Einführung

Im Zeitraum vom 01.03.2025 – 31.08.2025 habe ich mein Pflichtpraktikum bei der ilume Informatik AG in Mainz absolviert. Es handelt sich um ein IT-Beratungshaus, das sich auf CRM-Systeme, die Digitalisierung von Geschäftsprozessen sowie Cloud-Lösungen spezialisiert. Neben der Integration von Standardsoftware in bestehende IT-Landschaften entwickelt ilume auch individuelle Softwarelösungen für Kunden aus verschiedensten Branchen, darunter Pharma, Chemie, Bankwesen, Versicherungen und Logistik.

Das Unternehmen wurde im Jahr 2000 gegründet und beschäftigt mittlerweile rund 150 Mitarbeitende, die in drei Abteilungen organisiert sind: DEV (Custom Development), SMA (Smart Automation) und CRM (Customer Relationship Management). Während meines Praktikums war ich im Bereich DEV tätig und dort dem Team Digitalization zugeordnet, das sich vor allem mit Web- und Mobile-Entwicklung beschäftigt und sowohl externe Kunden- als auch interne Projekte umsetzt.

Vor dem Pflichtpraktikum war ich bereits als Werkstudentin bei ilume tätig, wodurch ich mit vielen Unternehmensprozessen vertraut war und mein Praktikum effizient beginnen konnte. Etwa 5–10 % meiner Aufgaben lagen im Content-Management-Bereich, in dem ich sowohl Magnolia CMS-Seiten als auch AWS-basierte Webseiten für Kunden aus der Pharma-Branche betreute. Dabei passte ich Inhalte an, implementierte Änderungen an Funktionen und übernahm die Bildbearbeitung. Ein wesentlicher Bestandteil meiner Tätigkeit war die Mitarbeit am internen KI-basierten Webprojekt „Blog Writer“. In den letzten beiden Monaten, von Juli bis August, lag mein Schwerpunkt auf der Einarbeitung in Compose Multiplatform.

1. Woche (03.03. – 07.03.2025)

In der ersten Woche meines Praktikums fand ein Meeting mit dem Management des DEV-Teams statt, bei dem das Thema für das Projekt vorgestellt wurde. Die Grundidee war, ein Tool zu entwickeln, das täglich aktuelle Nachrichtenartikel sammelt, diese anhand der individuellen Interessen eines ilume-Mitarbeiters bewertet und daraus KI-gestützte LinkedIn-Beiträge generiert. Diese Beiträge werden per E-Mail an die Mitarbeiter geschickt und können dort für LinkedIn genutzt werden.

Das Projekt wurde in einem Zweierteam umgesetzt, wobei ich die Verantwortung für die Backend-Entwicklung übernahm. Zur Abstimmung führten wir tägliche Meetings (Dailys) durch, in denen wir den Projektfortschritt sowie offene Fragen und Herausforderungen besprachen.

Als ersten Schritt erarbeitete ich einen Fragebogen für die Kollegen, der die notwendigen Profilinformationen abfragte und es der KI dadurch erleichterte, Artikel gezielt den jeweiligen Interessen zuzuordnen. Der Fragebogen umfasste Themen wie

Studium, aktuelle Position und Aufgaben, Berufserfahrung, Interessengebiete sowie den gewünschten Schreibstil, in dem ein LinkedIn-Beitrag formuliert sein sollte. Zusätzlich konnten die Mitarbeitenden optional weitere Angaben machen, um die Beiträge noch individueller und persönlicher zu gestalten.

Da mir bereits zuvor von ilume ein MacBook zur Verfügung gestellt wurde, richtete ich darauf meine Entwicklungsumgebung ein. Dazu übertrug ich das bestehende Projektgerüst aus dem Bitbucket-Repository (Atlassian) auf meinen Rechner, das zunächst nur aus einer grundlegenden Ordnerstruktur und wenigen Konfigurationsdateien bestand. In der zugehörigen pyproject.toml-Datei waren bereits einige Einstellungen und Bibliotheken, wie Python, Poetry (Paket- und Dependency-manager für Python) und LangChain (Framework zur Erstellung von Anwendungen mit KI-Modellen) vordefiniert. Die weiteren für das Projekt notwendigen Tools musste ich eigenständig recherchieren und in meiner Entwicklungsumgebung einrichten.

LangChain als Framework bildet die Grundlage für die KI-Anwendung und ermöglicht unter anderem die Verarbeitung von Textdaten und das Verknüpfen verschiedener Modelle zu Ketten (Chains). Darauf aufbauend beschäftigte ich mich mit der Erweiterung LangGraph zur Visualisierung und Strukturierung komplexer Workflows sowie mit Agenten, die es der KI erlauben, selbstständig Entscheidungen zu treffen und auf Basis von Regeln und Eingaben Aktionen auszuführen. Dazu arbeitete ich die Tutorials von DeepLearning.AI durch, um die Funktionsweise praxisnah kennenzulernen.

2. Woche (10.03. – 14.03.2025)

In der zweiten Woche stand die praktische Einarbeitung in die eingesetzten Methoden und Frameworks im Mittelpunkt. Mir wurde ein OpenAI-API-Key zur Verfügung gestellt, mit dem ich erste Experimente durchführen konnte. Ziel war es, die Funktionen von LangChain und seinen Erweiterungen systematisch zu verstehen und praxisnah anzuwenden.
Zunächst erarbeitete ich die Grundlagen der Textanalyse und Datenaufbereitung. Hierzu erstellte ich kleinere Python-Skripte, um zentrale Konzepte wie Clustering zu wiederholen und den Aufbau von Embeddings zu testen. Embeddings wandeln Texte in hochdimensionale Zahlenvektoren um, sodass ähnliche Inhalte in diesem Raum nahe beieinander liegen. Diese Darstellung bereitet die Daten für weitere Verarbeitungsschritte vor, insbesondere für Retrievers, die gezielt relevante Informationen aus einer Sammlung von Dokumenten abrufen können.

Auf dieser Grundlage begann ich, die LangChain-API praktisch anzuwenden. Zunächst arbeitete ich mit Chains, um Verarbeitungsschritte wie Dokumentenabruf, Textaufbereitung und Embeddings linear hintereinander zu testen. Chains sind einfach und eignen sich gut für standardisierte Workflows. Für komplexere Szenarien, in denen die KI Entscheidungen treffen, unterschiedliche Pfade wählen oder externe Tools

einbinden muss, kommen Graphs mit Agenten zum Einsatz. Die Agenten steuern den Ablauf innerhalb des Graphs, treffen selbstständig Entscheidungen und wählen z. B. aus, welche Dokumente oder Tools genutzt werden sollen. Dadurch lassen sich dynamische und interaktive Workflows flexibel abbilden.

Ein weiterer Schwerpunkt lag auf dem Prompt Engineering. Durch systematisches Üben verschiedener Eingabeformate konnte ich beobachten, wie sich die Qualität und Präzision der KI-Antworten durch klare Anweisungen, Kontextgestaltung und Beispiele beeinflussen lassen. Diese Erkenntnisse bildeten die Grundlage für spätere Aufgaben, bei denen Prompts gezielt für die Generierung von LinkedIn-Beiträgen gestaltet werden müssen.

Als nächstes beschäftigte ich mich mit der ersten Aufgabe für das „Blog Writer“-Projekt: wie die relevanten Nachrichtenartikel automatisiert gesammelt werden können. Zunächst testete ich klassische Web-Scraping-Methoden, unter anderem mit BeautifulSoup zur Extraktion von Textinhalten aus HTML-Seiten sowie Selenium, um dynamisch geladene Webseiten und Interaktionen im Browser zu simulieren. Zusätzlich prüfte ich APIs wie Tavily Search und Reddit API. API-basierte Suchvorgänge lieferten oft unvollständige oder thematisch unpräzise Ergebnisse. Selenium ermöglichte zwar die Simulation von Benutzerinteraktionen, führte jedoch zu hohem Aufwand und war nicht zuverlässig, da moderne Webseiten häufig automatisierte Browser blockieren oder zusätzliche Sicherheitsmechanismen implementieren. Auch die Kombination aus API und Web-Scraping konnte nicht alle notwendigen Inhalte stabil extrahieren.

Nach intensiver Recherche entschied ich mich schließlich für einen RSS-Feed-basierten Ansatz. Zwar müssen die Feed-Links zuvor definiert werden, dafür ermöglicht die Kombination aus der Python-Bibliothek feedparser und dem RSSFeedLoader von LangChain eine schnelle und stabile Erfassung der Artikel. feedparser übernimmt die Validierung, das Parsen und die Bereinigung der Feeds, einschließlich der Extraktion von Titel, Link, Veröffentlichungsdatum und Inhaltszusammenfassung. Der RSSFeedLoader liefert den vollständigen Artikeltext direkt als Fließtext, enthält jedoch häufig keine Datumsinformationen. Durch die Kombination beider Werkzeuge lassen sich sowohl vollständige Inhalte als auch wichtige Metadaten zuverlässig erfassen und in optimaler Qualität für die weitere Verarbeitung bereitstellen.

3. Woche (17.03. – 21.03.2025)

In der dritten Woche konzentrierte ich mich auf die Auswahl einer geeigneten Datenbank für das Projekt. Da die zu verarbeitenden Inhalte wie Artikeltexte, RSS-Feeds und Personenprofile sehr unterschiedlich aufgebaut sind und sich ihre Struktur häufig ändern kann, fiel die Entscheidung auf eine NoSQL-Datenbank.

Dabei habe ich zwei Ansätze genauer betrachtet: Neo4j als Graph-Datenbank und MongoDB als dokumentenorientierte Datenbank. Die Wahl fiel zunächst auf Neo4j, da es häufig in Kombination mit LLMs und Knowledge-Graph-Ansätzen empfohlen wird. Im Rahmen des Kurses „Knowledge Graph for RAG“ von DeepLearning.AI sowie durch die Arbeit mit der Neo4j-Dokumentation machte ich mich mit den Konzepten und Werkzeugen von Graph-Datenbanken vertraut. Neo4j eignet sich grundsätzlich für die Speicherung von vernetzten Informationen in Form von Knoten (Daten) und Kanten (Beziehungen). Für mein Projekt wäre der Einsatz von Neo4j möglich gewesen, allerdings ist der Aufwand für Modellierung, Abfragen und Setup deutlich höher.

Im nächsten Schritt habe ich mich mit MongoDB beschäftigt und diese für das Projekt ausgewählt. In MongoDB werden Daten in Clustern organisiert, die wiederum mehrere Datenbanken enthalten können. Innerhalb einer Datenbank werden die Informationen in Collections (vergleichbar mit Tabellen in SQL) gespeichert und jede Collection besteht aus Dokumenten, die im JSON-Format abgelegt sind. Diese Struktur ermöglicht es, Inhalte ohne starres Schema zu speichern und bei Bedarf zu erweitern. Gleichzeitig erlaubt die dokumentenorientierte Abfrage mit MongoDB eine effiziente Suche und Verarbeitung, auch wenn sich Datenfelder zwischen einzelnen Dokumenten unterscheiden. Ein weiterer Vorteil von MongoDB ist die Möglichkeit, Daten sowohl lokal (mit MongoDB Compass) als auch in der Cloud (mit MongoDB Atlas) zu verwalten. Dadurch lässt sich unkompliziert zwischen Entwicklungs- und Produktivumgebungen wechseln.

Für das Projekt definierte und legte ich die Domain-Entities wie Person, SourceArticle, WritingSession und GeneratedArticle an, die die Kerninformationen und Geschäftslogik der Anwendung repräsentieren. Jede Entity erbt von BaseEntity, das Methoden für __eq__ und __hash__ bereitstellt, sodass Entities zuverlässig verglichen und in Mengen oder als Schlüssel in Dictionaries verwendet werden können. Für jede Entity wurde ein abstraktes Repository-Interface erstellt, das die Methoden für Einfügen, Abrufen, Aktualisieren und Löschen von Daten vorgibt. Die konkrete Umsetzung für MongoDB erfolgt über spezialisierte Repository-Klassen und DataModels, die die Entities in Collections abbilden. So werden z. B. Person-Objekte als JSON-Dokumente in der Collection person_profiles gespeichert, validiert und bei Bedarf geladen.

4. Woche (24.03. – 28.03.2025)

Im Verlauf der vierten Woche beschäftigte ich mich mit der Implementierung des SourceArticleService und des SummarizationServiceGraph. Der SourceArticleService übernimmt die zentrale Verwaltung von Artikeln aus dem Internet und kennt dabei sowohl das Repository (SourceArticleRepository) als auch den SummarizationService, der für die Zusammenfassung der Artikel und die Extraktion relevanter Schlagworte (Tags) zuständig ist. Der Service prüft zunächst, ob ein Artikel bereits in der Datenbank vorhanden ist, verhindert so doppelte Einträge und verarbeitet neue Artikel über den

SummarizationService. Erst nach der Zusammenfassung und Generierung von Tags speichert der Service den Artikel vollständig in der Datenbank. Dabei kapselt er den direkten Datenbankzugriff, sodass externe Komponenten ausschließlich über den Service auf die Artikel zugreifen und nicht direkt mit den Repositories interagieren müssen. Darüber hinaus bietet der SourceArticleService Funktionen, um einzelne Artikel, alle Artikel eines bestimmten Feeds oder die gesamte Artikelsammlung zu entfernen sowie einzelne Artikel gezielt über Link oder ID abzurufen.

Für die Umsetzung des SummarizationServiceGraph nutzte ich LangGraph, eine auf LangChain aufbauende Bibliothek für Workflow-Modelle, um einen sogenannten Graphenworkflow zu erstellen. Ein solcher Workflow besteht grundsätzlich aus Knoten und Kanten: Knoten repräsentieren einzelne Aufgaben oder Verarbeitungsschritte, während Kanten die Reihenfolge und Abhängigkeiten zwischen diesen Schritten definieren. Um die Daten zwischen den Knoten zu verwalten, wird ein zentraler State genutzt. Technisch handelt es sich dabei um eine mit TypedDict definierte Datenstruktur, die ähnlich wie eine Klasse funktioniert: Sie legt vorab fest, welche Felder (z. B. article_content, messages) im Verlauf des Graphen gespeichert und weitergegeben werden. Die Knoten arbeiten mit Messages: SystemMessage enthält Anweisungen, wie das Modell die Aufgabe ausführen soll, HumanMessage liefert den Input des Nutzers oder den zu verarbeitenden Text, und das Modell antwortet anschließend mit einer AIMessage.

Der SummarizationServiceGraph setzt dieses Konzept konkret für die Artikelverarbeitung um. Er verwendet einen SummarizationState, der den Originaltext, die erzeugte Zusammenfassung und die Tags speichert. Lange Artikeltexte werden zuvor mit dem RecursiveCharacterTextSplitter in kleinere Abschnitte zerlegt, um die Verarbeitung durch das LLM zu erleichtern. Im Graphen sammelt der Knoten final_summary die Textabschnitte und erstellt daraus eine präzise, strukturierte Zusammenfassung auf Deutsch. Anschließend extrahiert der Knoten generate_tags aus der Zusammenfassung relevante Tags in englischer Sprache. Ein wesentlicher Teil der Arbeit bestand darin, die Prompts zu entwickeln, zu testen und kontinuierlich anzupassen, bis die Ergebnisse konsistent, inhaltlich korrekt und für die weitere Verarbeitung zuverlässig waren.

Außerdem habe ich eine Hilfsfunktion person_loader implementiert, die unstrukturierte Personendaten aus einer einfachen Textdatei verarbeitet. Die Datei bestand lediglich aus langen Zeichenketten mit Zeilenumbrüchen und Trennungen. Meine Funktion zerlegt diese Strings, ordnet die Inhalte thematisch und bereitet sie so auf, dass daraus automatisch eine Domain-Entität Person entsteht. Diese Entität umfasst Attribute wie Name, Profilbeschreibung, Interessen (als Liste) und Schreibstil. Den PersonService habe ich erweitert, sodass neue oder aktualisierte Personendaten jederzeit über den person_loader in die Datenbank hochgeladen werden können, sobald eine neue Person ihre Informationen bereitstellt.

5. Woche (31.03. – 04.04.2025)

In der fünften Woche wurde im Daily entschieden, für das BlogWriter-Projekt eine eigene UI-Oberfläche zu entwickeln. Anstatt die generierten Blogbeiträge per E-Mail zu versenden, erhalten die Nutzer:innen nun die Möglichkeit, die besten fünf Artikelvarianten direkt angezeigt zu bekommen und anschließend mit dem ausgewählten Artikel einen KI-Chat zu starten, der den Text individuell an das jeweilige Nutzerprofil anpasst.

Während mein Kollege die Umsetzung des Frontends übernahm, begann ich parallel mit der Implementierung des ArticleAssignmentService. Ziel dieses Services ist es, eingehende Artikel automatisch denjenigen Personen im System zuzuordnen, deren Interessen und Profile inhaltlich am besten passen. Dafür entwickelte ich ein Verfahren, das auf Embeddings und semantischer Ähnlichkeitssuche basiert.

Damit die Zuweisung von Artikeln zu Personen funktioniert, werden die Texte nicht nur im klassischen Sinne in der Datenbank gespeichert, sondern zusätzlich in einer lokalen Vektordatenbank (ChromaDB) abgelegt. Jeder Artikel wird zunächst als Document-Datentyp aufbereitet, der Titel, Zusammenfassung, Inhalt und Tags sowie Metadaten wie Link enthält. Beim Hinzufügen dieser Dokumente in ChromaDB wird automatisch das zuvor konfigurierte Embedding-Modell aufgerufen, sodass die Texte in Vektorrepräsentationen umgewandelt und gemeinsam mit den Metadaten in der Datenbank gespeichert werden. ChromaDB verwaltet damit sowohl die Dokumente als auch die erzeugten Embeddings und ermöglicht eine effiziente, lokale semantische Suche. Profilinformationen und Interessen einer Person werden ebenfalls vektorisiert und mit den gespeicherten Dokumentvektoren abgeglichen.

Für die eigentliche Suche probierte ich verschiedene Strategien aus, unter anderem similarity_search, similarity_search_with_score sowie max_marginal_relevance_search von ChromaDB. Die Ergebnisse waren bislang noch nicht in allen Fällen zufriedenstellend, weshalb ich dieses Thema zu einem späteren Zeitpunkt nochmals angehen musste. Insgesamt reduziert dieser Ansatz den Implementierungsaufwand, da keine separate Berechnung, Speicherung oder Synchronisierung von Embeddings erforderlich ist und die gesamte Verwaltung direkt durch ChromaDB erfolgt.

Im Vergleich zu einer klassischen relationalen Datenbank, in der lediglich Titel, Tags oder Fließtext gespeichert werden und Suchanfragen meist über exakte Schlüsselwörter oder einfache Textfilter laufen, ermöglicht eine Vektordatenbank semantische Abfragen. Das bedeutet, dass nicht nur exakte Wortübereinstimmungen gefunden werden, sondern auch inhaltlich ähnliche Texte, selbst wenn andere Formulierungen oder Synonyme verwendet werden. Für das BlogWriter-Projekt ist dies besonders wichtig, da Nutzerprofile häufig thematische Beschreibungen und Interessen in freier Textform enthalten.

Um nachvollziehen zu können, welche Artikel welchen Personen zugewiesen wurden, erweiterte ich die bestehende Datenstruktur um eine neue Entität – PersonArticlesAssignment. Parallel dazu implementierte ich die entsprechenden Repository- und Service-Klassen.

Für die Generierung des ersten LinkedIn-Posts implementierte ich den PostGenerationServiceGraph, der auf einem Graphenworkflow basiert. Dieser Graph besteht aus nur einem Knoten linkedin_post, der mithilfe des ChatOpenAI-Modells „o3-mini“ aus den Eingabedaten einen individuellen Beitrag erstellt. Dabei kommen SystemMessage und HumanMessage zum Einsatz: Die SystemMessage enthält die formalen Anweisungen und den Prompt, der Stil, Tonalität und Struktur des Posts vorgibt, während die HumanMessage den Artikelinhalt sowie Personendaten wie Name, Profil und Schreibstil übermittelt. Das Ergebnis des Knotens wird anschließend als neuer Post im Zustand gespeichert und beim Aufruf des Graphen als Zeichenkette erzeugt. Das verwendete Large Language Model (LLM) sowie Prompt können jederzeit von außen angepasst werden, ohne die Logik des Graphen selbst zu modifizieren.

6. Woche (07.04. – 11.04.2025)

In der nächsten Woche beschäftigte ich mich mit der Implementierung von ChatServiceGraph, der eine interaktive Überarbeitung bereits generierter LinkedIn-Beiträge ermöglicht. Der ChatServiceGraph ist bewusst einfach gehalten und besteht im Kern aus einem Knoten regenerate_post. Dort wird mithilfe des konfigurierten LLM ein neuer Beitrag erstellt, der sowohl die bisherigen Chat-Nachrichten als auch das Feedback der Person berücksichtigt. Durch das Zusammenspiel von SystemMessage, dem bisherigen Chatverlauf sowie einer zusätzlichen HumanMessage für Feedback wird das Sprachmodell angesteuert. Das Ergebnis ist ein überarbeiteter Beitrag in Form einer Zeichenkette sowie ein aktualisierter Chat-Verlauf, die beide gemeinsam im ChatState gespeichert werden.

Während der Implementierung des ChatServiceGraph stellte sich heraus, dass die Ergebnisse des LLM zunächst instabil waren: Die generierten Posts variierten stark in Tonalität, Struktur und Sprache. Häufig berücksichtigte das Modell nur das Feedback der Person, während Kontextinformationen aus dem Profil ignoriert wurden, oder es erwähnte wiederholt die Position der Person im Unternehmen, obwohl dies nicht gewünscht war. Um diese Probleme zu beheben, habe ich den Chat-Prompt deutlich erweitert und mit präzisen Bedingungen (Contraints) sowie umfassenden Kontextinformationen versehen. Zusätzlich testete ich verschiedene LLMs, um eine konsistente Qualität der Beiträge zu gewährleisten, ohne die Kosten unnötig zu erhöhen.

Die Entscheidung, getrennte Graphen für die Erstellung des ersten Beitrags und die spätere Überarbeitung einzusetzen, wurde bewusst gewählt. Technisch ergibt sich

daraus eine klarere Architektur: Der PostGenerationServiceGraph bleibt ein einmaliger, deterministischer Ablauf mit reinem Fokus auf Eingabedaten wie Artikel und Person, während der ChatServiceGraph einen iterativen Prozess abbildet: er nimmt den bestehenden Post als Ausgangspunkt und kombiniert es mit Chat-Verlauf und User-Feedback, um fortlaufend angepasste Versionen zu erzeugen. Diese Trennung verbessert die Wartbarkeit, da Änderungen an der Post-Erstellung den Chat nicht beeinflussen und umgekehrt. Zudem lassen sich die Graphen unabhängig erweitern oder austauschen und durch ihre Modularität einfacher testen.

Um Nutzern die interaktive Erstellung und Überarbeitung von LinkedIn-Beiträgen zu ermöglichen, implementierte ich den WritingSessionService inklusive der zugehörigen Entity WritingSession und des Repositories. Jede Person kann für jeden ihr zugewiesenen Artikel beliebig viele Sessions starten. Eine WritingSession erhält zunächst den Titel des ursprünglichen Artikels, kann jedoch später von der Person angepasst werden. Außerdem speichert die Session alle generierten Entwürfe eines Beitrags, die Chat-Nachrichten sowie das finale Ergebnis basierend auf dem Feedback der Person, sodass die Entstehung jedes Beitrags vollständig nachvollziehbar bleibt.

Beim Start einer Session wird über den PostGenerationServiceGraph der erste Beitrag erstellt. Ist die Person mit dem Beitrag nicht zufrieden, kann sie im Chat-Fenster dem KI-Bot entsprechende Anweisungen geben. Für Änderungen oder Rückmeldungen nutzt der WritingSessionService den ChatServiceGraph, der auf Basis des bisherigen Beitrags und des Feedbacks neue Versionen erzeugt.

Außerdem arbeitete ich an einem Bugfix in der Datenbankanbindung. Konkret ging es um die Speicherung der Entität PersonArticlesAssignment in MongoDB. Ursprünglich war lediglich ein eindeutiger Index auf das Feld person definiert, was jedoch dazu führte, dass pro Person nur ein Artikel zugewiesen werden konnte. Um sicherzustellen, dass jede Kombination aus Person und Artikel nur einmal gespeichert wird, implementierte ich einen zusammengesetzten Unique-Index über beide Felder. Dadurch wird verhindert, dass identische Zuweisungen mehrfach in der Datenbank landen, während gleichzeitig mehrere unterschiedliche Artikel pro Person möglich bleiben.

7. Woche (14.04. – 18.04.2025)

Zu Beginn der siebten Woche trat das Problem auf, dass die vom LLM generierten Nachrichten in der UI nicht nicht in Echtzeit angezeigt (nicht gestreamt) wurden: Das Ergebnis war erst sichtbar, nachdem der gesamte Post vollständig vom KI-Bot erzeugt worden war. Dies führte zu langen Wartezeiten und einer weniger interaktiven Benutzererfahrung. Um das Problem zu lösen, integrierte ich Streaming über die Callback-Mechanismen von LangChain. Konkret übergab ich beim Aufruf des ChatServiceGraph über die asynchronische Funktion ainvoke eine Liste von Callback-Handlern, darunter den AsyncIteratorAnswerCallbackHandler. Dieser empfängt die

generierten Tokens und stellt sie über einen asynchronen Iterator bereit. So können die Nachrichten Schritt für Schritt in der Chat-Oberfläche angezeigt werden, was die Interaktivität deutlich verbessert.

Nachdem die Grundfunktionen weitgehend implementiert waren, beschäftigte ich mich mit dem ArticleAssignmentService und überprüfte zusätzlich verschiedene Retriever-Methoden, um die Zuweisung von Artikeln zu Personen zu optimieren.

Dabei betrachtete ich unter anderem den SelfQueryRetriever. Dieser Ansatz erlaubt es, dass das LLM selbstständig semantische Suchanfragen generiert, um relevante Dokumente aus der Vektordatenbank zu identifizieren. Der Retriever berücksichtigt sowohl den Inhalt der Dokumente als auch deren Metadaten wie Tags oder Datum. Dadurch kann er komplexe Zusammenhänge erkennen und passende Artikel auch bei unterschiedlichen Formulierungen zuverlässig finden, während der Bedarf an manueller Filterlogik reduziert wird. Insgesamt bietet dieser Ansatz eine flexible Möglichkeit, die inhaltliche Passung von Artikeln zu Personen zu verbessern. Allerdings hängt die Genauigkeit stark vom LLM ab, die Ergebnisse können inkonsistent sein, für große Datensätze entstehen Performance-Herausforderungen, und jede Abfrage verursacht Kosten, da das LLM jedes Mal aufgerufen wird.

Als Nächstes testete ich zusätzlich den ContextualCompressionRetriever. Dieser Retriever wendet vor der Rückgabe eine Pipeline von Transformationen an, darunter Clustering, Redundanzfilterung und Neuordnung der Dokumente. Ziel war es, irrelevante oder doppelte Inhalte zu entfernen und die Reihenfolge der Ergebnisse zu optimieren.

Schließlich probierte ich die Kombination verschiedener Retriever mit dem MergerRetriever aus. Hierbei sollten die Vorteile der einzelnen Ansätze gebündelt werden: Der SelfQueryRetriever sollte komplexe Zusammenhänge erkennen, während der CompressionRetriever irrelevante Inhalte aussortierte. Zusätzlich wurden einfache Embedding- oder Keyword-basierte Retriever integriert, um klassische semantische Ähnlichkeit zu berücksichtigen. In der Praxis führte diese Merger-Strategie jedoch nicht zu zufriedenstellenden Ergebnissen. Teilweise wurden relevante Artikel nicht priorisiert oder durch die Kompression zu stark verändert.

Am Ende der Woche fand ein weiteres Meeting mit dem DEV-Management-Team statt. Dabei wurde der aktuelle Zwischenstand vorgestellt, einschließlich der bisherigen Änderungen und Erkenntnisse. Im Rahmen der Besprechung wurde beschlossen, dass ein Artikel nur einer Person zugewiesen werden darf. Hintergrund war, dass mehrere Personen dasselbe Interesse haben könnten und dadurch denselben Artikel erhalten würden. Um zu vermeiden, dass mehrere Personen denselben LinkedIn-Beitrag generieren, wurde diese Regel eingeführt. Außerdem wurde festgestellt, dass der Chat bei zusätzlichen Fragen aktuell jedes Mal einen neuen Beitrag generiert. Dieses Verhalten soll künftig angepasst werden, sodass bestehende Entwürfe weiterbearbeitet werden, statt neue Posts zu erzeugen.

8. Woche (21.04. – 25.04.2025)

In der achten Woche beschäftigte ich mich mit der Weiterentwicklung des ArticleAssignmentService und der Verbesserung der Zuweisung von Artikeln. Kernpunkte waren die Implementierung einer Pipeline, die mehrere Ansätze zur semantischen und inhaltlichen Suche miteinander kombiniert. Zunächst implementierte ich einen Custom-Retriever auf Basis von ChromaDB, der eine Score-basierte Suche nutzt. Standardmäßig verfügbare Retriever bieten diese Funktion nicht, wodurch es schwierig wäre, relevante Dokumente flexibel zu priorisieren. Mit dem Score-Retriever lassen sich die Ergebnisse nach Relevanz filtern, die Gewichtung kann dabei jederzeit angepasst werden, was die Genauigkeit der Artikelzuweisungen deutlich verbessert. Darauf aufbauend setzte ich einen EnsembleRetriever ein, der mehrere Retriever – beispielsweise custom RetrieverWithScores mit OpenAI-Embeddings und BM25Retriever – gewichtet zusammenführt. Zusätzlich wurde eine DocumentCompressorPipeline integriert, die Redundanzen entfernt und die relevantesten Textteile priorisiert. Abgeschlossen wird der Prozess durch einen ContextualCompressionRetriever, der die Dokumente filtert und sortiert, bevor sie dem LLM für die Artikel-Zuweisung übergeben werden. Dieses Setup erlaubt eine flexible, skalierbare und genauere Zuweisung von Artikeln an Personen.

Zusätzlich passte ich die Zuweisung so an, dass jeder Artikel stets nur einer Person aktiv zugeordnet ist, selbst wenn er für mehrere Personen relevant wäre – in diesem Fall erhält ihn die Person mit dem höchsten Score. Dafür implementierte ich die Entität ArticleAssignmentTracking mit dem zugehörigen Repository, das Änderungen an einem Artikel verfolgt. Dabei wird dokumentiert, welche Personen potenziell passen, wer den Artikel erhalten hat und wer einen Vorschlag gelöscht hat. Wird ein Vorschlag von einer Person entfernt, kann der Artikel automatisch der nächsten passenden Person zugewiesen werden. Gleichzeitig wird sichergestellt, dass ein gelöschter Artikel bei späteren Zuweisungen nicht erneut derselben Person vorgeschlagen wird. Alle Aktionen werden protokolliert, sodass die Historie der Zuweisungen jederzeit nachvollziehbar bleibt.

9. Woche (28.04. – 02.05.2025)

In der neunten Woche optimierte ich den ChatServiceGraph durch einen Agenten, der entscheidet, ob eine Nutzeranfrage die Regeneration eines LinkedIn-Beitrags oder die Beantwortung einer allgemeinen Frage erfordert. Technisch wird dies über einen DecisionNode im Graphen umgesetzt, der den aktuellen Chatverlauf, den letzten Post und das Feedback der Person auswertet. Je nach Entscheidung leitet der Graph die Verarbeitung an den regenerate_post-Knoten oder einen response-Knoten weiter. Beide Knoten nutzen SystemMessage- und HumanMessage-Strukturen, um das LLM zu steuern.

Darüber hinaus führte ich ein Refactoring der Prompts durch. Anstatt einfache Strings für Prompts zu verwenden, setzte ich nun auf das flexiblere Konstrukt ChatPromptTemplate von langchain. Dies ermöglicht eine klarere Trennung zwischen System- und Nutzernachrichten sowie die Verwendung dynamischer Platzhalter für Personendaten, Schreibstil oder den aktuellen Artikel. Ein weiterer Vorteil war die Anpassung des Outputs: Die generierten Beiträge werden nun in einem speziell markierten Abschnitt (<draft>...</draft>) ausgegeben. Dadurch lassen sie sich in der Oberfläche besser darstellen und vom restlichen Chat unterscheiden. Nutzerinnen und Nutzer sehen den Post so direkt als separaten Entwurf.

Zusätzlich passte ich den Prozess zum Einfügen neuer Artikel ins Repository an. Bisher wurden Artikel direkt in die Datenbank geschrieben, noch bevor Summary und Tags generiert waren. Trat während der Generierung für eine gesamte Artikelliste ein Fehler auf, entstanden dadurch unvollständige Einträge. Um dies zu vermeiden, erfolgt die Verarbeitung nun schrittweise: Für jeden Artikel werden zunächst die Summary und die Tags erzeugt und erst danach wird er in die Datenbank übernommen. Anschließend wird mit dem nächsten Artikel fortgefahren.

Außerdem wurde die Anbindung des LLM von der OpenAI-API auf OpenRouter umgestellt, wodurch über LangChain nun verschiedene Modelle flexibel genutzt werden können. Nach Tests entschied ich mich für anthropic/claude-3-haiku für die Artikelsummaries und anthropic/claude-3.5-haiku-20241022 für den ersten LinkedIn-Post sowie den Chat. Diese Kombination gewährleistet eine gute Balance zwischen Qualität, Geschwindigkeit und Kosten.

10. Woche (05.05. – 09.05.2025)

In der zehnten Woche stand zunächst ein Refactoring im Vordergrund. Nachdem die neue Funktionalität rund um das ArticleAssignmentTracking eingeführt worden war, mussten sämtliche Services und Repositories überprüft, an die Änderungen angepasst und bestehende Bugs behoben werden. Ein wesentlicher Teil der Arbeit bestand außerdem darin, entsprechende Unit Tests zu erstellen. Ziel war es, sicherzustellen, dass die neuen Mechanismen zur Artikelzuweisung korrekt mit den bestehenden Prozessen interagieren.

Schließlich hatte ich die Aufgabe alte Artikel aus den RSS-Feed in der Datenbank zu vermeiden, um die Auswahl auf aktuelle Themen zu beschränken. Dafür implementierte ich einen flexibel gestalteten ArticlesFilter. Der Filter überprüft das Veröffentlichungsdatum eines Artikels und lässt nur Beiträge zu, die innerhalb eines festgelegten Zeitraums liegen (standardmäßig die letzten 14 Tage, diese Frist lässt sich jedoch jederzeit einfach anpassen). Artikel, die älter sind, werden übersprungen. Dadurch wird sichergestellt, dass vor der Generierung von Tags und der Aufnahme in die Datenbank nur aktuelle Artikel verarbeitet werden.

11. Woche (12.05. – 16.05.2025)

In der elften Woche erweiterte ich das System um mehrere zentrale Funktionalitäten. Zunächst implementierte ich Methoden, mit denen nicht zugewiesene Artikel aus der Datenbank abgefragt werden können. Diese Grundlage war notwendig, um veraltete oder dauerhaft ungenutzte Artikel gezielt identifizieren und anschließend aus der Datenbank löschen zu können. Damit wird langfristig verhindert, dass unnötige Datenbestände anwachsen und die Datenbank mit irrelevanten Inhalten gefüllt bleibt.

Darüber hinaus ergänzte ich eine neue Funktion für Personenprofile: Sobald eine Person ihr Profil erstellt hat, steht nun ein Button „Vorschläge generieren“ zur Verfügung. Über diesen Button können direkt Artikel aus der Datenbank ausgewertet und passende Vorschläge für die jeweilige Person erzeugt werden. Bisher erfolgte die Bereitstellung ausschließlich über die automatische Zuweisung, die standardmäßig zweimal pro Woche durchgeführt wird. Ab sofort hat jede Person die Möglichkeit, individuell und on-demand passende Artikelvorschläge zu erhalten.

In einem Daily Meeting wurde entschieden, das System zu erweitern, um mehr Flexibilität zu ermöglichen. Künftig sollte jede Person ihre eigenen Feeds verwalten können, da die thematischen Interessen der Nutzer sehr unterschiedlich sind (z. B. Mobile Entwicklung, Marketing oder Management). Dafür wurde die bisher im SourceArticle hinterlegte einfache Source-URL in eine eigenständige Feed-Entität überführt. Es gibt nun zwei Arten von Feeds: öffentliche Feeds, die zentral im FeedRepository gespeichert und von allen Personen genutzt werden können, sowie custom Feeds, die direkt im Personenprofil in der PersonMongoDB abgelegt werden. Jede Person erhält eine Übersicht, in der ihre privaten und die allgemeinen öffentlichen Feeds getrennt angezeigt werden. Um diese Funktionalität umzusetzen, entwickelte ich den FeedService, der die Validierung von RSS-Links, das Laden und Parsen der Artikel sowie die spätere Speicherung in der Datenbank übernimmt. Beim Hinzufügen eines neuen Feeds werden die Artikel zunächst nur per feedparser schnell geladen, sodass die Person eine Vorschau der Inhalte erhält, ohne die Datenbank zu belasten. Erst wenn ein Feed tatsächlich gespeichert wird, erfolgt im Hintergrund das vollständige Laden über den RSSFeedLoader von LangChain, der die Artikel inklusive vollständiger Inhalte verarbeitet und als SourceArticle-Objekte in der Datenbank ablegt. Zusätzlich unterstützt der Service das Entfernen von Feeds samt zugehörigen Artikeln.

12. Woche (19.05. – 23.05.2025)

In der zwölften Woche beschäftigte ich mich mit der Anpassung des Retrieval-Prozesses. Bisher konnten beim Matching auch Artikel aus Quellen berücksichtigt werden, die für eine Person eigentlich gar nicht relevant waren. Nach der Einführung von öffentlichen und privaten Feeds musste daher sichergestellt werden, dass Artikel ausschließlich aus den Feeds einer Person (public + custom) stammen. Dafür

erweiterte ich den bestehenden RetrieverWithScores, der auf einer Vektor-Datenbank basiert. Neben der semantischen Ähnlichkeitssuche über Embeddings wurde ein Source-Filter auf Basis der Metadaten ergänzt. Dieser überprüft für jedes Dokument, ob dessen source-Attribut in der Liste der gültigen RSS-Links enthalten ist. Nur wenn diese Bedingung erfüllt ist, wird das Dokument weiterverarbeitet und anhand des Scores bewertet. Parallel dazu entwickelte ich einen FilteredBM25Retriever, der eine klassische BM25-Suche auf Textbasis durchführt. BM25 berücksichtigt dabei, wie oft ein Suchwort im Dokument vorkommt, wie selten es in der gesamten Sammlung ist und gleicht Unterschiede in der Dokumentlänge aus. So werden relevante Textpassagen präzise gefunden – auch ohne Embeddings. Vor der Berechnung werden alle Dokumente über die gleiche Filterlogik auf die erlaubten RSS-Quellen eingeschränkt. Erst danach wird das Ranking der Top-k Treffer bestimmt. Durch diese Kombination von semantischer Suche (Vektor-basiert) und lexikalischer Suche (BM25), jeweils ergänzt durch eine Feed-Filterung, konnte die Präzision der Zuweisung deutlich verbessert werden. Jede Person erhält damit nur Artikel aus den für sie freigegebenen Feeds.

Zusätzlich wurde die Logik zur Artikelzuweisung angepasst, sodass ein Artikel nun gleichzeitig in den Vorschlägen mehrerer Personen erscheinen kann. Bisher wurde jeder Artikel ausschließlich einer Person zugewiesen, um Überschneidungen zu vermeiden. Mit der neuen Umsetzung können nun mehrere Nutzer denselben Artikel erhalten, wenn er für ihre thematischen Interessen relevant ist. In der Benutzeroberfläche wird dabei angezeigt, dass auch andere Personen Zugriff auf denselben Artikel haben. Der Artikel bleibt aktiv und kann für eine WritingSession genutzt werden. Sobald jedoch eine Person eine Session mit dem Artikel startet, wird er für alle anderen Nutzer blockiert, die stattdessen einen Hinweis sehen, dass der Artikel bereits verwendet wird. Sollte eine Person alle Sessions mit diesem Artikel löschen, wird der Vorschlag wieder für alle relevanten Nutzer freigegeben. Zur Umsetzung dieser Funktionalität wurden die ArticleAssignment- und WritingSession-Services angepasst, sodass sowohl die Statusinformationen als auch die Blockierungen zuverlässig verwaltet werden.

13. Woche (26.05. – 30.05.2025)

In der dreizehnten Woche habe ich die IDs für die Entitäten von MongoDB-typischen ObjectIDs auf UUIDs umgestellt. Dadurch werden die Daten nun mit universell einsetzbaren Identifikatoren gespeichert, was das System flexibler und weniger abhängig von der Datenbanktechnologie macht und einen späteren Wechsel von MongoDB erleichtert. Gleichzeitig sorgt die Umstellung dafür, dass Backend und Frontend mit denselben ID-Typen arbeiten, ohne zusätzliche Konvertierungen. Um die UUID-Verwendung zentral zu regeln, habe ich eine Basisklasse BaseDataModel erstellt, die alle Entitäten erben. Diese Klasse enthält die Pydantic-Konfiguration, inklusive json_encoders, sodass jede UUID automatisch in einen String serialisiert

wird. Dadurch muss die ID-Logik nicht in jeder einzelnen Entität wiederholt werden und bleibt konsistent im gesamten System.

Außerdem fiel mir auf, dass bei gemischten Entitäten wie PersonArticleAssignment die Objekte Person oder Article zwar in einer eigenen MongoDB-Collection gespeichert sind, in anderen Collections jedoch lediglich als Kopien vorliegen. Änderungen an einem Objekt in einer Collection wirken sich daher nicht automatisch auf die Kopien in anderen Collections aus. Dies stellt zwar keinen Fehler dar, kann jedoch zu Inkonsistenzen führen. Um dieses Problem zu lösen, werden nun in den MongoDB-Repositories nur noch die IDs der referenzierten Entitäten gespeichert. Beim Einfügen (insert) müssen daher explizit die IDs aus den Objekten extrahiert werden und beim Abrufen (get) werden die vollständigen Objekte über Aggregation-Pipelines wieder zusammengeführt. Dadurch lassen sich Daten konsistent referenzieren, Änderungen bleiben nachvollziehbar und die Integration in bestehende Pipelines funktioniert zuverlässig. Für alle MongoDB-Repositories implementierte ich zudem Tests mit pytest, um die korrekte Funktion sämtlicher CRUD-Operationen sicherzustellen und die Stabilität der Implementierungen zu gewährleisten.

14. Woche (02.06. – 06.06.2025)

In dieser Woche arbeitete ich weiter an der Verarbeitung von Artikeln aus Feeds. Es wurde die Möglichkeit umgesetzt, dass eine Person direkt mit einem Artikel aus ihrem individuellen oder einem öffentlichen Feed eine Session starten kann. Dafür entwickelte ich den entsprechenden Usecase AssignArticleFromFeedUsecase, der die notwendige Logik zur Zuweisung abbildet. Besonders wichtig war dabei, sicherzustellen, dass Artikel nicht mehrfach in die Datenbank eingefügt werden. Befand sich ein Artikel bereits im System, wurde er nicht erneut gespeichert. Zusätzlich musste geprüft werden, ob andere Personen bereits eine Session mit demselben Artikel gestartet hatten. In diesem Fall erhielt der Nutzer den Hinweis, dass der Artikel momentan in Verwendung ist und zu einem späteren Zeitpunkt erneut verfügbar sein könnte, während er gleichzeitig in den persönlichen Vorschlägen erschien. War ein Artikel hingegen noch frei, konnte unmittelbar eine Session gestartet werden.

Ein weiterer Sonderfall ergab sich im Umgang mit älteren Artikeln aus Feeds. Automatisch geladene ältere Artikel wurden standardmäßig aussortiert, um die Aktualität zu gewährleisten. Wenn ein Nutzer jedoch bewusst einen älteren Artikel aus einem Feed auswählen wollte, wurde dies zugelassen. In diesem Fall erhielt der Artikel einen Score von 0.0, was die Nachverfolgbarkeit solcher gezielt gewählten Artikel ermöglichte.

Im Zusammenhang mit diesen Erweiterungen mussten außerdem einige Fehler behoben werden. Unter anderem trat das Problem auf, dass Sessions in seltenen Fällen doppelt erstellt wurden, wenn zwei Nutzer nahezu zeitgleich denselben Artikel

auswählten. Dies führte zu Inkonsistenzen in der Zuweisungslogik und wurde durch zusätzliche Prüfungen beim Anlegen einer Session behoben.

Darüber hinaus erfolgte in dieser Phase ein Wechsel vom bisherigen Dependency-Management mit Poetry hin zu uv, einem leichtgewichtigen Tool zur Verwaltung von Python-Paketen und virtuellen Umgebungen. Im Vergleich zu Poetry ist uv deutlich schneller bei der Installation und Verwaltung von Abhängigkeiten, bietet eine einfachere Handhabung und lässt sich unkompliziert in Container-Umgebungen integrieren. Im Anschluss führte mein Kollege Docker in das Projekt ein. Dadurch konnte ich die Container auch lokal ausprobieren und mich mit ihrer Funktionsweise vertraut machen. Insgesamt ergaben sich vier Container, die unterschiedliche Komponenten des Systems abbildeten: ein Container für die MongoDB, einer für die VectorDB, ein Webserver sowie ein Backendserver auf Basis von FastAPI.

15. Woche (09.06. – 13.06.2025)

In der fünfzehnten Woche implementierte ich einige zusätzliche Hilfsfunktionen im Zusammenhang mit der Zuweisung von Artikeln aus Feeds. Ich stellte sicher, dass ein Artikel nur genutzt werden konnte, wenn er noch frei war oder bereits von der jeweiligen Person verwendet wurde, wodurch Mehrfachnutzung verhindert wurde. Einen Bug in der Writing-Session, der fehlerhafte Zuweisungen verursachte, behob ich ebenfalls. Außerdem passte ich die Rückgabewerte der Repository-Methoden an, sodass die automatisierten Tests zuverlässig funktionierten. Die Session-Verwaltung erweiterte ich so, dass beim Löschen der letzten Session zu einem Artikel automatisch die Tracking-Informationen aktualisiert wurden und der Artikel wieder als frei verfügbar markiert war.

Zusätzlich implementierte ich einen weiteren Usecase zum Archivieren von Artikeln. Wenn ein Nutzer einen Artikel tatsächlich für einen Post verwenden wollte, konnte er auf einen entsprechenden Button klicken. Daraufhin verschwand der Artikel aus den Vorschlägen aller Nutzer und alle zugehörigen Sessions wurden automatisch entfernt. Dadurch wurde sichergestellt, dass der Artikel nach der Veröffentlichung nicht mehr mehrfach bearbeitet wurde. Außerdem implementierte ich Funktionen, die alle archivierten Artikel eines Nutzers sammeln, sodass der User diese jederzeit einsehen kann.

16. Woche (16.06. – 20.06.2025)

In der sechzehnten Woche implementierte ich einen Use Case zum Entfernen von Artikeln aus der Datenbank, die nicht mehr verwendet werden. Ziel war es, alte, nicht genutzte Artikel automatisch zu identifizieren und sowohl aus der Datenbank als auch aus dem VectorStore zu löschen. Dabei filterte ich zunächst alle vorhandenen Artikel und prüfte, welche davon nicht mehr relevant waren. Anschließend entfernte ich diese

Artikel aus dem VectorStore und aktualisierte parallel die Zuweisungen und Tracking-Daten in den Vorschlagslisten der Nutzer.

Im Anschluss führte ich ein umfangreiches Refactoring durch, da der ArticleAssignmentService zu groß geworden war. Im Application-Layer wurde anstelle des alten Services die abstrakte SuggestionsService definiert, die die Schnittstelle für zentrale Funktionen wie Artikelzuweisung, Tracking, Verfügbarkeitsprüfung und Archivierung vorgibt. Die konkrete Implementierung erfolgte über SuggestionsRetrieverService im Adapters/Services-Layer und kann bei Bedarf flexibel ersetzt werden. Sie koordiniert spezialisierte Services: SourceArticleService verwaltet Artikel, PersonService liefert Personenprofile, AssignmentService erstellt und entfernt Zuweisungen, AssignmentTrackingService pflegt das Tracking und archiviert Artikel, FeedRepository liefert Public-Feeds, VectorStoreAdapter speichert Artikel als Dokumente im Vektorspeicher, und RetrieverService kombiniert verschiedene Retriever wie RetrieverWithScores und FilteredBM25Retriever, um relevante Artikel effizient abzurufen. Diese Struktur trennt Verantwortlichkeiten klar und vermeidet direkte Abhängigkeiten zwischen Modulen.

Zusätzlich wurden entsprechende Tests für die Services implementiert. Dabei kamen pytest und MagicMock von unittest zum Einsatz, um einzelne Services zu isolieren und gezielt zu prüfen, ohne auf externe Systeme oder Daten angewiesen zu sein.

17. Woche (23.06. – 27.06.2025)

In der siebzehnten Woche wurden weiter Refactorings durchgeführt, insbesondere am FeedService, um die Verantwortlichkeiten weiter zu trennen. Der Service koordiniert die Verwaltung von Public- und Custom-Feeds, während der FeedParser die Validierung, das Parsen und die Aufbereitung der RSS-Daten übernimmt. Methoden im FeedService ermöglichen das Hinzufügen oder Entfernen von Custom-Feeds für Personen, die Pflege von Public-Feeds sowie das Abrufen, Aktualisieren oder Löschen einzelner Feeds. Die eigentliche Artikelverarbeitung übernimmt der FeedArticleLoaderService, der Artikel aus den RSS-Feeds lädt, nach Aktualität und Relevanz filtert und über den SourceArticleService speichert.

Außerdem wurden die Funktionalitäten weiter in Usecases ausgelagert und bestehende Services entsprechend angepasst. So sorgt der GetUnusedSuggestionsUsecase dafür, dass Personen nur Vorschläge erhalten, die noch nicht in einer abgeschlossenen Writing-Session verwendet wurden. Hierbei werden Artikel aus dem AssignmentService mit den bereits bearbeiteten Artikeln aus dem WritingSessionRepository abgeglichen, sodass nur nicht genutzte Vorschläge zurückgegeben werden.

Zudem unterstützte ich in der Woche meine Kollegen beim Firmen-Kontakttag an der Hochschule. Dabei stellte ich die Firma vor und beantwortete Fragen der Studierenden, um Einblicke in unsere Arbeitsbereiche und Projekte zu geben.

18. Woche (30.06. – 03.07.2025)

In Woche 18 testete ich alle Funktionen noch einmal gründlich. Dabei fiel ein Fehler auf: Ich hatte zuvor eine hartkodierte Nachricht der KI als erste Nachricht jeder neuen Session hinzugefügt, die den User begrüßte und den ersten generierten Beitrag anzeigte. Durch ein Cache-Problem wurde diese Nachricht bei jeder neuen Session mehrfach im Chat dupliziert. Das Problem entstand, weil Listen in Python mutable Objekte sind – wenn sie als Attribut in einer Entität einmal definiert werden, können sie bei mehreren Instanzen geteilt werden. Das Problem konnte ich beheben, indem ich die Chat-Liste beim Erstellen jeder neuen Session explizit als leere Liste initialisierte, sodass jede Session ihre eigene unabhängige Liste erhielt.

Im Anschluss fand ein weiteres Meeting mit dem DEV-Management-Team statt. Dabei wurde die finale Version des Projekts vorgestellt, inklusive aller neuen Features und Verbesserungen. Die Präsentation wurde vom Team positiv aufgenommen, die Kollegen waren mit der Funktionalität zufrieden und wollten die Version direkt selbst testen. Daraufhin wurde die Freigabe erteilt, die BlogWriter-Projekt Version 1.0 auf Azure zu deployen.

19. Woche (24.07. – 25.07.2025)

In der neunzehnten Woche erhielt ich in einem Meeting mit meinem Teamleiter die Aufgabe, mich in die für mich neue Technologie Kotlin sowie in Kotlin Multiplatform Compose einzuarbeiten. Ziel war es, mir ein fundiertes Verständnis anzueignen und darauf aufbauend eine interne Schulung für meine Kollegen vorzubereiten. Da insbesondere Compose in aktuellen Kundenanfragen zunehmend nachgefragt wird, bildete dies den zentralen Teil meiner Einarbeitung.

Zunächst richtete ich die Arbeitsumgebung ein. Dazu installierte ich IntelliJ IDEA und Android Studio inklusive der erforderlichen SDKs. Obwohl Compose in beiden IDEs unterstützt wird, entschied ich mich für die weitere Arbeit überwiegend für Android Studio, da es den Entwicklungsprozess aus meiner Sicht komfortabler gestaltete.

Danach arbeitete ich mich in die grundlegenden Sprachkonzepte von Kotlin ein. Ich begann mit der allgemeinen Syntax, die sich an Java orientiert, jedoch kompakter ist. Ein wichtiges Konzept ist Null-Safety, bei dem Variablen standardmäßig nicht null sein dürfen, was viele NullPointerExceptions verhindert. Ich lernte, wie man optionale Typen mit ? (nullable Typ) definiert, sichere Aufrufe mit ?. (Safe Call) durchführt, den Elvis-Operator ?: für Default-Werte verwendet, falls eine Variable null ist, und in Ausnahmefällen den Not-Null-Assertion-Operator !! einsetzt. Zusätzlich setzte ich den Safe-Cast-Operator as? ein, der ein Objekt auf einen bestimmten Typ castet und null zurückgibt, wenn der Cast nicht möglich ist, sowie ?.let, um Codeblöcke nur dann auszuführen, wenn eine Variable nicht null ist.

Ein weiterer Fokus lag auf dem Unterschied zwischen val und var: val definiert eine unveränderliche Variable (vergleichbar mit final in Java), während var veränderlich ist. Außerdem enthält Kotlin sogenannte Smart Casts, die es erlauben, Variablen nach einer Typprüfung automatisch in den richtigen Typ zu casten, und String Templates, die es ermöglichen, Variablen direkt in Strings einzubetten.

Außerdem unterstützt Kotlin funktionale Programmierung und bietet dafür spezielle Funktionen: Lambda Expressions für kompakte anonyme Funktionen, Extension Functions, um bestehenden Klassen neue Methoden hinzuzufügen, sowie Inline Functions, die Funktionsaufrufe zur Laufzeit optimieren.

Im objektorientierten Bereich bietet Kotlin neben normalen Klassen auch spezielle Konstrukte, die bestimmte Anwendungsfälle erleichtern. Data classes dienen zur Darstellung einfacher Datenobjekte und erzeugen automatisch Methoden wie equals(), hashCode() und toString(), was praktisch ist, wenn keine komplexe Logik benötigt wird. Sealed classes und Sealed Interfaces erlauben eine geschlossene Hierarchie, bei der nur eine festgelegte Anzahl von Unterklassen existiert – nützlich, um klar definierte Typen zu modellieren und alle Fälle z. B. in when-Ausdrücken abzudecken. Abstract classes dienen als Basisklassen mit teilweise implementierten Methoden, Enum classes definieren aufzählbare Typen mit festen Konstanten, object ermöglicht Singletons, und Companion Objects stellen statische Mitglieder innerhalb einer Klasse bereit.

Um die Konzepte praktisch nachzuvollziehen, entwickelte ich kleine Code-Snippets und testete verschiedene Sprachfeatures direkt. So konnte ich die Besonderheiten von Kotlin gezielt nachvollziehen und erste Erfahrungen für die spätere Umsetzung in größeren Projekten sammeln.

20. Woche (28.07. – 01.08.2025)

In der zwanzigsten Woche setzte ich mir das Ziel, Kotlin in Kombination mit Spring Boot praktisch einzusetzen. Als Lernprojekt entschied ich mich für die Entwicklung einer kleinen Notizen-App, bei der ein vollständiges Backend mit Datenbankanbindung umgesetzt werden sollte. Zurzeit ermöglicht Android Studio ein Multiplatform Projekt nur mit Ktor als Server zu erstellen. Darüber hinaus generierte ich das Projekt über den offiziellen “Spring Initializr” und wählte dabei Gradle mit Kotlin DSL als Build-System sowie Kotlin als Programmiersprache. Nach dem Import in Android Studio wurde der Build automatisch erstellt, sodass die Entwicklungsumgebung direkt einsatzbereit war. Die zentrale Datei für die Projektkonfiguration ist build.gradle.kts, in der Plugins, Versionen und Dependencies verwaltet werden. Die Einstiegsklasse Application.kt enthält den Main-Entry-Point für Spring Boot.

Das Projekt gliederte ich in Pakete: controllers für REST-Endpunkte und DTOs, services für die Geschäftslogik, database für das Datenmodell Note und das MongoDB-Repository sowie mappers für die Transformation zwischen den Schichten.

Die Datenbank realisierte ich mit einer lokalen MongoDB-Instanz, auf die CRUD-Operationen über Spring Data direkt ausgeführt werden konnten. Um die Funktionsweise zu überprüfen, testete ich die REST-Endpunkte mit Thunder Client und konnte erfolgreich neue Notizen anlegen, bestehende ändern, abrufen und löschen.

Insgesamt konnte ich feststellen, dass Kotlin und Java in Verbindung mit Spring Boot sehr ähnlich aufgebaut sind. Viele Konzepte und Annotationen wie z.B. @Service oder @RestController sind identisch, was den Lernprozess deutlich erleichterte. Allerdings hatte ich am Anfang Schwierigkeiten mit der ungewohnten Syntax von Kotlin – zum Beispiel bei Funktionen wie fun Note.toResponseDTO(). Solche Extension Functions sind zwar kompakter, erfordern aber eine andere Denkweise als in Java und wirken anfangs weniger intuitiv.

21. Woche (04.08. – 08.08.2025)

In der einundzwanzigsten Woche beschäftigte ich mich mit Compose Multiplatform, um die Benutzeroberfläche für meine geplante Notizen-App umzusetzen. Compose Multiplatform erweitert das Jetpack Compose Toolkit von Google und nutzt die gleichen APIs, Composables und Modifier wie bei Android-Apps. Dadurch kann der UI-Code in Kotlin geschrieben und größtenteils plattformübergreifend genutzt werden. Über Android Studio konnte ich direkt ein Projekt anlegen, das für Android, iOS und Desktop (Web wurde noch als Beta-Version angeboten) lauffähig ist. Nach dem Build sind die Pakete androidMain, commonMain, iosMain und jvmMain (für Desktop) sichtbar. Der Code im commonMain-Paket bildet die gemeinsame Basis für alle Plattformen, während spezifische Funktionen oder Layouts gezielt in den jeweiligen Plattformpaketen implementiert werden können. So ermöglicht Compose Multiplatform, Logik und UI weitgehend zu teilen und gleichzeitig plattformspezifische Anpassungen vorzunehmen.

Zu Beginn verschaffte ich mir einen Überblick über die grundlegenden Konzepte von Compose Multiplatform anhand der offiziellen JetBrains Dokumentation zu Compose Multiplatform sowie der Android Developer Dokumentation zu Jetpack Compose. Anhand der Dokumentationen konnte ich die Konzepte praktisch anwenden, indem ich kleine UI-Elemente und Composable Functions implementierte. Dabei setzte ich mich zudem mit Modifiers auseinander, um Layout, Styling und Verhalten der UI-Elemente anzupassen, und erlernte die grundlegenden Layout-Prinzipien, um einfache Oberflächen zu gestalten.

Um weitere Konzepte von Compose Multiplatform kennenzulernen, begann ich direkt mit der Umsetzung meines Notizen-App-Prototyps. Ich entschied mich, die App als MVVM-basiertes Projekt aufzubauen – basierend auf praktischen Erfahrungen aus meinem Studium mit Flutter. Die App enthält zwei Hauptscreens: eine Listenansicht für alle Notizen und eine Detailansicht zur Bearbeitung einzelner Notizen. Für die Paketstruktur orientierte ich mich an den offiziellen Kotlin Multiplatform Tutorials. So

übernahm ich Paketnamen und Layering, um die App sauber nach MVVM zu organisieren und gleichzeitig features-orientiert aufzubauen. Die zentralen Pakete gliedern sich wie folgt: app, core (für innerhalb der App wiederverwendbare Klassen und Hilfsfunktionen), di (für die Konfiguration der Dependency Injection) und das note-Paket, das in data, domain und presentation unterteilt ist, um Datenquellen, Geschäftslogik und UI-Funktionalität klar voneinander zu trennen.

Ich begann mit der Implementierung der Domain-Schicht, die zentrale Schnittstellen und Geschäftslogik für die Notizen kapselt. Dabei habe ich gelernt, dass neue oder experimentelle Features in Compose mit @OptIn markiert werden müssen, um sie verwenden zu können.

22. Woche (11.08. – 15.08.2025)

In der zweiundzwanzigsten Woche konzentrierte ich mich auf die Umsetzung der Presentation-Schicht der Notizen-App, speziell auf die Listenansicht der Notizen. Dabei erstellte ich Composable-Komponenten für einzelne Notizen, die Notizenliste selbst und eine Komponente zur Suche innerhalb der Notizenliste.

Im Sinne des MVVM-Musters übernimmt das ViewModel (NoteListViewModel) die Verwaltung des States (NoteListState) und die Verarbeitung von Nutzerinteraktionen (NoteListAction), während die UI-Komponenten in den Composable Functions ausschließlich für die Darstellung zuständig sind. Die Composables werden durch zwei zentrale Funktionen strukturiert: ScreenRoot (NoteListScreenRoot) und Screen (NoteListScreen). ScreenRoot dient als Einstiegspunkt von außen, hier wird das ViewModel über koinViewModel() mittels Dependency Injection eingebunden. Die Funktion stellt sicher, dass das ViewModel korrekt instanziiert, bereitgestellt und wiederverwendet wird. Screen ist privat und konzentriert sich ausschließlich auf die Darstellung auf Basis des übergebenen States und der Actions.

Um die Liste in den Emulatoren anzuzeigen, habe ich zunächst eine einfache, fest kodierte Notizenliste erstellt. Anschließend richtete ich Dependency Injection (DI) im di-Paket ein. DI ermöglicht es, Klassen ihre Abhängigkeiten von außen bereitzustellen, statt sie selbst zu erzeugen. In der Notizen-App nutzte ich dafür Koin, das sicherstellt, dass Repositorys, Datenquellen und ViewModels auf allen Plattformen automatisch instanziiert und wiederverwendet werden. Die zentrale Funktion initKoin startet Koin und lädt die Konfiguration der Module. Dadurch werden alle benötigten Abhängigkeiten beim App-Start registriert. In der Modules-Datei definierte ich die konkreten Module, in denen beispielsweise Repositorys, Datenquellen und auch die ViewModels instanziiert werden. Die ViewModels können anschließend in den Composable-Funktionen über koinViewModel() eingebunden werden, wodurch die UI direkt auf den State zugreifen kann, ohne dass manuelle Initialisierung nötig ist. Die Module-Datei kann jederzeit plattformspezifisch angepasst werden, weil es Implementierungen gibt, die nicht gleich auf jeder Plattform funktionieren.

Anschließend initialisierte ich in der App.kt-Datei NoteListScreenRoot, der das ViewModel über koinViewModel() verwendet. So konnte ich die fest kodierte Notizenliste in allen Emulatoren – iPhone, Android und Desktop – direkt anzeigen.

23. Woche (18.08. – 22.08.2025)

In der Woche dreiundzwanzig arbeitete ich an der Daten-Schicht, die aus einer lokalen und einer Remote-Datenquelle besteht. Zunächst implementierte ich die Remote-Anbindung mit Ktor (plattformübergreifender Kotlin-Framework für HTTP-Kommunikation), die HTTP-Anfragen verarbeitet und Daten zwischen DTOs und Domain-Modellen konvertiert.

Zur zentralen Steuerung dient ein Repository, das den Zugriff auf beide Datenquellen kapselt. Die benötigten Dependencies, darunter die Remote-Datenquelle und der plattformspezifische HttpClient, wurden über Dependency Injection bereitgestellt.

Für die lokale Datenbank nutzte ich Room – eine SQLite-basierte Lösung, bei der CRUD-Operationen über ein Data Access Object abstrahiert und Änderungen automatisch an die UI weitergegeben werden.

So kann das Repository sowohl auf Remote- als auch lokale Daten zugreifen. Für den Zugriff auf alle Notizen nutzt es primär die lokale Datenquelle und synchronisiert beim ersten Laden die Remote-Daten, falls die lokale Datenbank leer ist. So wird ein Offline-First-Verhalten ermöglicht.

24. Woche (25.08. – 29.08.2025)

In der letzten Woche meines Praktikums implementierte ich den NoteEdit-Screen zum Anlegen und Bearbeiten von Notizen einschließlich aller erforderlichen Komponenten.

Zur Verknüpfung der Screens setzte ich die Navigation über ein sealed interface Route um, das alle Navigationsziele typisiert definiert. Mit NavHost und NavController wird NoteList als Startscreen angezeigt, von dort gelangt man per Klick auf eine Notiz zum NoteEdit-Screen, wobei optional die Notiz-ID übergeben wird. Ohne ID wird automatisch eine neue Notiz erstellt. Die UI nutzt ein Scaffold mit TopAppBar und BottomAppBar: Die TopAppBar zeigt je nach Route einen Zurück-Button, die BottomAppBar enthält einen „+“-Button zum schnellen Anlegen neuer Notizen.

Bei der Arbeit mit Compose Multiplatform empfand ich vor allem den Umgang mit den Dependencies als eine der größten Schwierigkeiten. Es gibt keine vollständige zentrale Bibliothek oder einheitliche Übersicht und oft funktionieren bestimmte Versionen nur auf einer Plattform stabil, während der Build auf einer anderen Plattform (z. B. iOS) direkt fehlschlägt oder unerwartete Fehler anzeigt. Für plattformspezifische Implementierungen muss man außerdem genau wissen, wie jede Plattform funktioniert und welche Besonderheiten dort zu beachten sind. Trotz dieser Herausforderungen