Eidesstattliche Erklärung

Entwicklung eines Konzeptes zur Integration und Kopplung eines Finanzmarktdatensystems unter Einbeziehung von Cloud-Technologien

Diplomarbeit

Zur Erlangung des Grades

Diplom-Informatiker (FH)

im Studiengang Diplom-Informatik (1100)

der Wilhelm Büchner Hochschule

vorgelegt von

Andreas Mayer

Heinrich-Heine-Str. 4

80686 München

Betreuender Hochschullehrer:	Dipl.-Inform. Dipl.-Wirtsch.-Inform. (FH) MBM (Univ.) **Jürgen. R. Dietrich**
Eingereicht am:	[Veröffentlichungsdatum]

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Diplomarbeit selbständig und ohne unzulässige fremde Hilfe angefertigt habe. Die verwendeten Quellen sind vollständig zitiert.

Datum: [Veröffentlichungsdatum] **Unterschrift**_____

Andreas Mayer

ABSTRACT

Entwicklung eines Konzeptes zur Integration und Kopplung eines Finanzmarktdatensystems unter Einbeziehung von Cloud-Technologien

Andreas Mayer

Es gibt keine einheitlichen Konzepte, die darstellen, wie ein Finanzmarktdatensystem idealerweise konzipiert werden sollte. In diesem Zusammenhang wird ein generisches Konzept erarbeitet und anschließend die Fragestellung bearbeitet, inwieweit eine Nutzung der Cloud möglich ist. Hierbei werden diverse Cloud-Technologien evaluiert. Das Ziel soll eine art Referenzarchitektur sein, die individuell auf die Bedürfnisse der Nutzer zugeschnitten werden kann. Eine zentrale Aufgabe ist besteht darin, Zeitreihenkonzepte in der Cloud abzubilden.

Aktuelles System kostet ca. 455000€ pro Jahr.

There are no unified aproaches in building financial market data systems. Inside of this context a generic concept will be developed. In the next step the possibilities of using cloud technologies are evaluated. The goal will be a modular reference architecture which can be tailored to the users needs. A central aproach is the modelling of time-series within the cloud.

INHALTSVERZEICHNIS

Abstract

Inhaltsverzeichnis

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

DANKSAGUNG

Dankbarkeit ist nicht nur die größte aller Tugenden, sondern auch die Mutter von allen.

Marcus Tullius Cicero, (106 - 43 v. Chr.), römischer Redner und Staatsmann

An dieser Stelle bedanke ich mich bei allen Personen, die mich während der Zeit meines Studiums unterstützt haben.

Als erstes möchte ich mich herzlich bei Herrn **Dipl.-Inform. Dipl.-Wirtsch.-Inform. (FH) MBM (Univ.) Jürgen. R. Dietrich** für die sehr gute Betreuung meiner Abschlußarbeit bedanken.

Vielen Dank auch an Herrn **Prof. Dr.-Ing. habil. Jürgen Otten**, Präsident der Wilhelm Büchner Hochschule, und Herrn **Dr. Wolfgang Kliesch**, Studienleiter Informatik, für das Aufsetzen dieses hervorragen Fernstudienganges.

Besonderer Dank gilt Herrn **Prof. Dr. Ernst Denert**, Honorarprofessor und Ehrensenator (Technische Universität München, Fakultät für Informatik) für das Eröffnen neuer Wege in Bezug auf meine Promotion.

Ebenfalls danke ich Herrn **Prof. Dr. rer. nat. Florian Matthes**, Inhaber des Lehrstuhls Software Engineering for Business Information Systems (Technische Universität München, Institut für Informatik) für das mir entgegengebrachte Vertrauen und die Option zur Promotion.

Ich bedanke mich auch bei der **Gesellschaft für Informatik e. V.**, für das Austragen des InformatiCups. Meine Teilname hat den wertvollen Kontakt zwischen Prof. Dr. Ernst Denert und mir ermöglicht.

Mein Vorgesetzter, Herr **Thomas Vogg**, Head of Market Data Services (*IDS GmbH – Analysis and Reporting Services*, Allianz Group), hat mir Freiräume ermöglicht, welche mir das Lernen leichter gemacht haben – danke dafür.

Großer Dank gebührt auch meiner Freundin, Frau **Michaela Ratschnig**, für ihre Zuneigung und das Verständnis für meine sehr begrenzte Zeit.

Ebenfalls danke ich meinen Eltern, **Antal und Gabriele Mayer**, sowie meinem Bruder, **Emmanuel Mayer**, für die lebenslange Unterstützung und das Vertrauen in meine Fähigkeiten.

Zu guter Letzt möchte ich einigen besonders guten Freunden für viele wunderbare Jahre danken, wobei die Reihenfolge keine Gewichtung darstellt.

- **Basem** und **Nathalia Fares**
- **Thomas Wrany**
- **Katharina Fares** und **Bernd Reichelt**
- **Phil Worbs** und **Miriam Herb**

1 EINLEITUNG

Neue Herausforderungen erfordern neue Wege.

Marcus Tullius Cicero, (106 - 43 v. Chr.), römischer Redner und Staatsmann

Aufgrund der Turbulenzen auf den Finanzmärkten, welche durch die Finanzkrise im Jahre 2008 eingeläutet wurden, haben sich die regulatorischen Anforderungen verschärft und der Bedarf der Kunden hinsichtlich der Informationsdichte und Transparenz hat sich deutlich erhöht. Hieraus ergeben sich neue Anforderungen an Finanzmarktdatensysteme hinsichtlich der Maintenanceaufwände, des Durchsatzes sowie der Skalierbarkeit. Die Menge an zu verarbeitenden Daten nimmt ständig zu, weshalb neue Ansätze hinsichtlich der Struktur von Informationssystemen im Bereich der Finanzmarktdaten diskutiert werden müssen[1].

Die Einleitung beschreibt zuerst die Motivation hinter der Arbeit und geht dann auf Aspekte wie die Problemstellung und die Zielsetzung ein.

1.1 MOTIVATION

Einerseits ist die Cloud einer der IT-Trends des Jahres, weshalb es sinnvoll ist, sich mit diesem Bereich näher zu befassen, andererseits will ich herausfinden, ob es u. a. unter Kostengesichtspunkten sinnvoll wäre, die Cloud als mögliche Option für das bestehende System in Erwägung zu ziehen. Zudem hat sich gezeigt, dass das Bestandssystem immer wieder an Grenzen hinsichtlich der Performance und der Skalierbarkeit stößt, die durch ein Redesign ggf. beseitigt werden können. Hierzu sollen neue Konzepte entwickelt und im Rahmen dieser Arbeit teilweise überprüft werden. Aufgrund bisheriger Erfahrungen im Bereich des Betriebs von komplexen IT-Systemen, im Zusammenhang mit IT-fremdem Personal, wird der Technologie-Stack möglichst klein gehalten.

[1] Die Fragestellungen können in anderen Domänen ebenfalls Relevanz besitzten. Der Fokus dieser Arbeit liegt jedoch klar abgegrenzt auf dem Bereich der Finanzmarktdaten.

1.2 PROBLEMSTELLUNG

Die IT-Architektur des bestehenden Finanzmarktdatensystems ist nicht in der Lage, Last-Spitzen abzufangen und besitzt außerdem einen Single-Point-of-Failure im Bereich der Storagekonnektivität. Zudem ist das Monitoring des Bestandssystems sehr schwierig und erfordert lange Einarbeitungszeiten. Die knappen IT-Budgets lassen den Kauf neuer Server nicht zu. Aufgrund der im Normalbetrieb unterdurchschnittlichen Systemauslastung der zwei Produktivmaschinen, wäre dies keine wirtschaftliche Lösung. Deshalb soll eine Nutzung der Cloud am, am Beispiel Amazon Web Services (AWS), evaluiert werden. Einerseits müssen Zeitreihen-Konzepte entwickelt werden, die den Anforderungen an Skalierbarkeit und Performance genügen, andererseits muss das Marktdatensystem an die bestehende, im eigenen Rechenzentrum gehostete, Lösung gekoppelt werden.

1.3 ZIELSETZUNG

Es soll ein neuartiges cloud-basiertes Konzept für Finanzmarktdatensysteme entwickelt werden. Es werden bewusst neue Wege gegangen. Aspekte wie Transparenz und Wartbarkeit, Gesamtperformance und Durchsatz, Skalierbarkeit, Verfügbarkeit und Kosteneffizienz werden in dieses Modell einfließen, wobei nicht alle dieser Punkte in der gleichen Tiefe berücksichtigt werden können. Gesamtperformance, Durchsatz und Wartbarkeit sind Punkte die erst in größer angelegten Tests überprüft werden können, weshalb der Fokus dieser Arbeit auf den Aspekten Transparenz, Skalierbarkeit, Verfügbarkeit und Kosteneffizienz liegt.

1.4 STRUKTUR

Kapitel 2 beschäftigt sich mit den notwendigen Grundlagen. In Kapitel 2.1 wird das Thema Cloud Computing behandelt und in Kapitel 2.2 werden anschließend die Grundlagen bezüglich des Finanzmarktdatensystems geschaffen. Hierbei werden Ziele und Struktur des Systems beschrieben. Aufgrund des limitierten Rahmens werden die zu verarbeitenden Daten zur stichpunktartig erwänht.

Einleitung

In Kapitel 3 werden Konzepte formuliert, die für die jeweiligen Problemstellungen als Lösungsoption infrage kommen. Hier werden dem Leser verschiedene Möglichkeiten skizziert, wie die jeweiligen Komponenten oder Systemteile strukturiert werden können und welche Vor- und Nachteile sich hieraus jeweils ergeben. Der Leser wird in die Lage versetzt, ein eigenes maßgeschneidertes System entwerfen zu können.

In Kapitel 4 wird ein Gesamtkonzept erstellt und hinsichtlich der Kosten evaluiert. Außerdem wird ein Prototyp entwickelt, welcher den Ansatz der Speicherung von Finanzmarktinstrumenten, in serialisierter Form, in S3 überprüft.

Kapitel 5 bildet den Schluss, welcher eine abschließende Beurteilung sowie Ansatzpunkte für weitere Forschungsansätze enthält.

2 HAUPTTEIL - GRUNDLAGEN

Das Interesse denkt nicht, es rechnet. Die Motive sind seine Zahlen.

Karl Marx, (1818 - 1883), deutscher Philosoph, Sozialökonom und sozialistischer Theoretiker

2.1 DARSTELLUNG UND ANALYSE DES THEMAS CLOUD COMPUTING

Cloud Computing ist eines der aktuellen Buzz-Words der IT-Branche. Eine Google Anfrage nach dem Begriff „Cloud Computing“ ergab am 07.03.2011 noch über 26 Mio. Suchergebnisse. Heute, am 03.06.2011, sind es bereits über 43 Mio. Diese Entwicklung zeigt, dass das allgemeine Interesse an diesem Thema zunimmt.

Abbildung 1[2] - Suchergebnisse zum Thema Cloud Computing, 03.06.2011

Aus der *Software Architectures for the Cloud Konferenz*[3], welche vom 15. bis 16.07.2011 in München stattfand, sowie diversen Tech-Talks, weiß ich, dass es bzgl. des Themas Cloud Computing keine allgemeingültige Haltung gibt.

Die Befürworter sehen vor allem die Vorteile, wobei die Gegner eher die Risiken sehen und das Thema als *alten Wein in neuen Schläuchen* herunterspielen.

Ungeachtet der jeweiligen Haltung gibt es Umfragen, die Anlass zum Nachdenken geben[4].

[2] Quelle: Eigenerstellung

[3] http://wwwmatthes.in.tum.de/wikis/sebis/softarch2010

[4] http://www.computerworld.ch/news/it-branche/artikel/cloud-projekte-verfehlen-ziele-56965/

Es gibt einige Gründe, weshalb ich der Überzeugung bin, dass es sich bei diesem Trend um mehr handelt als um einen kurzen Hype.

- Gartner hat Cloud Computing als eines der strategischen Top 10 Technologiethemen für das Jahr 2011 gelistet[5].
- IT-Investitionen in das Thema Cloud Computing steigen[6][7].
- Cloud Computing war das Top-Thema der CeBIT 2011[8]. *„Der Cloud-Umsatz mit Geschäftskunden und Privatverbrauchern wird in diesem Jahr um rund 55 % auf insgesamt 3,5 Milliarden Euro steigen. 2015 werden etwa 10 % aller IT-Ausgaben in Deutschland auf Cloud-Technologie entfallen."* [1][9]

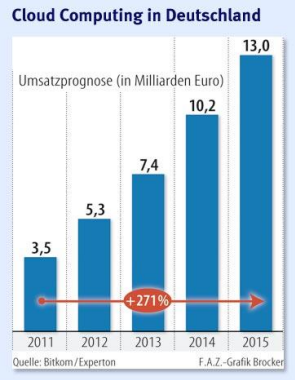


Abbildung 2[10] - Umsatzprognosen zum Thema Cloud Computing in Deutschland

2.1.1 Grundlagen und Paradigmen

An dieser Stelle würde ich gerne eine allgemein anerkannte Definition des Cloud Computing Begriffes anführen, aber diese existiert nicht.[11]

---

[5] http://www.gartner.com/it/page.jsp?id=1454221

[6] http://www.searchdatacenter.de/themenbereiche/cloud/management/articles/305073/

[7] http://www.saasmagazin.de/saasondemandmarkt/studien/wusys220311.html

[8] http://www.cebit.de/de/ueber-die-messe/themen-und-trends/top-themen/cloud-computing_1

[9] Messe-Hannover, „www.cebit.de," 24 Februar 2011. [Online]. Available: http://www.cebit.de/de/ueber-die-messe/daten-und-fakten/die-cebit-2011/cloud-computing-top-thema. [Zugriff am 03 Juni 2011].

[10] Quelle: Siehe Abbildung 8, Bitkom/Experton

[11] Vgl. K. Stanoevska-Slabeva et al. (eds.), Grid and Cloud Computing: A Business Perspective on Technology and Application, Springer-Verlag Berlin Heidelberg, 2010, p. 47 - 50

Tabelle 1 zeigt einige der vorhandenen Definitionen.

Tabelle 1 - Ausgewählte Definitionen des Cloud Computing Begriffes[12]

Source	Definition
Gartner	A style of computing in which massively scalable IT-related capabilities are provided *as a service* using Internet technologies to multiple external customers.
IDC	An emerging IT development, deployment and delivery model, enabling real-time delivery of products, services and solutions over the Internet (i.e., enabling cloud services).
Berkeley RAD Lab	Cloud Computing refers to both the applications delivered as services over the Internet and the hardware and systems software in the datacenters that provide those services. The services themselves have long been referred to as Software as a Service (SaaS). The datacenter hardware and software is what we will call a Cloud. When a Cloud is made available in a pay-as-you-go manner to general public, we call it a Public Cloud; the service being sold is Utility Computing. We use the term Private Cloud to refer to internal datacenters of a business or organization, not made available to the general public. Thus, Cloud Computing is the sum of SaaS and Utility Computing, but does not include Private Clouds. People can be users or providers of SaaS, or users or providers of Utility Computing.
Foster et al. (2008)	[A] large-scale distributed computing paradigm that is driven by economies of scale, in which a pool of abstracted, virtualized, dynamically-scalabe, managed computing power, storage, platforms, and services are delivered on demand to external customers over the Internet.

Eine grobe Klassifizierung kann anhand des zugrunde gelegten Blickwinkels erfolgen. Während Gartner und IDC sehr stark aus Sicht der Endbenutzer argumentieren, beziehen die anderen beiden Definitionen die technologischen und architektonischen Aspekte mit ein.

Im Jahre 2008 wurden über 22 Definitionsvorschläge zum Cloud Computing veröffentlicht. Diese Vorschläge wurden in [2] untersucht. Ziel war es, eine Definition zu erarbeiten, die wiederspiegelt, wie Cloud Computing in der heutigen Zeit begriffen wird[13]:

[12] Vgl. K. Stanoevska-Slabeva et al. (eds.), Grid and Cloud Computing: A Business Perspective on Technology and Application, Springer-Verlag Berlin Heidelberg, 2010, p. 48 - 49

[13] Vgl. K. Stanoevska-Slabeva et al. (eds.), Grid and Cloud Computing: A Business Perspective on Technology and Application, Springer-Verlag Berlin Heidelberg, 2010, p. 49

"Clouds are a large pool of easily usable and accessible virtualized resources (such as hardware, development platforms and/or services). There resources can be dramatically reconfigured to adjust to a variable load (scale), allowing also for an optimum resource utilization. This pool of resources is typically exploited by a pay-per-use model in which guarantees are offered by the Infrastructure Provider by means of customized SLAs." [3][14]

2.1.1.1 Die drei architektonischen Ebenen des Cloud Computing

Cloud Computing kann in drei grundlegende Ebenen gegliedert werden.

Infrastructure as a Service (IaaS) ist ein Konzept, in welchem Ressourcen, wie Rechenkapazitäten (CPU) oder Datenspeicher (Storage), als Dienstleistung (Service) angeboten werden. Die reine Hardware-Ebene wird als Fabric-Layer[15] bezeichnet.

Platform as a Service (PaaS) *kann als Abstraktionsschicht zwischen IaaS und SaaS angesehen werden. Zielgruppe von PaaS-Services sind Software-Entwickler. PaaS ermöglicht es, Applikationen anhand von Plattformspezifikationen zu entwickeln, ohne sich Gedanken um die darunter liegende Hardware Infrastruktur (IaaS) machen zu müssen. In der Regel sind die in die Plattform ausgelagerten Applikationen in der Lage, automatisch zu skalieren, wenn erhöhter Ressourcenbedarf besteht. Dies bietet sich z. B. besonders gut im Rahmen von Web-Applikationen an. Beispiele hierfür sind die Google App Engine, die Force.com Plattform von Salesforce* [3][16] und Microsoft Azure. Zudem werden vorgefertigte Komponenten frei angeboten oder auf Online-Marktplätzen[17] gehandelt.

---

[14] Vgl. K. Stanoevska-Slabeva et al. (eds.), Grid and Cloud Computing: A Business Perspective on Technology and Application, Springer-Verlag Berlin Heidelberg, 2010, p. 49

[15] Vgl. I. Foster, Z. Yong, I. Raicu, L. Raicu und S. Year, „Cloud Computing and Grid Computing 360-Degree Compared," in *Grid Computing Environments Workshop, 2008. GCE '08, 12-16*, 2008.

[16] Übersetzt aus dem Englischen von Andreas Mayer

[17] Bspw. http://aws.amazon.com/customerapps

Software as a Service (SaaS) bietet den höchsten Mehrwert für den Nutzer. Der Grundgedanke ist, dass eine für einen speziellen Zweck nutzbare Software, in der Cloud bereitgestellt und Nutzern, gegen Zahlung einer Nutzungsgebühr, zugänglich gemacht wird. Hierbei kann es sich bspw. um ein Customer Relationship Management (CRM) Tool oder um eine Office Suite handeln. Dem Nutzer bleiben jegliche technologischen Details verborgen, weshalb er sich auf die reine Nutzung der Applikation konzentrieren kann. Ob Serverkapazitäten, Storage, Netzwerkinfrastruktur etc. ausreichen, ist Sache des SaaS-Anbieters. Zu den bekanntesten Anbietern[18] in diesem Segment zählen Salesforce.com und Google Apps.

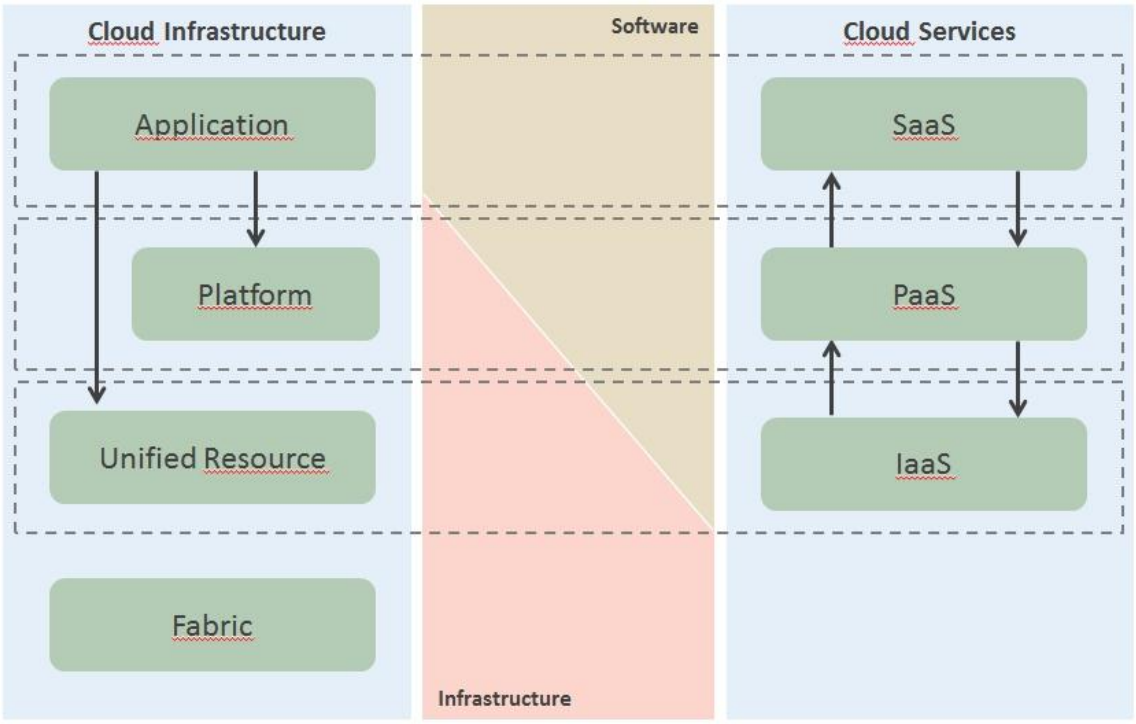


Abbildung 3[19] - Beziehung zwischen Cloud Architektur und Cloud Services

[18] Oft wird vergessen, dass es SaaS-Dienste bereits sehr lange gibt – allerdings ist die Terminologie SaaS neu. Beispiele sind das World-Wide-Web und Mailanbieter wie GMX, Yahoo, Web.de usw.

[19] Quelle: K. Stanoevska-Slabeva et al. (eds.), Grid and Cloud Computing: A Business Perspective on Technology and Application, Springer-Verlag Berlin Heidelberg, 2010, p. 53

Ferner kursieren Begriffe wie Business Process as a Service (BPaaS), Datacenter as a Service/Device as a Service/Desktop as a Service (DaaS), Orchestration as a Service (OaaS) uvm., weshalb die generische Abkürzung XaaS[20] eingeführt wurde.

2.1.1.2 Allgemeine Unterscheidung von Clouds

Generell gibt es drei Arten von Clouds[21]. In der Fachliteratur findet man hierfür auch den Begriff der *Deployment Models*[22].

Auf die *Public Cloud* können alle potenziellen Kunden über das Internet zugreifen. Hierfür werden anbieterspezifische Schnittstellen bereitgestellt. Die Dienstleistungen werden von einer dritten Partei angeboten, die sich um die gesamte Infrastruktur kümmert.

Eine *Private Cloud* ist der Öffentlichkeit nicht zugänglich. Sie dient spezifischen Unternehmen. In diesem Fall gibt es keine scheinbar *unendliche*[23] Anzahl an Ressourcen. Außerdem entfällt hier die Einsparung bezüglich der initialen Hardwarekosten.

Eine *Hybrid Cloud* ist eine Mischung aus einer Public und einer Private Cloud. Einige Teile des Systems werden im eigenen Rechenzentrum gehalten, wobei andere in einer Public Cloud gehostet werden. Ein Use-Case könnte sein, dass übermäßig starke Frequentierung der internen Ressourcen durch dynamische Allokation öffentlicher Ressourcen abgefangen wird.

2.1.2 Vorbereitung der Cloud-Nutzung

[20] Das X dient hier als Platzhalter für beliebige Buchstabenkombinationen.

[21] Der Begriff Cloud kann in diesem Kontext als die Summe der angebotenen Ressourcen verstanden werden.

[22] Vgl. K. Stanoevska-Slabeva et al. (eds.), Grid and Cloud Computing: A Business Perspective on Technology and Application, Springer-Verlag Berlin Heidelberg, 2010, p. 57

[23] In einer Public Cloud gibt es natürlich auch keine unendliche Menge an Ressourcen. Für den Kunden entsteht jedoch der Effekt, da er die zugrundeliegende Infrastruktur nicht sieht. Wenn der Kunde 100 weitere Server braucht, kann er sie bekommen. Um die Kapazitäten muss sich der Anbieter kümmern.

Ungeachtet der technischen Aspekte sollten im Vorfeld immer Kostenmodelle entwickelt werden, um beurteilen zu können ob eine Cloud-Nutzung überhaupt Vorteile bringen kann. Ein Beispielhaftes Modell ist in [4][24] beschrieben.

Im Rahmen des Cloud-Computing spielt das Vertrauen eine sehr große Rolle. Wenn dem Cloud-Anbieter kein Vertrauen entgegengebracht wird, sollte von einer Cloud-Nutzung abgesehen werden.

Bevor Daten in die Cloud migriert werden, müssen diese erst klassifiziert werden. Man muss sich darüber klar werden, welche Daten in die Cloud gegeben werden und welche besser in den eigenen Rechenzentren gehalten werden sollen.

Sensible Kundendaten, Geschäftsgeheimnisse und Neuentwicklungen, die ggf. einen großen Marktvorteil bringen etc., sollten u. U. lieber nicht in die Cloud ausgelagert werden. Die Vorgehensweise muss pro Unternehmen, in Abstimmung mit den Datenschutzexperten und dem Information Security Office (ISO), festgelegt werden.

Aus meiner Sicht ist das Vertrauen gegenüber dem Cloud-Anbieter eine notwendige Voraussetzung. Dieses Vertrauen darf aber nicht damit verwechselt werden, dass der Cloud-Anbieter sich ab sofort um alles kümmert. Der Cloud-Nutzer muss seiner Verantwortung im Umgang mit zu schützenden Daten nach wie vor gerecht werden.

Im Zweifel ist die Rechtsabteilung zu konsultieren[25].

Bevor ein Anbieter ausgewählt wird, sollte geprüft werden welche Sicherheitsstandards er erfüllt. Dies kann durch bestimmte Zertifizierungen „nachgewiesen“ werden.

---

[24] C. Christmann, J. Falkner, D. Kopperger und A. Weisbecker, „Schein oder Sein,“ *iX - Magazin für professionelle Informationstechnik,* Nr. 5, pp. 38-43, 2011.

[25] Wenn die Cloud in Dienstleistungen genutzt werden soll, sollte dies unbedingt in den Verträgen entsprechend formuliert werden.

Amazon Web Services hat sich nach Standards SAS70 Type II, PCI DSS Level 1, ISO 27001 und FISMA zertifizieren lassen[26].

2.1.3 Technologische Aspekte und Features

Im Rahmen des Cloud Computing gibt es einige Technologien und Features, die eine zentrale Rolle spielen.

2.1.3.1 Virtualisierung

Grundziele der **Virtualisierung**[27] sind die Steigerung der Effizienz sowie das Senken der IT-Investitionen.

Eine Virtualisierung kann auf mehreren Ebenen erfolgen.

Beispielsweise kann ein physikalischer Server in mehrere logische Server untergliedert werden. Es ist somit auch möglich, unterschiedliche Betriebssysteme parallel auf einem Server zu betreiben.

Der Grundgedanke der Server-Virtualisierung ist, dass die in der Regel unterdurchschnittlich ausgelasteten Systeme einen höheren Effizienzgrad erreichen, wenn mehrere logische Einheiten pro Server aktiv sind. Normalerweise sollte eine IT-Infrastruktur dafür ausgelegt sein, Peaks abhandeln zu können. Diese Peaks treten in der Regel selten auf. In den Zeiten, in denen ein Zustand durchschnittlicher Last herrscht, sind die Systeme stark unterfordert. In [3] wird eine durchschnittliche Auslastung von Serverumgebungen zwischen 5 % und 15 % genannt.

Ein großer Vorteil bietet sich auch in Bezug auf Deployment-Szenarien. Ein virtueller Server kann ggf. in wenigen Minuten aufgesetzt werden, wobei ein neuer physikalischer Server erst bestellt, im Rechenzentrum installiert und aufgesetzt werden muss.

---

[26] http://aws.amazon.com/de/security/

[27] Die verschiedenen Arten von Virtualisierung auf Wikipedia beschrieben: *http://de.wikipedia.org/wiki/Virtualisierung_(Informatik)*

Abbildung 4[28] - Abstrakte Struktur der Server-Virtualisierung

Eine Virtualisierung der Storage-Ebene ist ebenfalls möglich. Die folgende Abbildung illustriert die Prinzipien am Beispiel vom Logical Volume Manager (LVM).

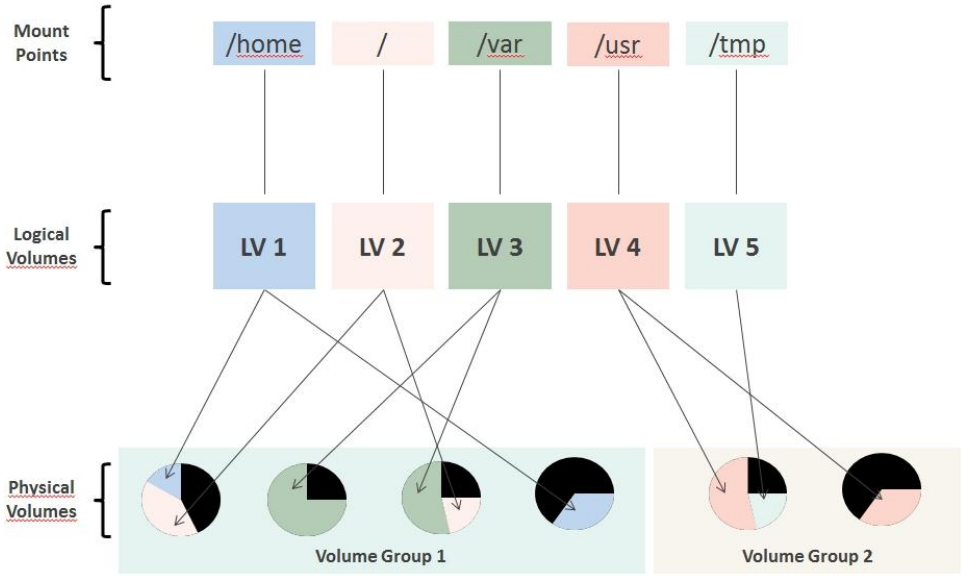


Abbildung 5[29] - Storage-Virtualisierung am Beispiel LVM

Die physischen Festplatten (Physical Volume) werden im ersten Schritt zu einer Gruppe (Volume Group) zusammengefasst. Aus dieser Volume Group werden dann Speicherbereiche zu logischen Einheiten (Logical Volume) zusammengefasst, die anschließend festplattenübergreifend allokiert werden können. Die logischen Einheiten können dann ins System eingebunden (mount) werden. Dieses Verfahren ermöglicht es, die logischen Volumes im laufenden Betrieb zu vergrößern, falls der Speicherplatz nicht mehr ausreicht. Möglicherweise muss das logische Volume zuerst

[28] Quelle: Eigenerstellung

[29] In Anlehnung an http://www.markus-gattol.name/misc/mm/si/content/lvm_components.png

aus dem System genommen (unmount) werden. Der Vorteil ist, dass in gewissen Grenzen[30] keine physischen Festplatten nachgerüstet werden müssen, um ein Volume zu erweitern. Dies erfordert allerdings eine vorausschauende Planung und Erfahrung in Bezug auf die notwendigen Volume-Größen[31].

Zuletzt kann die Netzwerkinfrastruktur virtualisiert werden. Da VLANs[32] in dieser Arbeit keine Rolle spielen, wird auf diesen Themenbereich nicht näher eingegangen.

2.1.3.2 Self-Service und bedarfsorientierte Bereitstellung von Ressourcen

Die Grundidee hinter diesem Aspekt ist, dass es keiner zusätzlichen, menschlichen Interaktion bedürfen soll, wenn ein Kunde bspw. einen neuen Server benötigt. Der Kunde soll einen zusätzlichen Server bei Bedarf[33] selbständig über ein Web-Interface aufsetzen können.

2.1.3.3 Reservierungs- und Verhandlungsmechanismen[34]

Manche Anbieter ermöglichen den Kunden eine vorausschauende Planung und Risikominimierung durch die Reservierung von Ressourcen. Wenn absehbar ist, dass in kürze 20 Server einer bestimmten Klassifizierung benötigt werden, können diese reserviert werden. Falls die Server dann doch nicht benötigt werden, wird nur eine geringe Pauschale gezahlt. Falls die Server jedoch benötigt werden, stehen sie garantiert in der gewünschten Güteklasse zur Verfügung.

---

[30] Die Gesamtkapazität ist natürlich durch die installierten Festplatten begrenzt.

[31] Um eine hohe Aussagefähigkeit herzustellen, sollte ein Capacity-Management aufgesetzt werden, welches Einblicke in die Entwicklung der genutzten Ressourcen bietet und somit Rückschlüsse auf die zukünftige Entwicklung erlaubt. Zur Darstellung bietet sich bspw. Cacti an (http://www.cacti.net/)

[32] http://de.wikipedia.org/wiki/Virtual_Local_Area_Network

[33] On-Demand Resource Provisioning

[34] Ungefähr Anfang 2011 habe ich eine Videopräsentation zum Thema Cloud Computing gesehen, in der ein neuer möglicher Ansatz thematisiert wurde. Es ging darum, die in der Cloud verfügbaren Ressourcen ähnlich wie Aktien zu handeln. Hier wären Systeme vorstellbar, die anhand diverser Verhandlungsparameter selbständig Ressourcen allokieren und ggf. wieder freigeben. Hierbei spielen Themen, wie Performance Prediction (z. B.: Palladio Component Model), auch eine große Rolle. Dies ist ein interessanter Aspekt, jedoch nicht Teil dieser Arbeit. Im Rahmen von AWS gibt es die sog. Spot Instances. Das genaue Verfahren ist auf der Webseite des Anbieters dokumentiert: http://aws.amazon.com/de/ec2/spot-instances/

2.1.3.4 ELASTIZITÄT

Ein großer Vorteil des Cloud Computing ist die bedarfsorientierte dynamische Allokation von Ressourcen.

In den bisher gültigen IT-Strukturen war es zwar generell möglich skalierbare Systeme zu entwerfen, allerdings waren hiermit immer initiale Investitionen verbunden. Wenn plötzlich Kapazitäten frei wurden, weil bspw. ein großer Kunde absprang, blieb die Hardware im Rechenzentrum verfügbar, die durchschnittliche Ressourcennutzung sank jedoch ggf. dramatisch.

Die Cloud bietet nicht nur Skalierbarkeit, sondern darüber hinaus Elastizität. Wenn heute 100 virtuelle Systeme im Einsatz sind und morgen nur noch 25 Systeme benötigt werden, kann das Gesamtsystem in der Cloud nach unten, im umgekehrten Fall nach oben skaliert werden.

2.1.3.5 BEZAHLUNG NACH NUTZUNG[35]

Die Elastizität wird dadurch interessant, dass nur die Ressourcen bezahlt werden, die tatsächlich genutzt werden. Wird die Serveranzahl von 100 auf 25 reduziert, dann werden ab diesem Zeitpunkt nur noch 25 Server abgerechnet.

Dies gibt Planungssicherheit und reduziert die Risiken, welche große IT-Investitionen mit sich bringen.

2.1.4 VORTEILE UND RISIKEN

Zu den größten Vorteilen, aus Kundensicht, zählen meiner Meinung nach folgende:

- Keine initialen Hardwarekosten
- Keine Lizenzkosten sowie kein Risiko von gekauften, aber nicht genutzten Lizenzen
- Keine Personal- und Wartungskosten für den Infrastrukturbereich

[35] Per-Usage Metering and Billing

- Keine Energie- und Infrastrukturkosten (Rechenzentrum, Rack-Space, Kühlung, Strom, etc.)

Zu den Risiken zählen u. a. folgende:

- Lock-In-Effekte[36]
- (Informations-)Sicherheit und Datenschutz
- Verfügbarkeit
- Durchsatz (Performance)

In Deutschland werden Datenschutz- und Compliance-Anforderungen sowie die Standardisierung der internen Prozesse als die beiden größten Herausforderungen angesehen[37].

Ein globaler Vorteil der konsolidierten IT-Rechenzentren liegt in der effizienten Nutzung der Ressourcen. Dies entspricht dem *Green-IT*[38] Gedanken.

2.2 DAS FINANZMARKTDATENSYSTEM – GRUNDLAGEN

Im Großen und Ganzen kann das Finanzmarktdatensystem als ein einheitlicher Datenanbieter verstanden werden. Hier werden die gesamten Vendorcharakteristiken abstrahiert und einheitliche Schnittstellen innerhalb des Unternehmens angeboten.

Die Marktdaten werden zum einen benötigt, um Aussagen über die Entwicklung der eigenen Assets, die in Portfolien zusammengefasst sind, treffen zu können. Zum Anderen sind diese Informationen aus Compliancegründen vorzuhalten, da die Investmentverhältnisse an regulatorische Einheiten[39] gemeldet werden müssen.

---

[36] Aufgrund fehlender Standards in den Bereichen IaaS, PaaS und Saas kann es unter Umständen sehr schwierig und aufwändig sein, die Lösung von einem Anbieter zum anderen zu migrieren (Anbieter Lock-In-Effekt).

[37] Vgl. M. Vehlow und C. Golowsky, *Cloud Computing,* PricewaterhouseCoopers AG, 2010.

[38] http://de.wikipedia.org/wiki/Green_IT

[39] Beispielsweise an die BaFin (Banken- und Finanzaufsicht).

2.2.1 NUTZEN DER ZENTRALISIERUNG

Der Nutzen einer Zentralisierung lässt sich an folgendem Schaubild verdeutlichen.

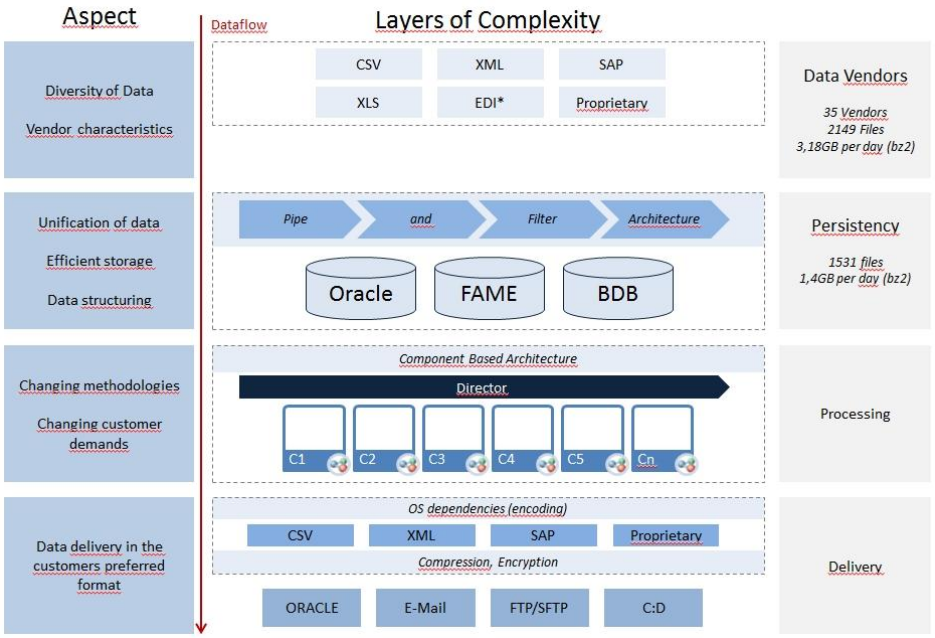


Abbildung 6[40] - Komplexität der Marktdatenverarbeitung

Die Marktdatenverarbeitung erfordert sehr spezielles Expertenwissen, welches nur durch jahrelange Erfahrung aufgebaut und entwickelt werden kann. Wenn Marktdaten in verschiedenen Bereichen und an verschiedenen Standorten verarbeitet werden sollen, muss dieses Wissen jeweils exklusiv aufgebaut werden.

Für eine gut strukturierte Verarbeitung ist eine spezielle Plattform notwendig, die nur mit Domänen-Experten, aus den Bereichen IT und Finance entwickelt werden kann.

2.2.2 FINANZMARKT-DATEN

In einem Finanzmarktdatensystem werden diverse Daten verarbeitet. Einige Beispiele sind folgend aufgelistet:

- Aktien (Equities)
- Indizes

[40] Quelle: Eigenerstellung

- Rohstoffe (Commodities)
- Wechselkurse (FX-Rates)
- Zinsen (Interest-Rates)
- Derrivate
- Sektorisierungs Daten

Weitere Informationen zu diesen Daten können in der einschlägigen Fachliteratur zu den Themen *Finanzmärkte*, *Financial Engineering* und *Quantitative Finance* sowie im Internet gefunden werden.

2.2.3 KOMPONENTEN DES SYSTEMS

Folgende Tabelle gibt einen groben Überblick über zentrale Komponenten des Systems.

Tabelle 2 - Wichtige Komponenten des Finanzmarktdatensystems

Komponente	Beschreibung
Spooler	Diese Komponenten sind für den Down- und Upload von Daten zuständig. Spooler-Prozesse sind ausserdem für das Persistieren und vorverarbeiten der Daten zuständig.
Parser	Der Parser ist für die Normalisierung, Reorganisation, Prüfung und Anreicherung der Originaldaten zuständig.
Loader	Der Loader speichert die Datensätze anhand einer Mapping-Definition, als Zeitreihen, in die entsprechenden Datenbanken.
Catalog	Der Catalog ist ein passives Element, welches als Inventar verstanden werden kann. Im Catalog werden alle im System befindlichen Instrumente, inkl. einiger interessanter Stammdaten, gespeichert.
Job Control	Die Job Control ist der Dirigent im System. Diese Komponente steuert den gesamten Post-Processing-Prozess, welcher nach dem Speichern der Daten in die Datenbanken beginnt. Die Verarbeitungsschritte sind als Abhängigkeitsgraphen definiert.
Build Constituents	In diesem Prozess werden Indizes anhand der Einzeltitel nachgebildet und anschließend werden die Marktwerte verglichen (Index vs. Aggregierter Marktwert). Dies ist eine Qualitätssicherungsmaßnahme, da die Anbieterdaten häufig qualitativen Schwankungen unterliegen.
Calculate Corporate Actions	Dieser Prozess wird bei Aktien Indizes genutzt. Hier werden Kapitalmaßnahmen wie bspw. Splits, Dividendenzahlungen uvm. durchgeführt.
Cross Referencing	Anhand einer künstlichen Id (Anchor) werden die unterschiedlichen Identifier einer Security zusammengefasst.
Export	Diese Komponente extrahiert die von den Kunden benötigten Daten anhand einer Template-Definition. Das Template umfasst Inhalt, Layout und Format des Exports.
Benchmark Calculation Engine	Dieses System erlaubt das berechnen beliebiger Benchmarks. Benchmarks werden als Vergleichszeitreihen genutzt, um das eigene Portfolio zu bewerten. Hauptnutzer dieser Daten sind Fonds- und Portfoliomanager.

2.3 DAS AMAZON WEB SERVICES (AWS) SERVICE-PORTFOLIO

In diesem Kapitel werden einige Services des AWS-Portfolios aufgelistet, die für eine Nutzung infrage kommen. Auf der AWS-Webseite[41] gibt viele weitere interessante Services, die jedoch nicht Teil dieser Arbeit sind. Das AWS-Service-Portfolio wird permanent erweitert.

Um die AWS-Services nutzen zu können, ist ein AWS-Account nötig, welcher auf der AWS-Webseite erstellt werden kann.

Abbildung 7[42] - Link zur AWS-Kontenerstellung

Das AWS Konto ist kostenlos. Für die Nutzung der einzelnen Services muss man sich jeweils zusätzlich registrieren, wofür eine gültige Kreditkarte notwendig ist.

Bei AWS zahlt der Kunde für das, was er nutzt. Die Kosten werden generell durch folgende Aspekte beeinflusst:

- Compute-Time
- Storage
- Data-Transfer (in/out)

AWS besitzt aktuell Rechenzentren in vier verschiedenen Regionen, wobei die Service-Preise sich je nach Region unterscheiden können. Hierdurch ergeben sich Möglichkeiten im Hinblick auf eine globalisierte IT-Infrastruktur. Wichtig ist in diesem Zusammenhang, dass nicht jeder Service in jeder Region angeboten wird. Genaue Informationen hierzu gibt es auf der Webseite des Anbieters.

---

[41] http://aws.amazon.com/de/

[42] Quelle: Eigenerstellung

Neue Services werden ggf. erst in bestimmten Regionen ausgerollt, bevor sie in weiteren Regionen zur Verfügung gestellt werden[43].

2.3.1 AWS-SERVICES

Folgende Services werden im Rahmen dieser Arbeit betrachtet.

- SQS (Simple Queue Service)
- SNS (Simple Notification Service)
- S3 (Simple Storage Service) [SSE (Server Side Encryption)]
- SimpleDB
- EC2 (Elastic Compute Cloud)
- VPC (Virtual Private Cloud)
- Direct Connect

Detaillierte Beschreibungen sowie Prototypische Implementierungen zu den AWS-Services sind in der hervorragenden Anbieter-Dokumentation zu finden[44]. Da eine Beschreibung der Services in dieser Arbeit den Rahmen sprengen würde, werden lediglich einige wichtige Aspekte der genutzten Services beschrieben.

2.3.1.1 ELASTIC COMPUTE CLOUD

Im Kontext dieser Arbeit ist die Unterscheidung der vorhandenen Instanz-Typen wichtog. Es gibt *On-Demand Instanzen* (ODI) und *Reserved Instanzen* (RI).

Bei den ODI wird lediglich für die Nutzung bezahlt und es fallen keine initialen Kosten, sprich Fixkosten an.

[43] AWS Import/Export unterstützt den Import sowie den Export von Daten in und aus Amazon S3-Buckets in den Regionen US Standard und EU (Irland). Quelle: http://aws.amazon.com/de/importexport/

[44] http://aws.amazon.com/de/

Bei Nutzung einer RI fällt eine Pauschale Zahlung an, die sich auf eine ein- oder dreijährige Laufzeit bezieht und nicht erstattet werden kann. Die Nutzungskosten pro Stunde sinken jedoch um ca. 30 %.

Es werden unterschiedliche Instanz-Typen angeboten, die sich hinsichtlich der Rechen-, Speicher- und Storage-Kapazität unterscheiden. Da die Kosten für die unterschiedlichen Systeme z. T. stark variieren, sollte der passende Typ im Vorfeld ermittelt werden. Die Instanzen werden mit einem sog. *Amazon Machine Image* (AMI) versehen, welches u. a. das Betriebssystem enthält. Es werden generell Linux/UNIX und Windows Server angeboten, die sich hinsichtlich der Kosten unterscheiden.

2.3.1.2 Simple Storage Service (S3)

S3 ist ein sog. *object store*, welcher das Speichern von Dateien mit einer Größe bis zu 5TB erlaubt. Der Zugriff erfolgt über eine http(s) basierte API. S3 bietet einen, aus Nutzersicht, nahezu unendlichen Speicherplatz.

Es werden zwei Speichermethoden angeboten. Beim *Standard Storage* werden die Daten über mehrere *availability zones* hinweg repliziert, weshalb dieses Verfahren einen höheren monatlichen Preis pro GB besitzt. Beim *reduced redundancy storage* entfällt diese Redundanz, was zu geringeren monatlichen Kosten pro GB führt.

2.3.2 Fazit zum AWS Service-Portfolio

AWS bietet interessante Services an, die es Entwicklungsteams und Unternehmen erlauben, sich auf die eigenen Kernaufgaben zu konzentrieren. Das bereitgestellte Service-Portfolio wirkt ausgereift und wächst ständig. Der große Vorteil dieses Angebots besteht darin, dass Unternehmen die einzelnen Dienste auch in bestehende Informationssysteme integrieren können und Teilkomponenten wie Queues oder Messaging-Systeme nicht erst, aus Infrastruktursicht, aufgesetzt werden müssen. Die Dienste Virtual Private Cloud, S3 SSE und Direct Connect dürften auch die Bedenken bezüglich der Datensicherheit etwas relativieren. Der Entwickler muss an dieser Stelle lediglich die Service APIs nutzen können.

Dies erspart es den Unternehmen jedoch nicht, sich über IT- und IS-Architekturen Gedanken zu machen. Die Services sind letztlich nur die technischen Vehikel innerhalb des Gesamtkontextes. Die reine Nutzung allein garantiert keine positiven Effekte.

Wichtig ist, dass sich als potenzielle Nutzer nicht von Serviceversprechen blenden lassen. Vor einer Nutzung müssen genaue Analysen durchgeführt werden, die neben den Vorteilen auch die Nachteile transparent machen. Die möglichen Schwachstellen müssen dann ggf. über innovative Prozesse abgefedert werden.

Obwohl die Anbieter intelligente und verteilte Infrastrukturen etabliert haben, kann von einer völligen Ausfallsicherheit nicht die Rede sein. Beispiele, auch aus der jüngsten Vergangenheit[45], belegen, dass potenziell jeder Anbieter von Ausfällen betroffen sein kann[46].

AWS geht mit den Vorfällen bspw. vorbildhaft um und versucht bezüglich der Ausfallgründe Transparenz zu schaffen[47]. Dies schafft Vertrauen in den Anbieter und somit in *das Produkt*.

[45] http://www.zdnet.de/news/41554905/googles-app-engine-von-ausfaellen-betroffen.htm
[46] http://www.computerwoche.de/management/cloud-computing/2492168/
[47] http://aws.amazon.com/de/message/2329B7//175-3908700-5925029

3 HAUPTTEIL – KONZEPTION

Alle großen Entdeckungen passieren zufällig.

Murphy's Technologie-Gesetze

In diesem Kapitel werden diverse grundlegende Konzepte evaluiert und bewertet, die zur Implementierung des Systems infrage kommen. Die jeweiligen Vor- und Nachteile werden individuell aufgeführt, so dass die zum spezifischen Kontext jeweils optimal passende Lösung ermittelt werden kann.

In die Unterkapitel fließen bereits *best-practices* ein, die sich in zehn Jahren praktischer Arbeit mit Finanzmarktdaten entwickelt haben. Da im Rahmen dieser Arbeit kein vollständiges System entwickelt werden kann, sollen dem Leser zumindest die Erfahrungen, die in diesem Umfeld bereits gesammelt wurden, zur Verfügung gestellt werden.

3.1 DATENSTRUKTUREN UND –MODELLIERUNG

Die wichtigsten Datenstrukturen im Marktdatensystem sind Zeitreihen, Stammdaten und Metadaten. Bevor diese Datentypen beschrieben werden, wird auf allgemeine strukturelle Notwendigkeiten eingegangen.

3.1.1 STRUKTURIERUNG DER DATEN

Um den Kundenanforderungen gerecht zu werden ist es notwendig einige Details bei der Strukturierung der Daten zu berücksichtigen.

Es gibt Marktdaten die von unterschiedlichen Preisquellen geliefert werden. Eine Preisquelle kann bspw. eine Börse, ein Datenanbieter oder ein Kontributor sein. Zusätzlich gibt es in bestimmten Bereichen die Notwendigkeit, Snapshots, die sich auf bestimmte Zeitpunkte beziehen, abzubilden. Um diesen Anforderungen von Beginn an

gerecht zu werden, macht es Sinn ein Konzept der folgenden Form zu etablieren – wobei die genaue Ausprägung variieren kann.

Jedes Instrument muss eine eindeutige ID besitzen, durch die es im Datenbestand identifiziert werden kann. Je nach Implementierung der Zeitreihen, *s. Kapitel 3.1.2*, könnten diese Informationen im Schlüssel oder in eigenen Datenobjekten abgebildet werden.

Wenn ein Instrument IX123123 heißt und über Bloomberg geliefert wird, wäre folgende Form für die Unique ID vorstellbar:

<VendorMnemonic><Instrument> <PricingSource> <SnapShotTime><TimeZone>

Würden zu diesem Instrument verschiedene Preisquellen und Snapshots geliefert werden, wären bspw. die Preisdaten unabhängig voneinander realisiert.

Tabelle 3 - Konzept von Preisquellen und Snapshots

UniqueID	Bid	Ask
BBIX123123 BGN 1730CET	100,3	100,6
BBIX123123 BGN 1600CET	99,8	100,0
BBIX123123 BGN 1300CET	96,7	97,0
BBIX123123 UN 1300CET	96,5	96,7

Ohne ein solches Konzept würden die Daten eines Objektes immer mit den Daten einer anderen Preisquelle überschrieben werden – nach dem Motto „der Letzte gewinnt!“.

Gleiches gilt auch für Wechselkurse. Diese können von verschiedenen Zentralbanken oder Daten-Brokern geliefert werden. Wenn es um Schlusskurse geht, wird ebenfalls ein Konzept benötigt um Indizes in verschiedenen Währungen abbilden zu können.

Diese Art der Dynamisierung kann schwer in einem allgemeingültigen Konzept umgesetzt werden. Es müssen ggf. pro Datenanbieter oder Marktdaten-Klasse spezifische Lösungen erarbeitet werden.

3.1.2 Zeitreihen-Konzeption

Um die Zeitreihen abbilden zu können gibt es generell drei Ansätze.

3.1.2.1 Nutzung einer für Zeitreihen optimierten Datenbank

Es gibt Datenbanken die auf die Speicherung von Zeitreihen zugeschnitten sind. In diesem Fall muss der Nutzer ggf. über Erfahrung in der Installation und Administration von IT-Systemen verfügen.

Beispiele solcher Datenbanken sind in der folgenden Tabelle aufgeführt.

Tabelle 4 - Beispiele für zeitreihenoptimierte Datenbanken

Produkt	Webseite
FAME	http://www.sungard.com/fame/
STSdb	http://stsdb.com/
OpenTSDB	http://opentsdb.net/

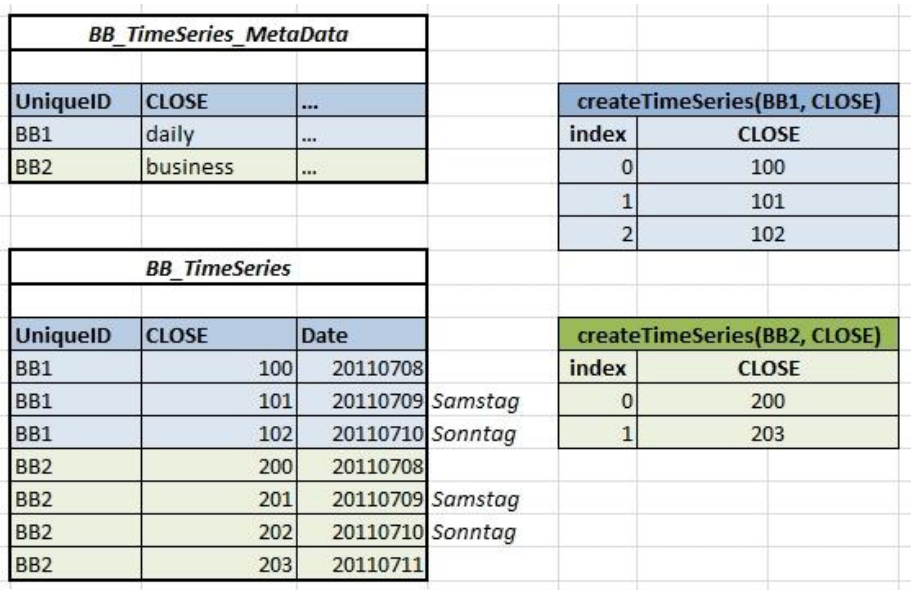
Spezifische Anforderungen, die in diesen Datenbanken nicht vorgesehen sind, müssten ggf. erst über Umwege implementiert werden. Wenn eine Datenbank bspw. nur Zeitreihen unterstützt, die eine Frequenz im Millisekunden Bereich besitzen, so kann dies problematisch werden, wenn tägliche oder monatliche Zeitreihen erzeugt werden müssen.

3.1.2.2 Modellierung der Zeitreihen in einer NoSQL Datenbank

Da es in NoSQL-Datenbanken keine Relationen gibt, müssen andere Strukturen gefunden werden, die eine sinnvolle Abbildung der Daten ermöglichen. Mögliche Unique IDs wurden bereits in *Kapitel 3.1.1* vorgestellt. Die notwendigen Nutzdaten könnten dann in den Spalten abgespeichert werden.

Hierbei sind ggf. vorhandene Grenzen in Bezug auf die abbildbaren Objekte und Datenmengen im Vorfeld zu berücksichtigen.

Um Zeitreihen unterschiedlicher Frequenz abbilden zu können, kann bspw. folgender Ansatz weiter vertieft werden, welcher zugleich auf ein mögliches Problem aufmerksam macht.



BB_TimeSeries_MetaData

UniqueID	CLOSE	...
BB1	daily	...
BB2	business	...

BB_TimeSeries

UniqueID	CLOSE	Date	
BB1	100	20110708	
BB1	101	20110709	Samstag
BB1	102	20110710	Sonntag
BB2	200	20110708	
BB2	201	20110709	Samstag
BB2	202	20110710	Sonntag
BB2	203	20110711	

createTimeSeries(BB1, CLOSE)

index	CLOSE
0	100
1	101
2	102

createTimeSeries(BB2, CLOSE)

index	CLOSE
0	200
1	203

Abbildung 8[48] - Konzept zur Abbildung von Zeitreihen in NoSQL Datenbanken

Es gibt Daten die auch an Wochenenden geliefert werden. Diese können in Zeitreihen der Frequenz *Business* nicht abgespeichert werden. Im Falle von *Abbildung 8 - Konzept zur Abbildung von Zeitreihen in NoSQL Datenbanken* könnte das Datenmodell für die Zeitreihe BB2.CLOSE zwar tägliche Daten darstellen, aber die Datenstrukturen wären dazu nicht in der Lage – dies ist eine Inkonsistenz die vermieden werden muss.

Bevor Daten ins System geladen werden, muss eine genaue Analyse der Originaldaten, idealerweise über mehrere Tage bzw. Wochen hinweg, durchgeführt werden.

3.1.2.3 Modellierung der Zeitreihen durch eigene Datenobjekte

Dieses Modell sieht vor, dass die Zeitreihen in irgendeiner Form persistiert wurden. Die Speicherung kann bspw. als BLOB in einer relationalen Datenbank oder als serialisiertes Objekt in einem Filestore erfolgen.

Hierzu müssen Datenstrukturen entwickelt werden, die Zeitreihen unterschiedlicher Frequenz darstellen können. Gängige Frequenzen im Marktdatenumfeld sind bspw. täglich, wöchentlich und monatlich.

[48] Quelle: Eigenerstellung

3.1.3 STAMMDATEN-KONZEPTION

Stammdaten sind weitestgehend statische Daten.

Beispiele hierzu sind *Index-Name*, *Index-Währung*, *Option Strike-Price*, *Bond Maturity Date*, *Bond Coupon* usw.

Da sich Stammdaten über den Lebenszyklus hinweg selten verändern, sind speicheroptimierte Lösungen zu entwickeln um die zu speichernde Datenmenge zu reduzieren. Sofern die Stammdaten sich nicht verändern, müssen sie nicht täglich abgespeichert werden. Dieses Konzept kann bspw. durch *sparse-series* realisiert werden.

Auch wenn die genutzte Datenbank hierfür keine eigenen Datenstrukturen bereitstellen sollte, kann dieses Prinzip modelliert werden.

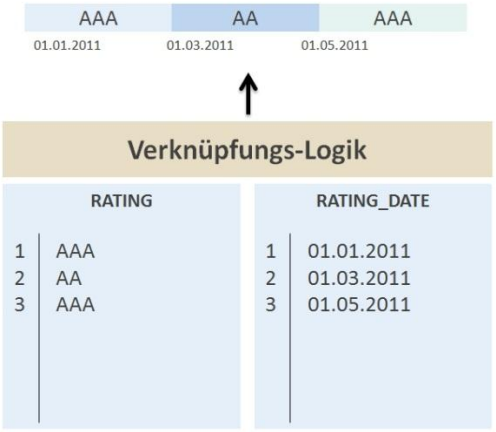


Abbildung 9[49] - Das Prinzip der Sparse-Series

Wenn nach dem letzten Gültigkeits-Datum der selbe Wert geliefert wird, wird kein neuer Eintrag in der Sparse-Series erzeugt. Stattdessen wird in der Verknüpfungs-Logik ein *forward-filling* Mechanismus etabliert. Die Stammdaten sind in diesem Fall immer

[49] Quelle: Eigenerstellung

ab einem bestimmten Zeitpunkt gültig. Die Gültigkeit endet erst, wenn ein neuer, unterschiedlicher Wert geliefert wird – ggf. auch ein leerer Wert.[50]

Wie folgende Abbildung zeigt, beinhaltet dieses Verfahren jedoch Risiken.

Abbildung 10[51] - Risiken des Forward-Filling am Beispiel des Ratings

Abbildung 11[52] - Risiken des Forward-Filling durch fehlerhafte Daten

Diese Risiken müssen in den operativen Prozessen berücksichtigt werden. Hier ist große Sorgfalt gefragt, da diese Fehler sich negativ auf die Datenqualität auswirken und im schlimmsten Fall beim Nutzer der Daten zu Fehlentscheidungen führen.

Notwendige Metadaten werden im weiteren Verlauf der Arbeit beschrieben.

3.2 Integration der Daten in das System

Um die Daten verarbeiten zu können, müssen sie zuerst ins System gebracht werden. Diese Funktion übernimmt der *Download Spooler*[53].

---

[50] Ob leere Werte gespeichert werden oder nicht ist sehr stark von der Fachdomäne abhängig.

[51] Quelle: Eigenerstellung

[52] Quelle: Eigenerstellung

[53] In unserer Domäne heißt diese Komponente Download Spooler. Es sind jedoch beliebig viele andere Namen denkbar.

3.2.1 DESIGNSTRATEGIEN

Dieses Kapitel beschäftigt sich mit grundlegenden Spooler-Designkonzepten. Abhängig vom Design lassen sich unterschiedliche Ziele erreichen.

3.2.1.1 EIN SPOOLER PRO DATENQUELLE[54]

Dieses Modell sieht eine 1:1-Beziehung zwischen Spooler und Datenquelle vor.

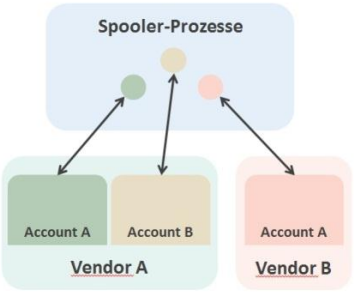


Abbildung 12[55] - Ein Spooler pro Datenquelle

Wenn ein Anbieter einen FTP-Server mit drei Accounts zur Verfügung stellt, ist pro Account ein eigener Spooler notwendig.

Die Spooler laufen in diesem Fall permanent, was eine einfache Überwachung ermöglicht. Die Spooler müssen dann allerdings über einen Heartbeat[56]-Mechanismus verfügen, der die korrekte Funktionalität an ein Monitoring System übermittelt.

Wenn ein Spooler-Prozess eine Fehlfunktion aufweist, sind die anderen Spooler davon nicht betroffen und laufen problemlos weiter. Je nachdem, wie die Persistenz-Schicht realisiert wird, kann es hier jedoch zu Problemen bezüglich der Skalierbarkeit oder zu einem SPOF kommen.[57]

---

[54] Unser Bestandssystem ist momentan nach diesem Konzept realisiert und es hat sich in der Praxis als durchaus tauglich erwiesen.

[55] Quelle: Eigenerstellung

[56] http://de.wikipedia.org/wiki/Cluster_Interconnect#Heartbeat_.28Cluster_Heartbeat.29

[57] Da die Spooler in unserem System auf demselben Filesystem arbeiten, egal auf welchem Knoten sie laufen, kommt es hier zu keinen Problemen. Bei exklusiven Filesystemen müssten die Daten jedoch ggf. erst migriert oder neu angefordert werden, bevor sie weiterverarbeitet werden können.

3.2.1.2 ORCHESTRIERTER SPOOLER-POOL

In diesem Modell gibt es eine zentrale, steuernde Instanz, die über einen Pool an Spoolern verfügt; den *Director*. Dieser verfügt über die gesamte Intelligenz und übergibt dem Spooler-Pool lediglich die Downloadaufträge. Die Spooler übermitteln dann ein Ergebnis an den Director, so dass dieser weiß, ob ggf. Probleme aufgetreten sind.

Der *Director* sollte auf jeden Fall als hochverfügbares Cluster realisiert werden, da im Falle eines Ausfalles keine Daten mehr in das System gelangen (SPOF).

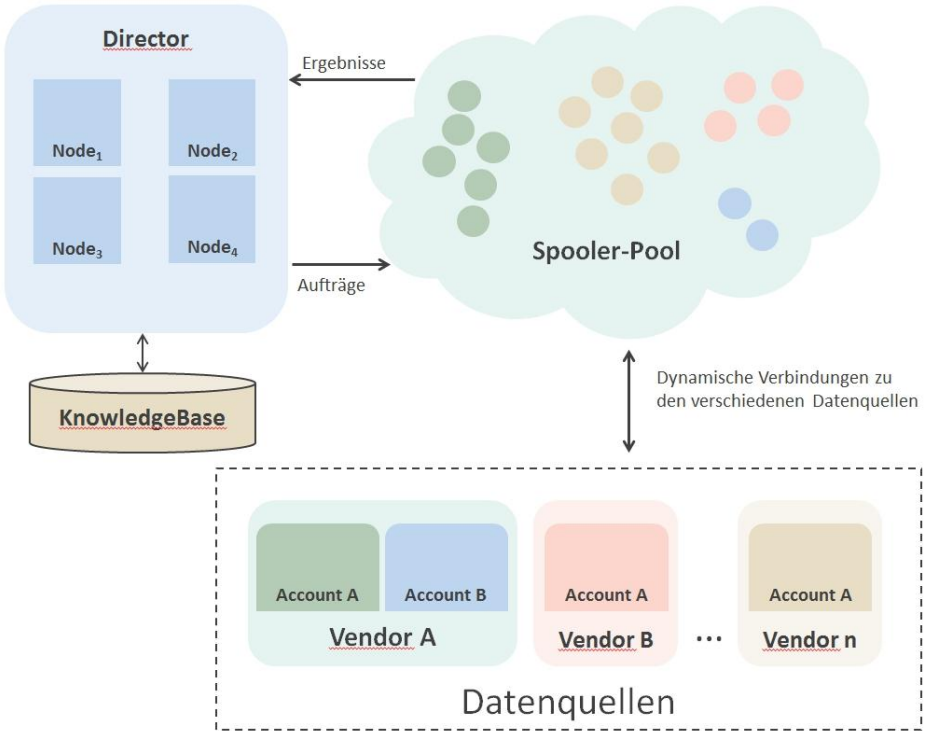


Abbildung 13[58] - Orchestrierter Spooler-Pool

Die unterschiedlichen Farben in *Abbildung 13 - Orchestrierter Spooler-Pool* stellen verschiedene physikalische Systeme dar. Dies soll illustrieren, dass der Pool über mehrere Systeme hinweg verteilt sein kann. Reichen die vorhandenen Kapazitäten nicht aus, können schnell und einfach neue Worker-Prozesse hinzugefügt werden.

[58] Quelle: Eigenerstellung

3.2.1.3 Weitere Abstrahierung

Man könnte sich einen Download- bzw. Upload-Spooler auch als *normalen* Job vorstellen. Der Job hieße in diesem Fall bspw. *getFile* bzw. *sendFile*. In diesem Fall müssten keine eigenen Spooler modelliert werden, sondern die entsprechenden Jobs müssten in einen entsprechend zu entwickelnden Gesamtkontext eingebettet werden.

3.2.1.4 Generischer Designvorschlag

Dieser Designvorschlag zeigt eine mögliche Implementierung. Wichtig ist die Unterteilung in *Connection*, *Configuration* und *TransferProcess*. Je nach Modellierung lassen sich hiermit alle vorgestellten Konzepte realisieren.

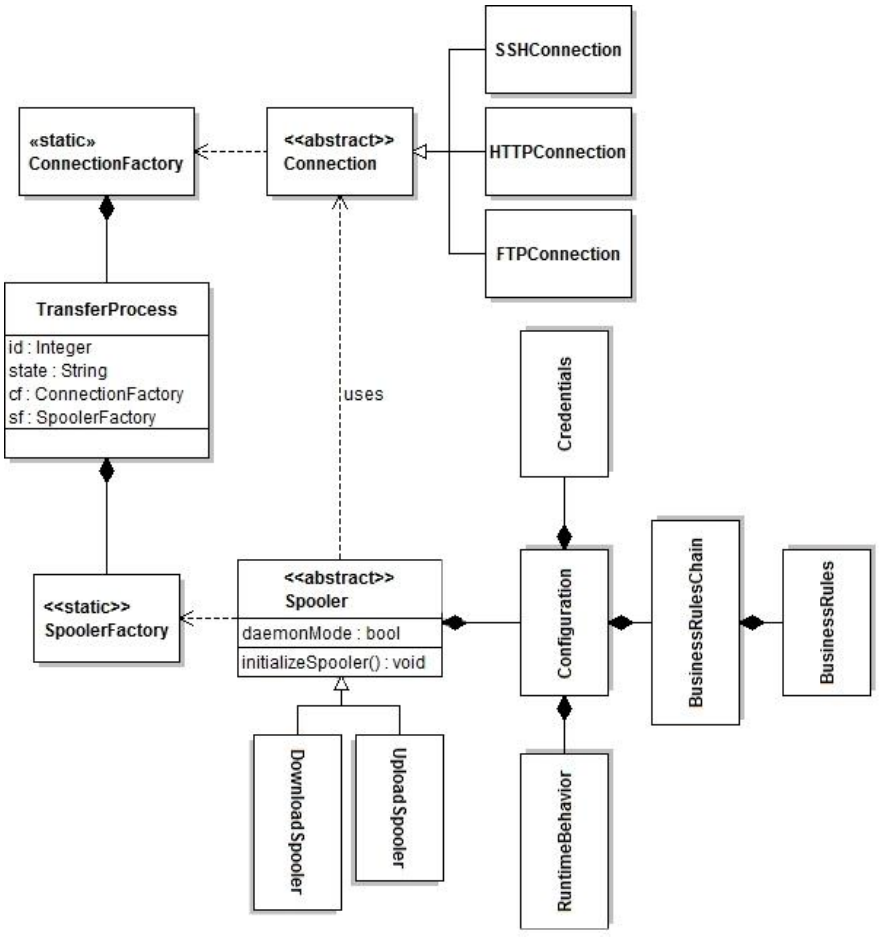


Abbildung 14[59] - Generischer Designvorschlag Spooler

[59] Quelle: Eigenerstellung

3.2.2 FILESYSTEMSTRUKTUR

Es ist bspw. aus Compliance-Gründen notwendig die unveränderten Anbieterdaten aufzubewahren. Dies ist außerdem im Falle einer späteren Problemanalyse hilfreich, wenn überprüft werden soll, ob die eigenen Anreicherungsprozesse sauber arbeiten.

Da sich mit der Zeit sehr viele Dateien im System ansammeln sollte eine übersichtliche Strukturierung auf Filesystem-Ebene vorgenommen werden.

Diese Unterteilung ermöglicht eine Überwachung auf niedriger Ebene. Sollte ein etabliertes Messaging bspw. ausfallen, könnte der Systemzustand trotzdem weiterhin explizit gemacht, und in einem Monitoring-Tool dargestellt werden.

3.3 PERSISTENZ DER DATEN

In *Kapitel 3.1.2* wurden bereits grundlegende Modelle der Datenhaltung angedeutet. In diesem Kapitel werden diese Ideen konkretisiert.

Die zentrale Komponente in Bezug auf die Persistenz der Daten ist der Loader[60]. Dieser Baustein speichert die Nutzdaten in der darunterliegenden Persistenz-Schicht ab.

Generell können hier zwei verschiedene Zielsetzungen verfolgt werden, Geschwindigkeit oder Flexibilität.

Wenn bspw. propritäre Datenbanken wie FAME im Einsatz sind, dann kann die Geschwindigkeit dadurch optimiert werden, dass der Loader in C++ geschrieben wird und direkt auf dem FAME CHLI[61] aufsetzt.

Flexibilität kann dadurch erreicht werden, dass eine Abstraktionsschicht eingezogen wird, die verschiedene Datenbank-Backends unterstützt. In diesem Fall wird allerdings

[60] In unserem System wird diese Komponente *Loader* genannt.

[61] C++ Host Language Interface

die Geschwindigkeit mit der die Daten in der Datenbank gespeichert werden geringer. Geringe Flexibilität hingegen kann zu einem Vendor-Lock-In führen[62].

Der Nutzer muss in diesem Fall abwägen, welches Ziel aus seiner Sicht wichtiger ist.

3.3.1 Persistenz in SimpleDB und Object-Store

In diesem Szenario werden Meta- und Stammdaten in SimpleDB und Zeitreihen in einem Object-Store gespeichert. Es gibt zwei generelle Implementierungen.

3.3.1.1 Coarse-Grained Modellierung

In diesem Modell werden die einzelnen Zeitreihenobjekte und Stammdaten in Containerobjekten gebündelt bevor sie abgespeichert werden. Zusätzlich enthalten die Objekte Informationen bezüglich der historischen Veränderung der Stammdaten.

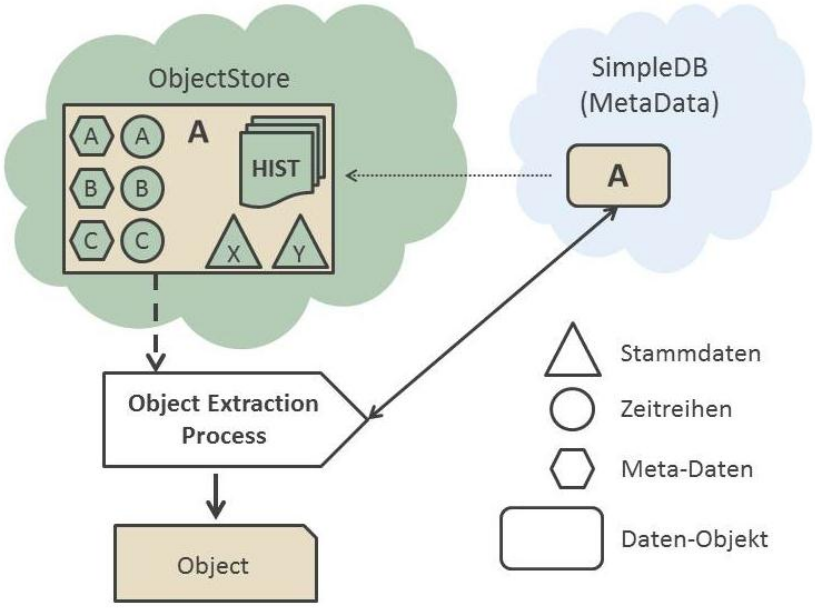


Abbildung 15[63] - Lesezugriff im Coarse-Grained Modell

[62] Ein Vendor-Lock-In muss nicht zwangsläufig negative Konsequenzen mit sich bringen. Im Gegenteil kann die bewusste Entscheidung für einen konkreten Anbieter sogar gewünscht sein. Probleme könnten auftreten, wenn die implementierte Lösung auf einen anderen Anbieter migriert werden soll.

[63] Quelle: Eigenerstellung. In diesem Modell befindet sich in SimpleDB lediglich eine Referenz auf das Objekt in S3. Wie in der Einleitung zum Kapitel *3.3.1* erwähnt, könnten die Metadaten (zusätzlich) auch in SimpleDB gespeichert werden. In *Abbildung 15* sind die Metadaten im Objekt enkapsuliert.

Das **Coarse-Grained Modell** hat seine Stärke im Bereich der AWS-Request. Da alle nötigen Zeitreihen in einem Objekt zusammengefasst sind, können diese mit einem Request aus der Persistenz-Schicht extrahiert werden. Wenn man bedenkt, dass es Objekte mit mehr als 200 Zeitreihen geben kann, wird die Bedeutung dieses Modells deutlich. Um ein Objekt um einen neuen Datensatz zu erweitern müssten hier im FGM pro Objekt mindestens 200 Lese-Requests und anschließend 200 Schreib-Requests an AWS gestellt werden.

3.3.1.2 Fine-Grained Modellierung

Dieses Modell sieht vor, dass jede Zeitreihe einzeln persistiert wird. Zur Erzeugung des jeweiligen Objektes werden die nötigen Metadaten aus SimpleDB extrahiert. Dort ist bspw. gespeichert welche Frequenz die Zeitreihe besitzt und unter welcher S3-URL das jeweilige Zeitreihenobjekt abgespeichert wurde. Zusätzlich werden die Stammdaten aus SimpleDB ausgelesen.

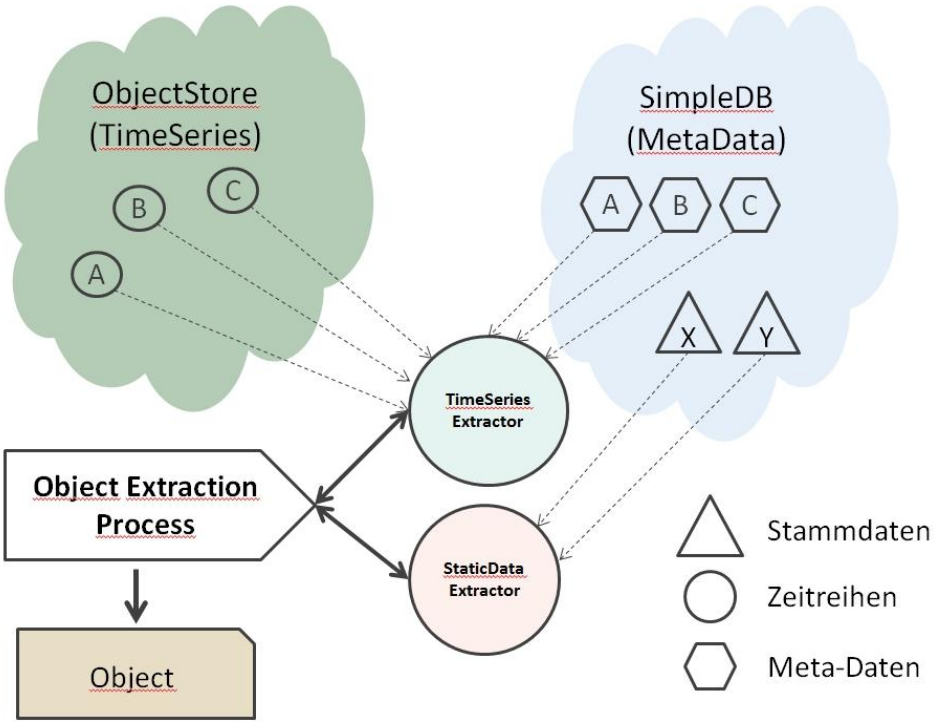


Abbildung 16[64] - Lesezugriff im Fine-Grained Modell

[64] Quelle: Eigenerstellung

Das **Fine-Grained Modell** hat seine Stärken im Bereich der Skalierbarkeit und Wartbarkeit, da jede Zeitreihe einzeln abgespeichert ist. Jede Zeitreihe kann somit einzeln bearbeitet und ggf. ausgetauscht werden. Die Speicherbelastung ist hier nur besser, wenn die Zeitreihen einzeln gelesen und geschrieben werden. Wenn das gesamte Objekt modelliert wird, ist der Speicherbedarf in diesem Modell ggf. etwas höher. In diesem Fall müssten erst alle einzelnen Zeitreihen in Objekte umgewandelt und anschließend in einem Containerobjekt gebündelt werden. Im CGM ist dieser Container bereits komplett persistiert. Da im FGM jede Zeitreihe ein eigenes Objekt ist, kann hier ein hoher Grad an Nebenläufigkeit erzielt werden.

3.3.1.3 Bewertung der Modelle

Beide Modelle haben Vor- und Nachteile, die individuell Bewertet werden müssen.

Tabelle 5 - Gegenüberstellung Fine- und Coarse-Grained Modell

Aspekt	Fine-Grained Modell	Coarse-Grained Modell
Anzahl AWS-Requests	-	+
Speicherbelastung	-/+	-/+
Skalierbarkeit	+	-
Wartbarkeit	+	-
Parallelisierungsgrad	+	-

Um das passende Modell auswählen zu können, müssen Beispielrechnungen angefertigt werden, welche die EC2-Berechnungsaufwände den SimpleDB-/S3-Requests gegenüberstellen. AWS bietet hierzu Online-Rechner zur Ermittlung der ungefähren monatlichen Kosten an[65].

3.3.2 Persistenz in NoSQL Datenbanken

Eine umfangreiche Übersicht zu den verschiedenen NoSQL-Datenbanken kann unter *http://nosql-database.org/* gefunden werden.

[65] http://calculator.s3.amazonaws.com/calc5.html

Für kleinere Systeme könnte SimpleDB ausreichend dimensioniert sein. Aufgrund der Limits[66] dieses Services, kommt dieses Konzept für große Marktdatensysteme nicht infrage.

Bei Nutzung einer NoSQL-Datenbank müssen die Zeitreihenkonzepte für diesen Datenbanktyp erst entwickelt werden[67]. Hier muss mit eindeutigen Identifiern gearbeitet werden, die eine Unterscheidung der Objekte ermöglichen und gleichzeitig aufschluß über die Eigenschaften geben[68].

3.3.3 Persistenz in relationalen Datenbanken

Dieses Modell wird im Rahmen dieser Arbeit nicht vertieft. Das Speichern der Daten als BLOBs in relationalen Datenbanken ist ein Ansatz, welcher auch in der Cloud realisierbar ist. AWS bietet einen eigenen, auf MySQL basierenden, Service an[69]. Es können im AWS-Umfeld aber auch andere Datenbanken wie bspw. ORACLE genutzt werden.

3.4 Mögliche Meta-Implementierungen

In diesem Unterkapitel werden Architekturen beschrieben, die als Basis der Implementierung des Finanzmarktdatensystems in Erwägung gezogen werden können. Der Fokus der Betrachtung liegt auf der Cloudnutzung. Die Auswahl der Modelle basiert u. a. auf der Prämisse, dass eine getrennte Datenhaltung und –verarbeitung bzw. Datenhaltung und -repräsentation keinen Sinn macht.

Würde die Präsentation der Daten in der Cloud liegen und die gesamte Verarbeitung im *On-Premise Bereich*, dann müssten die Daten zuerst in die Cloud repliziert werden um visualisiert werden zu können. Ein reines Monitoring hingegen könnte in der Cloud realisiert werden, obwohl das zugrundeliegende Informationssystem im eigenen

---

[66] Siehe http://docs.amazonwebservices.com/AmazonSimpleDB/latest/DeveloperGuide/SDBLimits.html

[67] Vgl. *Abbildung 8 - Konzept zur Abbildung von Zeitreihen in NoSQL Datenbanken, p. 33*

[68] Vgl. *Tabelle 3 - Konzept von Preisquellen und Snapshots, p. 31*

[69] Amazon Relational Database Service (Amazon RDS), http://aws.amazon.com/de/rds/

Rechenzentrum betrieben wird. Hier könnten bspw. Events in die Cloud gemeldet und dort aggregiert und grafisch repräsentiert werden. Die in den folgenden Unterkapiteln vorgestellten Szenarien können teilweise auch kooperativ genutzt werden.

3.4.1 ENTKOPPELUNG

Dieses Modell sieht eine Nutzung der Kommunikations-Services vor. Die lokal betriebenen Komponenten werden, über in der Cloud gehostete SQS-Queues, entkoppelt. Messaging erfolgt über SNS.

Der Vorteil ist die einfache Integration auf Basis des *Producer/Consumer Patterns*. Nachteile sind die Abhängigkeiten von der Verfügbarkeit der Internetverbindung sowie der Cloud-Plattform und die nötige Unterbrechung des Produktivbetriebs bei der Inbetriebnahme. Abhängig von der Implementierung des aktuellen Systems können die Komponenten ggf. Schrittweise umgestellt werden.

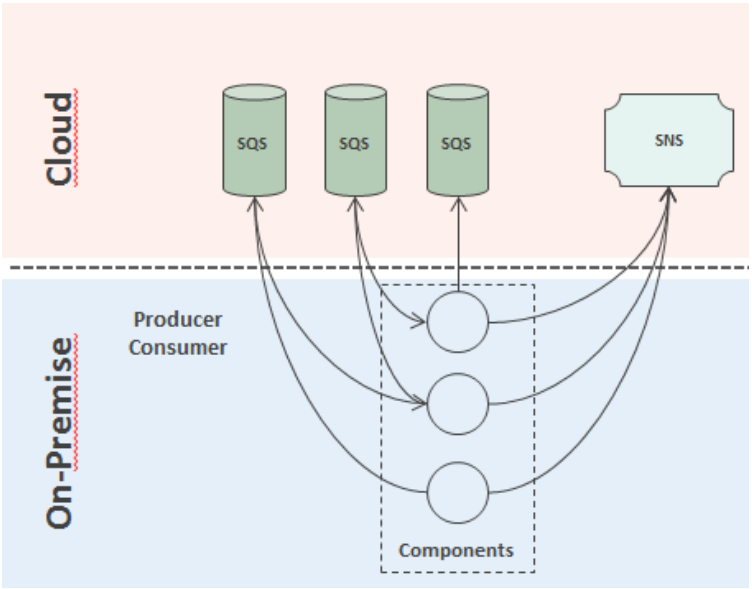


Abbildung 17[70] - Entkoppelung der internen Informationssysteme

[70] Quelle: Eigenerstellung

3.4.2 LANGZEITSPEICHERUNG

Die Integrationsaufwände dieses Ansatzes sind gering und es gibt kaum Nachteile. Die Implementierung der nötigen Programme kann ohne Unterbrechung des Produktivsystems erfolgen. Die Kosten sind außerdem transparent und gewähren, bei einem funktionierenden *Capacity Management*, Planungssicherheit.

Zu Problemen kann es kommen, wenn die in der Cloud gespeicherten Daten verloren gehen. Der einzige *Wehrmutstropfen* dieses Ansatzes ist, dass er keinen weiteren Mehrwert bringt.

Dieses Verfahren kann genutzt werden um erste Erfahrungen im Cloud-Umfeld zu erlangen, sowie um Vertrauen in die Kompetenzen des *Cloud-Providers* aufzubauen. Auf Basis dieser Grundlage könnten später weitere Cloud-Projekte umgesetzt werden.

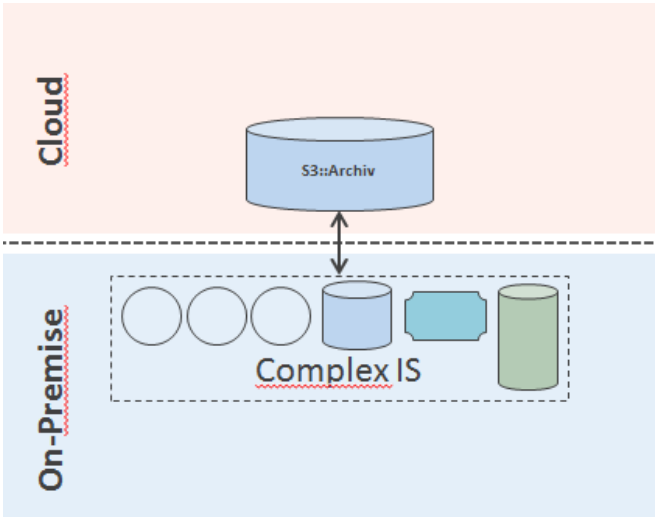


Abbildung 18[71] - Archivierung in der Cloud

3.4.3 BEDARFSORIENTIERUNG

Der Grundgedanke hinter diesem Ansatz ist es, bestimmte Aspekte der *BusinessLogic* in der Cloud zu replizieren. Überlasten im On-Site-Bereich können so über bedarfsorientierte Ressourcenallokation in der Cloud abgefangen werden.

[71] Quelle: Eigenerstellung

Der Vorteil dieses Ansatzes ist, dass er komplett unabhängig vom aktuellen Bestandssystem entwickelt werden kann. Die nötigen Teile können parallel in der Cloud abgebildet werden.

Nachteile sind die Replikation des Codes, welcher Synchron gehalten werden muss um keine fehlerhaften Ergebnisse zu produzieren sowie die zwangsläufige Datenmigration in die Cloud. Hier kann es zu Problemen bzgl. der Datenmenge oder des Datenschutzes kommen.

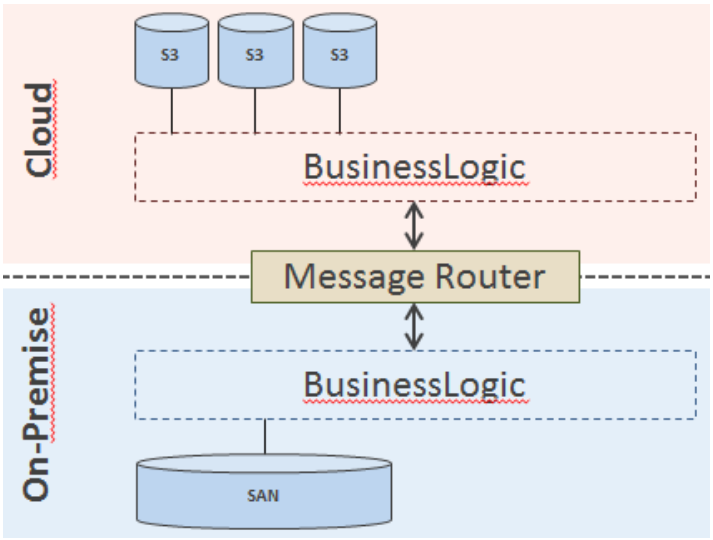


Abbildung 19[72] - Bedarfsorientierte Nutzung der Cloud

3.4.4 AUSLAGERUNG

Dieses Modell sieht eine komplette Auslagerung des Informationssystems in die Cloud vor.

Ein Nachteil ist, dass vor einer Nutzung das gesamte System in der Cloud abgebildet werden muss. Außerdem müssen Schnittstellen zu den weiteren im System befindlichen Applikationen entwickelt werden. Je nachdem wie die Integration dieser Systeme umgesetzt ist muss hier ggf. eine komplexe *Routing-Logik* entwickelt werden.

[72] Quelle: Eigenerstellung

Falls es sich um ein Bestandssystem handelt, müssen ggf. viele Systemteile komplett neu entwickelt werden. Je nach Komplexität der Kommunikationsbeziehungen zu anderen Applikationen, kann der Integrations- und Testaufwand sehr hoch werden.

Ein Vorteil dieser Lösung ist, dass das System direkt auf die Cloud zugeschnitten werden kann. Aspekte wie Skalierbarkeit und Transparenz können im Entwicklungsprozess von Beginn an berücksichtigt werden.

Besonders interessant ist diese Alternative, wenn ein komplett neues System entwickelt werden muss.

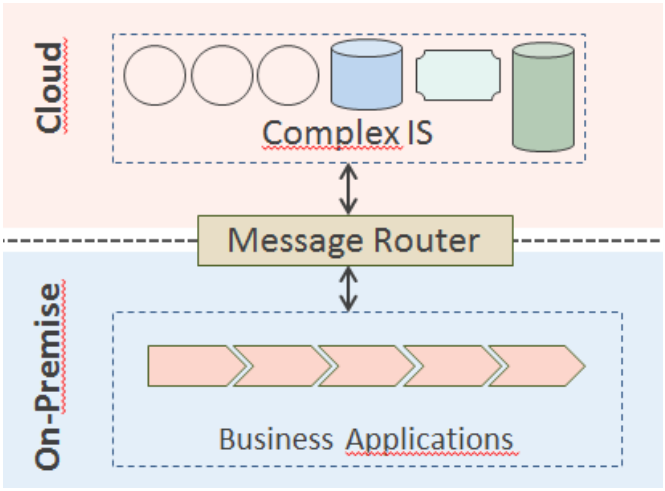


Abbildung 20[73] - Komplette Auslagerung eines Informationssystems in die Cloud

3.5 SCHNITTSTELLEN

Der Zugriff auf das System sollte über verschiedene Protokolle ermöglicht werden. Wichtig ist in diesem Zusammenhang das Thema der Lizensierung.

Um Daten im kommerziellen Umfeld nutzen zu können, sind in der Regel Lizenzverträge notwendig, die erst mit den jeweiligen Anbietern verhandelt werden müssen. Ein Nutzer darf nur auf Daten zugreifen, für die er eine Lizenz besitzt. Die Lizenzverwaltung und sollte aus diesem Grunde zentral erfolgen.

[73] Quelle: Eigenerstellung

Damit ein hoher Grad an Nachvollziehbarkeit gewährtleistet ist[74], sollte eine spezielle Zugriffsschicht[75] entwickelt werden. Diese Zugriffschicht muss jede Anfrage validieren und anhand einer Rechtestruktur zulassen oder ablehnen. Außerdem sollte diese Zugriffschicht alle Anfragen, sowie die getroffenen Entscheidungen, protokollieren.

Folgende Abbildung illustriert den DAL auf einer abstrakten architektonischen Ebene.

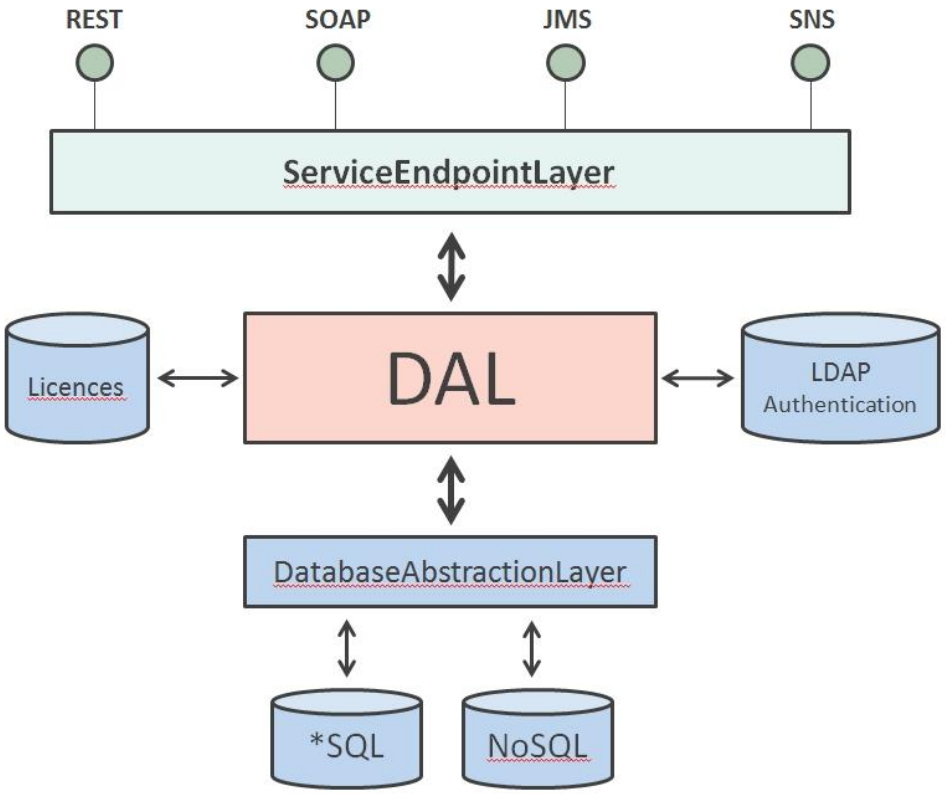


Abbildung 21 - Architektonische Integration eines DAL

[74] Beispielsweise für Audits.
[75] Data Access Layer (DAL)

4 HAUPTTEIL – GESAMTKONZEPT

Alle großen Entdeckungen passieren zufällig.

Murphy's Technologie-Gesetze

In diesem Kapitel wird ein generisches Gesamtkonzept entwickelt. Die Realisierung basiert auf dem Coarse-Grained Modell, welches in Kapitel 3.3.1.1 vorgestellt wurde. Großer Wert wird darauf gelegt die Abhängigkeitsbeziehungen zu spezifischen Anbietern etwas aufzulockern. Obwohl es noch keine anbieterübergreifenden Standards gibt, gibt es erste Entwicklungen in diese Richtung[76]. Seit Juli 2011 gibt es außerdem die *Open Cloud Initiative*[77].

Zuerst werden theoretische Modelle *auf der grünen Wiese* entwickelt und anschließend in ein Gesamtkonzept überführt für welches dann die ungefähren Kosten ermittelt werden. Die Kosten komplexer IT-Architekturen werden normalweise nicht auf Basis einer einjährigen Nutzung berechnet. Da das hier entwickelte Modell als Entscheidungsgrundlage einen ungefähren Überblick über die Kosten liefern soll kann diese Vereinfachung vorgenommen werden.

Im Anschluß werden ausgesuchte Aspekte der Lösung anhand einer prototypischen Implementierung überprüft.

Sollten vereinzelte EC2-Instanzen überdimensioniert erscheinen, liegt das daran, dass das Wachstum der Daten direkt von Begin an berücksichtigt wird. Sollten sich im späteren Verlauf Instanzen als überdimensioniert erweisen, können sie durch kleinere Instanzen ersetzt werden. In diesem Fall, darf jedoch nicht mit *Reserved Instanzen* gearbeitet werden, da diese Instanzen für eine bestimmte Vertragsdauer bezogen

[76] Siehe http://libcloud.apache.org/

[77] Siehe http://www.cloud-practice.de/news/offene-standards-fuer-freiheit-den-clouds

werden und die Pauschale nicht erstattet wird. Im Zweifel empfiehlt es sich jedoch den Break-Even-Point zwischen der kleineren *ODI* und der größeren *RI* zu berechnen.

4.1 Abstrakte Modellierung der Umwelt

Um die Anbieter-Unabhängigkeit zu ermöglichen, ist es notwendig, dass die hausinternen Applikationen nicht direkt mit den Cloud-Services kommunizieren. Diese enge Kopplung würde bei einem Anbieterwechsel zwangsläufig zu komplexen Migrationsszenarien führen.

Aus diesem Grund sieht das in dieser Arbeit entwickelte Modell eine Abstraktion, durch Nutzung von SDO (Service Data Objects) und DAS[78] (Data Access Service), vor. Genaue Informationen über dieses Konzept können dem *JSR 235*[79] entnommen werden.

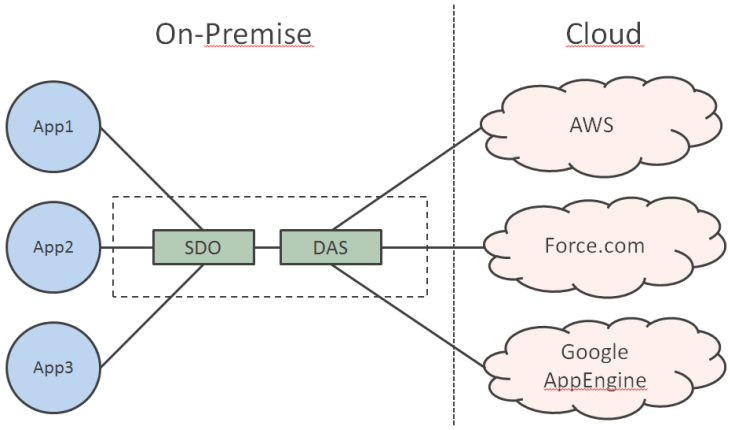


Abbildung 22[80] - Generische Architektur

Die internen Applikationen arbeiten ausschließlich mit den standardisierten SDOs. Dies ermöglicht eine einheitliche Implementierung im gesamten Applikations-Portfolio[81] unabhängig vom jeweiligen Anbieter und Backend.

[78] Diese Komponente wird manchmal auch DMS (Data Mediation Service) genannt.
[79] Java Specification Request 235, http://jcp.org/en/jsr/detail?id=235
[80] Quelle: Eigenerstellung
[81] Die einheitliche Implementierung ist hier auf eine spezifische Fachdomäne bezogen.

Um dieses Konzept zu verdeutlichen, wird in folgender Abbildung der *Data Access Service* näher betrachtet. Dieses Design ist nur ein Vorschlag und weitere Implementierungen sind denkbar.

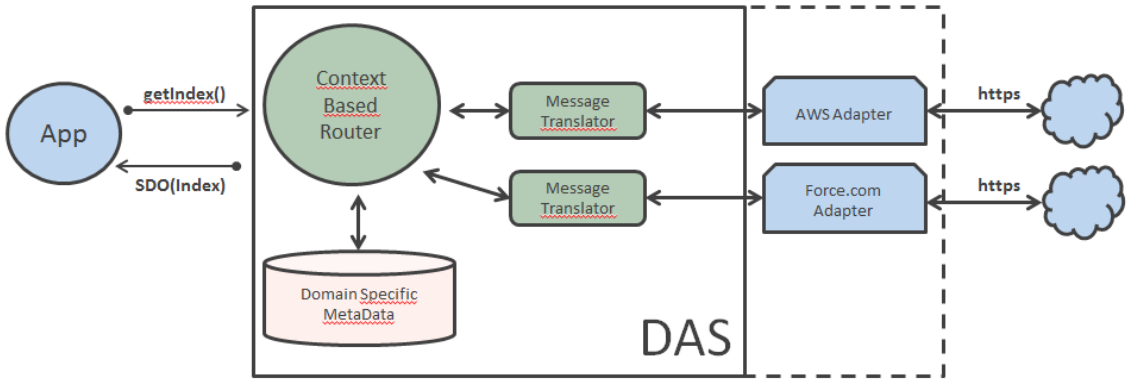


Abbildung 23[82] - Mögliche Implementierung des Data Access Service

In dieser Implementierung werden die internen Anfragen von einem Kontext basierten Router entgegengenommen und anhand von Metadaten an die entsprechenden Nachrichten-Übersetzter weitergeleitet, die die Nachrichten an die jeweilige Schnittstelle anpassen. Beispielsweise könnte die getIndex()-Funktion aus AWS-Sicht eine AWS-ID und einen AWS-KEY benötigen, wobei eine lokale HBase Instanz diese Informationen nicht benötigt usw.

Die ex-ante Einbeziehung der Metadaten ist notwendig, da der *DAS* nicht wissen kann, in welcher *Cloud* die Daten anzufragen sind – und ggf. sogar welcher Service hierfür genutzt werden muss.

Die Metadaten-Datenbank könnte als *in-memory Datenbank*[83] implementiert werden, um eine optimale Performance zu gewährleisten. Voraussetzungen hierfür sind Hardware mit einer ausreichenden Menge an Arbeitsspeicher und intelligente Cache-Management Strategien.

---

[82] Quelle: Eigenerstellung, in Anlehnung an G. Schmutz, D. Liebhart und P. Welkenbach, Service-Oriented Architecture: An Integration Blueprint, Packt Publishing Ltd., 2010.

[83] Hierfür bietet sich bspw. memcached an: http://memcached.org/

AWS bietet momentan[84] in der US-East (Virginia) Region eine Beta-Phase des ElastiCache-Service an. Hierbei handelt es sich um einen in-memory Cache.

4.2 Gesamtmodell auf Basis von AWS-Diensten

In diesem Kapitel wird ein komplettes Modell des Finanzmarktdatensystems entwickelt, welches konkrete AWS-Dienste und Güteklassen berücksichtigt. Folgende Abbildung zeigt die Integration des Marktdatensystems in die Unternehmensumwelt.

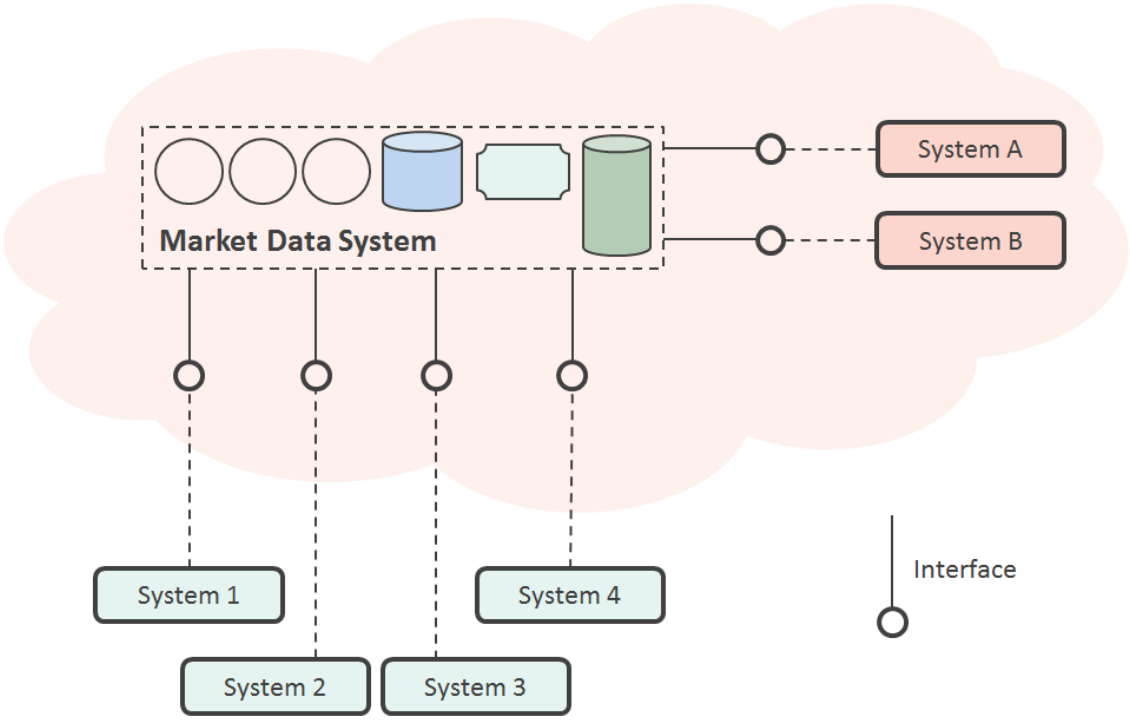


Abbildung 24[85] - Integration des Marktdatensystems in die Unternehmensumwelt

Auf das Marktdatensystem kann von externen Systemen aus, über definierte Schnittstellen[86], zugegriffen werden. Diese externen Systeme können sowohl innerhalb als auch außerhalb von AWS realisiert sein.

---

[84] 10. September 2011

[85] Quelle: Eigenerstellung

[86] Vgl. Abbildung 21, p. 40

4.2.1 INFRASTRUKTUR

Die infrastrukturelle Basis des Systems bildet EC2[87]. Aufgrund der Optimierung der variablen Kosten, durch kalkulierbare Fixkosten, werden in diesem Konzept *Reserved Instances (RI)* genutzt. Folgendes Beispiel zeigt die erzielbaren Kostenvorteile.

Tabelle 6[88] - Kostenvergleich Reserved Instanz (RI) vs. On-Demand Instanz (ODI)

	m1.xlarge (*RI*)	**m1.xlarge (*ODI*)**
Kosten pro CPU-Stunde	$0,32	$0,76
Initiale Kosten (1yr Term)	$1820	$0
Kosten der Nutzung pro Jahr basierend auf der CPU-Stunde und 8760 Stunden.	$2.803,2	$6.657,6
Gesamtkosten[89] pro Jahr	*$4.623,2*	*$6.657,6*
Verhältnis RI/ODI	*0,6944*	-

Die Kostenersparnis der *RI* gegenüber der *ODI* liegt bei über 30%. In [5][90] werden die jährlichen Gesamtkosten dieses Instanz-Typs, als ODI, mit ca. USD 8000 beziffert, wobei diese Zahl bereits weitere geschätzte Kosten mit berücksichtigt.

Im hier ausgearbeiteten Modell werden ausschließlich *RI* genutzt. Jede *RI* wird anhand der jährlichen Nutzungskosten zzgl. eines Zuschlages von 20%[91] kalkuliert. Als Basis der *RI* werden Verträge mit einer einjährigen Laufzeit angenommen.

Folgende Tabelle gibt eine Übersicht über die geschätzten Kosten, der in dieser Arbeit genutzten Instanz-Typen.

---

[87] Eine Übersicht hinsichtlich der angebotenen Instanz-Typen ist auf der AWS-Webseite zu finden: http://aws.amazon.com/de/ec2/instance-types/

[88] Die genutzten Zahlen stammen von der offiziellen AWS-Seite (http://aws.amazon.com/de/ec2/pricing/) und wurden am 15.08.2011 ermittelt. Aufgrund der schwankenden Wechselkurse nutze ich die gelisteten Preisangaben in USD. Die Preise beziehen sich auf die Nutzung von Linux-/UNIX-Instanzen.

[89] Die Gesamtkosten beziehen sich lediglich auf die CPU-Nutzung. Die Kosten für Datentransfer und zusätzliche, nicht in der Instanz enthaltene Storage-Kapazitäten sind nicht berücksichtigt.

[90] R. Linton, Amazon Web Services: Migrating your .NET Enterprise Application, Packt Publishing, 2011, p. 37

[91] Der Zuschlag berechnet sich wie folgt: Eine Micro Instanz wird in [5] mit $260 angegeben. Die reinen Nutzungskosten ergeben jedoch einen Preis von $219 ($0,025 x 8760h). Hieraus ergibt sich ein angenommener Zugschlag von ca. 20%.

Tabelle 7 - Kostenmodell für Reserved Instances, Region EU (Ireland)

Instanztyp	Fixkosten	Variable Kosten	Gesamtkosten
t1.micro	$54	$105,12	$159,12
m1.large	$910	$1681,92	$2591,92
m1.xlarge	$1820	$3363,84	$5183,84
c1.xlarge	$1820	$3363,84	$5183,84
m2.2xlarge	$2650	$5045,76	$7695,76

4.2.1.1 Infrastruktur-Konzept Persistenz

Das System wird in mehrere Pfade (engl. Lanes) unterteilt. Diese Unterteilung ist sinnvoll, da es Daten gibt, die besonders wichtig sind[92]. Diese Daten werden in der *Exclusive Processing Lane* verarbeitet, der Rest in der *Default Processing Lane*. Folgende Abbildung stellt den Aufbau des Systems abstrakt dar, um im Anschluss die ungefähren jährlichen Kosten ermitteln zu können.

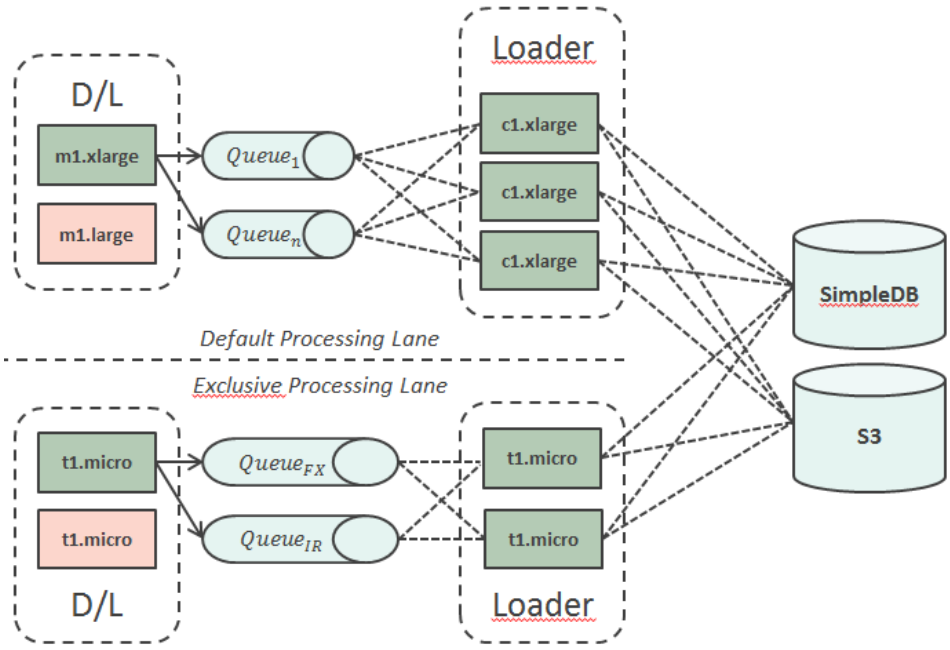


Abbildung 25[93] - High-Level IT-Architektur der Persistenz-Ebene

Die ungefähren Kosten für die Umsetzung dieser Persistenz-Ebene sind in folgender Tabelle dargestellt.

[92] Sehr wichtig sind bspw. Wechselkurse (FX Rates) und Zinssätze (Interest Rates)

[93] Quelle: Eigenerstellung

Tabelle 8 - Kostenschätzung: Persistenz-Ebene EC2

Instanz Typ	Anzahl	Fixkosten	Variable Kosten
t1.micro	4	$216	$420,48
m1.large	1	$910	$1681,92
m1.xlarge	1	$1820	$3363,84
c1.xlarge	3	$5460	$10091,52
Gesamtkosten pro Jahr	9	$8406	$15557,76

Die ins System laufende Datenmenge beläuft sich auf ca. 3,2GB pro Tag. Auf ein Jahr hochgerechnet, ergibt sich eine Datenmenge von ca. 1,152TB[94].

Tabelle 9 - Kostenschätzung: Persistenz-Ebene Storage (EBS)

	Bedarf in GB	Kosten pro GB pro Monat	Jährliche Kosten
EBS pro Instanz	150 x 5 = 750	$0,10	$900
EBS pro Micro-Instanz	75 x 4 = 300	$0,10	$360
EBS zur Datenhaltung	288	$0,10	$345,6
Gesamtkosten pro Jahr	1.338	$0,10	$1.605,6

Die Archivierung in S3 findet unabhängig davon statt. Folgende Tabelle gibt einen Überblick über die Kosten der Archivierung.

Tabelle 10 - Kostenschätzung: Datenarchivierung in S3

Monat	Datenaufkommen *kumuliert*	Kosten *Standard Storage*	Kosten *Reduced Redundancy Storage*
1 - 10	960GB	$1.344	$892,8
11-12	192GB	$240	$178,56
Gesamtkosten 1. Jahr	1.152GB	$1.584	$1.071,36
Die nächsten ca. 40 Jahre jährlich[95]	1.152GB	$1.440	$956,16

Die Kosten zur Datenarchivierung müssen, um die maximalen Kosten zu ermitteln, doppelt berechnet werden, da neben der Archivierung auch die Instrumente in S3 gespeichert werden und hier ggf. nur Teilmengen der Originaldaten genutzt werden. Da GET- und PUT-Requests in S3 extra berechnet werden, müssen die hier anfallenden Kosten ebenfalls betrachtet werden. Folgende Tabelle gibt einen Überblick.

[94] Grundlage der Berechnung sind 360 Tage, da es Börsenfeiertage gibt an denen das Datenvolumen geringer ausfällt. Aufgrund von Wochenenden ist diese Zahl vermutlich noch zu hoch angesetzt, aber für die Planung gibt dies einen größeren Puffer und kompensiert ggf. mittelfristiges Wachstum. Das angegebene Datenvolumen bezieht sich auf bereits mit bzip2 komprimierte Daten.

[95] Kein Datenwachstum angenommen.

Tabelle 11 - Kostenschätzung: Kommunikation mit dem S3-Service

Request	Anzahl pro Tag	Kosten pro Jahr	Beschreibung
PUT	7.500.000	$25.732,5	Persistenz der Instrumente
GET	7.500.000	$2.737,5	Extraktion der Instrumente
GET	1.000.000	$365	Zugriff durch Folgeprozesse
PUT	2.500	$0,91	Archivierung der Originaldaten
		$28.835,91	Gesamtkosten pro Jahr

Die SimpleDB Kosten können nach folgendem Verfahren berechnet werden.

„Rohe Byte-Angaben (GB) aller Element-IDs + 45 Byte pro Element + Rohe Byte-Angaben (GB) aller Attributnamen + 45 Byte pro Attributname + Rohe Byte-Angaben (GB) aller Attributwertpaare + 45 Byte pro Attributwertpaar … ist das Ergebnis (in GB) entsprechend mit 0,275 USD zu multiplizieren.“[96]

Tabelle 12 - Meta- und Stammdatenbedarf in SimpleDB

Attributname	Beschreibung	Rohe Byte-Angaben Attributname	Rohe Byte-Angaben Attributwert
UID	Abstrahierte UniqueID	3	32
LUS	Letztes Stammdaten Update	3	17
LUP	Letztes Preisdaten Update	3	17
LUF	Letztes Update aus File (MD5)	3	16
VT	Variante	2	5
CUR	Währung (Currency)	3	3
RTG	Rating	3	9
MB	Laufzeitband (Maturity Bucket)	2	6
ISIN	ISIN Identifier	4	12
SEDOL	SEDOL Identifier	5	7
VID	Vendor UniqueID	3	16
ISSUE	Bezeichnung des Instruments	5	96
ISSUER	Herausgeber des Instruments	6	32
OS	Object Store	2	2
OSL	Object Store Link	3	128
MSD	Markt Sektor Definition	3	12
PS	Preisquelle (Pricing Source)	2	4

Zur Bestimmung der zu speichernden Datenmenge werden die in *Tabelle 12* dargestellten Meta- und Stammdaten in *SimpleDB* angenommen. Aufgrund der Nutzung des Coarse-Grained Modells werden alle weiteren spezifischen Daten in den Objekten gespeichert.

[96] [Online]. Available: http://aws.amazon.com/de/simpledb/ [Zugriff am 17. August 2011]

Die Datenmenge bzgl. der Attributwerte sind Schätzwerte. Es wird davon ausgegangen, dass ca. 7,5 Mio. Instrumente im System gespeichert werden.

Auf dieser Basis lassen sich die Kosten wie folgt ermitteln:

$$\text{Gesamtkosten}_{\text{proJahr}} = \text{Datenmenge}_{\text{Gigabyte}} \text{ x } 0{,}275\text{USD}$$

$$\text{Datenmenge}_{\text{Gigabyte}} = \text{Metadaten}_{\text{Gigabyte}} + \text{Nutzdaten}_{\text{Gigabyte}}$$

Da die hier genutzten Metadaten nur ca. 815 Byte besitzen, werden sie mit 0 angesetzt, was faktisch bedeutet, dass die zu speichernde Datenmenge der Menge der Nutzdaten entspricht. Die Indizes werden zu den Nutzdaten gezählt.

Es seien I die Anzahl der im System gespeicherten Instrumente und
A die Anzahl der gespeicherten Attributnamen, dann gilt

$$\text{Nutzdaten}_{\text{Gigabyte}} = \frac{\text{I x A x } 45_{\text{Byte}} \text{ x } \left(\text{Setgroesse}_{\text{Byte}} + 1\right)}{1024^3}$$

$$= \frac{7500000 \text{ x } 17 \text{ x } 45 \text{ x } (367 + 1)}{2122317824} = 994{,}86\text{GB} \approx 995\text{GB}$$

$$\text{Gesamtkosten}_{\text{proJahr}} = 995 \text{ x } \$0{,}275 \approx \$274$$

Die zur Speicherung in SimpleDB nötigen Kosten belaufen sich somit auf ca. 3.288 USD jährlich.

4.2.1.2 Infrastruktur-Konzept Processing-Ebene

In der Processing-Ebene werden ebenfalls extra Ressourcen für die *Exclusive Processing Lane* eingeplant. Die Verarbeitungs-Ebene wird als Rechnerfarm realisiert. Es gibt Instanzen die erhöhte CPU- und erhöhte Memory-Kapazitäten bereitstellen.

Es wird angenommen, dass jede verarbeitende Instanz einen EBS-Bedarf von 500GB besitzt, die Micro Instanzen werden jeweils mit 50GB geplant.

Jede Instanz besitzt *n-Queues* über die die Jobs angenommen werden. Der Einfachheit halber werden diese, in der folgenden Abbildung, als einzelne Queue dargestellt.

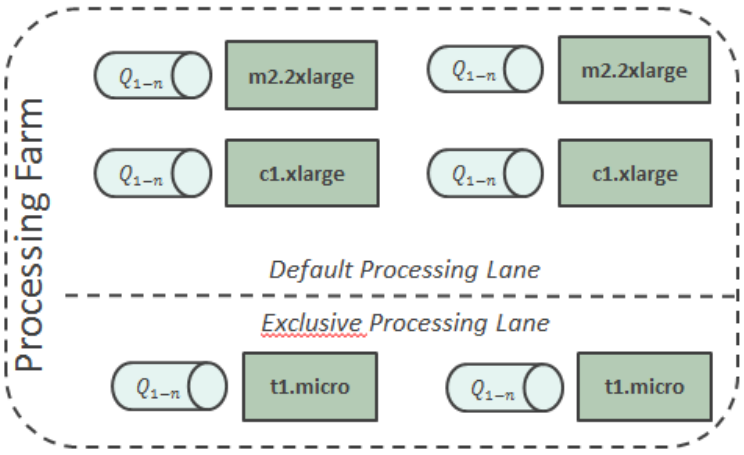


Abbildung 26[97] - High-Level IT-Architektur der Processing-Ebene

In diesem Konzept wird angenommen, dass der Director die CPU- bzw. Memory-Intensiven Job jeweils den geeigneten Instanzen zuordnet. Aus diesem Grunde sind Instanzen mit erhöhter CPU-Leistung bzw. mit erhöhter Memory-Kapazität vorhanden.

Basierend auf diesem Modell ergibt sich folgende Kostenschätzung[98].

Tabelle 13 - Kostenschätzung: Processing-Ebene EC2

Instanz Typ	Anzahl	Kosten
t1.micro	2	$318,24
c1.xlarge	2	$10.367,68
m2.2xlarge	2	$15.391,52
Gesamtkosten pro Jahr	*6*	*$26.077,44*

Innerhalb der Processing-Ebene wird von folgender Storage-Anforderung ausgegangen.

[97] Quelle: Eigenerstellung

[98] Es werden nur Fixkosten ausgewiesen, da die geplanten Instanzen alle *live* sind.

Tabelle 14 - Kostenschätzung: Processing-Ebene Storage (EBS)

	Bedarf in GB	Kosten pro GB pro Monat	Jährliche Kosten
EBS pro Instanz	125 x 4 = 500	$0,10	$600
EBS pro Micro-Instanz	50 x 2 = 100	$0,10	$120
Gesamtkosten pro Jahr	*600*	*$0,10*	*$720*

4.2.1.3 Infrastruktur-Konzept Steuerung und Monitoring

Der *Director* wird in diesem Konzept als Cluster implementiert. Zusätzlich gibt es ein Monitoring-System, welches über eine CEP [99] -Komponente verfügt und die gesammelten Nachrichten aggregiert. Zusätzlich wird eine Datenbank benötigt, die die Job-Hierarchien enthält.

Folgende Abbildung zeigt die benötigten Komponenten im Gesamtkontext.

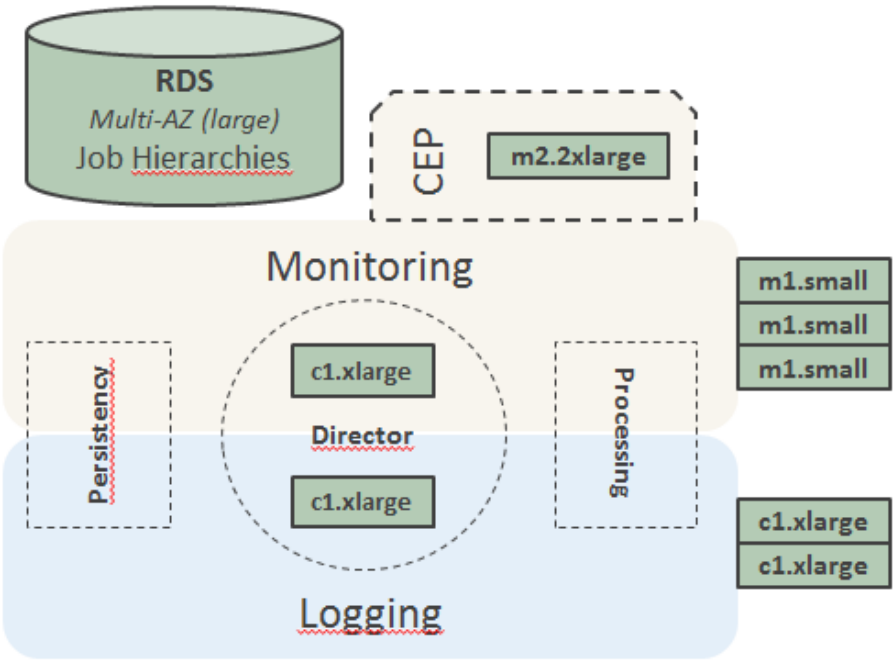


Abbildung 27[100] - High-Level IT-Architektur der Meta-Funktionalität

Hieraus ergeben sich folgende Kosten, wobei wie in *Tabelle 13* nur Fixkosten ausgewiesen werden.

[99] http://de.wikipedia.org/wiki/Complex_Event_Processing
[100] Quelle: Eigenerstellung

Tabelle 15 - Kostenschätzung: Meta-Ebene EC2

Instanz Typ	Anzahl	Kosten
m1.small	3	$477,36
c1.xlarge	4	$20.735,36
m2.2xlarge	1	$7.695,76
Gesamtkosten pro Jahr	*8*	*$28.908,48*

Die geschätzten Storage-Kosten sind in folgender Tabelle zusammengefasst.

Tabelle 16 - Kostenschätzung: Meta-Ebene Storage (EBS)

	Bedarf in GB	Kosten pro GB pro Monat	Jährliche Kosten
EBS pro Instanz (Monitoring)	300 x 3 = 900	$0,10	$1.080
EBS pro Instanz (Logging)	500 x 2 = 1.000	$0,10	$1.200
EBS pro Instanz (Director)	200	$0,10	$240
EBS (CEP) [101]	1.000	$0,10	$1.200
Gesamtkosten pro Jahr	*3.100*	*$0,10*	*$3.720*

Die CEP-Komponente kann ggf. entfallen. Um eine sinnvolle Entscheidung treffen zu können, müssen die Einsparungspotenziale im Bereich der operativen Personalkosten gegengerechnet werden.

Da die fertigen Daten nach den Anreicherungsprozessen noch zu den Kunden exportiert werden müssen und die Zielsysteme außerhalb der AWS Cloud liegen, fallen hierfür zusätzliche Kosten an[102]. Anhand der evaluierten Datenmenge ergeben sich folgende Kosten.

4.2.2 KOMMUNIKATIONS-KONZEPT

Da zur Kommunikation innerhalb des Systems sowohl Queues als auch Messages genutzt werden sollen, muss das ungefähre Volumen abgeschätzt werden um die Plankosten ermitteln zu können.

[101] Die CEP-Komponenten kann ggf. entfallen. Um eine sinnvolle Entscheidung treffen zu können, müssen die Einsparungspotenziale im Bereich der operativen Personalkosten gegengerechnet werden. Die Berechnung ist nicht Teil dieser Arbeit. Hier wird davon ausgegangen, dass die CEP-Komponente implementiert wird.

[102] Die Kommunikation zwischen EC2 und SNS ist nur innerhalb einer Region kostenfrei. Bei einer Region überschreitenden Kommunikation werden die gleichen Kosten erhoben, wie bei einer externen Kommunikation.

Der Granularitätsgrad, welcher dem Messaging zugrunde gelegt wird, muss aus den Anforderungen abgeleitet werden.

Das feingranulare Extrem wäre bspw. eine Meldung pro Instrument welches persistiert wurde bzw. eine Meldung für jeden Zwischenschritt innerhalb der Processing-Ebene. Das andere Extrem wäre eine Benachrichtigung auf Prozessniveau.

In diesem Konzept wird eine grobe Granularität gewählt und die Details werden in Logfiles abgebildet.

Auf Basis einer Prozessanzahl von täglich 3.500, ergeben sich folgende Kosten für die SNS Nutzung, wobei τ der Granularitätsfaktor ist, welcher in diesem Beispiel auf 1 gesetzt wird und somit keine Auswirkung hat. Da die ersten 100.000 SNS API-Anfragen im Monat kostenlos sind, werden diese subtrahiert.

$$\$_{\mathrm{SNS}} = \frac{(\mathrm{SNS}_{\mathrm{read}} + \mathrm{SNS}_{\mathrm{write}} + \mathrm{SNS}_{\mathrm{error}}) * 365 * \tau - 100000}{100000} * 0{,}06$$

$$= \frac{(3500_{\mathrm{read}} + 3500_{\mathrm{write}} + 500_{\mathrm{error}}) * 365 * 1 - 100000}{100000} + 0{,}06$$

$$= \frac{2637500}{100000} * 0{,}06 = \$1{,}5825$$

Die jährlichen Kosten für die Nutzung von SNS belaufen sich auf ca. $1,60.

Die SOAP Requests werden aus externen Systemen geschickt und die Antworten verlassen somit die AWS-Cloud. Zudem werden Daten aus dem System extrahiert und zu externen Systemen übertragen. Da AWS das erste GB pro Monat nicht abrechnet und das Datenvolumen monatlich unter einem GB liegt, fallen hier keine Kosten an. Sollte der Transfer zunehmen, werden bis zu einer Grenze von 10TB, $0,120 pro GB berechnet.

4.2.3 KOSTENPLANUNG DES DEPLOYMENT-KONZEPTS

Neben den bisher erarbeiteten Aspekten sind zusätzlich eine Entwicklungs- und eine Systemintegrationsumgebung notwendig. Da in dieser Arbeit davon ausgegangen wird, dass die Daten in der Cloud liegen, können neue Ansätze in Betracht gezogen werden.

Zuerst ist es notwendig ein *Version-Control-System*[103] bereitzustellen, welches die einzelnen Artefakte enthält. Es wird folgender *Prozess* zugrunde gelegt, der eine komplette Trennung von Entwicklung/System-Integration und Produktion gewährleistet.

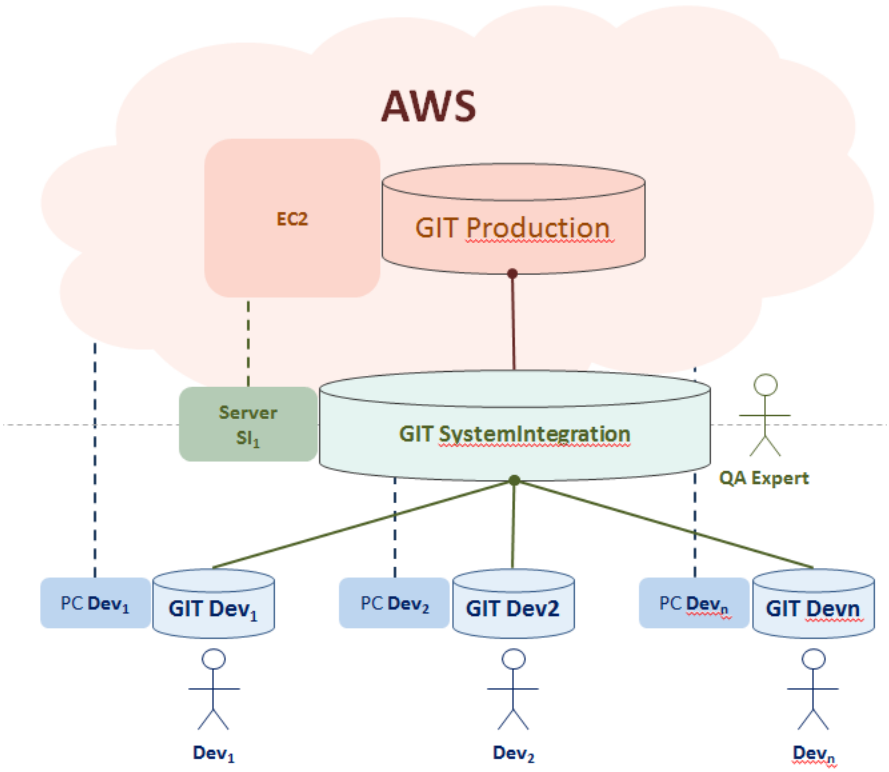


Abbildung 28[104] - Abstrakte Darstellung des Deployment Prozesses

Es wird davon ausgegangen, dass lediglich das Produktivsystem in der Cloud liegt. Die Entwicklung findet komplett in der lokalen Umgebung statt. Anschließend werden die ersten Tests in der lokalen Systemintegrationsumgebung durchgeführt. Hierdurch

[103] Dieses Konzept sieht die Nutzung von GIT vor. http://git-scm.com/
[104] Quelle: Eigenerstellung

können die bereits bestehenden Systeme weiterhin genutzt werden. Wenn es sich um Änderungen mit nicht kalkulierbarem Risiko handelt, müssen diese zuerst in der Cloud-Umgebung getestet werden. Hierfür werden die Änderungen in die Cloud-Systemintegrationsumgebung überführt und erst nach erfolgreichem Testbetrieb die Produktionsumgebung übernommen.

Die Entwickler-PCs sowie der Systemintegrations-Server haben Zugriff auf die AWS-Cloud. Ein schreibender Zugriff auf die produktiven Daten ist unbedingt zu verhindern.

Neuentwicklungen werden zuerst auf dem lokalen Entwickler-PC getestet und werden dann in das Systemintegrations-Repository übertragen. Hier prüft ein QA-Experte[105] auf der Systemintegrations-Umgebung, ob die Code-Basis funktional einwandfrei ist und überträgt den Code anschließend in das produktive Repository. Nach einer Übergabe an das Operations-Team, welches die Produktions-Umgebung verantwortet, werden die Änderungen dann kontrolliert in die Produktions-Umgebung überführt.

Da das Marktdatensystem in einer sehr instabilen Umwelt angesiedelt ist, lassen sich fixe Deployment-Zyklen nicht umsetzen. Es muss jederzeit auf Veränderungen reagiert werden können. Für die täglich nötigen Anpassungen und Erweiterungen hat sich an dieser Stelle P*rototyping*[106] bewährt. Zu diesem Zweck ist es sinnvoll *RI* vorzuhalten. Folgende Tabelle gibt einen Überblick über die geplanten Instanzen und Kosten.

Tabelle 17 - Kostenschätzung: Systemintegrationsumgebung

Instanztyp	Anzahl	Fixkosten	Variable Kosten	Gesamtkosten
t1.micro	1	$54	$105,12	$159,12
m1.large	2	$1.820	$3.363,84	$5.183,84
Gesamtkosten pro Jahr	*3*	*$1.874*	*$3.468,96*	*$5.342,96*

Als nötigen EBS-Speicherplatz werden 300MB pro Instanz angenommen. Die Aufschlüsselung der Kosten von 1.080 USD pro Jahr können *Tabelle 16* entnommen werden.

[105] Dieser QA-Experte muss ein Software-Engineer mit Expertenwissen aus der Fachdomäne sein.
[106] http://de.wikipedia.org/wiki/Prototyping_%28Softwareentwicklung%29

Die maximalen Instanz-Kosten der Systemintegrationsumgebung belaufen sich auf 6.622,96 USD.

4.2.4 Aggregierte Kostenplanung

Die in Kapitel 4.2.1 ermittelten Kosten werden in diesem Kapitel zusammengefasst und um weitere Aspekte ergänzt. Es wird davon ausgegangen, dass alle reservierten Instanzen permanent in Betrieb sind. Somit können die maximal anfallenden Kosten ermittelt und bewertet werden.

Das Bestandssystem kostet jährlich ca. 455.000 EUR. Um einen vergleichbaren Preis in USD zu ermitteln verwende ich den durchschnittlichen Close-Wert [107] des Währungspaares EUR/USD in der Periode vom 01.01.2000 bis zum 04.09.2011. Mit dem Durchschnittswert von 1,2129 ergeben sich jährliche Kosten von ca. 551.869 USD.

Tabelle 18 - Kostenschätzung: Gesamtsystem (aggregiert)

Ebene	Persistenz-Ebene	
Persistenz	EC2 Instanzen	$23.963,76
	EBS	$1.605,6
	SimpleDB	$3.288
	S3	$30.978,63
Processing	EC2 Instanzen	$26.077,44
	EBS	$720
Meta	EC2 Instanzen	$28.908,48
	EBS	$3.720
	RDS	$4.125,96
Systemintegrationsumgebung	EC2 Instanzen	$5.342,96
	EBS	$1.080
	Gesamtkosten pro Jahr (gerundet)	*$129.811*

Das in der Cloud realisierte Marktdatensystem wäre um über 75% günstiger als das Bestandssystem. Dies entspricht einer jährlichen Ersparnis von ca. 422.058 USD[108].

[107] Quelle: http://www.global-view.com/forex-trading-tools/forex-history/index.html

[108] In dieser Arbeit wurden Migrationskosten der bestehenden Lösung in die Cloud nicht berücksichtigt. Die wirkliche Ersparnis ergibt sich durch Gegenüberstellen der Ersparnis und der Migrationskosten. Im

4.3 AUSGEWÄHLTE PROTOTYPISCHE IMPLEMENTIERUNGEN

Der im Rahmen dieser Arbeit entwickelte Prototyp soll überprüfen ob die Speicherung der Instrumente in SimpleDB und S3 eine sinnvolle Lösung ist. Zu diesem Zweck wird ein stark vereinfachtes statisches Modell entwickelt. Im prototypischen Umfeld gelten folgende Prämissen:

- (TimeSeries)Configuration Objekte sind statisch und werden nicht dynamisch anhand der in SimpleDB gespeicherten Informationen erzeugt.
- Es werden nur DenseSeries der Frequenz daily unterstützt.
- TimeSeries können nur fortgeschrieben und nicht reorganisiert werden. Das Startdatum ist somit nicht veränderbar.
- Die Zeitreihen speichern werte vom Typ String.

Dieses vereinfachte Modell erlaubt eine prototypische Implementierung, die eine erste Einschätzung in Bezug auf die Eignung ermöglicht. Im Rahmen dieses Versuches spielt es keine Rolle ob die Konfigurationsparameter aus SimpleDB stammen oder nicht. Damit das Konzept jedoch schlüssig ist, wird die Modellierung der Datenstrukturen in SimpleDB trotzdem betrachtet.

Der implementierte Prototyp wird folgenden Ablauf ausführen:

1. Instrument erzeugen
2. Instrument serialisieren, Base64 enkodieren und in S3-Bucket speichern
3. Instrument aus S3-Bucket extrahieren, dekodieren und als Objekt initialisieren
4. Korrektheit des Instruments validieren

Rahmen der Migrationskosten muss auch die Mitarbeiterentwicklung berücksichtigt werden, die durch diesen Paradigmenwechsel notwendig ist.

4.3.1 GRUNDLAGEN

Der Prototyp ist in JAVA implementiert und nutzt zur Kommunikation mit S3 die Bibliothek *jets3t*[109]. Die Klassen und Methoden sind gut dokumentiert und es gibt viele Codebeispiele[110]. Ein weiterer Grund, weshalb ich *jets3t* gewählt habe ist, dass neben S3 auch der *Google Storage Service*[111] unterstützt wird. Hierdurch ergeben sich flexible Möglichkeiten hinsichtlich der Datenhaltung. Außerdem ist dieses Potenzial im Hinblick auf einen potentiellen Vendor-Lock-In sinnvoll. Als Basis wird folgendes, stark vereinfachte, Klassenmodell genutzt.

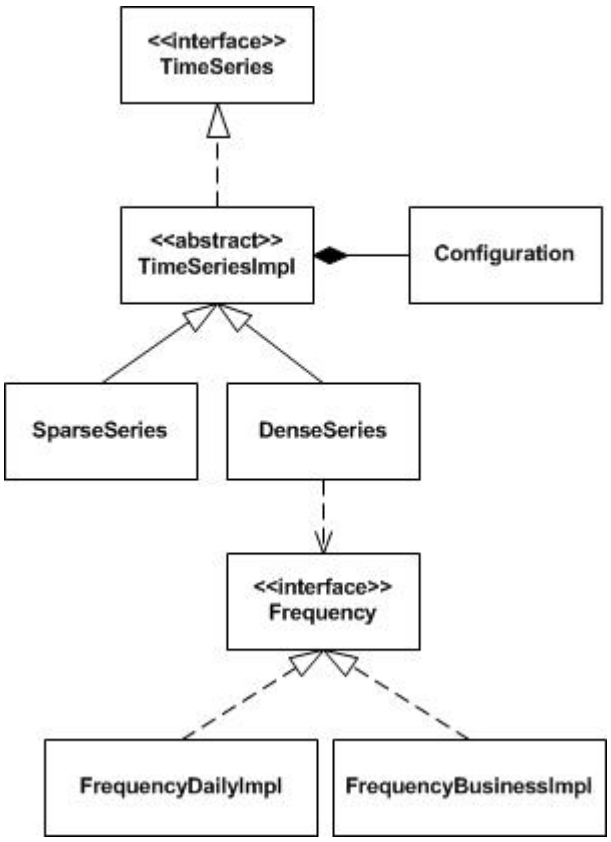


Abbildung 29[112] - UML Diagramm: TimeSeries

[109] http://jets3t.s3.amazonaws.com/index.html
[110] http://jets3t.s3.amazonaws.com/toolkit/code-samples.html
[111] http://code.google.com/intl/de-DE/apis/storage/
[112] Quelle: Eigenerstellung

Die folgende Abbildung illustriert den Versuchsaufbau.

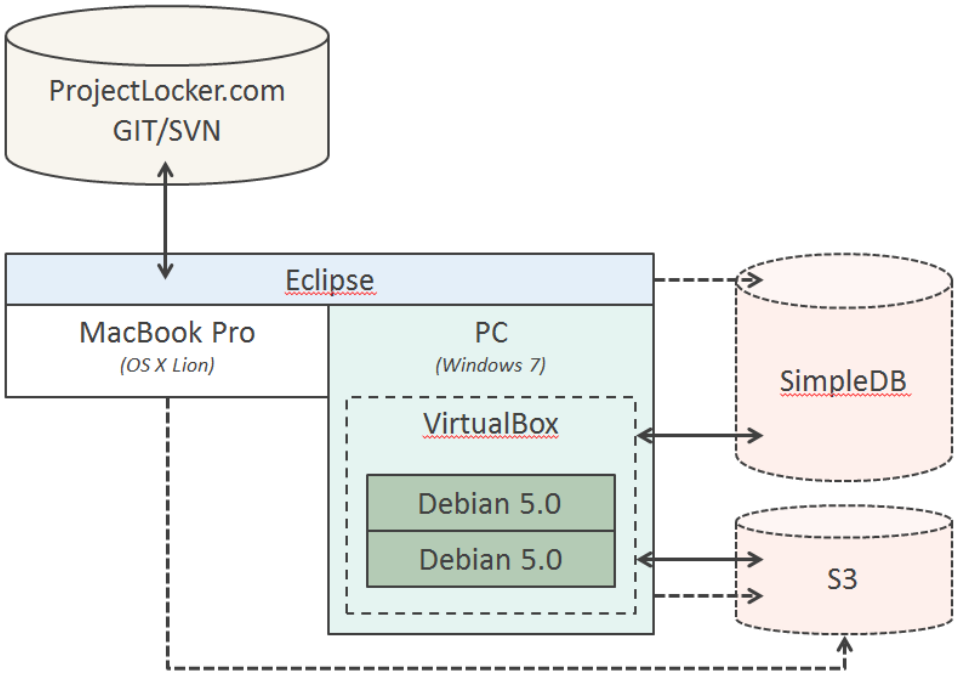


Abbildung 30[113] - Prototypischer Versuchsaufbau

4.3.2 SimpleDB Modell

Damit die Instrumente in S3 gefunden werden können, sind einige zusätzliche Informationen zu hinterlegen. Zusätzlich ist es sinnvoll, diverse Metadaten zu speichern, die von Post-Prozessen genutzt werden können. Eine Auswahl an Metadaten wurde bereits in *Tabelle 12* aufgelistet.

Ein zentraler Baustein des Finanzmarktdatensystems ist das Inhaltsverzeichnis, welches es erlaubt Instrumente zu finden. Dieses Verzeichnis wird folgend *Catalog* genannt. Der *Catalog* umfasst alle Anbieter und Klassen von Daten. Die folgende Tabelle gibt einen Überblick über mögliche[114] Bestandteile des *Catalog*.

[113] Quelle: Eigenerstellung

[114] Der konkrete Bedarf ist immer von den Anforderungen abhängig, weshalb der Catalog hinsichtlich seiner Komplexität variieren kann.

Tabelle 19 - Catalog-Layout in SimpleDB

Attributname	Beschreibung
UniqueID	Einzigartiger Identifier des Instruments
InstrumentName	Name des Instruments bspw. *JPM EMU InvestmentGrade*
Issuer	Herausgeber des Instruments
Domain	SimpleDB Domain dieses Instruments, welche auch den Anbieter enthält, bspw. *JPM_INDEX*
InstrumentClass	Klasse des gespeicherten Instruments, bspw. Index, Bond, etc.
ISIN	International Securities Identification Number
RIC	Reuters Instrument Code
SEDOL	Stock Exchange Daily Official List
CUSIP	Committee on Uniform Security Identification Procedures
WKN	Wertpapierkennnummer
{Metadaten}	Siehe *Tabelle 12, p. 54*
ObjectLocation	S3-URL des gespeicherten Objekts
ObjectS3Bucket	S3-Bucket des gespeicherten Objekts, falls S3 genutzt wird
ObjectStorageVendor	Bspw. *AWS-S3* oder *GoogleStorageService*
Availability	Erstes verfügbares Datum des Instruments (nicht Datum der ersten Speicherung)

Zusätzlich wird pro Domain eine Meta-Daten-Domain angelegt. In Bezug auf *Tabelle 19* würde diese Domain JPM_INDEX_META heißen. Diese Tabelle enthält Informationen über die in den Objekten gespeicherten Zeitreihen. Die folgende Tabelle verdeutlicht dieses Konzept anhand von ein paar Beispielen.

Tabelle 20 - Domain-Metadaten in SimpleDB

Fact	Description	TSClass	Frequency	Version
CLOSE	Letzer Handelskurs	DenseSeries	daily	1
ASK	Letzer Ask Preis	DenseSeries	daily	1
NAME	Name des Instruments	SparseSeries	daily[115]	1

Die Version ist notwendig, da sich die Anforderungen ggf. über die Zeit hinweg ändern können und die Objekte chronologisch konsistent sein müssen.

4.3.3 Ausgewählte Code Beispiele

Dieses Unterkapitel zeigt einige Ausschnitte aus dem Prototypen und dazu korrelierende Logmeldungen. Im folgenden Unterkapitel werden diese Codefragmente dann in ein übergeordnetes Konzept eingebettet.

[115] Die Default-Frequenz bei Sparse-Series ist daily.

Da im Prototyp die in der Einleitung dieses Kapitels genannten Prämissen gelten, sind die Beispiele nicht so dynamisch, wie sie in einem realen Szenario wären.

4.3.3.1 Erzeugen einer Zeitreihen-Konfiguration

Innerhalb des Prototyps wird lediglich der Name der Zeitreihe übergeben, da davon ausgegangen wird, dass es nur die Zeitreihen-Klasse *DenseSeries* und die Frequenz *daily* gibt.

```
Configuration c1 = new Configuration("CLOSE");

System.out.println("  Configuration c1 has the following values:");
System.out.println("     FACT:        " + c1.getFact());
System.out.println("     FREQUENCY:   " + c1.getTimeSeriesFrequency());
System.out.println("     SERIESCLASS: " + c1.getTimeSeriesClass());
```

Console Output:

```
Configuration c1 has the following values:
   FACT:        CLOSE
   FREQUENCY:   daily
   SERIESCLASS: DenseSeries
```

4.3.3.2 Erzeugen einer Zeitreihe

Die Zeitreihe nutzt ein Konfigurationsobjekt zur Erzeugung. Da es in diesem Modell pro Zeitreihe ein eigenes Configuration-Objekt geben muss, wäre dies im produktiven Betrieb nicht praktikabel. Eine Lösungsmöglichkeit wird in Kapitel *4.3.3.4* skizziert.

```
TimeSeries ts1 = new DenseSeries(c1);

System.out.println("  getting metadata startDate = " +
  ts1.getMetaData("startDate"));
System.out.println("  getting metadata notAvailable = " +
  ts1.getMetaData("notAvailable"));
```

Console Output:

```
getting metadata startDate = 20110801
metaDataStore doesn't contain the requested value for key notAvailable
getting metadata notAvailable = null
```

4.3.3.3 Erzeugen und aktualisieren eines Instrumentes

Das Instrument bekommt die UniqueID, die Instrumenten-Klasse sowie eine ArrayList mit TimeSeries-Objekten übergeben. Im realen Szenario müsste das Instrument aus einer UniqueID sowie einer *SimpleDB-Domain* heraus erzeugt werden können. Wie dies aussehen kann wird in Kapitel *4.3.3.4* beschrieben.

```
ArrayList<TimeSeries> ds = new ArrayList<TimeSeries>();
ds.add(ts1); ds.add(ts2);

Instrument idx = new Instrument("JIEMU_IG_TRADED.01TO03","Index",ds);
idx.printSummary();
```

Console Output:

```
Instrument object summary:
  UniqueID       : JIEMU_IG_TRADED.01TO03
  InstrumentClass: Index
  TimeSeries overview:
      CLOSE is a DenseSeries
        includes:
          STATIC START
      YIELD is a DenseSeries
        includes:
          STATIC START
```

Die Zeitreihen werden mit dem Wert „STATIC START“ initialisiert, damit sie nicht leer sind. Im Prototypen gibt es keine Startwerte, da die Objekte nicht aus Datenfiles erzeugt werden. Folgender Test soll zeigen, dass die tägliche Frequenz berücksichtigt wird – dies geschieht durch null-Werte in den Zeitreihen. Außerdem wird versucht eine nicht vorhandene Zeitreihe namens TEST zu aktualisieren.

```
idx.addElementToTimeSeriesByDate("CLOSE", 2011, 8, 2, "102.34");
idx.addElementToTimeSeriesByDate("CLOSE", 2011, 8, 3, "104.23");
idx.addElementToTimeSeriesByDate("CLOSE", 2011, 8, 4, "103.19");
idx.addElementToTimeSeriesByDate("CLOSE", 2011, 8, 5, "102.96");
idx.addElementToTimeSeriesByDate("CLOSE", 2011, 8, 8, "106.64");
idx.addElementToTimeSeriesByDate("YIELD", 2011, 8, 7, "105.10");
idx.addElementToTimeSeriesByDate("TEST", 2011, 8, 9, "106.75");
```

Console Output:

```
store 102.34 to TimeSeries CLOSE
store 104.23 to TimeSeries CLOSE
store 103.19 to TimeSeries CLOSE
store 102.96 to TimeSeries CLOSE
store 106.64 to TimeSeries CLOSE
store 105.10 to TimeSeries YIELD
Key {TEST} not member of HashMap::series
Instrument object summary:
  UniqueID       : JIEMU_IG_TRADED.01TO03
  InstrumentClass: Index
  TimeSeries overview:
CLOSE is a DenseSeries                  YIELD is a DenseSeries
  includes:                               includes:
    01.08.2011 STATIC START                 01.08.2011 STATIC START
    02.08.2011 102.34                       02.08.2011 null
    03.08.2011 104.23                       03.08.2011 null
    04.08.2011 103.19                       04.08.2011 null
    05.08.2011 102.96                       05.08.2011 null
    06.08.2011 null                         06.08.2011 null
    07.08.2011 null                         07.08.2011 105.10
    08.08.2011 106.64
```

4.3.3.4 KOMMUNIKATION MIT S3

Im folgenden Unterkapitel ist die gesamte S3-Kommunikation innerhalb des Prototyps dargestellt. Erst wird ein Instrument serialisiert und lokal[116] in ein File geschrieben, nachdem es über den *ObjectHandler* Base64 encodiert wurde. Anschließend wird dieses File in einen S3-Bucket kopiert. Das serialisierte Objekt wird dann aus S3 heruntergeladen und es wird nach der Decodierung durch den *ObjectHandler* ein Instrument erzeugt.

```
//Instrument is serialized, Base64 encoded and locally stored
ObjectHandler oh = new ObjectHandler();

String instrumentString = oh.objectToString(idx);
byte[] instrumentByteArray = instrumentString.getBytes();

FileOutputStream fos = new FileOutputStream("JIEMU_IG_TRADED.01TO03.ser");
fos.write(instrumentByteArray);
fos.close();

// S3 connection is initialized and S3Bucket is set
String awsAccessKey = "<secret>";
String awsSecretKey = "<secret>";

AWSCredentials awsCredentials = new AWSCredentials(awsAccessKey,
awsSecretKey);

S3Service s3Service = new RestS3Service(awsCredentials);
S3Bucket testBucket = new S3Bucket("de.virtuosi.dipl.test");

// File is locally read and then transferred to S3Bucket
File fileData = new File("JIEMU_IG_TRADED.01TO03.ser");
S3Object fileObject = new S3Object(fileData);
fileObject.setStorageClass(S3Object.STORAGE_CLASS_REDUCED_REDUNDANCY);

s3Service.putObject(testBucket, fileObject);

// Object is extracted from S3 and an Instrument is initialized
S3Object objectComplete = s3Service.getObject(testBucket,
"JIEMU_IG_TRADED.01TO03.ser");

// Validity of extracted String is ensured by comparison of the MD5 hashes
String textData =
ServiceUtils.readInputStreamToString(objectComplete.getDataInputStream(),
"ISO-8859-1");
boolean valid = objectComplete.verifyData(textData.getBytes("ISO-8859-1"));

// Object is created from the Base64 decoded string
Object instrumentFromS3 = oh.ObjectFromString(textData);

((Instrument)instrumentFromS3).printSummary();
```

[116] Der Umweg über das lokal gespeicherte File kann in einem realen Szenario entfallen. Da im Rahmen des Prototyps bspw. auch XML und JSON evaluiert wurden, war diese Vorgehensweise zu Debugging zwecken sinnvoll.

4.3.4 ERWEITERTER ABLAUF

In Bezugnahme zum SimpleDB-Modell aus *Kapitel 4.3.2* könnte der Prozess wie folgt ablaufen. Dieser Prozess bezieht bereits einen nicht näher spezifizierten Lock-Mechanismus mit ein, der ein paralleles verändern der Daten verhindern soll. Unabhängig vom Ergebnis werden die Locks am Ende des Prozesses freigegeben. Folgeaktivitäten wurden in diesem Ablauf-Plan nicht berücksichtigt.

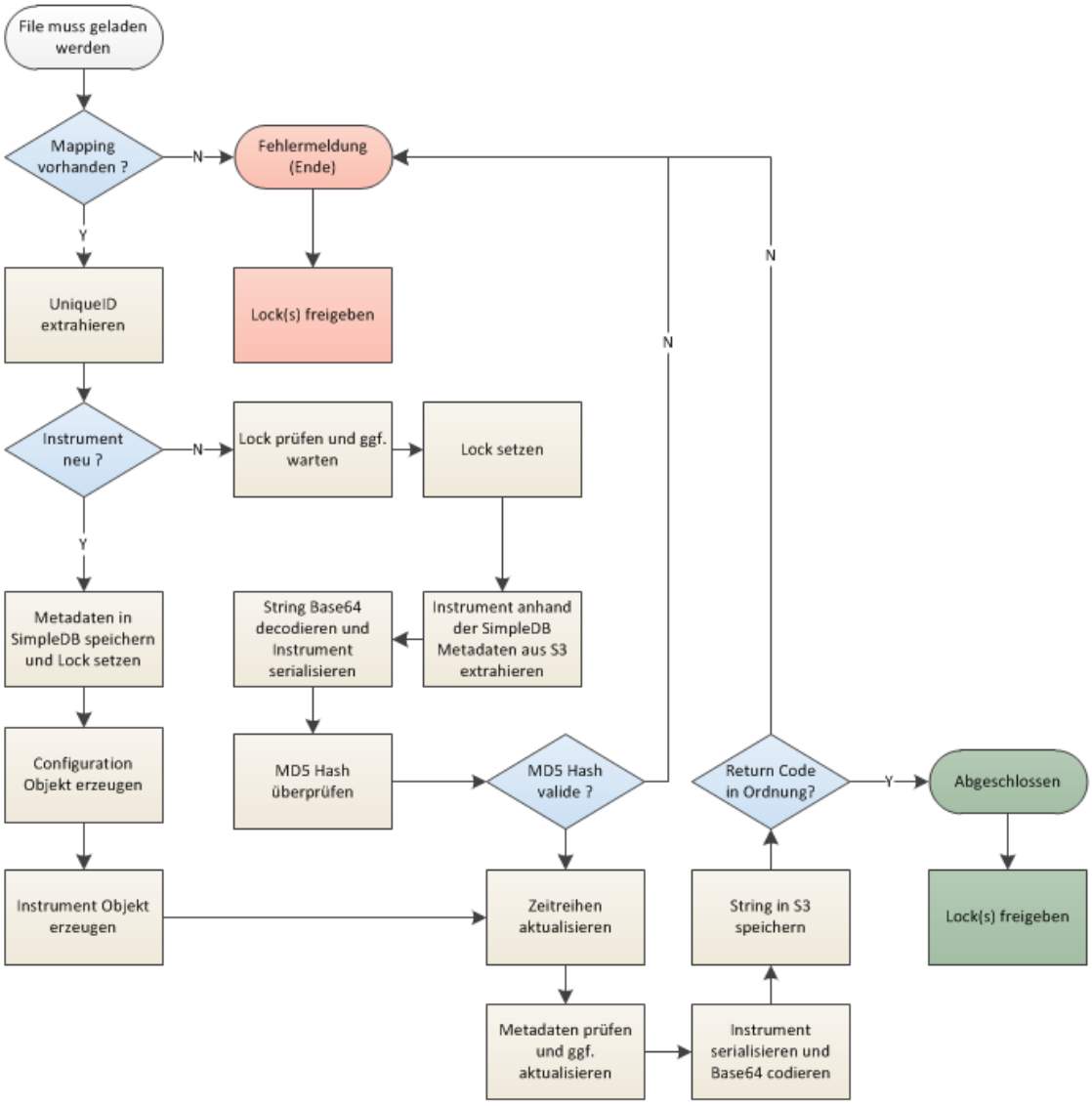


Abbildung 31[117] - Programm-Ablauf-Plan des Ladeprozesses

[117] Quelle: Eigenerstellung

5 SCHLUSS

Er ist noch weit vom Schluß entfernt,

Er hat das Ende nicht gelernt.

Johann Wolfgang von Goethe, (1749 - 1832), deutscher Dichter der Klassik, Naturwissenschaftler und Staatsmann

Quelle : »Zahme Xenien«

5.1 ZUSAMMENFASSUNG

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Maecenas porttitor congue massa.

5.2 BEWERTUNG

Die Arbeit hat gezeigt, dass die Cloud aus reinen Kostengesichtspunkten, bezogen auf die IT-Infrastruktur, eine interessante Option sein kann. Viele Fragen, die es im Vorfeld der Cloudnutzung zu klären gibt, wurden in der Arbeit nicht behandelt. Hierzu zählen Fragen der Daten- und Informationssicherheit sowie Fragen bezüglich der Mitarbeiterentwicklung. Das neue Paradigma führt nicht nur zu einem zwangsweisen Umdenken bei den Entwicklern, sondern es müssen ggf. Systemteile völlig neu entwickelt werden. Hierzu müssen die Entwickler Erfahrung in den Bereichen *Service Orientierte Architekturen* oder *Verteilte Systeme* mitbringen. Die tatsächlichen Kosten hinsichtlich einer Migration eines Bestandssystems in die Cloud werden demzufolge noch deutlich höher ausfallen, als es eine rein technische Analyse vermuten lässt.

Ob eine Migration Kostenvorteile bringt muss also umfassend geprüft und auf die lange Sicht bewertet werden. Die konkrete Prüfung darf demzufolge keine rein technische Bewertung sein.

5.3 AUSBLICK

Vor allem im Hinblick auf die Homogenisierung verschiedener Cloud-Services dürfte in Zukunft noch einiges zu erwarten sein. Bereits erwähnte Bibliotheken wie *Apache libcloud* sind ein erster wichtiger Schritt in die Richtung einer wirklichen offenen Public-Cloud Umgebung. Hierdurch können Cloud-Nutzer auf Basis einer einheitlichen API mit den verschiedenen Clouds interagieren, was auch den Lebenszyklus des erstellten Codes deutlich erhöht. Bedenken hinsichtlich eines möglichen Vendor-Lock-Ins werden hierdurch immer weiter aufgelöst.

Im Rahmen dieser Arbeit hat sich eine potentielle Geschäftsidee abgezeichnet. Das Ziel ist die Entwicklung eines mulimandantenfähigen *Financial Market Data Cloud Service*, welcher die gesamten Vendorcharakteristiken abstrahiert und kundenspezifsiche Datenexports und -schnittstellen anbietet. Große Probleme sind hier jedoch bezüglich der Datenlizensierung zu erwarten, da selbst das reine Weiterleiten von Marktdaten teure Lizenzen erfordert. Die Details müssten in einem Businessplan zusammengefasst und anschließend bewertet werden.

Im Rahmen der Cloudnutzung gibt es noch viele interessante Fragestellungen. Eine kleine Auswahl liste ich zum Schluss der Arbeit auf, damit interessierte Leser weitere Ansatzpunkte für eigene Studien haben.

- Wie verhält sich das Coarse- bzw. Fine-Grained Modell, wenn ein hoher Grad an Nebenläufigkeit umgesetzt wird? Welche Transaktionssteuerungen bzw. Lock-Mechanismen müssen hier implementiert werden? Reicht diese Lösung überhaupt noch aus, oder muss ggf. doch eine Datenbank eingesetzt werden?
- Wie verhält sich die AWS-Cloud bei Last und mit welchen Zugriffszeiten ist von externen Systemen aus zu rechnen? Hier könnten Benchmarks aufgesetzt werden.
- Wie gut funktionieren die Auto-Scaling-Features im Hinblick auf die Datenverarbeitung und –konstistenz?

- Kann das System ggf. nur mit Micro-Instanzen realisiert werden? Dies könnte deutliche Kostenvorteile bringen.
- Können externe Cloud-Dienste wie bspw. *Dropbox* [118]sinnvoll genutzt werden um bestimmte Daten zu synchronisieren?
- Wie lassen sich durch die Nutzung von bspw. *Puppet* [119]die operativen Kosten reduzieren?
- Wie verhält sich das System, wenn die einzelnen Teile in den Clouds unterschiedlicher Anbieter gehostet werden?
- Wie können verschiedene Availability-Zones genutzt werden um das Gesamtkonzept hinsichtlich HA-Kriterien zu gestalten?
- Wie kann ein Desaster Recovery Konzept auf einer Multi-Vendor-Basis konzipiert werden?
- Wie kann eine sinnvolle Security-Group Konfiguration aussehen?
- Wie wird der Zugriff auf die Cloudlösung realisiert, wenn Kunden keinen AWS-Account besitzen und ggf. auch nicht erstellen möchten?

[118] http://www.dropbox.com/
[119] http://puppetlabs.com/

6 LITERATURVERZEICHNIS

[1] Messe-Hannover, „www.cebit.de," 24 Februar 2011. [Online]. Available: http://www.cebit.de/de/ueber-die-messe/daten-und-fakten/die-cebit-2011/cloud-computing-top-thema. [Zugriff am 03 Juni 2011].

[2] Vaquero et al., A Break in the Cloud: Towards a Cloud Definition, in: ACM SIGCOMM Computer Communication Review, Bd. Volume 39 Number 1, 2009.

[3] K. Stanoevska-Slabeva et al. (eds.), Grid and Cloud Computing: A Business Perspective on Technology and Application, Springer-Verlag Berlin Heidelberg, 2010.

[4] C. Christmann, J. Falkner, D. Kopperger und A. Weisbecker, „Schein oder Sein," *iX - Magazin für professionelle Informationstechnik,* Nr. 5, pp. 38-43, 2011.

[5] R. Linton, Amazon Web Services: Migrating your .NET Enterprise Application, Packt Publishing, 2011.

[6] L. G. Nick Antonopoulos, Hrsg., Cloud Computing: Principles, Systems and Applications (Computer Communications and Networks), st Hrsg., Springer, Berlin, 2011.

[7] J. B. A. G. Rajkumar Buyya, Hrsg., Cloud Computing: Principles and Paradigms, John Wiley & Sons, Inc., 2011.

[8] M. Vehlow und C. Golowsky, *Cloud Computing,* PricewaterhouseCoopers AG, 2010.

[9] I. Foster, Z. Yong, I. Raicu, L. Raicu und S. Year, „Cloud Computing and Grid

Computing 360-Degree Compared,“ in *Grid Computing Environments Workshop, 2008. GCE '08, 12-16*, 2008.

[10] P. Changanti und R. Helms, Amazon SimpleDB Developer Guide, Packt Publishing, 2010, p. 252.

[11] W. Schäfer, Softwareentwicklung, Addison-Wessley in Kooperation mit Pearson Studium, 2010.

[12] P. Wilmott, Introduces Quantitative Finance, 2.Auflage Hrsg., John Wiley & Sons Ltd., 2007.

[13] J. van Vliet und F. Paganelli, Programming Amazon EC2, O'Reilly Media Inc., 2011.

[14] E. Freeman und E. Freeman, Head First - Design Patterns, O'Reilly Media Inc., 2004.

[15] AWS::Security1, „Amazon Web Services: Overview of Security Processes,“ [Online]. Available: http://aws.amazon.com/articles/1697?_encoding=UTF8&jiveRedirect=1. [Zugriff am 21 06 2011].

[16] AWS::SimpleDBPricing, „Amazon SimpleDB Pricing,“ AWS, 2011. [Online]. Available: http://aws.amazon.com/de/simpledb/pricing/. [Zugriff am 17 06 2011].

[17] AWS::S3, „Amazon Simple Storage Service (Amazon S3),“ [Online]. Available: http://aws.amazon.com/de/s3/. [Zugriff am 23 06 2011].

[18] AWS::SNS, „Amazon Simple Notification Service (Amazon SNS),“ [Online]. Available: http://aws.amazon.com/de/sns/. [Zugriff am 23 06 2011].

[19] G. Schmutz, D. Liebhart und P. Welkenbach, Service-Oriented Architecture: An Integration Blueprint, Packt Publishing Ltd., 2010.

[20] N. Weiher, „http://www.wr.informatik.uni-hamburg.de," [Online]. Available: http://www.wr.informatik.uni-hamburg.de/_media/teaching/sommersemester_2009/clcp_ss2009_nils_weiher_simpledb.pdf. [Zugriff am 4 8 2011].

[21] J. Murty, „JetS3t," [Online]. Available: http://jets3t.s3.amazonaws.com/index.html. [Zugriff am 01 09 2011].