HTBLuVA St. Pölten
Höhere Abteilung für Informatik

# DIPLOMARBEIT

# Entwicklung eines mobilen, cloudbasierten Webfrontends für eine bestehende Arztsoftware

## im Projekt CGM MAXX LITE

**Ausgeführt im Schuljahr 2024/25 von:**

Emily Atzinger, 5BHIF-01
Matthias Zimmermann, 5BHIF-23
Oskar Brödler, 5BHIF-02
Hannes Koppensteiner, 5BHIF-12

**Betreuer/Betreuerin:**

Dipl.-Ing. Werner Gitschthaler, BEd
Ing. Mag. Andreas Schönbichler

St.Pölten, am 24. Juli 2025

# Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Diplomarbeit selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche erkenntlich gemacht habe.

Emily Atzinger

Matthias Zimmermann

Oskar Brödler

Hannes Koppensteiner

St.Pölten, am 24. Juli 2025

# Danksagungen

# Zusammenfassung


**Entwicklung eines mobilen, cloudbasierten Webfrontends für eine bestehende Arztsoftware im Projekt CGM MAXX LITE:**

Das Ziel dieser Diplomarbeit ist die Entwicklung eines mobilen Prototyps der bestehenden Arztsoftware CGM MAXX, der auf Smartphones und Tablets genutzt werden kann, um Arbeitsabläufe in Arztpraxen zu optimieren.

Zu Beginn wird die Softwarearchitektur beleuchtet, indem Herausforderungen und deren Bedeutung hervorgehoben werden. Anschließend folgt eine Anforderungsanalyse als Grundlage für die Wahl eines Architekturstils. Dabei wird eine Auswahl relevanter cloudbasierter Architekturen beschrieben. Zudem werden DevOps-Praktiken und Infrastructure as Code (IaC) betrachtet, um eine effiziente Bereitstellung und Wartung von Software in der Cloud zu ermöglichen.

Das zweite Kapitel befasst sich mit der Gestaltung einer benutzerfreundlichen Weboberfläche. Dabei werden die wesentlichen Prinzipien der User Experience (UX) vorgestellt und Methoden erläutert, wie diese in der Praxis getestet und umgesetzt werden können. Im Anschluss wird der Prozess von der Konzeptentwicklung bis zur fertigen App beschrieben, einschließlich der praktischen Umsetzung und der verwendeten Designansätze.

Das dritte Kapitel widmet sich der Qualität des Quellcodes und behandelt Methoden zur Sicherstellung der Verständlichkeit des Codes, um eine einfache Weiterentwicklung in der Zukunft zu gewährleisten. Zudem wird auf die Bedeutung von statischer Codeanalyse und Code Reviews eingegangen, um Fehler frühzeitig zu identifizieren und die Wartbarkeit des Codes zu verbessern. Ein weiteres Thema ist Refactoring, das dazu dient, den Code zu optimieren, ihn strukturiert und skalierbar zu gestalten, sodass eine langfristige Erweiterung des Projekts möglich ist.

Im letzten Teil der Diplomarbeit wird erklärt, wie die Qualität des Codes anhand von Testing sichergestellt werden kann. Dabei liegt der Fokus auf Behavior-Driven Development (BDD) und End-to-End (E2E)-Testing. Mithilfe von Selenium und Cucumber wird die gesamte Benutzererfahrung getestet, indem Szenarien beschrieben werden, die das Verhalten des Systems aus der Sicht des Benutzers simulieren. Dies ermöglicht eine enge Zusammenarbeit zwischen Entwicklern und Fachabteilungen. Zusätzlich wird das Backend-Testing behandelt, wobei Stubs zum Einsatz kommen, um Komponenten zu simulieren und die Funktionalität des Systems zu überprüfen.

# Abstract

**Development of a Mobile, Cloud-Based Web Frontend for an Existing Medical Software in the CGM MAXX LITE Project:**

The goal of this thesis is to develop a mobile prototype of the existing medical software CGM MAXX, which can be used on smartphones and tablets to optimize workflows in medical practices.

The thesis begins by exploring the software architecture, highlighting the challenges and their significance. This is followed by a requirements analysis that serves as the basis for selecting an architectural style. A selection of relevant cloud-based architectures is described. Additionally, DevOps practices and Infrastructure as Code (IaC) are considered to enable efficient deployment and maintenance of software in the cloud.

The second chapter focuses on the design of a user-friendly web interface. It introduces the fundamental principles of User Experience (UX) and explains methods to test and implement these principles in practice. The process from concept development to the final app is then described, including the practical implementation and design approaches used.

The third chapter is dedicated to the quality of the source code and discusses methods for ensuring the understandability of the code to facilitate future development. The importance of static code analysis and code reviews is also highlighted to identify errors early and improve the maintainability of the code. Another topic is refactoring, which aims to optimize the code, making it structured and scalable, so that the project can be easily expanded in the long term.

The final part of the thesis explains how the quality of the code can be ensured through testing. The focus here is on Behavior-Driven Development (BDD) and End-to-End (E2E) testing. Using Selenium and Cucumber, the entire user experience is tested by describing scenarios that simulate the system's behavior from the user's perspective. This enables close collaboration between developers and business departments. Additionally, backend testing is addressed, using stubs to simulate components and verify the functionality of the system.

# Inhaltsverzeichnis

# Kapitel 1

# Softwarearchitektur mit Schwerpunkt auf cloudbasierten Systemen

## 1.1 Allgemeines

### 1.1.1 Einleitung

Softwarearchitektur - ein Begriff, den nahezu jede/r Softwareentwickler/in kennt. Viele sind sich ihrer Bedeutung bewusst, doch nur die wenigsten schenken ihr die nötige Beachtung. In einer Zeit, in der sich die Technik stetig weiterentwickelt und die Konkurrenz täglich wächst, ist es entscheidend, immer einen Schritt voraus zu sein, um erfolgreich zu bleiben.

Aus diesem Grund versuchen Entwickler/innen, so viele neue Funktionalitäten wie möglich zu implementieren, ohne sich ausgiebig mit der darunterliegenden Architektur zu befassen. Anfangs scheint diese Vorgehensweise gut zu funktionieren, bis der Punkt erreicht wird, an dem es so gut wie unmöglich ist, die Software weiterzuentwickeln oder zu verändern. Dies führt oft zu schwerwiegenden Problemen wie erhöhtem Wartungsaufwand, schwer nachvollziehbaren Abhängigkeiten und letztlich hohen Kosten. Um solche Szenarien vorzubeugen, ist es wichtig, sich genauer mit der Architektur der zu entwickelnden Software zu beschäftigen und sich darüber bewusst zu werden, wie umfangreich dieses Themengebiet überhaupt ist. Hierbei geht es nicht nur um die Strukturierung des Codes, sondern auch darum, wie die einzelnen Komponenten eines Systems miteinander interagieren. Durch eine gut durchdachte Architektur soll der gesamte Lebenszyklus des Softwaresystems unterstützt und so einfach wie möglich gestaltet werden. [EA:Book01, S. 10, S. 136-137]

### 1.1.2 Definition von Softwarearchitektur

Um die Wichtigkeit der Softwarearchitektur zu begreifen, muss zunächst die Frage geklärt werden, was darunter zu verstehen ist. Dies ist jedoch gar nicht so einfach, da keine allgemein gültige, allumfassende Beschreibung existiert.

Grundsätzlich ist die Softwarearchitektur als etwas Dynamisches zu betrachten. Das bedeutet, dass sie sich durch die fortschreitende Entwicklung in der IT ständig verändert. Damit ist auch jede heute gültige Definition in wenigen Jahren wahrscheinlich wieder veraltet. [EA:Book02, S. 1-3]

Befasst man sich genauer mit dem Begriff „Architektur“, wird schnell deutlich, dass es auch hier zahlreiche unterschiedliche Definitionen gibt. Dazu zählen Folgende:

- Organisation von Modulen, Verbindungen, Abhängigkeiten und Schnittstellen
- Modell, um das System als Ganzes zu verstehen
- Standards, Richtlinien und Einschränkungen
- Ergebnis strategischer und technischer Entscheidungen

Zusammenfassend kann gesagt werden, dass es sich hierbei um die **Struktur** eines bestimmten Objekts handelt. In diesem Fall um die Struktur eines Softwaresystems. [EA:Web01]

Bass, Clements und Kazman beschreiben die Softwarearchitektur folgendermaßen:

> “The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relations among them.“ [EA:Book03, S. 3]

Allerdings umfasst die Architektur weit mehr als nur die Struktur eines Systems. Im Mittelpunkt steht der Prozess der Architekturerstellung, bei dem architektonische Treiber wie funktionale und nicht-funktionale Anforderungen, Qualitätsmerkmale, Einschränkungen und Prinzipien in eine technische Lösung überführt werden. Dabei entsteht eine **Vision**[1], die an die Stakeholder kommuniziert werden muss, um eine einheitliche Sichtweise auf das zu entwickelnde Produkt zu gewährleisten. Nur so kann eine erfolgreiche Lösung geschaffen werden. [EA:Web01]

[1] Beschreibt einen wünschenswerten Zustand in der Zukunft

Um die zuvor definierten Aspekte der Softwarearchitektur weiter zu vertiefen, ist es hilfreich, das Thema in Teilbereiche zu zerlegen, die leichter beschrieben werden können.

### Applikationsarchitektur

Wenn man von der Architektur einer einzelnen, bereitstellbaren Applikation spricht, können sich die meisten Softwareentwickler/innen vermutlich etwas darunter vorstellen. Ein gutes Beispiel dafür ist eine Angular-Applikation.

Im Wesentlichen geht es dabei um die **Organisation und Strukturierung des Codes**. Dazu ist es notwendig zu verstehen, wie eine solche Applikation entworfen und implementiert wird. Um dies zu erreichen, ist es essenziell, sich mit den spezifischen Elementen der eingesetzten Technologien sowie mit verschiedenen Entwurfsmustern, Frameworks und Bibliotheken auseinanderzusetzen. [EA:Web01]

### Systemarchitektur

Die Systemarchitektur ist eine Stufe über der Applikationsarchitektur. Heutzutage besteht nämlich so gut wie jedes Softwaresystem aus mehreren auslieferbaren Einheiten, die miteinander interagieren. Betrachtet man beispielsweise eine Arztsoftware, so hat diese eine Benutzeroberfläche, die von medizinischem Fachpersonal oder entsprechenden Hilfskräften bedient werden kann. Hinzu kommt ein im Hintergrund laufendes Backend, das auf die Aktionen von Anwender/innen reagiert und Daten aus einer oder mehreren Datenbanken abruft oder dort speichert.

Diese separaten Applikationen müssen miteinander verbunden werden. Dabei ist zu beachten, dass diese auf unterschiedlichen Technologien basieren können, was unter Umständen zu Problemen führt. Das System wird also auf einer **höheren Ebene** betrachtet. Der Fokus liegt auf der Integration einzelner Anwendungen in ein übergeordnetes System.

Während sich die Applikationsarchitektur hauptsächlich mit Details, wie der genauen Implementierung einer bestimmten Funktionalität befasst, beschäftigt sich die Systemarchitektur sowohl mit **Software als auch Hardware**. Auch wenn die Hardware durch Virtualisierung und Cloud heute kein allzu großes Problem mehr darstellt, muss die Software irgendwo bereitgestellt werden. [EA:Web01]

#### Softwarearchitektur

Nachdem die Teilbereiche nun genauer beschrieben wurden, kann die Softwarearchitektur als **Kombination aus Applikations- und Systemarchitektur** charakterisiert werden.

Die Softwarearchitektur umfasst sowohl die Struktur des Codes als auch das Zusammenspiel der einzelnen Komponenten im System. Dabei stehen Aspekte wie saubere Code-Struktur, Erweiterbarkeit und die Sicherstellung der Funktionalität des Systems im Vordergrund, insbesondere in Bezug auf Sicherheitsanforderungen, Leistungsoptimierungen und zukünftige Entwicklungen. [EA:Web01]

Robert C. Martin veranschaulicht das Konzept der Softwarearchitektur mit einem guten Beispiel. Plant ein/e Architekt/in ein Gebäude, so werden grundlegende Dinge, wie der Grundriss, die äußere Erscheinung und die Raumteilung, geplant. Allerdings gibt es auch einige Kleinigkeiten, die berücksichtigt werden müssen. Dazu zählt unter anderem die Planung von Lichtschaltern, Steckdosen, Heizkörpern und Leitungen. [EA:Book01, S. 4]

Diese Details bezeichnet man als **„Low-Level Details“**, während die übergeordnete Ebene, die zuvor als Systemarchitektur definiert wurde, den **„High-Level Details“** entspricht. Beide Ebenen müssen beachtet werden, um ein Softwaresystem zuverlässig, sicher und wartbar zu gestalten.

### 1.1.3 Aufgaben und Ziele

Da nun klar sein sollte, was Softwarearchitektur ist und womit sie sich befasst, kann näher auf die Aufgaben und Ziele eingegangen werden. Prinzipiell ist es möglich, Software zu entwickeln, ohne sich intensiv mit dem Thema Architektur auseinanderzusetzen. Ein gut durchdachtes Konzept kann jedoch viel Arbeit und die damit verbundenen Kosten sparen.

Ziel ist es, das Entwicklerteam sowie die Projektbeteiligten schlank zu halten, um die Kosten zu minimieren und die Wünsche des Auftraggebenden effizient zu erfüllen. Kurzgefasst lässt sich sagen, dass der **Aufwand minimiert** und die **Produktivität maximiert** werden soll. [EA:Book01, S. 4-5]

Weiters dient die Softwarearchitektur dazu, die **Qualität der Software sicherzustellen**, indem funktionale als auch nicht-funktionale Anforderungen berücksichtigt werden. Diese bilden die Grundlage für die Architektur der zu entwickelnden Software. Was unter diesen Anforderungen zu verstehen ist, wird im Unterkapitel 1.2 näher be-

schrieben. Die Architektur bietet nicht nur den Kund/innen einen Mehrwert, sondern auch das Entwicklerteam profitiert davon. Sie kann als **Kommunikationswerkzeug** eingesetzt werden, um die Planung verschiedener Phasen einfacher zu gestalten und den Entwicklungsprozess zu beschleunigen. Außerdem kann die Architektur dabei helfen, das System besser zu analysieren, wodurch Änderungen schneller vorgenommen und Fehler leichter identifiziert werden können. [EA:Web02]

Eine gut überlegte Softwarearchitektur soll sowohl den Kund/innen als auch dem Projektteam zugutekommen: Aus Sicht der Kundschaft bedeutet das niedrige Kosten, aus Sicht des Projektteams einen reibungslosen Entwicklungsprozess mit klarer Kommunikationsgrundlage und ohne Einschränkungen für die Entwickler/innen.

### 1.1.4 Die Rolle von Softwarearchitekt/innen

Obwohl Entwickler/innen dafür verantwortlich sind, die Architektur der Software im Auge zu behalten und sich an dieser zu orientieren, ist es sinnvoll, eine Person zu haben, die sich auf dieses Themengebiet spezialisiert hat und stets den Überblick behält. Dies unterstützt nicht nur die Entwickler/innen dabei, den roten Faden nicht zu verlieren, sondern trägt auch dazu bei, die **Qualität der Software zu wahren**.

#### Definition

Genauso wie bei der Definition der Softwarearchitektur, ist es schwierig, die Rolle der Softwarearchitekt/innen eindeutig zu definieren. Grundsätzlich ist davon auszugehen, dass ein/e Softwarearchitekt/in eine erfahrene Fachkraft ist, welche die **Richtung vorgibt** und **für maximale Produktivität sorgt**.
[EA:Book01, S. 136-137] [EA:Book02, S. 7-8]

#### Verantwortlichkeiten

Da die Erwartungen und Aufgaben von Softwarearchitekt/innen stark vom Projekt abhängen, können hier nur ein paar allgemeine Grundsätze angeführt werden.
Diese sollten von jeder Person in dieser Rolle erfüllt werden.

Wie bereits erwähnt, muss der/die Architekt/in in der Lage sein, das Projektteam, eine Abteilung oder sogar das gesamte Unternehmen in eine bestimmte Richtung zu lenken, indem er/sie bestimmte **Entscheidungen trifft**. Hierbei ist es entscheidend, sich

nicht nur auf die persönliche Sichtweise zu beschränken, sondern auch dem Entwicklerteam die Möglichkeit zu geben, selbst Entscheidungen zu treffen. Ein Beispiel dafür wäre die Auswahl eines bestimmten Frameworks. Zudem muss der/die Architekt/in regelmäßig kontrollieren, ob diese Entscheidungen auch eingehalten werden.

Dass Softwarearchitekt/innen **viel Wissen und Erfahrung** benötigen, wurde bereits erwähnt. Allerdings reicht dies alleine nicht aus. Um auf zukünftige Entwicklungen gefasst zu sein oder um mit der Konkurrenz mithalten zu können, ist es wichtig, sich laufend mit den neuesten Trends und Technologien auseinanderzusetzen.

Es ist nicht nur technisches Können gefragt, sondern auch die **Fähigkeit, andere zu führen**. Dazu gehört die Koordination des Teams als auch die Kommunikation von Zielen, Prioritäten, Aufgaben und Deadlines. [EA:Book02, S. 7-8] [EA:Web03]

### 1.1.5 Herausforderungen und Probleme

Wie im Abschnitt 1.1.3 beschrieben, bietet eine gute Softwarearchitektur viele Vorteile. Es gibt jedoch einige Herausforderungen, die bewältigt werden müssen, um ihren vollen Nutzen ausschöpfen können. Werden diese Schwierigkeiten nicht überwunden, kann dies sowohl für die Kund/innen als auch für das durchführende Unternehmen zu ernsthaften Problemen führen. Im Folgenden werden die wesentlichen Herausforderungen beschrieben, deren Bewältigung entscheidend ist, um die später dargestellten Probleme zu vermeiden.

Eine der größten Schwierigkeiten besteht darin, **so viele Optionen wie möglich offen** zu **halten**. Der Hintergedanke dabei ist, dass Software entwickelt wurde, um das Verhalten von Maschinen schnell verändern zu können, weswegen Flexibilität eine große Rolle spielt. Ein System muss nicht nur zuverlässig funktionieren, sondern auch leicht veränderbar sein. Die Herausforderung besteht darin, die Business-Logik unabhängig von bestimmten technischen Details oder eingesetzten Technologien zu gestalten.

Ebenso ist es entscheidend, bereits **zu Beginn des Projekts auf die Architektur zu achten**, auch wenn das Team klein ist. Eine simple, monolithische Struktur kann bei einem wachsenden Team zunehmend unübersichtlich werden, besonders dann, wenn das Team groß genug ist, um in mehrere Untergruppen aufgeteilt zu werden. Die Software sollte daher so konzipiert sein, dass unabhängige Komponenten strikt voneinander getrennt sind, um die Übersichtlichkeit des Systems sicherzustellen.

Auch wenn die **Bereitstellung** der Software im ersten Moment keine große Rolle zu spielen scheint, ist es wichtig, sich bereits im Vorhinein Gedanken darüber zu machen, da dieser Prozess ebenfalls sehr aufwändig und kostspielig werden kann.

Zudem muss die **Wartung** der Software berücksichtigt werden, da eine unzureichende Architektur dazu führen kann, dass es lange dauert, eine geeignete Stelle für die Integration neuer Features zu finden. [EA:Book01, S. 136-141]

Doch nun stellt sich die Frage, welche Probleme auftreten können, wenn die zuvor genannten Herausforderungen nicht überwunden werden. Im Folgenden werden Diagramme eines Unternehmens herangezogen, die zeigen, welche Auswirkungen eine schlechte Softwarearchitektur auf die Produktivität und die Kosten haben kann.

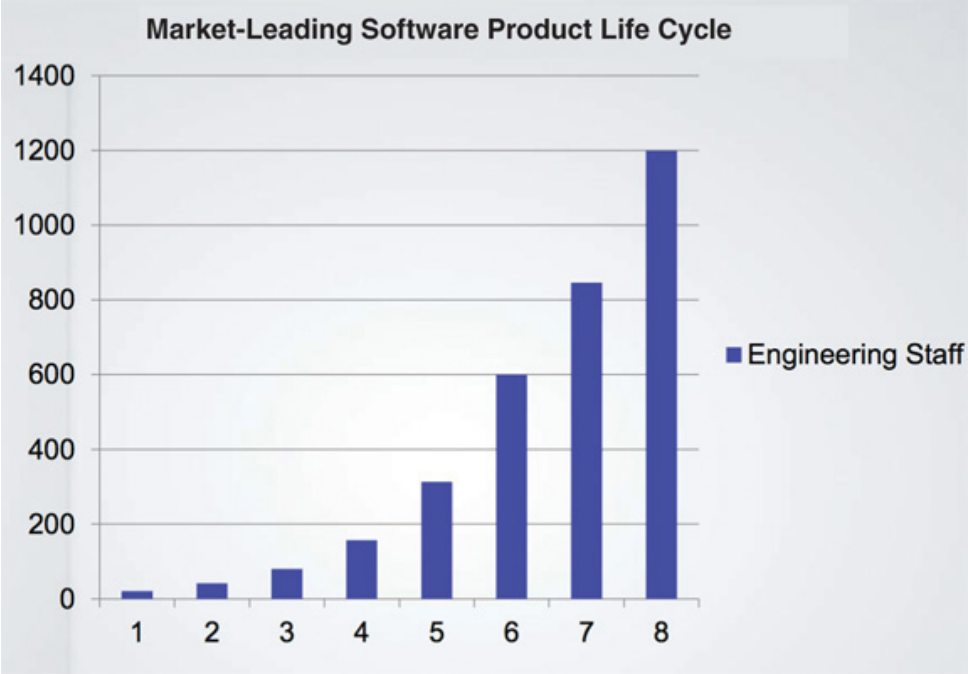


Abbildung 1.1: Personalzuwachs
[EA:Book01, S. 5]

In der Abbildung 1.1 wird dargestellt, wie der Personalaufwand im Laufe der Zeit ansteigt. Das Balkendiagramm zeigt dabei, wie viele Ingenieur/innen an einem bestimmten Release beteiligt sind.

Ein Personalzuwachs muss grundsätzlich nicht negativ sein. Allerdings kann eine zunehmend komplexe und schwer veränderbare Architektur dazu führen, dass immer mehr Personal benötigt wird, um die Software zu warten und weiterzuentwickeln. Eine solche Entwicklung geht häufig mit einer sinkenden Produktivität des Teams einher, da die steigende Komplexität die Effizienz beeinträchtigt und die Implementierung neuer Funktionen erschwert.

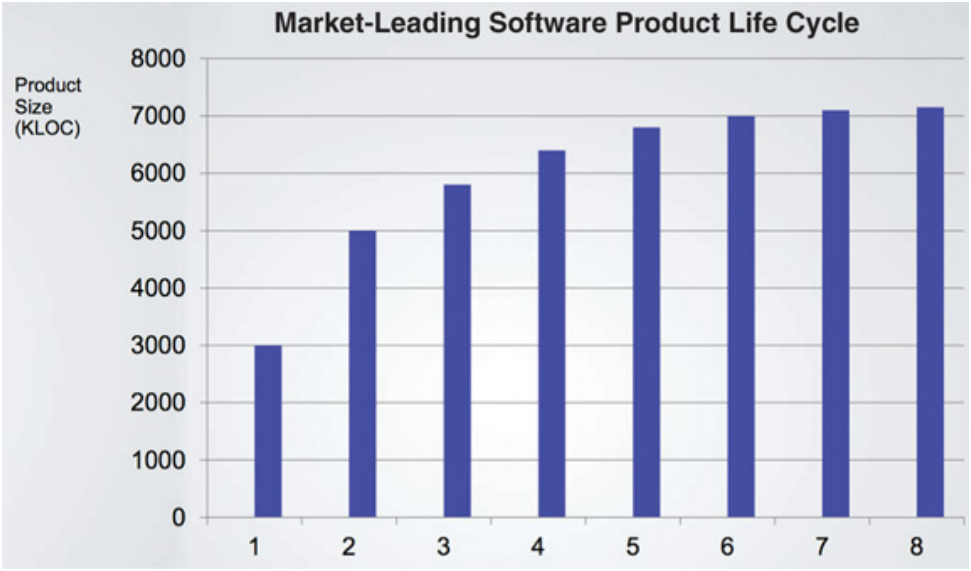


Abbildung 1.2: Produktivität
[EA:Book01, S. 6]

Betrachtet man anschließend die Abbildung 1.2, wird die oben genannte Problematik deutlich. Die Produktivität wird hier anhand der geschriebenen Codezeilen gemessen, was zwar wenig darüber aussagt, ob tatsächlich neue Funktionen oder Änderungen erfolgreich umgesetzt wurden. Mehr Code bedeutet schließlich nicht zwangsläufig, dass dadurch bessere Ergebnisse erzielt werden.

Zu Beginn scheint die Produktivität zwar zu steigen, nähert sich jedoch ab einem gewissen Punkt einer Asymptote[2]. Das bedeutet, dass das Hinzufügen weiterer Ressourcen nicht automatisch zu einer Steigerung der Produktivität führt.

Das Problem dabei ist nicht nur, dass das Unternehmen weitere Personen einstellen und bezahlen muss, sondern auch, dass die derzeitigen Mitarbeiter/innen darunter leiden könnten. Eine abnehmende Produktivität kann zu **Motivationsverlust** führen, sodass Mitarbeitende im schlimmsten Fall sogar das Unternehmen verlassen. [EA:Book01, S. 5-10]

[2]Linie, der sich eine Kurve immer weiter annähert, ohne sie jemals zu berühren

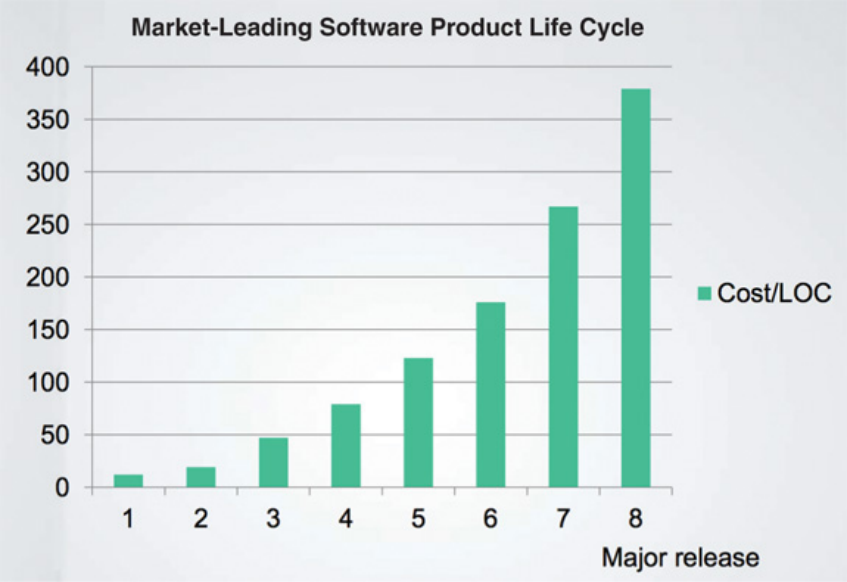


Abbildung 1.3: Kostenentwicklung
[EA:Book01, S. 7]

Ein Blick auf die Entwicklung der Kosten in Abbildung 1.3 zeigt, dass die Kosten trotz kaum ansteigender Produktivität (siehe Abbildung 1.2) und steigendem Personalbedarf (siehe Abbildung 1.1) immer weiter steigen. Dies bedeutet, dass der **Kunde oder die Kundin enorme Summen für kaum sichtbare Fortschritte investiert**.

Ist der Kunde oder die Kundin nicht gewillt, dafür zu bezahlen, kann dies zum Abbruch des Projekts führen, wodurch das Unternehmen in ein schlechtes Licht gerückt wird und im schlimmsten Fall mit der Auflösung des Unternehmens enden kann.
[EA:Book01, S. 5-10]

All diese Probleme können durch eine „gute“ Softwarearchitektur vermieden werden. Doch was versteht man darunter? Was ist gut und was ist schlecht? Fakt ist, dass es keine universelle Lösung für die Softwarearchitektur gibt. Abhängig vom Anwendungsfall, Anforderungen, Teamgröße und vielen weiteren Faktoren kann eine andere Architektur vorteilhaft sein. Im weiteren Verlauf wird näher auf die Anforderungen einer Software eingegangen, die notwendig sind, um einen passenden Architekturstil auszuwählen.

## 1.2 Anforderungsanalyse

Nachdem in der Einleitung 1.1 ausführlich erläutert wird, warum eine durchdachte Softwarearchitektur von so großer Bedeutung ist, wird anschließend detaillierter auf die Anforderungen einer Software eingegangen. Diese sind entscheidend für den Erfolg eines Softwaresystems.

Grundsätzlich wird angenommen, dass Anforderungen dazu dienen, ein spezifisches Problem zu lösen. Diese Definition lässt sich auch auf die Anforderungen eines Softwaresystems übertragen.
Der IEEE-Standard 729 definiert Softwareanforderungen folgendermaßen:

> "A condition or capability needed by a user to solve a problem or achieve an objective." [EA:Web05]

Anforderungen werden von verschiedenen Beteiligten wie Benutzern, Kunden, Entwicklern und Business Analysts definiert.

Zusammengefasst dienen Anforderungen dazu, festzulegen, **welche Probleme die zu entwickelnde Software lösen soll**. Die daraus gewonnen Informationen geben allen Beteiligten einen einheitlichen Überblick und dienen als Entscheidungsgrundlage für einen Architekturstil. Allerdings kann es vorkommen, dass die gewählte Architektur zu einem späteren Zeitpunkt nicht mehr optimal ist, da im Laufe des Projekts unerwartete Probleme auftreten können.
[EA:Web04, EA:Web05] [EA:Book02, S. 13-16]

Dies ist besonders relevant bei **agilen Vorgehensweisen**, bei denen die Zufriedenheit des Kunden im Vordergrund steht. **Laufende Anpassungen** werden vorgenommen, um den Erwartungen des Kunden gerecht zu werden, was bedeutet, dass sich die Anforderungen im Laufe der Zeit ändern können. [EA:Web06]

Grundsätzlich lassen sich Anforderungen in zwei Haupttypen unterteilen:

- Funktionale Anforderungen
- Nicht-funktionale Anforderungen

Diese werden im Folgenden genauer erklärt. Anschließend erfolgt eine Anforderungsanalyse für das Projekt CGM MAXX LITE, die dabei helfen soll, einen passenden Architekturstil für das Projekt zu finden, um die Entwicklung, Bereitstellung und Wartung so einfach und kostengünstig wie möglich zu gestalten.

### 1.2.1 Funktionale Anforderungen

Der Begriff „funktionale Anforderungen“ deutet bereits darauf hin, dass es um die konkrete Funktionalität der Software geht. Diese Anforderungen definieren, **welche Aufgaben ein System erfüllen muss**. Genauer gesagt handelt es sich um sichtbare Funktionen und Prozesse, die der Benutzer sehen und verwenden kann. Der Benutzer führt bestimmte Aktionen durch, die eine festgelegte Operation auslösen und als Antwort eine entsprechende Reaktion des Systems liefern.

Da funktionale Anforderungen immer spezifisch auf die zu entwickelnde Software zugeschnitten sind, **können** sie **je nach Projekt stark variieren**. Der genaue Inhalt dieser Anforderungen hängt davon ab, welche Ziele die Software verfolgt und welche Probleme sie lösen soll. Im Abschnitt 1.2.3 wird näher auf die funktionalen Anforderungen des Softwareprojekts CGM MAXX LITE eingegangen.

Funktionale Anforderungen werden im **agilen Entwicklungsprozess** häufig in Form von **User Stories** formuliert. Diese beschreiben aus der Perspektive einer bestimmten Rolle, welche Funktionen umgesetzt werden sollen. [EA:Web04, EA:Web07]

### 1.2.2 Nicht-funktionale Anforderungen

Neben den funktionalen Anforderungen gibt es auch die nicht-funktionalen Anforderungen, die auch als architektonische Eigenschaften bezeichnet werden. Dabei handelt es sich um bestimmte Kriterien, die erfüllt sein müssen, **damit ein Softwaresystem erfolgreich betrieben werden kann**.

Grundsätzlich werden diese Anforderungen in bestimmten Dokumenten angeführt. Allerdings gibt es einige Eigenschaften, welche zwar essentiell für den Projekterfolg sind, aber nicht explizit genannt werden, da davon ausgegangen wird, dass die Software verfügbar, verlässlich und sicher ist.

Diese Anforderungen können für die unterschiedlichsten Bereiche der Software definiert werden. Um ein besseres Verständnis für die verschiedenen Eigenschaften zu bekommen, können diese in Kategorien unterteilt werden. Anzumerken ist, dass es hierbei ebenfalls keine allgemein gültige Aufteilung gibt.

Allerdings ist es nahezu unmöglich, all diese Anforderungen wirklich zu erfüllen, da sich die einzelnen Eigenschaften gegenseitig beeinflussen können.
[EA:Web04, EA:Web05] [EA:Book02, S. 55-60]

### Betriebsrelevante Anforderungen

Für den Betrieb einer Software sind bestimmte Eigenschaften von großer Bedeutung. Dazu zählt insbesondere die **Verfügbarkeit**, die angibt, wann und wie lange das System zur Verfügung stehen muss und welche Maßnahmen im Falle eines Ausfalls ergriffen werden. Ein weiterer wichtiger Aspekt ist die **Performance**, die durch die maximale Antwortzeit[3] definiert wird. Hierbei muss berücksichtigt werden, dass die Reaktionszeit der Applikation je nach Nutzung und Anzahl der Benutzer stark variieren kann. Besonders in Systemen mit einer hohen Benutzerzahl kann es zu Verzögerungen kommen.

Weitere wesentliche Aspekte sind die **Wiederherstellbarkeit, Verlässlichkeit und die Sicherheit** des Systems. Vor allem bei Applikationen, bei denen Menschenleben gefährdet werden könnten, ist besondere Vorsicht geboten. Es muss nicht zwangsläufig ein vollständiger Systemausfall vorliegen. Alleine das bloße Fehlen oder der Verlust von wichtigen Informationen kann zu ernsthaften Komplikationen führen.
Grundsätzlich sollte bei jeder Software auf Sicherheit geachtet werden, um unbefugten Zugriff zu verhindern. Der potenzielle Schaden steigt mit der Anzahl der Nutzer, die auf das System zugreifen, sowie mit der Menge an verwalteten Daten. Dies gilt insbesondere für sensible und personenbezogene Daten.
[EA:Book02, S. 58-59] [EA:Web08]

Aus diesem Grund ist es sinnvoll, bereits in der Anforderungsanalyse die Wiederherstellbarkeit des Systems zu berücksichtigen. Klare Anforderungen, die definieren, wie das System im Falle eines Ausfalls oder Fehlers reagieren soll, helfen, die Software schnell und effektiv wiederherzustellen. Diese Maßnahmen tragen dazu bei, die Zuverlässigkeit des Systems zu gewährleisten und es auch nach Störungen weiterhin funktionsfähig zu halten. [EA:Web09]

### Strukturelle Anforderungen

Auch der Code darf nicht außer Acht gelassen werden, weshalb Kriterien festgelegt werden, um die Qualität der Software sicherzustellen. Dazu gehören Eigenschaften wie die **Erweiterbarkeit**, die eine wichtige Rolle spielt, wenn das System in Zukunft um weitere Funktionen ergänzt werden soll.

Die **Wiederverwendbarkeit** bestimmter Komponenten kann in einigen Anwendungsfällen von Bedeutung sein, ebenso wie die **Lokalisierung**[4] für größere Unternehmen, die in mehreren Ländern tätig sind.

[3] Zeitspanne zwischen Aktion eines Nutzers und der Antwort des Systems
[4] Unterstützung verschiedener Sprachen, Maßeinheiten und Währungen

Wie bereits im Abschnitt 1.1.5 erwähnt, muss auch die **Wartung** berücksichtigt werden, um die Kosten niedrig zu halten. Dabei muss überlegt werden, wie Änderungen effizient umgesetzt werden können, um das System zu verbessern.

Bei der Entwicklung eines Prototyps kann es ebenfalls wichtig sein, die **Portabilität** zu beachten, speziell dann, wenn Teile der Software in ein bereits bestehendes System integriert werden sollen, das auf einer anderen Technologie basiert.
[EA:Book02, S. 59-60]

### Bereichsübergreifende Anforderungen

Es gibt viele weitere Kriterien, die das Design der Software maßgeblich beeinflussen können. Dazu gehören die **Zugänglichkeit** in Bezug auf Barrierefreiheit, sowie bestimmte Sicherheitsaspekte wie die **Authentifizierung und Autorisierung**. Auch **rechtliche Aspekte** müssen berücksichtigt werden, etwa die Datenschutz-Grundverordnung (DSGVO). Zudem spielt **Usability** eine große Rolle, damit die Anwender/innen bedenkenlos mit der Software arbeiten können, ohne viel Zeit damit zu verbringen, nach den verfügbaren Funktionalitäten zu suchen.
[EA:Book02, S. 60]

### Überleitung

Im folgenden Abschnitt werden die Anforderungen des Projekts CGM MAXX LITE genauer analysiert. Hierbei werden sowohl die Erwartungen des Auftraggebers, wie sie im Pflichtenheft festgelegt sind, als auch die aus der Anwendersicht formulierten Anforderungen, die als User Stories in Jira dokumentiert sind, berücksichtigt. Ziel ist es, die Eigenschaften zu definieren, die das System schlussendlich erfüllen soll, um sowohl den Auftraggeber zufriedenzustellen als auch den Anforderungen der Anwender/innen gerecht zu werden. Außerdem sollen die daraus gewonnenen Erkenntnisse als Entscheidungsgrundlage für die Auswahl eines Architekturstils dienen, worauf im Unterkapitel 1.3 näher eingegangen wird.

### 1.2.3 Projektbezug - Anforderungsanalyse

Bevor die Anforderungen des Projekts CGM MAXX LITE genauer unter die Lupe genommen werden, muss zunächst die Frage geklärt werden, worum es sich hierbei überhaupt handelt. Alle Informationen, die in diesem Abschnitt genannt werden, stammen von der Website des Auftraggebers, aus dem Pflichtenheft und den User Stories und Tasks in Jira. Im Folgenden wird beschrieben, wie der Projektauftrag zustande kam und was genau durch dieses Projekt erreicht werden soll.

#### Ausgangssituation

Der Name verrät bereits, von welchem Unternehmen der Projektauftrag stammt. Dieses heißt **CGM Arztsysteme Österreich GmbH** und ist spezialisiert auf Software im Gesundheitssektor. Dabei ist **CGM MAXX** eines ihrer Hauptprodukte, ein Arztinformationssystem, das sowohl von Wahl- als auch von Kassenärzten genutzt wird.

Um etwas genauer zu sein, handelt es sich um eine webbasierte Arztsoftware, die diversen Ärzten digitale Dienste zur Verfügung stellt, wie die Patientenverwaltung, Befund-Dokumentierung, Terminplanung und vieles mehr. Allerdings ist diese Applikation **nur für den Desktop-Betrieb optimiert** und basiert auf veralteten Technologien. [EA:Web10]

#### Soll-Situation

Hier setzt das Projekt **CGM MAXX LITE** an. Ziel ist es, die Anwendung auch **auf mobilen Geräten** wie Smartphones und Tablets bereitzustellen, um **alltägliche Arbeitsabläufe zu erleichtern**.

Dabei soll ein Prototyp für ein mobiles Frontend entwickelt werden, das die wichtigsten Funktionalitäten von CGM MAXX enthält und zusätzlich Funktionen zur Foto- und Audioaufnahme bietet, um die Vorzüge mobiler Geräte zu nutzen. Durch die Bereitstellung einer Swagger-Dokumentation, die die einzelnen Schnittstellen der vorhandenen Software genau beschreibt, soll ein Demo-Backend implementiert werden, damit dieses gegebenenfalls mit dem bereits bestehenden Backend von CGM MAXX ausgetauscht und getestet werden kann.
Der Fokus liegt dabei auf einer **benutzerfreundlichen Oberfläche für mobile Endgeräte**. Zudem soll die Software so gestaltet werden, dass sie für zukünftige Erweiterungen offen bleibt, um die Weiterentwicklung und Wartung möglichst effizient und kostengünstig zu gestalten.

### Funktionale Anforderungen

Was genau unter funktionalen Anforderungen zu verstehen ist, kann im Abschnitt 1.2.1 nachgelesen werden. Kurz gesagt geht es darum, welche Funktionalitäten die Software bieten soll. Zur Definition der funktionalen Anforderungen werden die User Stories und Tasks aus Jira herangezogen.

Da es zu unübersichtlich wäre, jede einzelne User Story und jeden Task aufzulisten, folgt eine kurze Übersicht der Funktionalitäten, die die Software grundsätzlich erfüllen soll:

- Dark-/Light Mode
- Patientenliste
- Patientensuche
- Aufnahme und Darstellung von Bildern
- Aufnehmen und Abspielen von Audiodateien

Da diese Auflistung noch zu ungenau ist, um ein klares Bild zu vermitteln, werden die grundlegenden Funktionalitäten der zu entwickelnden Software nun näher beschrieben.

Sobald die Applikation gestartet wird, soll eine **Liste der Patient/innen angezeigt** werden, die am heutigen Tag einen Termin haben. Dies ermöglicht es dem/der Anwender/in, schnell an Informationen zu gelangen, ohne lange suchen zu müssen.
Des Weiteren soll es möglich sein, **Personen anhand ihres Namens oder ihrer Sozialversicherungsnummer zu suchen**. Diese können auf der Startseite angepinnt werden, falls öfter auf die Daten einer bestimmten Person zugegriffen werden muss.
Ein weiterer wichtiger Aspekt ist eine **Navigationsleiste**, die es dem/der Anwender/in ermöglicht, schnell durch die Applikation zu navigieren. Sie soll so platziert sein, dass sie einfach mit einer Hand erreicht werden kann, um die Benutzerfreundlichkeit zu erhöhen.
Wie bereits erwähnt, sollen die Vorteile mobiler Geräte genutzt werden, weshalb es möglich sein soll, **Bilder von medizinischen Befunden** direkt **in der Patientenakte** zu **hinterlegen**. Dadurch kann sich der/die Arzt/Ärztin auf den/die Patient/in konzentrieren, ohne zu viel Zeit mit administrativen Aufgaben zu verbringen.
Neben Fotos sollen auch **Audioaufnahmen gespeichert** werden können. Hierbei können Sitzungen aufgezeichnet werden, die in Zukunft genutzt werden können, um die wichtigsten Informationen mithilfe einer KI zu extrahieren und in kurzen Worten zusammenzufassen.

### Nicht-funktionale Anforderungen

Nicht-funktionale Anforderungen definieren Kriterien, die ein System erfüllen muss, um den Anforderungen der Nutzerinnen und Nutzer sowie den Qualitätsansprüchen des Projekts gerecht zu werden. Detaillierte Informationen hierzu sind im Abschnitt 1.2.2 zu finden.

Mit Ausnahme der **Benutzerfreundlichkeit** wurden vom Auftraggeber keine weiteren nicht-funktionalen Anforderungen festgelegt. Dennoch gibt es bestimmte Eigenschaften, die vor allem bei medizinischen Anwendungen, von besonderer Bedeutung sind. Ein zentraler Aspekt ist die **Performance**, die sicherstellen soll, dass alle Arbeitsabläufe reibungslos und ohne signifikante Verzögerungen ablaufen. Darüber hinaus ist die **Vertraulichkeit** von personenbezogenen Daten essenziell, um den Schutz sensibler Informationen zu gewährleisten wird. Ein weiterer wichtiger Faktor ist die **Ausfallsicherheit**. Insbesondere bei Applikationen, deren Funktionalität potenziell Auswirkungen auf die Sicherheit oder das Wohlbefinden von Menschen hat, muss sichergestellt werden, dass bei einem Fehler nicht das gesamte System abstürzt, sondern nur betroffene Teile.

In Bezug auf die Benutzerfreundlichkeit sollten folgende Punkte berücksichtigt werden:

- Eine leicht verständliche und selbsterklärende Benutzeroberfläche
- Der/die Anwender/in soll auf bestehende Probleme hingewiesen werden
- Arbeitsabläufe sollen mit minimalem Aufwand erledigt werden können
- Verwendung eines einheitlichen und schlichten Farbschemas

## 1.3 Architekturstile

Wie in Abschnitt 1.1.2 beschrieben, umfasst die Softwarearchitektur sowohl die Struktur eines Systems als auch die einzelnen Details seiner Umsetzung. Die Anforderungen eines Softwaresystems variieren je nach Art der Anwendung, weshalb unterschiedliche Ansätze erforderlich sind.

Im Laufe der Zeit haben sich bestimmte „Muster“ etabliert, die als Architekturstile bezeichnet werden. Benannt werden diese nach bestimmten architektonischen Eigenschaften, die Aufschluss über die Topologie oder potenzielle Herausforderungen geben.

Die Wahl eines geeigneten Architekturstils ist von entscheidender Bedeutung, da sie die grundlegende Struktur des Systems festlegt und den Entwickler/innen als Leitfaden dient. Zudem wird vermieden, dass ein **„Big Ball of Mud**[5]**“** entsteht.

Für eine fundierte Entscheidung wird die Anforderungsanalyse 1.2.3 herangezogen, die festlegt, welche Aspekte besonders relevant sind.

Ein paar gängige Beispiele:

- Monolithische Architektur
- Serviceorientierte Architektur (SOA)
- Eventbasierte Architektur
- Serverlose Architektur
- Microservices-Architektur

Um den Fokus auf cloudbasierte Systeme zu richten, wird im Anschluss eine Auswahl relevanter Architekturstile beschrieben, die in modernen Cloud-Umgebungen besonders häufig Anwendung finden. [EA:Book02, S. 125-126] [EA:Web60]

[5] Anti-Pattern, bei dem keine erkennbare Softwarearchitektur existiert

### 1.3.1 Microservices-Architektur

Die nach Fowler und Lewis benannte „Microservices-Architektur“ ist einer der am häufigsten eingesetzten Architekturstile der heutigen Zeit. Besonders in den letzten Jahren hat dieser Architekturstil stark an Bedeutung gewonnen.

Dabei wird nicht eine große Applikation entwickelt, die als eine Einheit bereitgestellt werden kann, sondern mehrere voneinander unabhängige Services, die miteinander interagieren und so ein System bilden. Es handelt sich um eine **verteilte Architektur**. Die Unterteilung in die einzelnen Komponenten erfolgt anhand des **Domain-Driven-Designs (DDD)**. [EA:Book02, S. 251-253]

Ziel ist es, die grundlegenden Komponenten des Systems eigenständig zu betreiben, sodass im Falle eines Fehlers nur ein Dienst ausfällt und nicht das gesamte System. [EA:Web61]

#### Aufbau

In Abbildung 1.4 wird dargestellt, wie der allgemeine Aufbau einer Microservices-Architektur aussieht:

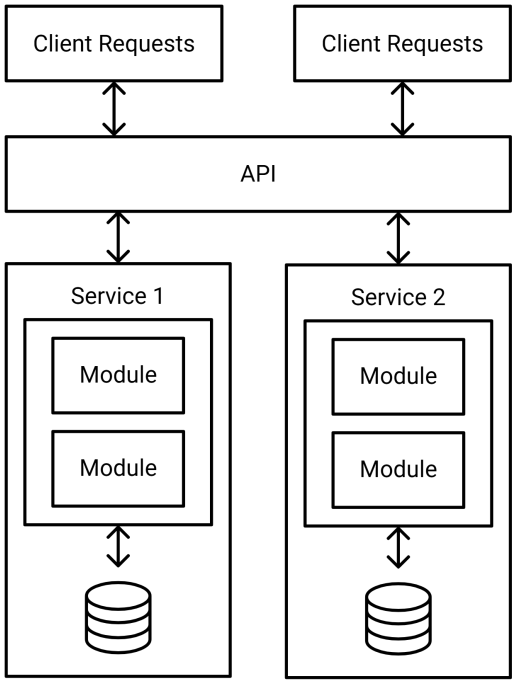


Abbildung 1.4: Microservices Topologie
[EA:Book02, S. 252]

Grundsätzlich gibt es eine zentrale API-Schnittstelle, die die Anfragen der Clients an die entsprechenden Microservices weiterleitet. Bestehen Abhängigkeiten zwischen den Diensten, erfolgt die Kommunikation ebenfalls über die gemeinsame Schnittstelle. [EA:Web61]

Eine dedizierte Datenbank pro Service soll dabei für **Fehlertoleranz** sorgen, sodass, wie vorhin bereits erwähnt, nur die vom Fehler betroffenen Teile des Systems ausfallen, während andere Dienste und Daten weiterhin verfügbar bleiben. [EA:Book02, S. 268]

Da es sich hierbei um einen cloudnativen Ansatz handelt, ist es üblich, moderne Methoden wie **DevOps** (siehe Abschnitt 1.5.2) zu verwenden, um Software einfach und schnell zu entwickeln und in der Cloud bereitzustellen (siehe Unterkapitel 1.4). Um dies zu ermöglichen, ist es oft notwendig, die Organisation so umzustrukturieren, dass sie in mehrere kleine Teams unterteilt wird, die für unterschiedliche Dienste verantwortlich sind. In der Theorie bietet dies den Vorteil, dass durch die unabhängigen Dienste weniger oder gar keine Konflikte mehr auftreten und die Auslieferung somit schneller erfolgen kann. [EA:Web62]

### Vor- und Nachteile

Die Microservices-Architektur bietet dabei einige Vor- und Nachteile, die tabellarisch in 1.1 dargestellt werden. Bei einer gut durchdachten Umsetzung können einige der aufgelisteten Nachteile vermieden werden.
[EA:Book02, S. 253, 268-269] [EA:Web62]

| Vorteile | Nachteile |
|---|---|
| + Unterstützung agiler Vorgehensweisen | - Netzwerkbedingte Performanceverluste |
| + Hohe Fehlertoleranz | - Komplexer Aufbau |
| + Flexible Skalierung | - Erschwertes Testen & Debuggen |
| + Einfache Wartung & Erweiterbarkeit | - Erhöhter Ressourcenbedarf |

Tabelle 1.1: Vor- und Nachteile der Microservices-Architektur

### Einsatzgebiete

Unternehmen, die eine Vielzahl unterschiedlicher Dienste über das Internet bereitstellen, profitieren von den Vorteilen der Microservices-Architektur. Diese hat nicht nur einen positiven Einfluss auf die Softwareentwicklung, sondern optimiert auch die Nutzererfahrung.

Denn durch die flexible Skalierung einzelner Applikationen lassen sich Engpässe durch gezieltes Hinzufügen oder Entfernen von Ressourcen effektiv vermeiden. [EA:Web62]

Folgende Beispiele verdeutlichen die Relevanz der Microservices-Architektur:

- **Social-Media-Plattformen**
  - z.B. Instagram, Facebook, X, usw.
- **Video-Streaming-Dienste**
  - z.B. Netflix, Amazon Prime, usw.
- **E-Commerce-Webseiten**
  - z.B. eBay, Amazon, usw.

### 1.3.2 Eventbasierte Architektur

Ein weiterer Architekturstil, der oft im Zusammenhang mit der Microservices-Architektur eingesetzt wird, ist der **eventbasierte Architekturstil**. Ähnlich wie bei der Microservices-Architektur ist auch dies eine verteilte Architektur, bei der die **Kommunikation** zwischen den einzelnen Bestandteilen des Systems **asynchron**[6] abläuft.
[EA:Book02, S. 183]

Wie bei anderen Architekturstilen gibt es verschiedene Möglichkeiten zur Umsetzung. In diesem Fall unterscheidet man zwischen **zwei Mustern**, die zur Realisierung einer eventbasierten Architektur genutzt werden:

- **Publisher & Subscriber**
  - Subscriber erhalten Benachrichtigungen von den abonnierten Kanälen und reagieren darauf.
- **Event Streaming**
  - Subscriber können den gesamten Stream lesen
  - Nur die für den Subscriber relevanten Informationen werden verarbeitet

[EA:Web63]

[6] Senden/Empfangen von Daten ist zeitlich versetzt

### Aufbau

Abbildung 1.5 stellt dabei eine Form der eventgesteuerten Architektur dar.

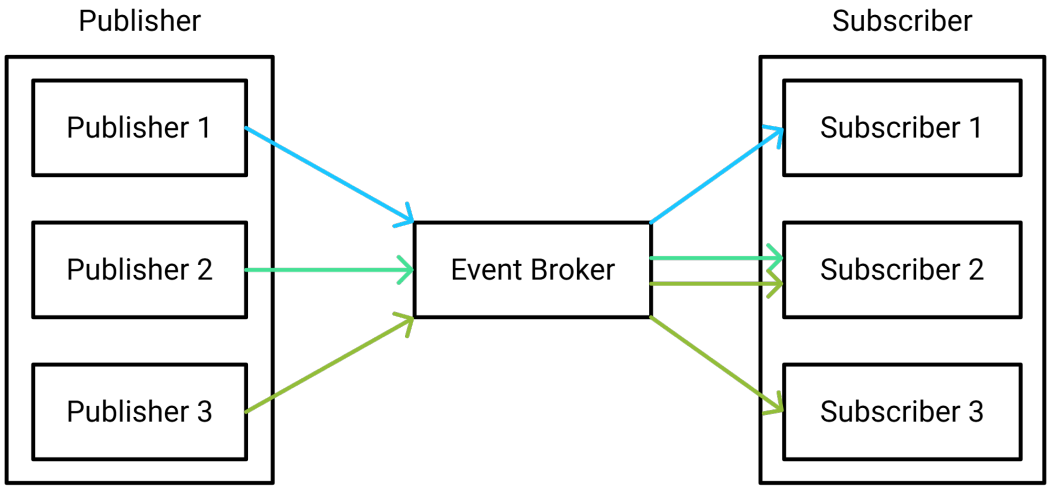


Abbildung 1.5: Eventbasierte Architektur - Publisher/Subscriber [EA:Img02]

Das Publisher/Subscriber-Muster besteht aus folgenden Bestandteilen:

- **Publisher**
  - Lösen Ereignisse aus und senden diese an den Event Broker
  - z.B. Bedienung einer Benutzeroberfläche
- **Subscriber**
  - Konsumieren und verarbeiten Ereignisse
  - Empfangen Ereignisse abonnierter Event Channels des Brokers
- **Broker**
  - Kümmert sich um die richtige Weiterleitung der Ereignisse

[EA:Web63]

### Vor- und Nachteile

Die eventbasierte Architektur bietet einige Vor- und Nachteile:

| Vorteile | Nachteile |
|---|---|
| + Hohe Fehlertoleranz | - Erschwertes Testen & Debuggen |
| + Hohe Performance | - Risiko von Datenverlusten |
| + Hohe Skalierbarkeit | - Komplexer Ablauf & Implementierung |
| + Einfache Erweiterbarkeit | - Schwierige Fehlerbehandlung |
| + Echtzeitunterstützung | |

Tabelle 1.2: Vor- und Nachteile der eventbasierten Architektur

Da dieser Architekturstil ebenfalls auf den Konzepten verteilter Systeme basiert, bietet er ähnliche Vorteile wie die Microservices-Architektur. Zudem eignet sich die ereignisgesteuerte Architektur für Echtzeit-Anwendungen, da sofort auf eintretende Ereignisse reagiert werden kann.

Allerdings bringt die asynchrone Kommunikation einige Herausforderungen mit sich. Dazu zählt insbesondere die **Koordination der einzelnen Events**, sodass deren Reihenfolge erhalten bleibt und ein kontrollierter Ablauf gewährleistet wird. Da Ereignisse dynamisch eintreten, ist es oft schwierig, den genauen Ablauf nachzuvollziehen und mögliche Fehlerquellen zu identifizieren.
[EA:Book02, S. 183, S. 205, S. 211-213]

### Einsatzgebiete

Der eventgesteuerte-Architekturstil hat zahlreiche Anwendungsfälle in den unterschiedlichsten Bereichen. Dazu gehören:

- **Finanzapplikationen**
  - z.B. Subscriber informieren, sobald sich der Kurs einer Aktie ändert
- **Internet of Things (IoT)**
  - z.B. Überwachung von Sensordaten
- **Software im Gesundheitswesen**
  - z.B. Automatische Aktualisierung von Patientendaten

[EA:Web64] [EA:Book02, S. 184]

### 1.3.3 Serverlose Architektur

Der letzte cloudnative Architekturstil, der im Zuge dieser Arbeit beschrieben wird, ist die **serverlose Architektur**. Bei dieser Art von Architektur steht weniger der eigentliche Aufbau eines Systems im Vordergrund, sondern die Verwaltung der benötigten Ressourcen. Die serverlose Architektur wird häufig in Kombination mit der Microservices-Architektur (siehe 1.3.1) oder der ereignisgesteuerten Architektur (siehe 1.3.2) eingesetzt.

Ziel dieser Architektur ist es, den Fokus der Entwickler/innen auf das Schreiben des eigentlichen Codes zu lenken, ohne sich mit den Anforderungen der für den Betrieb notwendigen IT-Infrastruktur auseinandersetzen zu müssen. Stattdessen übernimmt ein entsprechender Cloud-Dienstanbieter diese Aufgabe, einschließlich der Bereitstellung und Skalierung der Applikationen. Dadurch muss ein Unternehmen **nur für die tatsächlich verwendeten Rechenressourcen bezahlen**.

Wird eine bereitgestellte Applikation nicht verwendet, so werden auch keine Ressourcen benötigt. Das heißt, **Ressourcen werden dynamisch bereitgestellt**.

Die serverlose Architektur lässt sich dabei in zwei Bereiche eingliedern:

1. **Backend-as-a-Service (BaaS)**
    - Integration von Drittanbieterdiensten und Applikationen möglich
    - Kommunikation erfolgt über APIs
    - Auslagerung bestimmter Funktionalitäten - z.B. Authentifizierung
2. **Function-as-a-Service (FaaS)**
    - Ereignisgesteuert
    - Bereitgestellter Code wird nur bei Aufruf ausgeführt

Da es für diesen Architekturstil keine allgemein gültige Topologie gibt, sondern abhängig ist von der eingesetzten Technologie, wird auf die Darstellung des Aufbaus verzichtet. Stattdessen werden Technologien genannt, die für die Umsetzung einer serverlosen Architektur verwendet werden können:

- AWS Lambda
- Azure Functions
- Google Cloud Functions

[EA:Web65]

### Vor- und Nachteile

In Tabelle 1.3 werden die wichtigsten Vor- und Nachteile des serverlosen Architekturstils aufgelistet:

| Vorteile | Nachteile |
|---|---|
| + Kosteneinsparungen | - Neue Sicherheitsrisiken |
| + Erhöhte Produktivität | - Mögliche Einschränkungen |
| + Hohe Skalierbarkeit | - Erhöhte Latenz |
| + Integration von Drittanbietersoftware | - Weniger Kontrollmöglichkeiten |

Tabelle 1.3: Vor- und Nachteile der serverlosen Architektur

[EA:Web66]

### Einsatzgebiete

Wie für die zuvor genannten Architekturstile, kann auch die serverlose Architektur in unzähligen Bereichen eingesetzt werden:

- Backend-Applikationen
- Spezielle Aufgaben, die in einem festgelegten Intervall ausgeführt werden
  - Berichte generieren
  - Backup einer Datenbank erstellen
- Webapplikationen mit schwer vorhersehbarer Nachfrage

[EA:Web65]

### 1.3.4 Projektbezug - Architekturstil

Basierend auf der Anforderungsanalyse (siehe Abschnitt 1.2.3) wird eine geeignete Architektur für das Projekt ausgewählt. Die folgenden Punkte fassen die für das Projekt CGM MAXX LITE relevanten Anforderungen zusammen:

- **Funktionale Anforderungen:**
  - Patientensuche und Darstellung der zugehörigen Daten
  - Hinterlegung von Bildern und Audiodateien in der Patientenakte
- **Nicht-funktionale Anforderungen:**
  - Hohe Ausfallsicherheit
  - Skalierbarkeit einzelner Komponenten
  - Erweiterbarkeit für zukünftige Funktionalitäten

Die funktionalen Anforderungen lassen sich dabei in zwei Bereiche unterteilen:

1. Verwaltung von Patientendaten
2. Verwaltung von Dateien

Unter Berücksichtigung der zuvor beschriebenen Architekturstile (siehe Unterkapitel 1.3) eignet sich die **Microservices-Architektur** besonders gut. Zudem ermöglicht sie es, die Software in zwei Subdomänen zu unterteilen. Dadurch lassen sich die Kernfunktionalitäten der Software in zwei unabhängige Services auslagern, was die **Fehlertoleranz** und **Wartbarkeit** erhöht.

Sowohl die ereignisgesteuerte Architektur als auch die Ansätze des serverlosen Computings hätten sich ebenfalls für die Umsetzung von CGM MAXX LITE geeignet. Da der Fokus jedoch auf der Entwicklung der Benutzeroberfläche liegt und der Wunsch des Auftraggebers ist, die Software so zu gestalten, dass anstelle des von uns entwickelten Backends das Backend der Desktop-Applikation CGM MAXX angebunden werden kann, fiel die Entscheidung auf eine einfache Microservices-Architektur.

Zur Implementierung wird das Java-Framework **Spring Boot** in Kombination mit der **Spring Cloud** Library eingesetzt. Diese Entscheidung basiert sowohl auf den umfangreichen Funktionalitäten der Technologie als auch auf den Erfahrungen der Projektteammitglieder.

Im Folgenden wird die genaue Umsetzung von CGM MAXX LITE näher erläutert.

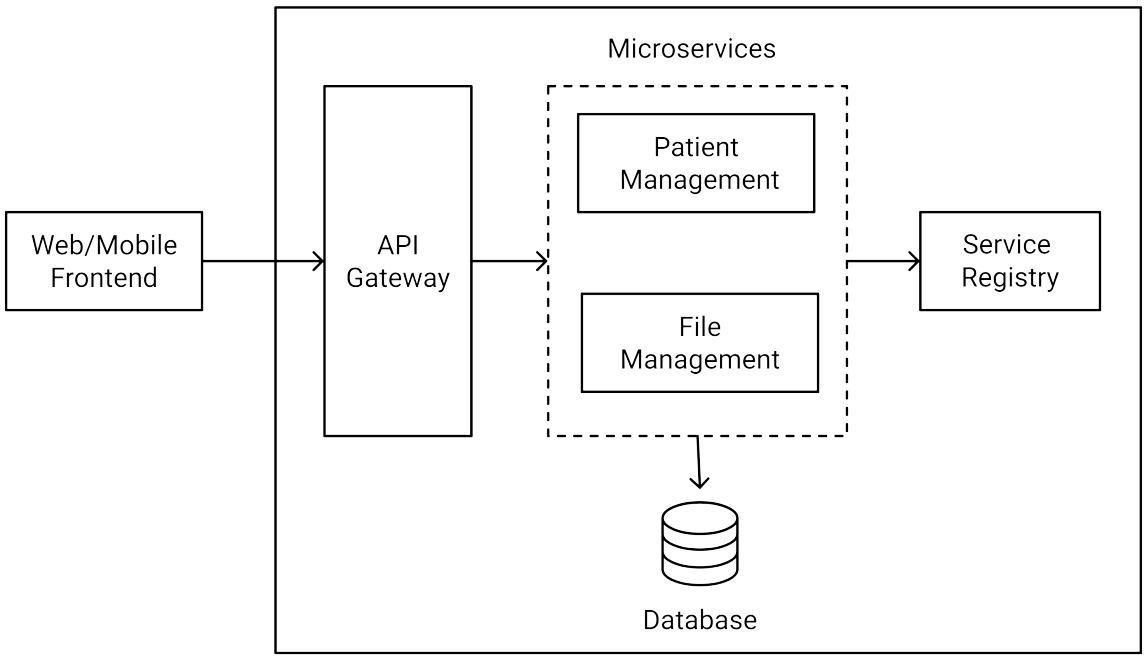


Abbildung 1.6: Microservice Architektur - CGM MAXX LITE

Abbildung 1.6 zeigt den Aufbau von CGM MAXX LITE mithilfe von Spring Boot. Die Software besteht aus folgenden Komponenten:

- **Web/Mobile Frontend**
  - Schnittstelle für Benutzerinteraktionen
  - Kommuniziert mit der API, um Daten zu empfangen und zu senden
- **API-Gateway**
  - Leitet eingehende Requests an die zuständigen Microservices weiter
- **Microservices**
  - Patient Management & File Management
- **PostgreSQL-Datenbank**
  - Speichert alle Patientendaten sowie zugehörige Dateien
- **Netflix Eureka Service Registry**
  - Registriert alle zum System gehörigen Microservices
  - Ermöglicht die Kommunikation zwischen diesen

In der Theorie wird für jedes Microservice eine eigene Datenbank erstellt, um eine höhere Ausfallsicherheit zu gewährleisten. Aufgrund begrenzter Ressourcen, insbesondere in Bezug auf die Hardwareleistung und den zeitlich begrenzten Projektrahmen, wird eine zentrale Datenbank verwendet.

Nun folgt eine genauere Beschreibung des Backends. Nicht nur die Microservices, sondern auch das API-Gateway müssen vom Service Registry registriert werden. Dies ist notwendig, um die vom Frontend kommenden Anfragen korrekt weiterzuleiten und die Kommunikation zwischen den Microservices zu ermöglichen. *Listing 1.1* zeigt die Konfigurationsdatei der Service Registry Anwendung.

```
1 spring.application.name=cml-service-registry
2
3 server.port=8761
4
5 eureka.client.register-with-eureka=false
6 eureka.client.fetch-registry=false
```

Listing 1.1: `application.properties` - Service Registry

Grundsätzlich läuft die Service Registry auf dem Port 8761. Wird dieser Port freigegeben, kann über den Browser ein Dashboard aufgerufen werden, das alle registrierten Services auflistet. Zudem ist sie so konfiguriert, dass sie sich nicht selbst registriert. Bei den Microservices müssen die Werte **register-with-eureka** und **fetch-registry** auf **true** gesetzt werden. *Listing 1.2* zeigt einen Ausschnitt der Konfiguration des File Service:

```
1  # Postgres configuration
2  spring.datasource.url=jdbc:postgresql://cml-db:5432/cml
3  spring.datasource.driverClassName=org.postgresql.Driver
4
5  eureka.client.register-with-eureka=true
6  eureka.client.fetch-registry=true
7  eureka.client.service-url.defaultZone=http://cml-registry:8761/eureka/
8  eureka.instance.prefer-ip-address=true
9
10 spring.cloud.openfeign.client.config.patient-service.url=http://cml-gateway
       :8080/ais/desktop/domain/patient
```

Listing 1.2: `application-docker.properties`

Zusätzlich wird definiert, über welche URL die Service Registry erreicht werden kann. Da der File Service vor dem Hochladen einer Datei überprüfen muss, ob ein Patient mit der angegebenen ID existiert, benötigt dieser Zugriff auf den Patient Service. In Zeile 10 wird die URL des Patient Services definiert. Diese muss mit der im API-Gateway festgelegten URL (siehe *Listing 1.6*) übereinstimmen.

Grundsätzlich ist kein API-Gateway notwendig. Allerdings erleichtert eine zentrale Schnittstelle die Verwaltung, da nur ein einziger Port nach außen hin freigegeben werden muss.

Im Folgenden werden jeweils ein REST-Controller des Patient Services und des File Services gezeigt, um einen Überblick über die verschiedenen Endpunkte zu geben.

```
1  @RestController
2  @RequestMapping("/ais/desktop/domain/patient")
3  public class PatientController implements PatientApi {
4
5      private static final Logger logger = LoggerFactory.getLogger(
           PatientController.class);
6
7      private final PatientService patientService;
8
9      @Autowired
10     public PatientController(PatientService patientService) {
11         this.patientService = patientService;
12     }
13
14     @Override
15     @PostMapping("/search/PatientSearchComponent/findPatients")
16     public List<PatientListEntryDto> findPatients(@RequestBody(required =
           false) PatientSearchRequestBody requestBody) {
17         if (requestBody == null) {
18             logger.error("Patient search request body is null");
19             throw new RequestBodyException("Patient search request body is
                   required");
20         }
21
22         if (requestBody.searchString() == null || requestBody.searchString
               ().isBlank()) {
23             return patientService.findAll();
24         }
25
26         String trimmed = requestBody.searchString().trim();
27
28         if(trimmed.matches("^[0-9]\\d*$|^0$")) {
29             return patientService.findBySocialSecurityNumber(trimmed);
30         } else {
31             return patientService.findByName(trimmed);
32         }
33     }
34
35     @Override
36     @PostMapping("/patient/AisPatientComponent/findById")
37     public PatientDto findPatientById(@RequestBody PatientRequestBody
           requestBody) {
38         if (requestBody == null || requestBody.patientId() == null) {
```

```
39              logger.error("Patient request body is null");
40              throw new RequestBodyException("No patient id provided.");
41          }
42          return patientService.findPatientById(requestBody.patientId());
43      }
44  }
```

Listing 1.3: `PatientController.java`

```
1   @RestController
2   @RequestMapping("/ais/desktop/filemanagement/files")
3   public class FileInfoController implements FileInfoApi {
4
5       private final FileInfoService fileInfoService;
6
7       @Autowired
8       public FileInfoController(FileInfoService fileInfoService) {
9           this.fileInfoService = fileInfoService;
10      }
11
12      @Override
13      @GetMapping("/{fileId}")
14      public FileDto findFileById(@PathVariable Integer fileId) {
15          return this.fileInfoService.findFileById(fileId);
16      }
17
18      @Override
19      @PostMapping
20      public List<FileMetadataDto> findFilesByPatientIdAndType(@RequestBody
            FileSearchRequestBody requestBody) {
21          return this.fileInfoService.findFilesByPatientIdAndType(requestBody
                .patientId(), requestBody.type());
22      }
23
24  }
```

Listing 1.4: `FileInfoController.java`

Damit soll gezeigt werden, dass die Endpunkte verschiedener Anwendungen, die auf unterschiedlichen Ports betrieben werden, mithilfe eines API-Gateways über eine einzige Schnittstelle erreicht werden können.

Eine kurze Zusammenfassung der wichtigsten Endpoints:

- **PatientController** (siehe *Listing 1.3*) - Patient Management (Port 8081)
  - findPatients
  - findById
- **FileInfoController** (siehe *Listing 1.4*) - File Management (Port 8082)
  - findFileById
  - findFilesByPatientIdAndType

Wird nun das Frontend gestartet, so werden verschiedene HTTP-Requests an das API-Gateway gesendet. Werden die Dev-Tools in einem beliebigen Browser geöffnet und der Netzwerk-Tab ausgewählt, so kann überprüft werden, ob die Requests an den richtigen Microservice weitergeleitet werden.

Klickt ein Arzt oder eine Ärztin auf eine bestimmte Person in der Patientenliste, so wird eine Detailansicht mit genaueren Informationen geöffnet. Dabei wird ein HTTP-Request an das Backend gesendet, um weitere Daten zu laden.

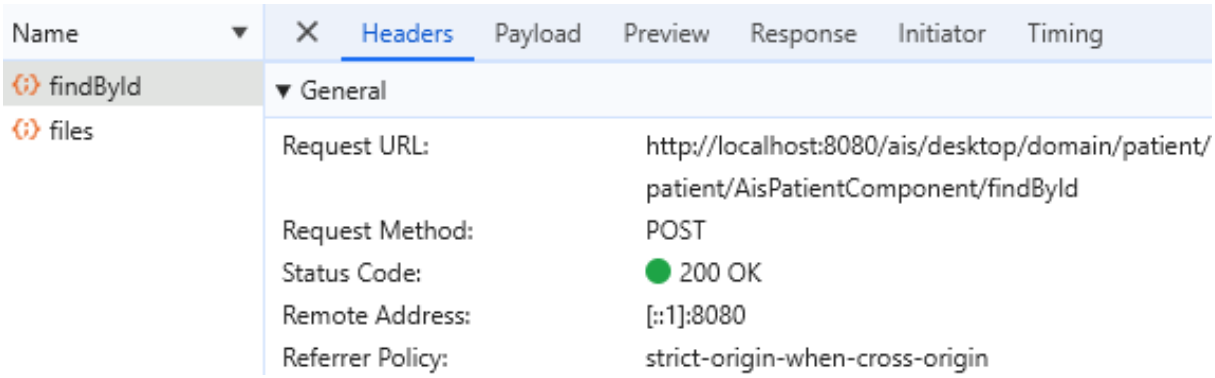


Abbildung 1.7: HTTP-Request an den Patient Service

Abbildung 1.7 zeigt den HTTP-Request an den Patient Service, der korrekt über das API-Gateway und den Port 8080 weitergeleitet wird.

Neben den Stammdaten werden auch Fotos und Audiodateien abgerufen, wobei der Request an den File Service weitergeleitet wird, wie in Abbildung 1.8 dargestellt.

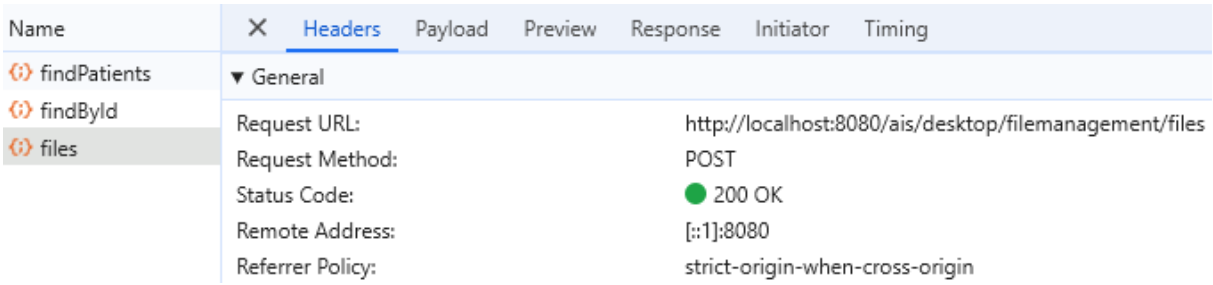


Abbildung 1.8: HTTP-Request an den File Service

Durch diese Architektur kann CGM MAXX LITE flexibel skaliert und durch die Entkopplung in mehrere Komponenten problemlos erweitert werden, was die Weiterentwicklung für die CGM Arztsysteme GmbH vereinfacht. Im nächsten Kapitel wird das Thema Cloud Computing näher untersucht, um eine geeignete Lösung für den Betrieb der Software zu finden und die Unterschiede verschiedener Cloud-Modelle zu verdeutlichen.

## 1.4 Cloud Computing

Cloud Computing ist aus der heutigen Zeit nicht mehr wegzudenken. Täglich entstehen enorme Datenmengen, die gespeichert, verarbeitet und abgerufen werden müssen. Immer mehr Menschen und Unternehmen nutzen die Vorteile des Internets, um unabhängig von Standort und Uhrzeit auf bestimmte Dienste zuzugreifen. Von einfachen Ablagen in der Cloud bis hin zu Social-Media-Plattformen wie Instagram und Facebook wird alles mithilfe von Cloud Computing umgesetzt.

Besonders für Unternehmen ergeben sich dadurch völlig neue Möglichkeiten. Insbesondere die Software-Auslieferung hat sich grundlegend verändert. Software-Releases müssen nicht mehr Monate im Voraus geplant werden, sondern können dank neuer Technologien innerhalb weniger Minuten oder sogar Sekunden erfolgen.
[EA:Web59]

All dies wäre ohne die Cloud undenkbar. Im Folgenden soll der Begriff Cloud Computing näher beschrieben werden.

### 1.4.1 Definition

Die Cloud besteht aus einem großen Netzwerk von Servern, die auf der ganzen Welt verteilt sind und miteinander kommunizieren. Dabei ist es möglich, IT-Ressourcen, die in einem Rechenzentrum betrieben werden, zu mieten (siehe Abbildung 1.9).

Beispiele für solche Ressourcen sind:

- Server
- Speicher
- Datenbanken

Dies bietet den Vorteil, dass kein Geld für die Anschaffung von Hardware investiert werden muss. Bei Cloud Computing gibt es keine allgemein festgelegte Architektur. Es gibt mehrere Cloud-Modelle, die genutzt werden können, um die Wünsche der Kund/innen zu erfüllen. Diese werden in Abschnitt 1.4.2 näher erläutert. [EA:Web55]

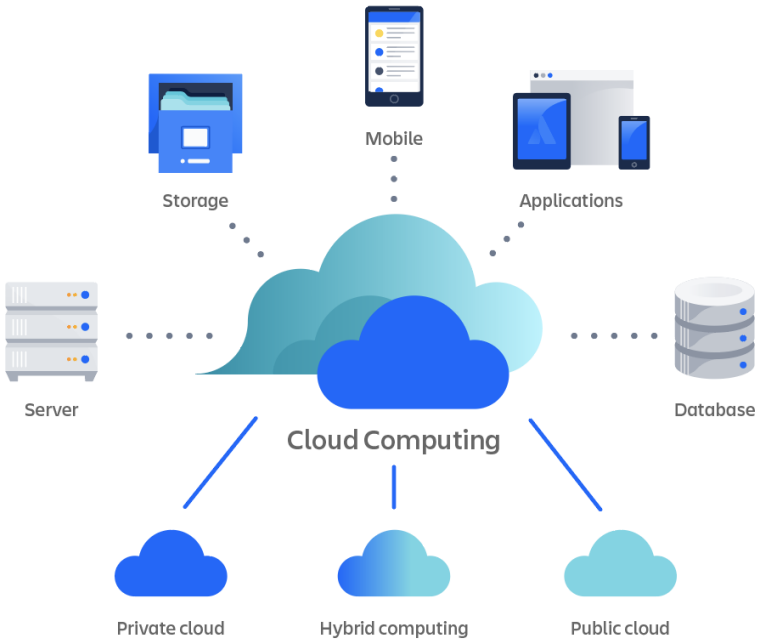


Abbildung 1.9: Cloud Computing
[EA:Web55]

### 1.4.2 Cloud-Modelle

Grundsätzlich kann zwischen drei verschiedenen Arten von Cloud-Computing unterschieden werden:

- Public Cloud
- Private Cloud
- Hybrid Cloud

#### Public Cloud

Bei diesem Cloud-Computing-Modell sind die gesamten Ressourcen im Besitz des Cloud-Serviceanbieters. Diese Cloud-Dienste können je nach Anbieter kostenlos oder kostenpflichtig sein und sind über das Internet erreichbar. Außerdem können die Ressourcen so ausgewählt werden, dass sie auf unterschiedliche Anwendungsfälle zugeschnitten werden können. Der/die Nutzer/in wählt dabei einfach die benötigten Ressourcen über eine Benutzeroberfläche aus und kann somit innerhalb kürzester Zeit

Software ausliefern. Alle zusätzlichen Aufgaben, wie die Bereitstellung und Verwaltung, werden vom Cloud-Anbieter übernommen. Der Einsatz spezieller Werkzeuge kann den Prozess der Systembereitstellung in der Cloud beschleunigen. Einige dieser Werkzeuge werden in Abschnitt 1.5.2 näher beschrieben. [EA:Web53, EA:Web54]

Zu den bekanntesten Anbietern gehören:

- Amazon Web Services (AWS)
- Google Cloud
- Microsoft Azure

#### Vor- und Nachteile der Public Cloud

Zudem gibt es weitere Vorteile, die durch die Verwendung einer Public Cloud geschaffen werden können. Beispielsweise können die **Infrastrukturkosten gesenkt** werden, da keine Betriebskosten für einen lokal installierten Server mehr anfallen. Stattdessen wird nur noch für die tatsächlich genutzten Ressourcen in der Cloud bezahlt. Durch die flexible Verwaltung von Ressourcen ist auch die **Skalierung** unproblematisch.

Allerdings hat auch die Public Cloud ihre Schattenseiten, die berücksichtigt werden müssen. Speziell dann, wenn es um den Umgang mit sensiblen Daten geht, ist besondere Vorsicht geboten, da eine **unzureichende Konfiguration** zu **Sicherheitslücken** führen kann. Darüber hinaus muss sichergestellt werden, dass **stets eine stabile Internetverbindung besteht**, damit die in der Cloud bereitgestellten Dienste auch jederzeit erreicht werden können. [EA:Web53]

#### Einsatzgebiete der Public Cloud

Durch die zahlreichen Möglichkeiten, die Public Clouds bieten, können diese für extrem unterschiedliche Anwendungsfälle eingesetzt werden. Allgemein gesagt, eignet sich die Public Cloud besonders für Softwarelösungen, bei denen die genaue Auslastung vorab schwer abgeschätzt werden kann und/oder sich die Anforderungen im Laufe des Projekts häufig verändern. [EA:Web53]

### Private Cloud

Da die Public Cloud trotz ihrer zahlreichen Vorteile nicht für jeden Anwendungsfall geeignet ist, gibt es weitere Arten von Cloud Computing, die in Betracht gezogen werden können. Eine davon ist die **Private Cloud**.

Der wesentliche Unterschied zur Public Cloud ist, dass die **Cloud-Computing-Ressourcen nicht mit anderen Personen geteilt** werden, sondern ausschließlich für Personen mit entsprechenden Zugriffsrechten zugänglich sind.

Dies wird in Abbildung 1.10 veranschaulicht:

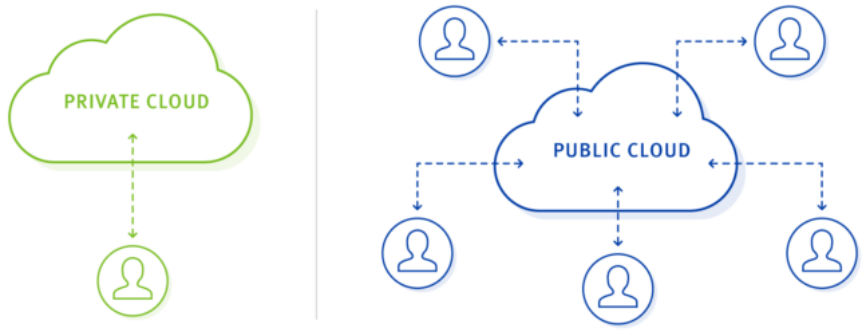


Abbildung 1.10: Private Cloud vs. Public Cloud
[EA:Img01]

Zudem kann die Private Cloud weiter spezifiziert werden:

- **Lokale Private Cloud**
  - IT-Infrastruktur befindet sich am Standort des Kunden
  - Bereitstellung & Verwaltung erfolgt intern
- **Cloud-Dienstanbieter**
  - Rechenressourcen werden in einem externen Rechenzentrum betrieben
  - Ein Drittanbieter übernimmt die Bereitstellung und Verwaltung

Verfügt ein Unternehmen über qualifiziertes Personal, das sich um die lokal installierte IT-Infrastruktur kümmert, kann auf externe Anbieter verzichtet werden. Ist dieses Kriterium nicht gegeben, so ist es wirtschaftlich sinnvoller, diese Aufgaben an einen

Cloud-Dienstanbieter auszulagern, um die Anforderungen der Kundschaft effizient zu erfüllen. [EA:Web54]

### Vor- und Nachteile der Private Cloud

Dadurch, dass die Private Cloud eine isolierte Umgebung ist, auf die fremde Personen und Unternehmen keinen Zugriff haben, bietet diese Art von Cloud Computing einige Vorteile in Bezug auf **Sicherheit und Kontrolle**. Die Ressourcen können so angepasst werden, dass sie den Anforderungen der Auftraggebenden entsprechen.

Wie die Public Cloud hat auch die Private Cloud ihre Nachteile. Um die möglichen Probleme der Private Cloud aufzuzeigen, wirft man einen Blick auf das Projekt CGM MAXX LITE. Wie in Abschnitt 1.2.3 beschrieben, handelt es sich bei diesem Projekt um die Entwicklung einer Arztsoftware. Da diese Anwendung mit sehr sensiblen Daten arbeitet, die aus datenschutzrechtlichen Gründen nicht von unbefugten Personen eingesehen werden dürfen, ist die Private Cloud eine gute Lösung.

Sollte ein Arzt die Software erwerben und On-Premise[7] betreiben wollen, dann muss dafür eine **entsprechende Infrastruktur angeschafft und konfiguriert** werden, was erstens sehr **kostspielig** sein kann und **zusätzlichen Wartungsaufwand** erfordert. Zudem ist die Infrastruktur weniger flexibel als bei einem Drittanbieter, was im schlimmsten Fall zu Engpässen führen kann. [EA:Web56]

### Einsatzgebiete der Private Cloud

Wie aus dem obigen Absatz hervorgeht, eignet sich der Einsatz einer privat betriebenen Cloud besonders dann, wenn in etwa abgeschätzt werden kann, wie viele Ressourcen benötigt werden, um die Hardware gezielt auszuwählen und einen reibungslosen Betrieb des Softwaresystems zu gewährleisten. Auch für Anwendungen, bei denen ein hohes Maß an Sicherheit erforderlich ist, eignet sich dieses Cloud-Computing-Modell hervorragend.

Weitere Beispiele wären Anwendungen in folgenden Bereichen:

- Finanzen
- Recht
- Politik

[EA:Web56]

[7] Software vor Ort betreiben

### Hybrid Cloud

Die letzte Art des Cloud Computings, die in dieser Arbeit beschrieben wird, ist die **Hybrid Cloud**. Im Vergleich zu den zuvor erläuterten Cloud-Modellen kommen hier mehrere unterschiedliche Clouds zum Einsatz. Es handelt sich also um eine „Mischform“ bestehend aus Public und Private Cloud (siehe Abbildung 1.11).

Die Clouds werden dabei so miteinander verbunden, dass Daten ausgetauscht werden können. Werden mehrere Clouds derselben Art (z.B. zwei öffentliche Clouds) eingesetzt, so spricht man von einer **Multi Cloud**.

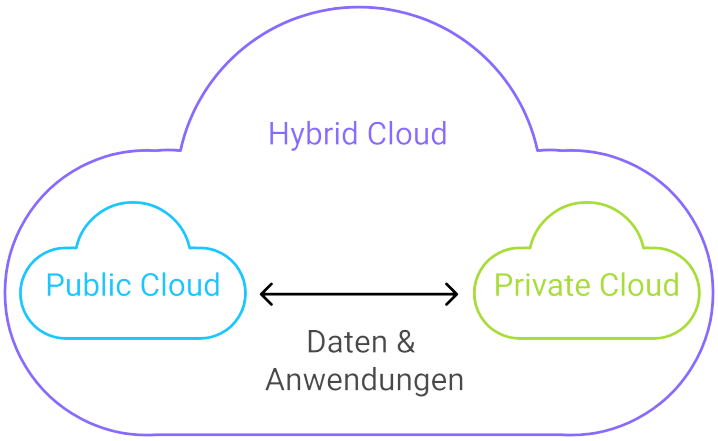


Abbildung 1.11: Hybrid Cloud

Speziell bei Anwendungen, bei denen ein großer Wert auf Datenschutz gelegt wird und worauf viele verschiedene Personen zugreifen können, ist es wichtig, die Daten sicher zu verwalten. In solchen Fällen ist es meist nicht mehr ausreichend, auf ein einziges Cloud-Modell zu vertrauen, weswegen immer häufiger auf hybride Cloud-Lösungen gesetzt wird. [EA:Web57]

Um auf das zuvor genannte Beispiel zurückzukommen, das erklärt, wofür sich eine private Cloud besonders eignet, folgt nun eine kurze Ergänzung in Bezug auf die hybride Cloud:

Arbeitet eine Applikation mit sensiblen Daten, so ist es sinnvoll, diese lokal oder in einer privat verwalteten Cloud abzulegen, während öffentlich zugängliche Dienste in der Public Cloud bereitgestellt werden sollten, um einen schnellen und einfachen Zugriff zu gewährleisten.

### Vor- und Nachteile der Hybrid Cloud

Eine gezielte Kombination verschiedener Cloud-Umgebungen ermöglicht eine effizientere Ressourcennutzung, steigert die Flexibilität und kann Sicherheitsanforderungen besser erfüllen. Je nachdem, was betrieben werden soll, eignet sich eine Lösung besser als die andere.

Trotz der Vorteile beider Cloud-Modelle bringt die Hybrid Cloud auch neue Herausforderungen mit sich. Ein Nachteil ist beispielsweise ein erhöhter Aufwand in Bezug auf Verwaltung und Konfiguration. Grund dafür sind die vielen beteiligten Komponenten, die miteinander interagieren und im Auge behalten werden müssen. [EA:Web58]

## 1.5 Architekturentwurf

Wie in Unterkapitel 1.1 bereits ausführlich beschrieben wurde, zählt die effiziente Umsetzung funktionaler und nicht-funktionaler Anforderungen des Auftraggebers zu den wichtigsten Zielen der Softwarearchitektur. Diese Anforderungen werden in Unterkapitel 1.2 näher erläutert. Basierend auf den Ergebnissen der Anforderungsanalyse wird ein Architekturstil ausgewählt, wobei eine Auswahl moderner Architekturstile in Unterkapitel 1.3 näher beschrieben wird. Darüber hinaus sind weitere Entscheidungen zu treffen, wie die Wahl der Technologien, die Unterteilung des Systems in kleinere, verwaltbare Teile und andere relevante Aspekte.

Dieser Prozess wird in regelmäßigen Abständen wiederholt, um anhand von Feedback und neuen Erkenntnissen, die während der Entwicklung gewonnen werden, Anpassungen vorzunehmen und die Architektur der Software kontinuierlich zu verbessern.

Bevor mit der Implementierung der Software begonnen wird, ist es essenziell, dass ein erster Entwurf der Architektur vorliegt. Eine gute Planung bietet den Projektteammitglieder nicht nur eine klare Orientierungshilfe, sondern hilft auch, potenzielle Herausforderungen frühzeitig zu erkennen. [EA:Web15, EA:Web16]

### 1.5.1 Architekturmodellierung

Wie in Abschnitt 1.1.4 beschrieben, muss der/die Softwarearchitekt/in in der Lage sein, das Projektteam in eine bestimmte Richtung zu lenken. Dabei ist es wichtig, dass alle Teammitglieder sowohl die funktionalen als auch die nicht-funktionalen Anforderungen des Projekts kennen und zumindest einen groben Überblick über die zugrundeliegende Softwarearchitektur haben.

Da die Architektur eines Softwaresystems sehr komplex und umfangreich sein kann, können **Visualisierungstechniken** eingesetzt werden, um die **Komplexität zu reduzieren**. Dies bezeichnet man als Architekturmodellierung, bei der die wesentlichen Eigenschaften des Systems identifiziert und dargestellt werden, um das System besser zu verstehen.

Das Ziel ist es, einen klaren Überblick zu schaffen und fundierte Entscheidungen anhand dieser Darstellungen zu treffen. Nicht nur das Projektteam profitiert davon, sondern auch neue Teammitglieder, die so schneller ins Projekt integriert werden können.

Es gibt zahlreiche Tools und Standards, die für die Visualisierung der Softwarearchitektur verwendet werden können. Im Folgenden werden zwei der bekanntesten Modellierungstechniken näher beschrieben. [EA:Web12, EA:Web13, EA:Web14]

### Unified Modeling Language (UML)

Die Unified Modeling Language (UML) ist eine standardisierte Modellierungssprache, die verwendet wird, um bestimmte Teile eines Softwaresystems grafisch darzustellen.

Durch den Einsatz von UML können Diagramme erstellt werden, die als Kommunikationsmittel zwischen unterschiedlichen Stakeholdern genutzt werden können.
Diese Diagramme bieten den Vorteil, dass sie auch von Personen verstanden werden können, die keine Programmierkenntnisse besitzen und komplexe Systeme einfach darstellen. [EA:Web18, EA:Web19]

Es gibt 14 offiziell anerkannte Arten von UML-Diagrammen, die in **zwei Kategorien** unterteilt werden:

- Strukturelle Diagramme
- Verhaltensdiagramme

In Abbildung 1.12 sind die verschiedenen UML-Diagrammtypen dargestellt:

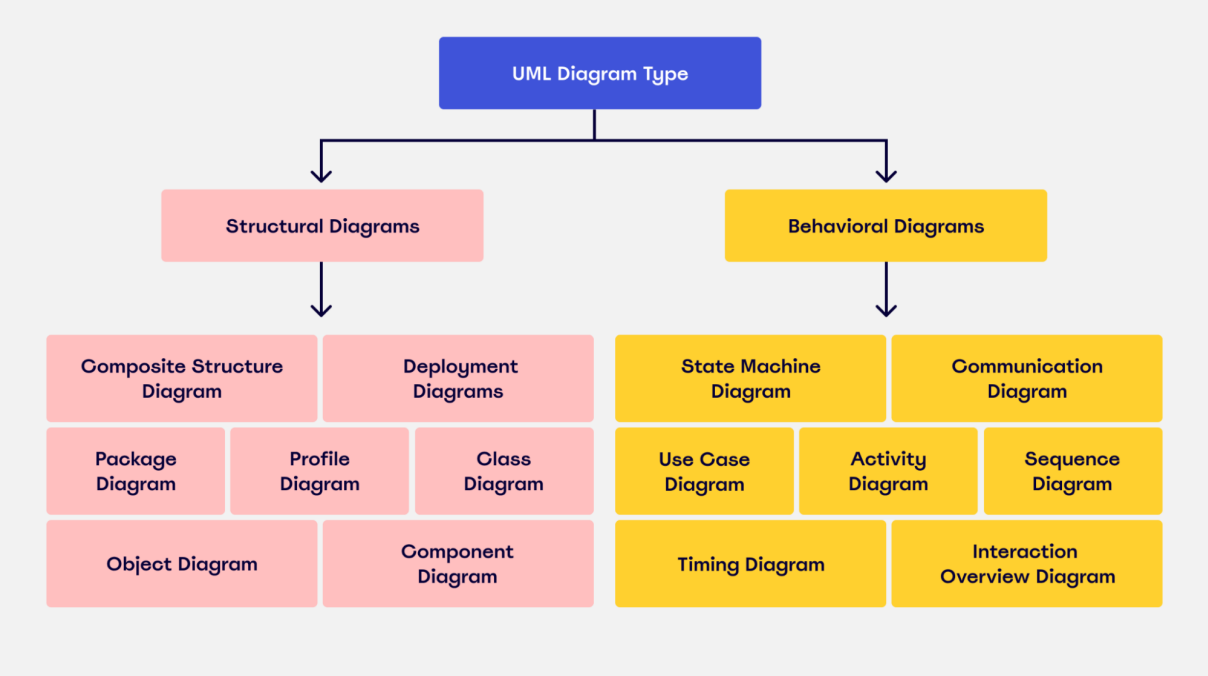


Abbildung 1.12: Arten von UML-Diagrammen
[EA:Web17]

Da es den Rahmen dieser Arbeit sprengen würde, jeden einzelnen UML-Diagrammtyp im Detail zu erklären, wird die allgemeine Unterscheidung zwischen Struktur- und Verhaltensdiagrammen betrachtet. Anschließend wird jeweils ein Struktur- und ein Verhaltensdiagramm näher beschrieben, die in der Praxis besonders häufig Anwendung finden und in Bezug auf das Projekt **CGM MAXX LITE** von Bedeutung sind.

**Strukturelle Diagramme** veranschaulichen den **Aufbau eines Softwaresystems** oder einzelner Teile davon. Dabei werden Komponenten wie Klassen, Objekte und Module sowie deren Beziehungen dargestellt, die für die Struktur der Software entscheidend sind.

Das bekannteste Strukturdiagramm ist das **Klassendiagramm**, wobei das System, oder nur ein Teil davon, durch Klassen abgebildet wird. Jede Klasse hat dabei bestimmte Attribute (Daten) und Methoden (Funktionen), die für unterschiedliche Aufgaben genutzt werden können. [EA:Web17, EA:Web18]

Ein Beispiel für ein Klassendiagramm ist in Abbildung 1.13 zu finden. Es zeigt die Klassen, die im Rahmen des Projekts CGM MAXX LITE für den File-Upload erforderlich sind.

Hierbei handelt es sich um die **Klassen einer Spring-Boot-Applikation**. Da diese Anwendung neben dem Hochladen von Dateien auch noch weitere Funktionalitäten bietet, konzentriert sich dieses Diagramm **nur** auf die **Klassen, die für das Hochladen von Dateien relevant sind**. Dadurch wird ein besserer Überblick gewährleistet und es ist einfacher das Diagramm zu verstehen, ohne sich von irrelevanten Klassen oder Methoden ablenken zu lassen.

Hier eine kurze Erklärung der wichtigsten Klassen:

- **`FileUploadController`** ... RESTful-Controller, der die Schnittstellen für die Kommunikation zwischen Frontend und Backend bereitstellt.
- **`FileUploadServiceImpl`** ... Service, der die gesamte Business-Logik umfasst. Dazu gehört nicht nur das Speichern von Fotos und Audioaufnahmen, sondern auch die richtige Benennung dieser Dateien.
- **`FileInfoService`** ... wird verwendet, um die Anzahl der Dateien zu ermitteln, die an einem bestimmten Tag für einen bestimmten Patienten hochgeladen wurden. Dies ist notwendig für eine einheitliche Benennung der Dateien.
- **`PatientService`** ... überprüft, ob der/die Patient/in, für den eine Datei hochgeladen werden soll, existiert.
- **`FileRepository`** ... verwaltet **`File`**-Entitäten (CRUD[8]-Operationen)

[8] Abkürzung für Create, Read, Update und Delete

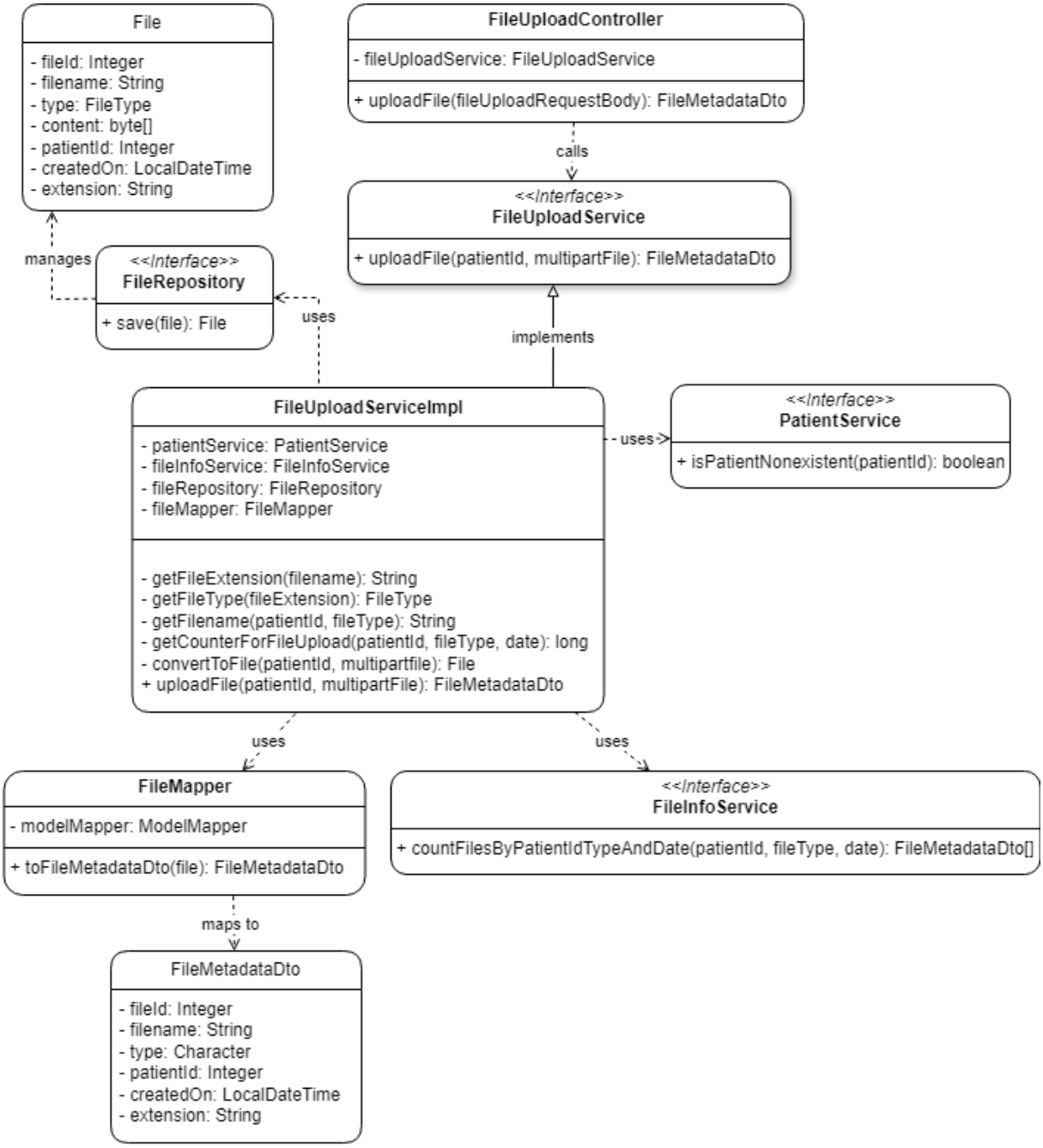


Abbildung 1.13: Klassendiagramm - File Upload im Projekt CGM MAXX LITE

Neben den Klassen in Abbildung 1.13 sind auch die Beziehungen zwischen diesen durch unterschiedliche Pfeile gekennzeichnet, die durch das Hinzufügen von Beschriftungen noch leichter nachvollzogen werden können.

Im oben abgebildeten Klassendiagramm kommen nur zwei Arten von Beziehungen zum Einsatz:

- Vererbung (Inheritance)
- Abhängigkeit (Dependency)

Dabei implementiert `FileUploadServiceImpl` das Interface `FileUploadSerive`. Die restlichen Beziehungen zeigen Abhängigkeiten, bei denen bestimmte Klassen mithilfe von **Dependency Injection**[9] eingebunden werden, um bestimmte Methoden aufzurufen. [EA:Web21]

Auf den ersten Blick wirkt das Diagramm möglicherweise etwas komplex, betrachtet man es jedoch genauer, erkannt man, dass es **nach außen hin nur eine Schnittstelle** gibt, auf die zugegriffen werden kann. Im Folgenden wird gezeigt, wie auf die Schnittstelle zugegriffen werden kann.

```java
@RestController
@RequestMapping("/ais/desktop/domain/files/files/FileUploadComponent")
public class FileUploadController implements FileUploadApi {

    private final FileUploadService fileUploadService;

    @Autowired
    public FileUploadController(FileUploadService fileUploadService) {
        this.fileUploadService = fileUploadService;
    }

    @Override
    @PostMapping("/saveUpload")
    public FileMetadataDto uploadFile(@ModelAttribute FileUploadRequestBody
         requestBody) {
        return this.fileUploadService.uploadFile(requestBody.patientId(),
            requestBody.file());
    }

}
```

Listing 1.5: `FileUploadController.java`

Der in *Listing 1.5* definierte REST-Controller ermöglicht das Hochladen einer Datei für eine/n Patient/in über einen **POST-Request**. Dabei wird der `FileUploadService` aufgerufen, um die Datei korrekt zu benennen und anschließend in der Datenbank zu speichern.

[9] Technik, bei der Objekte ihre Abhängigkeiten von anderen Objekten erhalten [EA:Web22]

Neben strukturellen Diagrammen gibt es auch **Verhaltensdiagramme**. Diese stellen das dynamische Verhalten eines Systems dar. Dabei spielt nicht nur das Geschehen innerhalb des Systems eine Rolle, sondern auch wie das System mit anderen Systemen und Benutzern interagiert. Verhaltensdiagramme eignen sich deshalb besonders gut, um **Abläufe und Geschäftsprozesse** zu **visualisieren**. [EA:Web18, EA:Web20]

Um ein Diagramm zu erstellen, das nicht nur die Struktur der Klassen darstellt, sondern auch erklärt, in welcher Reihenfolge bestimmte Interaktionen erfolgen, kann ein sogenanntes **Sequenzdiagramm** verwendet werden. Dies ermöglicht es, den genauen Ablauf eines bestimmten Prozesses nachzuvollziehen. [EA:Web23, EA:Web24]

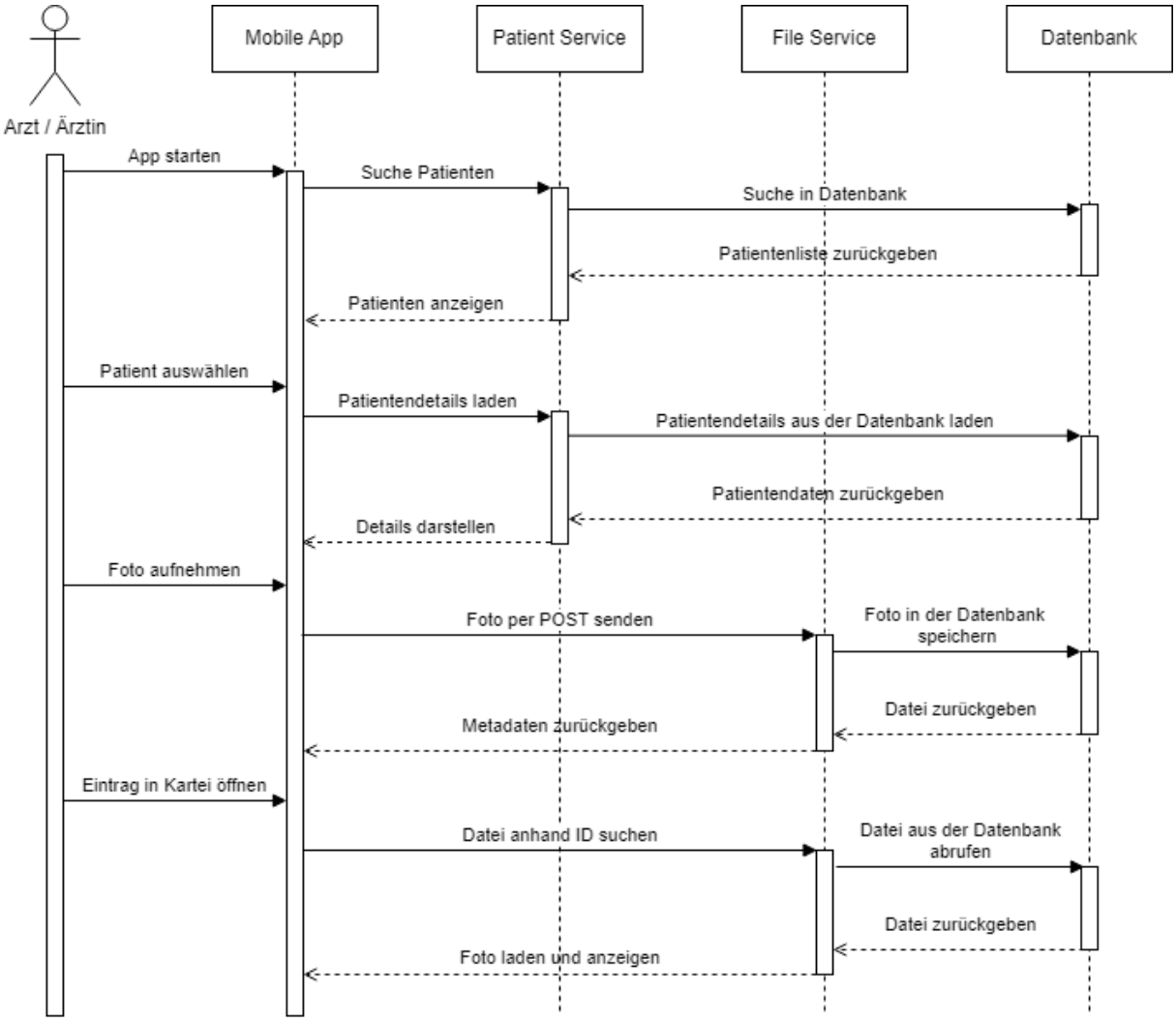


Abbildung 1.14: Sequenzdiagramm - File Upload im Projekt CGM MAXX LITE

Um den Unterschied zwischen strukturellen Diagrammen und Verhaltensdiagrammen zu verdeutlichen, stellt das Sequenzdiagramm in Abbildung 1.14 ebenfalls den File-Upload dar. Während das Klassendiagramm in Abbildung 1.13 die Struktur der Klassen beschreibt und einen Bezug zum Code herstellt, zeigt das Sequenzdiagramm die notwendigen Schritte, um eine Datei in der Datenbank abzulegen und später in der App abzurufen.

Folgende Elemente werden im Sequenzdiagramm 1.14 verwendet:

- Akteur/innen
- Objekte
- Lebenslinien
- Aktivitätsbalken
- Synchrone Nachrichten (durchgezogene Linie mit ausgefüllter Spitze)
- Asynchrone Antworten (strichlierte Linie mit offener Spitze)

Im Folgenden wird der Ablauf des Sequenzdiagramms näher beschrieben, um den Einsatz und die Bedeutung der einzelnen Elemente zu erklären. [EA:Web24]

Da die Software CGM MAXX LITE hauptsächlich von **Ärzten/innen** verwendet wird, gibt es hier **nur einen Akteur/eine Akteurin**. Akteur/innen sind also Entitäten, die mit dem System interagieren. Die **Objekte** stellen **einzelne Teile des Systems** dar. Dieses besteht aus einer mobilen Benutzeroberfläche, einer zentralen Datenbank und zwei Microservices. Der eine Microservice ist verantwortlich für die Patientenverwaltung, das andere kümmert sich um die Verwaltung von Fotos und Audioaufnahmen. Die **strichlierten Linien unterhalb der Objekte** stellen deren **Lebenslinien** dar. Die **darüberliegenden Balken** bezeichnet man als **Aktivitätsbalken**. Diese stellen dar, wann auf bestimmte Objekte zugegriffen wird. Was genau passiert, wird durch **verschiedene Pfeile und kurze Beschreibungen** dargestellt. Die Akteur/innen können dabei nur auf die mobile Applikation zugreifen, die im Hintergrund auf die einzelnen Services zugreift.

Möchte ein Arzt oder eine Ärztin ein Foto hochladen, werden zunächst die Patient/innen aus der Datenbank geladen. Anschließend kann der/die gewünschte Patient/in ausgewählt werden. Dabei wird die Detailansicht geöffnet und es kann ein Foto aufgenommen werden. Ein POST-Request wird an den File Service gesendet, der das Foto verarbeitet, benennt es korrekt, in der Datenbank speichert und anschließend die Metadaten der hochgeladenen Datei zurückgibt, um zu bestätigen, dass der Upload erfolgreich war.

### C4-Modell

Neben UML gibt es zahlreiche weitere Methoden, um die Architektur einer Software zu modellieren. Eine davon ist das **C4-Modell**, dessen Name für **Context, Containers, Components und Code** steht. Ähnlich wie UML zielt auch das C4-Modell darauf ab, das Verständnis unterschiedlicher Interessensgruppen zu verbessern, um eine Grundlage für erfolgreiche Projektergebnisse zu schaffen. Dabei wird das **System in vier Ebenen dargestellt**. [EA:Web25, EA:Web26, EA:Web28]

Als nächstes werden die einzelnen Ebenen näher beschrieben:

Die **erste Ebene** bezieht sich auf den **Systemkontext**. Bei einem Systemkontext-Diagramm liegt der Fokus darauf, einen **groben Überblick über das System** zu schaffen, ohne näher auf spezifische Technologien und technische Details einzugehen. Genauer gesagt beschreibt es, wofür das System zuständig ist, von wem es verwendet wird und wie es mit anderen Systemen interagiert. Solche Diagramme eignen sich besonders gut, um mit nicht-technischen Stakeholdern zu kommunizieren. [EA:Web27]

In Abbildung 1.15 wird das Systemkontext-Diagramm für das Projekt CGM MAXX LITE dargestellt:

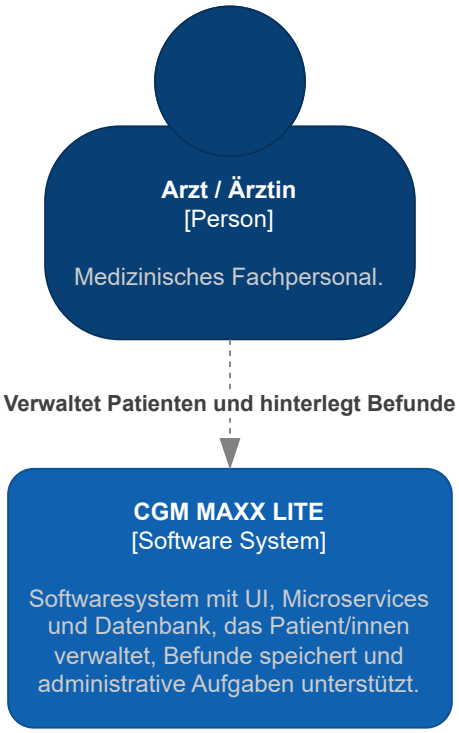


Abbildung 1.15: Systemkontext-Diagramm - CGM MAXX LITE

Die **zweite Ebene** stellt die **einzelnen Container eines Systems** dar. Container stellen die grundlegenden Bestandteile eines Systems dar, die unabhängig voneinander gestartet, bereitgestellt und betrieben werden können. **Container-Diagramme** zeigen zusätzlich auch die verwendeten Technologien und wie die Container miteinander kommunizieren. Sie können eingesetzt werden, um **Softwareentwickler/innen bei der Entwicklung zu unterstützen**. Das bietet den Vorteil, dass Abhängigkeiten zwischen Container einfacher nachvollzogen werden können. [EA:Web28]

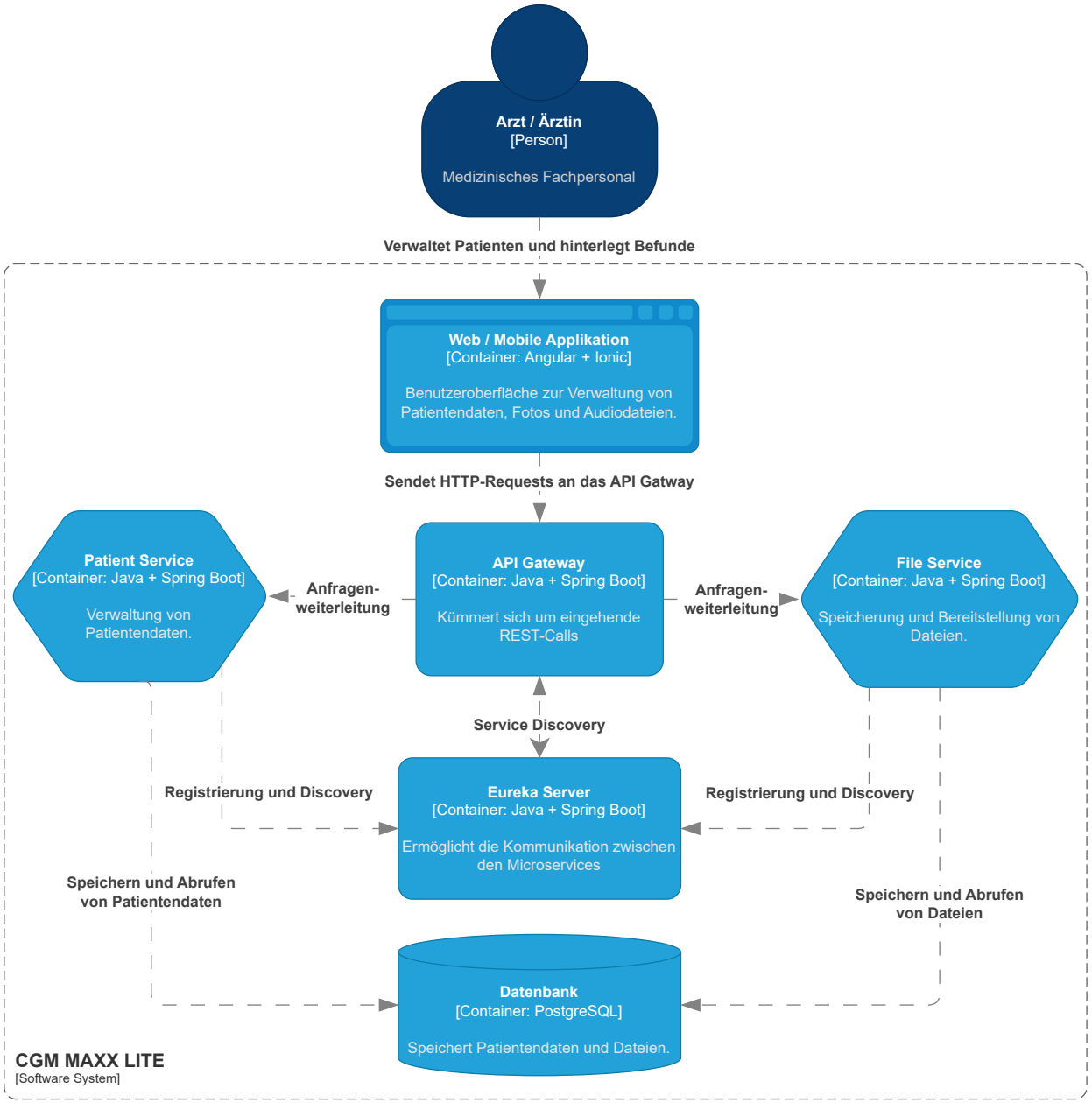


Abbildung 1.16: Container-Diagramm - CGM MAXX LITE

In Abbildung 1.16 wird ein Container-Diagramm dargestellt, das die wichtigsten Bestandteile des Projekts CGM MAXX LITE zeigt.

Dabei gibt es unterschiedliche Arten von Containern:

- Web-App / Mobile App
- API-Gateway
- Microservices
- Service Discovery Server
- Datenbank

Eine genaue Beschreibung der einzelnen Container ist in Abschnitt 1.3.4 zu finden.

Da es zu umfangreich wäre, für jede Ebene des C4-Modells ein spezifisches Diagramm für CGM MAXX LITE zu erstellen, und es zudem schwierig ist, alle Komponenten auf einer Seite darzustellen, werden die folgenden Ebenen ausschließlich textlich beschrieben.

Die **dritte Ebene** ermöglicht einen genaueren Blick auf die **Elemente innerhalb eines Containers**. Mithilfe eines **Komponentendiagramms** kann dargestellt werden, welche Aufgaben die einzelnen Komponenten erledigen, wie sie miteinander interagieren und welche Technologien oder Implementierungen dabei zum Einsatz kommen. [EA:Web27, EA:Web28]

Ein Beispiel für eine Komponente ist ein REST-Controller in Spring Boot. Bezieht man sich dabei auf das Projekt, so wäre die Java-Klasse **FileUploadController.java** in *Listing 1.5* eine Komponente.

Die **vierte** und letzte **Ebene** des C4-Modells betrachtet die Implementierung einzelner Komponenten. Dazu können **UML-Klassendiagramme** eingesetzt werden. Auch hierfür gibt es bereits ein Beispiel in Abbildung 1.13 Solche Diagramme müssen oft nicht selbst erstellt werden, da sie mithilfe von IDEs[10] automatisch erstellt werden können. [EA:Web28]

[10]Integrierte Entwicklungsumgebung

#### Überleitung

Bevor mit dem nächsten Thema fortgesetzt wird, hier eine kurze Zusammenfassung der wichtigsten Erkenntnisse:

Es gibt unzählige verschiedene Möglichkeiten, um die Architektur einer Software grafisch darzustellen. Diese Visualisierungen können als Kommunikationsmittel eingesetzt werden und helfen dabei, ein besseres Verständnis für das System zu schaffen, unabhängig davon, ob eine Person technische Fähigkeiten besitzt oder nicht. Basierend auf der Zielgruppe können unterschiedliche Diagramme verwendet werden. Je nach Anwendungszweck eigenen sich bestimmte Diagramme besser als andere.

### 1.5.2 Cloud-spezifische Architekturentscheidungen

Da der Entwurf einer Software weit mehr als nur die Modellierung betrifft, gibt es einige weitere Entscheidungen, die getroffen werden müssen. Wie der Name dieser Arbeit bereits verrät, liegt der Fokus auf cloudbasierten Systemen. Neben Planung und Entwicklung muss die Software auch in der Cloud bereitgestellt werden, um sie für die Nutzer/innen zugänglich zu machen.

Damit Anwendungen sicher, zuverlässig und skalierbar in der Cloud bereitgestellt werden können, gibt es **Entwurfsmuster**, die eingesetzt werden, um bestimmte Probleme der Cloud zu umgehen. Um Software schneller bereitzustellen, kann die Infrastruktur mithilfe von **Infrastructure as Code** automatisch erstellt werden. **DevOps und Automatisierung** spielen ebenfalls eine bedeutende Rolle, um Anwendungen einfach bereitzustellen. [EA:Web30, EA:Web31]

Im weiteren Verlauf wird näher auf diese Themen eingegangen.

#### Entwurfsmuster für die Cloud

Wie zuvor erwähnt werden Entwurfsmuster eingesetzt, um **Software zuverlässig in der Cloud zu betreiben**. Vor allem bei verteilten Systemen, bei denen mehrere Komponenten über das Netzwerk miteinander kommunizieren, kann es zu Schwierigkeiten kommen. Beispiele dafür sind unzuverlässige Netzwerke, hohe Latenzen oder begrenzte Bandbreiten.

Durch den gezielten Einsatz von Entwurfsmustern werden spezifische Probleme zwar nicht beseitigt, aber die Wahrscheinlichkeit, dass diese eintreten, wird dadurch deutlich verringert. Microsoft stellt eine Sammlung von Entwurfsmustern bereit, die auf unterschiedliche Herausforderungen zugeschnitten sind. Im Anschluss wird eine Auswahl von Entwurfsmustern beschrieben, die für das Projekt CGM MAXX LITE relevant sind. [EA:Web31]

Ein Entwurfsmuster, das für CGM MAXX LITE von großer Bedeutung ist, heißt **Backends for Frontends**. Bei diesem Muster wird **für jede Benutzeroberfläche ein eigenes Backend** erstellt.

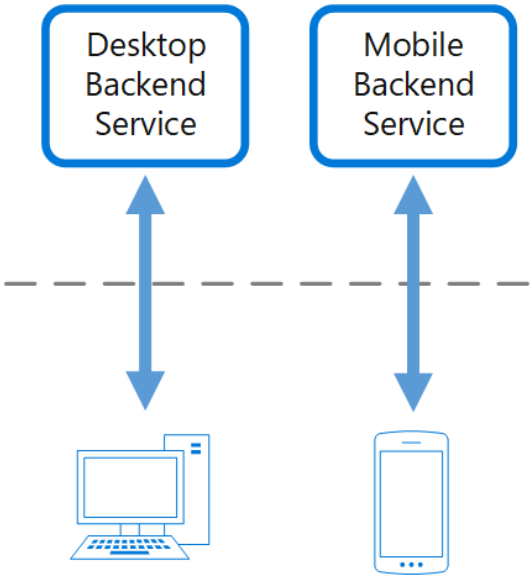


Abbildung 1.17: Backends for Frontends [EA:Web32]

Wie in Abschnitt 1.2.3 erwähnt, existiert bereits ein Backend für die Desktop-Applikation CGM MAXX. Da diese nicht für den Betrieb auf mobilen Geräten ausgelegt ist, wird im Zuge des Projekts ein Frontend entwickelt, das sich auf mobile Geräte fokussiert. Da die mobile Applikation nur einen Teil der Aufgaben der Desktop-Applikation abdeckt und zusätzlich Funktionen für Kamera und Mikrofon bietet, ist es sinnvoll, ein separates Backend zu entwickeln. So gibt es kein zentrales Backend, das die Anforderungen beider Frontends erfüllen muss, wodurch es zu keinen Konflikten kommen kann. Das bedeutet, dass die Applikationen unabhängig voneinander verändert werden können. [EA:Web32]

Ein weiteres interessantes Muster ist das **Compute Resource Consolidation Pattern**. Softwaresysteme bestehen oftmals aus mehreren Applikationen, die unabhängig voneinander bereitgestellt werden. Dies führt zu einem höheren Bedarf an Computerressourcen, steigenden Bereitstellungskosten und einem größeren Wartungsaufwand.

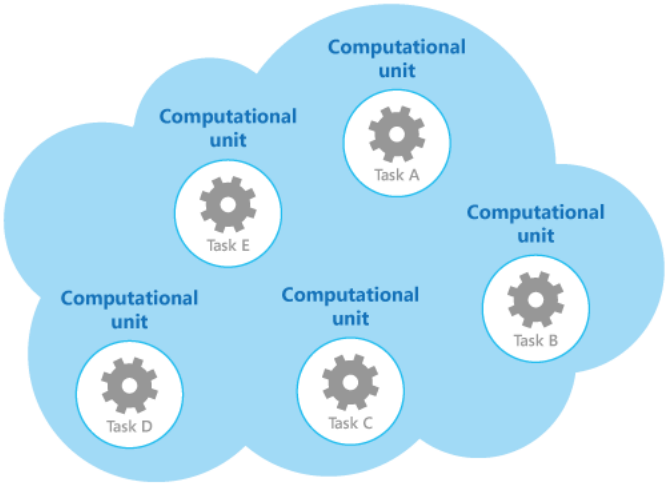


Abbildung 1.18: Vereinfachte Struktur einer cloudbasierten Lösung [EA:Web33]

Abbildung 1.18 zeigt vereinfacht, wie ein verteiltes System in der Cloud aussieht. Eine Recheneinheit kann beispielsweise genutzt werden, um ein Microservice zu betreiben.

In Systemen mit vielen Microservices entstehen jedoch häufig unnötige Kosten, da auf bestimmte Services öfter zugegriffen wird als auf andere. Selbst dann, wenn ein Service gar nicht verwendet wird, entstehen Kosten. Daher ist es wichtig, die Aufteilung der Dienste sorgfältig zu planen, um eine kostengünstige und effiziente Lösung zu schaffen.

Dieses Muster zielt darauf ab, die **Auslastung der Recheneinheiten** zu **erhöhen**, um die **Kosten für die Ressourcen** zu **senken**. Erreicht wird dies, indem bestimmte **Dienste zusammengefasst** und **auf einer gemeinsamen Recheneinheit betrieben** werden. Außerdem können die Dienste so direkt miteinander kommunizieren, was dazu beiträgt, dass bestimmte Aktionen schneller ausgeführt werden können.

**Eingesetzt** wird dieses Muster **bei Diensten, die die meiste Zeit im Leerlauf verbringen**, um durch das Gruppieren zusammenhängender Einheiten die Kosten zu senken. Weniger geeignet ist für fehlertolerante Vorgänge, da andere Dienste, die auf derselben Recheneinheit betrieben werden, beeinträchtigt werden könnten.
[EA:Web33]

Das letzte Entwurfsmuster, das in dieser Arbeit erläutert wird, ist das **Gateway Routing Pattern**. Es kommt zum Einsatz, wenn ein Client innerhalb einer Applikation auf mehrere Dienste zugreift. Dabei fungiert ein **API-Gateway** als zentraler Kommunikationspunkt, der die **Anfragen des Clients empfängt** und **an die entsprechenden Dienste weiterleitet**. Dadurch soll die Komplexität der Kommunikation zwischen dem Client und mehreren Diensten reduziert werden, da der Client nur über einen einzigen Endpunkt mit den Diensten interagiert. [EA:Web34]

In Abbildung 1.19 ist das Gateway Routing Pattern dargestellt:

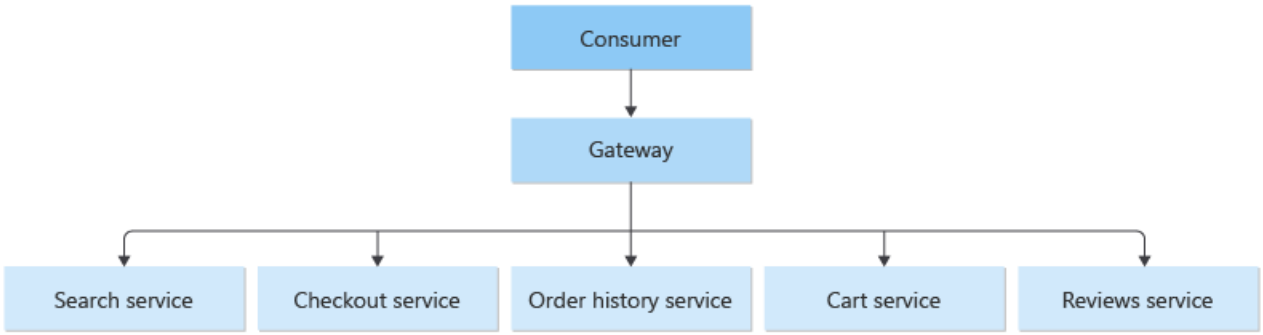


Abbildung 1.19: Gatway Routing
[EA:Web34]

Um die Funktionsweise des Musters näher zu verdeutlichen, wird näher auf die Konfiguration des API-Gateways im Projekt CGM MAXX LITE eingegangen.

In *Listing 1.6* wird die **Konfiguration des Gateways** dargestellt:

```yaml
spring:
  application:
    name: cml-api-gateway
  cloud:
    gateway:
      mvc:
        routes:
          - id: patient-service
            uri: lb://cml-patient-management
            predicates:
              - Path=/ais/desktop/domain/patient/**
          - id: file-service
            uri: lb://cml-file-management
            predicates:
              - Path=/ais/desktop/**

server:
  port: 8080
```

Listing 1.6: `application.yml - API Gateway`

Das API-Gateway lauscht auf den Port 8080 und ermöglicht durch die Definition von Routen den Zugriff auf die dahinterliegenden Microservices. Eine genauere Beschreibung der einzelnen Microservices findet sich in Abschnitt 1.3.4.

Damit Nutzer/innen diese Services nutzen können, erfolgt der Zugriff über das Frontend. Dabei gibt es eine Konfigurationsdatei mit dem Namen **environment.ts**, in der spezifische Umgebungsvariablen gesetzt werden können. In *Listing 1.7* werden die URLs für den Zugriff auf das Patient Service und File Service festgelegt. Beide greifen dabei auf den Port 8080 zu, also auf das API Gateway, das die Anfragen an die entsprechenden Microservices weiterleitet:

```
export const environment = {
  production: false,
  baseUrlSearch: "http://localhost:8080/ais/desktop/domain/patient",
  baseUrlFile: "http://localhost:8080/ais/desktop",
};
```

Listing 1.7: `environment.ts`

### DevOps und Automatisierung

Nachdem die Rolle von Entwurfsmustern in der Cloud erklärt wurde, wird im Anschluss der Prozess der Bereitstellung thematisiert.

Der Begriff DevOps ist eine **Abkürzung für Development Operations**. Es handelt sich dabei um eine Entwicklungsmethodik, die es ermöglicht, die Bereitstellung von Software zu beschleunigen, indem bestimmte **Arbeitsschritte automatisiert** und die Zusammenarbeit zwischen dem Entwicklungsteam und IT-Operations-Team[11] gefördert wird. [EA:Web35]

DevOps bietet die Möglichkeit, nicht nur alle paar Monate neue Änderungen zu veröffentlichen, wie es vor einigen Jahren noch der Fall war, sondern sogar mehrmals am Tag neue Releases bereitzustellen. [EA:Web36]

Zusätzlich bietet DevOps weitere Vorteile:

- Bessere Qualität der Software
- Effizienteres Arbeiten durch die Zusammenarbeit sonst isolierter Teams
- Wettbewerbsvorteil durch die schnelle Bereitstellung neuer Funktionen

[11] Personen, die für die Bereitstellung von IT-Services verantwortlich sind

Wie zu Beginn bereits erwähnt, geht es darum, Software schneller ausliefern zu können. Allerdings bezieht sich dies nicht nur auf die Bereitstellung. Damit Software schneller auf den Markt gebracht werden kann, müssen auch die vorhergehenden Phasen berücksichtigt werden. Dieser **iterative und automatisierte Prozess** wird als **Lebenszyklus** bezeichnet. [EA:Web35]

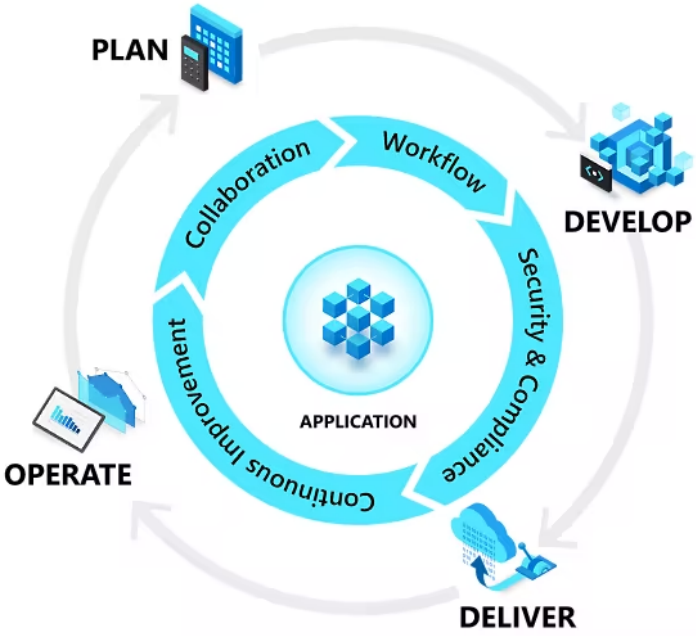


Abbildung 1.20: DevOps-Lebenszyklus
[EA:Web37]

In Abbildung 1.20 wird dargestellt, wie der DevOps-Lebenszyklus aussieht. Der Prozess **beginnt mit der Planung**. Dabei wird festgelegt, welche Funktionalitäten im nächsten Sprint umgesetzt werden sollen, um Kund/innen einen Mehrwert zu bieten. Hierbei wird auf **agiles Projektmanagement** gesetzt, um auf das Feedback der Anwender/innen einzugehen und falls gewünscht, entsprechende Änderungen vorzunehmen.

Nachdem die Planung abgeschlossen ist, kann mit der **Entwicklung** begonnen werden. Neben der Implementierung neuer Funktionalitäten und der Behebung von Bugs wird der geschriebene Code auch getestet (siehe Kapitel ??). Bevor eine neue Version der Software bereitgestellt wird, sollte diese vorher ausgiebig getestet werden. Dazu werden automatische Tests durchlaufen und anschließend kann die Software automatisch ausgeliefert werden. Realisiert wird dies mithilfe von CI/CD[12], das im Anschluss kurz beschrieben wird. Eine genauere Beschreibung kann in Kapitel ?? gefunden werden.

[12]Continuous Integration and Continuous Delivery

Die nächste Phase befasst sich mit der **Bereitstellung**. Dabei sollen entsprechende Anwendungen zuverlässig in einer Produktionsumgebung bereitgestellt werden. Dazu gehört auch die Konfiguration und Verwaltung der Infrastruktur, was beispielsweise mit Infrastructure as Code (IaC) erreicht werden kann. Dies wird im Abschnitt 1.5.2 näher beschrieben.

Treten in der Produktionsumgebung keine Fehler auf, so kann die Software für die Nutzer/innen zugänglich gemacht werden. Diese Phase bezeichnet man als **Ausführungsphase**. Die Software wird dabei überwacht und im Falle eines Fehlers wird versucht, diesen zu beheben.

Dieser Prozess wiederholt sich und trägt dazu bei, dass das **DevOps-Team kontinuierlich dazulernt** und somit bessere Ergebnisse liefern kann. [EA:Web35, EA:Web37]

Damit einzelne Phasen automatisiert und optimiert werden können, gibt es verschiedene Möglichkeiten, die eingesetzt werden können. Eine davon ist CI/CD. Dabei handelt es sich um eine Kombination aus kontinuierlicher Integration (Continuous Integration) und kontinuierlicher Lieferung (Continuous Delivery). Dadurch wird neuer Code ausgeliefert, ohne diesen manuell bereitstellen zu müssen.

Die **kontinuierliche Integration (CI)** sorgt dafür, dass der Code eines bestimmten Branches, in der Regel des main-Branches, bei jedem neuen Commit getestet wird, indem die Anwendung gebaut wird.

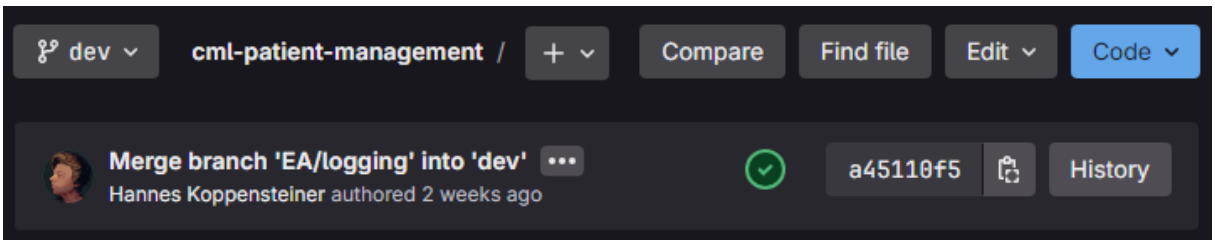


Abbildung 1.21: Erfolgreicher Build - CGM MAXX LITE

Abbildung 1.21 zeigt einen Ausschnitt aus einem **GitLab-Repository**, das den erfolgreichen CI-Build eines Projekts dokumentiert. Im gezeigten Beispiel handelt es sich um den Microservice zur Verwaltung von Patientendaten. Sobald neue Änderungen am dev-Branch vorgenommen werden, wird versucht, die Anwendung zu bauen. In diesem Fall wurde Logging zur Anwendung hinzugefügt. Der grüne Haken auf der rechten Seite symbolisiert den erfolgreichen Build-Prozess.

Sobald der CI-Prozess abgeschlossen ist, sorgt die **kontinuierliche Lieferung (CD)** dafür, dass die Software so vorbereitet wird, dass sie jederzeit bereitgestellt werden kann. [EA:Web38]

### Infrastructure as Code (IaC)

Infrastructure as Code (IaC) ist eine **in DevOps angewandte Methode**, die insbesondere **in der kontinuierlichen Bereitstellung von Software Anwendung findet**.
Dabei wird Code geschrieben, der es ermöglicht, die Infrastruktur automatisch zu konfigurieren und bereitzustellen, sodass manuelle Eingriffe vermieden werden.
[EA:Web37]

Diese Methode eignet sich daher besonders gut, um **komplexe und umfangreiche Umgebungen** bereitzustellen, die häufig aufgesetzt werden müssen. Dazu zählen unter anderem Entwicklungs- und Test-Umgebungen.
[EA:Web30, EA:Web39]

IaC bietet dabei einige Vorteile:

- Umgebungen können einfach dupliziert werden
- Weniger Fehler durch automatische Erstellung
- Schnelle und zuverlässige Bereitstellung

Um Infrastruktur mithilfe von Code zu erzeugen, gibt es zwei verschiedene Ansätze:

- **Deklarativer Ansatz**
  - Der gewünschte Endzustand des Systems wird durch die Definition von Ressourcen und deren Einstellungen beschrieben.
- **Imperativer Ansatz**
  - Der gewünschte Endzustand wird durch die Beschreibung aller notwendigen Schritte zur Konfiguration der Ressourcen erreicht.

Damit die oben genannten Vorteile realisiert werden können, stehen zahlreiche Tools von verschiedenen Herstellern zur Verfügung. [EA:Web40]
Zu den bekanntesten IaC-Tools gehören:

- Terraform
- AWS CloudFormation
- Ansible

Neben den genannten Tools existieren zahlreiche weitere Werkzeuge, die für die Erstellung und Verwaltung von Infrastruktur genutzt werden können. Da sich all diese Tools in bestimmten Punkten unterscheiden, ist es wichtig, ein für den Anwendungsfall passendes Werkzeug auszuwählen.

Um die Unterschiede der genannten IaC-Tools zu verdeutlichen, werden diese in Tabelle 1.4 miteinander verglichen:

| **Kriterium** | **Terraform** | **CloudFormation** | **Ansible** |
|---|---|---|---|
| Anbieter | HashiCorp | Amazon Web Services | Red Hat |
| Ansatz | Deklarativ | Deklarativ | Deklarativ/imperativ |
| Konfigurationssprache | HCL/JSON | YAML/JSON | YAML |
| Multi-Provider-Support | Ja | Nein | Ja |
| Kosten und Lizenzierung | Kostenlos | AWS-Ressourcen | Kostenlos |

Tabelle 1.4: IaC-Tools

Im weiteren Verlauf wird erklärt, wofür die einzelnen Tools verwendet werden, und wie sie sich in ihrer Funktionsweise unterscheiden. [EA:Web41, EA:Web42, EA:Web43]

### Terraform

Terraform ist ein von HashiCorp entwickeltes Tool für Infrastructure as Code (IaC), das die Bereitstellung und Verwaltung von Infrastruktur sowohl in Cloud-Umgebungen als auch On-Premise ermöglicht. Geschrieben werden die Konfigurationsdateien in der HashiCorp Configuration Language (HCL) oder in JSON, die beide eine simple und leicht verständliche Struktur aufweisen.

Ein wesentlicher Vorteil von Terraform ist die Unterstützung zahlreicher Cloud-Dienstanbieter, darunter AWS, Azure und Google Cloud Platform. [EA:Web44]

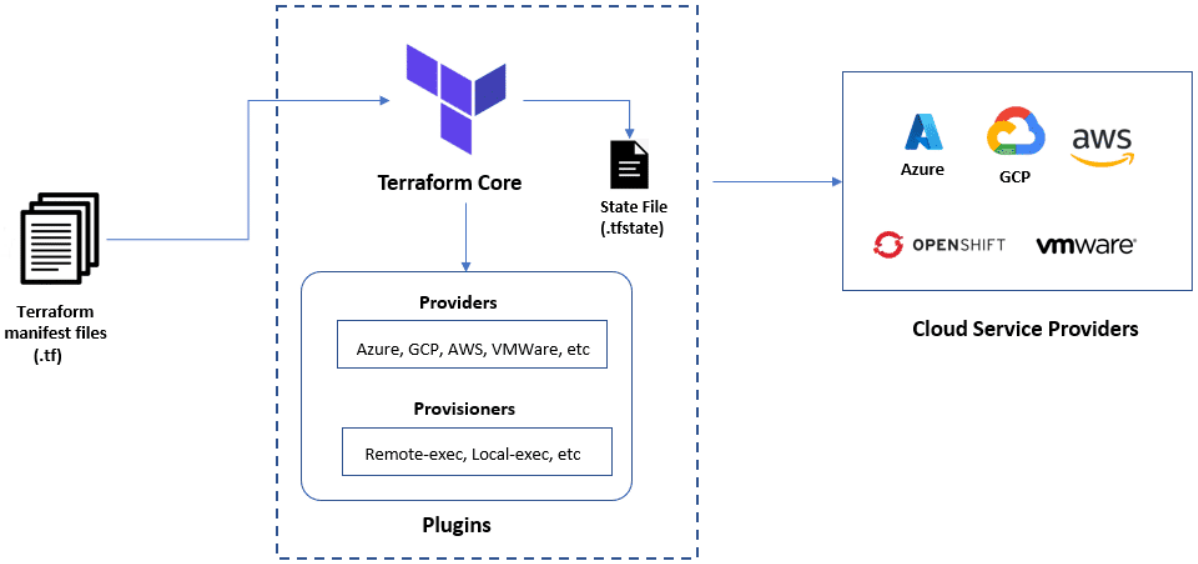


Abbildung 1.22: Terraform Architektur
[EA:Web49]

Abbildung 1.22 veranschaulicht die grundlegende Architektur von Terraform. Der Prozess beginnt mit dem Erstellen einer Konfigurationsdatei, in der alle notwendigen Einstellungen vorgenommen werden, um den gewünschten Endzustand des Systems zu erreichen. Anschließend wird mithilfe der **Terraform CLI**[13] ein Ausführplan erstellt. Erst wenn dieser bestätigt wird, werden die Änderungen durchgeführt.

Die Terraform CLI, auch Terraform Core genannt, ist das zentrale Werkzeug für die Verwaltung von Infrastruktur. Weiters interagiert sie mit **Providern**, die die Kommunikation mit unterschiedlichen Diensten ermöglichen. Dabei werden die vorliegenden Konfigurationen in die für den Dienstanbieter vorgesehenen API Calls umgewandelt und passen die Einstellungen entsprechend an.
[EA:Web44, EA:Web49]

[13]Command-line Interface

### AWS CloudFormation

Wie der Name bereits andeutet, handelt es sich um einen Dienst von Amazon Web Services, der speziell auf die **Bereitstellung von AWS-Ressourcen** ausgerichtet ist. Der Dienst selbst ist kostenlos. Es entstehen lediglich Kosten für die bereitgestellten Ressourcen, wie beispielsweise EC2-Instanzen.

In Abbildung 1.23 wird der Ablauf der Bereitstellung gezeigt:

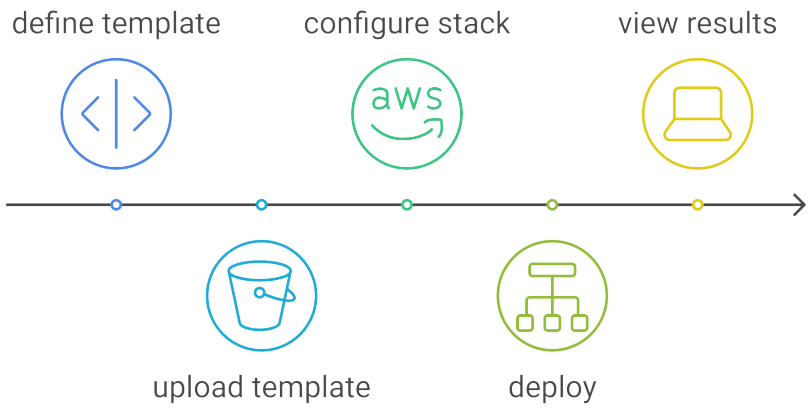


Abbildung 1.23: AWS CloudFormation Funktionsweise

Ähnlich wie bei Terraform, muss auch bei CloudFormation eine **Vorlage** erstellt werden, welche die benötigten Ressourcen beschreibt. Bei der Konfigurationssprache kann entweder YAML oder JSON eingesetzt werden. Alle Ressourcen, die in einer Vorlage definiert werden, werden als eine Einheit betrachtet, die auch als **Stack** bezeichnet wird.

Um die Änderungen einer Vorlage zu übernehmen, muss ein **Änderungssatz** erstellt werden. Dabei wird auf den Speicherort der ausgewählten Vorlage verwiesen, die entweder **lokal oder in einem Amazon S3-Bucket abgespeichert** werden kann. Der Änderungssatz zeigt dabei, welche Änderungen vorgenommen werden und ähnelt somit einem Ausführungsplan in Terraform. [EA:Web50, EA:Web51]

Der Unterschied zwischen diesen beiden Tools besteht darin, dass wenn in CloudFormation ein Fehler beim Änderungsvorgang auftritt, der Stack auf den letzten funktionsfähigen Stand zurückgesetzt wird. In Terraform hingegen werden nur die Ressourcen isoliert, die vom Fehler betroffen sind. Zudem unterstützt Terraform neue AWS-Funktionen oft schneller als CloudFormation selbst. [EA:Web52]

### Ansible

Ansible ist kein IaC-Tool im eigentlichen Sinne. Es ist vielmehr ein Konfigurationswerkzeug, das sich auf die Automatisierung unterschiedlicher Tasks fokussiert. Dazu zählen unter anderem Systemupdates, die Installation von Software, das Einrichten einer Firewall, und vieles mehr. Um dies zu realisieren, werden sogenannte **Playbooks** eingesetzt, die in der Datenserialisierungssprache YAML geschrieben werden.

Während Terraform besser für die Bereitstellung von Infrastruktur geeignet ist, spielt Ansible seine Stärken bei der Konfigurationsverwaltung aus. Beide Werkzeuge haben ihre Vor- und Nachteile. In der Praxis werden Terraform und Ansible häufig kombiniert, um die Stärken beider Tools optimal zu nutzen. [EA:Web45, EA:Web48]

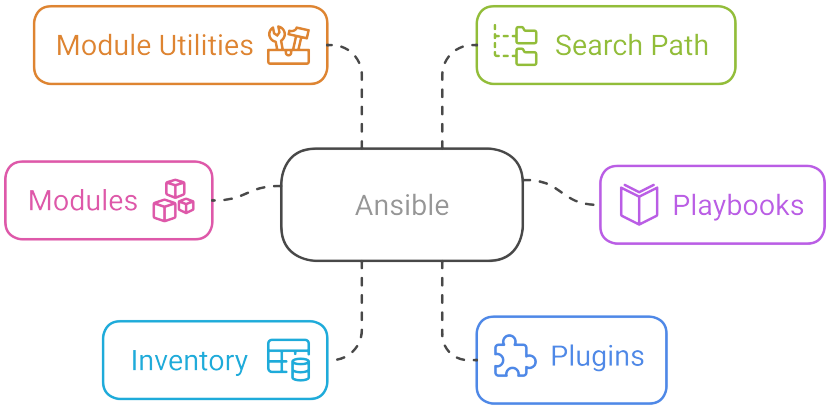


Abbildung 1.24: Ansible Architektur

Um die Funktionsweise von Ansible besser zu verstehen, wird die Architektur anhand der Grafik in Abbildung 1.24 näher erklärt. Diese setzt sich aus folgenden Komponenten zusammen:

- **Modules**
  - Skripte, die auf Managed Nodes[14] kopiert und dort ausgeführt werden
  - Standardmäßig erfolgt die Ausführung über SSH
  - Erledigen die in den Playbooks definierten Tasks
- **Module Utilities**
  - Mehrfach vorkommende identische Codeausschnitte werden gespeichert
  - Redundanter Code wird vermieden und die Wartung wird erleichtert

[14] Geräte, die von Ansible verwaltet werden

- **Plugins**
  - Ergänzt Ansible um neue Funktionen
  - Eigene Plugins können mithilfe von Python geschrieben werden
- **Playbooks**
  - YAML-Dateien für die Konfiguration von Managed Nodes
  - Einfach zu lesen
- **Inventory**
  - Liste mit den Managed Nodes, die durch Ansible verwaltet werden
  - Gruppierung von Hostnamen und IP-Adressen möglich
- **Search Path**
  - Gibt an, wo benötigte Module, Plugins, Playbooks, usw. zu finden sind

[EA:Web46, EA:Web47]

Abschließend lässt sich sagen, dass es nicht das eine „perfekte“ Werkzeug gibt. Daher ist es wichtig, den genauen Anwendungsfall zu kennen, um auf dieser Grundlage ein passendes Tool auszuwählen. Das Ziel sollte sein, eine einfache und schnelle Lösung zu finden, um Infrastruktur und/oder Software bereitzustellen und so einen Wettbewerbsvorteil zu erzielen.

Zudem wird der Entwicklungsprozess erleichtert, da durch die einfache Reproduzierbarkeit von Systemen weniger Fehler auftreten. Auf diese Weise können sich die Entwickler/innen auf ihre eigentlichen Aufgaben konzentrieren, ohne stundenlang nach der Ursache eines Fehlers suchen zu müssen.

## 1.6 Fazit und Ausblick

Bevor die Erkenntnisse, die im Laufe dieser Arbeit gewonnen wurden, zusammengefasst werden, soll ein letztes Mal darauf hingewiesen werden, dass die behandelten Themen bei Weitem nicht alle Teile der Softwarearchitektur abdecken. Das Ziel dieser Arbeit ist es, die Bedeutung der Softwarearchitektur hervorzuheben und zu vermitteln, dass sie weit mehr als nur die Implementierung von sauberem Quellcode umfasst.

Softwarearchitektur befasst sich mit dem gesamten Lebenszyklus des Systems, beginnend bei der Planung bis hin zum Betrieb.

### 1.6.1 Zusammenfassung der Ergebnisse

Bei der Softwarearchitektur handelt es sich nicht nur um einen Bauplan, der angibt, wie ein Softwaresystem aufgebaut ist und wie die einzelnen Bestandteile miteinander interagieren. Es ist auch ein wichtiges Instrument für die Kommunikation. Durch verschiedene Formen der Darstellung, die in Abschnitt 1.5.1 näher beschrieben werden, können Grafiken erzeugt werden, die verwendet werden können, um mit verschiedenen Stakeholdern zu kommunizieren.

Um eine Softwarearchitektur zu entwerfen, die den Wünschen der Kund/innen entspricht, spielt die Anforderungsanalyse (siehe 1.2) eine entscheidende Rolle. Durch die Definition funktionaler und nicht-funktionaler Anforderungen wird festgelegt, was die Software können soll, um die Nutzer/innen zufriedenzustellen.

Die Ergebnisse der Anforderungsanalyse dienen dabei als Entscheidungsgrundlage für den zugrundeliegenden Architekturstil (siehe 1.3). Je nachdem, was die Anforderungen sind, eignen sich bestimmte Stile besser als andere, wobei auch hier jeder Stil seine Vor- und Nachteile hat.

Da sich diese Arbeit hauptsächlich auf cloudbasierte Anwendungen fokussiert, kann es von Relevanz sein, grundlegende Entwurfsmuster für die Cloud (siehe Abschnitt 1.5.2) zu kennen, um häufig auftretenden Problemen zu entweichen. Zudem gibt es heutzutage bereits einige Möglichkeiten, um die Entwicklung und Bereitstellung von Software zu beschleunigen. Methoden wie DevOps (siehe Abschnitt 1.5.2) werden immer bedeutender und ermöglichen es, Releases schneller und einfacher zu veröffentlichen, was einen Wettbewerbsvorteil schaffen kann.

Blickt man nun auf die Ergebnisse des Projekts CGM MAXX LITE, so wurden alle Grundfunktionalitäten erfolgreich umgesetzt und erfüllen die Erwartungen des Auftraggebers. Die Softwarearchitektur ist so konzipiert, dass das System in Zukunft einfach erweitert und angepasst werden kann. Um für hohe Leistung zu sorgen, wurde die Kommunikation zwischen Frontend und Backend so gestaltet, dass nur notwendige Daten übertragen werden. Beim gewählten Architekturstil handelt es sich um eine Microservices-Architektur, die gewährleisten soll, dass wenn ein Service ausfällt, übrige Services des Systems weiterhin zur Verfügung stehen. Dadurch ist sichergestellt, dass im Falle eines Fehlers nur bestimmte Teile des Systems vorübergehend nicht erreichbar sind. Wie die Architektur aussieht, kann mithilfe des Container-Diagramms in Abbildung 1.16 dargestellt werden.

Durch das Hinzufügen einer CI/CD-Pipeline wird nach jeder Änderung am dev-Branch geprüft, ob die Applikation erfolgreich gebaut werden kann und wird anschließend am Portainer bereitgestellt, wodurch das System von überall aus getestet und dem Auftraggeber präsentiert werden kann. Zusätzlich wird das Frontend durch ESLint überprüft, was in Abschnitt ?? genauer thematisiert wird.

Da es sich um die Entwicklung einer Arztsoftware handelt, bei der mit sensiblen Daten gearbeitet wird und Zuverlässigkeit, Fehlertoleranz sowie Sicherheit gewährleistet sein müssen, werden neben einfachen Unit-Tests auch End-To-End-Tests geschrieben, die testen, ob die Benutzeroberfläche korrekt funktioniert. Darauf wird in Kapitel ?? näher eingegangen.

Wie die Benutzeroberfläche der mobilen Arztsoftware schlussendlich aussieht, wird in Kapitel ?? gezeigt. Dabei liegt der Fokus auf einer benutzerfreundlichen Oberfläche, die leicht bedient werden kann, um alltägliche Arbeitsabläufe zu beschleunigen.

### 1.6.2 Zukünftige Entwicklungen

Wie bereits zu Beginn der Arbeit erwähnt, ist die Softwarearchitektur dynamisch und entwickelt sich ständig weiter, weshalb es schwierig ist vorherzusagen, wohin sich die Architektur von Software in Zukunft entwickeln wird. Dennoch ist klar, dass Cloud Computing und die zunehmende Verlagerung von Anwendungen in die Cloud eine immer zentralere Rolle spielen werden.

Trotz all dieser Veränderungen wird sich an einer Sache nichts ändern: Eine gut durchdachte Softwarearchitektur ist maßgeblich für den Erfolg eines Softwaresystems verantwortlich. Deswegen ist es wichtig, sich kontinuierlich mit neuen Technologien und Entwurfsmustern zu beschäftigen, um mit der Konkurrenz mithalten zu können.

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Listings

# Literaturverzeichnis


[EA:Book01] Robert C. Martin:
*Clean Architecture: A Craftsman's Guide to Software Structure and Design*
Pearson Education Inc., 2018
ISBN: 978-0-13-449416-6

[EA:Book02] Mark Richards & Neal Ford:
*Handbuch moderner Softwarearchitektur*
O'Reilly, 2021
ISBN: 978-3-96009-149-3

[EA:Book03] Len Bass, Paul Clements, Rick Kazman:
*Software Architecture in Practise*
Addison-Wesley Professional, 2003
ISBN: 978-0321154958

[EA:Web01] https://leanpub.com/software-architecture-for-developers/read
Definition der Softwarearchitektur
19.10.2024

[EA:Web02] https://thestory.is/en/process/development-phase/software-architecture/
Aufgaben und Ziele der Softwarearchitektur
29.10.2024

[EA:Web03] https://www.redhat.com/en/blog/what-is-software-architect
Aufgaben des Softwarearchitekten
29.10.2024

[EA:Web04] https://www.geeksforgeeks.org/functional-vs-non-functional-requirements
Funktionale und nicht-funktionale Anforderungen
02.11.2024

[EA:Web05] https://www.geeksforgeeks.org/software-engineering-classification-of-software-requirements
Anforderungsanalyse
02.11.2024

[EA:Web06] https://en.wikipedia.org/wiki/Agile-software-development
Agile Softwareentwicklung
02.11.2024

[EA:Web07] https://www.studysmarter.de/ausbildung/ausbildung-in-it/fachinformatiker-anwendungsentwicklung/funktionale-anforderungen
Funktionale Anforderungen
02.11.2024

[EA:Web08] https://www.ibm.com/docs/en/aix/7.1?topic=implementation-performance-requirements-documentation
Nicht-funktionale Anforderungen - Performance
16.11.2024

[EA:Web09] https://www.geeksforgeeks.org/recovery-testing-in-software-testing
Nicht-funktionale Anforderungen - Recovery Testing
16.11.2024

[EA:Web10] https://www.cgm.com/aut-de/produkte/arztpraxis/cgm-maxx.html
CGM MAXX
22.11.2024

[EA:Web11] https://www.franchiseportal.at/definition/unternehmensvision-a-4974
Vision
11.12.2024

[EA:Web12] https://medium.com/@alastairallen/visualising-software-architecture-why-its-important-and-how-i-do-it-5c2f4b65b31a
Architekturmodellierung
18.12.2024

[EA:Web13] https://miro.com/diagramming/what-is-software-architecture-diagramming
Softwarearchitektur - Diagramme
18.12.2024

[EA:Web14] https://www.sciencedirect.com/topics/computer-science/architecture-model
Architekturmodellierung
22.12.2024

[EA:Web15] https://www.studysmarter.de/studium/informatik-studium/softwareentwicklung/softwarearchitektur
Architekturentwurf
23.12.2024

[EA:Web16] https://www.redhat.com/en/blog/software-architecture-tips
Architekturentwurf
23.12.2024

[EA:Web17] https://miro.com/blog/uml-diagram
UML-Diagramme
23.12.2024

[EA:Web18] https://www.geeksforgeeks.org/unified-modeling-language-uml-introduction
UML-Diagramme
23.12.2024

[EA:Web19] https://miro.com/de/diagramm/was-ist-ein-uml-diagramm
UML-Diagramme
23.12.2024

[EA:Web20] https://t2informatik.de/en/smartpedia/behavior-diagram
Verhaltensdiagramme
27.12.2024

[EA:Web21] https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial
Klassendiagramme
27.12.2024

[EA:Web22] https://www.freecodecamp.org/news/a-quick-intro-to-dependency-injection-what-it-is-and-when-to-use-it-7578c84fa88f
Dependency Injection
27.12.2024

[EA:Web23] https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-sequence-diagram
Sequenzdiagramm
27.12.2024

[EA:Web24] https://www.lucidchart.com/pages/de/uml-sequenzdiagramme
Sequenzdiagramm
27.12.2024

[EA:Web25] https://www.freecodecamp.org/news/how-to-create-software-architecture-diagrams-using-the-c4-model
C4-Modell
30.12.2024

[EA:Web26] https://miro.com/de/diagramm/was-ist-das-c4-model-in-software-architektur
C4-Modell
30.12.2024

[EA:Web27] https://www.lucidchart.com/blog/c4-model
C4-Modell
31.12.2024

[EA:Web28] https://c4model.com/diagrams
C4-Modell - Diagramme
31.12.2024

[EA:Web29] https://c4model.com/diagrams/component
C4-Modell - Komponentendiagramm
01.01.2025

[EA:Web30] https://aws.amazon.com/de/what-is/iac
Infrastructure as Code (IaC)
01.01.2025

[EA:Web31] https://learn.microsoft.com/en-us/azure/architecture/patterns
Entwurfsmuster für die Cloud
01.01.2025

[EA:Web32] https://learn.microsoft.com/en-us/azure/architecture/patterns/backends-for-frontends
Backends for Frontends
01.01.2025

[EA:Web33] https://learn.microsoft.com/de-de/azure/architecture/patterns/compute-resource-consolidation
Compute Resource Consolidation
01.01.2025

[EA:Web34] https://learn.microsoft.com/de-de/azure/architecture/patterns/gateway-routing
Gatway-Routing
02.01.2025

[EA:Web35] https://www.ibm.com/de-de/topics/devops
DevOps
02.01.2025

[EA:Web36] https://www.atlassian.com/de/devops
DevOps
02.01.2025

[EA:Web37] https://azure.microsoft.com/de-de/resources/cloud-computing-dictionary/what-is-devops
DevOps
03.01.2025

[EA:Web38] https://about.gitlab.com/de-de/topics/ci-cd
CI/CD
03.01.2025

[EA:Web39] https://www.ibm.com/think/topics/infrastructure-as-code
Infrastructure as Code (IaC)
03.01.2025

[EA:Web40] https://www.redhat.com/en/topics/automation/what-is-infrastructure-as-code-iac
Infrastructure as Code (IaC)
04.01.2025

[EA:Web41] https://developer.hashicorp.com/terraform/tutorials/aws-get-started/infrastructure-as-code
Terraform
23.01.2025

[EA:Web42] https://www.redhat.com/en/topics/automation/ansible-vs-terraform
Ansible
23.01.2025

[EA:Web43] https://aws.amazon.com/de/cloudformation
AWS CloudFormation
23.01.2025

[EA:Web44] https://developer.hashicorp.com/terraform/intro
Terraform
23.01.2025

[EA:Web45] https://www.geeksforgeeks.org/difference-between-terraform-vs-ansible
Ansible vs Terraform
24.01.2025

[EA:Web46] https://docs.ansible.com/ansible/latest/dev-guide/overview-architecture.html
Ansible Architektur
24.01.2025

[EA:Web47] https://docs.ansible.com/ansible/latest/network/getting-started/basic-concepts.html
Ansible Grundbegriffe
24.01.2025

[EA:Web48] https://spacelift.io/blog/ansible-vs-terraform
Ansible vs Terraform
24.01.2025

[EA:Web49] https://spacelift.io/blog/terraform-architecture
Terraform Architektur
26.01.2025

[EA:Web50] https://aws.amazon.com/de/cloudformation
AWS CloudFormation
26.01.2025

[EA:Web51] https://docs.aws.amazon.com/de-de/AWSCloudFormation/latest/UserGuide/cloudfo
overview.html
AWS CloudFormation Funktionsweise
26.01.2025

[EA:Web52] https://www.porscheinformatik.com/en/cloudformation-vs-terraform
CloudFormation vs Terraform
01.02.2025

[EA:Web53] https://www.atlassian.com/de/devops/frameworks/public-cloud
Public Cloud
02.02.2025

[EA:Web54] https://www.redhat.com/de/topics/cloud-computing/public-cloud-vs-private-cloud-and-hybrid-cloud

Cloud Computing
02.02.2025

[EA:Web55] https://www.atlassian.com/microservices/cloud-computing
Cloud Computing
03.02.2025

[EA:Web56] https://www.atlassian.com/de/devops/frameworks/private-cloud
Private Cloud
03.02.2025

[EA:Web57] https://cloud.google.com/learn/what-is-hybrid-cloud
Hybrid Cloud
03.02.2025

[EA:Web58] https://www.ibm.com/think/insights/hybrid-cloud-advantages-disadvantages
Hybrid Cloud - Vor- und Nachteile
04.02.2025

[EA:Web59] https://www.codestone.com/news/the-importance-of-cloud-computing
Bedeutung der Cloud
04.02.2025

[EA:Web60] https://www.turing.com/blog/software-architecture-patterns-types
Architekturstile
04.02.2025

[EA:Web61] https://www.redhat.com/en/topics/microservices/what-are-microservices
Microservices-Architektur
05.02.2025

[EA:Web62] https://www.atlassian.com/de/microservices
Microservices-Architektur
05.02.2025

[EA:Web63] https://www.sap.com/austria/products/technology-platform/what-is-event-driven-architecture
Eventbasierte Architektur
06.02.2025

[EA:Web64] https://www.geeksforgeeks.org/event-driven-architecture-system-design
Eventbasierte Architektur
06.02.2025

[EA:Web65] https://www.redhat.com/en/topics/cloud-native-apps/what-is-serverless
Serverlose Architektur
07.02.2025

[EA:Web66] https://www.elastic.co/de/what-is/serverless-computing
Serverlose Architektur
07.02.2025

[EA:Img01] https://anexia.com/blog/de/public-cloud-erklaerung
Private Cloud vs. Public Cloud
03.02.2025

[EA:Img02] https://blog.hubspot.com/marketing/event-driven-architecture
Eventbasierte Architektur
06.02.2025

## Kapitelzuordnung

| **Kapitel** | **Autor** |
|---|---|
| Softwarearchitektur mit Schwerpunkt auf cloudbasierten Systemen | Emily Atzinger |
| Gestaltung einer benutzerfreundlichen Weboberfläche (UX, UI, Designmethoden) | Matthias Zimmermann |
| Entwicklung von erweiterbarer Software in Bezug auf verständlichen Sourcecode (Designpattersn, Orthogonalität, Komponentisierung) | Oskar Brödler |
| Testverfahren und Qualitätssicherung mit Fokus auf Angular Anwendungen | Hannes Koppensteiner |

# Arbeitsprotokolle

## Emily Atzinger

| KW | Beschreibung | Stunden |
|---|---|---|
| | **2024** | |
| 36 | Grundrecherche + Themenaufstellung | 14 |
| 37 | Overleaf Setup | 4 |
| 38 | Literaturrecherche + Inhaltsverzeichnis | 11 |
| 39 | Allgemeines | 6 |
| 40 | Architekturstile | 7.5 |
| 42 | Einleitung | 3 |
| 42 | Definition | 10 |
| 42 | Aufgaben und Ziele | 2 |
| 44 | Rolle des Softwarearchitekten | 4 |
| 44 | Herausforderungen und Ziele | 4.5 |
| 44 | Anforderungsanalyse | 4.5 |
| 45 | Microservices | 11.5 |
| 46 | Anforderungsanalyse | 5.5 |
| 47 | Microservices | 3 |
| 47 | Anforderungsanalyse | 1.5 |
| 48 | Herausforderungen & Probleme | 1.5 |
| 48 | Architekturstile | 3.5 |
| 51 | Architekturentwurf | 3 |
| 52 | Architekturentwurf | 13.5 |
| 1 | Architekturentwurf | 10 |
| | **2025** | |
| 1 | Architekturentwurf | 17 |
| 3 | Microservices | 3 |
| 4 | Architekturentwurf | 12.5 |
| 5 | Architekturentwurf | 2.5 |
| 5 | Architekturstile | 1 |
| 5 | Cloud Computing | 3 |
| 6 | Cloud Computing | 10 |
| 6 | Architekturstile | 21.5 |
| 8 | Feinschliff | 3.5 |
| 9 | Feinschliff | 6.5 |
| 11 | Feinschliff | 5 |
| **Gesamtaufwand** | | **209** |