$$C_m = \frac{p_{\text{in},m} \cdot T_{\text{in}} + p_{\text{out},m} \cdot T_{\text{out}}}{10^6}$$

Dabei steht C_m für die Kosten einer Anfrage beim Modell m, T_{in} für die Anzahl der Input-Tokens und T_{out} für die Anzahl der Output-Tokens.

Für o4-mini ergibt sich damit:

$$C_{\text{o4-mini}} = \frac{1.10 \cdot T_{\text{in}} + 4.40 \cdot T_{\text{out}}}{10^6}$$

Für GPT-5 mini ergibt sich:

$$C_{\text{5-mini}} = \frac{0.25 \cdot T_{\text{in}} + 2.00 \cdot T_{\text{out}}}{10^6}$$

Zu Beginn der Entwicklungsphase wurde dennoch zunächst o4-mini verwendet. Der Grund dafür war nicht, dass GPT-5 mini fachlich ungeeignet gewesen wäre, sondern der zeitliche Projektkontext: In der ersten Projektphase im Sommer 2025 existierte GPT-5 mini noch nicht. Erst im späteren Verlauf der Arbeit konnte dieses Modell in den Vergleich aufgenommen werden. Die ursprüngliche Wahl von o4-mini war damit historisch bedingt und nicht das Ergebnis eines direkten Vergleichs mit GPT-5 mini.

Zur besseren Messung der Unterschiede wurde anschließend ein Benchmark durchgeführt, der beide Modelle direkt im Workflow auf Basis von benötigter Zeit, verwendeten Tokens, Kosten und Qualität des Outputs verglich. Dazu wurde derselbe durch Azure Document Intelligence extrahierte OCR-Text jeweils drei Mal an beide Modelle übergeben. Tabelle 3 zeigt die wichtigsten Ergebnisse dieses Benchmarks.

Tabelle 3: Benchmark-Ergebnisse von o4-mini und GPT-5 mini

Kriterium	**o4-mini**	**GPT-5 mini**
Durchschnittliche Latenz	29 302 ms	15 337 ms
Minimale Latenz	27 762 ms	13 528 ms
Maximale Latenz	31 123 ms	17 016 ms
Ø Input-Tokens	1 225	1 225
Ø Output-Tokens	2 497,3	1 248,7
Ø Gesamt-Tokens	3 722,3	2 473,7
Geschätzte Kosten pro Anfrage	$0.012336	$0.002804
Wörtliche Genauigkeit	hoch	hoch
Semantische Normalisierung	gut	sehr gut

Die Messung zeigt, dass GPT-5 mini im konkreten Test deutlich schneller war. Die durchschnittliche Latenz lag bei 15 337 ms, während o4-mini im Mittel 29 302 ms benötigte. Damit war GPT-5 mini in diesem Benchmark nahezu doppelt so schnell. Gleichzeitig verwendeten beide Modelle gleich viele Input-Tokens, GPT-5 mini erzeugte jedoch deutlich weniger Output-Tokens.

Im Mittel lagen die Output-Tokens bei 1 248,7 gegenüber 2 497,3 bei o4-mini. Dadurch war auch die Gesamtzahl der Tokens bei GPT-5 mini deutlich geringer.

Dieser Unterschied wirkte sich direkt auf die Kosten aus. Auf Basis der gemessenen durchschnittlichen Tokenanzahl ergaben sich für o4-mini geschätzte Kosten von $0.012336 pro Anfrage, für GPT-5 mini dagegen nur $0.002804. Im konkreten Benchmark war GPT-5 mini damit nicht nur schneller, sondern auch deutlich günstiger.

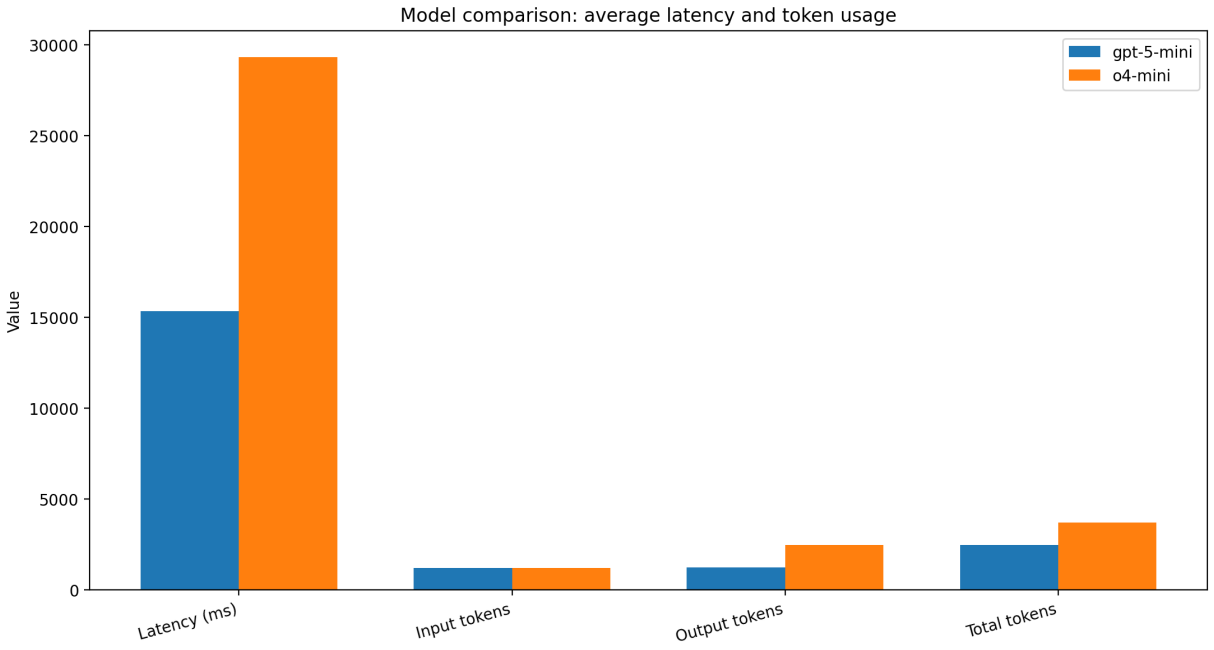


Abbildung 3: Vergleich der durchschnittlichen Latenz sowie der durchschnittlichen Input-, Output- und Gesamt-Tokens beider Modelle

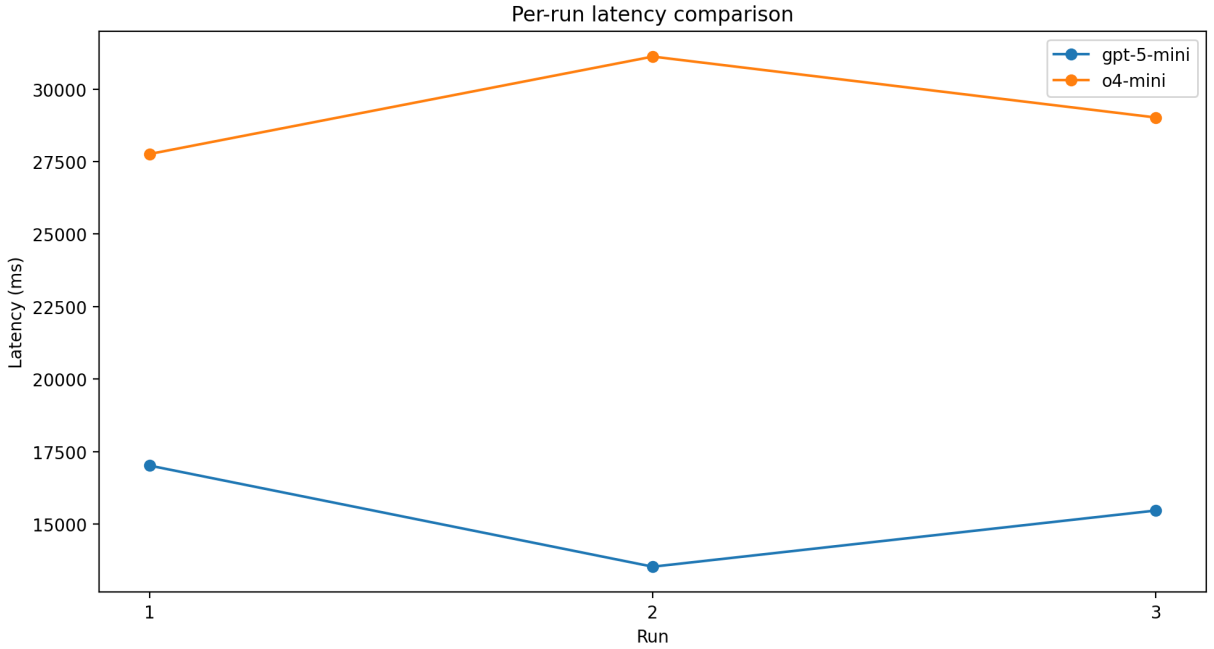


Abbildung 4: Latenzvergleich der drei Benchmark-Durchläufe

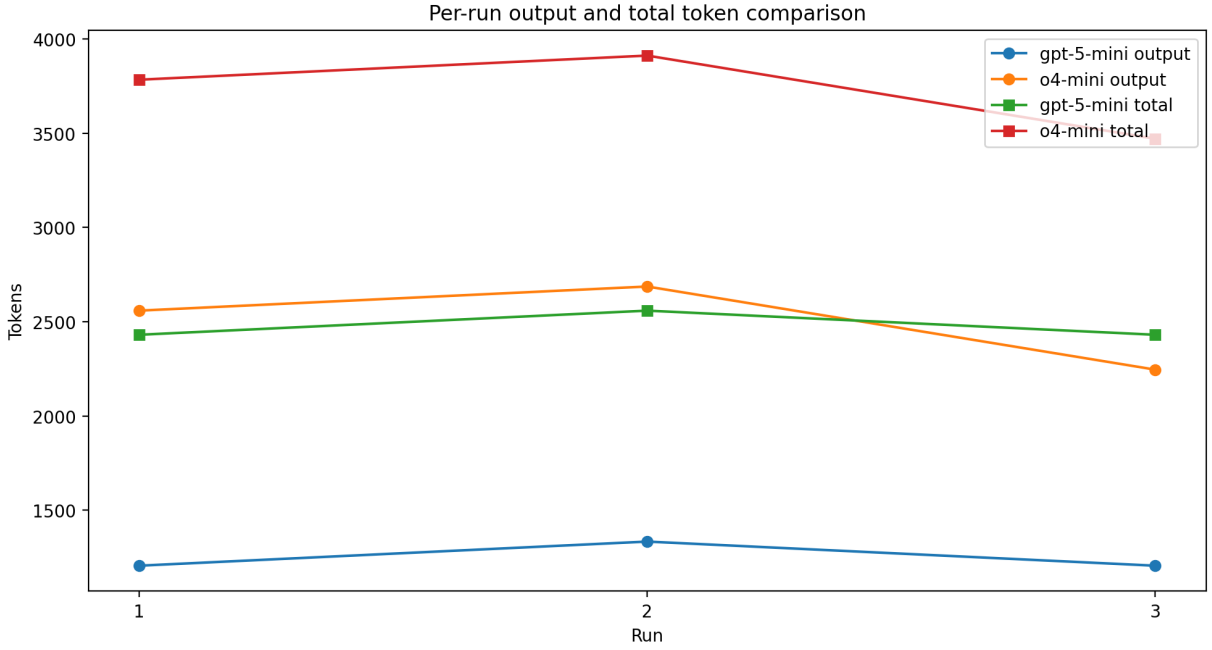


Abbildung 5: Vergleich der Output- und Gesamt-Tokens über alle drei Benchmark-Durchläufe

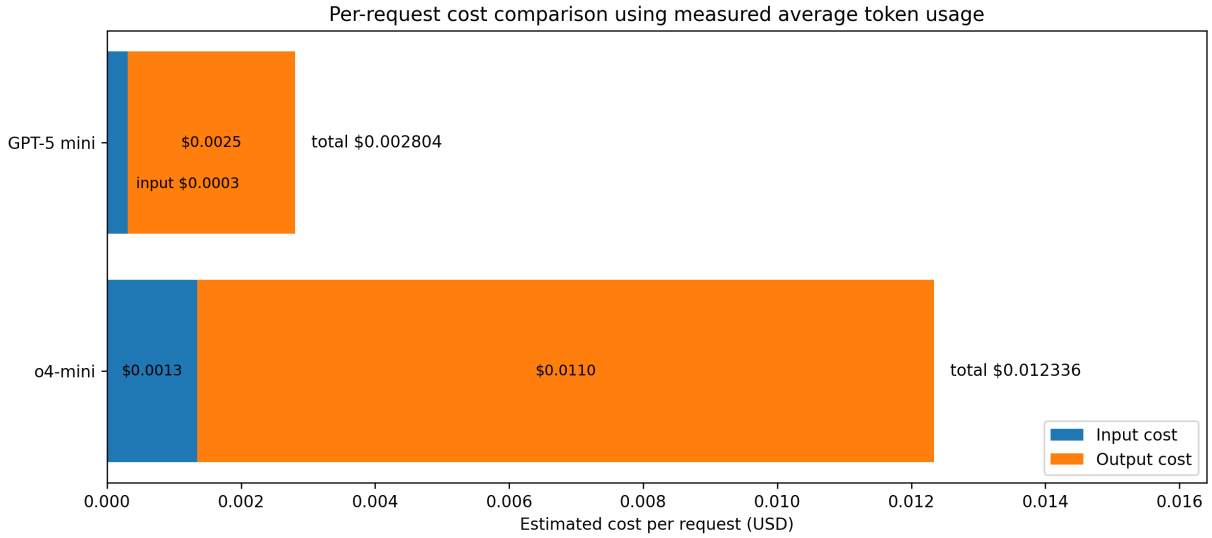


Abbildung 6: Geschätzte Kosten pro Anfrage auf Basis der im Benchmark gemessenen durchschnittlichen Tokenanzahl

Neben Latenz und Kosten wurde auch die Qualität des Outputs betrachtet. Dafür wurde das strukturierte JSON mit dem ursprünglichen Schadenformular verglichen. Dabei wurden zwei Sichtweisen unterschieden: Einerseits die wörtliche Genauigkeit, also ob ein Wert exakt so übernommen wurde, wie er im Formular stand, und andererseits die semantische Normalisierung, also ob ein Wert inhaltlich richtig interpretiert und dem passenden Zielfeld zugeordnet wurde.

Bei der wörtlichen Genauigkeit zeigten beide Modelle insgesamt gute Ergebnisse. In allen Durchläufen wurden die zentralen Identifikationsfelder wie Zertifikatsnummer, Schadennummer, Vorname, Nachname, Geburtsdatum und JMBG korrekt extrahiert. Abweichungen zeigten sich

vor allem bei Kontakt- und Ortsfeldern. Im konkreten Formular war beispielsweise im Feld „Adresa / Address“ eine E-Mail-Adresse eingetragen. Eine rein wörtliche Auswertung bewertet dies streng genommen als Adresswert. Wird derselbe Wert jedoch vom Modell als E-Mail erkannt und in das Feld `email` übernommen, ist die wörtliche Genauigkeit zwar geringer, die semantische Interpretation jedoch fachlich sinnvoller.

Genau in diesem Punkt zeigte GPT-5 mini die robustere Nachverarbeitung. Semantisch relevante Zuordnungen wurden insgesamt konsistenter vorgenommen, während sich bei o4-mini etwas stärkere Schwankungen bei der Zuordnung von „Grad / Place“ beziehungsweise bei der Trennung von `city`, `place` und `email` zeigten. Die wörtliche Genauigkeit war bei beiden Modellen insgesamt hoch, im Bereich der semantischen Normalisierung schnitt GPT-5 mini jedoch besser ab.

Zusammenfassend zeigt der Benchmark, dass GPT-5 mini im hier untersuchten Anwendungsfall die insgesamt bessere Wahl für die OCR-Nachverarbeitung ist. Das Modell war im Test schneller, erzeugte weniger Tokens, verursachte dadurch geringere Kosten und lieferte zugleich die konsistentere semantische Normalisierung des OCR-Texts. Die anfängliche Verwendung von o4-mini war historisch durch den Projektzeitpunkt begründet. Für die finale Version der Anwendung wurde daher auf GPT-5 mini umgestellt.

3.2.7 Dateiverwaltung mit Azure Blob Storage

Für die Speicherung hochgeladener Dokumente wird Azure Blob Storage verwendet. Die Dateien selbst werden im Objektspeicher abgelegt, während in der relationalen Datenbank ausschließlich Metadaten gespeichert werden (z. B. Dateiname, Blob-Pfad, Upload-Zeitpunkt und fachliche Zuordnung über `CertificateNumber`).

Dieser Ansatz bietet mehrere Vorteile:

- **Skalierbarkeit**: große Binärdateien belasten nicht die SQL-Datenbank.
- **Klare Verantwortlichkeit**: strukturierte Daten und Dateien sind sauber getrennt.
- **Performance**: Dateien können effizient gestreamt und direkt im Frontend angezeigt werden.

Die API bildet den vollständigen Datei-Lifecycle ab:

- Upload von Claim-/Invoice-Dateien,
- Auflistung vorhandener Dateien,
- Inline-Anzeige (z. B. PDF/Bild),

- Löschung in Storage und Datenbank.

3.2.8 Swagger

Wie bereits in Abschnitt 2.2.9 beschrieben, wird Swagger bzw. OpenAPI in diesem Projekt verwendet, um die vorhandenen API-Endpunkte automatisch zu dokumentieren und während der Entwicklung einfach testen zu können. In der Implementierung wurde Swagger direkt im Startup der Anwendung konfiguriert.

Dafür wird zunächst mit `AddEndpointsApiExplorer()` die notwendige Grundlage für die Beschreibung der Endpunkte registriert. Anschließend wird über `AddSwaggerGen(...)` das eigentliche Swagger-Dokument erzeugt. In diesem Schritt werden auch Metadaten wie Titel, Version und Beschreibung der API gesetzt. Zusätzlich wird ein eigener `OperationFilter` eingebunden, damit Datei-Uploads in der Dokumentation korrekt dargestellt werden.

```
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen(c =>
{
    c.OperationFilter<FileUploadOperationFilter>();

    c.SwaggerDoc("v1", new OpenApiInfo
    {
        Title = "Claims Invoice API",
        Version = "v1",
        Description = "API for processing claims and invoices."
    });
});

app.UseSwagger(c =>
{
    c.PreSerializeFilters.Add((swaggerDoc, httpReq) =>
    {
        swaggerDoc.Servers = new List<OpenApiServer>
        {
            new OpenApiServer { Url = $"{httpReq.Scheme}://{httpReq.Host.Value}" }
        };
    });
});
app.UseSwaggerUI();
```

Listing 9: Swagger

Ein wichtiger Teil der Konfiguration ist außerdem der `PreSerializeFilter`. Dieser sorgt dafür, dass die Server-URL im generierten Swagger-Dokument dynamisch aus der aktuellen Anfrage übernommen wird. Das ist vor allem dann hilfreich, wenn die Anwendung in unterschiedlichen Umgebungen läuft, da dadurch nicht jedes Mal eine feste URL im Code hinterlegt werden muss.

Mit `app.UseSwagger()` wird das OpenAPI-Dokument zur Verfügung gestellt, während `app.UseSwaggerUI()` die grafische Oberfläche aktiviert. Dadurch konnten die Endpunkte während der Entwicklung direkt im Browser aufgerufen und getestet werden. Das hat die Arbeit mit der API deutlich vereinfacht, da Requests und Responses schnell überprüft werden konnten, noch bevor ein vollständiges Frontend angebunden war.

3.2.9 Konfiguration, Dependency Injection und Middleware

Die zentrale Initialisierung erfolgt in ASP. NET Core über den Application-Startup in `Program.cs`. Dort werden Konfigurationswerte gebunden, Services registriert und die HTTP-Pipeline aufgebaut.

Ein zentrales Konzept ist die integrierte *Dependency Injection*. Externe Clients (z. B. Azure SDKs), Datenzugriff (`DbContext`) sowie projektspezifische Services werden zentral registriert und danach automatisch in Controller bzw. Services injiziert.

Zusätzlich werden Funktionen über Middleware konfiguriert:

- **CORS** zur kontrollierten Kommunikation zwischen Frontend und Backend,
- **JSON-Optionen** für konsistente Serialisierung,
- **Swagger/OpenAPI** zur Dokumentation und interaktiven Endpunktprüfung,
- **Umgebungsspezifisches Verhalten** (z. B. Developer Exception Page in Development).

3.2.10 Validierung und Fehlerbehandlung

Die API implementiert eine Eingangsvalidierung, um fehlerhafte Requests frühzeitig abzufangen und eindeutige Rückmeldungen an das Frontend zu liefern. Dazu gehören insbesondere:

- Prüfung auf notwendige Eingabedaten,
- Prüfung von Datei-Eigenschaften (z. B. leere Dateien, erlaubte Dateitypen),
- Konsistenzprüfungen zwischen Routenparametern und Payload-Inhalten,

Für Fehlerfälle werden passende HTTP-Statuscodes verwendet:

- `400 Bad Request` bei ungültigen Eingaben,
- `404 Not Found` bei nicht vorhandenen Ressourcen,
- `500 Internal Server Error` bei unerwarteten Laufzeitfehlern.

Durch dieses Vorgehen ist das API-Verhalten transparent und für aufrufende Clients eindeutig interpretierbar.

3.2.11 Zusammenfassung der Backend-Implementierung

Zusammenfassend wurde das Backend als klar geschichtete ASP.NET-Core-Web-API umgesetzt. Die Controller kapseln die HTTP-Schnittstelle, während die Fachlogik in separaten Service-Klassen liegt. Entity Framework Core übernimmt den relationalen Datenzugriff. Die Trennung zwischen Entitäten und DTOs sowie das zentrale Mapping sorgen zusätzlich dafür, dass die API-Strukturen nach außen stabil bleiben.

Für die Dokumentverarbeitung wird eine Pipeline aus OCR und anschließender LLM-basierter Strukturierung eingesetzt. Die Texterkennung erfolgt mit Azure Document Intelligence, während die weitere Aufbereitung der Daten über Azure OpenAI umgesetzt wird. Hochgeladene Dateien werden in Azure Blob Storage gespeichert und über eigene Endpunkte bereitgestellt.

Insgesamt ergibt sich dadurch eine wartbare und erweiterbare Backend-Implementierung, die gut zu den Anforderungen dieser Anwendung passt. Gleichzeitig ist die Struktur so gewählt, dass spätere Änderungen und Erweiterungen des Systems vergleichsweise einfach umgesetzt werden können.

3.3 Datenbank

3.3.1 Azure SQL Database

Für die persistente Speicherung der fachlichen Daten wurde eine relationale Datenbank in Form einer Azure SQL Database eingesetzt. Azure SQL Database ist ein vollständig verwalteter Platform-as-a-Service-Dienst (PaaS), der sich besonders für Webanwendungen mit strukturierter Datenhaltung eignet. Der Betrieb, sicherheitsrelevante Basisfunktionen, Hochverfügbarkeit und Wartungsaufgaben (z. B. Patches) werden weitgehend durch die Azure-Plattform übernommen.

Im Rahmen der Implementierung wurde eine dedizierte Datenbankinstanz für die Anwendung bereitgestellt und über den `DbContext` des Backends angebunden. Die Tabellenstruktur basiert auf den Modelklassen des Backends und wird über Entity Framework Core verwaltet. Dadurch kann das Datenbankschema versioniert weiterentwickelt werden, ohne manuelle SQL-Skripte für jede Änderung pflegen zu müssen.

Rolle im Gesamtsystem

Die Azure SQL Database übernimmt die Speicherung von:

- Claim-Stammdaten,

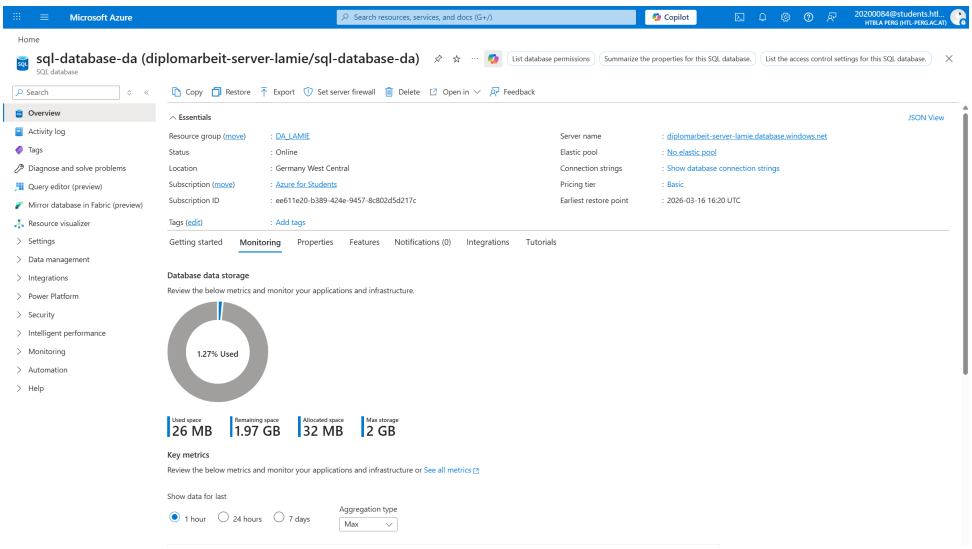


Abbildung 7: Azure SQL Database Dashboard

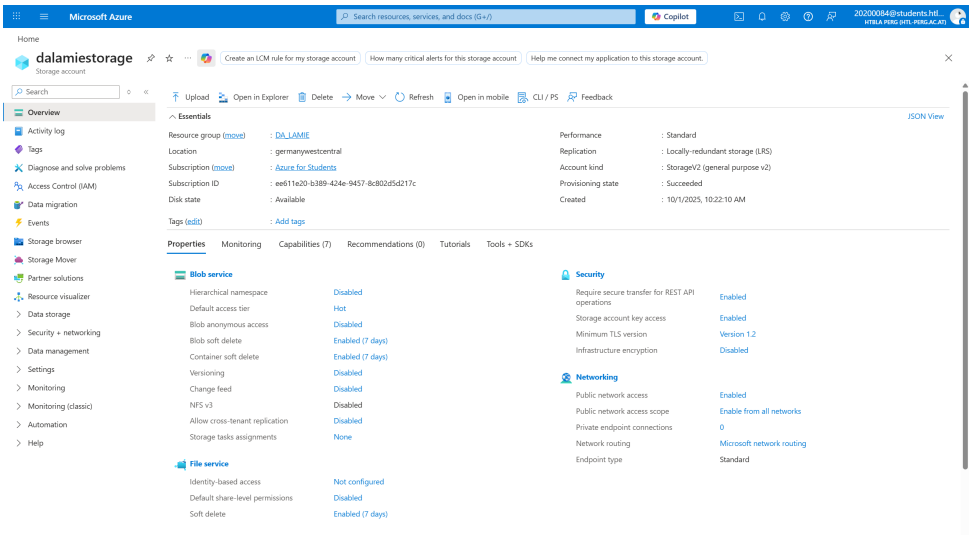


Abbildung 8: Azure Blob Storage Dashboard

- zugehörigen Detailentitäten (z. B. versicherte Person, Begünstigte, Vertrags- und Reisedaten),
- Rechnungsdaten und Rechnungspositionen,
- Datei-Metadaten (die eigentlichen Dateien liegen im Blob Storage).

Damit ist die Datenbank die zentrale Quelle für alle strukturierten Geschäftsdaten, während Binärdateien getrennt in Azure Blob Storage gehalten werden.

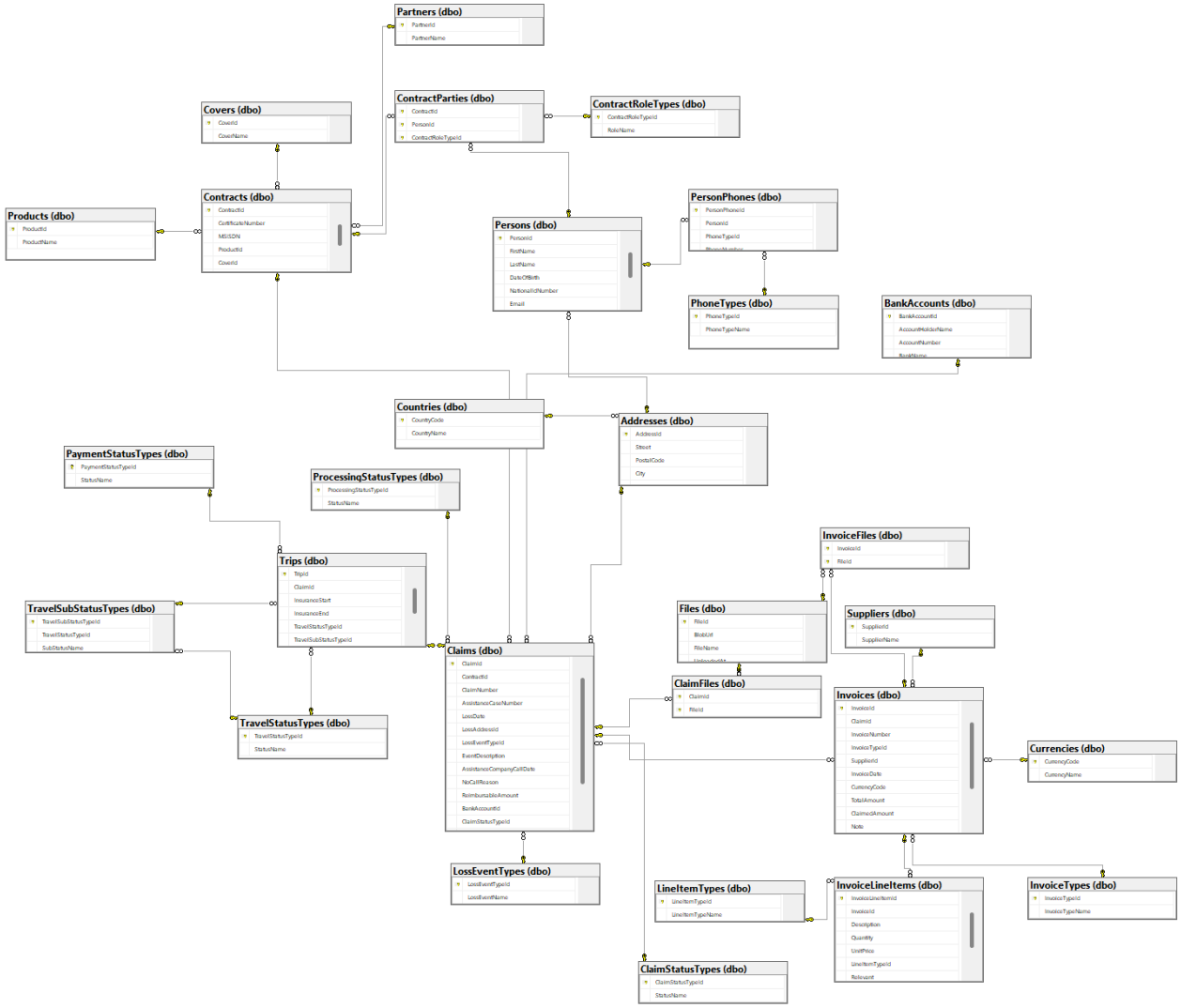


Abbildung 9: ERD in 3. Normalform

Schemaentwicklung mit Migrationen

Die Weiterentwicklung des Schemas erfolgt mit EF-Core-Migrationen. Änderungen an Entitäten werden in versionierten Migrationen abgebildet und reproduzierbar auf die Datenbank angewendet. Dieses Vorgehen verbessert Nachvollziehbarkeit, Teamarbeit und Deployment-Sicherheit.

3.3.2 ERD und Entitätsbeschreibungen

Countries **Zweck:** Speichert normalisierte Länder-Stammdaten.

Primärschlüssel: `CountryCode`

Beziehungsnotizen: Referenziert durch `Addresses.CountryCode`.

Tabelle 4: Countries

Spalte	Datentyp	Pflicht	Beschreibung
`CountryCode`	`CHAR(2)`	Ja	Primärschlüssel; kurzer Länderidentifier.
`CountryName`	`NVARCHAR(100)`	Ja	Lesbarer Ländername.

Currencies **Zweck:** Speichert gültige Währungscodes und Währungsnamen für Rechnungen.

Primärschlüssel: `CurrencyCode`

Beziehungsnotizen: Referenziert durch `Invoices.CurrencyCode`.

Tabelle 5: Currencies

Spalte	Datentyp	Pflicht	Beschreibung
`CurrencyCode`	`CHAR(3)`	Ja	Primärschlüssel; ISO-ähnlicher Währungscode.
`CurrencyName`	`NVARCHAR(100)`	Ja	Lesbarer Währungsname.

Products **Zweck:** Speichert Versicherungsprodukte unabhängig von Verträgen.

Primärschlüssel: `ProductId`

Beziehungsnotizen: Referenziert durch `Contracts.ProductId`.

Tabelle 6: Products

Spalte	Datentyp	Pflicht	Beschreibung
`ProductId`	`INT IDENTITY`	Ja	Primärschlüssel.
`ProductName`	`NVARCHAR(150)`	Ja	Produktbezeichnung.

Covers **Zweck:** Speichert Deckungsarten unabhängig von Verträgen.

Primärschlüssel: `CoverId`

Beziehungsnotizen: Referenziert durch `Contracts.CoverId`.

Tabelle 7: Covers

Spalte	Datentyp	Pflicht	Beschreibung
`CoverId`	`INT IDENTITY`	Ja	Primärschlüssel.
`CoverName`	`NVARCHAR(150)`	Ja	Bezeichnung der Deckungsart.

Partners **Zweck:** Speichert Partnerorganisationen getrennt von Vertragsdaten.

Primärschlüssel: `PartnerId`

Beziehungsnotizen: Referenziert durch `Contracts.PartnerId`.

Tabelle 8: Partners

Spalte	Datentyp	Pflicht	Beschreibung
PartnerId	INT IDENTITY	Ja	Primärschlüssel.
PartnerName	NVARCHAR(150)	Ja	Partnername.

ContractRoleTypes **Zweck:** Speichert Rollenarten im Vertragskontext.

Primärschlüssel: `ContractRoleTypeId`

Beziehungsnotizen: Referenziert durch `ContractParties.ContractRoleTypeId`.

Tabelle 9: ContractRoleTypes

Spalte	Datentyp	Pflicht	Beschreibung
ContractRoleTypeId	INT IDENTITY	Ja	Primärschlüssel.
RoleName	NVARCHAR(50)	Ja	Rollenbezeichnung (z. B. SUBSCRIBER, INSURED, BENEFICIARY).

PhoneTypes **Zweck:** Speichert Kategorien von Telefonnummern.

Primärschlüssel: `PhoneTypeId`

Beziehungsnotizen: Referenziert durch `PersonPhones.PhoneTypeId`.

Tabelle 10: PhoneTypes

Spalte	Datentyp	Pflicht	Beschreibung
PhoneTypeId	INT IDENTITY	Ja	Primärschlüssel.
PhoneTypeName	NVARCHAR(50)	Ja	Typbezeichnung der Telefonnummer.

ClaimStatusTypes **Zweck:** Speichert zulässige fachliche Claim-Statuswerte.

Primärschlüssel: `ClaimStatusTypeId`

Beziehungsnotizen: Referenziert durch `Claims.ClaimStatusTypeId`.

Tabelle 11: ClaimStatusTypes

Spalte	Datentyp	Pflicht	Beschreibung
ClaimStatusTypeId	INT IDENTITY	Ja	Primärschlüssel.
StatusName	NVARCHAR(50)	Ja	Statusbezeichnung.

ProcessingStatusTypes **Zweck:** Speichert interne Verarbeitungszustände.

Primärschlüssel: `ProcessingStatusTypeId`

Beziehungsnotizen: Referenziert durch `Claims.ProcessingStatusTypeId`.

Tabelle 12: ProcessingStatusTypes

Spalte	Datentyp	Pflicht	Beschreibung
`ProcessingStatusTypeId`	`INT IDENTITY`	Ja	Primärschlüssel.
`StatusName`	`NVARCHAR(50)`	Ja	Statusbezeichnung.

LossEventTypes **Zweck:** Speichert normalisierte Schadenereignis-Kategorien.

Primärschlüssel: `LossEventTypeId`

Beziehungsnotizen: Referenziert durch `Claims.LossEventTypeId`.

Tabelle 13: LossEventTypes

Spalte	Datentyp	Pflicht	Beschreibung
`LossEventTypeId`	`INT IDENTITY`	Ja	Primärschlüssel.
`LossEventName`	`NVARCHAR(100)`	Ja	Ereignisbezeichnung.

TravelStatusTypes **Zweck:** Speichert übergeordnete Reisestatuswerte.

Primärschlüssel: `TravelStatusTypeId`

Beziehungsnotizen: Referenziert durch `Trips.TravelStatusTypeId`; Parent von `TravelSubStatusTypes`

Tabelle 14: TravelStatusTypes

Spalte	Datentyp	Pflicht	Beschreibung
`TravelStatusTypeId`	`INT IDENTITY`	Ja	Primärschlüssel.
`StatusName`	`NVARCHAR(50)`	Ja	Statusbezeichnung.

TravelSubStatusTypes **Zweck:** Speichert Unterstatuswerte innerhalb eines bestimmten Reisestatus.

Primärschlüssel: `TravelSubStatusTypeId`

Beziehungsnotizen: Viele Unterstatuswerte gehören zu einem Reisestatus; Referenz durch `Trips.TravelSubStatusTypeId`.

Tabelle 15: TravelSubStatusTypes

Spalte	Datentyp	Pflicht	Beschreibung
`TravelSubStatusTypeId`	`INT IDENTITY`	Ja	Primärschlüssel.
`TravelStatusTypeId`	`INT`	Ja	Fremdschlüssel zu `TravelStatusTypes`.
`SubStatusName`	`NVARCHAR(50)`	Ja	Unterstatusname (pro Reisestatus eindeutig).

PaymentStatusTypes **Zweck:** Speichert standardisierte Zahlungszustände.

Primärschlüssel: `PaymentStatusTypeId`

Beziehungsnotizen: Referenziert durch `Trips.PaymentStatusTypeId`.

Tabelle 16: PaymentStatusTypes

Spalte	Datentyp	Pflicht	Beschreibung
`PaymentStatusTypeId`	`INT IDENTITY`	Ja	Primärschlüssel.
`StatusName`	`NVARCHAR(50)`	Ja	Statusbezeichnung.

InvoiceTypes **Zweck:** Speichert Rechnungskategorien.

Primärschlüssel: `InvoiceTypeId`

Beziehungsnotizen: Referenziert durch `Invoices.InvoiceTypeId`.

Tabelle 17: InvoiceTypes

Spalte	Datentyp	Pflicht	Beschreibung
`InvoiceTypeId`	`INT IDENTITY`	Ja	Primärschlüssel.
`InvoiceTypeName`	`NVARCHAR(50)`	Ja	Rechnungsart-Bezeichnung.

LineItemTypes **Zweck:** Speichert standardisierte Klassifikationen für Rechnungspositionen.

Primärschlüssel: `LineItemTypeId`

Beziehungsnotizen: Referenziert durch `InvoiceLineItems.LineItemTypeId`.

Tabelle 18: LineItemTypes

Spalte	Datentyp	Pflicht	Beschreibung
`LineItemTypeId`	`INT IDENTITY`	Ja	Primärschlüssel.
`LineItemTypeName`	`NVARCHAR(50)`	Ja	Positions-Typbezeichnung.

7.2 Stammdatentabellen

Addresses **Zweck:** Speichert Adressdatensätze, die von Personen und Schadenorten referenziert werden können.

Primärschlüssel: `AddressId`

Beziehungsnotizen: Referenziert durch `Persons.AddressId` und `Claims.LossAddressId`.

Tabelle 19: Addresses

Spalte	Datentyp	Pflicht	Beschreibung
`AddressId`	`INT IDENTITY`	Ja	Primärschlüssel.
`Street`	`NVARCHAR(200)`	Nein	Straße und Hausnummer.
`PostalCode`	`NVARCHAR(20)`	Nein	Postleitzahl als Text.
`City`	`NVARCHAR(100)`	Nein	Stadt/Gemeinde.
`CountryCode`	`CHAR(2)`	Nein	Fremdschlüssel zu `Countries`.

Persons **Zweck:** Zentrale Person-Entität als Ersatz für duplizierte Personentabellen.

Primärschlüssel: `PersonId`

Beziehungsnotizen: Referenziert durch `ContractParties.PersonId` und `PersonPhones.PersonId`.

Tabelle 20: Persons

Spalte	Datentyp	Pflicht	Beschreibung
`PersonId`	`INT IDENTITY`	Ja	Primärschlüssel.
`FirstName`	`NVARCHAR(100)`	Ja	Vorname.
`LastName`	`NVARCHAR(100)`	Ja	Nachname.
`DateOfBirth`	`DATE`	Nein	Geburtsdatum.
`NationalIdNumber`	`NVARCHAR(100)`	Nein	Optionale staatliche ID (falls gesetzt eindeutig).
`Email`	`NVARCHAR(254)`	Nein	Primäre E-Mail-Adresse.
`AddressId`	`INT`	Nein	Fremdschlüssel zu `Addresses`.

PersonPhones **Zweck:** Speichert eine oder mehrere Telefonnummern pro Person inklusive Typ.

Primärschlüssel: `PersonPhoneId`

Beziehungsnotizen: Viele Telefonnummern können einer Person zugeordnet sein.

Tabelle 21: PersonPhones

Spalte	Datentyp	Pflicht	Beschreibung
PersonPhoneId	INT IDENTITY	Ja	Primärschlüssel.
PersonId	INT	Ja	Fremdschlüssel zu Persons.
PhoneTypeId	INT	Ja	Fremdschlüssel zu PhoneTypes.
PhoneNumber	NVARCHAR(50)	Ja	Telefonnummer.

BankAccounts **Zweck:** Speichert Erstattungs-Kontodaten unabhängig von Claims.

Primärschlüssel: BankAccountId

Beziehungsnotizen: Referenziert durch Claims.BankAccountId.

Tabelle 22: BankAccounts

Spalte	Datentyp	Pflicht	Beschreibung
BankAccountId	INT IDENTITY	Ja	Primärschlüssel.
AccountHolderName	NVARCHAR(200)	Ja	Name des Kontoinhabers.
AccountNumber	NVARCHAR(100)	Ja	Eindeutige Kontonummer / IBAN-ähnlicher Wert.
BankName	NVARCHAR(150)	Nein	Name der Bank.

Suppliers **Zweck:** Speichert Rechnungssteller zentral.

Primärschlüssel: SupplierId

Beziehungsnotizen: Referenziert durch Invoices.SupplierId.

Tabelle 23: Suppliers

Spalte	Datentyp	Pflicht	Beschreibung
SupplierId	INT IDENTITY	Ja	Primärschlüssel.
SupplierName	NVARCHAR(200)	Ja	Lieferantenname.

7.3 Vertrags- und Rollentabellen

Contracts **Zweck:** Repräsentiert den versicherten Vertrag/Polizze.

Primärschlüssel: ContractId

Beziehungsnotizen: Parent von Claims; Parent von ContractParties.

Tabelle 24: Contracts

Spalte	Datentyp	Pflicht	Beschreibung
ContractId	INT IDENTITY	Ja	Primärschlüssel.
CertificateNumber	NVARCHAR(450)	Ja	Eindeutiger fachlicher Vertragsidentifier.
MSISDN	BIGINT	Nein	Mobilfunk-Identifier des Abonnenten (falls vorhanden).
ProductId	INT	Nein	Fremdschlüssel zu Products.
CoverId	INT	Nein	Fremdschlüssel zu Covers.
PartnerId	INT	Nein	Fremdschlüssel zu Partners.

ContractParties **Zweck:** Verknüpfungstabelle zwischen Personen und Verträgen mit Rolleninformation.

Primärschlüssel: (ContractId, PersonId, ContractRoleTypeId)

Beziehungsnotizen: M:N-Beziehung zwischen Contracts und Persons mit Rollensemantik.

Tabelle 25: ContractParties

Spalte	Datentyp	Pflicht	Beschreibung
ContractId	INT	Ja	Fremdschlüssel zu Contracts.
PersonId	INT	Ja	Fremdschlüssel zu Persons.
ContractRoleTypeId	INT	Ja	Fremdschlüssel zu ContractRoleTypes.

7.4 Claim- und Reise-Tabellen

Claims **Zweck:** Speichert einen Schadenfall unter einem Vertrag.

Primärschlüssel: ClaimId

Beziehungsnotizen: Viele Claims gehören zu einem Contract; Parent von Trips, Invoices, ClaimFiles.

Tabelle 26: Claims

Spalte	Datentyp	Pflicht	Beschreibung
ClaimId	INT IDENTITY	Ja	Primärschlüssel.
ContractId	INT	Ja	Fremdschlüssel zu Contracts.
ClaimNumber	NVARCHAR(100)	Nein	Optionale eindeutige Claim-Nummer.
AssistanceCaseNumber	NVARCHAR(100)	Nein	Optionale eindeutige Assistance-Fallnummer.
LossDate	DATE	Nein	Datum des Schadenereignisses.
LossAddressId	INT	Nein	Fremdschlüssel zu Addresses.
LossEventTypeId	INT	Nein	Fremdschlüssel zu LossEventTypes.
EventDescription	NVARCHAR(MAX)	Nein	Textuelle Beschreibung des Ereignisses.
AssistanceCompanyCallDate	DATE	Nein	Datum der Kontaktaufnahme mit der Assistance.
NoCallReason	NVARCHAR(300)	Nein	Grund für unterlassene Kontaktaufnahme.
ReimbursableAmount	DECIMAL(18,2)	Nein	Erstattungsbetrag.
BankAccountId	INT	Nein	Fremdschlüssel zu BankAccounts.
ClaimStatusTypeId	INT	Nein	Fremdschlüssel zu ClaimStatusTypes.
ProcessingStatusTypeId	INT	Nein	Fremdschlüssel zu ProcessingStatusTypes.
CreatedAt	DATETIME2(7)	Ja	Erstellzeitpunkt (Default SYSUTCDATETIME()).
LastEditedAt	DATETIME2(7)	Nein	Letzter Änderungszeitpunkt.

Trips **Zweck:** Speichert reisespezifische Daten zu einem Claim.

Primärschlüssel: TripId

Beziehungsnotizen: Höchstens ein Trip-Datensatz pro Claim (ClaimId eindeutig).

Tabelle 27: Trips

Spalte	Datentyp	Pflicht	Beschreibung
TripId	INT IDENTITY	Ja	Primärschlüssel.
ClaimId	INT	Ja	Eindeutiger Fremdschlüssel zu Claims.
InsuranceStart	DATE	Nein	Beginn des Versicherungszeitraums.
InsuranceEnd	DATE	Nein	Ende des Versicherungszeitraums.
TravelStatusTypeId	INT	Nein	Fremdschlüssel zu TravelStatusTypes.
TravelSubStatusTypeId	INT	Nein	Fremdschlüssel zu TravelSubStatusTypes.
PaymentStatusTypeId	INT	Nein	Fremdschlüssel zu PaymentStatusTypes.

7.5 Rechnungs- und Dateitabellen

Invoices **Zweck:** Speichert Rechnungen, die einem Claim zugeordnet sind.

Primärschlüssel: InvoiceId

Beziehungsnotizen: Viele Invoices gehören zu einem Claim; Parent von InvoiceLineItems und InvoiceFiles.

Tabelle 28: Invoices

Spalte	Datentyp	Pflicht	Beschreibung
InvoiceId	UNIQUEIDENTIFIER	Ja	Primärschlüssel; Default NEWSEQUENTIALID().
ClaimId	INT	Ja	Fremdschlüssel zu Claims.
InvoiceNumber	NVARCHAR(100)	Nein	Referenznummer der Rechnung.
InvoiceTypeId	INT	Nein	Fremdschlüssel zu InvoiceTypes.
SupplierId	INT	Nein	Fremdschlüssel zu Suppliers.
InvoiceDate	DATE	Nein	Rechnungsdatum.
CurrencyCode	CHAR(3)	Nein	Fremdschlüssel zu Currencies.
TotalAmount	DECIMAL(18,2)	Nein	Gesamtbetrag der Rechnung.
ClaimedAmount	DECIMAL(18,2)	Nein	Für den Claim relevanter Betrag.
Note	NVARCHAR(MAX)	Nein	Zusätzliche Notiz.
CreatedAt	DATETIME2(7)	Ja	Erstellzeitpunkt (Default SYSUTCDATETIME()).
LastEditedAt	DATETIME2(7)	Nein	Letzter Änderungszeitpunkt.

InvoiceLineItems **Zweck:** Speichert die Positionen einer Rechnung; Amount ist eine persistierte berechnete Spalte.

Primärschlüssel: `InvoiceLineItemId`

Beziehungsnotizen: Viele Positionen gehören zu einer Rechnung.

Tabelle 29: InvoiceLineItems

Spalte	Datentyp	Pflicht	Beschreibung
`InvoiceLineItemId`	`INT IDENTITY`	Ja	Primärschlüssel.
`InvoiceId`	`UNIQUEIDENTIFIER`	Ja	Fremdschlüssel zu `Invoices`.
`Description`	`NVARCHAR(500)`	Ja	Positionsbeschreibung.
`Quantity`	`DECIMAL(18,4)`	Ja	Menge.
`UnitPrice`	`DECIMAL(18,2)`	Ja	Preis pro Einheit.
`LineItemTypeId`	`INT`	Nein	Fremdschlüssel zu `LineItemTypes`.
`Relevant`	`BIT`	Ja	Markierung, ob Position claim-relevant ist.
`Amount`	`Computed DECIMAL(18,2)`	Ja	Persistierter Wert: `Quantity * UnitPrice`.

Files **Zweck:** Speichert Dateimetadaten zentral.

Primärschlüssel: `FileId`

Beziehungsnotizen: Referenziert durch `ClaimFiles` und `InvoiceFiles`.

Tabelle 30: Files

Spalte	Datentyp	Pflicht	Beschreibung
`FileId`	`INT IDENTITY`	Ja	Primärschlüssel.
`BlobUrl`	`NVARCHAR(2048)`	Ja	Eindeutige Storage-URL.
`FileName`	`NVARCHAR(260)`	Ja	Original-/Anzeigename der Datei.
`UploadedAt`	`DATETIME2(7)`	Ja	Upload-Zeitpunkt (Default `SYSUTCDATETIME()`).

ClaimFiles **Zweck:** Verknüpfungstabelle zwischen Dateien und Claims.

Primärschlüssel: `(ClaimId, FileId)`

Beziehungsnotizen: Zuordnung zwischen `Claims` und `Files`.

Tabelle 31: ClaimFiles

Spalte	Datentyp	Pflicht	Beschreibung
`ClaimId`	`INT`	Ja	Fremdschlüssel zu `Claims`.
`FileId`	`INT`	Ja	Fremdschlüssel zu `Files`.

InvoiceFiles **Zweck:** Verknüpfungstabelle zwischen Dateien und Rechnungen.

Primärschlüssel: `(InvoiceId, FileId)`

Beziehungsnotizen: Zuordnung zwischen `Invoices` und `Files`.

Tabelle 32: InvoiceFiles

Spalte	Datentyp	Pflicht	Beschreibung
`InvoiceId`	`UNIQUEIDENTIFIER`	Ja	Fremdschlüssel zu `Invoices`.
`FileId`	`INT`	Ja	Fremdschlüssel zu `Files`.

4 Ergebnis

In diesem Kapitel wird das Ergebnis der Arbeit vorgestellt. Anhand der Benutzeroberfläche wird der Workflow von ClarioAI erläutert, sodass die Anwendung ohne Installation nachvollziehbar wird.

4.1 Frontend

4.1.1 Overview

Hier ist die Overview-Seite. Diese Seite ist die Startseite des Workflows. Man sieht die Navbar mit Logo links und zwei Buttons zur Navigation an der rechten Seite (siehe I in Abb. 10). Nähere Informationen zu dieser Seite siehe 3.1.4 Seiten.

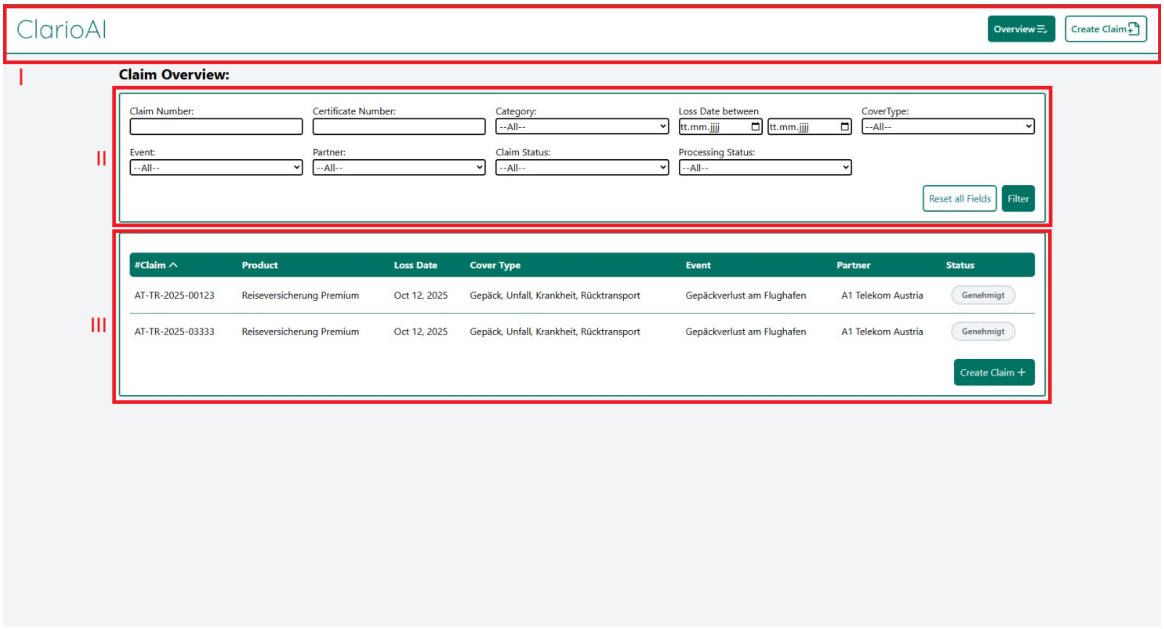


Abbildung 10: Overview der ClarioAI-Applikation

Zum Inhalt der Overview-Seite gehören:

- **FilterBox**

  Die Komponente hier dient zur Filterung der Claims, welche darunter (siehe III Abb. 10) angezeigt werden. Die FilterBox-Komponente (siehe II Abb. 10) besteht, wie man sehen kann, aus mehreren Feldern verschiedener Art. So gibt es normale Text-Inputs, Dropdowns

und Date-Inputs. Im unteren Bereich der FilterBox sieht man auch zwei Buttons. Beim linken Button werden alle Inputfelder jeder Art dieser Komponente zurückgesetzt. Daneben auf den Button mit der Aufschrift „Filter“ wird das Filtern mit den eingeben Parametern gestartet dafür wird ein Service (siehe 3.1.6 Services & API-Calls) benutzt.

- **Claim Overview**
  Der letzte Teil der Seite bietet eine Übersicht über alle Claims oder jenen die den Filterkriterien in II Abb. 10 nicht entsprechen. Die Tabelle ist auf jedes Attribut, welches in der Tabelle steht, sortierbar, wobei durch erneutes Klicken nach dem Attribute aufsteigend beziehungsweise absteigend sortiert wird. Durch Klicken auf eine Zeile der Tabelle wird auf die Detailansicht dieses Claims gezeigt. Durch den Button mit der Aufschrift „Create Claim“ wird man weiter auf die Seite für die Erstellung eines neuen Claims weitergeleitet (siehe 4.1.4 Create Claim).

4.1.2 Claim Detail

Auf diese Seite wird man geroutet, wenn man auf der Overview-Seite auf eine Zeile klickt. Hier werden Details eines Claims angezeigt. Die Seite besteht aus:

- **File Upload**
  Diese Komponente ist dieselbe wie auf der *CreateClaim*-Seite, jedoch im „View-Modus“. Das bedeutet, man kann weder neue Files hinzufügen noch bestehende Files löschen. Sie dient ausschließlich der Anzeige.

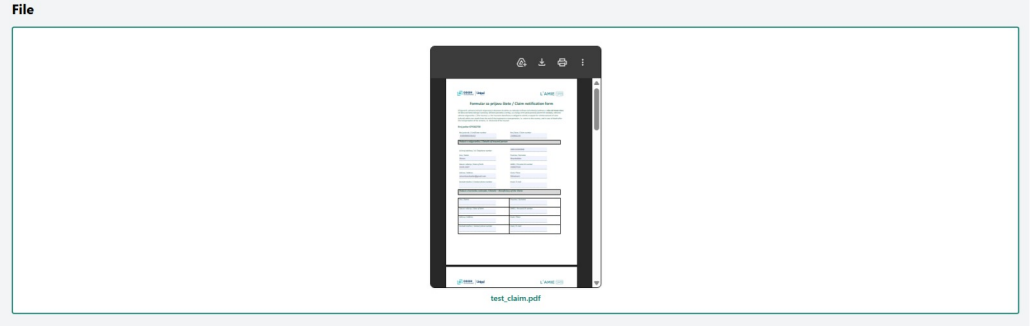


Abbildung 11: FileUpload vom Claim, im View-Modus

- **Claim Details, Contract, Trip & Impacted Person**

  Diese Teile der Seite stellen alle restlichen Details des Claims dar. Auch hier dienen die Felder ausschließlich der Anzeige der Daten. Standardmäßig sind „Impacted Person“ und „Beneficiary Person“ in der minimalen Ansicht dargestellt. Für zusätzliche Details müsste man auf „Show more“ klicken. Weiters wird für die „Beneficiary Person“ keine eigene Box angezeigt, wenn diese gleichzeitig die „Impacted Person“ ist.

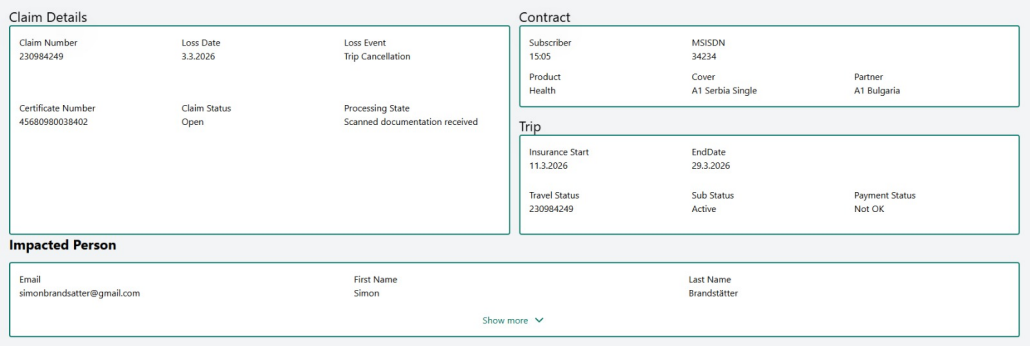


Abbildung 12: Claim Details, Contract, Trip und Impacted Person im View-Modus

- **Invoice Overview**

  Zu einem Claim gehören auch die dazugehörigen Invoices. Dafür gibt es diese Übersichtstabelle. Der Sinn dieses Teils der Seite ist, dass man die wichtigsten Daten für jede Invoice schnell erkennen kann. Eines der wichtigsten Felder in dieser Tabelle sind wahrscheinlich „Claimed Amount“, „Accepted Amount“ und dessen Icon daneben. „Claimed Amount“ ist die Summe aller *Invoice Lines*, die auf der „Invoice“ sind. Wohingegen „Accepted Amount“ nur jene Invoice Lines einberechnet, welche von der Versicherung übernommen werden. Das kreisähnliche Icon soll darstellen, ob die ganze Invoice übernommen wird oder nur ein Teil davon. In Abb. 13 sieht man, dass hier nicht hundert Prozent übernommen werden.

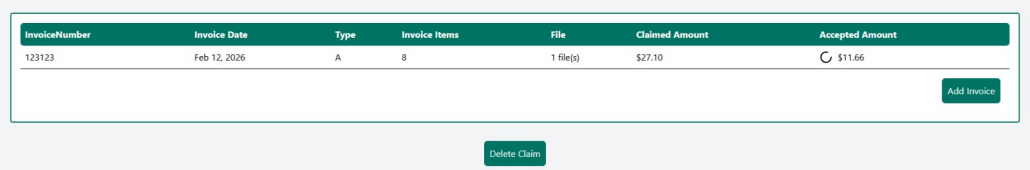


Abbildung 13: Overview der Invoices

4.1.3 Invoice Detail

- **File Upload**

  Wie bei 4.1.2 Claim Detail ist hier die FileUpload-Komponente wieder nur im View-Modus vorhanden. Testweise wurde hier eine nichtssagende Test-Invoice hochgeladen.

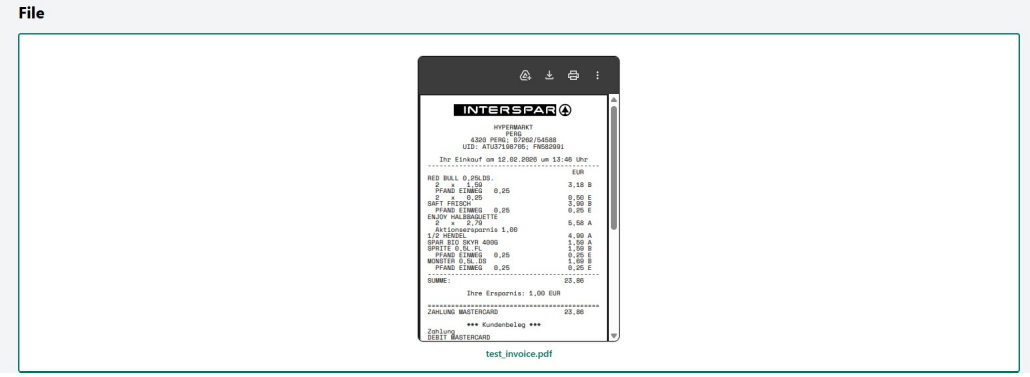


Abbildung 14: FileUpload vom Invoice, im View-Modus

- **Invoice Lines**

  Hier sind die Invoice Lines ersichtlich. Man kann nach dem Erstellen alle Attribute anpassen oder neue Invoice Lines hinzufügen.

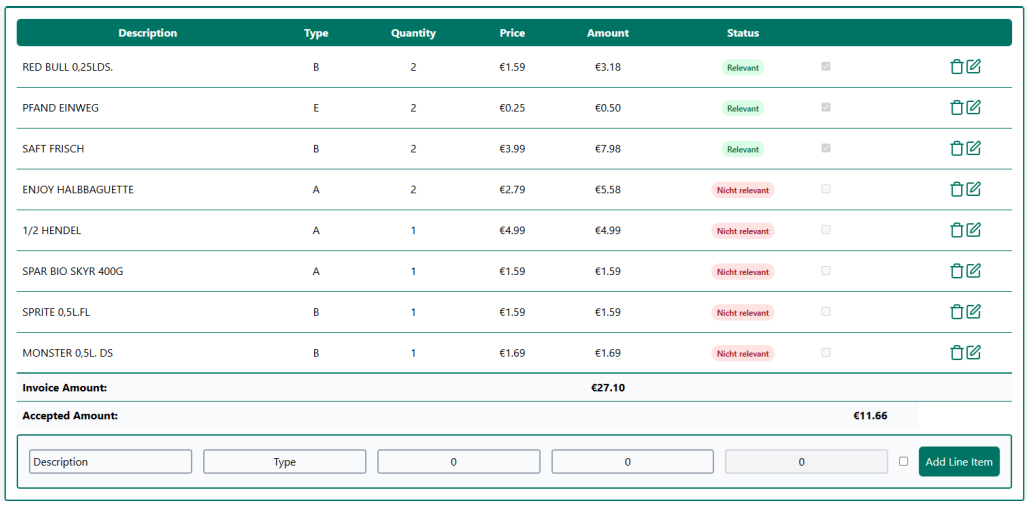


Abbildung 15: Invoice Lines

- **Invoice Details**

  Hier stehen die restlichen Informationen zu dieser Invoice. Hier sind auch nochmals die zwei verschiedenen Beträge abgebildet. Außerdem steht hier auch nochmals die Währung, in der die Beträge gegeben sind. Das ist wichtig, weil es sich hier um internationale Invoices in Fremdwährungen handeln kann.

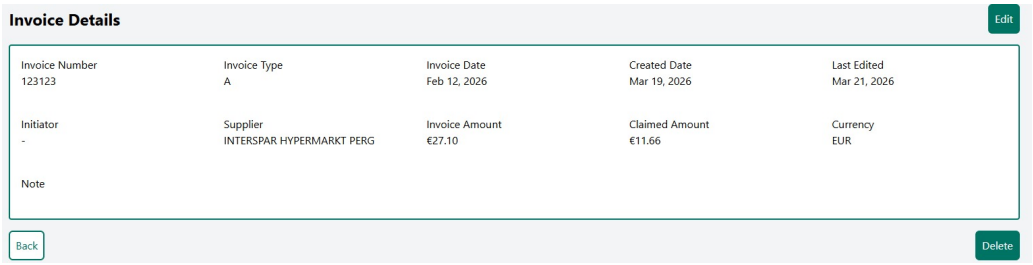


Abbildung 16: Invoice Details, im View-Modus

4.1.4 Create Claim

Diese Seite wird für das Erstellen eines Claims benutzt. Für genauere Informationen siehe Create Claim in 3.1.4. Die Create-Claim-Seite besteht aus:

- **File Upload**
  Hier können die User:innen durch Drag-and-Drop oder durch das Dateiexplorerfenster, welches sich öffnet, wenn man auf das Icon klickt, Dateien in mehreren Fomaten hochladen. In weiterer Folge können die Fotos der Claims analysiert werden. Und die Inputfelder im weiteren Verlauf werden mit den erkannten Daten ausgefüllt. Dadurch spart sich ein:e Mitarbeiter:in das genau Entziffern von Handschriften.

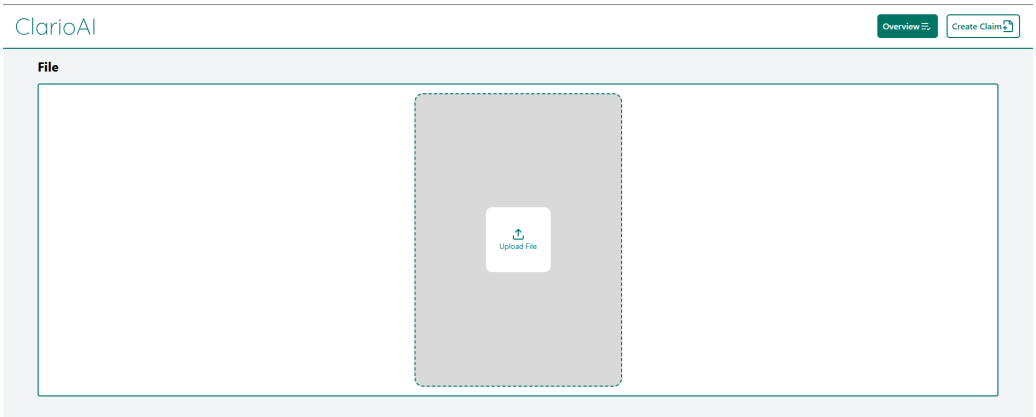


Abbildung 17: FileUpload auf der CreateClaim-Seite

Nach dem mindestens eine Datei hochgeladen geworden ist, ist es den User:innen möglich, diese Dateien zu analysieren. Wie im Hinweis steht werden die Felder darunter ausgefüllt, soweit im File die passenden Informationen erkannt wurden. Siehe Abb. 18 für Details. Durch Klicken auf das X rechts oben in der Ansicht des Files kann der User die Datei wieder aus dem CreateClaim-Formular entfernen.

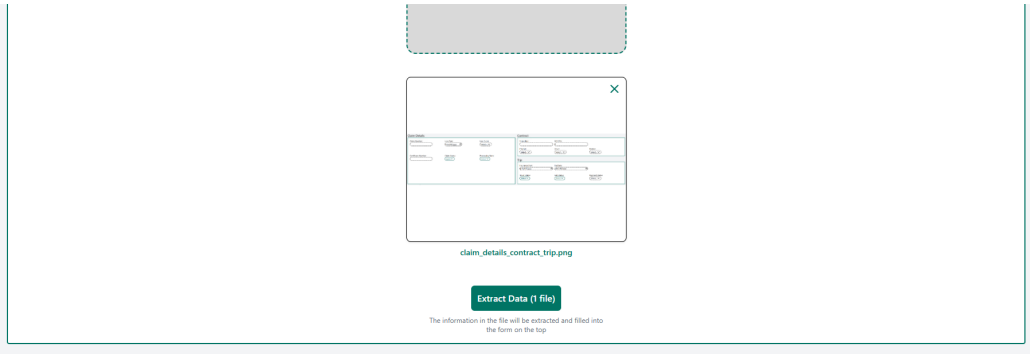


Abbildung 18: FileUpload auf der CreateClaim-Seite mit hochgeladener Datei

- **Claim Details, Contract & Trip**

  Die Felder, welche im Grid angeordnet sind (siehe Abb. 19), gehören zu den benötigten Daten eines Files. Die meisten Felder werden hier durch die darüberstehende FileUpload-Komponente ausgefüllt, sofern die analysierten Dateien für das Backend lesbare Werte enthalten.

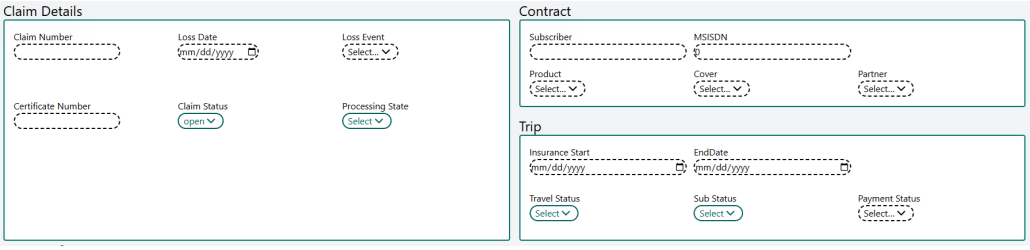


Abbildung 19: Claim Details, Contact und Trip

Wie man in Abb. 19 erkennen kann, handelt es sich hier nicht um ein Formular, welches nur aus Text-Inputs besteht. Es kommen auch Date-Inputs und Dropdowns vor. Für Letzteres gibt es aus optischen Gründen (siehe 3.1.1 Design und Prototyping) zwei Arten von Dropdowns. Diese unterscheiden sich jedoch nur auf visueller Ebene.

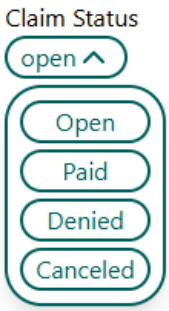


Abbildung 20: Dropdown der ersten Art

Abbildung 21: Dropdown der zweiten Art

- **Impacted Person**

  Als vorletzter Teil dieser Seite findet sich ein Input für die Impacted Person. Diese besteht mehrheitlich aus Text-Input-Feldern. Diese Komponente ist standardmäßig ausgeklappt. Mit dem Button (siehe II in Abb. 22) können alle Attribute, welche unter der strichlierten Linie (siehe I in Abb. 22) sind, eingeklappt werden. Hierbei wurde man stark von dem bestehenden Luis-System (mehr zum Luis-System ist in 1.4 beschrieben) inspiriert, das setzte das nämlich sehr ähnlich um. Die eingeklappte Ansicht ist in Abb. 23 zu sehen.

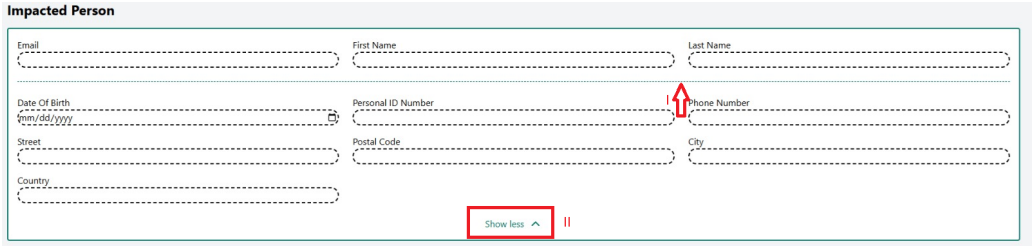


Abbildung 22: Impacted Person, Vollansicht

Abbildung 23: Impacted Person, Teilansicht

- **Beneficiary Person**

  Zusätzlich zur Impacted Person gibt es noch eine Beneficiary Person, welche die Person ist, welche die Leistung der Versicherung empfangen soll. Da es sich aber hierbei gelegentlich um ein und dieselbe Person handelt, existiert eine Checkbox (siehe I in Abb. 24), welche fragt, ob das den der Fall sei. Ist das der Fall, wird die Box der Beneficiary Person

nicht angezeigt. Ansonsten verhält sich die Beneficiary Person-Komponente genauso wie die Impacted Person Komponente. Es ist also auch möglich, mittels des Button (siehe II in Abb. 24) die Ansicht zwischen minimierter Ansicht und maximierter Ansicht zu wechseln.imüpac

Abbildung 24: Beneficiary Person, volle Ansicht

4.1.5 Create Invoice

Diese Seite basiert auf denselben Komponenten wie 4.1.3 Invoice Detail, jedoch mit dem Unterschied, dass hier alle Komponenten nicht im View-Modus vorhanden sind. Um zur Invoice-Erstellung zu kommen muss man in der Detail View eines Claims am unteren Ende dieser Seite

Add Invoice (siehe I in Abb. 25) auf

Add Invoice) klicken.

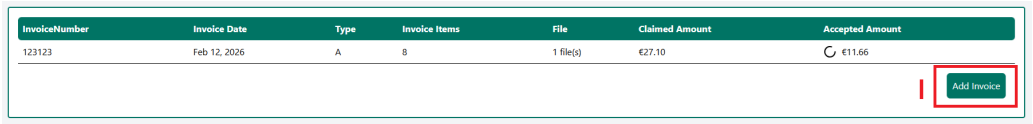


Abbildung 25: Navigation zum Erstellen einer Invoice

- **Insurance Coverage**

  Hier wählt man welche Art von Versicherung der Kunde hat. Das ist wichtig, weil in weiteren Schritten das KI-Modell diese Information als Kontext benötigt für eine *Prediction*.

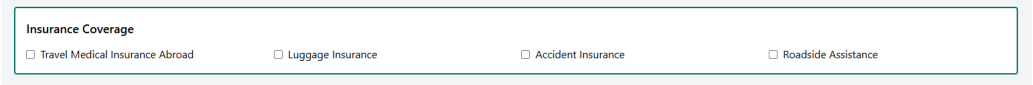


Abbildung 26: Versicherungsleistungen Auswahl

- **Analyze Invoice**

  An dieser Stelle der Seite werden die Dokumente über die Invoice hochgeladen und im

weiteren Schritt analysiert. Nach dem Drücken des „Extract Data“-Button bekommt das Frontend vom Backend die erkannten Invoice Lines und die Predicition vom KI-Modell, ob diese Invoice Line von der Versicherung des Kunden gedeckt wird.

Abbildung 27: Extract Data, der Invoice Dokumente

- **Invoice Lines**

  In diesem Teil der Seite werden alle Invoice Lines aufgelistet, welche in der Ausgangsdatei erkannt worden sind. Außerdem ist hier die Bewertung von der OpenAI-API sichtbar gemacht. So sieht man in der Spalte „Status“, ob die Invoice Line von der Versicherung übernommen wird. Dadurch ergeben sich zwei Summen. Die Summe aller Invoice Lines und die Summe aller Invoice Lines, welche relevant sind beziehungsweise übernommen werden. Da das KI-Modell Fehler machen kann, kann man jede Prediction auch korrigieren und nicht erkannte Invoice Lines manuell hinzufügen.

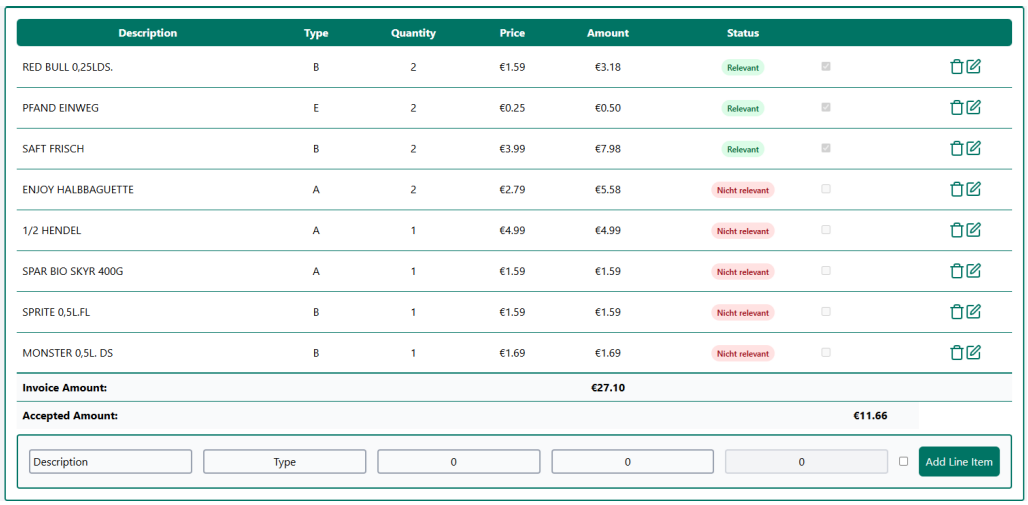


Abbildung 28: Invoice Lines

- **Invoice Details**

  In dieser Komponente werden die restlichen Daten eingetragen. Wobei „Created Date“ und „Last Edited“ nicht veränderbar sind.

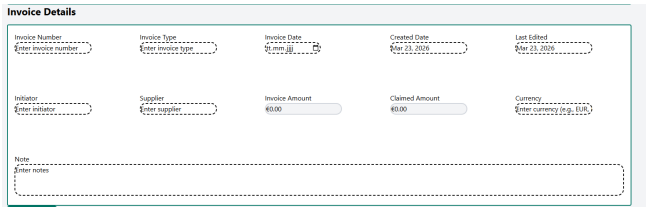


Abbildung 29: Invoice Details im Create-Formular

4.2 Backend

Das Backend stellt eine REST-Schnittstelle zur Verfügung. Über sie wird vom Frontend auf das Backend zugegriffen und das Verarbeiten von Claims und Invoices ermöglicht.

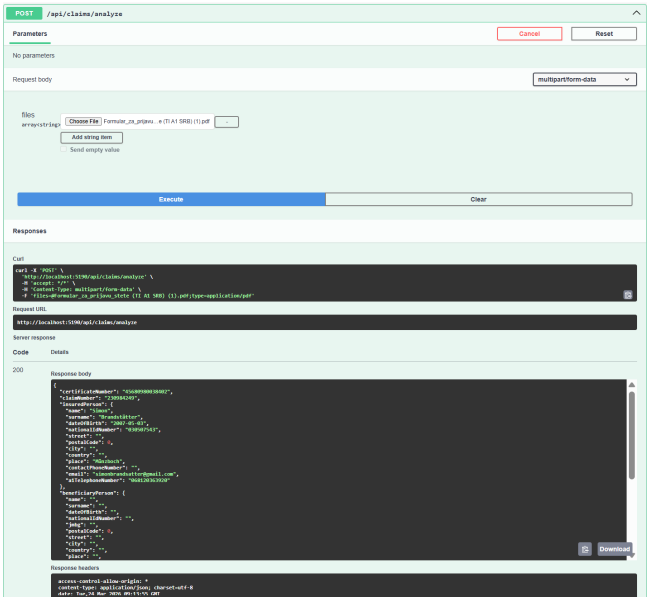


Abbildung 30: Claims Analysieren Endpunkt

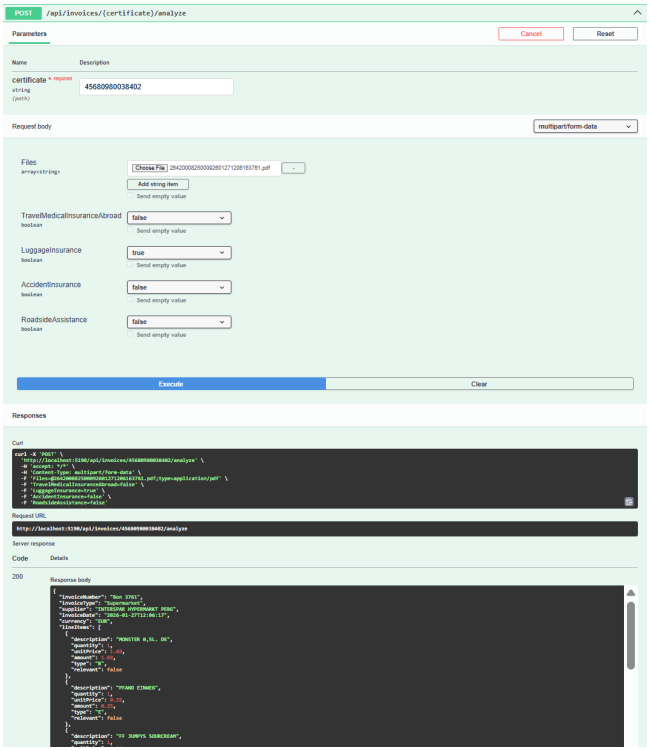


Abbildung 31: Invoices Analysieren Endpunkt

Diese beiden Endpunkte bilden das Herzstück der Diplomarbeit. Sie ermöglichen das Analysieren von Claims und Invoices mit OCR und dann einer Einordnung der Daten mit einem passenden OpenAI modell.

5 Resümee

Die Arbeit dokumentiert die Entwicklung von *ClarioAI*. Es ist ein Prototyp zur Automatisierung von Prozessen in der Versicherungsbranche bei der Firma Lamie. Im Rückblick ist das Entwicklungsteam zufrieden mit dem Ergebnis. Es wurde eine Webapplikation entwickelt, die mithilfe von *AI* und Bilderkennung den Prozess deutlich vereinfacht, wie das Team findet.

5.1 Resümee Frontend

Obwohl ein positives Ergebnis das Produkt dieser Arbeit ist, war die Implementierung mit mehreren technischen Herausforderungen verbunden. Ein wesentliches Problem am Anfang des Entwicklungsprozesses war die Integration von *TailwindCSS* im *Angular*-Projekt. Das liegt daran, dass die offizielle Dokumentation zu diesem Zeitpunkt für *Angular 20* veraltet war. Durch professionelle Hilfe unseres Auftraggebers konnte dieses Problem aber gelöst werden. Es zeigt sich, dass die jahrelange Expertise eines Profis in solchen Situationen sehr wertvoll ist.

Ein Downgrade auf die Version 19 kam jedoch nicht in Frage, weil das keine nachhaltige Lösung gewesen wäre. Die Firma Lamie ist aktuell (März 2026) dabei, sämtliche *Angular*-Projekte auf *Angular 20* zu aktualisieren. Somit macht es nur Sinn, sich dem Umfeld der Firma anzupassen. Bei einer potenziellen Einbindung in das *LUIS-System* (siehe Abschnitt 2.1.3 LUIS) müsste die Firma sonst auch *ClarioAI* upgraden.

Es zeigten sich auch Einschränkungen hinsichtlich der Browserkompatibilität. Während die Anwendung unter *Google Chrome* einwandfrei funktioniert, traten im *Brave*-Browser Fehler auf bei bestimmten Features der Seite. Das verdeutlicht, wie notwendig Cross-Browser-Optimierung wäre.

Zusammenfassend bietet dieser Prototyp eine fundierte Basis, auch wenn für einen produktiven Einsatz noch weitere Optimierungen, insbesondere im Bereich Browser-Kompatibilität, nötig wären.

5.1.1 Resümee Backend

Auch im Backend konnte im Rahmen dieser Arbeit ein funktionsfähiger Prototyp umgesetzt werden, der die zentralen Anforderungen der Anwendung erfüllt. Dabei zeigte sich, dass vor

allem eine gute Strukturierung des Backends wichtig für die Übersichtlichkeit des Systems ist. Die Trennung von Controllern, Services, Datenmodellen und DTOs entpuppte sich als sehr sinnvoll.

Darüber hinaus wurde deutlich, dass moderne Backend-Entwicklung weit über nur die reine Bereitstellung einzelner Endpunkte hinausgeht. Aspekte wie Datenvalidierung, Mapping, Dateiverwaltung, Schnittstellendefinition und die Anbindung externer Dienste spielen eine wesentliche Rolle für die Gesamtfunktionalität. Insbesondere die Verwendung einer klaren REST-Struktur war für die zusammenarbeit mit dem Frontend essentiell.

Ein weiterer wichtiger Punkt war die Verbindung des Backends mit Cloud-Diensten und persistenter Datenspeicherung. Dadurch konnte eine Grundlage geschaffen werden, die schon in die Richtung einer realitätsnahen Anwendung geht. Gleichzeitig zeigte sich, dass mit zunehmender Komplexität des Systems saubere Architekturentscheidungen immer wichtiger wurden.

Zusammenfassend bildet das Backend damit eine stabile Basis für den entwickelten Prototyp. Die umgesetzten Strukturen und Technologien ermöglichen die Weiterentwicklung des Programms.

5.2 Aufgabenverteilung

Die Erstellung dieser Diplomarbeit erfolgte in enger Zusammenarbeit. Die folgende Auflistung gliedert die Verantwortlichkeiten der einzelnen Kapitel nach den jeweiligen Projektmitgliedern:

David Romani:

1 Einleitung
- 1.1 Kurzfassung
- 1.2 Abstract
- 1.3 Motivation
- 1.4 Zielsetzung
- 1.6 Projektumfeld
- 1.7 Projektabgrenzung

2 Theoretische und fachpraktische Grundlagen und Methoden
- 2.1 Grundlegende Fachbegriffe
- 2.2 Verwendete Technologien
  - 2.2.1 Angular
  - 2.2.2 TailwindCSS

3 Implementierung
- 3.1 Frontend

4 Ergebnis
- 4.1 Frontend

5 Resümee
- 5.1 Resümee Frontend

Raphael Becherer:

1 Einleitung

- 1.5 Projektinhalt – Überblick

2 Theoretische und fachpraktische Grundlagen und Methoden

- 2.2 Verwendete Technologien
  - 2.2.3 .NET / C#
  - 2.2.4 SQL Server / Entity Framework

3 Implementierung

- 3.2 Backend
- 3.3 Datenbank

4 Ergebnis

- 4.2 Backend

5 Resümee

- 5.2 Resümee Backend

6 Anhang

6.0.1 ClarioAI-Logo

Abbildung 32: Logo der Diplomarbeit; ClarioAI

6.0.2 Lamie-Logo

Abbildung 33: Logo des Auftraggebers Lamie

Literaturverzeichnis

[1] Piyush Mishra u. a. „Extraction of Information from Handwriting using Optical Character recognition and Neural Networks“. In: *2020 4th International Conference on Electronics, Communication and Aerospace Technology (ICECA)*. Nov. 2020, S. 1328–1333. DOI: 10.1109/ICECA49313.2020.9297418.

[2] State of JS. *State of JavaScript 2024: Front-end Frameworks*. https://2024.stateofjs.com/en-US/libraries/front-end-frameworks/. Abgerufen am 05.01.2026. 2024.

[3] Google. *Angular Documentation*. https://angular.dev/. Abgerufen am 05.01.2026. 2024.

[4] Google Central. *Dependency Injection in Angular*. Offizielle Angular-Dokumentation. 2026. URL: https://angular.dev/guide/di.

[5] Google Central. *Reactive Forms in Angular*. Offizielle Angular-Dokumentation. 2026. URL: https://angular.dev/guide/forms/reactive-forms.

[6] RxJS Community. *RxJS Introduction and Overview*. Reactive Extensions Library für JavaScript. 2026. URL: https://rxjs.dev/guide/overview.

[7] RxJS Community. *Observables in RxJS*. Reactive Extensions Library für JavaScript. 2026. URL: https://rxjs.dev/guide/observable.

[8] Tailwind Labs. *Tailwind CSS Documentation*. https://tailwindcss.com/docs. Abgerufen am 16.02.2026. 2026.

[9] John Polacek. *Let's Define Exactly Atomic CSS*. Abgerufen am 16.02.2026. CSS-Tricks. 2020. URL: https://css-tricks.com/lets-define-exactly-atomic-css/.

[10] Microsoft. *Overview of ASP.NET Core*. 2025. URL: https://learn.microsoft.com/en-us/aspnet/core/overview?view=aspnetcore-10.0.

[11] Microsoft. *ASP.NET Core fundamentals overview*. 2025. URL: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/?view=aspnetcore-10.0.

[12] Microsoft. *Create web APIs with ASP.NET Core*. 2025. URL: https://learn.microsoft.com/en-us/aspnet/core/web-api/?view=aspnetcore-10.0.

[13] Roy T. Fielding, Mark Nottingham und Julian Reschke. *RFC 9110: HTTP Semantics*. https://www.rfc-editor.org/rfc/rfc9110.html. RFC 9110. 2022.

[14] Tim Bray. *RFC 8259: The JavaScript Object Notation (JSON) Data Interchange Format*. https://www.rfc-editor.org/rfc/rfc8259.html. RFC 8259. 2017.

[15] Microsoft. *Format response data in ASP.NET Core Web API*. 2025. URL: https://learn.microsoft.com/en-us/aspnet/core/web-api/advanced/formatting?view=aspnetcore-10.0.

[16] Microsoft. *Handle requests with controllers in ASP.NET Core MVC*. 2024. URL: https://learn.microsoft.com/en-us/aspnet/core/mvc/controllers/actions?view=aspnetcore-10.0.

[17] Microsoft. *ASP.NET Core Middleware*. 2026. URL: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-10.0.

[18] Microsoft. *Dependency injection in ASP.NET Core*. 2026. URL: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-10.0.

[19] Nisarg Subramani u. a. „A Survey of Deep Learning Approaches for OCR and Document Understanding“. In: *NeurIPS 2020 Workshop on Machine Learning Retrospectives, Surveys & Meta-Analyses*. 2020. URL: https://ml-retrospectives.github.io/neurips2020/camera_ready/29.pdf.

[20] Microsoft. *Read model OCR data extraction – Document Intelligence*. 2025. URL: https://learn.microsoft.com/en-us/azure/ai-services/document-intelligence/prebuilt/read?view=doc-intel-4.0.0.

[21] Microsoft. *OCR - Optical Character Recognition*. 2025. URL: https://learn.microsoft.com/en-us/azure/ai-services/computer-vision/overview-ocr.

[22] Microsoft. *Document Intelligence APIs analyze document response*. 2025. URL: https://learn.microsoft.com/en-us/azure/ai-services/document-intelligence/concept/analyze-document-response?view=doc-intel-4.0.0.

[23] Microsoft. *Document processing models - Document Intelligence*. 2025. URL: https://learn.microsoft.com/en-us/azure/ai-services/document-intelligence/model-overview?view=doc-intel-4.0.0.

[24] Microsoft. *What Is Azure Document Intelligence in Foundry Tools?* 2025. URL: https://learn.microsoft.com/en-us/azure/ai-services/document-intelligence/overview?view=doc-intel-4.0.0.

[25] Microsoft. *Overview of Entity Framework Core*. 2024. URL: https://learn.microsoft.com/en-us/ef/core/.

[26] Microsoft. *What Is SQL Server?* 2025. URL: https://learn.microsoft.com/en-us/sql/sql-server/what-is-sql-server?view=sql-server-ver17.

[27] Microsoft. *Databases - SQL Server*. 2025. URL: https://learn.microsoft.com/en-us/sql/relational-databases/databases/databases?view=sql-server-ver17.

[28] Microsoft. *Migrations Overview - EF Core*. 2023. URL: https://learn.microsoft.com/en-us/ef/core/managing-schemas/migrations/.

[29] Microsoft. *Get started with Azure Blob Storage and .NET*. 2025. URL: https://learn.microsoft.com/en-us/azure/storage/blobs/storage-blob-dotnet-get-started.

[30] Microsoft. *Quickstart: Azure Blob Storage client library for .NET*. 2024. URL: https://learn.microsoft.com/en-us/azure/storage/blobs/storage-quickstart-blobs-dotnet.

[31] Microsoft. *BlobClient Class (Azure.Storage.Blobs)*. 2026. URL: https://learn.microsoft.com/en-us/dotnet/api/azure.storage.blobs.blobclient?view=azure-dotnet.

[32] Microsoft. *Delete and restore a blob with .NET - Azure Storage*. 2024. URL: https://learn.microsoft.com/en-us/azure/storage/blobs/storage-blob-delete.

[33] Microsoft. *How to use structured outputs with Azure OpenAI*. 2026. URL: https://learn.microsoft.com/en-us/azure/foundry/openai/how-to/structured-outputs.

[34] Microsoft. *Create and deploy an Azure OpenAI resource*. 2026. URL: https://learn.microsoft.com/en-us/azure/foundry-classic/openai/how-to/create-resource.

[35] Microsoft. *How to switch between OpenAI and Azure OpenAI endpoints*. 2025. URL: https://learn.microsoft.com/en-us/azure/developer/ai/how-to/switching-endpoints.

[36] Microsoft. *How to use JSON mode with Azure OpenAI*. 2026. URL: https://learn.microsoft.com/en-us/azure/foundry/openai/how-to/json-mode.

[37] MapsterMapper. *Mapster*. 2026. URL: https://github.com/MapsterMapper/Mapster.

[38] Microsoft. *Use the Azure OpenAI Responses API*. 2026. URL: https://learn.microsoft.com/en-us/azure/foundry/openai/how-to/responses.

[39] Martin Fowler. *Data Transfer Object*. 2003. URL: https://martinfowler.com/eaaCatalog/dataTransferObject.html.

[40] OpenAI. *o4-mini Model*. OpenAI API Documentation, abgerufen am 26.03.2026. 2025. URL: https://developers.openai.com/api/docs/models/o4-mini.

[41] OpenAI. *GPT-5 mini Model*. 2026. URL: https://developers.openai.com/api/docs/models/gpt-5-mini.

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis