ingenieur
wissenschaften
htw saar

Hochschule für
Technik und Wirtschaft
des Saarlandes
University of
Applied Sciences

Ausarbeitung

zum Thema Lambda-Architektur
im Rahmen des Kurses Software-Architektur
an der Hochschule für Technik und Wirtschaft des Saarlandes

Ausarbeitung und Evaluation - Lambda Architektur

vorgelegt von
Tobias Kronser
Moritz Schönenberger
Leon Weyand

betreut und begutachtet von
Prof. Dr. Markus Esch

Saarbrücken, 28.02.2025

Inhaltsverzeichnis

1 Einleitung

Im Rahmen der Vorlesung "Software-Architektur" haben wir eine Fallstudie zur Lambda-Architektur durchgeführt. In diesem Kapitel erläutern wir zunächst, warum wir uns für die Betrachtung dieser Architektur entschieden haben. Wir stellen die Idee unserer Fallstudie dar, in welcher ein simuliertes Parkhaussystem analysiert wird. Anschließend definieren wir die Forschungsziele, die wir mit dieser Fallstudie erreichen wollen, und geben einen Überblick über die eingesetzten Technologien.

1.1 Motivation

Heutzutage werden immer größere Datenmengen verarbeitet. Moderne Anwendungen müssen daher in der Lage sein, große Datenströme effizient zu verarbeiten, zu speichern und bereitzustellen, ohne dabei Verlust an Leistung oder Zuverlässigkeit zu riskieren.

Ein zentrales Problem bei der Verarbeitung großer Datenmengen wird durch das CAP-Theorem beschrieben. Dieses Theorem besagt, dass ein verteiltes Datensystem immer drei grundlegende Eigenschaften berücksichtigen muss:

- **Consistency (Konsistenz)**: Jeder Lesevorgang liefert entweder den aktuellsten verfügbaren Wert oder schlägt fehl.
- **Availability (Verfügbarkeit)**: Jede Anfrage erhält garantiert eine Antwort, auch wenn einige Knoten des Systems ausgefallen sind.
- **Partition Tolerance (Partitionstoleranz)**: Das System funktioniert weiter, auch wenn Teile des Netzes ausfallen oder Kommunikationsprobleme auftreten.

Allerdings definiert das CAP-Theorem hierbei eine grundlegende Einschränkung: Ein System kann immer nur zwei dieser drei Eigenschaften garantieren. In datenintensiven Anwendungen, die eine hohe Fehlertoleranz (Partition Tolerance) und Verfügbarkeit (Availability) erfordern, kann daher nicht sichergestellt werden, dass jede Abfrage immer die aktuellsten Daten liefert. Dies stellt eine große Herausforderung für Systeme dar, die sowohl Echtzeitdatenverarbeitung als auch Langzeitanalysen ermöglichen sollen.

Die Lambda-Architektur ist ein Architekturmuster, das speziell für Big-Data-Anwendungen entwickelt wurde, also für Systeme, die große Datenmengen effizient verarbeiten müssen. Ihr Ziel ist es, eine fehlertolerante und hochverfügbare Architektur bereitzustellen, die trotz der Einschränkungen des CAP-Theorems eine möglichst hohe Datenkonsistenz gewährleistet. Sie begegnet dem Problem des CAP-Theorems, indem sie zwei Verarbeitungswege kombiniert: den Batch Layer für konsistente Langzeitanalysen und den Speed Layer für schnelle Echtzeitabfragen.

In dieser Fallstudie wollen wir diese Eigenschaften der Lambda-Architektur analysieren und bewerten. Dazu simulieren wir einen realen Anwendungsfall, in dem sowohl die Langzeitanalyse großer Datenmengen als auch die schnelle Bereitstellung neuer Daten eine zentrale Rolle spielen.

Wir orientieren uns dabei an bestehenden Anwendungsgebieten der Lambda-Architektur. Diese Architektur ist insbesondere in Web- und IoT-Anwendungen weit verbreitet, da sie

eine skalierbare Verarbeitung großer Datenmengen bei gleichzeitig schneller Verfügbarkeit aktueller Daten ermöglicht.

Für unsere konkrete Fallstudie haben wir uns für eine IoT-Anwendung im Bereich der Parkhausanalyse entschieden. Ziel ist es, ein Parkhaussystem zu simulieren, das mithilfe von Kameras die Ein- und Ausfahrten in mehreren Parkhäusern erfasst und analysiert. Dabei sollen fahrzeugspezifische Informationen gespeichert und verarbeitet werden. Dieses Szenario eignet sich aus unserer Sicht gut für die Lambda-Architektur, da sich Anwendungsfälle sowohl für Langzeitanalysen (z.B. Belegungsmuster über mehrere Wochen) als auch für Echtzeitabfragen (z.B. aktuelle Parkplatzverfügbarkeit) ergeben.

1.2 Forschungsfragen

Mit dem Ziel unserer Fallstudie vor Augen können wir die folgenden Forschungsfragen definieren, die wir durch unsere Implementierung der Lambda-Architektur beantworten möchten:

- **RQ1:** Mit welchen Technologien kann eine Lambda-Architektur umgesetzt werden?
- **RQ2:** Welche architektonischen Eigenschaften der Lambda-Architektur lassen sich in unserer Fallstudie identifizieren?
- **RQ3:** Was sind Vor- und Nachteile der Lambda-Architektur?
- **RQ4:** Welche Herausforderungen treten bei der Implementierung der Lambda-Architektur in einer kleinen, nicht skalierten Umgebung auf?

1.3 Forschungsansatz

Unser Ziel in dieser Fallstudie ist es, eine einfache, aber funktionsfähige Implementierung der Lambda-Architektur in einer simulierten Umgebung zu entwickeln. Der Fokus liegt dabei nicht auf der Entwicklung einer produktionsreifen Lösung für ein reales Parkhaussystem, sondern vielmehr auf der Schaffung eines praxisnahen Szenarios, das hilft, die Architektur besser zu verstehen und ihre Vor- und Nachteile zu evaluieren.

Um dieses Ziel zu erreichen, entwickeln wir eine Datenquelle, die Ein- und Ausfahrten verschiedenartiger Autos simuliert. Diese Daten durchlaufen die verschiedenen Schichten der Lambda-Architektur, so dass sowohl Echtzeit- als auch Batch-Verarbeitung getestet werden können.

Um die erste Forschungsfrage (RQ1) zu beantworten, haben wir uns für bewährte Open-Source-Technologien entschieden. Anstatt eine eigene Architektur von Grund auf neu zu entwickeln, verwenden wir existierende Werkzeuge, die bereits in der Praxis für die Lambda-Architektur eingesetzt werden. Die Auswahl basiert auf einer Literaturrecherche zu typischen Technologien für die drei Schichten der Lambda-Architektur. Eine detaillierte Erläuterung der Komponenten und Funktionsweisen der Lambda-Architektur erfolgt in Kapitel 2.

Der Speed Layer der Lambda-Architektur ist für die Echtzeitverarbeitung zuständig. Im Speed Layer müssen die Daten schnell verarbeitet werden. Für die Implementierung des Speed Layers wurde daher Apache Kafka gewählt. Kafka ermöglicht eine schnelle und zuverlässige Verarbeitung und Weiterleitung von Datenströmen.

Der Serving Layer übernimmt die Aggregation und Bereitstellung der verarbeiteten Daten. Er verbindet den Speed- und den Batch-Layer und bietet eine zusammenführende Sicht auf die gespeicherten Informationen. Wir haben uns entschieden, den Serving Layer

mit Apache Cassandra zu realisieren. Cassandra ist eine NoSQL-Datenbank, die speziell für schnelle Abfragen optimiert ist und in vielen realen Implementierungen der Lambda-Architektur für den Serving Layer verwendet wird.

Im Batch Layer werden alle historischen Rohdaten gespeichert und verarbeitet. In der klassischen Lambda-Architektur werden hier häufig verteilte Systeme wie Apache Hadoop oder Apache Spark eingesetzt, da diese für den Umgang mit großen Datenmengen optimiert sind. Da in unserer Fallstudie jedoch nicht mit so großen Datenmengen gearbeitet wird, verzichten wir auf eine skalierbare, verteilte Speicherung. Stattdessen verwenden wir auch hier Apache Cassandra, da es sowohl für schnelle Lesezugriffe als auch für die Langzeitspeicherung großer Datenmengen optimiert ist.

Die Anwendung wird in C# entwickelt, da diese Sprache eine gute Integration mit Apache Kafka und Apache Cassandra bietet. Für die Verwaltung und Bereitstellung der verschiedenen Komponenten verwenden wir Docker. Diese Container-Technologie ermöglicht eine einfache Installation, Verwaltung und Reproduzierbarkeit unserer Umgebung, da alle benötigten Dienste in isolierten Containern laufen und unabhängig von der Konfiguration des Host-Systems sind.

2 Verwandte Arbeiten

In diesem Kapitel geben wir einen Überblick über bestehende Arbeiten zur Lambda-Architektur und deren Einfluss auf die Entscheidungen, die wir bei der Entwicklung unserer Fallstudie getroffen haben. Zunächst erläutern wir die grundlegende Struktur der Lambda-Architektur, ihre Kernkomponenten und ihre Ziele. Anschließend betrachten wir existierende Implementierungen, die als Inspiration für unser eigenes Fallstudienszenario dienten.

Darauf aufbauend analysieren wir die architektonischen Eigenschaften der Lambda-Architektur sowie ihre bekannten Vor- und Nachteile, um eine Grundlage für die spätere Bewertung unserer Fallstudie zu schaffen. Abschließend vergleichen wir die Lambda-Architektur mit der Kappa-Architektur, einer alternativen Architektur, die in bestimmten Szenarien die Schwächen der Lambda-Architektur umgehen soll.

2.1 Überblick über die Lambda-Architektur

Die Lambda-Architektur wurde von Nathan Marz [7] als Antwort auf die wachsenden Herausforderungen bei der Verarbeitung großer Datenmengen im Zeitalter von Big Data entwickelt. Sie entstand aus der Erkenntnis, dass herkömmliche Architekturen oft zu komplex, fehleranfällig und schwer skalierbar sind. In vielen Systemen kann bereits ein einziger Datenfehler oder der Ausfall einer Komponente dazu führen, dass das gesamte System inkonsistent wird, Daten verloren gehen oder verfälscht werden und das System nicht mehr zuverlässig arbeitet.

Ein weiteres Problem klassischer Architekturen ist, dass sie Echtzeit- und historische Daten nicht effizient kombinieren können. Bei Big-Data-Anwendungen muss das System nicht nur jederzeit auf die Daten zugreifen und auf Anfragen reagieren können, sondern auch sicherstellen, dass die gelieferten Informationen stets aktuell sind. Dies würde bedeuten, dass bei jeder einzelnen Anfrage alle gespeicherten Daten vollständig durchsucht und verarbeitet werden müssten, um die aktuellsten Werte zu gewährleisten. Bei wachsenden Datenmengen wird dies jedoch schnell ineffizient und führt zu hohen Latenzzeiten und Systembelastungen.

Um diese Probleme zu lösen, verfolgt die Lambda-Architektur einen Ansatz, der große Datenmengen effizient verarbeitet und gleichzeitig Fehlertoleranz, Skalierbarkeit und Konsistenz gewährleistet. Hierzu werden Berechnungen nach ihrer Komplexität in Echtzeit- und Batch-Berechnungen unterschieden. Echtzeit-Berechnungen werden direkt auf eintreffenden Daten ausgeführt und ermöglichen eine aktuellen Überblick. Hierzu dürfen diese nur wenig Zeit in Anspruch nehmen, weshalb es sich in der Regel um inkrementelle Berechnungen handelt. Batch-Berechnungen werden periodisch auf der Gesamtheit aller Daten ausgeführt und können eine lange Zeit in Anspruch nehmen. Diese Berechnungen können beliebig komplex sein, ihre Ergebnisse sind allerdings stets erst verzögert verfügbar.

Das Hauptziel der Lambda-Architektur ist es, die aktuellen Daten aus Echtzeit-Berechnungen mit den Ergebnissen der letzten Batch-Berechnungen zu kombinieren, um Anfragen über historische und aktuelle Daten hinweg zu ermöglichen. Hieraus ergibt sich eine Aufteilung der Architektur in drei Schichten unterteilt (siehe Abbildung 1):

- **Batch Layer:** Speichert alle Rohdaten unverändert und dient als Langzeitarchiv. Dadurch können die Daten wiederholt verarbeitet und für regelmäßige Langzeitanalysen verwendet werden.
- **Speed Layer:** Verarbeitet Datenströme in Echtzeit, um sofortige Ergebnisse zu liefern. Da der Batch-Layer nicht für schnelle Abfragen geeignet ist, übernimmt der Speed-Layer die Aufgabe, aktuelle Daten schnell zur Verfügung zu stellen.
- **Serving Layer:** Verbindet Batch- und Speed-Layer und stellt die Daten für Benutzeranfragen bereit. Kombiniert die Ergebnisse der beiden Schichten und ermöglicht so eine konsistente und effiziente Datenbereitstellung.

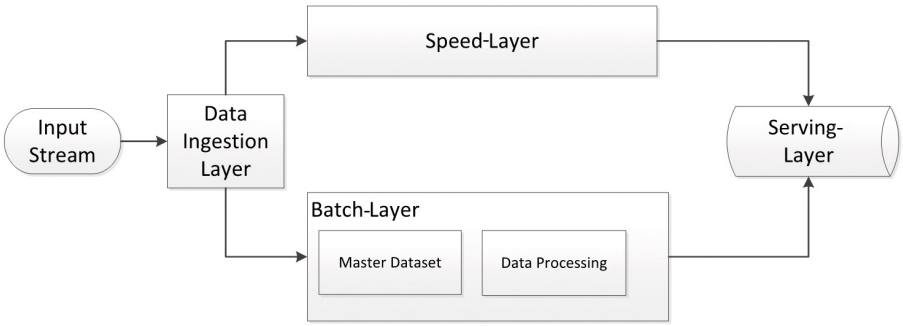

Abbildung 2.1: Aufbau der Lambda-Architektur [6].

Durch diese klare Trennung der Verarbeitungsebenen ermöglicht die Lambda-Architektur ein Gleichgewicht zwischen geringer Latenz und hoher Genauigkeit. Während die Batch Layer eine robuste und fehlertolerante Verarbeitung großer Datenmengen gewährleistet, wäre es zu zeitaufwändig, diese Schicht bei jeder Anfrage vollständig zu durchlaufen, um aktuelle Daten bereitzustellen. Hier kommt der Speed Layer ins Spiel: Er verarbeitet neue Daten in Echtzeit und stellt sie sofort zur Verfügung, um eine schnelle Reaktionszeit zu gewährleisten.

2.2 Anwendungsfälle der Lambda-Architektur

Da traditionelle Systeme wie relationale Datenbanken und klassische Batch-Processing-Systeme zunehmend an ihre Grenzen stoßen, wenn es darum geht, die stetig wachsenden Datenmengen effizient zu verarbeiten, gewinnen Big-Data-Architekturen für moderne Anwendungen immer mehr an Bedeutung. Insbesondere im Zeitalter des Internets, in dem kontinuierlich riesige Datenmengen generiert, übertragen und analysiert werden, hat sich die Lambda-Architektur als leistungsfähige Lösung etabliert [2, 3, 5].

Yuvraj Kumar [5] beschreibt in seinem Paper eine Vielzahl von modernen Anwendungsfällen der Lambda-Architektur sowie die Technologien, die häufig für deren Implementierung verwendet werden. Große Technologieunternehmen wie Twitter, LinkedIn, Netflix und Amazon nutzen die Lambda-Architektur, um ihre enormen Datenmengen effizient zu analysieren. Vor allem im Bereich der personalisierten Werbung spielt sie eine zentrale Rolle: Historische Kundendaten werden mit aktuellen Nutzerinteraktionen kombiniert, um in Echtzeit maßgeschneiderte Werbung auszuspielen. Auch im Finanzsektor nutzen Unternehmen die Lambda-Architektur, um historische Transaktionsdaten mit Echtzeitanalysen zu verknüpfen. So können verdächtige Muster identifiziert und Betrugsversuche frühzeitig erkannt werden. Auch im Internet of Things (IoT) findet die Lambda-

Architektur breite Anwendung. So wird sie beispielsweise in Smart Cities eingesetzt, um Logistikprozesse zu optimieren oder die Abfallentsorgung effizienter zu gestalten.

Das Paper von Kiran et al. [3] zeigt, dass insbesondere Smart City Anwendungen stark von der Lambda-Architektur profitieren, da hier große Mengen an Sensordaten kontinuierlich erfasst und analysiert werden müssen. Die Autoren beschreiben eine konkrete Implementierung der Lambda-Architektur zur Verarbeitung und Analyse von Sensordaten in einer Netzwerkumgebung. Hierzu untersuchen Kiran et al. die Verarbeitung von Netzwerk- und Sensordaten aus dem Energy Sciences Network (ESnet). Ziel der Implementierung ist die Erkennung von Anomalien und die Optimierung der Netzwerkauslastung. Die Lamda-Architektur ist für diese Anwendung besonders geeignet, da sie die gleichzeitige Verarbeitung historischer und aktueller Netzdaten ermöglicht. Während im Batch-Layer Langzeitanalysen durchgeführt werden, um wiederkehrende Muster zu identifizieren, kann der Speed-Layer Anomalien in Echtzeit erkennen und darauf reagieren.

Die in der Literatur beschriebenen Anwendungsfälle, insbesondere die Arbeiten von Kiran et al. [3] und Kumar [5], haben uns geholfen, ein geeignetes Szenario für unsere Fallstudie zu identifizieren. Die Forschung zeigt, dass die Lambda-Architektur besonders häufig in Smart City-Anwendungen eingesetzt wird, da sie die Echtzeitverarbeitung großer Sensordatenströme mit Langzeitanalysen kombiniert.

Darauf aufbauend wollten wir für unsere Fallstudie eine Anwendung im Smart City Kontext wählen, die ebenfalls Echtzeitanalysen mit Langzeitprognosen kombiniert. Dies führte uns schließlich zur Parkhausanalyse. Ähnlich wie in der Arbeit von Kiran et al. [3] ist es in unserem Szenario sinnvoll, historische Daten zur Vorhersage der Parkhausauslastung mit Echtzeitinformationen über die aktuelle Belegung zu verknüpfen.

2.3 Technologien der Lambda-Architektur

Der vorherige Abschnitt hat gezeigt, dass die Lambda-Architektur eine weit verbreitete Big-Data-Architektur ist, die bereits in einer Vielzahl von Anwendungen implementiert wurde. In der Literatur werden verschiedene Technologien beschrieben, die zur Implementierung der drei Schichten der Lambda-Architektur verwendet werden können.

Ein konkretes Beispiel für die technologische Umsetzung liefert das Paper von Kiran et al. [3], das eine Cloud-basierte Implementierung der Lambda-Architektur für die Sensordatenverarbeitung beschreibt. Die Autoren verwenden unterschiedliche Technologien für den Batch Layer, den Speed Layer und den Serving Layer.

Für den Batch Layer verwenden Kiran et al. [3] das Apache Hadoop Distributed File System (HDFS). HDFS ist eine der am häufigsten verwendeten Technologien für die Lambda-Architektur, da es ein skalierbares, fehlertolerantes und verteiltes Dateisystem darstellt, das speziell für die Speicherung großer Datenmengen entwickelt wurde [1]. Aufgrund seiner Architektur ist HDFS jedoch relativ langsam bei Lese- und Schreibzugriffen, weshalb es hauptsächlich für Batch-Processing-Aufgaben verwendet wird. In der Implementierung von Kiran et al. wird HDFS verwendet, um historische Rohsensordaten von Netzwerkroutern unverändert zu archivieren. Da Lesezugriffe auf HDFS relativ langsam sind, können periodische Batch-Analysen sehr zeitaufwändig sein.

Für die Speed Layer verwenden Kiran et al. [3] Apache Spark Streaming. Apache Spark ist eine vielseitige Technologie, die sowohl für Batch Processing als auch für Stream Processing eingesetzt werden kann. Spark Streaming ist eine Erweiterung von Apache Spark und ermöglicht die Verarbeitung von Datenströmen in Echtzeit, indem eingehende Daten kontinuierlich analysiert und aggregiert werden. Der Vorteil von Spark Streaming gegenüber klassischen Batch-Technologien ist, dass es Echtzeitdatenverarbeitung mit geringer Latenz ermöglicht, ohne dass die Daten vorher vollständig gespeichert werden müssen.

Für den Serving Layer verwenden Kiran et al. [3] Amazon S3, um die Sensordaten aus dem Batch Layer zu aggregieren und für historische und Echtzeit-Abfragen bereitzustellen. Die Wahl von Amazon S3 basiert darauf, dass die Autoren eine Cloud-basierte Implementierung der Lambda-Architektur entwickelt haben, die auf skalierbare Speicherdienste angewiesen ist.

Im Gegensatz dazu planen wir in unserer Fallstudie eine lokale Implementierung der Lambda-Architektur, weshalb alternative Technologien für die Speicherung und Aggregation im Serving Layer in Betracht gezogen werden müssen.

In der Literatur werden zahlreiche weitere Technologien beschrieben, die für die verschiedenen Schichten der Lambda-Architektur eingesetzt werden können. Besonders hervorzuheben ist die Arbeit von Kumar [5], der mehrere alternative Technologien diskutiert, die auch für unsere Fallstudie vielversprechend sind.

Laut Kumar [5] ist Apache Kafka eine der am häufigsten verwendeten Technologien für die Lambda-Architektur. Kafka ist ein verteiltes Publish-Subscribe-Messaging-System, das für Echtzeit-Datenströme optimiert ist [4]. Es ermöglicht das Publizieren neuer Daten in Echtzeit in eine Queue. Von dort aus können die Daten sowohl an den Batch-Layer zur langfristigen Speicherung als auch an den Speed-Layer zur sofortigen Verarbeitung weitergeleitet werden. Damit ist sichergestellt, dass Echtzeit- und Langzeitverarbeitung parallel erfolgen können.

Eine weitere Technologie, die laut Kumar [5] häufig in der Lambda-Architektur eingesetzt wird, ist Apache Cassandra. Cassandra ist eine hochskalierbare NoSQL-Datenbank, die sich besonders für den Serving Layer eignet [7]. Im Serving Layer werden Echtzeitabfragen aus dem Speed Layer und aggregierte Batch Views aus dem Batch Layer zusammengeführt und gespeichert. Cassandra zeichnet sich durch schnelle Lese- und Schreibzugriffe aus, was für die kontinuierliche Aktualisierung der Daten im Serving Layer wichtig ist.

2.4 Eigenschaften der Lambda-Architektur

Nachdem wir uns in den vorherigen Abschnitten mit verschiedenen Anwendungsfällen und Technologien der Lambda-Architektur beschäftigt haben, wollen wir nun die zentralen Eigenschaften dieser Architektur näher analysieren. Basierend auf den betrachteten Anwendungsfällen identifizieren wir die wesentlichen Stärken und Schwächen der Lambda-Architektur. Diese identifizierten Eigenschaften wollen wir später als Richtlinien für die Implementierung und Evaluierung unserer Fallstudie nutzen.

2.4.1 Vorteile der Lambda-Architektur

Die Lambda-Architektur zeichnet sich durch eine klare Trennung der Datenverarbeitung in drei Schichten aus. Diese Struktur ermöglicht es, jede Schicht unabhängig voneinander zu optimieren und weiterzuentwickeln. Wie Marz [7] beschreibt, sind die einzelnen Verarbeitungsebenen nur gering voneinander abhängig, so dass sie mit unterschiedlichen Technologien realisiert werden können. Dadurch kann jede Schicht flexibel ausgetauscht oder skaliert werden, ohne dass die gesamte Architektur überarbeitet werden muss. Diese Skalierbarkeit macht die Lambda-Architektur besonders geeignet für Big-Data-Anwendungen, da sie flexibel an steigende Datenmengen angepasst werden kann.

Ein weiteres zentrales Merkmal der Lambda-Architektur ist die Fehlertoleranz durch die Verwendung von Immutable Data [7]. Da die Rohdaten unverändert in der Batch Layer gespeichert werden, bleiben sie jederzeit abrufbar und können erneut verarbeitet werden, ohne dass das System inkonsistent wird. Marz [7] betont, dass dieser Ansatz

Datenverluste nahezu ausschließt, da Daten nicht gelöscht oder überschrieben, sondern nur ergänzt werden. Tritt ein Fehler in der Verarbeitung oder in den Daten selbst auf, kann dieser durch eine erneute Verarbeitung der unveränderten Rohdaten behoben werden. Dies ist insbesondere in Anwendungen mit hohen Anforderungen an die Datenintegrität von Vorteil. Die Arbeit von Kiran et al. [3] zur Netzwerkanalyse im ESnet zeigt, dass diese Eigenschaft es ermöglicht, Sensordaten langfristig zu speichern und Fehler in der Netzwerkanalyse durch Nachberechnungen zu korrigieren.

2.4.2 Nachteile der Lambda-Architektur

Obwohl die Lambda-Architektur viele Vorteile bietet, bringt sie auch Nachteile mit sich.

Die Trennung der Verarbeitungsebenen führt zu einem erhöhten Entwicklungs- und Wartungsaufwand. Die von Marz [7] beschriebene geringe Abhängigkeit zwischen den Schichten bietet zwar Flexibilität, allerdings ergibt sich auch ein geringes Potenzial für gemeinsame Implementierungen beziehungsweise die Wiederverwendung von Komponenten. Dies trifft umso mehr zu, wenn die Schichten unterschiedliche Technologien einsetzen.

Kumar [5] beschreibt insbesondere die hohe Komplexität der Kombination von Speed Layer und Batch Layer im Serving Layer. Da für beide Verarbeitungsebenen separate Pipelines entwickelt werden müssen, ist es notwendig, die Ergebnisse beider Ebenen in der Serving Layer zu integrieren. Diese Integration kann fehleranfällig sein und erfordert eine sorgfältige Synchronisation der Daten, um konsistente Abfragen zu gewährleisten. Während die Lambda-Architektur auf konzeptioneller Ebene einfach zu verstehen ist, stellt die technische Umsetzung eine Herausforderung dar. Die verschiedenen Schichten werden mit unterschiedlichen Open-Source-Technologien realisiert, die miteinander kommunizieren müssen [2].

Ein weiteres Problem der Lambda-Architektur ist der hohe Speicherbedarf durch die doppelte Datenhaltung. Da sowohl die Speed Layer als auch die Batch Layer Daten speichern, werden viele Daten redundant gespeichert, um eine konsistente Verarbeitung in beiden Schichten zu gewährleisten. Während der Batch-Layer alle Rohdaten langfristig speichert, verwendet der Speed-Layer einen separaten Speicher für die Echtzeitverarbeitung. Dies führt insbesondere bei Anwendungen mit großen Datenmengen zu einem erhöhten Speicherbedarf [5].

3 Konzept

3.1 Beschreibung des Fallstudien-Szenarios

Wie in Abschnitt 2.2 erläutert, wird die Lambda-Architektur besonders häufig in Smart City-Anwendungen eingesetzt, da diese Systeme eine kontinuierliche Verarbeitung großer Sensordatenmengen erfordern. Diese Erkenntnis aus der Literatur hat uns dazu inspiriert, ein Szenario aus einem ähnlichen Bereich zu wählen, das die Anforderungen an eine Lambda-Architektur erfüllt. Wir haben uns für die Analyse von Parkhäusern entschieden, da hier eine Kombination aus Echtzeit- und Langzeitanalyse erforderlich ist, um wertvolle Erkenntnisse zu gewinnen.

Ziel unserer Fallstudie ist es, eine reale Anwendungssituation zu simulieren, in der es sinnvoll ist, die Lambda-Architektur zur Datenverarbeitung und -analyse einzusetzen. Unser Szenario basiert auf einem Parkhaus, das an den Ein- und Ausfahrten mit Kameras ausgestattet ist. Diese Kameras erfassen ein- und ausfahrende Fahrzeuge, deren Kennzeichen und Fahrzeugmerkmale gespeichert werden. Zusätzlich werden Zeitstempel der Ein- und Ausfahrten erfasst (Siehe Abbildung 3.1).

Abbildung 3.1: Aufbau der Lambda-Architektur [6].

Die gesammelten Sensordaten können für zwei verschiedene Arten von Analysen verwendet werden. Zum einen ermöglicht die Langzeitanalyse die Untersuchung von Spitzenzeiten, typischen Auslastungsmustern und Trends über mehrere Wochen oder Monate. Zum anderen erlaubt die Echtzeitanalyse die Berechnung der aktuellen Anzahl geparkter Fahrzeuge.

Um sicherzustellen, dass unser Szenario tatsächlich eine große Menge an Sensordaten verarbeiten kann, erweitern wir die Fallstudie auf mehrere Parkhäuser, die zentral verwaltet werden. Dies stellt eine zusätzliche Herausforderung an die Architektur dar, da nun nicht nur die Daten eines einzelnen Parkhauses verarbeitet werden müssen, sondern die Datenströme mehrerer Standorte in einem verteilten System aggregiert und analysiert werden (Siehe Abbildung 3.2).

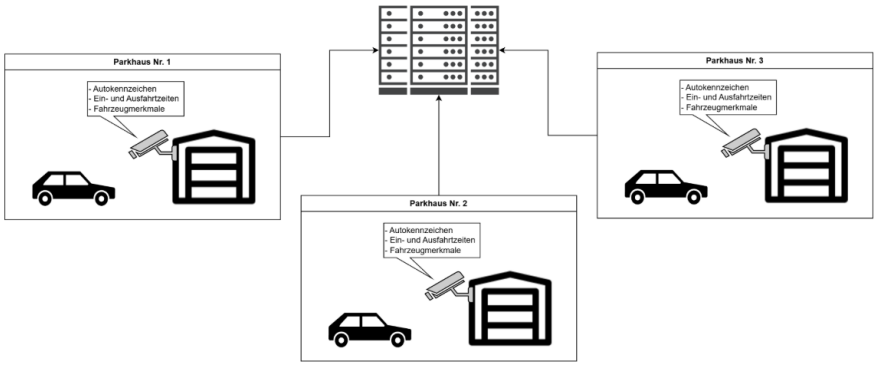

Abbildung 3.2: Aufbau der Lambda-Architektur [6].

Da wir in unserer Fallstudie keine realen Kameras zur Datenerfassung verwenden können, werden wir einen Sensordatengenerator entwickeln, der realistische Ein- und Ausfahrtsereignisse simuliert. Dieser Generator wird kontinuierlich synthetische Sensordaten erzeugen, die in unser System eingespeist werden und so einen realistischen Datenfluss simulieren.

3.1.1 Warum eignet sich dieses Szenario für die Lambda-Architektur?

In Abschnitt 2.4 wurden anhand der Literatur die zentralen Eigenschaften der Lambda-Architektur identifiziert. Es wurde analysiert, welche Vor- und Nachteile diese Architektur aufweist und für welche Anwendungsfälle sie besonders geeignet ist. Aufbauend auf diesen Erkenntnissen kann begründet werden, warum das von uns gewählte Szenario optimal für die Umsetzung mit der Lambda-Architektur geeignet ist.

Ein wesentlicher Aspekt unseres Szenarios ist die zentrale Verwaltung beliebig vieler Parkhäuser, wodurch kontinuierlich große Mengen an Sensordaten erfasst und verarbeitet werden müssen. Diese Daten stammen von Kameras an den Ein- und Ausfahrten der Parkhäuser, die Informationen über ein- und ausfahrende Fahrzeuge einen zugehörigen Zeitstempel liefern. Wenn hinreichend viele Parkhäuser Daten liefern, stellt ein solcher kontinuierlicher Datenstrom hohe Anforderungen an die Datenverarbeitung. Genau für solche Szenarien wurde die Lambda-Architektur entwickelt, da sie die Verarbeitung kontinuierlich eingehender Sensordaten mit Langzeitdatenanalysen kombiniert.

Einfache Datenverarbeitung basierend auf einer Datenhistorie in einer Datenbank wäre in unserem Szenario ineffizient, da die zu verarbeitende Datenmenge mit jedem neuen Ereignis zunimmt. Wenn beispielsweise die aktuellen Füllstände der Parkhäuser aus immer aus den in einer Datenbank hinterlegten Park-Ereignissen berechnet würden, müsste bei jeder Ein- oder Ausfahrt eines Fahrzeugs oder bei jedem Abruf des Füllstandes alle vorhandenen Daten neu analysiert werden, um die aktuelle Auslastung des Parkhauses zu berechnen. Die Beschränkung auf den aktuellen Tag führt zu verfälschten Ergebnissen, da Autos nicht am selben Tag in ein Parkhaus Ein- und Ausfahren müssen. Daher würden mit stetig wachsender Anzahl der Datensätze Abfragezeiten steigend und die Performance sinken. Die Lambda-Architektur bietet hier eine Lösung durch die Kombination von Echtzeit- und Langzeitanalyse: Der Speed-Layer ermöglicht die sofortige Berechnung der aktuellen Parkhausauslastung, relativ zum letzten Batch-Füllstand, während der Batch-Layer Langzeitanalysen zu Füllhistorie, Stoßzeiten oder Nutzungsmustern liefert.

Ein weiterer wichtiger Faktor ist die Skalierbarkeit unseres Szenarios. Da unser System nicht auf ein einzelnes Parkhaus beschränkt ist, sondern potenziell viele Standorte ver-

walten kann, müssen neue Datenquellen problemlos integriert werden können. In einer herkömmlichen Architektur würde die Integration zusätzlicher Sensordaten zu einem erhöhten administrativen und technischen Aufwand führen. Im Gegensatz dazu wurde die Lambda-Architektur speziell für hochskalierbare verteilte Systeme entwickelt, so dass neue Parkhäuser und Sensordatenquellen dynamisch hinzugefügt werden können, ohne die bestehende Architektur wesentlich zu verändern. Die Skalierbarkeit wird durch den Einsatz verteilter Technologien gewährleistet.

Darüber hinaus spielt die Fehlertoleranz in unserem Szenario eine zentrale Rolle. Sensordaten sind fehleranfällig, sei es durch technische Defekte der Kameras oder durch unvollständige Erfassung der Fahrzeuge. Die Lambda-Architektur bietet mit ihrem Konzept der Immutable Data eine Lösung: Da alle Rohdaten beibehalten werden, können fehlerhafte oder unvollständige Daten nachträglich korrigiert werden, sofern die für eine Korrektur notwendigen Informationen zur Verfügung stehen, wie beispielsweise im Falle einer falsch kalibrierten Waage. Nach der Korrektur können das Batch-Layer basierend auf den korrigierten Rohdaten nochmal alle sich daraus ergebenden Daten berechnen.

4 Implementierung

Das Projekt ist in der Programmiersprache CSharp geschrieben. Lokale Instanzen von Apache Kafka und Cassandra werden in Docker-Containern gestartet.

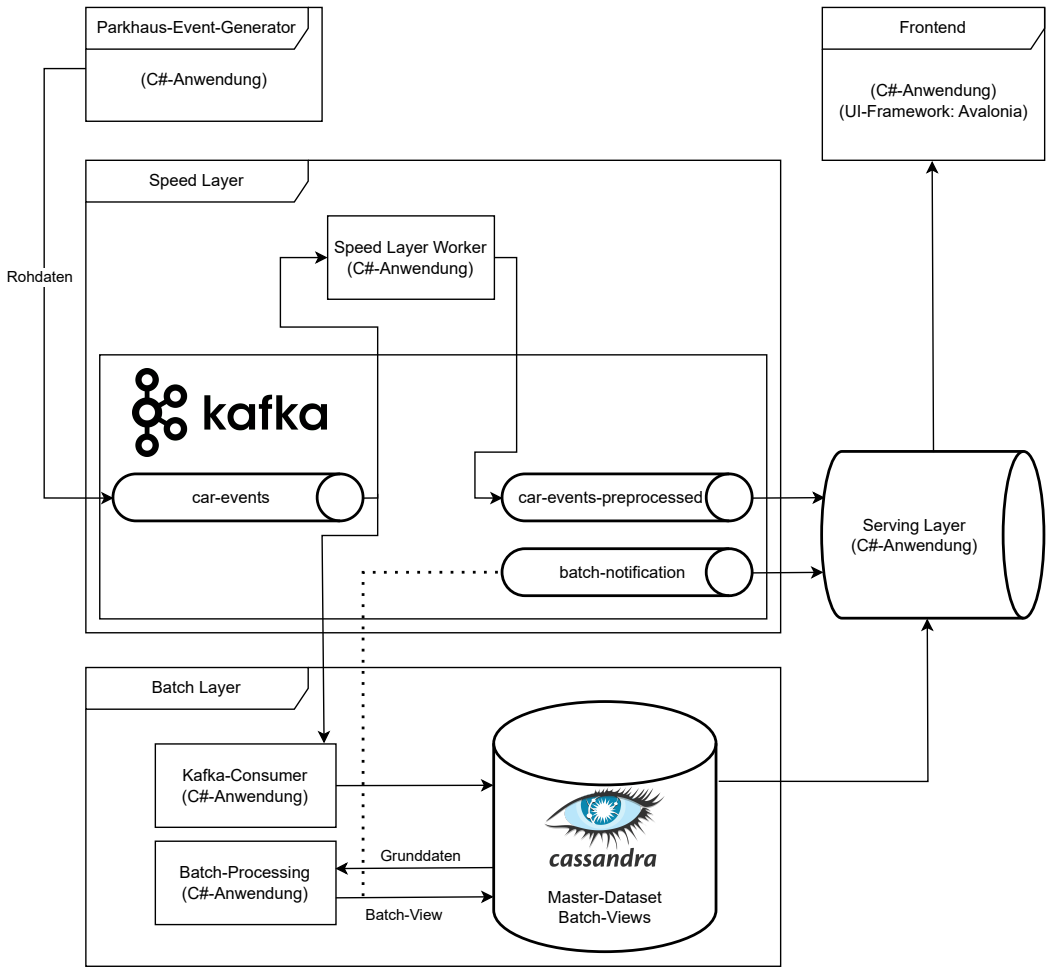

Abbildung 4.1: Programmstruktur

4.1 Datenbankstruktur

In der Tabelle 4.2 sind alle Parkhäuser, die im System existieren mit ihrem Standort, sowie ihrer Kapazität, hinterlegt. Die Daten hierin werden aktuell manuell angelegt.

In der Tabelle 4.3 sind alle in der Vergangenheit geschehenen Park-Events gespeichert. Jedes dieser Events stellt die Ein- beziehungsweise Ausfahrt eines Autos in ein Parkhaus dar. Bei diesen Daten handelt es sich um reine Rohdaten, die den unverarbeiteten Daten von Parkhäusern entsprechen.

Abbildung 4.2: Parkhaus-Tabelle

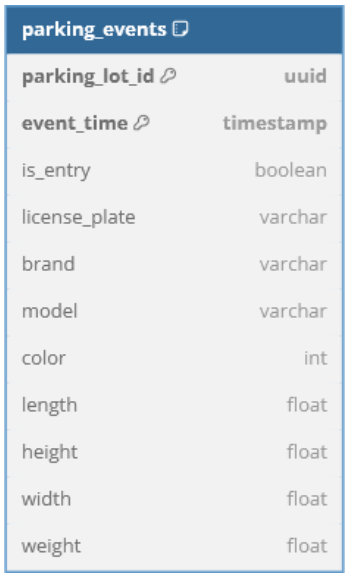

Abbildung 4.3: Parkevent-Tabelle

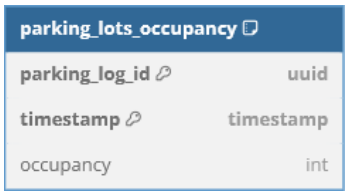

Abbildung 4.4: Zwischensummen-Tabelle

In der Tabelle 4.4 werden Füllstände der Parkhäuser zu bestimmten Zeitpunkten gespeichert. In der aktuellen Konfiguration wird je Parkhaus der Füllstand zu jeweils vollen 10 Minuten berechnet. Diese Daten werden periodisch vom Batch-Layer berechnet und können bei Bedarf, beispielsweise nach Rohdaten-Korrektur oder Batch-Prozess-Korrektur nochmal über alle Daten hinweg neu berechnet werden.

4.2 Core-Bibliothek

Auf der Core-Bibliothek bauen alle anderen Projekte auf. In dieser Bibliothek befinden sich alle geteilten Programmteile:

- **Konfiguration**: string Konstanten für Kafka-Topics, sowie Cassandra-Aufbau bestehend aus Tabellen und -Spaltennamen
- **Geteilte Interfaces**: ICarModel (allgemeine Autoinformationen), ICarData (ICarModel + spezifische Autoinformationen), IParkingEventData (ICarData + Parkhaus + Einfahrts- oder Ausfahrtszeit)
- **Geteilte Datenstrukturen**: CarEntryData (rerpäsentiert ein vollständiges Einfahrts-Event), CarExitData (rerpäsentiert ein vollständiges Ausfahrts-Event), OccupancyState (repräsentiert den exakten Füllstand eines Parkhauses zu einem gegebenen Zeitpunkt)
- **Serialisierungs- und Deserialisierungsfuntionen**: Datenstrukturen binär oder als String serialisieren und wiedererstellen
- **CassandraHelper**: Hilfsklasse, die die Verbindung zur Cassandra-Datenbank verwaltet und alle in anderen Projekten verwendeten Datenbankzugriffe mit Hilfe von CQL (Cassandra Query Language) umsetzt
- **Setup-Validierung**: ermöglicht Anwendungen, die Kafka oder Cassandra benötigen zu überprüfen, ob Kafka und Cassandra korrekt aufgesetzt sind

4.3 Datengrundlage

Da keine realen Daten zur Verfügung stehen, verwenden wir einen selbst geschriebenen Datengenerator, der Ein- und Ausfahrten in ein Parkhaus über Zeit simuliert. Dabei werden Parkevent-Daten “CarEntryData” und “CarExitData” bestehend aus Parkhaus-Id, Zeitstempel, Nummernschild, Automarke, Automodell, Farbe, gemessener Länge, gemessener Breite, gemessener Höhe und gemessenem Gewicht generiert. Die Parkevent-Daten werden dann direkt in die Kafka-Topic “car-events” geschrieben. Ausfahrts-Events werden stets basierend auf einem zuvor generierten Einfahrts-Event, zu dem es noch kein zugehöriges Ausfahrts-Event gibt generiert. Dieser Prozess geschieht stark beschleunigt, um einfaches Testen zu ermöglichen: Innerhalb von Minuten werden Daten für einen ganzen Tag generiert. Um die Daten für mehrere Parkhäuser auf einmal zu generieren, können beliebig viele Parkhäuser konfiguriert werden. Für jedes Parkhaus wird dann ein entsprechender Generator-Thread gestartet. Der Generator berücksichtigt die maximale Kapazität eines Parkhaus und generiert morgens mehr Einfahrts-Events und abends mehr Ausfahrts-Events. In der Datenbank stehen Einfahrts- und Ausfahrts-Events in der selben Tabelle und werden nur durch ein bool-Flag, bei dem **True** ein Einfahrts-Event markiert, unterschieden.

4.4 Speed Layer

Der Speed Layer beinhaltet nur ein einzelnes Projekt “SpeedLayerParkingDeckWorker”, das gleichzeitig Kafka-Consumer und Kafka-Producer ist. Dabei handelt es sich um eine Kommandozeilenanwendung, die die eingehenden Grunddaten aus der Kafka-Topic “car-events” erhält und für ein einzelnes Parkhaus mehrere Events zu einer Zwischensumme zusammenfasst. Diese Zwischensumme entspricht der Differenz der im Parkhaus stehenden Autos im Vergleich zum Stand vor den Events, sie wird gebildet, damit das Serving-Layer nicht alle Park-Events einzeln verarbeiten muss. Wird eine Obergrenze an Park-Events für ein Parkhaus erreicht, oder ist eine Zeitspanne seit dem ersten Park-Event

abgelaufen, so wird die Differenz gemeinsam mit der Parkhaus-Id in eine zweite Kafka-Topic "car-events-preprocessed" geschrieben. Die Obergrenze an Parkevents, sowie die Zeitspanne sind im Worker konfigurierbar. Es können beliebig viele Speed-Layer-Worker gleichzeitig gestartet werden, wobei je Parkhaus ein Speed-Layer-Worker vorgesehen ist. Standardmäßig wird beim Starten des Projekts auch ein Worker je Parkhaus gestartet. Durch die Verwendung der selben GruppenId (abhängig vom Parkhaus des Workers) ist es auch möglich, mehrere Worker für das selbe Parkhaus zu starten, während Kafka jedes Event nur an einen dieser Worker verteilt. Praktisch ist das in unserer Fallstudie allerdings nicht sinnvoll, da ein einzelnes reales Parkhaus nur eine überschaubare Menge an Events produzieren kann.

4.5 Batch Layer

Das Batch Layer besteht aus mehreren Projekten:

- **Kafka-Consumer** persistiert alle Grunddaten aus Kafka in der Datenbank.
- **Batch-Processing** führt regelmäßig verschiedene Batch-Berechnungen basierend auf den in der Datenbank persistierten Daten aus und schreibt die Ergebnisse in die Datenbank zurück.

4.5.1 KafkaConsumer

Der Kafka-Consumer des Batch Layers ist eine Kommandozeilenanwendung, die beim Start die Kafka-Topic "car-events" abonniert und alle hier anfallenden Events erhält. Diese Events werden dann in Cassandra übertragen, hierdurch bleiben alle Rohdaten langfristig verfügbar. Die Übertragung geschieht der Einfachheit halber in einzelnen Datenbankoperationen. In einem realen System, in dem mehr Daten anfallen, wäre es hingegen wichtig, die Events zu bündeln und eine Menge Events gesammelt an die Datenbank zu übertragen. Es können beliebig viele Kafka-Consumer gleichzeitig gestartet werden. Alle Instanzen teilen die selbe Kafka-GroupId, wodurch jedes Event nur an genau eine Instanz übertragen wird. In der Kommandozeile werden Ein- und Ausfahrten ausgegeben.

4.5.2 Batch-Processing

Das Batch-Processing des Batch Layers ist eine Kommandozeilenanwendung, die periodisch die Daten der vergangenen halben Stunde aus der Datenbank läd und darauf basierend Füllstände je Parkhaus in 10 Minuten Intervallen berechnet. Die im Batch-Processing berechneten Daten (Batch-Viwes) werden in die Datenbank zurückgeschrieben. Immer, wenn ein Batch-Vorgang abgeschlossen ist, schreibt die Anwendung eine Benachrichtigung in die Kafka-Topic "batch-notification". Diese Nachricht signalisiert dem Serving Layer, dass es seine Batch-Daten neu laden und veraltete Speed-Layer-Daten verwerfen soll. Da der Datengenerator beschleunigt Daten generiert, läuft diese Anwendung auch beschleunigt und berechnet alle 2 Minuten die Daten einer halben Stunde.

Es existiert ein weiterer Batch-View, der tägliche Tagesmetriken berechnet. Diese beinhalten Durchschnitts- und Maximalwerte für Länge, Breite, Höhe und Gewicht der Autos, die sich an einem Tag in einem Parkhaus befunden haben. Dazu wird die Anzahl der für die Berechnung der Durchschnittswerte genutzten Autodaten mitgespeichert, um Durchschnittswerte über andere Zeitfenster wie Wochen, Wochentage, Monate und Jahre bilden zu können. Die Idee dahinter war, Metriken für Statik und Einteilung zukünftiger Parkhäuser berechnen zu können. Aus Aufwandsgründen existiert hierzu allerdings nur die

Batch-Berechnung, sowie eine zugehörige Cassandra-Tabelle. Die Integration in Serving Layer und Frontend ist nicht mehr geschehen.

4.6 Serving Layer

Das Serving Layer besteht nur aus einem Projekt "ServingLayer". Hier handelt es sich um eine Kommandozeilenanwendung, die Daten aus Kafka und Cassandra zusammenführt und für Benutzeranwendungen bereitstellt. Beim Start liest der Serving Layer alle existierenden Parkhäuser aus der Cassandra-Datenbank und legt sich je Parkhaus einen Cache an. Dazu werden die Kafka-Topics "car-events-preprocessed", deren Daten im Speed Layer berechnet werden, und "batch-notification" abonniert. Dieser Cache beinhaltet die aktuelle Anzahl an Autos im Parkhaus, die sich aus dem Ergebnis des letzten Batch-Views, sowie der seitdem angefallenen Summe an Park-Events in "car-events-preprocessed" ergibt. Immer wenn eine Nachricht in "batch-notification" den Abschluss eines weiteren Batch-Vorgangs signalisiert, wird der neue Batch-View geladen und der veraltete Teil der Speed Layer Daten aus "car-events-preprocessed" verworfen. Aufgrund des Aufbaus des Speed-Layers, sowie der asynchronen Natur von Batch- und Speed-Layer kann ein Ergebnis des Speed-Layers Daten aus 2 Batch-Berechnungen beinhalten. Das kann zu temporären Inkonsistenzen führen, die aber nach der nächsten Batch-Berechnung korrigiert sind. Anwendungen können die Daten des Serving Layers über TCP/IP abfragen. Wenn Historische Daten, die sich nicht im Cache befinden, wie Füllstände zu einer früheren Zeit abgefragt werden, leitet das Serving Layer die Anfrage an die Cassandra-Datenbank durch.

4.7 Frontend

Zur Visualisierung des Projektes, insbesondere im Zuge der Präsentation, wurde eine grafische Anwendung geschrieben. Diese ist wie der Rest des Projektes ebenfalls in C# geschrieben und verwendet Avalonia, ein von WPF (Windows Presentation Foundation) inspiriertes Cross-Platform UI-Framework. Zur Visualisierung der Daten in Form von Diagrammen wird LiveCharts2 eingesetzt.

Das Frontend kommuniziert über TCP/IP mit dem Serving Layer und hat keine Kenntnis von den restlichen Schichten. Hierüber fragt das UI periodisch die aktuellen Daten ab.

5 Evaluierung

In diesem Kapitel soll die Fallstudie in Hinblick auf die Forschungsfragen genauer evaluiert werden. Wir möchten bewerten, ob unsere Fallstudie dazu geeignet ist, die architektonischen Eigenschaften der Lambda-Architektur zu verwenden. Außerdem wollen wir in diesem Kapitel noch einmal die Herausforderungen erläutern, auf die wir bei der Implementierung gestoßen sind

Evaluierung der Fallstudie Grundsätzlich ist unser Fallstudienszenario für die Implementierung mit der Lambda-Architektur geeignet. Das Serving Layer unserer Fallstudie ist in der Lage, die Daten der Batch und Speed Layer zu verbinden, um Fragen über den aktuellen Füllstand zu beantworten.

Dennoch ist die Lambda-Architektur für unsere konkrete Implementierung der Fallstudie überdimensioniert, da Generierung und Verarbeitung auf einem einzigen Gerät stattfinden müssen. Somit sind wir nur in der Lage, eine relativ geringe Datenmenge zu generieren.

Die Fallstudie zeigt die Trennung der Architektur in 2 Quanten, Speed und Batchlayer.

Die Erweiterung des Projektes um neue Berechnungen im Batch Layer ist aufwändig. Neben der Implementierung der Berechnung selbst muss eine Datenbanktabelle zum persistieren des Ergebnis (Batchview) angelegt werden und das Servinglayer um eine entsprechende Schnittstelle erweitert werden. Soll die Berechnung mit Echtzeitdaten aus dem Speedlayer angereichert werden, muss dieses ebenfalls erweitert werden.

Durch das Beibehalten sämtlicher Rohdaten im Batchlayer ist die Architektur fehlerresistent. Beispielsweise können als fehlerhaft identifizierte Datenpunkte im Batchlayer korrigiert oder entfernt werden, womit die Fehler nach der Berechnung des nächsten Batchviews behoben sind.

Eine fundierte Bewertung der Systemperformance ist nicht möglich, da das gesamte System mangels einer Serverfarm auf einer einzigen Maschine ausgeführt wird und auch keine riesigen Datenmengen zur Verfügung stehen.

Dank dem Einsatz von Apache Kafka und Apache Cassandra ist das System grundsätzlich skalierbar. Es kann einfach um weitereParkhäuser erweitert werden, wobei für jedes ein eigener SpeedLayerWorker instanziiert werden kann. Neue Batch-Berechnungen können auf anderen Maschinen ausgeführt werden. Das Servinglayer kann beliebig oft instanziiert und auf verschiedenen Maschinen ausgeführt werden, allerdings wäre die Erweiterung um einen Loadbalancer notwendig, damit die TCP/IP-Anfragen auf die unterschiedlichen Instanzen verteilt werden können.

Die Implementierung insbesondere des ServingLayers ist relativ anspruchsvoll. Zusätzlich entsteht durch die Aufteilung in 3 Schichten ein hoher Kommunikationsaufwand.

Literatur

[1] Ilya Ganelin, Ema Orhian, Kai Sasaki und Brennon York. *Spark: Big data cluster computing in production*. John Wiley & Sons, 2016.

[2] Jaideep Katkar. „STUDY OF BIG DATA ARHITECTURE LAMBDA ARHITECTURE“. In: (2015).

[3] Mariam Kiran, Peter Murphy, Inder Monga, Jon Dugan und Sartaj Singh Baveja. „Lambda architecture for cost-effective batch and speed big data processing“. In: *2015 IEEE international conference on big data (big data)*. IEEE. 2015, S. 2785–2792.

[4] Jay Kreps, Neha Narkhede, Jun Rao u. a. „Kafka: A distributed messaging system for log processing“. In: *Proceedings of the NetDB*. Bd. 11. 2011. Athens, Greece. 2011, S. 1–7.

[5] Yuvraj Kumar. „Lambda architecture–realtime data processing“. In: *Available at SSRN 3513624* (2020).

[6] Lukas Berle. *Lambda vs. Kappa – Welche Architektur passt zu Ihrem Anwendungsfall?* Zugriff am 28. Februar 2025. 2025. URL: `https://entwickler.de/software-architektur/lambda-vs-kappa`.

[7] James Warren und Nathan Marz. *Big Data: Principles and best practices of scalable realtime data systems*. Simon und Schuster, 2015.

Kolophon

Dieses Dokument wurde mit der LaTeX-Vorlage für Abschlussarbeiten an der htw saar im Bereich Informatik/Mechatronik-Sensortechnik erstellt (Version 2.25, August 2024). Die Vorlage wurde von Yves Hary und André Miede entwickelt (mit freundlicher Unterstützung von Thomas Kretschmer, Helmut G. Folz und Martina Lehser). Daten: (F)10.95 – (B)426.79135pt – (H)688.5567pt