Entwicklung eines Dashboards für eine Industrial DevOps Monitoring Plattform

Bachelorarbeit

David Benedikt Wetzel

25. September 2019

Christian-Albrechts-Universität zu Kiel
Institut für Informatik
Arbeitsgruppe Software Engineering

Betreut durch: Prof. Dr. Wilhelm Hasselbring
Sören Henning, M.Sc.

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel, 25. September 2019

Zusammenfassung

Die voranschreitende Digitalisierung in der produzierenden Industrie und der Trend hin zur *Industrie 4.0* potenziert Nachfrage und Anforderungen an das Monitoring der Produktionsprozesse. In Verbindung mit neuen Ansätzen und Methoden der Softwareentwicklung entstehen neue Forschungsfelder in der Wissenschaft und Handlungsmöglichkeiten aus Perspektive industrieller Stakeholder.

Im Kontext zunehmender Automatisierung industrieller Fertigungsprozesse bilden Dashboardlösungen zur Gewährleistung effektiver manueller Kontrolle eine Schlüsseltechnologie. In dieser Arbeit stellen wir eine Dashboardlösung vor, die Monitoringdaten der Industrial DevOps Plattform Titan vereinheitlicht visualisiert. Titan ist eine verteilte auf microservicebasierende Plattform, die Monitoringdaten aus verschiedenen Bereichen industrieller Produktion erhebt, verarbeitet und darstellt. Titan wird für industrielle Integrations- und Automatisierungsprozesse eingesetzt. Für die Entwicklung einer einheitlichen Dashboardlösung nutzen wir die Analyse- und Monitoringplattform Grafana. Um Grafana als Visualisierungssoftware einsetzen zu können, integrieren wir einen REST Adapter, welcher es ermöglicht, dass Grafana auf verschiedene Microservices der verteilten Titan Plattform zugreift. Zusätzlich integrieren wir das Monitoring- und Datenbanksystem Prometheus in die Titan Plattform. Dazu speichern wir unterschiedliche Metriken über verschiedene Kommunikationswege in Prometheus und greifen über Grafana auf gespeicherte Zeitreihen zu.

Zum jetzigen Zeitpunkt liegt der Fokus des Forschungsprojekts Titan auf dem Monitoring des Energieverbrauchs in Watt. Zusätzlich zum Energieverbrauch integrieren wir in einer Fallstudie Ereignisprotokolldateien einer industriellen Produktionsstätte. Die Anbindung der Produktionsstätte wird nach dem Prinzip des Flow-based Programming modelliert und mit der Titan Flow Engine implementiert.

Die verschiedenen Monitoringdaten stellen wir in mehreren Grafana Dashboards mittels diverser Darstellungsformen dar.

Neben der Anbindung Grafanas an eine verteilte Monitoring Plattform zeigen wir auf, inwiefern von einer Plattform wie Grafana, im Gegensatz zu einer Eigenentwicklung mit Vue.js, profitiert werden kann. In einer Gegenüberstellung des Grafana- und des derzeit eingesetzten Dashboards evaluieren wir unsere Arbeit und stellen anwendungsbezogene Vorteile der jeweiligen Ansätze dar. Grafana bietet insbesondere den Vorteil, flexibler Anpassungsmöglichkeiten, eine Entwicklung mit Vue.js ist hingegen stark individualisierbar und spezifizierbar.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1. Motivation

In Zeiten der voranschreitenden vierten industriellen Revolution hin zur Industrie 4.0 [Lasi u. a. 2014] gewinnen digitale Systeme, Softwarekomponenten und Computer-Infrastruktur signifikant an Bedeutung. Insbesondere stehen kleine und mittelständische Unternehmen (KMU) vor der Herausforderung, neue Computersysteme zu entwickeln und in den Produktionsbetrieb zu integrieren [Ernst & Young 2018]. Neben der Integration neuer Systeme ist es elementar Kenntnisse über Auslastungs- und Produktionsdetails bereits existierender Systeme zu gewinnen.

Ein Aspekt dabei ist der Stromverbrauch der eingesetzten Produktionssysteme. Um den Stromverbrauch zu analysieren und auf Grundlage dieser Analyse die Produktion anzupassen, ist es wichtig, den Stromverbrauch einzelner Maschinen messen und überwachen zu können. Durch regelmäßige Messungen ist ein Lastmanagement [Gellings 1985] möglich, dieses zielt darauf ab, den Gesamtverbrauch eines Unternehmens kontrollieren zu können. So kann es möglich sein, Lastspitzen zu vermeiden und den Stromverbrauch möglichst konstant zu halten [Ashok und Banerjee 2000]. Motiviert wird diese Bemühung dadurch, dass die durch den Netzbetreiber erhobenen Stromkosten durch hohe Lastspitzen steigen [Wagenblass 2018; Albadi und El-Saadany 2008].

Dashboads bieten ein Anwendungsbeispiel, welches Komplexe Informationslandschaften industrieller Produktionsstätte intuitiv zugänglich macht. Durch sie ist es möglich Wissen zu generieren [Eckerson 2010]. Zugleich können sie das gewonnene Wissen durch Visualisierung der schwer verständlichen Datensätze dergestalt nutzen, das Erkennen von Trends und Zusammenhängen zu vereinfachen [Rohrer 2000].

An diesem Punkt innovativer Umsetzung und Gestaltung der Industrie 4.0 setzt das Industrial DevOps Projekt *Titan* [Titan Projekt 2019; Hasselbring u. a. 2019a] an. Titan ist ein Forschungsprojekt, das die Industrial DevOps-Idee entwickelt und mit dem Ziel erforscht, KMU bei einem Digitalisierungsprozess zu unterstützen und dabei Kosten und Risiken zu minimieren. Im Fokus des Projekts steht die Bereitstellung einer branchenunabhängigen Software-Plattform für iterative Produktintegration, Automatisierung und Monitoring. Für das Energie-Monitoring werden Daten von stromverbrauchmessenden Sensoren industrieller Maschinen in unterschiedlicher Form empfangen, gespeichert, verarbeitet und visualisiert. Neben Erhebungen über die Stromausauslastung wird die Integration weiterer

Messwerte fokussiert, sodass beispielsweise Messwerte eines Fertigungsprozesses dem Stromverbrauch gegenübergestellt werden können.

Aus technischer Perspektive (siehe Kapitel 2) basiert die Titan Plattform auf dem Microservice-Architekturmuster, durch welches ein agiler Entwicklungsprozess mit geringer Kopplung zwischen den einzelnen Komponenten ermöglicht wird. Ein anderer Teil der Titan Plattform ermöglicht die Entwicklung und Integration kleinerer Komponenten, indem Flow-based Programming unterstützt wird.

Die bestehenden Visualisierungsmöglichkeiten zu erweitern ist Bestand dieser Arbeit. Ziel dabei ist es, einen Ansatz zu entwickeln, der es ermöglicht, Daten einer verteilten, auf Microservices basierenden Monitoringplattform durch ein gemeinsames Dashboard vereinheitlicht zu visualisieren. Dabei sollen die einzelnen Microservices möglichst unabhängige Schnittstellen und Datenformate anbieten können und dem Anwender soll zeitgleich eine homogene Konfigurationsoberfläche geboten werden. Exemplarisch nutzen wir dafür die Analyse und Monitoring Plattform Grafana (siehe Abschnitt 2.5) als Frontend, welche wir in die bestehende Titan Plattform integrieren. Grafana stellt verschiedene Möglichkeiten zur Verfügung, die das Administrieren von Dashboards elementar simplifizieren. So bietet Grafana eine Plattform zur Visualisierung, die auf diverse Datenbanksysteme zugreifen und Daten auf unterschiedlichste Art und Weise per Drag and Drop visualisieren kann. Zudem ist es möglich, Grafana stark zu individualisieren und anwendungsspezifische Grafana Plug-ins zu entwickeln und einzubinden.

Die einzelnen Microservices des Titan Control Centers stellen unterschiedliche Funktionalitäten zur Verfügung und nutzen dafür unterschiedliche Technologien. Der History Microservice, der die gemessenen Daten persistiert, greift auf Apache Cassandra [Lakshman und Malik 2010] als Datenbanktechnologie zurück. Zu dem jetzigen Zeitpunkt ist eine direkte Kommunikation zwischen Cassandra und Grafana nicht möglich, außerdem soll die funktionale Kapselung der Microservice-Architektur beibehalten werden. Wir stellen folglich einen Ansatz vor, der es ermöglicht, die durch Titan gemessenen und übertragenen Werte ohne direkten Zugriff auf die Cassandra-Datenbank in Grafana zu importieren und zu visualisieren.

Kumulativ zu den bereits vorhandenen Messwerten der Titan Plattform ist – es wie oben beschrieben – wichtig, Informationen aus industriellem Produktionsbetrieb erheben, verarbeiten und visualisieren zu können. Anhand von Ereignisprotokolldateien des Produktionsprozesses eines industriellen Druckzentrums zeigen wir einen Ansatz auf, der die Visualisierung ermöglicht.

1.2. Ziele

In dieser Arbeit beabsichtigen wir einen Ansatz aufzuzeigen, der sowohl einheitliche Visualisierung ermöglicht als auch die Vorteile eines Dashboard Frameworks aufzeigt. Vor diesem Hintergrund soll exemplarisch die Analyse und Monitoring Plattform Grafana (siehe Abschnitt 2.5) als weiteres Visualisierungsfrontend in die bestehende Titan Plattform

(siehe Abschnitt 2.3) integriert werden. Zu diesem Zweck soll eine Verbindung geschaffen werden, die benötigte Informationen und Daten aus den entsprechenden Komponenten Titans in Grafana übertragt. Diese Verbindung soll durch drei unterschiedliche Kommunikationswege hergestellt werden. Durch diese drei Kommunikationswege wird ermöglicht, dass auf die bestehende Persistenz der Sensordaten von Grafana direkt zugegriffen werden kann. Dazu wird auf die REST[1]-API [Fielding und Taylor 2002] des History Services zugegriffen. Ferner soll eine Alternative zur aktuellen Datenbank aufgesetzt und integriert werden. Diese Datenbank ermöglicht einen direkten Zugriff auf die Messwerte durch Grafana, ohne den modularen Ansatz der Microservice-Architektur zu verletzen. Darüber hinaus soll die Titan Flow Engine (siehe Abschnitt 2.3.1) in die Lage versetzt werden Daten direkt zu persistieren. Als Datenbank wird beispielhaft die Zeitreihendatenbank Prometheus (siehe Abschnitt 2.4) eingesetzt, welche als Zwischenglied zwischen Grafana und den Komponenten der Titan Plattform fungiert.

Ziel 1: Verbindung durch REST

Die Microservices des Titan Control Centers (siehe Abschnitt 2.3.2) stellen REST APIs zur Verfügung, durch welche auf die zugrunde liegenden Datenbanken zugegriffen werden kann. Die REST APIs der Microservices History und Configuration sollen genutzt werden, um aus dem Monitoring gewonnene Daten in Grafana zu importieren. Die Kommunikation soll an dieser Stelle über das Netzwerkprotokoll HTTP[2] im Sinne des REST Programmierparadigmas erfolgen.

Ziel 2: Verbindung mithilfe von Kafka und Prometheus

Neben dem Zugriff auf die bestehende Datenbank soll eine weitere Möglichkeit geschaffen werden, Messwerte in Grafana zu visualisieren. Dafür sollen die Daten des Monitorings zusätzlich in einer weiteren Zeitreihendatenbank (Prometheus) gespeichert werden. Da Grafana in der Lage ist direkt mit Prometheus als Datenquelle zu kommunizieren, stellt die Integration von Prometheus als Datenbank die entscheidende Kommunikationsmöglichkeit dar. Um die zu speichernden Daten zu erheben, wird auf das intern verwendete Kommunikationssystem Apache Kafka (siehe Abschnitt 2.6) zugegriffen. Aus Kafka sollen die übertragenen Messwerte des Energieverbrauchs gelesen werden, die Messwerte liegen in Form von Active- und AggregatedActivePowerRecords vor.

[1] Representational State Transfer
[2] Hypertext Transfer Protocol

Ziel 3: Verbindung der Flow Engine mit Prometheus

Für die Titan Flow Engine (siehe Abschnitt 2.3.1) soll prototypisch die Möglichkeit geschaffen werden weitere Datenformate einzulesen, zu konvertieren und in der Zeitreihendatenbank Prometheus abzuspeichern. Das Ziel dahinter ist, dass Daten, die über die Flow Engine erhoben wurden, in Grafana dargestellt werden können.

Ziel 4: Visualisierung von Messdaten durch die Flow Engine

Die Integration der Titan Flow Engine soll dazu genutzt werden Messwerte des *Druckzentrums der Kieler Nachrichten*[3] zunächst zu empfangen, zu konvertieren und in Prometheus abzuspeichern, anschließend sollen diese in Grafana visualisiert werden. Bei den Daten handelt es sich um echte Messdaten des Energieverbrauchs von Druckmaschinen. Der Energieverbrauch soll mit ebenfalls vorliegenden Informationen über die Produktion korreliert werden. Der elektrische Energieverbrauch wird in Form der Wirkleistung in Watt gemessen. Die Produktionsdaten beinhalten Informationen über die Geschwindigkeit des Drucks in Form von gedruckten Zeitungen pro Zeiteinheit und der Rotationsgeschwindigkeit der Druckmaschine. Letztendlich ist die Korrelation zwischen Stromverbrauch und Rotationsgeschwindigkeit einer Druckmaschine in visualisierter Form darzustellen.

1.3. Aufbau

Im Anschluss an dieses Kapitel erläutern wir in Kapitel 2 die wichtigsten Grundlagen für die vorliegende Arbeit. In Kapitel 3 stellen wir die Architektur unseres Ansatzes vor und zeigen in Kapitel 4, wie dieser Ansatz prototypisch implementiert und mittels Grafana visualisiert werden kann. Anschließend studieren wir in Kapitel 5 ein Fallbeispiel, indem wir unter Rückgriff auf die vorherigen Kapitel zeigen, wie unser Verfahren praktisch eingesetzt werden kann. In Kapitel 6 dokumentieren wir die Evaluation unserer Arbeit, ziehen in Kapitel 8 ein Fazit und erläutern abschließend Aspekte, die im Anschluss an unsere Arbeit weiter exploriert werden können.

[3] `https://kn-druckzentrum.de/`

Kapitel 2

Grundlagen und Technologien

Im Folgenden erläutern wir zentralen Technologien und Frameworks dieser Arbeit. Besonderes Augenmerk liegt dabei auf der Titan Plattform und den exemplarisch eingesetzten Komponenten Grafana und Prometheus.

2.1. Microservice-Architektur

Das Microservice-Architekturmuster [Hasselbring 2016; Newman 2015; Hasselbring und Steinacker 2017] ist ein Entwurfsmuster, welches die Grundstruktur einer Software festlegt. Die Software besteht dabei aus der Kombination verschiedener Unterkomponenten, den sog. Microservices. Ein Microservice ist für die Umsetzung einer spezifischen Aufgabe zuständig und kommuniziert über festgelegte sprachunabhängige Schnittstellen mit anderen Microservices. Dieser Ansatz sorgt dafür, dass eine starke Modularisierung der Software stattfindet. Ein Microservice stellt dabei einen technisch eigenständigen und unabhängigen Service dar. Ein solcher Service stellt jeweils eine einzelne Funktionalität zur Verfügung. Das Kommunizieren und Agieren zwischen den Microservices findet zustandslos statt, das heißt, es wird kein interner Zustand geteilt. Zustandslose Kommunikation kann bspw. mithilfe von REST erfolgen. Dieses Vorgehen gewährleistet, dass einzelne Microservices unabhängig voneinander entwickelt, ausgeführt und ausgetauscht werden können, wodurch eine hohe Skalierbarkeit des Systems unterstützt wird. Abbildung 2.3 zeigt die wichtigsten Microservices des Titan Control Centers.

2.2. Industrial DevOps

Industrial DevOps [Hasselbring u. a. 2019a] ist eine Sammlung von DevOps[1] [Jabbari u. a. 2016] Methoden, Techniken und Prinzipien im Kontext industriellen Handelns. DevOps stellt Methoden zur effizienteren Zusammenarbeit zwischen Entwicklern und Bedienenden zur Verfügung. Industrial DevOps ist ein Ansatz der DevOps-Methoden mit industriellen Abläufen verbindet und genutzt wird, um Systeme im Kontext von industriellen Produktionsumgebungen zu integrieren und zu monitoren. Im Vordergrund stehen dabei Prinzipien wie ein kontinuierlicher Anpassungs- und Verbesserungsprozess, ein zyklischer

[1] Abkürzung für Development (Entwicklung) and Operation (Betrieb)

Vorgang mit Phasen des Monitorings und der Analyse, der Anforderungsanalyse und Entwicklung, des Testens und der Inbetriebnahme sowie der Phase des produktiven Betriebs. Sie ermöglicht eine schlanke Organisationsstruktur, die für die effiziente Integration von Softwarelösungen mit dem Fokus steht, einen lösungsorientierten Diskurs zwischen allen Akteuren und Interessenvertretern stattfinden zu lassen.

2.3. Die Industrial DevOps Plattform Titan

Die Industrial DevOps Plattform Titan [Titan Projekt 2019] ist eine Software-Plattform, die dabei hilft, Industrial DevOps (siehe Abschnitt 2.2) umzusetzen. Im Fokus steht dabei sowohl die System-Integration als auch das Monitoring von industrieller Produktionsumgebung. Die Titan Plattform unterstützt bei der Umsetzung von Industrial DevOps und besteht aus mehreren lose gekoppelten Komponenten, welche im Folgenden beschrieben werden. Neben den hier beschriebenen Komponenten existiert ein Single-Page Frontend, durch welches Messwerte angezeigt und Konfigurationen vorgenommen werden können[2].

2.3.1. Titan Flow Engine

Die *Titan Flow Engine* [Hasselbring u. a. 2019a] stellt eine grafische Modellierungssprache zur Konfiguration der Titan Software zur Verfügung. Über diese grafische Oberfläche ist es möglich, dass die Konfiguration von Facharbeitern oder Produktionsleitern vorgenommen werden kann. Für die Integration von Produktionsmaschinen, Software oder Datenbanken werden kombinierbare Softwarekomponenten kreiert, diese Softwarekomponenten heißen *Bricks*. Neben der Anbindung externer Systeme können Bricks auch Aufgaben wie die Konvertierung, Filterung oder Weiterleitung von Daten haben. Ein Brick ist also eine Komponente, welche Daten erheben, verarbeiten, nutzen oder bereitstellen kann.

[2] `http://samoa.se.informatik.uni-kiel.de:8185/`

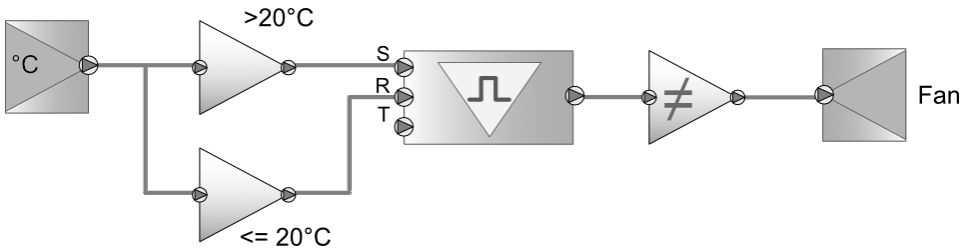

Abbildung 2.1. Visualisierung eines *Titan Flows* zur Lüftersteuerung [Hasselbring u. a. 2019b]. Links ist ein *Inlet*, dieses ist ein produzierendes Brick, ganz rechts ist ein *Outlet*, ein konsumierendes Brick. Die dreieckigen Symbole stellen filternde Bricks dar, das mittlere viereckige Symbol stellt einen *Switch-Brick* dar.

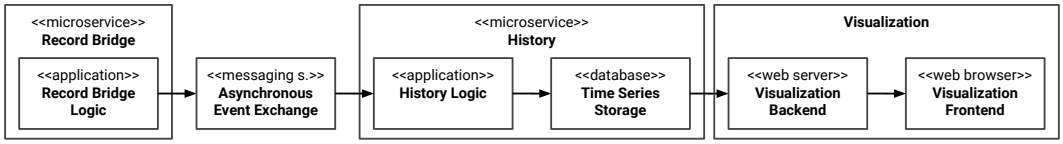

Abbildung 2.2. Architektur des Datenflusses innerhalb des Control Centers [Henning u. a. 2019]. Die Record Bridge empfängt Daten und schreibt diese in einem verarbeitbaren Format in den Kafka Datenstrom. Der History Service nimmt die Nachricht aus Kafka entgegen und speichert diese in einer Datenbank (hier Apache Cassandra). Das Frontend bezieht die Daten über den History Service und visualisiert diese.

Die Titan Flow Engine folgt damit dem Programmierparadigma des *Flow-based Programming* [Morrison 2010]. Flow-based Programming ist ein Ansatz, der eine Anwendung nicht als geschlossenen, deterministischen Prozess betrachtet, sondern als Netzwerk von (asynchronen) Prozessen (hier Bricks), die über Datenströme (hier Flows) kommunizieren. Bricks können in diesem Sinne zahlreich kombiniert werden, um neue Anwendungen zu erstellen. Dadurch entsteht die Möglichkeit, Geschäfts- und Produktionsprozesse grafisch durch Kombination von Bricks zu erstellen. Bricks besitzen *Input-* und *Output-Ports* und können dabei durch einen oder mehrere Ports verbunden werden. Die modellierten Prozessabläufe heißen *Flows*. Flows können auch zur Initiierung von Aktionen, wie dem Einschalten eines Lüfters oder dem Abspeichern von Monitoringdaten, benutzt werden. Abbildung 2.1 zeigt exemplarisch einen Flow zur Lüftersteuerung.

2.3.2. Titan Control Center

Das Titan Control Center [Henning u. a. 2019; Henning 2018] ist ein verteiltes, skalierbares Softwaresystem und stellt die benötigte Infrastruktur zum Monitoren und Analysieren bereit. Es setzt dazu auf das Microservice-Architekturmuster. Abbildung 2.3 zeigt die wichtigsten Komponenten des Titan Control Centers. Die Kommunikation der einzelnen Microservices findet sowohl synchron über HTTP/REST als auch asynchron über Apache Kafka (siehe Abschnitt 2.6) statt. Das Frontend bezieht die Messdaten über den History Service, welcher die Daten in unterschiedlicher Form vorhält und für die Speicherung der Informationen verantwortlich ist. Abbildung 2.2 visualisiert diesen Vorgang. Über den Configuration Service bezieht das Frontend Konfigurationsinformationen. In den folgenden Absätzen werden die in dieser Arbeit verwendeten Microservices kurz vorgestellt.

History Service Der History Service ist ein Microservice und ist für die Speicherung von Messwerten zuständig. Neben der Speicherung werden verschiedene Messwerte aggregiert. Zum Persistieren wird intern das Datenbanksystem Apache Cassandra eingesetzt. Über eine REST API stellt der History Service eine Zugriffsmöglichkeit auf diese Datenbank zur Verfügung. Der History Service speichert zum einen die Messwerte eines physikalischen Sensors ab, dazu wird das *ActivePowerRecord*-Format (siehe Listing 2.1) genutzt. Auf der

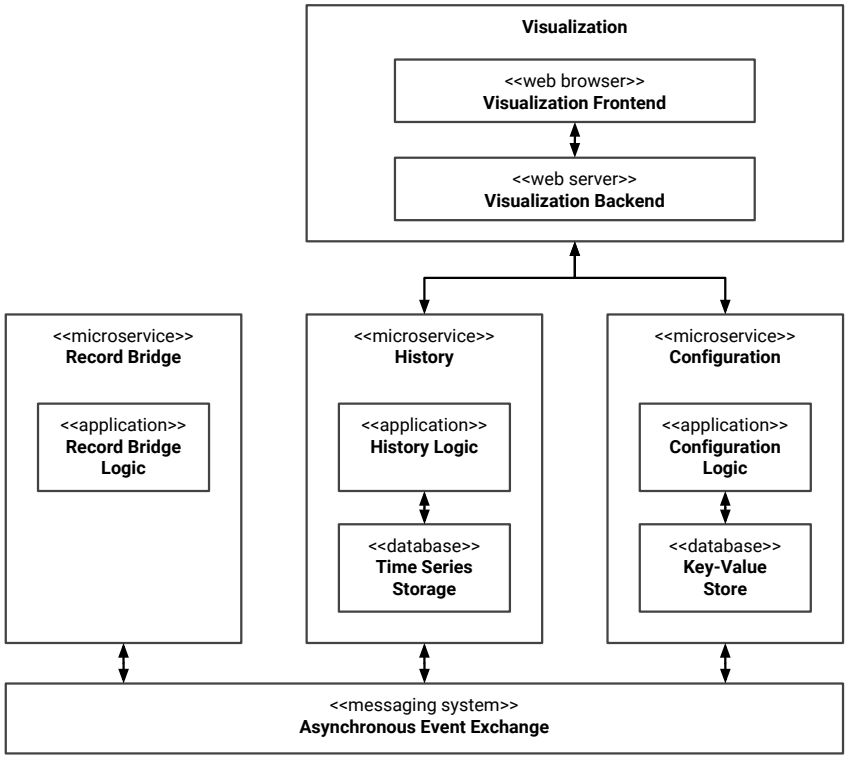

Abbildung 2.3. Abbildung der wichtigsten Microservices der aktuellen Architektur des Titan Control Centers [Henning u. a. 2019]. Das Visualisierungsfrontend greift auf den History und den Configuration Microservice zu. Die Microservices sind untereinander durch Kafka (siehe Abschnitt 2.6) verbunden.

anderen Seite existiert die Möglichkeit, logisch aggregierte Sensoren zu erstellen. Diese aggregierten Sensoren speichern berechnete Werte einer Gruppe von Sensoren, dazu wird das *AggregatedActivePowerRecord*-Format (siehe Listing 2.2) genutzt. ActivePowerRecord werden dazu aus einem Kafka Topic gelesen und AggregatedActivePowerRecord in ein anderes geschrieben, wenn diese berechnet wurden. Eine detaillierte Spezifikation ist in Henning [2018] zu finden.

Configuration Service Der Configuration Service organisiert systemweite Konfigurationen und Einstellungen. Insbesondere wird hier die Registrierung neuer Sensoren vorgenommen und Auskunft über bereits registrierte erteilt. Dazu wird ein *SensorRegistry*-Objekt versendet, welches die benötigten Informationen beinhaltet.

```
1   public class ActivePowerRecord extends AbstractMonitoringRecord {
2     ...
3     /** property declarations. */
4     private final String identifier;
5     private final long timestamp;
6     private final double valueInW;
7     ...
8     }
```

Listing 2.1. Felder eines ActivePowerRecord-Objektes

```
1   public class AggregatedActivePowerRecord extends AbstractMonitoringRecord {
2     ...
3     /** property declarations. */
4     private final String identifier;
5     private final long timestamp;
6     private final double minInW;
7     private final double maxInW;
8     private final long count;
9     private final double sumInW;
10    private final double averageInW;
11    ...
12    }
```

Listing 2.2. Felder eines AggregatedActivePowerRecord-Objektes

Record Bridge Das Kernaufgabengebiet der Record Bridge ist es, Konvertierungen von Sensorsignalen vorzunehmen. Die Ausgabeformate von Sensoren sind nicht genormt, deswegen sind verschiedene Ausprägungen der Record Bridge dafür zuständig, unterschiedliche Eingangssignale so zu konvertieren, dass diese durch das Titan Control Center verarbeitet werden können.

2.4. Die Zeitreihendatenbank Prometheus

Prometheus ist ein Open Source Datenbank-, Monitoring- und Alerting-Toolkit (MAT) [Turnbull 2018; Prometheus 2019]. Die NoSQL Datenbankkomponente Prometheus ist eine *Zeitreihendatenbank (Time Series Database (TSDB))* die besonders auf Aufgaben des Monitorings ausgelegt ist. Das Monitoring-Tool ist stark skalierbar. Es ist möglich Speicherknoten verteilt einzusetzen, jeder einzelne Knoten ist jedoch autonom von anderen tätig.

Prometheus besteht aus mehreren Komponenten zur Abfrage und Speicherung von

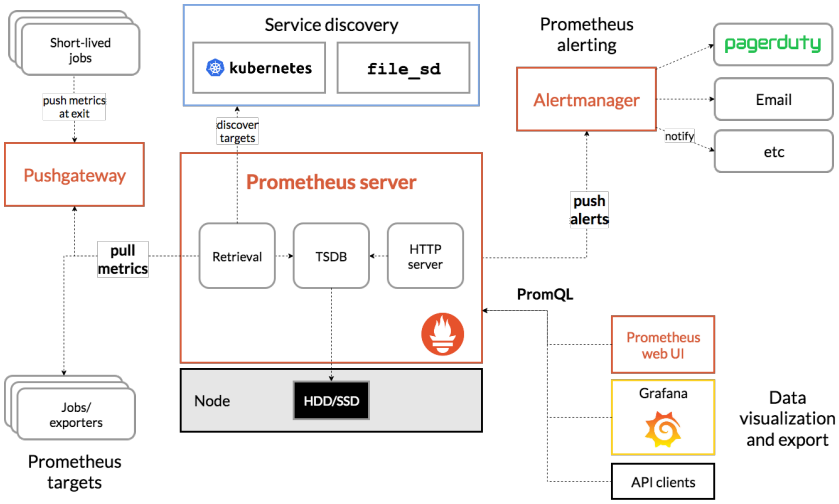

Abbildung 2.4. Überblick über die Prometheus-Architektur[3].

Metriken und Daten, hierzu verwendet Prometheus grundsätzlich das *Pull-Prinzip*, d. h. zu monitorende Ziele sind Prometheus bekannt und werden periodisch durch Prometheus abgefragt. Außerdem besitzt Prometheus einen integrierten Alert-Manager, welcher den Part des Alertings übernimmt. Zur Visualisierung besitzt Prometheus auf der einen Seite eine eigene Weboberfläche, auf der anderen Seite wird Grafana (siehe Abschnitt 2.5) häufig mit Prometheus kombiniert, um auch komplexere Visualisierungen zu ermöglichen.

Features und Komponenten: Prometheus stellt für verschiedene Programmiersprachen Client Libraries zur Verfügung. In dieser Arbeit wird auf den Java[3]- und Phyton[4]-Client zurückgegriffen. Die Libraries stellen unterschiedliche Funktionalitäten zur Verfügung, die die Entwicklung von Prometheus Exportern ermöglicht.

Prometheus Exporter

Das Pull-Prinzip des Prometheussystems basiert auf dem Grundsatz, dass einzelne HTTP/ REST APIs abgefragt werden, welche die zu speichernden Metriken zurückliefern. Als Exporter werden Softwarekomponenten bezeichnet, die eine solche Schnittstelle bereitstellen. Ein Exporter stellt dabei Metriken eines Ziels zur Verfügung. Ein Ziel ist dabei das,

[3] https://prometheus.io/assets/architecture.png
[3] https://github.com/prometheus/client_java
[4] https://github.com/prometheus/client_python

was gemonitort werden soll, also beispielsweise ein Computer. Metriken könnten in diesem Fall Messdaten der CPU-Auslastung oder der Nutzung des Speichers sein.

Prometheus Query Language

Die *Prometheus Query Language (PromQL)*[5] ist die Datenbanksprache des Prometheus Toolkits. Über PromQL können Daten abgefragt und mithilfe bestimmter Operationen aggregiert werden. Über die Weboberfläche des Prometheusservers können Abfrageergebnisse in Form eines Graphen oder einer Tabelle dargestellt werden. PromQL wird außerdem genutzt, um Datenbankabfragen an die HTTP API zu stellen.

2.5. Das Visualisierungsfrontend Grafana

Grafana [Grafana Labs 2019] ist eine offene Plattform für Analyse und Monitoring. Grafana stellt die Komponente des Dashboards, welches Visualisierungen enthält, bereit. Zur Visualisierung von Messdaten existieren verschiedene Darstellungsformen mit jeweils unterschiedlichen Einstellungsmöglichkeiten. Um auf Messwerte zuzugreifen, benutzt Grafana *Datenquellen*. Die Datenquellen werden der Plattform durch Aktivierung eines Plug-ins hinzugefügt. Anschließend können Graphen und andere Visualisierungsformen auf diese Datenquelle zugreifen und mithilfe von *Queries* Anfragen an diese stellen, um Daten zu erhalten. Grafana Labs stellt eine Demonstrationsoberfläche[6] online zur Verfügung, die exemplarisch verschiedene Darstellungsformen zeigt und die Erkundung Grafanas ermöglicht.

2.5.1. Grafana Plug-ins

Wesentliche Bestandteile von Grafana sind drei verschiedene Typen von Plug-ins, die für unterschiedliche Zwecke in Grafana genutzt werden können. Zum einen wird das *Panel Plug-in* für Visualisierungen eingesetzt. Ein Panel Plug-in greift auf ein *Datenquellen Plug-in* zu, ein Datenquellen Plug-in interagiert in der Regel mit einem Datenbanksystem. Die standardisierte Schnittstelle zwischen Panel- und Datenquellen Plug-ins ermöglichen es, dass ein Panel Plug-in für verschiedene Datenquellen genutzt werden kann. Ein weiterer Typ ist das *App Plug-in*, dieses ermöglicht die Kombination von Panel- und Datenquellen Plug-in derart, dass ein (App-)Panel beispielsweise zum Steuern weiterer Komponenten über HTTP Requests genutzt werden kann.

Für diese Arbeit werden folgende Plug-ins in Grafana hinzugefügt:

[5] https://prometheus.io/docs/prometheus/latest/querying/basics/

[6] https://play.grafana.org/

SimpleJson Plug-in Das SimpleJson Plug-in[7] ist ein Open Source Plug-in, welches die Integration von REST APIs in Grafana ermöglicht. Die akzeptierte REST API hat festgelegte Schnittstellen. Die abgerufenen Daten werden als JSON (JavaScript Object Notation)[8]-String übertragen und müssen in einem bestimmten Format gesendet werden.

Prometheus Plug-in Das Prometheus Plug-in[9] wird offiziell von Grafana Labs angeboten und stellt den Zugriff auf eine Prometheus Datenbank her. Anschließend können Queries direkt abgegeben und die Daten abgefragt werden.

2.6. Das verteilte Nachrichtensystem Apache Kafka

Apache Kafka [Kreps 2011; Apache Software Foundation 2017] ist ein verteiltes Nachrichtensystem zum Senden und Empfangen von Nachrichten, ein besonderer Fokus liegt dabei auf hohem Durchsatz und geringer Latenz. Im Mittelpunkt der zugrundeliegenden Architektur sind *Topics, Producer, Consumer, Broker* und *Cluster*. Ein Datenfluss von Nachrichten einer bestimmten Kategorie ist als *Topic* definiert. Ein *Producer* schreibt Nachrichten (*Records*) in ein Topic, *Consumer* lesen Nachrichten aus einem Topic. Dabei ist sowohl eine 1:1 Beziehung zwischen Producer und Consumer als auch eine 1:n Beziehung möglich. Die verteilte Infrastruktur von Kafka basiert auf *Clustern*. Ein Cluster speichert Nachrichtenströme in Topics. Ein *Record* besteht aus einem Schlüssel (*key*), einem Wert (*Value*) und einem Zeitstempel (*Timestamp*). Dabei stellt das Nachrichtensystem Kafka sicher, dass Nachrichten in der gleichen Reihenfolge gespeichert und von den Consumern gelesen, wie durch die Producer gesendet werden. Außerdem stellt Kafka eine hohe Toleranz gegenüber Serverausfällen zur Verfügung, da diese oftmals durch das verteilte System, ohne Datenverlust kompensiert werden können.

[7] https://grafana.com/plugins/grafana-simple-json-datasource
[8] http://json.org/
[9] https://grafana.com/plugins/prometheus

Kapitel 3

Architektur

Die Vereinheitlichung der Visualisierung basiert auf dem Ansatz der Entkopplung einzelner Komponenten der Titan Plattform von der Visualisierungsschicht. In unserem Ansatz nutzen wir das Dashboard Framework Grafana um Daten zu visualisieren. Die einzelnen Komponenten der Titan Plattform stellen ihre Daten über unterschiedliche Schnittstellen bereit, eine Softwareschicht zwischen einzelnen Komponenten Titans und Grafana ermöglicht die Kommunikation. Die Grafana Dashboard-Oberfläche ermöglicht das Erstellen und Erweitern von Dashboards. Der Zugriff auf Datenquellen findet abstrahiert von einzelnen Komponenten der Titan Plattform statt. Dadurch wird ermöglicht, dass auch BenutzerInnen ohne weitreichende Kenntnisse der Informatik oder der internen Struktur der Titan Plattform Dashboards administrieren können.

Unser Verfahren stellt auf der einen Seite die Verbindung über REST zu den Titan Microservices Configuration und History her. Auf der anderen Seite integrieren wir Nachrichten des Nachrichtensystems Kafka, indem wir übertragene Nachrichten abspeichern. Zu diesem Zweck nutzen wir exemplarisch das Prometheus Toolkit. Darüber hinaus zeigen wir einen Weg auf, Monitoringdaten über die Titan Flow Engine in Prometheus zu übertragen und anschließend in Grafana zu visualisieren.

3.1. Architektur Überblick

Aufbauend auf die bestehenden Komponenten der Titan Plattform haben wir zur erweiterten Visualisierung eine zusätzliche Komponente entwickelt und integriert. Diese Komponente besteht aus einem Grafana Frontend und einem Grafana Backend. Abbildung 3.1 zeigt den schematischen Aufbau der Visualisierungskomponente auf Ebene eines UML-Komponentendiagramms [Rumbaugh u. a. 2004]. Abbildung 5.5 illustriert schematisch die Kommunikation zwischen verschiedenen Komponenten.

Unser Ansatz der Visualisierungskomponente besteht aus mehreren lose gekoppelten Komponenten. Die lose Kopplung erhöht die Wartbarkeit der Komponenten, da diese getrennt entwickelt werden können. Abhängigkeiten bestehen lediglich in den definierten Schnittstellen einzelner Komponenten, welche im Umfang minimal gehalten wurden.

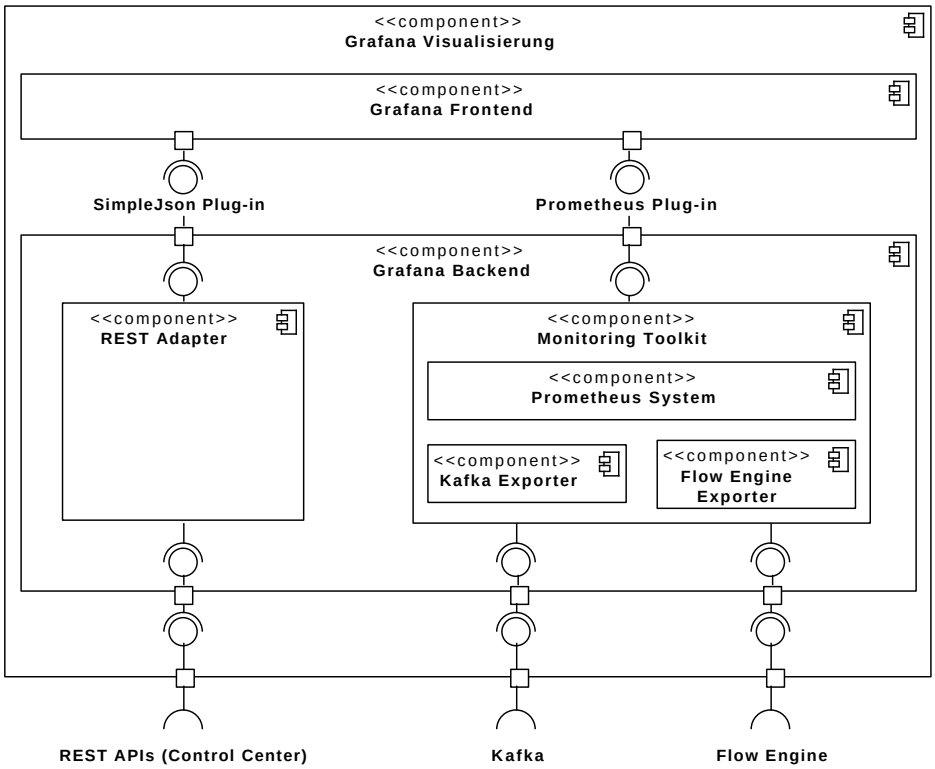

Abbildung 3.1. Schematische Darstellung unseres Architekturansatzes. Das Frontend greift auf Schnittstellen der Backend Komponente zu, welche Schnittstellen der Titan Plattform abfragt, die Metriken transformiert, (zwischen-) speichert und bereitstellt.

3.1.1. Das Frontend unseres Ansatzes

Die Komponente des Frontends ist eine Grafana Single-Page Anwendung und dient der Visualisierung von Monitoringdaten. Um die Abfrage auf verschiedene Datenquellen zu ermöglichen, können in Grafana verschiedene Plug-ins hinzugefügt werden, die auf eine spezifische Datenquelle zugreifen können. In unserem Ansatz haben wir ein Plug-in aktiviert, welches den Datentransfer über das HTTP Protokoll nach dem REST Programmierparadigma (im Folgenden HTTP/REST) unterstützt. Ein weiteres Plug-in ermöglicht die Kommunikation mit einem Prometheusserver. Mithilfe dieser Plug-ins werden Datenabfragen gestellt und Ergebnisse empfangen und interpretiert. Abgefragte Daten haben wir in Form verschiedener Darstellungsarten visualisiert. Der Zugriff auf Datenquellen erfolgt über das Netzwerkprotokoll HTTP. Die HTTP Nachrichten werden durch die aktivierten Plug-ins formuliert und entsprechen den Formaten, die die abgefragte Datenquelle erwartet.

3.1.2. Das Backend unseres Ansatzes

Die Komponente des *Grafana Backends* stellt die Metrikdaten für das Grafana Frontend bereit. Die Metrikdaten des Titan Control Centers liegen entweder in aggregierter oder in nicht aggregierter Form vor und sind Objekte des Typs *IMonitoringRecord*. Ein weiterer Record Typ ist der *vip-log-record*[1], diese Records erhalten wir über die Titan Flow Engine, indem wir einen Flow betrachten, welcher Metriken mit Informationen über laufende Produktionen verarbeitet (siehe Abschnitt 1.2, Ziel 4).

Für den Datenzugriff des Frontends auf Metrikdaten stellt das Grafana Backend zwei Schnittstellen zur Verfügung, die es ermöglichen, dass das Frontend die Daten in benötigten Formaten und Protokollen abfragen kann. Es steht damit je eine Schnittstelle pro aktiviertem Datenquellen Plug-in in Grafana zur Verfügung.

Das Grafana Backend fungiert als Schnittstelle zwischen verschiedenen Modulen der Titan Plattform, die Daten erheben und verarbeiten und dem Grafana Frontend. Das Grafana Backend besteht aus zwei Unterkomponenten. Die erste Unterkomponente ist der REST Adapter, welcher den Kommunikationskanal über HTTP/REST ermöglicht und in Java implementiert ist. Der REST Adapter fragt die vorhandenen Schnittstellen anderer Microservices ab, transformiert die Daten und stellt sie bereit, sodass das SimpleJson Plug-in diese empfangen und verarbeiten kann. Insbesondere wird auf den History Service zugegriffen, der die Daten des Monitorings zur Verfügung stellt. Ein Zugriff auf den Configuration Service erfolgt zusätzlich, um die Konfiguration des Systems abzufragen. Der Configuration Service erteilt in einer baumartigen Struktur Auskunft über registrierte Sensoren und deren Aggregationsmöglichkeiten.

Die zweite Unterkomponente des *Grafana Backends* ist die Monitoring Toolkit Komponente. In dieser haben wir ein weiteres Monitoringsystem mit Datenbankkomponenten eingerichtet, welches ähnlich zur Konfiguration der *Cassandra*-Datenbank des History Services sämtliche in Kafka übertragenen IMonitoringRecords in Form von Zeitreihen persistiert. Dazu haben wir den Webserver des Prometheussystems so konfiguriert, dass dieser die Daten der Exporter scrapt[2] und als Zeitreihendaten speichert. Abbildung 3.2 zeigt den schematischen Aufbau der Monitoring Toolkit Komponente. Weitere Bestandteile dieser Komponente sind zwei Prometheus Exporter, welche die Metrikdaten, die über das Nachrichtensystem Kafka respektive die Flow Engine zur Verfügung stehen, für den Prometheusserver bereitstellen.

Die Komponente des Kafka Exporters in Java implementiert und *subscribt* dafür das Nachrichtensystem Apache Kafka und konsumiert Records verschiedener Topics. Die konsumierten Daten werden daraufhin derart transformiert und zusammengesetzt, dass diese abrufbar vorliegen. Der zweite Exporter ist in Form von Python-Skripten implementiert. Die an dieser Stelle exportierten Metrikdaten sind zuvor durch die Ausführung eines *Flows*

[1] VIP ist das Druck-Management-System des Druckzentrums der Kieler Nachrichten. In unserem Fall nutzen wir eine Ereignisprotokolldatei dieses Systems, welches Informationen über Druckvorgänge abspeichert.

[2] scraping bezeichnet hier den Vorgang, in dem der Prometheusserver aktiv hinterlegte Ressourcen abfragt und vorhandene Metrikdaten abspeichert (Pull-Prinzip)

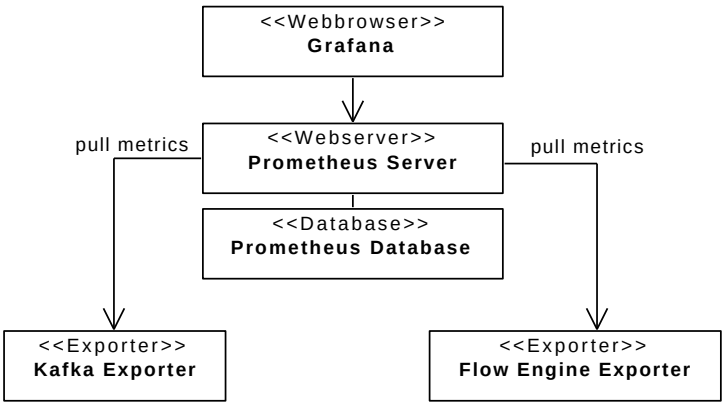

Abbildung 3.2. Die Grafik zeigt den Datenfluss verschiedener Bestandteile des Grafana Backends. Der Prometheusserver einer Prometheus Datenbankinstanz greift auf zwei Exporter zu, speichert die Metriken und bietet eine Schnittstelle zu Grafana an.

mit der Titan Flow Engine berechnet worden.

Die Exporter stellen HTTP APIs zur Abfrage von Metriken durch den Prometheusserver bereit. Der Prometheus Datenbankserver stellt wiederum eine HTTP API zur Verfügung, diese Schnittstelle ist außerhalb der *Grafana Backend* Komponente zugreifbar und stellt die Schnittstelle zu Grafana dar.

3.2. Industrielle Fallstudie: Zeitungsdruckerei

Neben den bereits vorgestellten Architekturansätzen zu Komponenten für das Frontend und Backend, betrachten wir in einer Fallstudie verschiedene Bricks, die echte Monitoringdaten einlesen, verarbeiten und über die diskutierten Wege in Prometheus abspeichern. In Kapitel 5 stellen wir die Fallstudie detailliert vor.

Für diese Fallstudie haben wir zwei weitere *Input-Bricks* implementiert, welche dazu in der Lage sind, Monitoringdaten eines Produktionsbetriebs entgegenzunehmen und die Daten, transformiert in Ujo Nachrichten[3], an ein nächstes Brick zu senden. Ein weiterer Brick bewirkt die Verarbeitung der empfangenen Daten und transformiert diese in ein weiteres intern verwendetes Ujo Format. Die transformierten Daten sind anschließend innerhalb der Flow Engine abrufbar. Die Kommunikation zwischen den Bricks erfolgt in Ujo und wird durch das *Flow Engine* Framework organisiert und ausgeführt. Sämtliche Bricks sind in der Programmiersprache Python implementiert und werden in prototypischen Flows kombiniert. Diese Flows sorgen dafür, dass Daten eingelesen, transformiert und gespeichert werden, um in Grafana visualisiert werden zu können.

[3] `https://libujo.org/`

Kapitel 4

Implementierung

In diesem Kapitel beschreiben wir die Implementierung unseres Ansatzes. Dazu haben wir basierend auf der bereits vorgestellten Architektur entsprechende Komponenten entwickelt. Die Struktur dieses Kapitels ist an die Reihenfolge der Ziele aus Kapitel 1, siehe Abschnitt 1.2, angelehnt. Zunächst stellen wir die Kommunikationskanäle, mit denen wir die Titan Plattform mit Grafana verbinden, getrennt voneinander vor. Zur Einordnung der einzelnen Komponenten verweisen wir dabei auf das vorherige Kapitel, welches in abstrahierter Form einen Überblick bietet. Im Anschluss folgen Beschreibungen und Grafiken zu unserem Grafana Dashboard.

4.1. Verbindung durch HTTP/REST

In unserem Verfahren führt ein Verbindungskanal über HTTP. Im Folgenden stellen wir eine prototypische Implementierung vor, welche primär auf der Idee basiert, einen REST Adapter in die Kommunikation zwischen Grafana und der Titan Plattform, zu integrieren. Dieser REST Adapter ermöglicht die Kommunikation zwischen Grafana und verschiedenen Schnittstellen Titans. Um die Kommunikation über HTTP/REST zu ermöglichen, ist das *SimpleJson Plug-in* in Grafana aktiviert. Durch dieses Plug-in kann eine REST Schnittstelle in Grafana als *Datenquelle* hinzugefügt werden. Auf eine aktivierte Datenquelle kann in Grafana von verschiedenen *Panels* zugegriffen werden, welche gezielt Metriken abfragen und visualisieren. Das SimpleJson Plug-in greift auf fest definierte Schnittstellen zu, diese sind anders als jene, die durch die Microservices (History, Configuration) des Titan Control Centers zur Verfügung gestellt werden. Um die Kommunikation zu ermöglichen, sorgt der REST Adapter für die Konvertierung der Übertragungsformate und bewirkt eine Umleitung der Aufrufe.

Der REST Adapter ist im Stil des Adapter-Entwurfsmusters [Gamma u. a. 1995] mithilfe des Web Frameworks *javalin*[1] implementiert und stellt eine Umsetzung von Ziel 1 (siehe Abschnitt 1.2) dar. Das Adapter-Entwurfsmuster ist ein Entwurfsmuster aus der Kategorie der Strukturmuster und adaptiert zwei zueinander inkompatible Schnittstellen, sodass eine Kommunikation ermöglicht wird. In dieser Arbeit verwenden wir den Ansatz des Objekt-Adapters, ein Ansatz in dem Aufrufe durch Delegation an den eigentlichen Empfänger übergeben und dort bearbeitet werden. Das Ergebnis wird mit angepasster

[1] https://javalin.io/

Tabelle 4.1. Angebotene Schnittstellen des REST Adapters

Schnittstelle	Verwendungszweck
GET host/	Wird genutzt, um eine neue Datenquelle in Grafana zu aktivieren.
POST host/search	Wird genutzt, um die vorhandenen Sensoren abzufragen.
POST host/query	Wird genutzt, um Metrikdaten eines Sensors abzufragen.

Schnittstelle und ggf. in transformierter Form an den Aufrufer übergeben. Die Verwendung des Adapter-Entwurfsmusters ermöglicht somit die Nutzung inkompatibler Schnittstellen ohne Anpassung des Source Codes.

4.1.1. Schnittstellen

Die API unseres REST Adapters stellt Schnittstellen zur Abfrage verschiedener Daten bereit. Tabelle 4.1 zeigt die URLs[2] der wichtigsten Schnittstellen mit den entsprechenden Anfragemethoden des HTTP Protokolls. Die angebotenen Adressen werden durch das in Grafana aktivierte Plug-in SimpleJson adressiert.

Die Schnittstelle **host/** wird für das Einrichten einer neuen Datenquelle benötigt. Grafana erwartet eine Response-Nachricht mit dem HTTP Statuscode 200 zurück, um die Datenquelle zu aktivieren. Die Schnittstellen für **host/search** und **host/query** stehen für HTTP POST und HTTP Option Anfragen zur Verfügung. Das liegt daran, dass SimpleJson Anfragen über den HTTP Typ POST stellt. Beim Aufruf der **host/query** Schnittstelle wird dabei ein JSON-Objekt übertragen, welches die Anfrageparameter kapselt. Die Adresse **host/search** wird aufgerufen, um vorhandene Metriknamen abzufragen.

4.1.2. Abfragen

Konzeptionell wird innerhalb des REST Adapters für jede eingehende Anfrage ein HTTP Request an eine Schnittstelle des Titan Control Centers gestellt. Das Ergebnis der Abfrage wird transformiert und als HTTP Response Nachricht an den Aufrufenden zurückgesendet. Wir haben in dieser Arbeit Schnittstellen zum Laden der aktuellen Konfiguration und zum Abrufen von ActivePowerRecords und AggregatedActivePowerRecords adaptiert. Abhängig von der angefragten Ressource (Konfiguration, ActivePowerRecord, AggregatedActivePowerRecord) werden unterschiedliche Prozeduren durchgeführt, deren Mechanismen wir im Folgenden beschreiben.

Konfiguration Das SimpleJson Plug-in sendet jedes Mal, wenn in Grafana eine weitere Metrik abgefragt werden soll, einen POST-Request an die URL host/search. Der REST Adapter

[2]Uniform Resource Locator

```
1    public class SimpleJsonTimeserie {
2      private final String target; // id of the sensor
3      private final Object[][] datapoints; // array of datapoints (tmp, val)
4      ...
5      }
```

Listing 4.1. SimpleJsonTimeserie-Klasse innerhalb des REST Adapters.

ruft daraufhin die REST API des Configuration Services auf, welcher die aktuelle Konfiguration in Form eines *SensorRegistry*-JSON-Objektes überträgt.

Unsere Adapterimplementierung extrahiert die IDs aller Sensoren aus einem SensorRegistry-Objekt und speichert für jede Sensor-ID die Information, ob diese einem aggregierten Sensor entspricht. Unsere Implementierung sendet anschließend ein String-Array in Form eines JSON-Objektes an Grafana, welches die IDs aller verfügbaren Sensoren enthält.

Metrikdaten SimpleJson fragt Metrikdaten über POST-Request an, übertragen wird ein JSON-Objekt, welches sämtliche Parameter der Anfrage beinhaltet. Intern verwenden wir ein *SimpleJsonQuery*-Objekt, um die Anfrage zu kapseln. Über dieses Objekt ist der Zugriff auf Anfrageparameter wie beispielsweise die Namen der abgefragten Metriken oder den Abfragezeitraum möglich.

Der von uns entwickelte REST Adapter greift zunächst auf die Sensor-ID zu und überprüft, ob der Sensor aggregiert ist. Das Ergebnis der Überprüfung bestimmt die abzufragende Schnittstelle des History Services, welcher für aggregierte Sensoren eine gesonderte Schnittstelle bereitstellt. SimpleJson fragt Daten für einen bestimmten Zeitraum (*from*, *to*) ab, an das Titan Control Center kann derzeit ausschließlich die Variable *from* als Parameter *after* übergeben werden. Der REST Adapter überträgt den Parameter *to* ebenfalls als Anfrageparameter. Die Antwortnachricht an SimpleJson (respektive Grafana) beinhaltet alle Daten, die neuer als der Abfrageparameter *from* sind.

Die Metrikdaten jedes Sensors werden in *SimpleJsonTimeseries*-Objekte konvertiert, welche dem erwarteten Format des Grafana SimpleJson Plug-ins entspricht. Listing 4.1 zeigt die Objektstruktur: Zu jeder Sensor-ID wird ein Array von Datenpunkten gespeichert, ein Datenpunkt besteht aus einem Paar von Timestamp und einem zugehörigen Wert.
Die SimpleJsonTimeserie-Objekte aller angefragten Metriken werden in einem Array (vom Typ *SimpleJsonTimeserie[]*) zusammengesetzt und als JSON an das SimpleJson Plug-in gesendet. Das SimpleJson Plug-in visualisiert für jedes SimpleJsonTimeseries-Objekt des Response-Arrays eine Zeitreihe. Werden mehr oder andere SimpleJsonTimeseries-Objekte zurückgesendet als angefragt wurden, werden auch diese visualisiert. Es findet an dieser Stelle also keine Prüfung auf Konsistenz in Grafana statt. Diesen Umstand nutzen wir bei Metriken des Typs *AggregatedActivePowerRecord*. Den Vorgang erläutern wir im folgenden Abschnitt.

Abbildung 4.1 zeigt schematisch den Datenfluss einer Metrikabfrage durch SimpleJson: Die Abfrage wird intern so behandelt, dass eine Abfrage der Schnittstellen des Titan Control Centers möglich wird. Das Abfrageergebnis wird wiederum transformiert und zurückgesendet. Damit adaptiert der REST Adapter beide Schnittstellen und stellt Kompatibilität her.

4.1.3. Aggregated- und ActivePowerRecords

Der History Service hält sowohl Metrikdaten einzelner Sensoren als auch aggregierte Messwerte vor (siehe Abschnitt 2.3.2). Nicht aggregierte Metriken beinhalten neben der *Id* und dem *Timestamp* genau ein Feld für einen Messwert (siehe Listing 2.1). Damit besteht eine 1 : 1 Zuordnung zwischen *Id, Timestamp* und *Value*. Die transformierte *SimpleJsonTimeserie* eines solchen Sensors besteht folglich aus einer Id und *SimpleJson-Datapoints (Timestamp, Value)*-Paaren dieser Metrik. Bei aggregierten Sensoren besteht eine 1 : 5 Beziehung zwischen *Id, Timestamp* und *Values*. Wie aus Listing 2.2 ersichtlich, besitzen aggregierte Sensoren mehrere Felder für Werte. Dabei gibt es Felder für

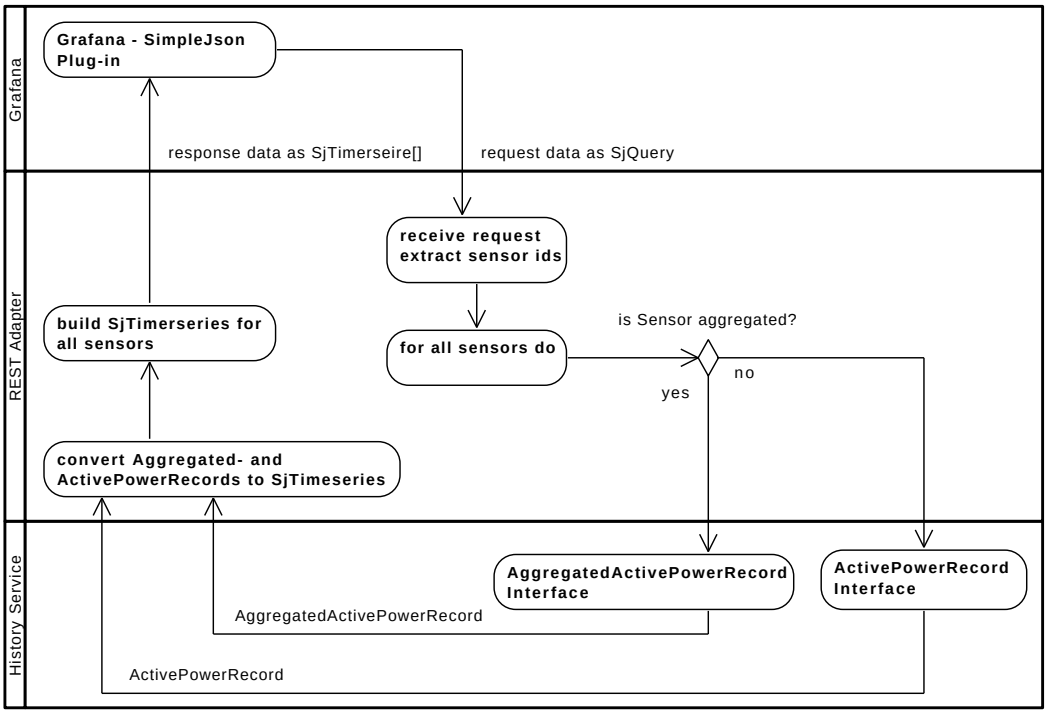

Abbildung 4.1. Schematische Abbildung des Datenflusses einer Metrikabfrage.

folgende Werte: *maxInW, minInW, count, averageInW, sumInW*. Es existieren also für jedes Paar aus Id und Timestamp fünf Werte. Durch die Verwendung von SimpleJson und folglich von *SimpleJsonTimeseries*-Objekten ist in unserem Ansatz eine $1 : 5$ Beziehung nicht möglich. Aus diesem Grund wird die Zeitreihe eines aggregierten Sensors in Form von fünf SimpleJsonTimeseries-Objekten bereitgestellt; wie oben beschrieben visualisiert Grafana alle zurückgesendeten Metriken. Pro Datenwert besteht genau ein SimpleJsonTimeseries-Objekt im Response-Array. Die Namenskonvention der einzelnen SimpleJsonTimeseries in diesem Array ergibt sich durch: *Sensornamen_x*, wobei $x \in \{maxInW, minInW, count, averageInW, sumInW\}$. Die hier vergebenen Sensornamen werden in Grafana als Bezeichnung für die Zeitreihen angezeigt.

4.1.4. Fehlerbehandlung

Der Adapter sendet grundsätzlich auf jede Anfrage von Zeitreihendaten ein *SimpleJsonTimeseries*-Objekt zurück. In dem Fall, dass ein Fehler aufgetreten ist, wird anstelle der angefragten Metriken eine leere Metrik mit dem Namen *error* zurückgesendet. Sollte ein Request nicht erfolgreich ausgeführt werden können, wird in unserem Ansatz nicht automatisch eine erneute Abfrage initiiert. Das liegt daran, dass Grafana automatisch in konfigurierbaren Zeitabständen einen Request ausführt, wodurch die Daten erneut abgefragt werden können.

Sollte während der Abfrage der Konfiguration ein Fehler auftreten, wird abhängig vom auftretenden Fehler eine Metrik *error* oder *could not connect to titan ccp* übertragen.

4.2. Verbindung mittels Apache Kafka und Prometheus

Neben der direkten Verbindung über HTTP/REST stellen wir ein Verfahren vor, um Records aus Kafka zu lesen und in Prometheus abzuspeichern. Das hat zum einen den Vorteil, dass wir den Datenzugriff durch Grafana auf eine Stelle zentrieren können, was nicht zuletzt für den Endbenutzer des Dashboards den Komfort erhöht. Zum anderen profitieren wir durch den Einsatz von Prometheus von den Eigenschaften des Systems. Prometheus ist als Monitoringsystem konzipiert und erlaubt es durch die Abfragesprache PromQL direkt in der Abfrage verschiedene Operationen auszuführen.

Im Folgenden erklären wir, wie die Verbindung zwischen Prometheus, dem Control Center und Grafana aufgebaut ist. Dafür ist in Grafana das Prometheus Plug-in aktiviert, welches Datenbankzugriffe auf die Prometheus-Datenbank durch Grafana ermöglicht. Mithilfe des Plug-ins stellt Grafana Datenbankanfragen über HTTP an die HTTP API des Prometheusservers gesendet. Der Prometheusserver greift zur Datenerhebung auf den *Kafka Exporter* zu und speichert vorhandene Metriken. Der Kafka Exporter wiederum stellt dem Prometheusserver alle über Kafka übertragenen IMonitoringRecords zu Verfügung. Abbildung 3.1, Abbildung 3.2 und Abbildung 5.5 zeigen verschiedene Ansichten auf diesen Vorgang.

Das Grundprinzip unseres Kafka Exporters ist wie folgt aufgebaut und stellt die Umsetzung von Ziel 2 (siehe Abschnitt 1.2) dar. Für die Entwicklung des Kafka Exporters nutzen wir die Prometheus Client Library für Java, welche einige Schnittstellen und Funktionalitäten bereitstellt, um Abfragen des Prometheusservers bearbeiten zu können. Um die Metriken zu erheben, werden verschiedene Topics des eventbasierten Nachrichtensystems Apache Kafka des Titan Control Centers subscript. Die übertragenen Records werden zwischengespeichert und für die Abfrage des Prometheusservers aufbereitet. Bei einem Zugriff durch Prometheus wird auf eine Schnittstelle der Library zugegriffen. Dieser Zugriff initiiert eine Prozedur, in welcher die zwischengespeicherten Metriken transformiert und als Response-Nachricht zurückgesendet werden.

4.2.1. Kafka Exporter

Die *Main-Class* des *KafkaExporters* stellt für den Webzugriff eine HTTP API zur Verfügung, dafür wird eine Instanz eines Jetty-Servers[3] genutzt. Auf diese API greift der Prometheusserver während des Scrape-Vorgangs zwecks Metrikabfrage zu. Um den Zugriff bearbeiten zu können, nutzen wir die Prometheus Library, die Library ermöglicht das Hinzufügen eines Servlets zu einem Jetty-Server. In diesem Fall wird ein *MetricServlet* hinzugefügt, welches einen prometheusspezifischen Abfrageprozess startet. Im Kern dieses Abfrageprozesses werden *Collectoren* aufgerufen, deren Schnittstellen ebenfalls durch die Prometheus Library definiert sind. Beim Zugriff auf eine Collector Instanz wird die *collect()*-Methode aufgerufen, welche hinterlegte Metriken transformiert und in einem Prometheus kompatiblen Format zurückliefert.

Dabei ist es möglich mehrere Collectoren zu hinterlegen, die bei einem Aufruf von Prometheus abgefragt werden sollen. In unserem Fall benutzen wir zwei Collector-Instanzen. Die Prometheus Library stellt zwei zentrale Mechanismen bereit, der erste Mechanismus ermöglicht es einen Collectoren zu registrieren. Der Zweite ermöglicht es, dass das eingesetzte MetricServlet bei allen registrierten Collectoren eine Metrikabfrage stellt.

Neben den Collectoren instanziiert unsere Implementierung zwei Kafka Consumer Objekte, welche die Topics für *ActivePowerRecords* und *AggregatedActivePowerRecords* subscriben. Jeder gelesene Record wird durch den Kafka Subscriber einem Collector hinzugefügt. Der Collector speichert diese und stellt die Metriken beim Scrape-Vorgang durch Aufruf der *collect()*-Methode bereit.

4.2.2. Collectoren

Die Komponente des Kafka Exporters nutzt zwei verschiedene Collectoren, einen typisiert für *ActivePowerRecords*, einen für *AggregatedActivePowerRecords*. Eine Collector-Instanz greift intern auf eine Implementierung des Interfaces *Adapter* zu, welche entsprechend dem Typ des Collectors typisiert ist. Das Adapter Interface stellt Schnittstellen zur Transformierung

[3] https://www.eclipse.org/jetty/

von IMonitoringRecord-Objekten zu MetricFamilySamples zur Verfügung. Es werden also Formate, die innerhalb der Titan Plattform verwendet werden, in Formate umgewandelt, die durch Prometheus weiterverarbeitet werden können. Derzeit gibt es Implementierungen für den Typ ActivePowerRecord und AggregatedActivePowerRecord. Entsprechend der generischen Implementierung unterscheiden sich die Collectoren während der Ausführung ausschließlich in ihrer Typisierung und dem verwendeten Adapter.

Die Collector-Klasse bietet zwei Schnittstellen: die *add()-* und die *collect()-Methode*. Über den Aufruf der add()-Methode können Records des Typs IMonitoringRecord gespeichert werden. Der Speichervorgang wird durch einen KafkaConsumer immer dann ausgelöst, wenn ein Record aus einem Topic gelesen wurde. Zum Zwischenspeichern von Records nutzen wir an dieser Stelle eine Multimap, welche die Sensor-ID eines Records und die Record-Objekte derart mapt, dass unter einer ID alle zugehörigen Records hinterlegt sind.

Der Hauptpart des Collectors stellt die collect()-Methode dar, welche verfügbare Records zu Prometheus kompatiblen Metriken transformiert, Listing 4.2 zeigt den Quellcode der Methode. In unserem Ansatz wird in einem collect() Aufruf zunächst eine temporäre Liste der zwischengespeicherten Records erzeugt und die Liste der zwischengespeicherten Records geleert. Zugriffe dieser Art finden *synchronized* statt. Anschließend rufen wir für jede Liste von Records (also für alle Records einer ID) die *transform()*-Methode des Adapters auf, welche die Metriken transformiert und zurückgibt.

Transformation der Records zu Prometheus Metriken Die *transform()-Methode* erstellt für jeden Record ein Sample-Objekt, das sind Datenpunkte einer Metrik zu einem bestimmten Zeitpunkt. Listing 4.2 zeigt in Zeile 11 den entsprechenden Aufruf. Sample-Objekte bestehen aus Namen, Label Namen, Label Values, Value und Timestamp. Da unsere Metriken nicht gelabelt sind, übergeben wir stattdessen leere Listen. Die erstellten Sample-Objekte werden anschließend genutzt, um ein *MetricFamilySamples*-Objekt zu erzeugen. Ein MetricFamilySamples-Objekt besteht aus Namen, Metrik-Typ, einem Help String und einer Liste von Samples. Der Name dieser Zeitreihe entspricht dem Identifier der Records. Als Metrik-Typ haben wir *Gauge* gewählt, eine Metrik-Art, deren Werte beliebig erhöht oder verringert werden kann. Der *Help-String* kann im Nachhinein angezeigt werden und ermöglicht die Übergabe weiterer Informationen, wir unterscheiden an dieser Stelle zwischen *ActivePowerRecords* und *AggregatedActivePowerRecords*. Nachdem dieser Vorgang für alle gespeicherten IDs durchgeführt wurde, wird eine Liste von MetricFamilySamples zurückgegeben.

4.2.3. Aggregated- und ActivePowerRecords

Aggregated- und ActivePowerRecords speichern unterschiedliche Werte. AggregatedActivePowerRecords speichern fünf Werte für eine aggregierte Gruppe (siehe Abschnitt 2.3.2) von Sensoren. Eine Gruppe von Sensoren bedeutet den logischen Zusammenschluss mehrerer Sensorwerte. ActivePowerRecords speichern die Messwerte eines einzelnen Sensors.

```
1   @Override
2   public List<MetricFamilySamples> collect() {
3     final List<MetricFamilySamples> metrics = new ArrayList<>();
4     final Multimap<String, T> tmp;
5
6     synchronized (this.records) {
7       tmp = LinkedListMultimap.create(this.records);
8       this.records.clear();
9     }
10    tmp.asMap().forEach((k, v) -> {
11      metrics.addAll(this.adapter.transform(v));
12    });
13
14    return metrics;
15  }
```

Listing 4.2. Collect()-Methode des ActivePowerRecordCollector

Vor diesem Hintergrund haben wir zwei entsprechende Adapter-Implementierungen entwickelt.

Der ActivePowerRecordAdapter erstellt für jede Liste von Records einer ID ein MetricFamilySamples-Objekt. Der AggregatedActivePowerRecordAdapter hingegen speichert für jede Liste von Sensoren einer ID fünf Sample-Listen, für jeden hinterlegten Wert eine. Anschließend wird für jede Sample-Liste ein MetricFamilySamples-Objekt erzeugt. Die Namenskonvention ist an dieser Stelle analog zum Vorgehen des REST Adapters (siehe Abschnitt 4.1). Das bedeutet, dass für jeden aggregierten Sensor fünf Zeitreihen in Prometheus angelegt werden, welche separat abgefragt werden können.

Verwendung mehrerer Zeitreihen anstelle einer gelabelten Zeitreihe Wir haben uns für die Erzeugung von fünf separaten Zeitreihen und gegen die Verwendung von Labels entschieden. Da Prometheus jede eindeutige Kombination aus *Label Namen* und *Label Value* intern in Form einer neuen Zeitreihe speichert, ergäbe sich durch die Einführung von Labels kein Effizienzgewinn. Außerdem müsste eine Namenskonvention entweder beim Vorgang des Abspeicherns in Prometheus oder bei Erstellung der Aggregation derart vorgenommen werden, dass kenntlich wird, welche Metrik welche Labels als Filteroption unterstützt.

4.2.4. Wiederverwendbarkeit

Wir haben bei unserer Implementierung einen Fokus auf die Wiederverwendbarkeit des Codes gelegt. Vor diesem Hintergrund haben wir die entscheidenden Komponenten

generisch implementiert, sodass mit geringem Aufwand weitere Record-Arten des Typs IMonitoringRecord verarbeitet werden können.

4.3. Verbindung der Flow Engine mit Prometheus

Ein weiterer Aspekt unseres Ansatzes ist die Herstellung der Verbindung zwischen der Titan Flow Engine und Prometheus, wieder mit dem Ziel, die Daten mit einem Dashboard Framework wie Grafana visualisieren zu können. Der Aufbau dieser Verbindung entspricht der Umsetzung von Ziel 3 (siehe Abschnitt 1.2). Dazu stellen wir zunächst ein Verfahren vor, wie wir direkt über Bricks zusammen mit einem Python Server-Skript Daten in Prometheus übertragen können. Anschließend zeigen wir auf, wie Informationen über die Titan Flow Engine erhoben und über die Kommunikationskanäle des Titan Control Centers an Prometheus übertragen werden können.

4.3.1. Direkte Verbindung der Flow Engine mit Prometheus

In Kapitel 5 zeigen wir einen Weg, wie Daten durch die Flow Engine eingelesen und derart verarbeitet werden können, dass diese als Metriken vorliegen. Wir gehen an dieser Stelle davon aus, dass bereits Metriken zum Abspeichern vorliegen.

Der Verbindungsaufbau besteht im Kern aus drei Komponenten. Die erste Komponente ist ein Prometheus-Exporter-Brick, welcher als Bestandteil eines Flows eingehende Metriken abspeichert. Die zweite Komponente ist eine Datenbank, die innerhalb und außerhalb der Flow Engine erreichbar ist. Die dritte Komponente ist ein Server-Skript, welches Metriken für Prometheus bereitstellt. Im Folgenden zeigen wir auf, wie wir diese Komponenten implementiert haben.

Wir haben die Implementierung dieser Komponenten auf die in Kapitel 5 dargelegten Datenformate angepasst. Konzeptionell ist die Implementierung für andere Formate und Metriken analog aufgebaut. In Abschnitt 4.3.2 zeigen wir einen weiteren Weg, wie Daten über die Titan Flow Engine erhoben und in Prometheus abgespeichert werden können.

Prometheus-Exporter-Brick

Der letzte Brick eines Flows, der Daten in Prometheus abspeichern soll, ist der Prometheus-Exporter-Brick. Dieser Brick nimmt Metrikdaten entgegen und sorgt ausschließlich dafür, dass alle Records einer ID in einer Liste abgespeichert werden. Auf diese Daten wird anschließend durch das im Folgenden beschriebene Server-Skript zugegriffen. Die Datenbank, in der die Records zwischengespeichert werden, stellt die Schnittstelle zwischen dem Flow der Titan Flow Engine und dem Server-Skript dar. Als Datenbanktechnologie wird die leichtgewichtige Key-Value Speicher Python PickleDB[4] angesetzt.

[4] https://pythonhosted.org/pickleDB/

Server-Skript

Das *Server*-Skript stellt eine HTTP API bereit, auf welche der Prometheusserver im Rahmen des Scrape-Vorgangs zugreift. An dieser Stelle nutzen wir die Prometheus Client Library für Python. Diese Library stellt analog zu den beschriebenen Mechanismen der Java Library Funktionalitäten bereit, welche den Zugriff durch Prometheus ermöglichen. Dafür werden die Daten in Formate transformiert, die von Prometheus interpretiert und gespeichert werden können.

Scrape-Vorgang Prometheus ist als Monitoringsystem derart aufgebaut, dass ein Fokus auf aktuelle Messwerte gelegt wird. Vor diesem Hintergrund werden Metriken, die nicht in einem Intervall von +/- 60 Minuten zum Zeitpunkt des Scrape-Vorgangs liegen, nicht abgespeichert. Das Server-Skript stellt die Records zu einer Liste von *MetricFamilySamples* derart zusammen, dass lediglich diejenigen Records an Prometheus gesendet werden, deren Timestamps im entsprechenden zeitlichen Intervall liegen. Dieses Vorgehen ist nötig, da Records mit unterschiedlichen Timestamps eingelesen werden können. Um Fehler zu vermeiden, haben wir dieses Zeitfenster etwas kleiner als jenes gesetzt, welches Prometheus akzeptiert. Das Zeitfenster unserer Implementierung ist definiert durch $t_{min} < t_{record} < t_{max}$ wobei $t_{min} = t_{current} - 15sek$ und $t_{max} = t_{current} + 5sek$ gilt. Diejenigen Records, die in diesem Intervall liegen, werden an Prometheus gesendet und aus der Datenbank gelöscht. Die Records, für die $t_{record} < t_{min}$ gilt, liegen außerhalb des akzeptierten Zeitintervalls und werden verworfen, da die Records zu alt sind. Records, für die gilt $t_{record} > t_{max}$, werden zunächst unberührt gelassen und zu einem späteren Zeitpunkt erst wieder betrachtet, wenn eine der ersten beiden Bedingungen zutrifft.

Der Fall, dass Metriken verworfen werden, tritt nur dann ein, wenn ein langfristiges Problem mit dem Scrape-Vorgang auftritt. Andernfalls ist unser Prometheusserver so konfiguriert, dass dieser in Zeitintervallen von wenigen Sekunden den Exporter nach neuen Metriken abfragt. Bis ein Sample außerhalb des Zeitintervalls liegt sind mehrere Scrape-Vorgänge durchgeführt worden.

4.3.2. Indirekte Verbindung der Flow Engine mit Prometheus

Im vorherigen Abschnitt haben wir gezeigt, wie die Anbindung der Flow Engine an das Prometheus-Monitoringsystem aufgebaut werden kann. Zu diesem Zweck haben wir einen direkten Zugriffspunkt erstellt, um Metriken mithilfe von Prometheus abzugreifen. In diesem Abschnitt nutzen und manipulieren wir bereits bestehende Bricks derart, dass ActivePowerRecords, die über die Flow Engine erhoben und mithilfe von Kafka an das Titan Control Center gesendet werden, über den bereits vorgestellten Kafka Exporter an Prometheus exportiert werden können.

Energy Management Flow

Zum Zeitpunkt dieser Arbeit existieren bereits verschiedene Bricks, um das Einlesen der Messdaten zu bewerkstelligen. Der dazugehörige Flow besteht aus zwei Bricks, ein erster Input-Brick liest die Messdaten, transformiert diese in ActivePowerRecords und sendet sie an einen zweiten Brick, den Control Center Adapter-Brick. Dieser Brick sendet die ActivePowerRecords in ein zugehöriges Kafka Topic. Dieses Kafka Topic wird, wie in Abschnitt 4.2.1 beschrieben, durch den Kafka Exporter für den Prometheusserver zugänglich gemacht. Insgesamt werden die Messdaten also eingelesen, transformiert, in Kafka publiziert, und wieder aus Kafka ausgelesen, um letztendlich einen Zugriff durch den Prometheusserver zu ermöglichen. Vor dem Hintergrund, dass auf diesem Weg auch Dateien mit alten Messwerten eingelesen werden können, haben wir einen Mechanismus implementiert, der die Timestamps[5] der Records versetzt. Dieses Vorgehen ist nötig, da Prometheus alte Timestamps nicht akzeptiert. Dafür haben wir den Input-Brick derart angepasst, dass dieser beim Einlesen von Daten zwischen dem Absenden zweier Records wartet. Die Wartezeit des Bricks entspricht genau der zeitlichen Differenz, die zwei Records auseinanderliegen. Ein Record, der herausgesendet wird, bekommt den Timestamp des aktuellen Zeitpunktes übergeben. Der Adapter-Brick schreibt alle empfangenen Records direkt in Kafka, woraus diese ohne weitere Überprüfung des Timestamps für Prometheus bereitgestellt werden.

4.4. Das Grafana Dashboard

Unser Entwurf beinhaltet verschiedene vorkonfigurierte Dashboards, welche ausgewählte Messdaten zum Teil in korrelierter Form visualisieren. Abbildung 4.2 und Abbildung 4.8 zeigen Screenshots von zwei Dashboards, die Demowerte der Titan Plattform anzeigen. Der Grund für zwei Dashboards ist zunächst, dass auf diese Art zusammengehörende Informationen gekapselt angezeigt werden können. Darüber hinaus ist es möglich, für jedes Dashboard einen Zeitraum zu konfigurieren, in welchem Daten angezeigt werden.

In Grafana existieren Benutzerkategorien mit unterschiedlichen Zugriffsberechtigungen. Im Folgenden beschreiben wir den Blickwinkel eines Administrators. Abhängig von der jeweiligen Benutzergruppe kann es sein, dass nicht alle Optionen zugreifbar sind.

Das Grafana Dashboard Framework ermöglicht das einfache Erstellen und Editieren von Dashboards. Ein Dashboard besteht aus verschiedenen Panels. Panels sind Anwendungen, welche verschiedene Funktionalitäten zur Verfügung stellen und oftmals der Visualisierung von Metriken dienen. Die Grafana Oberfläche bietet außerdem die Möglichkeit, eigene Panels zu entwickeln und diese in ein Dashboard zu integrieren.

Für das Visualisieren von Daten bietet Grafana ein Repertoire verschiedenster Anzeigeformate, welche für unterschiedliche Information prädestiniert sind [Khan und Shah 2011]. Im Folgenden stellen wir die wichtigsten von uns genutzten Panels (Darstellungsformen)

[5] Zeitpunkt der Messung/des Log-Eintrags

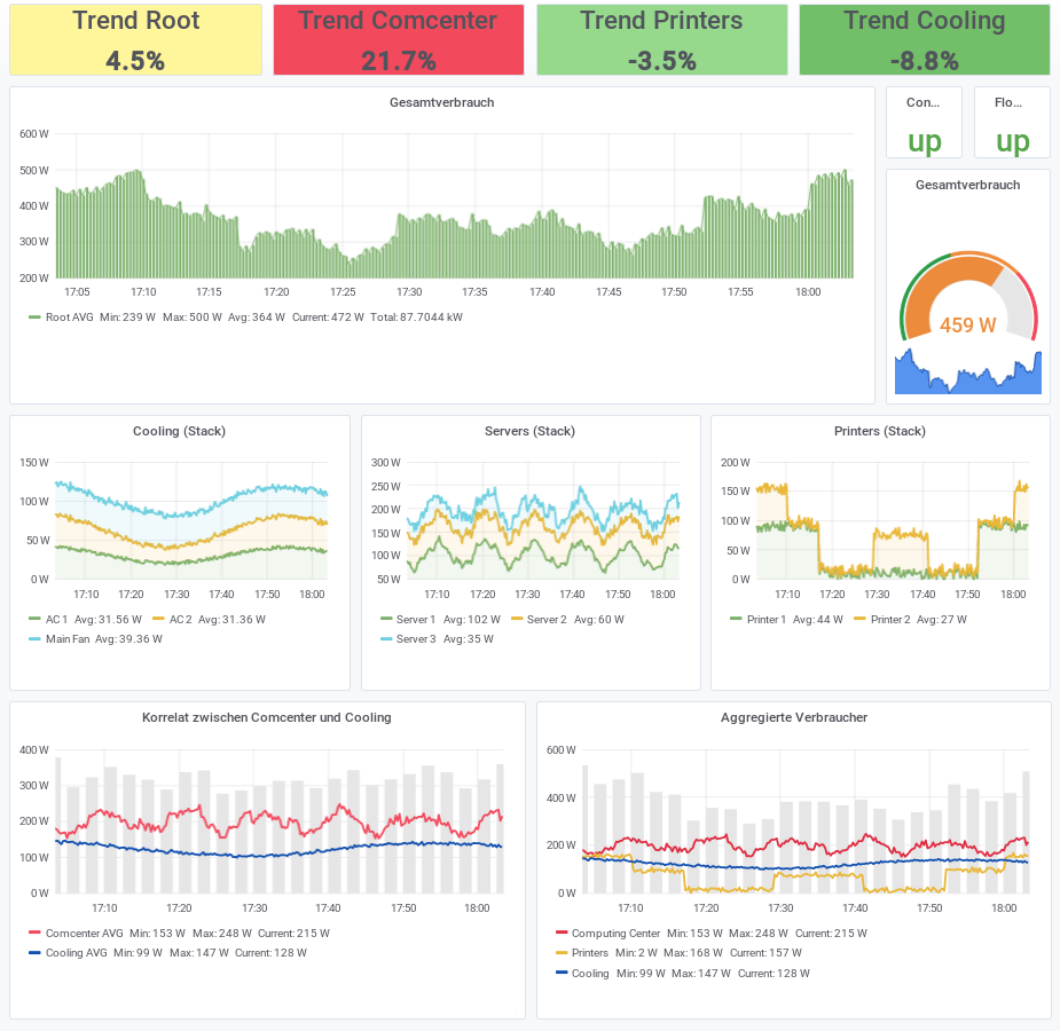

Abbildung 4.2. Ansicht auf die Übersichtsanzeige des Grafana Dashboards. Im oberen Bereich haben wir Trendanzeigen platziert, die den prozentualen Trend in der Zeitperiode des Dashboards anzeigen. Darunter befindet sich die Anzeige des Gesamtverbrauchs in Form eines Graphen und eines Gauges. Die *up*-Anzeigen über dem Gauge zeigen an, dass beide Prometheus Exporter online sind. Anschließend zeigen wir Stack-Diagramme für die physikalischen Sensoren aggregierter Sensoren. Im unteren Bereich zeigen wir verschiedene Korrelationen. Im Hintergrund der Korrelationen zeigen wir die jeweils angezeigten Metriken kumuliert in Form von Balkendiagrammen.

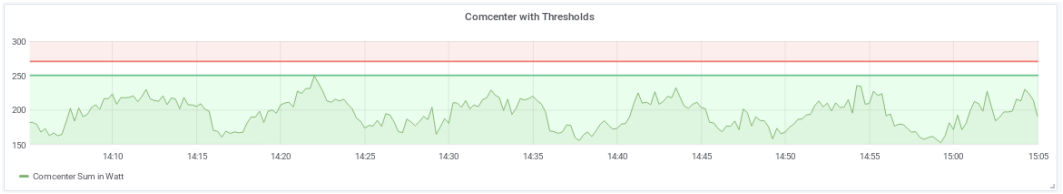

Abbildung 4.3. Grafana Graph Panel mit Schwellenwerten.

vor und zeigen exemplarisch Konfigurationsmöglichkeiten auf, durch welche Darstellungsformen stark angepasst werden können. Anschließend gehen wir auf Grafana Variablen (siehe Abschnitt 4.4.2) ein, welche eine dynamische und interaktive Nutzung eines Grafana Dashboards ermöglichen.

4.4.1. Primär eingesetzte Darstellungsformen

Graph Panel Ein Graph Panel ist eine Visualisierungsform, welche Zeitreihen in einem klassischen Graphen abbildet. Dabei wird auf der Abszissenachse die Zeit und auf der Ordinatenachse der Wert eines Messpunktes abgebildet. Das Graph Panel bietet diverse Möglichkeiten an, die Darstellungsform zu variieren. Beispielsweise können im Graph Panel Minimum-, Maximum- oder Durchschnittswerte einer Zeitreihe in Form einer Tabelle angezeigt werden. Die entsprechende Berechnung erfolgt im Panel selbst.

Abbildung 4.3 zeigt eine weitere Modifikation. In diesem Fall werden bestimmte Wertebereiche farblich markiert, um einzuordnen, in welchem Bereich der jeweilige Wert anzusiedeln ist. In diesem Fall sind Messungen < 340 Watt im grünen, Messungen > 350 Watt im roten Bereich.

Das Graph Panel kann außerdem als Balken- oder Tabellendiagramm sowie als Anzeige für Histogramme genutzt werden. Unabhängig von der Anzeigeart ist das Graph-Panel in der Lage die gewählte Darstellungsform als Stack-Version[6] anzuzeigen. Im folgenden Abschnitt zeigen wir, wie prägnante Werte einer Zeitreihe zusammen mit der Kurve der Zeitreihe visualisiert werden können.

Graph Panel als Extremaintervall Wie erläutert bietet Grafana verschiedenste Visualisierungsformen, ein Beispiel ist hier das oben beschriebene Graph Panel. Wir zeigen nun eine Variante, wie wir Metriken einer Zeitreihe zusammen mit prägnanten Punkten derselben Zeitreihe der letzten 30 Sekunden anzeigen. Das Ziel dieses Vorgehens ist es, den aktuellen Messpunkt mit vorherigen Werten vergleichen zu können. Ein weiterer Vorteil dieser Variante ist, dass die Kurve geglättet angezeigt wird. Dafür zeigen wir die Kurve

[6]Stack-Diagramm (Stapel-Diagramm) ist eine Diagrammart, welche mehrere Teilwerte (hier Zeitreihen) eines Kriteriums (bspw. Watt) im selben Diagramm übereinandergestapelt anzeigt. Dadurch wird der Gesamtwert (bspw. gemeinsamer Stromverbrauch) gebildet und gleichzeitig der Anteil jedes Teilwertes (bspw. ein Sensor) angezeigt.

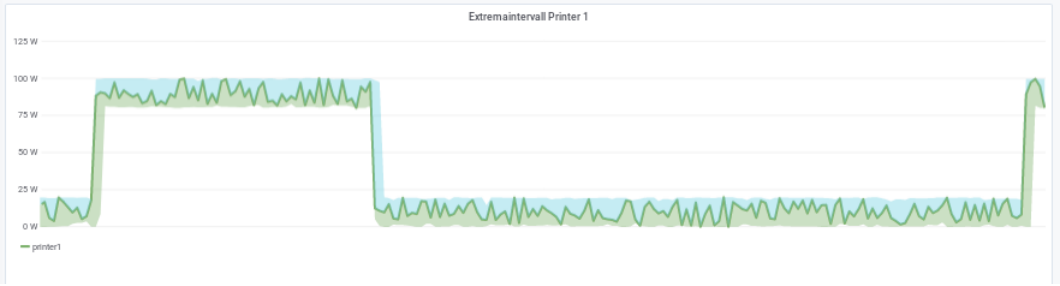

Abbildung 4.4. Grafana Graph Panel, konfiguriert als Extremaintervall Graph

dieser Metrik zusammen mit dem höchsten und dem niedrigsten Wert der Zeitreihe in den letzten 30 Sekunden an.

Abbildung 4.4 zeigt einen Graphen, der als Extremaintervall konfiguriert ist. Die in diesem Fall blau ausgefüllte Fläche ist der Bereich, der einen höheren Wert als der jeweilige Messpunkt innehat, der grüne Bereich zeigt Werte unterhalb des Messpunktes an. Die für diese Darstellung benötigten Daten werden folgendermaßen generiert: Zunächst stellen wir drei Datenbankabfragen an Prometheus. Die erste Abfrage fragt die Metrik ab, beispielhaft nutzen wir an dieser Stelle die Metrik *printer1*. Zwei weitere Abfragen geben Auskunft über die minimale respektive maximale Messung der letzten 30 Sekunden. Für diese Abfrage nutzen wir die *Prometheus Query Language (PromQL)*. Die zweite Anfrage ist `min_over_time(printer1[30s])`, die dritte Anfrage `max_over_time(printer1[30s])`. Die Aufrufe `min_over_time()` und `max_over_time()` entsprechen dabei Funktionen, welche direkt zu PromQL gehören. Die eckigen Klammern [30s] geben den Zeitraum (in diesem Fall 30 Sekunden) an, in welchem das Minimum bzw. Maximum gesucht wird. Mithilfe dieses Vorgehens ist es ebenso möglich, ein Konfidenz-Intervall anzuzeigen, wenn die entsprechenden Werte dafür vorliegen.

Die abgefragten Metriken ermöglichen nun die graphische Darstellung eines Intervalls als Kurve eines Graphen. Dazu nutzen wir die Einstellungsmöglichkeiten Grafanas zur Anpassung der Visualisierung. Zunächst setzen wir die Mode-Option Fill auf 0, dadurch erreichen wir, dass die Fläche unter einer Kurve nicht ausgefüllt wird. Im nächsten Schritt überschreiben wir die aktuellen Darstellungen. Die Anzeige der Linien der Graphen von *max Printer* und *min Printer* deaktivieren wir und sorgen dafür, dass die Fläche zwischen der Kurve von max Printer und Printer1 (bzw. min Printer und Printer1) ausgefüllt wird.

Bar Gauge Eine weitere Form der Visualisierung ist der *Bar Gauge*. Dieser zeigt in Form von verschiedenen Balken die aktuellen Werte von Metriken. Farblich sind die Balken in Intervalle unterteilt, in unserem Fall in Grün, Orange und Rot. Abbildung 4.5 zeigt ein Bar Gauge Diagramm, welches den Verbrauch von drei Servern je einmal einzeln und einmal aufsummiert anzeigt. Das Bar Gauge Diagramm bietet die Möglichkeit konkrete Werte einzelner Metriken und deren Differenzen zu anderen sichtbar zu machen. Grafana stellt

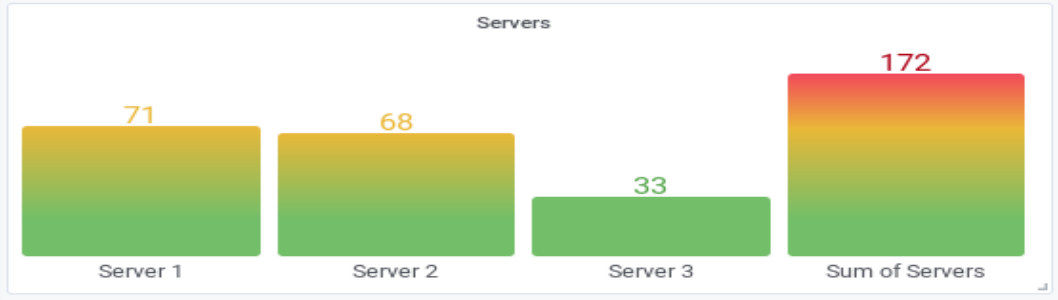

Abbildung 4.5. Grafana Bar Gauge Panel. Visualisiert werden Metriken für Server1, Server2, Server3, der vierte Balken zeigt den aktuellen kumulierten Verbrauch aller Servereinheiten.

hierfür verschiedene Spezifikationsmöglichkeiten zur Entscheidung über die Auswahl der angezeigten Werte bereit. Insbesondere ist es möglich den letzten Messpunkt zu wählen oder sich für eine der folgenden Optionen zu entscheiden: First, Min, Max, Mean, Total. Eine andere Möglichkeit ist es, die letzten n Messpunkte einer Metrik durch n Balken anzuzeigen (wobei $n \in \mathbb{N}$).

SingleState Ein weiteres Panel ist das Singlestat Panel, in diesem kann lediglich eine einzelne Metrik visualisiert werden. Für die Anzeige der Metrik gibt es jedoch vielfältige Darstellungsformen. Abbildung 4.6 zeigt exemplarisch eine Variante, in welcher in der Mitte ein *Gauge* abgebildet ist, der den aktuellen Messwert zeigt. Der Wert wird einerseits numerisch, andererseits durch eine halbrunde Grafik dargestellt. Im Hintergrund wird die Metrik durch einen blauen Graphen angezeigt. Ein Vorteil dieses Panels ist es, dass viele verschiedene Informationen auf einen Blick erkennbar sind. Ein Nachteil ist, dass eine Kombination von verschiedenen Zeitreihen nicht möglich ist.

Trend Box Das Trend Box Panel ermöglicht es, den Trend einer Zeitreihe in Prozent zu ermitteln und anzuzeigen. Die Berechnung des Trends erfolgt durch Kalkulation der prozentualen Differenz des ersten und des letzten Messpunktes einer Metrik, wobei sich der erste und der letzte Punkt auf den Abfragezeitraum beziehen. Dabei ist es möglich, unterschiedlichen Prozentbereichen unterschiedliche Farben zuzuordnen.

Diagramm Panel Das Diagramm Panel ermöglicht es Diagramme wie Flow-Charts oder Gantt-Charts zu erstellen. Dafür wird auf mermaid.js[7] zurückgegriffen. Diagramme werden in der Mermaid-Syntax definiert. Die einzelnen Elemente eines Diagramms können beispielsweise die Werte einzelner Zeitreihen anzeigen und entsprechend eingestellten Schwellenwerten die Farbe ändern. Wir können die so erstellten Diagramme nutzen, um die derzeitige Konfiguration grafisch darzustellen.

[7] https://mermaidjs.github.io/

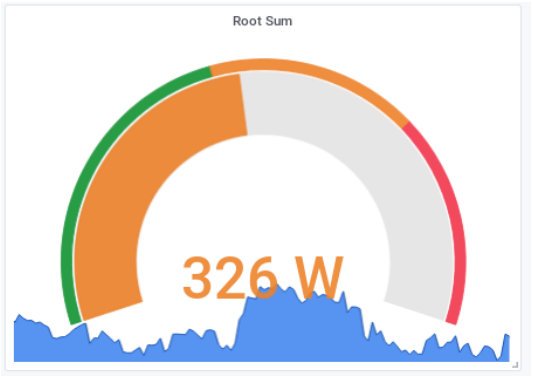

Abbildung 4.6. Grafana Single State Panel. Exemplarisch werden die Metrikdaten der Zeitreihe root_sumInW genutzt. Der Name der Zeitreihe ergibt sich aus der ID des entsprechenden aggregierten Sensors. In diesem Fall steht hinter der Bezeichung *root* der Gesamtverbrauch.

4.4.2. Dashboard Interaktion

Eine besondere Bedeutung für Dashboardlösungen besteht in der Möglichkeit zur Interaktion zwischen Nutzer und Dashbord. Solche Interaktionsmöglichkeiten können dem Nutzer dabei helfen, Informationen und Daten zu verstehen, zu analysieren und zu explorieren [Khan und Shah 2011].

Zur Interaktion bietet Grafana neben der Möglichkeit in die Darstellungen von Metriken hinein- oder herauszuzoomen, den Einsatz von individuell wählbaren Variablen an. Wie im folgenden Abschnitt erläutert, kann ein Benutzer mithilfe von Variablen dynamisch auswählen welche Metriken angezeigt werden.

Grafana Variablen

Grafana bietet die Möglichkeit Variablen zu definieren und in den Abfragen an verschiedene Datenquellen einzusetzen. Die Variablen werden im oberen Bereich eines Dashboards in Form eines Drop-Down-Menüs angezeigt, sodass ein Dashboard-Benutzer die Variablenbelegung auswählen kann. Ändert der Benutzer auf diese Art und Weise den Wert einer Variablen, so werden alle Panels aktualisiert, die diese Variable in einer Datenbankabfrage benutzen, sodass die ausgewählte Metrik angezeigt wird.

Variablen können verschiedene Typen besitzen. So ist es beispielsweise möglich, die auswählbaren Werte einer Variablen direkt festzulegen oder - abhängig von der eingesetzten Datenquelle - eine Abfrage zu stellen, die alle Metriknamen zurückliefert, die mit einem bestimmten Präfix übereinstimmen. Dieses Präfix kann in Form eines regulären Ausdrucks übergeben werden.

Wir haben auf der zweiten Oberfläche unserer Dashboardlösung Variablen eingesetzt.

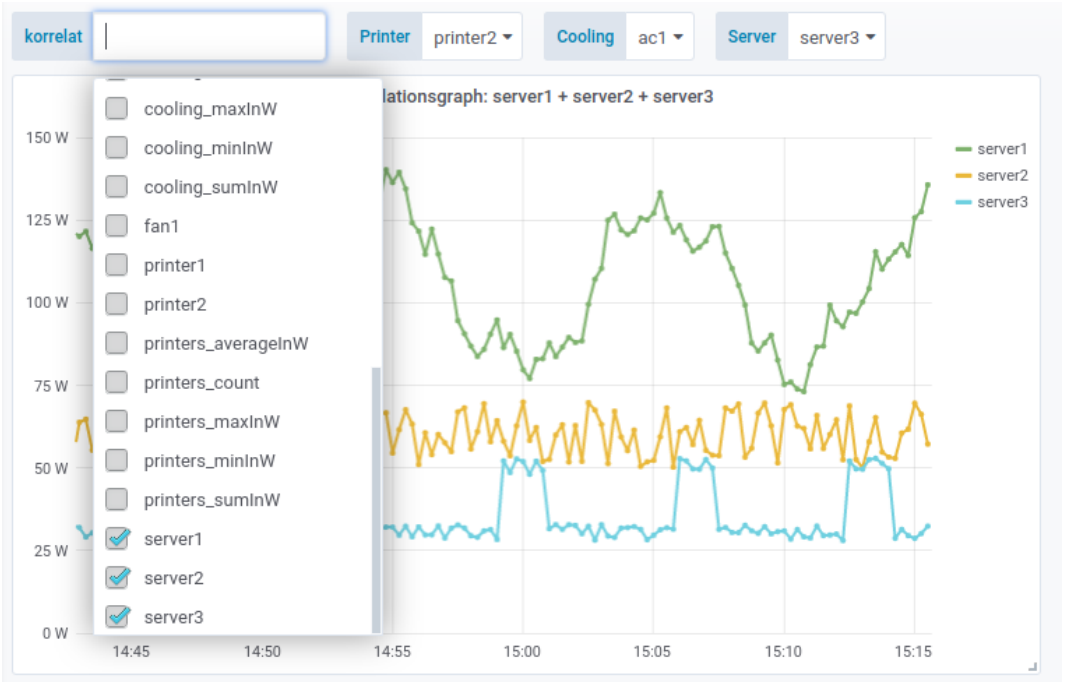

Abbildung 4.7. Menüband der Grafana Variablen. An dieser Stelle sind server1, server2 und server3 ausgewählt. Die Bezeichnungen der Zeitreihen entsprechen der eingeführten Namenskonvention.

Abbildung 4.8 zeigt die Oberfläche, Abbildung 4.7 einen vergrößerten Ausschnitt des Menübands der Grafana Variablen derselben Oberfläche. Im oberen Bereich haben wir ein großes Graph Panel angeordnet, welches dynamisch auf Grundlage der Werte der Grafana Variable *korrelat* Metriken lädt und anzeigt. Daneben ist ein Balkendiagramm, welches ebenfalls die Werte der Variable korrelat anzeigt. Das Balkendiagramm zeigt dabei als Stack-Diagramm den prozentualen Anteil einer Metrik an dem Gesamtwert. Wir haben das Dashboard so konfiguriert, dass die Variable mehrere Werte beinhalten kann. Ein Benutzer kann sowohl eine Zeitreihe anzeigen lassen, als auch beliebig viele Metriken in Form von mehreren Kurven korrelieren. Abgesehen von dem beschriebenen Graph Panel haben wir weitere Darstellungsformen mit Variablen belegt. In diesem Fall kann ein Benutzer je eine detaillierte Darstellung eines Sensors aus der Gruppe Server, Printer, Cooling anzeigen lassen.

Abbildung 4.8. Detail- und Korrelationsoberfläche des Grafana Dashboards. Im oberen Bereich ist ein Graph, über den mithilfe von Variablen verschiedene Korrelationen aufgezeigt werden können. Rechts neben dem Graphen haben wir ein Balkendiagramm konfiguriert, welches dieselbe Variable benutzt wie der Graph. Das Balkendiagramm ist als Stack-Diagramm konfiguriert, welches den Anteil einer Metrik in Prozent visualisiert. Darunter haben wir Graphen, die einen auswählbaren Wert aus den Bereichen Printer, Server oder Cooling darstellen. Jeweils zugehörig zu den Gruppen sind Singlestat Panels vorhanden, die die Werte der physikalischen Sensoren illustrieren. Für jede Gruppe haben wir im unteren Bereich exemplarisch eine weitere Darstellung gewählt: das Extremaintervall für Printer1, ein Kreisdiagramm für das Cooling und ein Bar Gauge für die verschiedenen Server.

Kapitel 5

Industrielle Fallstudie: Zeitungsdruckerei

In diesem Kapitel zeigen wir, wie unser theoretischer Ansatz für das Monitoring industrieller Produktion praktisch eingesetzt werden kann. Als professionelle Produktionsumgebung integrieren wir an dieser Stelle ein industrielles Druckzentrum. In einem Druckzentrum liegen verschiedene Messwerte vor, die unterschiedlichen Informationsgehalt über Stromverbrauch und Produktionsdetails besitzen. Beispielhaft akquirieren wir verschiedene Log- und Messwerte des Druckzentrums der Kieler Nachrichten. In Abschnitt 5.1 und Abschnitt 5.2 zeigen wir, wie eine Integration sowohl über die Flow Engine als auch über das Titan Control Center aufgebaut werden kann. In Abschnitt 5.3 zeigen wir, wie wir die Daten korreliert in Grafana visualisiert haben. Abschließend stellen wir in Abschnitt 5.4 ein Dashboard für das Druckzentrum vor.

5.1. Integration von Messwerten über das Control Center

Zum Zeitpunkt dieser Arbeit liegen uns mehrere Dateien vor, die Messungen von ActivePowerRecords und AggregatedActivePowerRecords beinhalten. Die Visualisierung der Messwerte des Druckzentrums ist an dieser Stelle prototypisch, da wir diese Messwerte direkt in die Cassandra Datenbank des History Services importiert haben, ohne diese über die Infrastruktur der Titan Plattform zu erheben. Der Grund dafür ist, dass uns für diese Arbeit ausschließlich ein Datensatz mit den Messwerten und keine direkten Anbindungen der Sensoren vorliegen. Das ist auch der Grund dafür, dass wir an dieser Stelle nur Messwerte eines bestimmten Zeitraumes abbilden können und keine Visualisierung von aktuell erhobenen Daten stattfindet. Für den History Service ist dieser Umstand nicht von Bedeutung, sodass die Schnittstellen ohne Einschränkungen genutzt werden können.

Zur Visualisierung dieser Daten nutzen wir den in Abschnitt 4.1 vorgestellten REST Adapter, indem wir über Grafana und den REST Adapter auf den History Service zugreifen.

5.2. Integration von Messwerten über die Flow Engine

Für die Integration von Records über die Flow Engine gehen wir nach dem in Abschnitt 4.3 vorgestellten Konzept vor und entwickeln verschiedene Bricks. Die entwickelten Bricks sind

spezifisch an die Formate und Protokolle des Druckzentrums angepasst und ermöglichen das Einlesen von Ereignisprotokolldateien. In dem hier betrachteten Anwendungsfall steht eine Datei zur Verfügung, die Informationen über den Druckbetrieb des Druckzentrums der Kieler Nachrichten beinhaltet. Auf Grundlage dieser Daten berechnen wir Durchschnittswerte für die Druckgeschwindigkeit und extrahieren Werte der Rotationsgeschwindigkeit der Druckmaschinen. Diese Informationen stellen wir dem Prometheusserver in Form verschiedener Zeitreihen zur Verfügung.

5.2.1. Entwicklung der Bricks

Im Folgenden erläutern wir verschiedene Bricks, deren Kombinationen die prototypischen Flows ergeben, welche die verschiedenen Ereignisprotokolldateien einlesen, aufbereiten und für Prometheus bereitstellen.

Zwei Input-Bricks lesen log-Daten einer XML- bzw. CSV-Datei, transformieren diese in Ujo und senden diese an den nächsten Brick. Der darauffolgende Brick sorgt für eine Konvertierung der gelesenen VIP-Records in *vip-log-records*. Diese enthalten Informationen über die Anzahl der pro Stunde angefertigten Bruttokopien (intern als GrossCopies bezeichnet)[1] respektive Nettokopien (intern als NetCopies bezeichnet)[2] und die Rotationsgeschwindigkeit der Druckmaschine. Diese Records werden an einen weiteren Brick gesendet, welcher die Records abspeichert. Auf diese abgespeicherten Records greift wiederum ein Python Server-Skript zu. Das Server-Skript wird durch den Prometheusserver zur Metrikabfrage aufgerufen, die abgefragten Metriken werden in Prometheus gespeichert. Abbildung 5.1 visualisiert das Zusammenwirken der unterschiedlichen Komponenten.

XML-Input-Brick

Der erste Brick unseres Flows ist ein Input-Brick, also ein Brick, der Daten einliest und in das Kommunikationsnetz der Titan Flow Engine hineinsendet. In diesem Fall liest der Brick XML-Dateien, welche Log-Einträge des VIP-Systems des Druckzentrums der Kieler Nachrichten beinhalten. Die einzelnen Log-Einträge beinhalten Informationen über den aktuellen Zustand einer Druckmaschine. Listing 5.1 zeigt einen Eintrag, der zu Produktionsbeginn gespeichert wird. Ein VIP-Log-Eintrag kann verschiedene Felder besitzen. Dies ist für unsere Arbeit von Bedeutung, da Felder für *GrossCopies*, *NetCopies* sowie *Speed* im Converter-Brick für verschiedene Berechnungen benötigt werden (siehe folgenden Abschnitt). Der Input-Brick transformiert jede *row* der VIP-Log Datei in eine UjoMap und sendet diese an den nächsten Brick. Der Input-Brick löscht nach dem Lesen hinterlegte Log-Dateien an ihrem Ursprungsort und fragt einen konfigurierbaren Speicherort für Log-Dateien regelmäßig ab.

[1] GrossCopies sind alle gedruckten Zeitungen, inklusive fehlerhafter Exemplare
[2] NetCopies sind die fehlerfreien Exemplare

CSV-Input-Brick

Neben dem oben vorgestellten Brick für die beschriebenen Log-Dateien haben wir einen weiteren Brick entwickelt, welcher Einträge aus einer CSV-Datei liest und analog zu dem XML-Input-Brick in einem Ujo-Format an den Converter-Brick sendet. Die Verwendung beider Formate ist darin begründet, dass uns unterschiedliche Ereignisprotokolldateien desselben Systems vorliegen. Die Log-Daten, die im XML-Format abgespeichert werden, sind Informationen eines echten Produktionsbetriebes. In der mit diesem Brick verwendeten CSV-Datei ist eine einzelne Produktion exemplarisch eingetragen, welche allerdings mehr Log-Einträge zu dieser Produktion aufweist als die XML-Log Files des VIP-Systems.

Converter-Brick

Der Converter-Brick konvertiert die eingehenden VIP-Records in *vip-log-records*. Ein vip-log-record beinhaltet Angaben über die durchschnittliche Produktionsgeschwindigkeit für GrossCopies und NetCopies, die ID des Druckauftrages, einen Timestamp und die eingesetzte Druckmaschine FALZ 1, FALZ 2, FALZ 3 sowie die Rotationsgeschwindigkeit. Listing 5.2 zeigt den Aufbau eines *vip-log-records*.

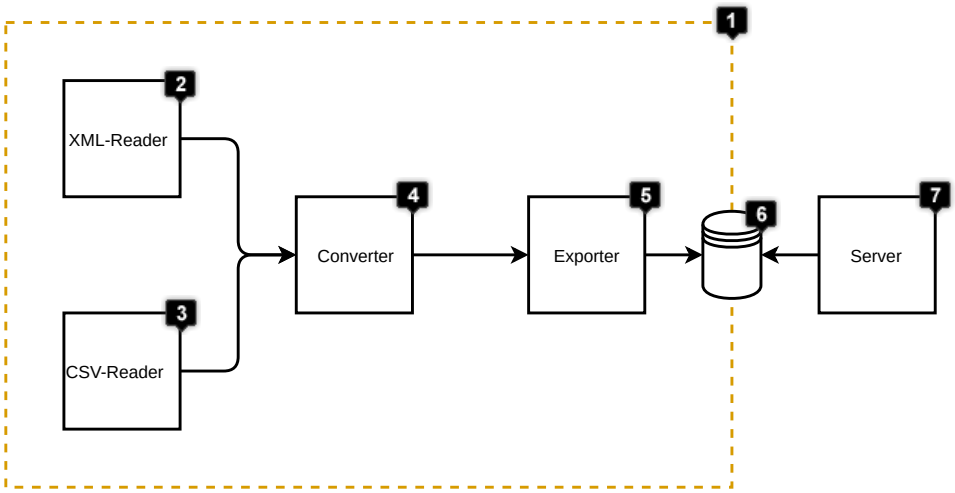

Abbildung 5.1. Grafische Darstellung des Flows zum Einlesen von VIP-Records. Die Bricks ② - ⑤ innerhalb des gestrichelten Rechteckes ① werden durch die Flow Engine ausgeführt. Brick ② und ③ lesen Daten und senden diese an den Converter-Brick ④. Dieser konvertiert die Daten in das vip-log-record Format und sendet die Daten an den Exporter-Brick ⑤, welcher die Records in der Datenbank ⑥ zwischenspeichert. Das Server-Skript ⑦ befindet sich außerhalb des Einflusses der Flow Engine und greift auf die gemeinsam genutzte Datenbank zu ⑥.

```
1   <row>
2     <PublicationShortLabel>STT</PublicationShortLabel>
3     <EditionGroupShortLabel>STT</EditionGroupShortLabel>
4     <PartProductShortLabel>HP</PartProductShortLabel>
5     <DateOfIssue>2019-01-22T00:00:00</DateOfIssue>
6     <PressName>FW03</PressName>
7     <SetDatetime>2019-01-21T22:31:34</SetDatetime>
8     <DelDatetime>2019-01-21T23:10:35</DelDatetime>
9     <MessageClassID>3</MessageClassID>
10    <MessageNumber>9809</MessageNumber>
11    <MessageText>Produktionsbeginn</MessageText>
12    <MessageLocation>FALZ 3 </MessageLocation>
13    <GrossCopies>0</GrossCopies>
14    <NetCopies>0</NetCopies>
15    <Speed>0</Speed>
16  </row>
```

Listing 5.1. Exemplarischer Eintrag der XML-Ereignisprotokolldatei

Berechnung Die Berechnung der durchschnittlichen Druckgeschwindigkeit wird pro Druckauftrag in der Regel aus zwei Messwerten berechnet, dem jeweiligen Anfangsrecord und dem entsprechenden Endrecord des Zeitintervalls des Druckauftrages. Das liegt daran, dass zu Beginn eines Druckvorgangs ein Log-Eintrag erstellt wird, welcher unter anderem den Startzeitpunkt und die bisher gedruckten GrossCopies und NetCopies abspeichert. Listing 5.1 zeigt einen Log-Eintrag zum Startzeitpunkt eines Druckauftrages. Zum Ende eines Druckauftrages existiert in unseren Testdaten ein weiterer Eintrag, dieser beinhaltet neben dem zugehörigen Endzeitpunkt die Anzahl der gedruckten GrossCopies und NetCopies. Außerdem beinhalten beide Log-Einträge das Feld *MessageLocation*, welches die Druckmaschine referenziert.

Beim Eintreffen des ersten Logs eines Druckauftrages wird der eingehende VIP-Record gespeichert. Sobald ein weiterer Record eingeht, der über die Felder GrossCopies und NetCopies verfügt, wird das Timedelta zwischen den Records sowie die Differenz zwischen gespeicherter Anzahl und eingehender Anzahl GrossCopies und NetCopies berechnet. Aus diesen Informationen berechnen wir die durchschnittliche Druckgeschwindigkeit von GrossCopies und NetCopies pro Stunde. Unser Converter-Brick erzeugt nun zwei *vip-log-records*, eines für den Zeitpunkt des Startrecords, eines für den Zeitpunkt des Endrecords. Beiden Records werden die Druckgeschwindigkeiten zugewiesen, die dem Durchschnitt der beiden Punkte entsprechen.

Wenn mehrere Records existieren, die die entsprechenden Informationen beinhalten, werden analog zu dem vorgestellten Verfahren Durchschnittswerte zwischen zwei aufeinanderfolgenden Records berechnet. In diesem Fall wird jedoch lediglich ein weiterer

```
1     Record = UjoVariantMap()
2     Record.set_value("identifier", identifier, UjoVariantStringC)
3     Record.set_value("timestamp", timestamp, UjoVariantFloat64)
4     Record.set_value("grossCopiesPerHour", gross_copy_per_hour,UjoVariantFloat64)
5     Record.set_value("netCopiesPerHour", net_copy_per_hour,UjoVariantFloat64)
6     Record.set_value("messageLocation", location_falz, UjoVariantStringC)
7     Record.set_value("speed", speed, UjoVariantFloat64)
```

Listing 5.2. Aufbau eines vip-log-record als UjoMap innerhalb des Converter-Bricks.

Record erzeugt, der dem Zeitpunkt des jeweiligen Messpunktes entspricht.

Neben den durchschnittlichen Druckgeschwindigkeiten, die wir mit netCopiesPerHour und grossCopiesPerHour bezeichnen, existieren in einigen Log-Einträgen Messwerte der Rotationsgeschwindigkeit. Diese Messwerte speichern wir ohne weitere Berechnung in einer weiteren Zeitreihe, die als *speed* bezeichnet ist, ab.

5.2.2. Testdaten

Der uns vorliegende XML-Datensatz beinhaltet Ereignisprotokoll-Einträge, welche unterschiedliche Informationen dokumentieren. So existieren lediglich für zwei Druckaufträge zwei Ereignisprotokoll-Einträge[3], welche Felder für GrossCopies und NetCopies beinhalten. Listing 5.1 zeigt in Zeile 13 und 14 die entsprechenden Felder für GrossCopies und NetCopies. Zum Testen haben wir den Datensatz insofern manipuliert, dass es für jeden Druckauftrag zwei Log-Einträge gibt, welche uns Auskunft über bisher gedruckte GrossCopies und NetCopies geben. Um die eingegebenen Daten möglichst realitätsnah zu wählen, haben wir die Anzahl der NetCopies am Ende eines Druckauftrages auf den *Sollwert* des entsprechenden Druckauftrages gesetzt. Dies war möglich, da in dem vorliegenden Datensatz für jeden Druckauftrag zu Beginn ein *Sollwert* gesetzt wurde, der die Anzahl zu druckender Exemplare festlegt. Das Feld für GrossCopies haben wir nach dem Zufallsprinzip gesetzt. Insgesamt haben wir bei der Generierung der Testdaten beachtet, dass der Wert für GrossCopies immer größer als der Wert für NetCopies ist, und umso größer der Eintrag für NetCopies ist, desto höher ist auch die Abweichung zum Wert des Feldes für GrossCopies. Vereinzelt waren in den Einträgen der Ereignisprotokolldatei Informationen über die aktuelle Geschwindigkeit zu finden. Auch diese Daten haben wir bei weiteren Records hinzugefügt, um eine möglichst variable Schnittstelle zu den Ereignisprotokolldateien anzubieten.

Setzen des Timestamps für Testdaten Der vorliegende Datensatz enthält Log-Einträge über Druckaufträge aus dem Januar 2019, die Timestamps der Einträge dieser Log-Datei

[3] Anfangs- und Endzeitpunkt

sind zu dem Zeitpunkt dieser Arbeit etwa 6 Monate alt.

Das Monitoringsystem Prometheus speichert ausschließlich Metriken von Zeitreihen, die in einem Intervall von +/- 60 Minuten zum Zeitpunkt des Scrape-Vorgangs liegen. Diese Einschränkung seitens Prometheus lässt es nicht zu die Log-Einträge unserer Testdatei durch den Scrape-Vorgang zu speichern. Aus diesem Grund versetzen wir den Timestamp, sodass er in dem benötigten Intervall liegt.

Für das Verschieben der Timestamps speichern wir zunächst den Timestamp des ersten Records einer Druckmaschine und verschieben den Zeitpunkt dieses Records auf den aktuellen Zeitpunkt. Für einen weiteren Record dieser Druckmaschine berechnen wir die Differenz zu dem gespeicherten Record. Diese Differenz skalieren wir mit einem konfigurierbaren Stauch- oder Streckfaktor $scale\_factor$ und addieren das Ergebnis auf den aktuellen Zeitpunkt. Dieses Vorgehen verschiebt die Timestamps in die Zukunft, die relativen Zeitdifferenzen der Records einer Druckmaschine bleiben jedoch bestehen. Mathematisch ausgedrückt, gilt folgende Relation:

$t_{new} = t_{current} + (t_{record} - t_{first}) \cdot scale\_factor$ wobei:
t_{new} = Der Wert, auf den Timestamp des Records gesetzt wird.
$t_{current}$ = Der aktuelle Zeitpunkt zur Ausführung.
t_{record} = Der originale Timestamp des Records.
t_{first} = Der Timestamp des ersten Records dieser Druckmaschine.
$scale\_factor$ = Skalierungsfaktor zum Testen, damit die Dauer eines Druckauftrages nicht Stunden, sondern Minuten entspricht.

Keine Labels in Zeitreihen Unser Testdatensatz besteht aus Log-Einträgen für drei unterschiedliche Druckmaschinen, für jede Druckmaschine werden drei Zeitreihen generiert, eine für netCopiesPerHour, eine für grossCopiesPerHour und eine für die Rotationsgeschwindigkeit. Die Zeitreihen sind nicht gelabelt, das heißt, die Information, welcher Druckauftrag zu welchem Zeitpunkt ausgeführt wurde, geht in der aktuellen Implementierung verloren.

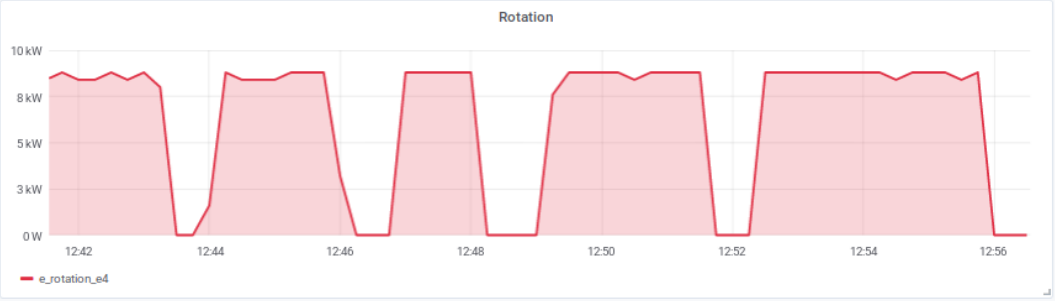

Abbildung 5.2. Visualisierung der Stromverbrauchsdaten einer Druckmaschine mit der Bezeichnung e_rotation_e4. Die Abbildung zeigt den Stromverbrauch gestaucht an.

Gelabelte Prometheus Zeitreihen sind Zeitreihen, welche über zwei weitere Felder für $Label_{Name}$ und $Label_{Value}$ verfügen. Durch dieses Vorgehen können weitere Informationen zu einer Zeitreihe gespeichert werden, außerdem ist es möglich, beim Abfrageprozess nach einzelnen $Label_{Values}$ zu filtern. Unsere Entscheidung gegen Labels ist zum einen darin begründet, dass pro Druckmaschine lediglich ein Druckauftrag je Zeitpunkt ausgeführt werden kann. Das bedeutet, dass wir zu keinem Zeitpunkt Metriken von zwei unterschiedlichen Druckaufträgen derselben Druckmaschine abspeichern. Zum anderen würden wir für jeden neuen Druckauftrag einen weiteren Wert für das Feld $Label_{Value}$ eingeben, sodass wir langfristig eine sehr hohe Anzahl an Werten für das Feld $Label_{Value}$ abspeichern würden. Dieses Vorgehen würde dazu führen, dass die Speicherauslastung der mehrdimensionalen Datenbank stark ansteigt und wird von Prometheus nicht empfohlen.
Der Anstieg der Speicherressourcen ist darauf zurückzuführen, dass Prometheus für jedes $Label_{Value}$ intern eine neue Zeitreihe abspeichert.

5.3. Visualisierung von Monitoringdaten

Die oben beschriebenen Log-Einträge der Druckmaschinen sollen mit Messungen des Stromverbrauchs der Maschinen verglichen und ihre Korrelation aufgezeigt werden. Ziel dieser Korrelation ist es, den Stromverbrauch und die Produktionsgeschwindigkeit der Druckmaschinen gleichzeitig zu visualisieren. Dazu haben wir das in Abschnitt 4.3.2 vorgestellte Verfahren genutzt und eine CSV-Datei eingelesen, welche Messwerte der Druckmaschinen in Form von ActivePowerRecords beinhaltet.

Wir haben die Visualisierung mithilfe eines Graph Panels (siehe Abschnitt 4.4.1) vorgenommen. Abbildung 5.2 zeigt einen Ausschnitt der Stromverbrauchsdaten. In Abschnitt 4.4 stellen wir weitere Formen der Visualisierung vor. Wir haben die Zeitabstände zwischen einzelnen Messungen stark verringert, die relativen Abstände sind jedoch, wie in Abschnitt 5.2.2 erläutert, identisch geblieben.

Neben der Darstellung des Stromverbrauchs haben wir die durchschnittliche Druckauslastung visualisiert. Dazu haben wir pro Zeitreihe einen Wert für die durchschnittliche Anzahl produzierter Brutto- und Nettokopien pro Stunde erstellt. Abbildung 5.3 zeigt beispielhaft die Visualisierung unserer Testdaten für die *Falz 1*. Grundsätzlich bestand an dieser Stelle die Herausforderung, aus wenigen Log-Einträgen eine Zeitreihe zu generieren. Zur Lösung dieses Problems haben wir pro Druckauftrag den Anfangs- und den Endpunkt markiert. Die Zeit zwischen Anfangs- und Endrecord repräsentiert die Druckdauer der Maschine. Während der Zeitperiode zwischen zwei Messpunkten gehen wir von einer durchschnittlichen Druckgeschwindigkeit aus.

Prometheus speichert für jeden Eintrag einer Zeitreihe den angegebenen Wert für fünf Minuten[4], wenn es keinen aktuelleren Wert abspeichern kann. Nach Ablauf dieser fünf

[4]Fünf Minuten ist die Standardeinstellung. Wir haben diese nicht verändert, da andernfalls nicht erkenntlich wird, zu welchem Zeipunkt ein bestimmer Wert (nicht) vorliegt (dies ist insbesondere für Metriken der Stromverbrauchsdaten wichtig).

Minuten existiert für die Zeitreihe kein Wert, außer es wurde ein aktuellerer Eintrag gespeichert. Nach dem Ende einer Produktion und vor dem Anfang des nächsten Druckauftrages nehmen wir an, dass die Druckmaschine stillsteht. Aus diesem Grund haben wir künstlich zwei zusätzliche Punkte erzeugt. Zum einen wird vor dem Start jedes Druckauftrages ein vip-log-record erzeugt, bei dem die Produktionsgeschwindigkeit bei 0 liegt, zum anderen wird nach jedem Druckauftrag ein vip-log-record erzeugt, dessen Werte ebenfalls die Produktionsgeschwindigkeit auf 0 setzen (Leerlauf-Records).

Für die Darstellung der Zeitreihen für die VIP-Monitoringdaten bietet Grafana unterschiedliche Möglichkeiten. Zur Visualisierung dieser Log-Einträge haben wir Grafana derart konfiguriert, dass Messpunkte durch eine grafische Linie miteinadner verbunden werden, auch wenn dazwischen *Null*-Werte[5] liegen. Diese Art der Visualisierung ermöglicht, dass wir (trotz weniger Messpunkte) in Grafana eine möglichst vollständige Zeitreihe in Form einer Kurve eines Graph Panels visualisieren können. Der Nachteil dieser Form der bildlichen Darstellung besteht darin, dass die Kurve des angezeigten Graphens nicht kontinuierlich aufgebaut wird, sondern vor der Zeichnung eines neuen Kurvensegments stets auf den nächsten Messpunkt gewartet wird.

[5]Null-Werte treten in einer Zeitreihe auf, wenn zu dem abgefragten Zeitpunkt kein Wert vorliegt. Fünf Minuten nach dem letzten Messpunkt ist das Ergebnis einer Prometheus-Abfrage *Null*. Vor Ablauf der fünf Minuten wird der letzte Messpunkt ausgegeben.

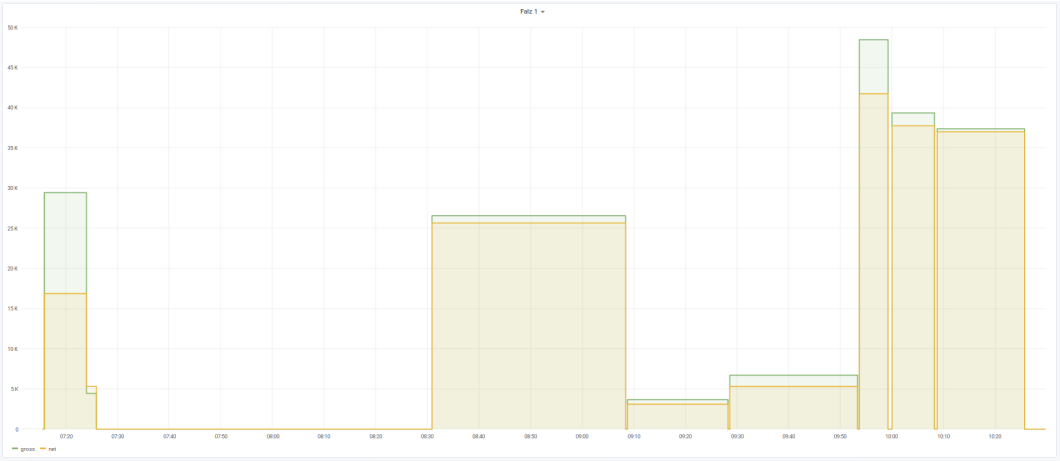

Abbildung 5.3. Visualisierung der Druckgeschwindigkeit in Form von netCopies- und grossCopiesPerHours von Druckmaschine 1. Die grüne Kurve repräsentiert dabei die Anzahl GrossCopies, die gelbe Kurve NetCopies. An dieser Stelle wird der VIP-XML-Datensatz verwendet.

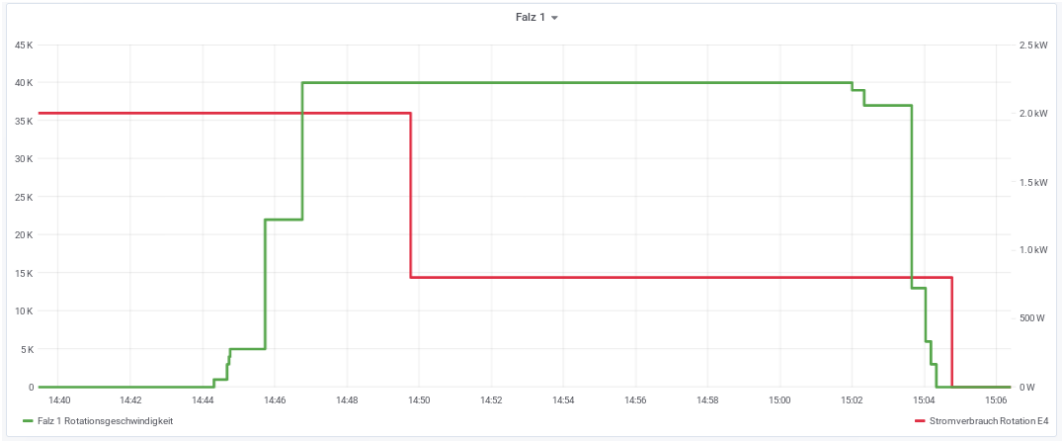

Abbildung 5.4. Korrelation zwischen der Geschwindigkeit und dem Stromverbrauch einer Druckmaschine. Die Rotationsgeschwindigkeit ist in Grün abgebildet und entsprechend der linken Ordinate skaliert. Der Stromverbrauch entspricht der roten Kurve und orientiert sich an der Skalierung der rechten Ordinate. Die Daten zeigen einen Ausschnitt von etwa 40 Minuten aus dem laufenden Druckbetrieb.

5.3.1. Versetzen der Timestamps

Unser Testdatensatz beinhaltet Einträge für Druckaufträge derart, dass es vorkommen kann, dass ein Druckauftrag in derselben Sekunde endet, in der der nächste beginnt. Wir haben oben beschrieben, dass wir zu Beginn und zum Ende eines Druckauftrages vip-log-records erzeugen, die die Druckgeschwindigkeiten auf 0 setzen (Leerlauf-Records). Um einen Konflikt zwischen Einträgen für Produktionsende, Produktionsbeginn und Leerlauf-Records zu vermeiden, werden die Timestamps entsprechend gesetzt. Der erste Leerlauf-Record eines Druckauftrages beginnt 5 Sekunden vor dem jeweiligen Produktionsbeginn. Der vip-log-record der den Produktionsbeginn kennzeichnet wird 15 Sekunden in die Zukunft gesetzt. Der Leerlauf-Record, der nach dem VIP-Eintrag Produktionsende gesendet wird, besitzt einen Timestamp, der 5 Sekunden nach dem des Produktionsendes liegt. Durch dieses Vorgehen können Ungenauigkeiten zu Beginn und zum Ende eines Druckauftrages entstehen, die Ungenauigkeiten sind jedoch im Sekundenbereich (max. 20 Sekunden), die Dauer eines Druckauftrags liegt hingegen im Bereich von Stunden.

5.3.2. Korrelat Druckgeschwindigkeit und Stromverbrauch

Wie in Abschnitt 4.3 beschrieben, steht uns neben dem in Abbildung 5.3 dargestellten XML-Datensatz eine weitere Ereignisprotokolldatei zur Verfügung. Die zweite Datei enthält detaillierte Informationen zu einem einzigen Druckauftrag. Zu diesem Druckauftrag liegen uns ebenfalls Messwerte des Stromverbrauchs (i. F. v. AggregatedActivePowerRecords

und ActivePowerRecords) vor. Abbildung 5.4 zeigt die Korrelation beider Messwerte. An dieser Stelle nutzen wir eine Kombination der in Abschnitt 4.3 vorgestellten Kommunikationskanäle. Die Messwerte der Geschwindigkeit werden direkt über die Flow Engine für Prometheus bereitgestellt. Abbildung 5.1 zeigt schematisch den Kommunikationsweg. Die Messwerte des Stromverbrauchs hingegen haben wir, wie in Abschnitt 4.3.2 beschrieben, über die Flow Engine an das Titan Control Center übermittelt und anschließend, wie in Abschnitt 4.3 dargelegt, über das Nachrichtensystem Kafka für Prometheus bereitgestellt.

5.3.3. Datenfluss

Wir haben neben den Ausführungen in diesem Kapitel in Kapitel 3 und in Kapitel 4 verschiedene technische Blickwinkel auf unseren Ansatz, unsere Implementierung und diese Fallstudie gezeigt. Für die Anzeige der Korrelation in Abbildung 5.4 haben wir verschiedenste Komponenten unseres Ansatzes eingesetzt. Abbildung 5.5 zeigt schematisch den Datenfluss des Einlesens, Verarbeitens, Speicherns und Visualisierens. Die Grafik veranschaulicht das Einlesen von Daten über zwei Bricks, welche mithilfe von zwei Flows verarbeitet werden. Der eine Flow stellt die Daten direkt für Prometheus bereit, der andere sendet Metriken über Kafka an das Control Center. Über den Kafka Exporter werden die Daten anschließend für Prometheus bereitgestellt. Das Grafana-Frontend greift anschließend auf Prometheus zwecks Datenabfrage zu. Die Abbildung zeigt außerdem den Datenfluss der in Abbildung 5.6 dargestellten Daten. Dafür wird über den REST Adapter auf den History Service des Control Centers zugegriffen. In diesem Fall haben wir die Monitoringdaten, wie in Abschnitt 5.4 beschrieben, manuell importiert. Abbildung 4.2 und Abbildung 4.8 visualisieren Messwerte, die über Mechanismen innerhalb des Control Centers erhoben wurden.

5.4. Dashboard einer industriellen Zeitungsdruckerei

Die in Abschnitt 4.4 abgebildeten Grafiken zeigen generierte Werte der Demonstrations-Version des Titan Control Centers. Demgegenüber stellen wir in diesem Abschnitt ein Dashboard für die Messwerte des Druckzentrums der Kieler Nachrichten vor. Abbildung 5.6 zeigt einen Screenshot dieses Dashboards. Dazu nutzen wir die manuell in die Cassandra-Datenbank importierten Monitoringdaten des Druckzentrums. Die importierten Daten bestehen aus Messungen diverser normaler Sensoren, die verschiedenen Messpunkten aus der Produktion und Verwaltung entsprechen. Zusätzlich dazu existiert ein aggregierter Sensor, welcher den Druckverbrauch des gesamten Druckzentrums wiedergibt.

An dieser Stelle ist es möglich weitere Dashboards zu erstellen, um weitere Daten korreliert anzuzeigen oder die Darstellungsformen zu modifizieren. Dazu könnten beispielsweise die in Abschnitt 4.4.2 vorgestellten Grafana Variables eingesetzt werden, um die interaktive Exploration der Messwerte zu ermöglichen.

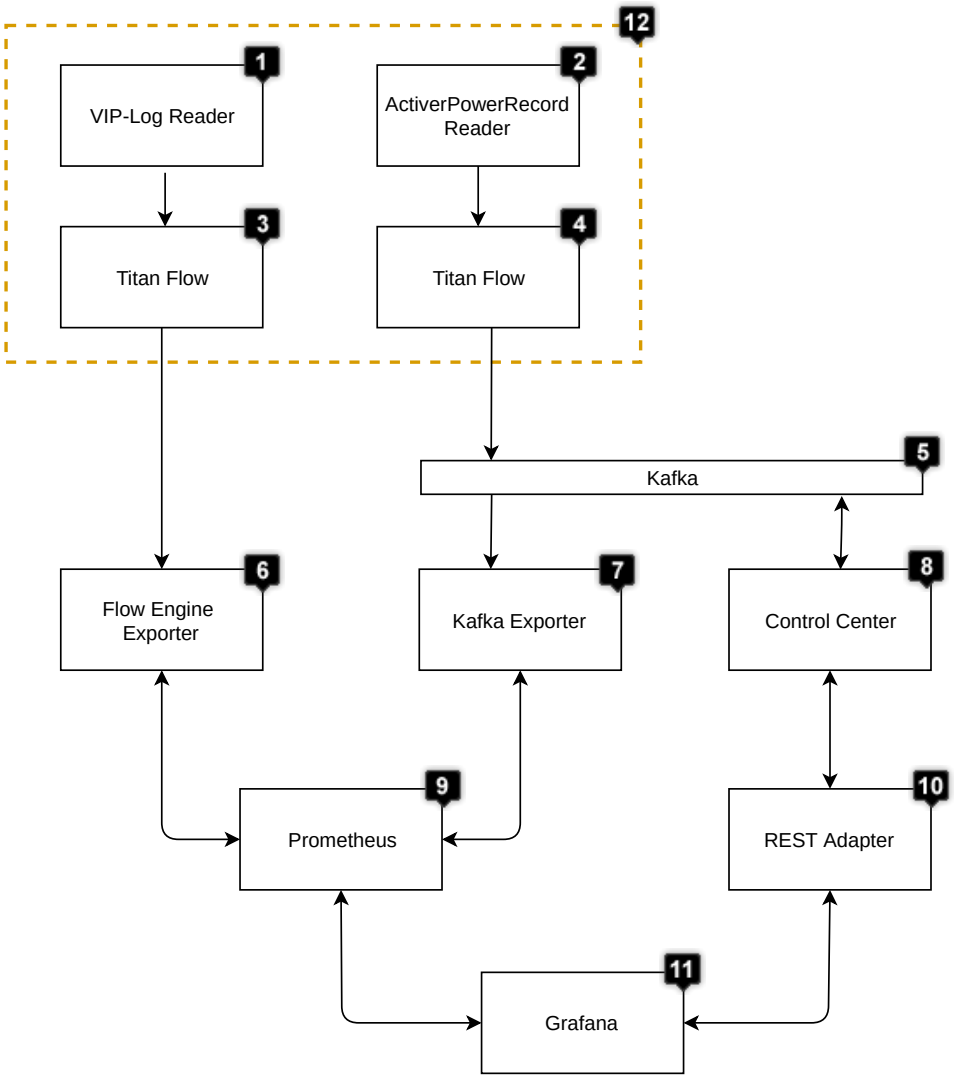

Abbildung 5.5. Schematische Darstellung des Datenflusses vom Einlesen bis zur Visualisierung der in Abbildung 4.2, Abbildung 4.8, Abbildung 5.4 und Abbildung 5.6 dargestellten Daten. Die Pfeile kennzeichnen die Richtung des Datenflusses. Analog zur Notation in Abbildung 5.1 zeigt das gestrichelte Rechteck den Wirkungsbereich der Flow Engine (12). Die Rechtecke stellen verschiedene Komponenten dar, die wir in unserer Arbeit genutzt oder entwickelt haben, die Größe eines Rechteckes ist nicht repräsentativ für die Komplexität der assoziierten Komponente. Komponente (1), (3) und (6) stellen wir in Abbildung 5.1 vor, Komponenten (2) und (4) erläutern wir in Abschnitt 4.3.2. Komponente (7) erläutern wir in Abschnitt 4.2.1. Komponente (10) stellen wir in Abschnitt 4.1 vor. Die Komponenten (5), (8), (9), (11) und (12) werden in Kapitel 2 eingeführt.

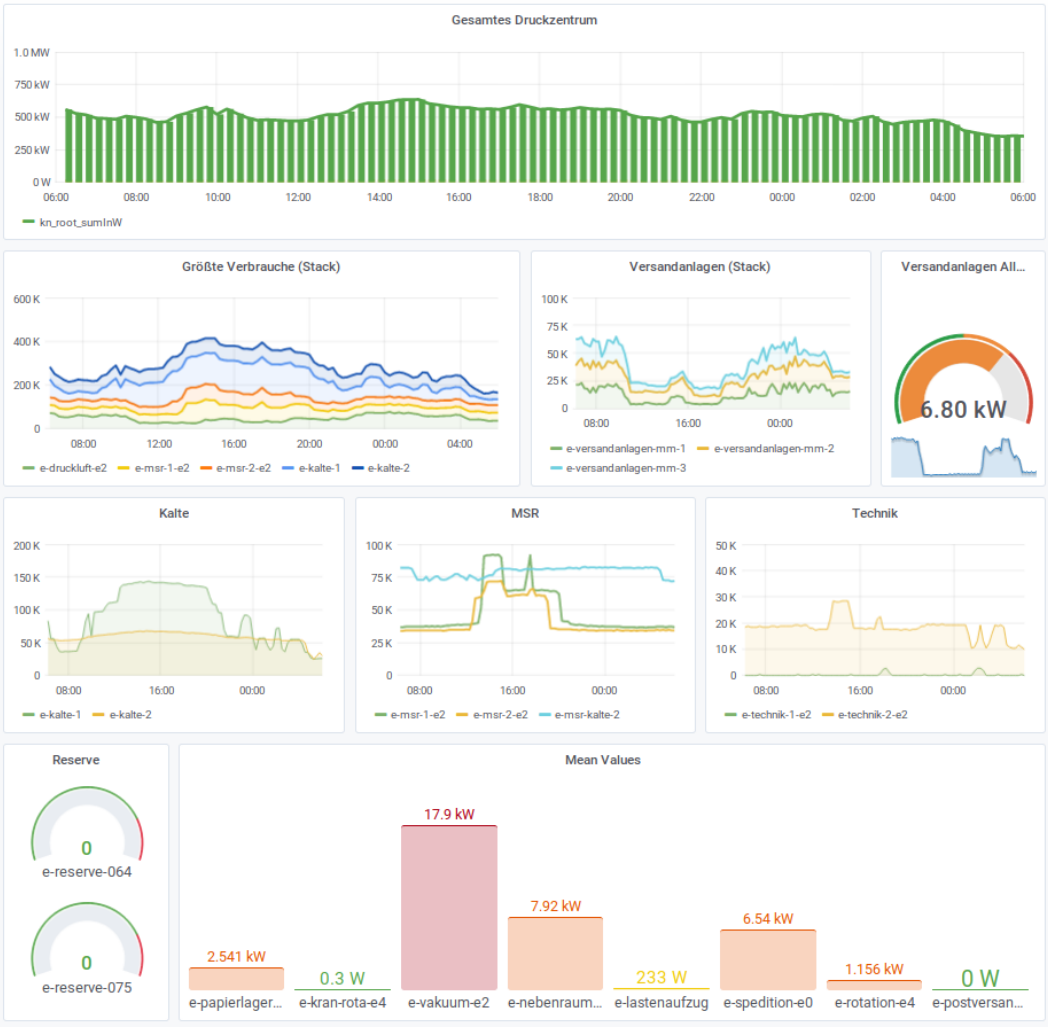

Abbildung 5.6. Dashboard Oberfläche mit Messwerten des Druckzentrums der Kieler Nachrichten. Das Dashboard zeigt in diesem Fall Messwerte des Zeitraums vom 01.08.2018 06:00:00 Uhr bis zum 02.08.2018 06:00:00 Uhr. Oben wird der Gesamtverbrauch abgebildet, darunter zeigen wir Graphen und Korrelationen verschiedener Verbraucher. Die Gauges im unteren Teil des Dashboards zeigen Verbraucher, deren Energieverbrauch vermeintlich weniger Relevanz besitzt.

Kapitel 6

Evaluation

In diesem Kapitel stellen wir die Evaluation dieser Arbeit vor. In Abschnitt 6.1 erfolgt eine Gegenüberstellung der unterschiedlichen Kommunikationswege, mit denen wir Grafana und die Titan Plattform miteinander verbunden haben. In Abschnitt 6.2 vergleichen wir, basierend auf Few [2006] und Yau [2014], verschiedene Aspekte unseres Dashboards mit dem derzeit eingesetzten Frontend des Titan Control Centers. In Tabelle 6.1, Tabelle 6.2 und Tabelle 6.3 formulieren wir die wichtigsten Punkte, die aus den jeweiligen Vergleichen resultieren. In Abschnitt 6.3 diskutieren wir diese Ergebnisse abschließend.

6.1. Gegenüberstellung der Kommunikationswege

Wir haben je zwei Wege aufgezeigt, mit denen sowohl das Titan Control Center mit Grafana verbunden werden als auch die Flow Engine mit Prometheus kommunizieren kann. Im Folgenden vergleichen wir zunächst die Kommunikationskanäle, mit denen das Titan Control Center angebunden ist, und stellen darauffolgend die unterschiedlichen Möglichkeiten der Anbindung der Flow Engine gegenüber.

6.1.1. Vergleich von REST Adapter und Prometheus

Wir haben in Kapitel 4 verschiedene Wege aufgezeigt, wie Verbindungen zwischen einer Monitoring Plattform (hier Titan) und einem Frontend Analyse- und Visualisierungsframework (hier Grafana) aufgebaut werden können. Dafür haben wir die Kommunikation über HTTP/REST aufgebaut und beispielhaft Prometheus als Zwischenglied integriert. Beide Kommunikationswege haben spezifische Vor- und Nachteile. Die Verbindung über HTTP/REST hat den Vorteil, dass kein zusätzliches System Ressourcen verbraucht, ist aber dafür darauf angewiesen, dass zu jedem Zeitpunkt sämtliche Microservices verfügbar sind und Metrikdaten abgefragt werden können. Die Integration von Prometheus hingegen erfordert das Aufbringen von zusätzlichen Ressourcen. Dafür ermöglicht Prometheus das (Zwischen-) Speichern von Metrikdaten und hält diese abrufbar bereit. Dadurch können Daten abgerufen werden, obwohl die Microservices derzeit nicht aktiv sind. Prometheus kann also als zusätzliche Isolationsschicht betrachtet werden. Ähnliches wäre mit dem

REST Adapter nur bedingt in Form von Caching[1] möglich.

Der Kommunikationsweg über REST ist auf die definierten Schnittstellen angewiesen und ermöglicht es damit nicht, flexible Abfragen zu Gestalten. Prometheus hingegen nutzt bei der Abfrage von Metriken die eigens entwickelte Sprache PromQL. Mit PromQL ist es möglich, Zeitreihen für einen frei wählbaren Zeitraum abzufragen. Demgegenüber ermöglicht unsere Implementierung des REST Adapters dieses Vorgehen derzeit noch nicht, da seitens des History Services ein solches Zeitintervall nicht vorgesehen ist. Hier werden bei der Abfrage alter Metriken alle Records dieser Zeitreihe mitgesendet, die neuer als der abgefragte Zeitraum sind. Das führt in der verwendeten Cassandra-Datenbank des History Services zu enorm großen Abfrageergebnissen, welche zu einem hohen Ressourcenverbrauch innerhalb des History Services und des Netzwerks führen. Wir haben dieses Problem in Abschnitt 8.2 genauer ausgeführt. Für Abfragen, die weit in der Vergangenheit liegen, erscheint die Nutzung von Prometheus geeigneter zu sein. Für Abfragen, die den Schnittstellen der Microservices entsprechen und nahe dem aktuellen Zeitpunkt liegen, kann die Kommunikation via REST ebenfalls als geeignet angesehen werden. Tabelle 6.1 zeigt wichtige Punkte dieses Vergleichs.

6.1.2. Vergleich der Anbindungswege der Flow Engine an Prometheus

Wir haben die Flow Engine und Prometheus über zwei unterschiedliche Kommunikationswege verbunden. Auf der einen Seite haben wir, wie in Abschnitt 4.3.1 beschrieben, ein Python-Skript erstellt, welches einen direkten Scrape-Punkt für Prometheus zur Verfügung stellt. Auf der anderen Seite haben wir einen bereits vorhandenen Brick genutzt, welcher die Anbindung eines Flows an das Titan Control Center ermöglicht. Diesen Weg haben wir in Abschnitt 4.3.2 erläutert. Für diese Art der Verbindung werden sowohl durch die Flow Engine als auch durch das Control Center dieselben Kafka Topics genutzt. Dadurch findet die Kommunikation geregelt über festgelegte Kafka Topics statt und es werden sämtliche Microservices angebunden, die das entsprechende Topic nutzen.

Die erste Variante bietet die Möglichkeit, dass ein Flow direkt mit Prometheus kommunizieren kann, ohne dass das Control Center ausgeführt werden muss. Andererseits ist es nicht mehr möglich, Verarbeitungsschritte innerhalb des Control Centers durchzuführen. Dies könnte beispielsweise die Nutzung von Microservices sein, um auf Statistiken zuzugreifen oder sie zu erstellen. Szenarien, in denen dieser Umstand sinnvoll eingesetzt werden könnte, wären beispielsweise die Entwicklung und das Testen von Bricks und Flows. Das liegt daran, dass Prometheus direkt die Möglichkeit einer einfachen Visualisierung in Form von Graphen anbietet, ohne zusätzliche Komponenten wie Grafana oder das Control Center zu benötigen. Weiter sind Anwendungsfälle, die nur dazu dienen, Mess- oder Log-Formate eines speziellen Geräts (einmalig) über einen Flow beispielsweise in Grafana zu visualisieren, denkbar. Diese Szenarien können insbesondere dann sinnvoll

[1] Beim Caching werden Informationen zwischengespeichert, um folgende Anfragen schneller bearbeiten zu können.

eingesetzt werden, wenn Messwerte diverser unterschiedlicher Geräte ohne weitere Berechnungsschritte bildlich dargestellt werden sollen. In einem solchen Fall muss abgewogen werden, wie der Mehraufwand der Implementierung für die Anbindung an das Control Center und die entsprechender Microservices und Schnittstellen zu Prometheus oder anderen Datenbanksystemen im Verhältnis zum Nutzen steht. Die direkte Anbindung an Prometheus kann auch dadurch motiviert werden, dass Prometheus durch die Sprache PromQL einige Möglichkeiten bietet, um Daten zu verarbeiten und besondere Messwerte zu filtern.

Zusammenfassend kann man sagen, dass die direkte Anbindung der Flow Engine an Prometheus ressourcenschonend ist, einen geringeren Entwicklungsaufwand benötigt und eine hohe Unabhängigkeit ermöglicht. Ein Nachteil ist dabei jedoch der geringere Grad der Integration von zusätzlichen Microservices. Wenn komplexere Aufgaben bewältigt werden müssen, kann demnach die indirekte Anbindung über das Control Center dennoch Vorteile bieten. Die Unterschiede der beiden Wege sind in Tabelle 6.2 zusammengefasst.

6.1.3. Eigenschaften der Nutzung von Prometheus

Durch die Nutzung von Prometheus ist es möglich, über die Abfragesprache PromQL verschiedene Operationen auf Metriken durchzuführen. Neben Operationen auf Zeitreihen kann PromQL auf die Labels verschiedener Metriken zugreifen und diese ebenfalls für Berechnungsschritte nutzen. Dadurch ist es ohne weiteres Zutun Titans möglich, den Minimum- oder Durchschnittswert einer Metrik zu ermitteln oder verschiedene Zeitreihen zu kombinieren. Auch ist es über die Abfragesprache möglich, Aggregationen ähnlich zu denen zu bilden, die in Titan als AggregatedActivePowerRecord bezeichnet und von dem History Service vorgenommen werden.

Wir haben uns in Kapitel 4 dafür entschieden, AggregatedActivePowerRecord nicht als gelabelte Serie darzustellen, sondern fünf getrennte Zeitreihen erstellt. Zum jetzigen Zeitpunkt würden wir eine gelabelte Zeitreihe für jeden aggregierten Sensor erstellen. Dafür würden wir ein Präfix, wie *titan_aggregated* wählen, sodass ein Metrikname beispielsweise *titan_aggregated_root* heißen würde, und für das Label $Label_{Name} = type$, die $Label_{Values}$ minInW, maxInW, sumInW, averageInW, count besitzen. Ähnlich zu diesem Vorgehen würden wir ActivePowerRecords den Präfix *titan_machine* hinzufügen, um den Unterschied zu einem aggregierten Sensor kenntlich zu machen. Der Grund für die Aktualisierung unseres Ansatzes ist, dass über PromQL viele Operationen auf Labels möglich sind und wir davon ausgehen, dass die Nutzbarkeit verbessert wird.

Im Gegensatz zu herkömmlichen Datenbanksystemen, wie beispielsweise der Cassandra-Datenbank, speichert das auf Monitoring ausgelegte Prometheussystem Daten lediglich für einen begrenzten Zeitraum, die Default-Einstellung beträgt hierfür 15 Tage. Anschließend werden die ältesten Daten mit neuen überschrieben. Wir haben Prometheus so konfiguriert, dass Daten für einen Zeitraum von 10 Jahren gespeichert werden. Um Daten aperiodisch, für *immer*, zu speichern, kann auf externe Speichersysteme zugegriffen werden.

6.2. Gegenüberstellung des Grafana Dashboards und des Frontends des Control Centers

Im Folgenden vergleichen wir das derzeit bestehende Frontend des Titan Control Centers[2] mit der von uns entwickelten Dashboardlösung (siehe Abschnitt 4.4). Dabei gehen wir zunächst auf die unterschiedlichen Visualisierungsformen ein. Anschließend zeigen wir Unterschiede zwischen einer High-Level Plattform wie Grafana und einem Frontend Framework wie Vue.js[3] auf. Wir haben die wesentlichen Unterschiede in Tabelle 6.3 aufgezeigt.

6.2.1. Gegenüberstellung der eingesetzten Visualisierungsformen

Sowohl das bestehende Frontend des Titan Control Centers als auch unser Entwurf setzen ähnliche Darstellungen zur Visualisierung von Messwerten ein. Bei den dargestellten Messwerten der ActivePower- und AggregatedActivePowerRecords handelt es sich um Zeitreihendaten. Bei der Visualisierung von Zeitreihendaten liegt ein Fokus darauf, deutlich zu machen, welche Werte sich verändert haben und welche gleichgeblieben sind [Yau 2014]. Beide Frontends benutzen Graphen, Kreisdiagramme und Balkendiagramme. Few [2006] und Yau [2014] favorisieren Balkendiagramme anstelle von Kreisdiagrammen und begründen dies damit, dass diese schneller lesbar seien. Der Grund hierfür sei,

[2] `http://samoa.se.informatik.uni-kiel.de:8185/`
[3] `https://vuejs.org/`

Tabelle 6.1. Bedeutendste Unterschiede zwischen der Kommunikation über Prometheus oder über den REST Adapter

Aspekt	Prometheus	REST Adapter
Ressourcenverbrauch	•hoch, ein weiteres System wird durchgehend betrieben	•gering, der REST Adapter adaptiert ausschließlich Anfragen
Abfrageflexibilität	•hoch, für Abfragen, die mit PromQL formuliert werden können, andernfalls gering	•eher gering, die Implementierung des REST Adapters und den Schnittstellen eines Microservices müssen angepasst werden. •komplexe Abfragen möglich
Abhängigkeit	•eher gering, Prometheus fungiert als Datenspeicher und lässt Abfragen auch dann zu, wenn die Titan Plattform offline ist	•eher hoch, der angefragte Microservice muss online sein

dass Veränderungen von mehrdimensionalen geometrischen Formen schwieriger zu erkennen seien als Unterschiede in Länge oder Position von Balken. Hollands und Spence [1992] stimmen der Argumentation zu. Sie konstatieren jedoch, dass Kreisdiagramme sinnvoll eingesetzt werden können, wenn Proportionen von unterschiedlichen Gruppen (in unserem Fall beispielweise aggregierte Sensoren) dargestellt werden sollen. Das Grafana Dashboard bietet zusätzlich die Visualisierung durch eine halbrunde Gauge-Darstellung, außerdem werden einige Diagrammtypen in Form von Stack-Diagrammen dargestellt. Für die Darstellung von Trendanzeigen nutzen die Dashboards unterschiedliche Darstellungsformen. Das Frontend des Titan Control Centers nutzt entsprechend der Veränderung eingefärbte Pfeile, um den derzeitigen Trend anzuzeigen. Unser Grafana Dashboard nutzt Panels, die ebenfalls entsprechend der aktuellen Tendenz eingefärbt sind. Der Entwicklungsverlauf wird jedoch in Form der prozentualen Abweichung in Prozent angezeigt (siehe Abbildung 4.8). Das bestehende Frontend des Titan Control Centers greift auf eine Schnittstelle des History Services zu, um Trend-Metriken zu erhalten. In Grafana haben wir dafür das in Abschnitt 4.4.1 vorgestellte Trend Box Panel genutzt, welches die Berechnung der Tendenz direkt aus der Zeitreihe im Frontend vornimmt. Ein weiterer Unterschied ist, dass unser Grafana Dashboard beispielsweise in Stack-Diagrammen oder bei der Anzeige verschiedener Korrelate mehrere Zeitreihen pro Graph gleichzeitig darstellt. Durch die Anzeige mehrerer Zeitreihen in einem Diagramm wird ein Vergleich unterschiedlicher Messpunkte und das Einordnen verschiedener Metriken in den Kontext anderer Metriken möglich. Yau [2014] beschreibt die Wichtigkeit, solche Vergleiche zuzulassen, um das Datenverständnis zu steigern und Schlussfolgerungen zu ermöglichen. Das bestehende Frontend zeigt pro

Tabelle 6.2. Bedeutendste Unterschiede zwischen der Speicherung von Daten über das Control Center und der direkten Speicherung in Prometheus

Aspekt	Direkte Kommunikation (Flow Engine -> Prometheus)	Indirekte Kommunikation (Flow Engine -> Control Center -> Prometheus)
Ressourcenverbrauch	•gering	•hoch, das Control Center inkl. Kafka wird benötigt
Entwicklungsaufwand	•eher gering, es wird lediglich ein Prometheus Exporter benötigt	•hoch, die Anbindung an das Control Center muss hergestellt werden
Abhängigkeit	•gering, Speicherprozesse ohne das Control Center möglich	•eher hoch, Control Center und Kafka müssen online sein
Grad der Integration	•gering, Analysekomponenten der Titan Plattform können kaum genutzt werden	•hoch, Titan Plattform kann genutzt werden (bspw. Microservice für Statistiken)

Graph eine Metrik an, bietet jedoch die Möglichkeit, mehrere Zeitreihen für einen Vergleich gemeinsam anzuzeigen.

Das derzeitige Frontend des Titan Control Centers ist mit mehr Microservices verbunden als unser Grafana Dashboard, insbesondere ist der Microservice für Statistiken eingebunden. Da in unserem prototypischen Ansatz zum jetzigen Zeitpunkt diese Metriken noch nicht vorliegen, können wir keinen Vergleich der Darstellungsformen aufzeigen.

Wir haben das von uns entwickelte Dashboard so aufgebaut, dass es mehrere Dashboards gibt, jede einzelne Seite jedoch genau auf eine Fenstergröße skaliert wird. Wie in Abbildung 4.2 und Abbildung 4.8 zu sehen, sind dafür Darstellungen zum Teil nebeneinander angeordnet. Das aktuell eingesetzte Frontend stellt Graphen grundsätzlich untereinander dar, wodurch das Dashboard nicht auf einer Seite skaliert werden kann. Few [2006] problematisiert die Notwendigkeit des Wechsels der Ansicht durch Scrollen oder Umschalten, da ein Betrachter sich während des Umschaltens nur wenige Informationen merken kann. Ein Wechsel verschiedener Perspektiven sei jedoch dann nützlich, wenn berücksichtigt wird, dass zusammenhängende Informationen gemeinsam angezeigt werden können, ohne dass ein Umschalten benötigt wird.

Die eingesetzten Visualisierungsformen beider Dashboards sind einfach und übersichtlich gehalten und zeigen die reinen Metrikdaten. Die einzelnen Fenster der Visualisierung besitzen keine aufwendigen Verzierungen oder Animationen, wodurch die Messwerte gut lesbar sind. Für Few [2006] ist dies ein wichtiges Kriterium für ein gutes Dashboard-Design, da das Hauptaugenmerk auf das Wesentliche – die Messwerte – ausgerichtet ist. Der Fokus ist so stark auf die Messwerte ausgelegt, dass der Hintergrund der Daten (bspw. aggregierter Sensoren) nicht weiter ausgeführt wird. Yau [2014] gibt zu bedenken, dass die schriftliche Erklärung von Daten dem Verständnis dienlich sein kann. Wir gehen allerdings an dieser Stelle davon aus, dass die Benutzer beider Dashboardlösungen hinreichend mit dem Hintergrund der Daten vertraut sind, sodass die Übersichtlichkeit der Darstellung als oberste Priorität angesehen wird.

Beide Dashboards sind in der Lage, kontinuierlich neue Metrikdaten abzufragen. Das bestehende Frontend des Titan Control Centers aktualisiert die Metrikdaten in einem festgelegten Intervall und bietet die Möglichkeit, den Vorgang zu pausieren. Das Grafana Dashboard ermöglicht zusätzlich die Einstellung der gewünschten Abfrageperiode. Weiter ist es mithilfe Grafanas möglich, einen gezielten Zeitraum abzufragen. Abbildung 5.6 zeigt die Abfrage eines Zeitraumes in der Vergangenheit.

Ein weiterer Unterschied besteht im Aufbau der Navigation. So ist es im bestehenden Frontend möglich, durch die bestehende Sensor-Konfiguration zu navigieren und untergeordnete Sensoren anzuzeigen oder zu einem übergeordneten Sensor zu navigieren. Abbildung 6.1 zeigt die entsprechende Navigationsleiste.

Bei der Implementierung unseres Dashboards haben wir uns nicht an der Struktur der Konfiguration orientiert. Unser Fokus liegt auf einem Top-Down-Ansatz. So zeigt das erste vorkonfigurierte Dashboard unseres Entwurfs die als am wichtigsten angesehenen Verbraucher. Das zweite Dashboard zeigt detaillierte Messwerte unterschiedlicher Sensoren

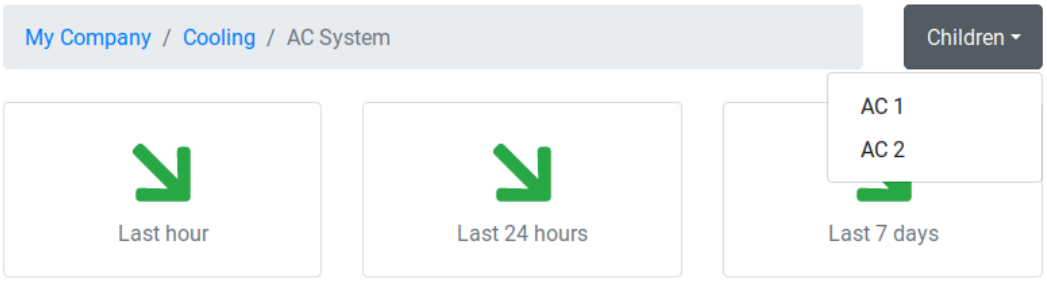

Abbildung 6.1. Navigationsleiste des aktuellen Frontends des Control Centers, um zwischen der Anzeige unterschiedlicher Werte umzuschalten. Die Pfeile zeigen Trendanzeigen.

und ermöglicht es, interaktiv neue Korrelationen in vorhandenen Darstellungsformen zu konfigurieren. Das Dashboard für die Daten des Kieler Druckzentrums haben wir so konfiguriert, dass diejenigen Verbraucher angezeigt werden, welche den höchsten Energiebedarf aufweisen. Außerdem haben wir verschiedene Messwerte miteinander korreliert.

6.2.2. Gegenüberstellung von Grafana und Vue.js

In dem von uns vorgestellten Verfahren nutzen wir Grafana als Grundlage für eine Single-Page Anwendung. Grafana ist eine häufig eingesetzte Plattform für Analyse und Monitoring von Daten. Grafana ermöglicht die Kreation von Dashboards per Drag and Drop. Das bestehende Frontend des Titan Control Centers ist unter Nutzung des Single-Page Frameworks Vue.js geschrieben. Im Gegensatz zu einer Eigenentwicklung, wie sie mithilfe von Vue.js möglich ist, profitiert man durch Nutzung Grafanas davon, dass verhältnismäßig wenig eigene Entwicklungsarbeit nötig - aber möglich ist. Das heißt, Grafana wird stetig weiterentwickelt, neue Datenquellen werden angebunden und Fehler behoben. Gleichzeitig ist es aber auch möglich, eigene Plug-ins zu entwickeln. Wie in Abschnitt 8.2 ausgeführt könnte ein *Admin Plug-in* entwickelt werden, welches die Aggregation von Sensoren analog zu dem bereits vorhandenen Titan Frontend ermöglicht. Mit dieser Entwicklung wäre Grafana als Frontend ohne Einschränkung nutzbar. Durch das Wegfallen der Entwicklungsarbeit an dem Frontend und die Möglichkeit, weitere Datenquellen oftmals ohne Entwicklungsarbeit anbinden zu können, können Ressourcen anderweitig eingesetzt werden. Um die Kommunikation mit Grafana aufbauen zu können, würde grundsätzlich der Einsatz von Datenbanken genügen, mit denen Grafana kommunizieren kann[4]. Durch die einfache Möglichkeit weitere Datenquellen anzubinden könnte auch der Integrationsprozess von Systemen und Produktionsstätten begünstigt werden. Das liegt daran, dass für die Integration unterschiedlicher Systeme und Datenformate verschiedene Datenbanktechnologien prädestiniert sein können. So könnte die Ausführung spezifisch angepasster Flows

[4] https://grafana.com/grafana/plugins?type=datasource

Tabelle 6.3. Bedeutendste Unterschiede zwischen dem aktuellen Frontend und dem Grafana Dashboard

Aspekt	Aktuelles Frontend des Control Centers	Grafana Dashboard
Technologie	•Vue.js	•Grafana
Anpassungsaufwand	•eher hoch, dafür große Freiheiten	•eher gering, dafür eingeschränkt durch die Grenzen Grafanas
Visualisierungsformen	•Graph, Kreisdiagramm, Balkendiagramm, Trendanzeige (Pfeil), Histogramm	•(Stack-) Graph, Kreisdiagramm, (Stack-) Balkendiagramm, (Bar-) Gaugediagramm, Trendanzeige (Prozentangabe)
Abfrage- und Visualisierungsmöglichkeiten	•Es wird auf die bestehenden Microservices zugegriffen •Visualisierungen sind abhängig von der Implementierung, dafür keine Einschränkung	•Es kann zusätzlich PromQL eingesetzt werden •Visualisierungen über das Frontend möglich, aber abhängig von Grafana

Informationen in verschiedene Datenbanken speichern, auf die beispielsweise über Grafana zugegriffen werden kann.

Eine Eigenentwicklung hingegen ermöglicht die Entwicklung einer Plattform, die in ihrer Gänze an die Anforderungen angepasst werden kann. Dieser Punkt ist insbesondere dann wichtig, wenn das Frontend der Titan Plattform für weitere Aufgaben, die über die Anzeige von Monitoringdaten hinausgehen, ausgebaut werden soll.

6.3. Diskussion

Wir haben im vorherigen Abschnitt unterschiedliche Aspekte unserer Arbeit untersucht und an verschiedenen Stellen Unterschiede herausgearbeitet und diese in Tabelle 6.1, Tabelle 6.2 und Tabelle 6.3 dargestellt.

Aufbauend auf den Ergebnissen aus Tabelle 6.1 gehen wir davon aus, dass ein zeitgleicher Betrieb beider Kommunikationswege ein sinnvolles Einsatzszenario ergibt. Der Einsatz von Prometheus ermöglicht das flexible Gestalten von Anfragen und folglich von Visualisierungen. Den zeitgleichen Einsatz des HTTP/REST Kommunikationskanals halten wir für sinnvoll, da auf diese Weise weitreichendere Berechnungen, als die, die mit PromQL ermöglicht werden, durchgeführt werden können. Wir haben uns in dieser Arbeit nicht mit der Effizienz der Datenbanksprache PromQL beschäftigt. Aus diesem Grund können

wir an dieser Stelle keine Aussage darüber treffen, ob beispielsweise die Aggregation von Sensoren über Prometheus oder über das eventbasierte Berechnungsverfahren des History Services effizienter ist oder besser skaliert. Auch wäre es möglich, dass berechnete Werte verschiedener Microservices in einer Prometheus Datenbank gespeichert werden. So könnte einerseits von der Berechnung der Microservices und andererseits von PromQL profitiert werden.

Tabelle 6.2 zeigt Unterschiede auf, die bei der Verbindung der Flow Engine mit Grafana entstehen. So haben wir den Weg über das Control Center und einen Weg vorgestellt, wie Daten direkt in Prometheus gespeichert werden können. An dieser Stelle halten wir es für sinnvoller, wenn die Kommunikation für ActivePowerRecords über das Titan Control Center durchgeführt wird. Das liegt daran, dass Metrikdaten in diesem Fall im System der Titan Plattform vorliegen und verarbeitet werden können. Allerdings kann beispielsweise für das Erheben von Ereignisprotokollen oder die Integration von selten genutzten Maschinen auch der direkte Weg favorisiert werden. Insbesondere aus wirtschaftlicher Perspektive kann der direkte Kommunikationskanal Entwicklungsarbeit sparen und damit eine Kostenreduktion bewirken.

In Tabelle 6.3 haben wir die wichtigsten Unterschiede und Gemeinsamkeiten beider Dashboards tabellarisch dargestellt. Es fällt auf, dass die Unterschiede vor allem in technischen Aspekten bestehen, da ähnliche Visualisierungsformen eingesetzt werden. Auf der technischen Seite sind beide Lösungen jedoch äußerst verschieden, dies wird vor allem dann deutlich, wenn ein Fokus auf den Entwicklungsaufwand oder auf die Möglichkeit gelegt wird, flexible Metrikabfragen zu stellen und zu visualisieren.

Abhängig von Anforderungen und individuellem Einsatzgebiet erscheint die Integration von Grafana oder die Entwicklung eines Dashboards mithilfe eines JavaScript Frameworks wie Vue.js vorzuziehen zu sein. Für den Datenzugriff kann es daher für verteilte Anwendungsszenarien von Vorteil sein, die Kommunikation über eine Kombination aus einem Monitoring Toolkit wie Prometheus und stark individualisierbaren REST Schnittstellen aufzubauen. Diese Kombination ermöglicht unseres Erachtens nach sowohl das Ausführen von flexiblen Operationen auf Zeitreihen als auch die Übertragung und Darstellung komplexer Berechnungen.

Kapitel 7

Verwandte Arbeiten

In diesem Kapitel verorten wir unsere Arbeit im Kontext ähnlicher wissenschaftlicher Publikationen. Wir konnten zum Zeitpunkt dieser Arbeit keine Forschungsergebnisse über Dashboarding anderer verteilter Monitoring Plattformen als Titan finden, die einen Fokus auf die Integration eines Dashboards in ein vorhandenes System legen. In Abschnitt 7.1 ordnen wir eingesetzte Technologien ein, in Abschnitt 7.2 den vorgestellten Ansatz.

7.1. Verortung eingesetzter Technologien

Die wichtigsten eingesetzten Technologien unserer Arbeit sind das Monitoring- und Alertingtoolkit (MAT) Prometheus sowie die Analyse- und Visualisierungsplattform Grafana (siehe Kapitel 2). Prometheus ist eine Open Source Plattform für Monitoring und Alerting und basiert auf einer Zeitreihendatenbank. Graphite[1] basiert ebenfalls auf einer Zeitreihendatenbank und kann durch weitere externe Systeme ebenfalls als MAT eingesetzt werden. Prometheus ermöglicht durch das verwendete Datenmodell und PromQL weitreichendere Operationen mit den Daten. Graphite hingegen ist besser geeignet, wenn historische Daten verarbeitet werden. Genau wie Graphite ist InfluxDB[2] ein push-basiertes Datenbanksystem und verfolgt damit einen anderen Ansatz als Prometheus. Durch einen pull-basierten Ansatz wird es beispielsweise möglich, an einer Stelle zu konfigurieren, in welchen Intervallen Metriken gespeichert werden sollen. In einem push-basierten Ansatz ist diese Konfiguration an jedem Ort vorzunehmen, an denen Speichervorgänge initiiert werden. Für alle drei Systeme existieren Grafana Datenquellen Plug-ins, sodass eine Kommunikation mit Grafana aufgebaut werden kann.

Neben der häufig eingesetzten Kombination aus Prometheus und Grafana existiert ein weiterer Software-Stack für das Erheben, Monitoren und Visualisieren von Daten: der ELK-Stack. Dabei werden durch ElasticSearch[3] Metriken gespeichert, durch Logstach[4] erhoben und verarbeitet. Anschließend werden die Metriken mithilfe der zu Grafana existierenden Alternative Kibana[5] visualisiert.

[1] https://graphite.readthedocs.io/en/latest/
[2] https://www.influxdata.com/
[3] https://www.elastic.co/de/
[4] https://www.elastic.co/de/products/logstash
[5] https://www.elastic.co/de/products/kibana

7.2. Verortung unserer Arbeit

Das Frontend des Titan Control Centers (siehe Abschnitt 2.3.2) ist wie unsere Entwicklung des Grafana Dashboards als Frontend der Titan Plattform konzipiert. Anders als unser Ansatz setzt das Frontend des Titan Control Centers dabei auf eine Eigenentwicklung (siehe Abschnitt 6.2). Das Frontend des Titan Control Centers unterstützt im Gegensatz zu unserem Dashboard die Möglichkeit zu konfigurieren, welche Sensoren automatisch zu einer Gruppe aggregiert werden. In unserem Ansatz setzen wir hingegen den Fokus darauf, eine einheitliche Visualisierung verschiedener Systeme bereitzustellen und von der Nutzung einer high-level Dashboard-Plattform zu profitieren. In unserem Ansatz setzen wir weiter darauf, ein weiteres Monitoringsystem zu integrieren, welches zum Anzeigen von Metriken durch Grafana aufgerufen wird.

Ein anderes Verfahren stellt Chan [2019] vor, dabei werden Metriken von unterschiedlichen Systemen erhoben und ebenfalls in Grafana visualisiert. Anders als bei der Kombination von Grafana und einer Monitoring Plattform stellt Chan ein Verfahren vor, um Metriken eines Superclusters zu erheben und anzuzeigen. Im Gegensatz zu unserem Verfahren wird kein Prometheussystem integriert, stattdessen werden Metriken verschiedener Endpunkte über Telegraf[6] erhoben und in einer InfluxDB-Datenbank gespeichert. Eine Verbindung zwischen Telegraf und Prometheus ist ebenfalls möglich. Auf diese Datenbank wird zwecks Visualisierung von Grafana zugegriffen. Neben der Speicherung in der InfluxDB werden ebenfalls Metriken in einer PostgrSQL[7]-Datenbank gespeichert. Als Begründung für dieses Vorgehen wird angegeben, dass komplexere Metrikabfragen möglich werden, um weitere Korrelationen und Informationen zu gewinnen. Unser Ansatz sieht hingegen vor, auf verschiedene Microservices der Titan Plattform zuzugreifen, um komplexere Berechnungen wie beispielsweise Statistiken zur Visualisierung abzufragen (siehe Abschnitt 8.2).

Ein experimentelles Dashboard für Management und Monitoring einer Microservice-Architektur wird in Mayer und Weinreich [2017] vorgestellt. Hier wird ein zentrales Dashboard eingesetzt, um Daten verschiedener Microservices darzustellen. Die Kommunikation zwischen dem Dashboard und den Microservices ist über das REST Programmierparadigma umgesetzt. Im Fokus dieser Arbeit steht die Visualisierung von Informationen für verschiedene Zielgruppen. So werden beispielsweise Informationen über die Laufzeit für Entwickler oder Informationen über die Systemauslastung für Betreiber (Operator) dargestellt. Dieser Ansatz geht über die derzeitig genutzten Visualisierungswerte unseres Dashboards hinaus. In Abschnitt 8.2 zeigen wir die Möglichkeit auf, die Titan Plattform zu monitoren. Mit dieser Erweiterung wären wir ebenfalls in der Lage eine Entwicklerperspektive und eine Kundenperspektive unseres Dashboards darzustellen. Grundsätzlich verfolgt unser Dashboard jedoch denselben Ansatz, dass für verschiedene Bereiche verschiedene Dashboard-Ansichten zur Verfügung stehen (in unserem Fall Titan View 1, Titan View 2, Kieler Nachrichten).

[6] https://www.influxdata.com/time-series-platform/telegraf/

[7] https://www.postgresql.org/

Ähnlich wie die Titan Plattform basiert das Softwarevisualisierungstool Explorviz [Fittkau u. a. 2017] ebenfalls auf dem Microservice-Architekturmuster. Explorviz ermöglicht es, Kommunikation, Auslastung und Ressourcenverbrauch einer Software zur Laufzeit grafisch darzustellen. Krippner [2019] untersucht Design- und Implementierungsmöglichkeiten eines Dashboards für die Exlporviz Plattform. Dabei werden die Daten, die zur Laufzeit einer zu visualisierenden Software erhoben werden, in einem neuentwickelten Dashboard dargestellt. Dafür wird, ähnlich zu unserem Ansatz, eine Backend und eine Frontend Komponenten entwickelt. Innerhalb von Exlporviz wird zur Kommunikation zwischen den Microservices Apache Kafka genutzt, dieses Vorgehen ist ähnlich wie in der Titan Plattform. Die Backend Komponente greift auf die über Kafka übertragenen Daten zu und stellt diese über eine REST API derart zur Verfügung, dass das Frontend diese in benötigten Formaten abgreifen kann. Dieses Vorgehen ähnelt der Architektur des Kafka Exporters, der in unserem Ansatz Metriken für Prometheus zur Verfügung stellt. Im Gegensatz zu unserem Dashboard-Entwurf stellt die Entwicklung der Frontend Komponente für Explorviz eine Erweiterung der bestehenden Weboberfläche dar. Die Frontend Komponente ermöglicht dem Nutzer ebenfalls die flexible und individuelle Gestaltung des Dashboards durch das Hinzufügen und Konfigurieren von Widgets, die zur Visualisierung genutzt werden. Dieses Vorgehen kann mit dem Platzieren und Konfigurieren verschiedener Panel Plug-ins in Grafana verglichen werden. Die eingesetzten Technologien beider Ansätze unterscheiden sich an verschiedenen elementaren Punkten: Die Erweiterung des Explorviz-Frontends basiert auf dem Javascript Framework Ember.js[8] und ist damit eher mit der Technologie des derzeit eingesetzten Frontends des Titan Control Centers als mit der in unserem Ansatz eingesetzten Grafana Plattform zu vergleichen.

[8] https://emberjs.com/

Kapitel 8

Fazit und Ausblick

8.1. Fazit

Das übergeordnete Ziel dieser Bachelorarbeit war aufzuzeigen, wie eine Analyse- und Monitoring Plattform wie Grafana als Frontend einer verteilten Industrial DevOps Plattform integriert werden kann. Dazu haben wir gezeigt, wie verschiedene Kommunikationskanäle zwischen der Titan- und der Grafana Plattform hergestellt werden können. Insbesondere sind wir dabei darauf eingegangen, welche Eigenschaften die Nutzung eines Monitoring- und Alerting-Toolkits wie Prometheus mit sich bringt, wenn dieses als Zwischenglied zwischen Titan und Grafana platziert wird.

Zur Gestaltung des Dashboards haben wir zwei Datensätze nutzen können. Auf der einen Seite waren dies generierte Testdaten, die die Demoversion der Titan Plattform bereitstellt, auf der anderen Seite echte Messwerte des Druckzentrums der Kieler Nachrichten. Insbesondere der zweite Datensatz ermöglichte es uns, verschiedene Korrelate herzustellen. Dazu haben wir verschiedene Technologien und Kommunikationswege genutzt, sodass wir Informationen über das Control Center und die Flow Engine erheben konnten. Zur Visualisierung haben wir unterschiedliche Möglichkeiten der Grafana Plattform angewendet und mehrere Dashboards erstellt, die verschiedene Perspektiven auf die vorhandenen Messungen zeigen.

Abschließend haben wir gezeigt, welche Vor- und Nachteile der Einsatz einer Plattform wie Grafana für ein Projekt wie Titan entgegen einer Eigenentwicklung bedeuten kann und haben verwendete Visualisierungsformen gegenübergestellt und nach Few [2006] und Yau [2014] verglichen.

8.2. Ausblick

Aufbauend auf unsere bisherige Arbeit kann die Dashboardlösung in vielfältiger Weise weiterentwickelt werden. Ein erster Schritt zukünftiger Arbeit könnte die Anpassung des REST API des History Services sein. Während der Kommunikation über den REST Adapter wird derzeit bei der Abfrage einer Metrik lediglich der Anfangszeitpunkt, nicht aber der Endzeitpunkt für den Aufbau der Response-Nachricht genutzt. Das liegt daran, dass der History Microservice zum Zeitpunkt dieser Arbeit nur einen Parameter *after* entgegennimmt. Die Erweiterung der REST API um einen Parameter, der den Endzeitpunkt

kennzeichnet, könnte zur Effizienzsteigerung der Kommunikation führen. Um die Unterstützung eines zweiten Anfrageparameters herzustellen, haben wir den History Service entsprechend angepasst und einen entsprechenden *Merge Request* gestellt.

Weiter könnte die Namenskonvention der Sensoren dahingehend geändert werden, dass für aggregierte Sensoren der Präfix *titan_aggregated* und für normale Sensoren *titan_machine* gesetzt wird. Während der Nutzung von Prometheus könnten mit dieser Namenskonvention Labels für die Werte aggregierter Sensoren genutzt werden.

Auch die Entwicklung eines Admin Plug-ins könnte eine sinnvolle Erweiterung der entwickelten Dashboardlösung bieten, um die Aggregation von Sensoren über das Grafana Dashboard zu ermöglichen. Dazu bietet die Grafana Plattform die Möglichkeit, eigene Plug-ins zu entwickeln[1] und individuelle Funktionalitäten im Dashboard zu platzieren. Grafana Plug-ins können in JavaScript (beispielsweise mit Angular[2], React[3], Vue.js) implementiert werden. Unser Grafana Dashboard bietet derzeit keine Möglichkeit zu konfigurieren, welche Sensoren automatisch zu einer Gruppe (AggregatedActivePowerRecord) aggregiert werden sollen.

Ein weiterer Vorschlag ist die Integration des in Abschnitt 4.4.1 vorgestellten Panels zur Anzeige von verschiedenen Diagrammtypen. Das Panel ermöglicht die Abfrage der Konfiguration über die Angabe einer URL. Hierfür könnte eine weitere Schnittstelle am REST Adapter hinzugefügt werden. Eine grafische Anordnung der Sensor-Hierarchie könnte dem Verständnis der Monitoringwerte dienen und die Konfiguration weiterer Dashboards erleichtern.

Eine weitere Idee ist, nicht nur Metrikdaten von Geräten und Produktionsstraßen zu erheben, sondern auch Kennzahlen über die Auslastung der Titan Plattform zu ermitteln. Die Titan Plattform kommuniziert intern über das Nachrichtensystem Kafka. Es ist möglich, das Nachrichtensystem zu monitoren. Dafür stellt Prometheus einen Metrik Exporter[4] zur Verfügung, der in der Lage ist, JMX-Metriken zu exportieren. Zusätzlich zu diesem Exporter könnte beispielsweise die REST API des History Services derart angepasst werden, dass die Anzahl der Speichervorgänge oder der Speicherverbrauch der Datenbank durch Prometheus abgefragt werden können. Diese Erweiterung könnte nötige Kennzahlen beispielsweise für Skalierungs- oder Debugzwecke liefern.

Ein weiterer Vorschlag ist die Einführung eines Alarmsystems. Bisher fokussiert die Titan Plattform im Wesentlichen das Visualisieren und Verarbeiten von Messwerten. Neben der Visualisierung ermöglicht Grafana das Alerting, durch welches ein Alarm ausgelöst wird, wenn ein Messwert einen kritischen Schwellenwert überschreitet. Ein Alarm kann über diverse Kommunikationswege (u. a. E-Mail, HTTP, Kafka, Slack[5]) versendet werden. So könnten Verantwortliche benachrichtigt werden, wenn das Kühlsystem so wenig Strom verbraucht, dass eine erfolgreiche Kühlung der Produktion unwahrscheinlich erscheint.

[1] https://grafana.com/docs/plugins/developing/development/

[2] https://angular.io/

[3] https://reactjs.org/

[4] https://github.com/prometheus/jmx_exporter

[5] https://slack.com/intl/de-de/

Literaturverzeichnis

[Albadi und El-Saadany 2008] M. Albadi und E. El-Saadany. A summary of demand response in electricity markets. *Electric Power Systems Research* 78.11 (2008), Seiten 1989–1996. DOI: 10.1016/j.epsr.2008.04.002. (Siehe Seite 1)

[Apache Software Foundation 2017] Apache Software Foundation. Kafka. Accessed: 2019-05-04. 2017. URL: https://kafka.apache.org. (Siehe Seite 12)

[Ashok und Banerjee 2000] S. Ashok und R. Banerjee. Load-management applications for the industrial sector. *Applied Energy* 66.2 (2000), Seiten 105–111. DOI: 10.1016/S0306-2619(99)00125-7. (Siehe Seite 1)

[Chan 2019] N. Chan. A Resource Utilization Analytics Platform Using Grafana and Telegraf for the Savio Supercluster. In: *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (Learning)*. PEARC '19. Chicago, IL, USA: ACM, 2019, 31:1–31:6. DOI: 10.1145/3332186.3333053. (Siehe Seite 58)

[Eckerson 2010] W. W. Eckerson. Performance Dashboards: Measuring, Monitoring, and Managing Your Business. Wiley, 2010. (Siehe Seite 1)

[Ernst & Young 2018] Ernst & Young. Digitalisierung und Industrie 4.0 im Mittelstand. Accessed: 2019-09-03. 2018. URL: https://www.ey.com/Publication/vwLUAssets/ey-digitalisierung-und-industrie-4-0-im-mittelstand/$FILE/ey-digitalisierung-und-industrie-4-0-im-mittelstand.pdf. (Siehe Seite 1)

[Few 2006] S. Few. Information Dashboard Design: The Effective Visual Communication of Data. O'Reilly Series. O'Reilly Media, Incorporated, 2006. (Siehe Seiten 47, 50, 52, 61)

[Fielding und Taylor 2002] R. T. Fielding und R. N. Taylor. Principled Design of the Modern Web Architecture. *ACM Trans. Internet Technol.* 2.2 (Mai 2002), Seiten 115–150. DOI: 10.1145/514183.514185. (Siehe Seite 3)

[Fittkau u. a. 2017] F. Fittkau, A. Krause und W. Hasselbring. Software landscape and application visualization for system comprehension with ExplorViz. *Information and Software Technology* 87 (Juli 2017), Seiten 259–277. DOI: 10.1016/j.infsof.2016.07.004. (Siehe Seite 59)

[Gamma u. a. 1995] E. Gamma, R. Helm, R. Johnson und J. Vlissides. Design Patterns: Elements of Reusable Object-oriented Software. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. (Siehe Seite 17)

[Gellings 1985] C. W. Gellings. The concept of demand-side management for electric utilities. *Proceedings of the IEEE* 73.10 (Okt. 1985), Seiten 1468–1470. DOI: 10.1109/PROC.1985.13318. (Siehe Seite 1)

[Grafana Labs 2019] Grafana Labs. The open observability platform. Accessed: 2019-05-03. 2019. URL: https://grafana.com/. (Siehe Seite 11)

[Hasselbring 2016] W. Hasselbring. Microservices for Scalability: Keynote Talk Abstract. In: *International Conference on Performance Engineering (ICPE 2016)*. ACM, März 2016, Seiten 133–134. DOI: 10.1145/2851553.2858659. (Siehe Seite 5)

[Hasselbring u. a. 2019a] W. Hasselbring, S. Henning, B. Latte, A. Möbius, T. Richter, S. Schalk und M. Wojcieszak. Industrial DevOps. In: *2019 IEEE International Conference on Software Architecture Workshops (ICSAW)*. März 2019, Seiten 123–126. DOI: 10.1109/ICSA-C.2019.00029. (Siehe Seiten 1, 5, 6)

[Hasselbring u. a. 2019b] W. Hasselbring, S. Henning, B. Latte, A. Möbius, T. Richter, S. Schalk und M. Wojcieszak. Industrial DevOps (Presentation). In: *2019 IEEE International Conference on Software Architecture*. März 2019. (Siehe Seite 6)

[Hasselbring und Steinacker 2017] W. Hasselbring und G. Steinacker. Microservice Architectures for Scalability, Agility and Reliability in E-Commerce. In: *Proceedings 2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. IEEE, Apr. 2017, Seiten 243–246. DOI: 10.1109/ICSAW.2017.11. (Siehe Seite 5)

[Henning u. a. 2019] S. Henning, W. Hasselbring und A. Möbius. A Scalable Architecture for Power Consumption Monitoring in Industrial Production Environments. In: *2019 IEEE International Conference on Fog Computing (ICFC)*. Juni 2019, Seiten 124–133. DOI: 10.1109/ICFC.2019.00024. (Siehe Seiten 7, 8)

[Henning 2018] S. Henning. Prototype of a Scalable Monitoring Infrastructure for Industrial DevOps. Masterarbeit. Kiel University, Department of Computer Science, Aug. 2018. (Siehe Seiten 7, 8)

[Hollands und Spence 1992] J. G. Hollands und I. Spence. Judgments of Change and Proportion in Graphical Perception. *Human Factors* 34.3 (1992). PMID: 1634243, Seiten 313–334. DOI: 10.1177/001872089203400306. (Siehe Seite 51)

[Jabbari u. a. 2016] R. Jabbari, N. bin Ali, K. Petersen und B. Tanveer. What is DevOps?: A Systematic Mapping Study on Definitions and Practices. In: *Proceedings of the Scientific Workshop Proceedings of XP2016*. XP '16 Workshops. Edinburgh, Scotland, UK: ACM, 2016, 12:1–12:11. DOI: 10.1145/2962695.2962707. (Siehe Seite 5)

[Khan und Shah 2011] M. Khan und S. Shah. Data and Information Visualization Methods, and Interactive Mechanisms: A Survey. *International Journal of Computer Applications* 34 (Dez. 2011), Seiten 1–14. (Siehe Seiten 27, 32)

[Kreps 2011] J. Kreps. Kafka: a Distributed Messaging System for Log Processing. In: 2011. (Siehe Seite 12)

[Krippner 2019] F. Krippner. Design und Implementation eines Dashboards für ExplorViz. in press. Bachelorarbeit. Kiel University, Sep. 2019. (Siehe Seite 59)

[Lakshman und Malik 2010] A. Lakshman und P. Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.* 44.2 (Apr. 2010), Seiten 35–40. DOI: 10.1145/1773912.1773922. (Siehe Seite 2)

[Lasi u. a. 2014] H. Lasi, P. Fettke, H.-G. Kemper, T. Feld und M. Hoffmann. Industry 4.0. *Business & Information Systems Engineering* 6.4 (Aug. 2014), Seiten 239–242. DOI: 10.1007/s12599-014-0334-4. (Siehe Seite 1)

[Mayer und Weinreich 2017] B. Mayer und R. Weinreich. A Dashboard for Microservice Monitoring and Management. In: Apr. 2017, Seiten 66–69. DOI: 10.1109/ICSAW.2017.44. (Siehe Seite 58)

[Morrison 2010] J. P. Morrison. Flow-Based Programming, 2nd Edition: A New Approach to Application Development. Paramount, CA: CreateSpace, 2010. (Siehe Seite 7)

[Newman 2015] S. Newman. Building Microservices. 1st. O'Reilly Media, Inc., 2015. (Siehe Seite 5)

[Prometheus 2019] Prometheus. Prometheus - Monitoring system and time series database. Accessed: 2019-05-03. 2019. URL: https://prometheus.io/. (Siehe Seite 9)

[Rohrer 2000] M. W. Rohrer. Seeing is believing: the importance of visualization in manufacturing simulation. In: *2000 Winter Simulation Conference Proceedings (Cat. No.00CH37165)*. Band 2. Dez. 2000, 1211–1216 vol.2. DOI: 10.1109/WSC.2000.899087. (Siehe Seite 1)

[Rumbaugh u. a. 2004] J. Rumbaugh, I. Jacobson und G. Booch. Unified Modeling Language Reference Manual, The (2nd Edition). Pearson Higher Education, 2004. (Siehe Seite 13)

[Titan Projekt 2019] Titan Projekt. The industrial devops platform for agile process integration and automation. Accessed: 2019-09-04. 2019. URL: https://www.industrial-devops.org. (Siehe Seiten 1, 6)

[Turnbull 2018] J. Turnbull. Monitoring with Prometheus. Turnbull Press, 2018. (Siehe Seite 9)

[Wagenblass 2018] D. Wagenblass. Spitzenlastmanagement: So lässt sich der Strompreis senken. Accessed: 2019-09-03. 2018. URL: https://partner.mvv.de/blog/spitzenlastmanagement-im-unternehmen. (Siehe Seite 1)

[Yau 2014] N. Yau. Einstieg in die Visualisierung: Wie man aus Daten Informationen macht. Sybex, Wiley-VCH, 2014. (Siehe Seiten 47, 50–52, 61)