# Design und Implementierung zur Messung und Limitierung der Nutzung von Pipeline Ressourcen bei JValue

BACHELOR THESIS

**Florian Oberndörfer**

Eingereicht am 23. Februar 2024


Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik
Professur für Open Source Software

Betreuer:
Georg Schwarz, M. Sc.
Prof. Dr. Dirk Riehle, M.B.A.


Friedrich-Alexander-Universität
Technische Fakultät

# Eidesstattliche Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

---

Erlangen, 23. Februar 2024

# Lizenz



---

Erlangen, 23. Februar 2024

# Abstract

*General instructions:* When it comes to executing code in the cloud, the use of resources always plays a major role. That's because the usage of CPU, RAM, storage and Co ist definitely something, that can quickly become very expensive. That's why tools, which provide cloud infrastructure for the execution of code, are interested in limiting the resource usage, which arises through the execution of their users code.
This can be reached by implementing a quota system, which limits the resource usage for each user.
Goal of this work is now to develop such a quota system, which limits the resource usage per user. This includes the development of a possible architecture as well as a possible implementation for the developed quota system and finally implementing this for the JValue Hub, which provides cloud infrastructure for the execution of ETL-Pipelines in Jayvee, a simple and easy understandable data engineering language.
Therefore we will develop a general quota model, for which we will present a possible architecture and a reference implementation in the JValue project.

# Zusammenfassung

*General instructions:* Bei der Ausführung von Code spielt die Ressourcennutzung eine große Rolle. So ist die Verwendung von CPU, Arbeitsspeicher, Speicherplatz und Co durchaus eine kostspielige Angelegenheit. Tools, welche Cloud Infrastruktur zur Ausführung von Code anbieten, sind somit daran interessiert, die Ressourcennutzung, die durch die Ausführung des Codes ihrer Nutzer entsteht, zu limitieren. Dies kann beispielsweise durch ein Quota System erreicht werden, welches die Ressourcenverwendung pro Nutzer limitert.
Ziel dieser Arbeit ist es, ein solches Quota System zu entwerfen, und ebenso eine mögliche Umsetzung dessen zu erarbeiten. Zusätzlich soll dieses Quota System für den JValue Hub, welcher Cloud Infrastruktur zur Ausführung von ETL-Pipelines in Jayvee, einer leicht verständlichen Data Engineering Sprache, anbietet, implementiert werden.
Hierfür wird zunächst ein allgemeines Quota Modell entworfen, für das daraufhin eine mögliche allgemeine Architektur zur Umsetzung vorgestellt wird und welches anschließend spezifisch für den JValue Hub implementiert wird.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Akronyme

**API** Application Programming Interfaces

**AWS** Amazon Web Services

**REST** Representational State Transfer

**DTO** Data transfer object

**DEV** Development

**MVP** Minimum Viable Product

**NIST** National Institute of Standards and Technology

**Q&A** Questions and Answers

# 1 Einleitung

„Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.“ (Mell & Grance, 2011). So wird Cloud Computing durch die US-amerikanische Standardisierungsschnittstelle National Institute of Standards and Technology (NIST) beschrieben. Die Nachfrage nach Cloud Computing steigt seit Jahren stetig. Laut einer Umfrage von 552 Unternehmen mit mehr als 20 Mitarbeitern nutzten im Jahre 2022 schon 84% dieser deutschen Unternehmen Cloud Computing.
Für die Anbieter von Cloud Computing ist es jedoch wichtig die Nutzung ihrer Cloud Infrastruktur für ihre Kunden zu begrenzen. Zum einen können so Kosten eingespart werden und zum anderen muss verhindert werden, dass Anwender durch übermäßige Nutzung der Infrastruktur andere Anwender bei der Nutzung dieser blockieren. Dies kann zwar durch die Begrenzung der Nutzung pro Anwender nicht direkt verhindert werden, das Risiko dafür sinkt jedoch deutlich.
Die Begrenzung der Nutzung kann durch verschiedene Optionen erreicht werden. Eine dieser Möglichkeiten ist hierbei die Einführung eines Abo-Modells mit verschiedenen Preisstufen, welche eine unterschiedlich hohe monatliche Nutzung der Ressourcen der Cloud Infrastruktur ermöglichen. Dabei kann auch eine Priorisierung der Runs von Kunden höherer Abo-Stufen inkludiert sein.
Ziel dieser Arbeit ist es ein Quota System zu entwerfen, welches die Nutzung von Ressourcen bei Cloud Computing limitiert und so allen Anwendern eine positive Nutzererfahrung bietet und zudem dem Anbieter die Ressourcennutzung seiner Nutzer planbar und profitabel macht.
Anschließend soll dieses Quota System für den JValue Hub, welcher Cloud Infrastruktur zur Ausführung von ETL-Pipelines in Jayvee, einer leicht verständlichen Data Engineering Sprache, anbietet, implementiert werden.
Ziel des JValue Projektes ist hierbei: „Make using data easy, safe and reliable“. Um die hierbei angesprochene Zuverlässigkeit (reliable) auch sicherstellen zu können, muss die Nutzung von JValue limitiert werden, sodass jeder Nutzer zu jeder Zeit ETL-Pipeline Runs starten kann und nicht mit langen Wartezeiten konfrontiert wird.

Zunächst werden wir hierfür Lösungen ähnlicher Tools vergleichen, sowie in der Literatur nach Lösungsansätzen suchen. Dann stellen wir Anforderungen an das Quota System und entwerfen dieses daraufhin. Danach präsentieren wir Wireframes für das Frontend Design, sowie eine Architektur, anhand derer wir das Quota System anschließend für den JValue Hub implementieren.
Abschließend werden wir die erreichten bzw. nicht erreichten Anforderungen evaluieren und Möglichkeiten zur Erweiterung und Optimierung des Quota Systems herausarbeiten.

# 2 Recherche

Um ein allgemeines Quota Modell zu entwickeln, ist es wichtig, Lösungen ähnlicher Tools zu dieser Problemstellung zu vergleichen. Zudem wollen wir deren Anwendbarkeit auf JValue prüfen.
Dies wird im Folgenden anhand von fünf vergleichbaren Tools getan. Anschließend gehen wir noch auf einen Vorschlag aus der Literatur ein.

## 2.1 Lösungen anderer Tools

### 2.1.1 Github Actions

GitHub Actions ist GitHubs Lösung zu Continous Integration und Continous Deployment, also dem Bauen, Testen und Verteilen von Code. GitHub Actions ist somit eigentlich kein populärer Weg für die Ausführung von Code.
Da GitHub Actions jedoch jedem Entwickler bekannt sein sollte, wird hier auch der Quota Mechanismus von GitHub Actions vorgestellt und dessen Eignung für JValue analysiert.
Unterschieden werden muss hierbei zwischen GitHub Actions für öffentliche und für private Repositories.

#### Öffentliche Repositories

Bei öffentlichen Repositories ist die Nutzung von GitHub Actions kostenlos und nicht limitiert. (GitHub Team, n. d.)
Die Ausführung von Pipeline Runs im JValue Hub für öffentliche Projekte kostenlos und unbegrenzt durchzuführen, hätte den Vorteil, dass dadurch voraussichtlich mehr Projekte öffentlich gemacht würden, und dies JValue einen höheren Bekanntheitsgrad verschaffen könnte. Ein zusätzlicher positiver Effekt wäre, dass potenziell zukünftige Nutzer die Möglichkeiten hätten, deutlich mehr Projekte zu vergleichen und im besten Falle Eines zu finden, welches ihrer Problemstellung möglichst nahe kommt.
Da jedoch JValue keine unbegrenzten Ressourcenkapazitäten zur Verfügung stehen, ist dieses Modell für uns keine zielführende Lösung, denn auch der Anteil

von Runs öffentlicher Projekte an der Ressourcenverwendung würde enorme Ressourcenkapazitäten fordern.

#### Private Repositories

Bei privaten Repositories wird ein Abo-Modell genutzt, in welchem, je nach Abo-Stufe, monatlich eine gewisse Anzahl an Pipeline Minuten, sowie eine gewisse Menge an Speicherplatz inkludiert sind.
Wenn diese Kontingente aufgebraucht sind, wird ein Einheitspreis pro zusätzlich genutzter Pipeline Minute und pro Tag und GB an zusätzlich belegtem Speicherplatz berechnet. (GitHub Team, n. d.)
Dieses Modell wäre für JValue gut umsetzbar und hätte mehrere Vorteile.
Erstens ist dieses Modell für den Nutzer transparent, leicht verständlich und zudem den meisten Entwicklern bereits von GitHub Actions bekannt.
Zweitens erleichtert es JValue, den voraussichtlichen Bedarf an Rechenkapazitäten, die zur Verfügung gestellt werden müssen, anhand der gewählten Abo-Stufen der Nutzer, abzuschätzen.
Einen Nachteil dieses Modells stellt jedoch dar, dass hier nicht zwischen verschiedenen Ressourcen unterschieden wird. Anders ist dies bei einigen der im Folgenden vorgestellten Tools, bei denen zum Beispiel die Nutzung von Arbeitsspeicher anders bepreist wird als die Nutzung der CPU.

### 2.1.2 AWS Data Pipeline

Beim alten Abrechnungsmodell von Amazon Web Services (AWS) Data Pipeline wird pro Pipeline ein konkreter monatlicher Betrag verlangt. Unterschieden wird hierbei nur zwischen hoher Frequenz mit mehreren Ausführungen täglich, und niedriger Frequenz mit maximal einer Ausführung täglich.
Zusätzlich wird eine Testphase von einem Jahr mit begrenzter Nutzung angeboten. (AWS Team, n. d. b)
Die Abrechnung in Abhängigkeit von der Frequenz der Ausführung ist zumindest zum jetzigen Stand von JValue noch nicht sinnvoll, denn zur Festlegung einer festen Ausführungsfrequenz, müsste es die Möglichkeit zur Planung der Ausführung von Pipelines geben, sodass die Pipeline beispielsweise jeden Tag um 15 Uhr angestoßen würde.
Da dieses Feature jedoch noch nicht existiert, macht die Abrechnung nach Frequenz der Ausführung derzeit wenig Sinn.
Wenn dieses Feature jedoch in Zukunft verfügbar sein sollte, würde diese Art der Abrechnung die Planung von benötigten Ressourcenkapazitäten deutlich erleichtern.
Eine Unterteilung in hohe und niedrige Frequenz wäre bei JValue jedoch trotzdem schwer, denn die Ausführfrequenz kann von einmal pro Minute bis einmal

pro Jahr reichen.
Zudem müsste man sich überlegen, wie man mit Nutzern umgeht, welche nur sporadisch Pipeline Runs anstoßen.
Das Angebot einer Testphase ist zur Gewinnung von neuen Nutzern und deren Vertrauen sehr nützlich, und durch die Begrenzung der Nutzung während der Testphase hält sich die Belastung auf die Rechenkapazität in Grenzen.

### 2.1.3 AWS mit Apache AirFlow

Bei AWS mit Apache AirFlow erfolgt die Bepreisung nach genutzten Stunden. Hierbei gibt es die Varianten Small, Medium und Large, welche eine unterschiedliche Anzahl an nutzbaren vCPUs, also virtuellen Prozessoren, ermöglichen, woraus schließlich ein unterschiedlicher Stundenpreis resultiert und eine unterschiedliche Ausführgeschwindigkeit erreicht werden kann.
Zudem haben die Nutzer die Möglichkeit, sich zusätzliche Worker und Planer vCPUs dazuzubuchen.
Speicherplatz wird hierbei pro belegtem GB und Monat berechnet. (AWS Team, n. d. a)
Einen Vorteil dieses Modells stellt die Flexibilität dar, die dem Nutzer, durch zubuchbare Worker, eine schnellere Ausführung zu ermöglichen, geboten wird.
Für Nutzer, die jedoch nur laienhaftes Wissen besitzen, kann es durchaus schwierig sein, abzuschätzen, welche Variante sie wählen sollten (S, M oder L), und ob bzw. wie viele zusätzliche Worker sie benötigen.
Aktuell arbeitet JValue jedoch noch nicht parallel bei der Ausführung von Pipelines, weshalb dieser Aspekt sich auf die Zukunft beziehen würde.

### 2.1.4 Azure Data Factory

Bei der Azure Data Factory wird ebenfalls in Stunden abgerechnet. Der Stundenpreis wird hierbei jedoch nach Art des Arbeitsschrittes unterschieden. Ebenso gibt es eine Klassifizierung in Small, Medium und Large mit jeweils unterschiedlichem Stundenpreis, welche anhand der Anzahl an DAGs stattfindet. Ein DAG ist ein gerichteter azyklischer Graph und kann genutzt werden, um die Reihenfolge von Arbeitsschritten für die Ausführung einer Pipeline zu beschreiben.
Zudem wird ein Betrag pro 1000 Runs für deren Orchestrierung berechnet.
Zusätzliche Worker lassen sich hier ebenso dazubuchen. (Azure Data Factory Team, n. d.)
Im Vergleich zu den bisherigen Modellen mit Stundenpreis, ist zudem von Vorteil, dass der Preis je nach Arbeitsschritt unterschieden wird, da unterschiedliche Arbeitsschritte unterschiedlich ressourcenlastig und somit unterschiedlich kostspielig sind.
Im Vergleich zum Modell von AWS mit Apache AirFlow, ist hierbei von Vorteil, dass dem Nutzer die Entscheidung abgenommen wird, ob er Small, Medium oder

Large benötigt, denn hier wird dies transparent anhand der Anzahl an DAGs entschieden.

### 2.1.5 Google Dataflow

Bei Google Dataflow wird bei der Abrechnung nach Art der genutzten Ressourcen unterschieden. So wird beispielsweise die Nutzung der CPU pro Stunde berechnet und die Nutzung des Arbeitsspeichers pro bereitgestelltem GB an Arbeitsspeichergröße pro Stunde.
Die Möglichkeit, mehr Worker Instanzen zu nutzen, gibt es hier in Form von Autoscaling mit der Angabe einer Maximalanzahl. Hierbei werden bei Bedarf automatisch weitere Worker Instanzen bis zur Erreichung der angegebenen Maximalanzahl hinzugebucht. (Google Dataflow Team, n. d.)
Da die Bereitstellung von CPU wiederum andere Kosten verursacht, wie beispielsweise die Bereitstellung von Arbeitsspeicher, ist diese Art der Abrechnung sehr sinnvoll.
Durch die Option von Autoscaling wird dem Nutzer die Entscheidung abgenommen, wie viele Worker er benötigt, wobei stets die Möglichkeit besteht, die Kosten durch Angabe einer maximalen Anzahl der Worker zu beschränken.

### 2.1.6 Zusammenfassung

Grundsätzlich lässt sich zunächst zwischen der Abrechnung nach Einheitspreis und der Abrechnung nach Pauschalpreis differenzieren.
Abrechnung per Einheitspreis findet bei AWS mit Apache AirFlow, Azure Data Factory, sowie Google Dataflow statt.
Die Möglichkeit, sich zusätzliche Worker dazuzubuchen, gibt es hierbei überall. Ein wesentlicher Unterschied jedoch ist, dass bei AWS mit Apache AirFlow der Preis nicht nach der Art der Ressource unterschieden wird, bei den beiden Anderen jedoch schon. Am genauesten und transparentesten geschieht dies bei Google Dataflow, nämlich anhand der konkreten Nutzung der verschiedenen Ressourcen.
Unter den Modellen mit Einheitspreis ist somit für JValue das Modell von Google Dataflow zu bevorzugen.
Die Abrechnung nach Pauschalpreis findet bei GitHub Actions, sowie AWS Data Pipeline statt.
Zum aktuellen Stand von JValue kommt lediglich die Variante von GitHub Actions in Frage, denn für das Abrechnen pro Pipeline in Abhängigkeit der Frequenz der Ausführung fehlt, wie bereits beschrieben, das Feature zur geplanten Ausführung von Pipelines. Auch auf Dauer gesehen gestaltet sich die Unterteilung in verschiedene Frequenzkategorien schwierig, da die Frequenz der Pipeline Ausführungen sehr unterschiedlich sein kann.
Wenn man nun das Modell von GitHub Actions mit dem Modell von Google Dataflow vergleicht, so hat GitHub Actions den Vorteil der besseren Planbarkeit,

denn hier lässt sich anhand der gewählten Abo-Modelle gut abschätzen, welche Kapazität an Ressourcen zur Verfügung gestellt werden muss.
Beim Modell von Google Dataflow lässt sich dies jedoch erst nach einer gewissen Zeit anhand der Nutzerdaten der vergangenen Monate abschätzen.
Google Dataflow bietet dafür den Vorteil, dass hier genau anhand genutzten Ressourcen abgerechnet wird und diese zudem unterschiedlich anhand der Kosten zur Bereitstellung der jeweiligen Ressource bepreist werden können.

## 2.2 Vorschlag aus der Literatur

Im Folgenden beschreiben wir das hierarchische Quota System von Barisits (2015).
Allgemein geht es hierbei um die Speicherung einer enormen Anzahl an Dateien und der Belegung großer Mengen an Speicherplatz.
Hierfür soll ein Quota System eingeführt werden, welches Ausnahmen zulassen soll, wofür ein hierarchischer Aufbau sorgen soll.
Ein Storage Element bezieht sich hierbei auf eine Menge von gespeicherten Dateien.
Jede Einschränkung bzw. Limitierung ist dabei durch folgendes Quintupel definiert:
(*Account*, *RSEexp*, *#bytes*, *#files*, *#level*)

- *Account* steht für den Account des Nutzers, für den diese Limitierung gelten soll
- *RSEexp* beschreibt für welche Storage Elemente die Einschränkung gelten soll
- *#bytes* legt die maximale Anzahl an Bytes fest, die innerhalb der definierten Storage Elemente in Summe geschrieben werden dürfen
- *#files* legt die Anzahl an Dateien fest, welche innerhalb der definierten Storage Elemente geschrieben werden dürfen
- Das *#level* ist eine Zahl $>= 0$, welche nun für die hierarchische Struktur sorgt

Bei Quintupeln mit unterschiedlichem *Level* hat dasjenige mit höherem *Level* Gültigkeit.
Bei gleichem *Level* gelten von beiden die jeweils strengeren Limitierungen.
Ausnahmen lassen sich dann durch ein höheres *Level* umsetzen.
Wäre also beispielsweise die eigentliche Regel, dass der Nutzer Tom maximal 20 GB in Summe schreiben darf, er aber bei den Storage Elementen im Speicherort Studium unendlich viel schreiben dürfen soll, so könnte die allgemeine Regel *Level 0* haben und die Ausnahme *Level 1*.

Im vorgestellten Artikel geht es nicht darum, die beste Bepreisung für Ressourcen zu finden, sondern darum, wie ein Quota System umsetzbar ist, insbesondere mit verschiedenen Hierarchien an Regeln, bzw. Ausnahmen. Dieses Quota System bezieht sich lediglich auf die Begrenzung von Speicherplatz. Das vorgestellte Quintupel ließe sich dabei aber auch um Parameter und so auch um Ressourcen erweitern. So könnte man beispielsweise für jede Ressource, die man begrenzen möchte, ein Limit zum Tupel hinzufügen. Man könnte ebenso für jede Ressource einen eigenen *Level* Parameter erstellen, sodass für jede Ressource eine eigene Hierarchie entsteht. Dies würde das System deutlich flexibler, jedoch aber auch deutlich komplexer machen.

# 3 Anforderungen

In diesem Kapitel werden die Anforderungen an ein Quota Management System für JValue definiert. Die im Folgenden dargestellten Anforderungen wurden größtenteils durch einen Anforderungsworkshop mit den drei Entwicklern Georg Schwarz, Philip Heltweg und Johannes Jablonski gesammelt. Ziel des Workshops war die kollaborative Erstellung einer Vision für ein Quota Management System. Aus der Sammlung an Anforderungen wurde ein Subset an Anforderungen für diese Arbeit ausgewählt und priorisiert.

Unterteilt werden die Anforderungen nach deren Priorität:
**A-Anforderungen** stellen diejenigen Anforderungen dar, welche in jedem Falle umgesetzt werden müssen, um ein voll funktionsfähiges Quota System bereitzustellen.
**B-Anforderungen** stellen diejenigen Anforderungen dar, welche umgesetzt werden sollten, jedoch für die Grundfunktionalität nicht zwangsläufig notwendig sind.
**C-Anforderungen** stellen diejenigen Anforderungen dar, welche umgesetzt werden könnten, jedoch nicht zur allgemeinen Funktion notwendig sind und lediglich Features darstellen, welche für den Nutzer oder JValue nützlich sein könnten.
**D-Anforderungen** stellen diejenigen Anforderungen dar, welche für die Zukunft eine Option zur Umsetzung darstellen, jedoch auch teilweise mit dem aktuellen Stand von JValue noch gar nicht umsetzbar sind. Sie stellen eher Ideen für die Zukunft dar, welche noch genauer ausgearbeitet werden müssen.

Die vorgestellten Anforderungen beziehen sich hierbei auf ein Quota System bei JValue, stellen jedoch nicht die Anforderungen an diese Bachelorarbeit dar, da sie den Umfang einer Bachelorarbeit bei weitem überschreiten würden.
Im Rahmen dieser Arbeit wurden folgende Anforderungen als Ziel definiert:

- Die Gesamtheit aller *A-Anforderungen* stellt ein Minimum Viable Product (MVP) dar, welches im Rahmen dieser Arbeit designed und implementiert werden soll.
- Die aufgeführten *B-Anforderungen* stellen Stretch Goals für diese Arbeit dar. Diese weiterführenden Anforderungen dienen der Erweiterung des MVPs, falls die Bearbeitungszeit der Arbeit dies ermöglicht.

- Ähnliches gilt für *C-* und *D-Anforderungen*, allerdings haben diese geringere Priorität gegenüber den *B-Anforderungen*.

## 3.1 A-Anforderungen

**A1.1:** Die Anzahl an Pipeline Runs pro Projekt und pro Nutzer müssen aufgezeichnet werden
**A1.2:** Die Ausführung von Pipeline Runs pro Nutzer und Monat muss begrenzt werden.
**A2:** Es muss ein Dashboard geben, in welchem dem Nutzer folgende Daten angezeigt werden:
**A2.1:** Anzahl Pipeline Runs pro Nutzer und Monat
**A2.2:** Anzahl Pipeline Runs pro Projekt
**A3:** Es muss mindestens 2 Abo-Stufen geben, durch welche das Kontingent an Pipeline Runs pro Nutzer und Monat vorgegeben wird:
**A3.1:** Eine mit limitierter Anzahl an Pipeline Runs pro Monat
**A3.2:** Eine mit unlimitierter Anzahl an Pipeline Runs pro Monat

## 3.2 B-Anforderungen

**B1.1:** Die von einem Pipeline Run benötigten Ressourcen (CPU-Zeit, genutzter Arbeitsspeicher) sollen gemessen und dargestellt werden.
**B1.2:** Die Pipeline Runs sollen anhand des genutzten Arbeitsspeichers und der genutzten CPU-Minuten in verschiedene Kategorien mit unterschiedlicher Bepreisung unterteilt und anhand dessen begrenzt werden.
**B2.1:** Der genutzte Speicher für die Bereitstellung der Ergebnisse eines Pipeline Runs soll gemessen und dargestellt werden
**B2.2:** Die Nutzung von Speicher soll begrenzt werden.
**B3:** Es soll mindestens 2 Abo-Stufen geben, durch welche das Speicherplatzkontingent des jeweiligen Nutzers vorgegeben wird:
**B3.1:** Eine mit limitierter Menge an Speicherplatz insgesamt
**B3.2:** Eine mit unlimitierter Menge an Speicherplatz insgesamt

## 3.3 C-Anforderungen

**C1:** Es könnte mehr als 2 Abo-Stufen geben
**C2**: Es könnte ein „Soft Limit“ geben, sodass Nutzer die Quota Limits um einen vordefinierten Prozentsatz überschreiten dürfen.
**C3:** Es könnte die Option zur Auswahl geben, dass bei Erreichen des Speicherlimits automatisch die ältesten Runs mit deren Resultaten gelöscht werden.
**C4:** Es könnte einen Schutz vor „schlechten“ Pipelines geben, wie zum Beispiel

in Form einer Begrenzung der Ausführungsdauer.
**C5:** Ad hoc Ausführungen, wie beispielsweise bei der Ausführung im Editor, könnten nicht zur Quota Nutzung zählen.

## 3.4 D-Anforderungen

**D1:** Es wäre eine Option, Nutzern die Möglichkeit zu geben Quota, Limits pro Projekt zu setzen.
**D2:** Es wäre eine Option, den Nutzern die Wahl zu geben, ob bei Erreichung des Quota Limits das Ausführen weiterer Runs blockiert werden soll, oder ob ab Erreichung des Limits nach Einheitspreis, mit Angabe eines Maximalbetrags, abgerechnet werden soll.
**D3:** Es wäre eine Option, die Ausführung von Runs auf privater Infrastruktur anders zu bepreisen.
**D4:** Es wäre eine Option, im Dashboard eine Schätzung für den Verbrauch bis Monatsende anzuzeigen.
**D5:** Es wäre eine Option, dem Nutzer auf Wunsch eine Erinnerungsmail zu senden, wenn ein Quota Limit zu 90% erreicht ist.
**D6:** Es wäre eine Option, spezielle Limits und Optionen für Organisationen anzubieten, wie beispielsweise mehrere Nutzer mit gemeinsamen Organisationslimits.
**D7:** Es wäre eine Option, Admins die Möglichkeit zu geben, die Limits von Nutzern zu ändern.
**D8:** Es wäre eine Option, Nutzer mit höherer Abo-Stufe bei der Ausführung von Pipelines zu bevorzugen.
**D9:** Es wäre eine Option, ad hoc Ausführungen anders zu priorisieren.
**D10:** Es wäre eine Option, bei fehlgeschlagenen Runs einen Button anzubieten, durch welchen dieser Pipeline Run nicht zur monatlichen Nutzung dazuzählt. Diesen Button kann der Nutzer nutzen, wenn er der Meinung ist, dass JValue schuld am fehlgeschlagenen Run ist. Sollte sich jedoch herausstellen, dass ein Nutzer diesen Button missbraucht, wird diesem Nutzer der Button nicht mehr angeboten.
**D11:** Pipeline Runs in der „DEV“ Phase könnten nicht zur monatlichen Nutzung zählen.

# 4 Quota Modell

## 4.1 Allgemeines Quota Modell

Im Folgenden werden wir anhand verschiedener Aspekte der bereits vorgestellten Modelle ein Quota Modell entwickeln, welches allgemein auf Tools anwendbar ist, die Cloud Infrastruktur zur Ausführung von Code zur Verfügung stellen, aus welcher dann zu speichernder Output resultiert.
Eine Ausführung des Codes wird im Folgenden *Run* genannt.

### 4.1.1 Einheitspreis oder Pauschalpreis?

Bei der Entscheidung zwischen der Abrechnung nach Pauschalpreis und der Abrechnung nach Einheitspreis wählen wir eine Mischform.

#### Pauschalpreis via Abo-Stufen

Als Basis dient hierbei die Abrechnung nach Pauschalpreis mit der Auswahl zwischen verschiedenen Abo-Stufen, welche jeweils eine unterschiedlich intensive Nutzung der Ressourcen ermöglichen. Die Begrenzung der Nutzung geschieht anhand der später vorgestellten Metriken.

#### Einheitspreis bei Nutzung über Abo-Stufe hinaus

Zusätzlich wird dem Nutzer die Möglichkeit geboten, die, über die in der gewählten Abo-Stufe hinausgehende, Nutzung per Einheitspreis abrechnen zu lassen. Dies ermöglicht dem Anwender flexible Nutzung ohne Einschränkungen.
Um dennoch eine Beschränkung der Kosten zu erhalten, hat der Nutzer die Möglichkeit, ein Kostenlimit anzugeben, bei dessen Erreichen die weitere Nutzung blockiert wird und somit keine weiteren Kosten für den jeweiligen Monat entstehen.

Diese Art der Abrechnung ist für alle potenziellen Kunden gut geeignet. Kunden, die mit einem festen monatlichen Betrag rechnen müssen, haben die Möglichkeit,

die flexible Option mit der Abrechnung nach Einheitspreis bei Überschreitung des Kontingents zu deaktivieren. Nutzer, die jedoch flexibel sein wollen, können diese Option aktivieren.
Es ist aber ebenso möglich, keinerlei Abrechnung nach Einheitspreis anzubieten und dem Nutzer stattdessen einen unlimitierten Plan anzubieten. Dies würde die Flexibilität für den Anwender erhalten, jedoch die Kostenkontrolle für den Anbieter der Cloud Infrastruktur deutlich einschränken, da der Nutzer die Ressourcen so intensiv nutzen kann, wie er möchte, ohne dafür mehr als den monatlichen Abo-Preis zu bezahlen.
Die bereits erwähnten Abo-Stufen werden im Folgenden *Pläne* genannt.

### 4.1.2 Quota Einschränkungen

Die eben schon genannten Metriken zur Beschränkung der Nutzung werden im Folgenden vorgestellt. Hierzu stellen wir die Metrik *Punkte* und die Metrik *Speicherplatz* vor.
Grundlage zur Bestimmung der Metriken stellt die Messung der Nutzung der Ressourcen dar. Die Umsetzung dessen stellen wir im **Kapitel Implementierung** vor.

#### Punkte

Wir führen nun das abstrakte Maß *Punkte* ein. In einem *Plan* ist eine gewisse Anzahl an *Punkten* monatlich enthalten. Die *Punkte* sind monatlich und nicht einmalig enthalten, da es sich bei den Ressourcen, die in die Metrik *Punkte* hineinfließen, um Ressourcen mit einmaliger und nicht fortlaufender Nutzung handelt. Pro *Run* wird, je nach Ressourcenlastigkeit des *Runs*, eine gewisse Anzahl an Punkten fällig. Um die Anzahl an *Punkten* eines *Runs* festzulegen, wird zunächst ein *Run Score* berechnet.
In diesen *Run Score* können beliebig viele Ressourcen einbezogen und jeweils unterschiedlich gewichtet werden. Wichtig ist, dass es sich hierbei nur um eine einmalige Ressourcenverwendung handelt. Fortlaufende Ressourcen, wie beispielsweise belegter Speicherplatz, gehören hier nicht dazu.
Im Vergleich zu einem Modell, in welchem jeder *Plan* von jeder Ressource ein gewisses Kontingent beinhaltet, hat unser Modell mit der abstrakten Metrik *Punkte* den Vorteil für den Nutzer, dass er sein Kontingent komplett ausschöpfen kann. So stellt nicht das Kontingent einer Ressource einen Flaschenhals dar, während das Kontingent anderer Ressourcen nicht ausgeschöpft wird.
Denn angenommen die *Runs* eines Nutzers sind besonders CPU-lastig, benötigen jedoch sehr wenig Arbeitsspeicher, so würde bei einem Modell mit einem eigenen Kontingent pro Ressource am Ende des Monats das Kontingent für die CPU-Zeit zwar aufgebraucht sein, das Kontingent für den Arbeitsspeicher wäre aber noch fast komplett verfügbar.

Bei unserem Modell wird dies jedoch durch das gemeinsame Punktekontingent für alle Ressourcen verhindert.
Zur Festlegung dieser *Punkte* wird zunächst ein *Run Score*, wie folgt, berechnet:

$$s(g_1, ..., g_i, r_1, ..., r_i, d) = \frac{\sum_{k=0}^{i} g_k * r_k}{d}$$

Hierbei steht $g_i$ für die Gewichtung der i-ten Ressource und $r_i$ für den verbrauchten Wert der genutzten Ressource. Der Parameter $d$ ist hierbei ein optionaler Divisionsfaktor, der zur Verkleinerung des *Runs Scores* genutzt werden kann.
Ressourcen können hierbei zum Beispiel CPU oder Arbeitsspeicher sein.
Aus dem *Run Score* wird nun eine *Run Größe* berechnet, von welcher dann wiederum die fälligen *Punkte* abhängen.
Die Abhängigkeiten können der folgenden Tabelle entnommen werden. Die Werte $x_s$, $x_m$ und $x_l$ können beliebig gewählt werden, solange $x_s < x_m < x_l$ gilt.

| *Run Score (s)* | *Run Größe* | *Punkte* |
|---|---|---|
| $0 < s < x_s$ | $S$ | 1 |
| $x_s <= s < x_m$ | $M$ | 2 |
| $x_m <= s < x_l$ | $L$ | 3 |
| $x_l <= s$ | $L+$ | $3 + (s - x_l) * 0,5$ |

**Tabelle 4.1:** Allgemeine Punkteverteilung

Bei der Benennung der *Run Größen* haben wir uns für T-Shirt-Größen entschieden, denn diese Klassifizierung sollte für jeden verständlich und intuitiv sein, da die Einteilung in vielen Bereichen in T-Shirt-Größen stattfindet, wie beispielsweise in der agilen Entwicklung zur Aufwandsschätzung.
Für die *Run Größen* $S$, $M$ und $L$ gibt es einen festen *Punkte* Wert, der für *Runs* dieser *Run Größe* fällig wird.
Ist jedoch der *Run Score* größer als $x_l$, so ist dieser *Run* der Größe $L+$ zuzuordnen, deren *Punkte* variabel in Abhängigkeit des *Run Scores* $s$ berechnet werden.
Die Werte für $x_s$, $x_m$ und $x_l$ sollten so gewählt werden, dass die meisten *Runs* den *Run Größen* $S$, $M$ und $L$ zuordenbar sind.
Die *Run Größe* $L$ stellt kein nach oben geöffnetes Intervall dar, denn sonst wäre es möglich, viele kleinere *Runs* in einem sehr großen *Run* zu vereinen, und dadurch müsste man trotzdem nur den *Punkte* Wert für die *Run Größe* $L$ aufbringen.
Die *Run Größe* $L+$ verhindert dies durch die variable *Punkte* Berechnung in Abhängigkeit des *Run Scores*.

### Speicherplatz

Nun kommen wir zu den fortlaufenden Ressourcen. Im Folgenden werden wir uns nur auf den häufigsten Fall konzentrieren, nämlich, dass hierbei die einzige

fortlaufende Ressource der *Speicherplatz* ist. Sollten jedoch mehrere fortlaufende Ressourcen existieren, so kann aus diesen, ebenso wie im Unterkapitel zuvor, ein *Score* und daraus eine Punktzahl errechnet werden.
Wie eben schon beschrieben, wollen wir den *Speicherplatz*, der einem Nutzer zur Speicherung der Resultate seiner *Runs* zur Verfügung steht, limitieren.
Hierzu gibt es die drei folgenden Optionen:

**Zwei-Abo-Modell:** Die erste Option stellt ein getrenntes Abo-Modell für *Speicherplatz* dar, wodurch die Nutzer nun zwei Abos benötigen würden. Eines, über das sie die Punkte zur Ausführung der Runs bekommen und eines, durch das sie den benötigten *Speicherplatz* zur Speicherung der Resultate der *Runs* zur Verfügung gestellt bekommen.
Diese Option hätte den Vorteil, dass die Nutzer spezifisch jeweils den *Plan* wählen können, den sie benötigen, und beispielsweise nicht mehr *Speicherplatz* bekommen, als sie eigentlich bräuchten, nur, weil sie die Anzahl an *Punkten* benötigen, die in diesem *Plan* enthalten ist.
So kann ein Nutzer, der zwar viele *Runs* durchführt, jedoch immer nur das aktuelle Resultat benötigt, einen *Plan* mit vielen *Punkten* beim Abo der *Punkte* wählen, beim *Speicherplatz* jedoch einen günstigen *Plan* mit wenig *Speicherplatz*.
Ein Nachteil ist die Komplexität durch zwei Abos. Für die Nutzer ist es deutlich einfacher nur ein Abo zu buchen, welches bereits alle Ressourcen inkludiert, die sie benötigen .

**Ein-Abo-Modell mit zubuchbarem Speicherplatz:** Die zweite Option stellt ein 1-Abo-Modell dar, in welchem sowohl eine gewisse Anzahl an *Punkten*, als auch eine gewisse Menge an *Speicherplatz* inkludiert ist. Sollte der Nutzer jedoch mehr *Speicherplatz* benötigen, kann er ein zusätzliches *Speicherplatz*-Abo buchen, welches ihm zusätzlichen *Speicherplatz* zu dem bereits im normalen Abo enthaltenen zur Verfügung stellt.
Vorteil dieses Modells ist, dass der durchschnittliche Nutzer, dem der *Speicherplatz* des normalen Abos ausreicht, kein 2. Abo benötigt. Zusätzlich besteht die Flexibilität bei Bedarf *Speicherplatz* dazuzubuchen, da der *Speicherplatz* in der Regel der limitierende Faktor sein wird. Denn dieser wird im Gegensatz zu den *Punkten* nicht monatlich zurückgesetzt.
Nachteil dieses Modells ist zum einen das 2. Abo, welches für manche Nutzer nötig ist, um ausreichend *Speicherplatz* zu bekommen. Zum anderen kann es bei diesem Modell passieren, dass ein Nutzer, der wenig *Speicherplatz*, aber viele *Punkte* benötigt, deshalb einen teuren *Plan* buchen muss, in welchem auch viel *Speicherplatz* inkludiert ist, welchen dieser jedoch gar nicht benötigt.
Gelöst werden kann dieses Problem, durch die Einheitspreis Option für *Punkte*.

**Ein-Abo-Modell mit Einheitspreis für Speicherplatzüberschreitung:** Die dritte Option setzt es so um wie bei der Metrik *Punkte*. Es gibt ein gemeinsames Abo, in welchem sowohl *Speicherplatz* als auch *Punkte* enthalten sind. Sollte ein Nutzer jedoch mehr *Speicherplatz* benötigen, so kann er für den zusätzlichen die Abrechnung per Einheitspreis aktivieren.
Diese Option bietet für den Nutzer maximale Flexibilität, bei geringster Komplexität. Es wird lediglich ein Abo benötigt und durch die mögliche Abrechnung nach Einheitspreis wird dennoch maximale Flexibilität geboten.
Der einzige Nachteil für den Nutzer ist hierbei, dass, wenn er den inkludierten *Speicherplatz* in einem Monat überschreitet und somit per Einheitspreis nachzahlen muss, er dies in den folgenden Monaten auch muss, solange er nicht *Runs* löscht, um wieder Speicherplatz freizugeben.
Da diese Option sowohl am wenigsten komplex ist, als auch am meisten Flexibilität für den Nutzer bietet, entscheiden wir uns für diese Option.

Zusammengefasst haben wir nun somit einen *Plan*, der eine konkrete Anzahl an monatlichen *Punkten* enthält und eine konkrete Menge an *Speicherplatz* insgesamt zur Verfügung stellt. Sollten mehr *Punkte* oder mehr *Speicherplatz* benötigt werden, so kann der Nutzer die Abrechnung nach Einheitspreis aktivieren.

#### Optionale Features eines Plans

Optional können je nach *Plan* auch *Runs* von Nutzern mit höherem *Plan* bei der Ausführung priorisiert werden. Ebenso könnte man je nach *Plan* eine unterschiedliche Art des Supports anbieten.

### 4.1.3 Erreichung des Limits

Wir haben nun zwei Kontingente, zum einen das für die Anzahl an monatlich nutzbaren *Punkten*, und zum anderen das für den zur Verfügung stehenden *Speicherplatz* zur Persistierung der Resultate der *Runs*.
Sollte eines der beiden Kontingente aufgebraucht sein, so kann der Nutzer zunächst keine neuen *Runs* mehr starten.
Sollte das Kontingent an *Speicherplatz* aufgebraucht sein, so hat der Nutzer die Möglichkeit *Runs* und somit auch deren Resultate zu löschen, um wieder *Speicherplatz* zur Verfügung zu haben und somit neue *Runs* starten zu können. Anstatt *Runs* zu löschen, kann der Nutzer auch die Abrechnung nach Einheitspreis für aktivieren.
Sollte jedoch das Kontingent der *Punkte* aufgebraucht sein, muss er entweder auf den Start des nächsten Rechnungsmonat warten, bei welchem sein *Punkte* Kontingent zurückgesetzt wird, oder er aktiviert die Abrechnung nach Einheitspreis, sodass er wieder *Runs* starten kann, welche dann pro genutzten *Punkt* bepreist werden. Hierbei hat er, wie oben bereits beschrieben, die Möglichkeit ein Limit für sich festzulegen.

### 4.1.4 Automatische Freigabe von belegten Ressourcen

Sollte das Speicherplatzkontingent eines Nutzers komplett ausgereizt sein, so kann dieser eigentlich keine *Runs* mehr starten. Um jedoch automatisch wieder *Speicherplatz* freizugeben, führen wir die *Auto Delete* Funktion ein.
Wenn der Nutzer diese Funktion aktiviert hat und sein Speicherplatzkontingent aufgebraucht ist, so kann er nun dennoch *Runs* ausführen, indem dabei die ältesten *Runs* automatisch gelöscht werden, sodass wieder *Speicherplatz* für den neuen *Run* frei wird. Hierbei steht die *Auto Delete* Option über der *Einheitspreis* Option, sodass, sollten beide Optionen aktiviert sein, die *Auto Delete* Funktion genutzt wird, da diese für den Nutzer mit keinen zusätzlichen Kosten verbunden ist.

### 4.1.5 Vorteile des Modells

#### Planbarkeit trotz Flexibilität

Ein großer Vorteil dieses und anderer Modelle mit Pauschalpreis ist der Aspekt der Planbarkeit für den Anbieter. Denn anhand der Anzahl der Nutzer und deren gewählter *Pläne* lässt sich gut abschätzen, wie viele Ressourcen zur Verfügung gestellt werden müssen.
Dennoch bleibt bei unserem Modell die Flexibilität für den Nutzer bestehen, indem bei Überschreitung des Kontingents die Abrechnung nach Einheitspreis gewählt werden kann.

#### Beliebige Anzahl an Ressourcen

Einen weiteren positiven Aspekt stellt die Möglichkeit der Berücksichtigung beliebig vieler Ressourcen und deren individuelle Gewichtung dar.
Es können beliebig viele Ressourcen genutzt werden, um den *Run Score* zu bestimmen und letztendlich dadurch auch einen *Run* zu bepreisen. Zusätzlich ist jede Ressource einzeln zu gewichten, denn jede Ressource ist unterschiedlich kostenintensiv.

#### Volle Nutzbarkeit des Kontingents für den Kunden

Bei einem Modell mit eigenem Kontingent pro Ressource kann ein Nutzer, sobald eines dieser Kontingente ausgereizt ist, keine Runs mehr starten, selbst wenn die anderen Kontingente bei weitem noch nicht ausgereizt sind. Dieses Problem lösen wir mit unserem Quota Modell, indem wir ein gemeinsames Kontingent für alle einmaligen Ressourcen haben.

#### Flexibilität für den Kunden

Kunden, die mit einem fixen monatlichen Betrag planen müssen, können ganz normal die Abo-Funktion nutzen.
Kunden, die jedoch flexibel in der Nutzung der Ressourcen sein wollen, können die Option der Abrechnung nach Einheitspreis aktivieren und so weiterhin *Runs* starten.

### 4.1.6 Nachteile des Modells

#### Komplexität für den Nutzer

Das *Punkte* Modell ist deutlich komplexer als andere, bei denen beispielsweise einfach pro *Run* ein konkreter Betrag verlangt wird. Dadurch ist dieses Modell für den Nutzer sicherlich nicht so leicht verständlich wie manch andere Modelle. Um diesen Faktor für den Nutzer zu minimieren, werden wir im **Kapitel 5** diverse Frontend Komponenten vorstellen, die dem Nutzer klar und einfach, sowie detailliert seine Quota Nutzung anzeigen sollen.

#### Komplexität für die Entwickler des Tools

Ein Quota Modell, welches lediglich anhand der Anzahl der *Runs* bepreisen würde, wäre sicherlich deutlich einfacher und kostengünstiger zu implementieren als das gewählte Modell. Denn bei unserem Modell muss zum Beispiel die Nutzung der Ressourcen gemessen, sowie auch die Berechnung der *Punkte* durchgeführt werden.
Auf die genaueren Anforderungen an die Implementierung wird zu einem späteren Zeitpunkt im **Kapitel 7** noch eingegangen.
Um die Komplexität für die Entwickler zu verringern, wird das Ganze als eigenes Modul entwickelt. Dies erleichtert die Wartbarkeit und verringert die Fehleranfälligkeit.

## 4.2 Quota bei JValue

### 4.2.1 Quota Modell für JValue

Das gerade erarbeitete allgemeine Modell wollen wir nun auf JValue anwenden. Den *Run Score* nennen wir von nun an *Pipeline-Score*, die *Run Größe Pipeline-Size* und die *Punkte Pipeline-Points*.
Bei der Wahl der Ressourcen entscheiden wir uns für CPU und Arbeitsspeicher, jedoch können in Zukunft beliebig viele Ressourcen in die Formel miteinbezogen werden. Wir messen hierbei die genutzte CPU-Zeit in Millisekunden und den genutzten Arbeitsspeicher in MB.

Mit den gewählten Ressourcen entsteht nun folgende Formel:

$$s(g_1, g_2, r_1, r_2, d) = \frac{g_1 * r_1 + g_2 * r_2}{d}$$

Die Gewichtung der einzelnen Ressourcen ergibt sich aus dem Verhältnis der Kosten zur Bereitstellung der CPU und des Arbeitsspeichers. Mehr dazu und zum Divisionsfaktor im **Anhang A.1**.
Die Grenzwerte für den *Pipeline-Score* wählen wir wie folgt: $x_s = 2$, $x_m = 4$ und $x_l = 6$.

| *Pipeline-Score (s)* | *Pipeline-Size* | *Pipeline-Points* |
|---|---|---|
| $0 < s < 2$ | $S$ | 1 |
| $2 <= s < 4$ | $M$ | 2 |
| $4 <= s < 6$ | $L$ | 3 |
| $6 <= s$ | $L+$ | $3 + (s-6) * 0,5$ |

**Tabelle 4.2:** JValue Punkteverteilung

### 4.2.2 Pläne und Erweiterbarkeit des Quota Modells bei JValue

Im Folgenden werden Beispiele für die bereits angesprochenen *Pläne* vorgestellt. Eine beispielhafte Bestimmung der Kontingente kann dem **Anhang A.2** entnommen werden
Hierbei können verschiedene Leistungen in die *Pläne* inkludiert werden, wie hier beispielsweise die Art des Supports, sowie die Priorisierung von *Runs*.

| Name | *Points* | *Speicherplatz* | Support | Priorisierung | Preis |
|---|---|---|---|---|---|
| FREE | 30 | 10 MB | COMMUNITY | NEIN | 0€ |
| BASIC | 300 | 600 MB | COMMUNITY | NEIN | 4,99€ |
| PRO | 2500 | 18000 MB | PRIORITY | JA | 9,99€ |
| STUDENT | 2500 | 18000 MB | COMMUNITY | NEIN | 0 |

**Tabelle 4.3:** JValue Pläne

Community Support bedeutet Support über eine Questions and Answers (Q&A) Seite, auf welcher andere Nutzer antworten können, wo aber auch der JValue Support reagiert.
Priority Support bedeutet Support per direktem Kontakt. Ebenso werden Antworten auf direkte Anfragen intern priorisiert im Vergleich zu Antworten auf Fragen im Q&A.

# 5 Design der Nutzeroberfläche

Zunächst werden wir Vorschläge für das allgemeine Modell erarbeiten und diese anschließend auf JValue anwenden.

## 5.1 Allgemeines Design der Nutzeroberfläche

Für ein allgemeines Modell der Nutzeroberfläche stellen wir verschiedene Komponenten vor, welche genutzt werden können.

### 5.1.1 Quota Card

Die *Quota Card* ist eine kleine Komponente, die lediglich die wichtigsten Quota Informationen sowie den Plan des Nutzer anzeigt. Die genutzten Punkte und der belegte Speicherplatz werden jeweils in einer *Progress Bar* angezeigt. Zusätzlich kann der Nutzer durch Klick auf *Details* sein *Quota Dashboard* erreichen, welches als Nächstes vorgestellt wird.

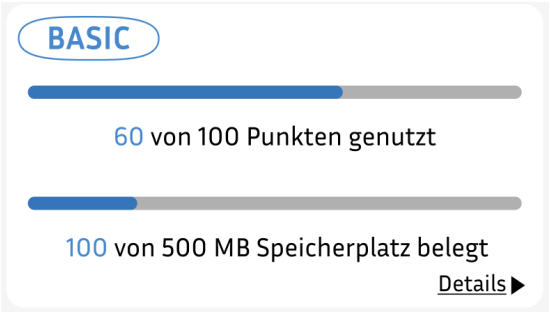


**Abbildung 5.1:** Allgemeines Quota Card Wireframe

### 5.1.2 Quota Dashboard

Das *Quota Dashboard* soll dem Anwender genauere Informationen über die bisherige Nutzung seiner Punkte und seines Speicherplatzes liefern. Die Speicherplatzbelegung zeigen wir hierbei als Fortschrittsleiste an.

Zudem kann der Nutzer auf dieser Seite auch die *Auto Delete* Funktion über das *Toggle* unten rechts aktivieren.
Zusätzlich zu den gerade beschriebenen Komponenten, erstellen wir noch weitere, sodass schließlich folgendes *Quota Dashboard* entsteht:

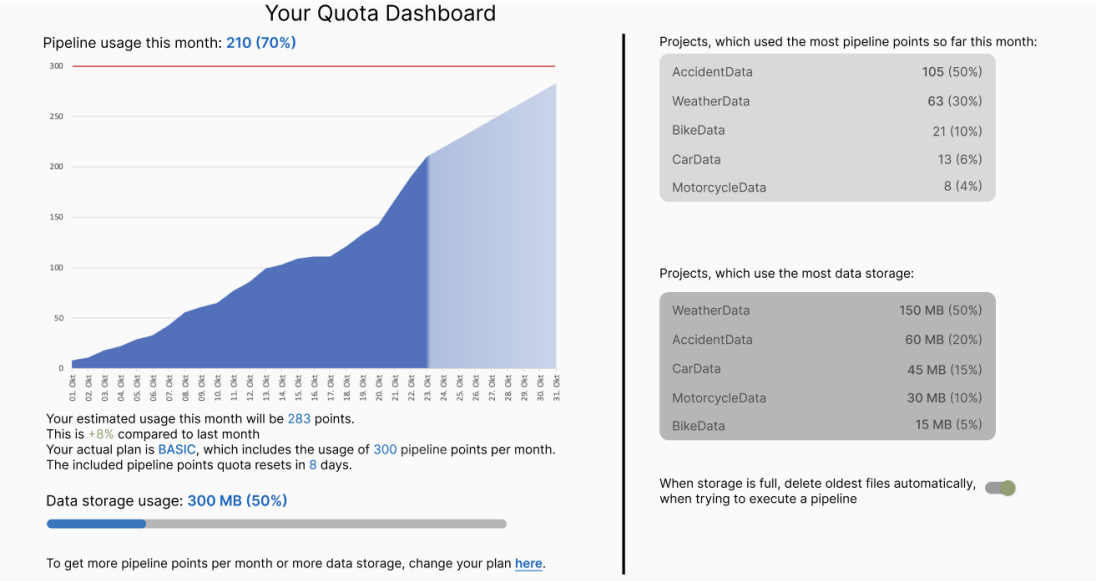


**Abbildung 5.2:** Allgemeines Quota Dashboard Wireframe

### Punkte Diagramm

Im *Punkte Diagramm* wird dem Nutzer angezeigt, wie viele Punkte er bis zu den jeweiligen Tagen des Monats verbraucht hat. Wir haben uns für eine kumulierte Form des Diagramms entschieden, damit der Nutzer die einzelnen Tage nicht addieren muss.
Die x-Achse des Diagramms stellt die Zeit in der Einheit *Kalendertage* dar.
Die y-Achse zeigt kumuliert die *genutzten Punkte* an.
Zusätzlich wird dem Nutzer in Rot, die in seinem Plan inkludierte Anzahl an Punkten angezeigt.
Ebenso wird für die noch kommenden Tage des Monats eine Approximation der Nutzung für den restlichen Monat berechnet, sodass der Nutzer erkennen kann, ob ihm sein aktuell gewählter Plan voraussichtlich ausreichen wird.
Unter dem Diagramm können schließlich noch Daten, wie beispielsweise die Anzahl an Tagen bis zum Beginn des nächsten Rechnungsmonats, also die Tage, bis das Punkte Kontingent des Nutzers wieder zurückgesetzt wird, sowie eine Nutzungstendenz im Vergleich zum letzten Monat angezeigt werden.

#### Projekte/Instanzen mit höchster Punkte Nutzung

Je nachdem, ob es bei dem Tool Projekte gibt oder lediglich Instanzen des Codes, welcher ausgeführt werden soll, können in dieser Komponente diejenigen Projekte/Instanzen aufgelistet werden, welche bisher in diesem Monat am meisten Punkte benötigten.
Hierbei wird der Name des Projekts/der Instanz angezeigt, welcher gleichzeitig als Link zum Projekt/Instanz dient. Zudem werden die jeweils genutzten Punkte und deren Anteil am Gesamtverbrauch angezeigt. Die Komponente kann dem oben dargestellten *Quota Dashboard* entnommen werden.

#### Projekte/Instanzen mit höchster Speicherplatzbelegung

Diese Komponente ist die Gleiche wie die Vorherige mit dem einzigen Unterschied, dass hierbei die Projekte/Instanzen mit höchster Speicherplatzbelegung angezeigt werden. Die Komponente kann ebenso dem oben dargestellten *Quota Dashboard* entnommen werden.

### 5.1.3 Change Plan Page

Die *Change Plan Page* bietet dem Nutzer die Möglichkeit, seinen aktuell gewählten Plan zu ändern.
Um den Nutzer bei der Entscheidung für einen Plan bestmöglich zu unterstützen, wird zunächst dessen Punktenutzung und Speicherplatzbelegung angezeigt. Darunter wird dem Nutzer erklärt, wie die *Run Größe* zustande kommt. Für genauere Informationen zur Berechnung der *Run Größe* kann der Anwender noch auf den Verweis in der Fußnote zu einer weiteren Seite klicken, auf welcher transparent die Funktionsweise zur Bestimmung der Run Größe und die Berechnung der Punkte erklärt werden.
Die wählbaren Pläne werden jeweils als abgerundetes Rechteck dargestellt, wobei der aktuell gewählte Plan mit einem weißen Rahmen hervorgehoben wird.
Darin befinden sich die Details zu den jeweiligen Plänen:

- Anzahl der Punkte, die in diesem Plan monatlich enthalten sind
- Menge an Speicherplatz, die dem Nutzer in diesem Plan zur Speicherung der Run Resultate zur Verfügung steht
- Der monatliche Preis des Plans

Zusätzlich können noch weitere Informationen zu den Plänen angezeigt werden, wie beispielsweise die Art des Supports oder eine mögliche Priorisierung.
Ein Plan kann hierbei durch Klick darauf ausgewählt werden. In Zukunft würde hierbei dann ein Fenster aufgehen, in welchem die Zahlung abgewickelt würde.
Unter der Darstellung der Pläne befindet sich noch ein Hinweis, dass es sich

hierbei um Pläne für einen Nutzer handelt. Pläne für mehrere Personen, wie zum Beispiel für Firmen, müssen über den Link zur Kontaktanfrage beantragt werden.

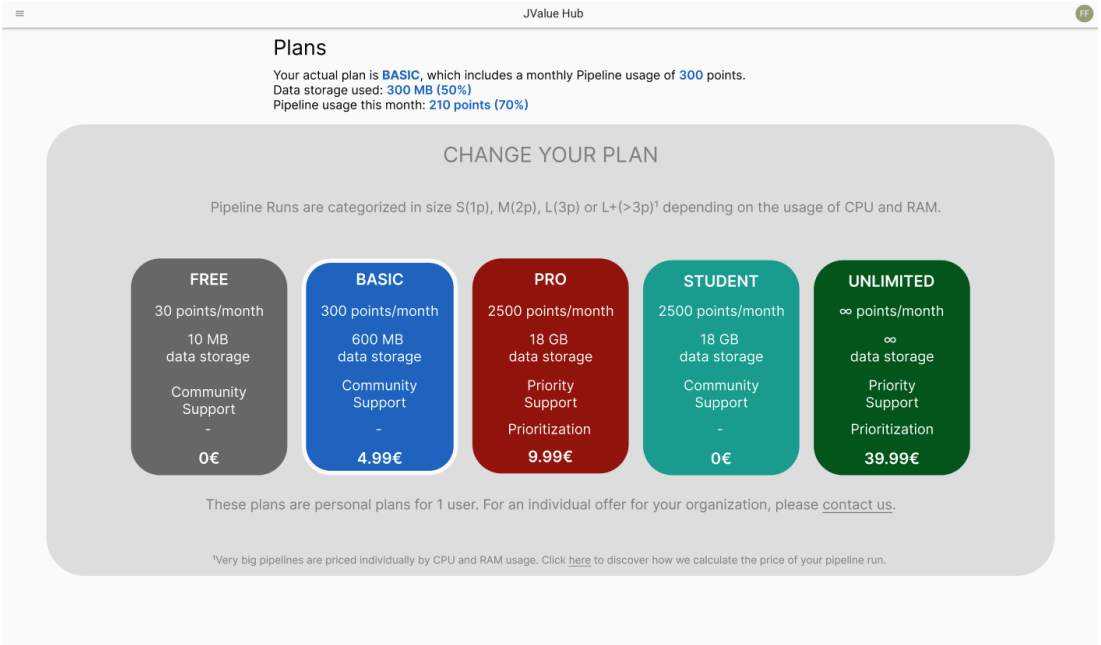


**Abbildung 5.3:** Allgemeines Change Plan Page Wireframe

### 5.1.4 Quota Informationen für Runs

In einer weiteren Komponente wollen wir dem Nutzer bei jedem Run anzeigen, wie viele Punkte dieser benötigt hat und wie viel Speicherplatz die Resultate des Runs belegen.
Diese Informationen wollen wir dort darstellen, wo auch die Runs angezeigt werden. Dies wird je nach Tool unterschiedlich sein, weswegen wir hier keine allgemeine Visualisierung erstellen können.
Ein Beispiel hierfür kann jedoch dem **Kapitel 5.2.2** entnommen werden.
Anzeigen wollen wir die *Run Größe* und die Darstellung der Punkte, die daraus resultieren, sowie die Menge an Speicherplatz, der belegt wird. Zusätzlich bauen wir eine Hover Funktion ein, sodass dem Nutzer, wenn er mit dem Cursor über die Run Größe geht, angezeigt wird, welche Ressourcen in welcher Menge genutzt wurden.

### 5.1.5 Quota Informationen für Projekte/Instanzen

Sollte das Tool eine Darstellung für Projekte oder Instanzen beinhalten, so wollen wir dort auch anzeigen, wie viele Punkte die Ausführungen der zugehörigen Runs

diesen Monat schon benötigt haben, sowie wie viel Speicherplatz die Resultate der zugehörigen Runs insgesamt belegen.

## 5.2 Design der Nutzeroberfläche für JValue

Für JValue nutzen wir alle oben vorgestellten Komponenten und passen diese für den JValue Hub an.
Das *Quota Dashboard*, sowie die *Change Plan Page* müssen dabei nicht verändert werden.

### 5.2.1 Quota Card

Die *Quota Card* integrieren wir in die *Profile Page*. Hierzu ändern wir die *Quota Card* so ab, dass die beiden *Progress Bars* nebeneinander liegen, und die *Card* dann dieselbe Höhe wie die anderen *Cards* der *Profile Page* hat. Die Breite der *Card* wird dadurch verdoppelt.
Hieraus resultiert schließlich folgende *JValue Quota Card*:

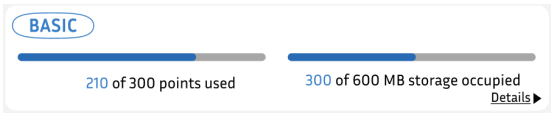


**Abbildung 5.4:** JValue Quota Card Wireframe

### 5.2.2 Quota Informationen für Runs

Unter den allgemeinen Informationen zur Pipeline Instanz befindet sich eine Tabelle mit einer Zeile pro Run dieser Pipeline Instanz.
Zu der Tabelle fügen wir nun zwei Spalten mit den Informationen zu den Punkten und der Run Größe, sowie zum belegten Speicherplatz hinzu. Den Hover Effekt nutzen wir ebenso.

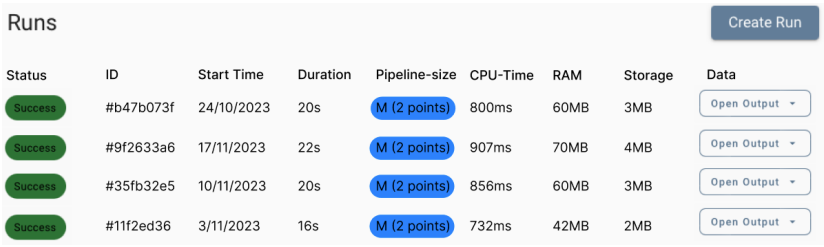


**Abbildung 5.5:** JValue Quota Informationen für Runs Wireframe

### 5.2.3 Quota Informationen für Instanzen

Auf der *Pipeline Page* soll dem Nutzer angezeigt werden, wie viele Pipeline-Points diese Pipeline Instanz diesen Monat schon verbraucht hat, und wie viel Speicherplatz die Resultate der Runs dieser Pipeline Instanz belegen.
Diese Informationen stellen wir oben in den allgemeinen Infos dieser Pipeline Instanz dar.


**Abbildung 5.6:** JValue Quota Informationen für Runs Wireframe

# 6 Architektur

Dieses Kapitel wollen wir ebenso wieder zweiteilen und somit zuerst eine allgemeine Architektur vorstellen und diese anschließend für JValue spezifizieren.

## 6.1 Allgemeine Architektur

### 6.1.1 Datenbankstruktur

#### Speicherung der Nutzerdaten

Wir gehen davon aus, dass bereits eine Tabelle zur Speicherung der Nutzerdaten existiert.
Zu dieser Tabelle fügen wir die Spalte *plan* hinzu, welche die Id des Plans speichert, den der jeweilige Nutzer ausgewählt hat.

#### Speicherung der Quota Daten

Für die Speicherung der Daten zur Quota Nutzung erstellen wir zwei neue Tabellen.
Die Quota Daten in eigenen Tabellen zu speichern und nicht mit in die Nutzertabelle zu übernehmen hat vor allem zwei Vorteile.
Zum einen ist das Quota System so abgekoppelt und Änderungen daran erfordern keine Änderungen in der Nutzertabelle. Dies erhöht die Wartbarkeit.
Zum anderen bietet uns die Speicherung der Punkte Quota in einer eigenen Tabelle die Möglichkeit, auch eine Historie der Punktenutzung eines Nutzers anzubieten, falls gewünscht.
Wir erstellen somit die folgenden zwei Tabellen in der Datenbank.

**Punkte Quota** Für jeden Nutzer muss die Punktenutzung im aktuellen Monat gespeichert werden. Dies geschieht in der folgenden *Punkte Quota* Datenbanktabelle.
Diese Tabelle enthält fünf Spalten. In der ersten Spalte wird eine eindeutige *Id* gespeichert, welche als Primärschlüssel dient.

Die restlichen Spalten enthalten dann die notwendigen Daten.
Zuerst benötigen wir eine Spalte, in welcher der Nutzer, bzw. seine *User Id* gespeichert wird.
Zusätzlich wollen wir nicht nur die Punktenutzung im aktuellen Monat speichern, sondern auch die Punktenutzung der vergangenen Monate. So benötigen wir noch zwei weitere Spalten, die jeweils den Monat bzw. das Jahr speichern, auf die sich die Punktenutzung bezieht, welche in der fünften Spalte gespeichert wird.
Daraus resultiert nun folgende Tabelle:

| *Id* | *User Id* | *Month* | *Year* | *Used Points* |
|---|---|---|---|---|

**Speicherplatz Quota** Ebenso müssen wir die Informationen zur Speicherplatzbelegung der Nutzer speichern.
Hierfür erstellen wir ebenfalls eine Tabelle in der Datenbank, die *Speicherplatz Quota* Tabelle.
Diese enthält drei oder vier Spalten, je nachdem, ob man die *Auto Delete* Funktion anbieten möchte oder nicht.
Die erste Spalte speichert wieder eine eindeutige *Id*, welche als Primärschlüssel dient.
Die zweite Spalte speichert die *User Id*, welcher die Speicherplatzbelegung zuzuordnen ist.
In der dritten Spalte speichern wir nun den belegten Speicherplatz. Je nachdem mit welcher Größenordnung hierbei zu rechnen ist, kann man als Einheit KB, MB oder auch eine andere Einheit wählen. Dies muss jedoch genau definiert werden, sodass hierbei keine Inkonsistenzen entstehen.
Die letzte Spalte speichert schließlich noch, falls gewünscht, einen boolean Wert, welcher angibt, ob der jeweilige Nutzer die *Auto Delete* Funktion aktiviert hat.
Den Monat oder das Jahr müssen wir hierbei nicht speichern, da sich das Speicherplatz Quota nicht monatlich zurücksetzt.

| *Id* | *User Id* | *Storage* | Auto *Delete* |
|---|---|---|---|

**Pläne** Für die Pläne müssen diverse Parameter gespeichert werden.
Zunächst zur Identifizierung des jeweiligen Plans eine eindeutige *Id*.
Ebenso müssen wir die für den Nutzer wichtigen Informationen zu den Plänen speichern, also die enthaltenen Punkte pro Monat, der insgesamt enthaltene Speicherplatz und natürlich auch der Preis des Plans.
Zusätzlich wäre es sicherlich gut, den Plänen auch noch Namen zu geben, welche dann in einem eigenen Parameter zu speichern wären. Nun bleiben noch zwei weitere Parameter, welche bei Bedarf gewählt werden können.
Zum einen der *available* Parameter, welcher angibt, ob der jeweilige Plan aktuell wählbar sein soll. Dies ist beispielsweise sinnvoll, wenn es alte Pläne gibt, die inzwischen zwar nicht mehr wählbar sein sollen, es jedoch noch Nutzer gibt, welche

noch Altverträge mit diesen Plänen haben.
Zum anderen der optionale Parameter *color*, welcher den HEX Code einer Farbe speichert. Dieser Parameter ist notwendig, falls jeder Plan in einer spezifischen Farbe dargestellt werden soll.
Sollten in den Plänen noch weitere Features, wie beispielsweise die Priorisierung von bestimmten Abo-Stufen, enthalten sein, so können für diese noch weitere Parameter notwendig sein.
Zur Speicherung dieser Daten haben wir nun zwei Optionen.
Die Speicherung in einer Datenbanktabelle oder die Speicherung in einer Datei.

**Option 1: Datenbanktabelle** Die erste Option stellt die Speicherung der Pläne in einer eigenen Datenbanktabelle dar.
Aus den bereits vorgestellten nötigen Parametern resultiert schließlich folgende Tabelle:

| *Id* | *Included Points* | *Included Storage* | *price* | *name* | (*color*) | (*available*) |
|---|---|---|---|---|---|---|

Vorteil dieser Option ist sicherlich die bessere Wartbarkeit des Codes, wenn keine Daten im Code gespeichert werden. Zudem erhöht es die Lesbarkeit der Daten, bzw. ermöglicht es auch Nicht-Entwicklern, die Daten der Pläne zu bearbeiten, bzw. neue Pläne zu erstellen.
Nachteil hierbei ist jedoch, dass für die Speicherung in einer Datenbank deutlich mehr Code benötigt wird, zum einen für den Zugriff auf die Tabelle in der Datenbank und zum anderen aber auch für eine eigene Seite im Frontend, über welche die Pläne bearbeitet werden können.

**Option 2: Datei** Die zweite Option stellt die Speicherung in einer Datei im Repository dar, welche beispielsweise eine Liste oder ein Array der Pläne enthält.
Vorteil hierbei ist die einfache Umsetzbarkeit, denn hierfür wird kaum neuer Code benötigt, und die Umsetzung ist somit deutlich kostengünstiger.
Nachteil bei dieser Option ist, dass das Ändern dieser Datei nur von Entwicklern vorgenommen werden kann.

### Speicherung der Run Daten

Hierfür gehen wir davon aus, dass bereits eine Tabelle zur Speicherung der Informationen zu den Runs existiert.
Zu dieser Tabelle fügen wir x+2 Spalten hinzu, wobei x für die Anzahl an Ressourcen steht, welche in den Run Score einbezogen werden sollen. Jede Ressource bekommt ihre eigene Spalte, in welcher die genutzte Menge der Ressource zum jeweiligen Run gespeichert wird, für CPU zum Beispiel die CPU-Millisekunden. Die beiden zusätzlichen Spalten speichern die Anzahl an Punkten, die für diesen Run fällig wurden, sowie die Menge an Speicherplatz, den die Resultate des Runs belegen.

### 6.1.2 REST API

Wir nehmen an, dass bereits ein Frontend und ein Backend existiert, welches per *Representational State Transfer (REST) Application Programming Interfaces (API)* miteinander kommuniziert.
Folgendes Schaubild zeigt zunächs,t die für das vollständige Quota System mit allen Frontend Komponenten aus dem **Kapitel Design Frontend** nötige, *REST API*:

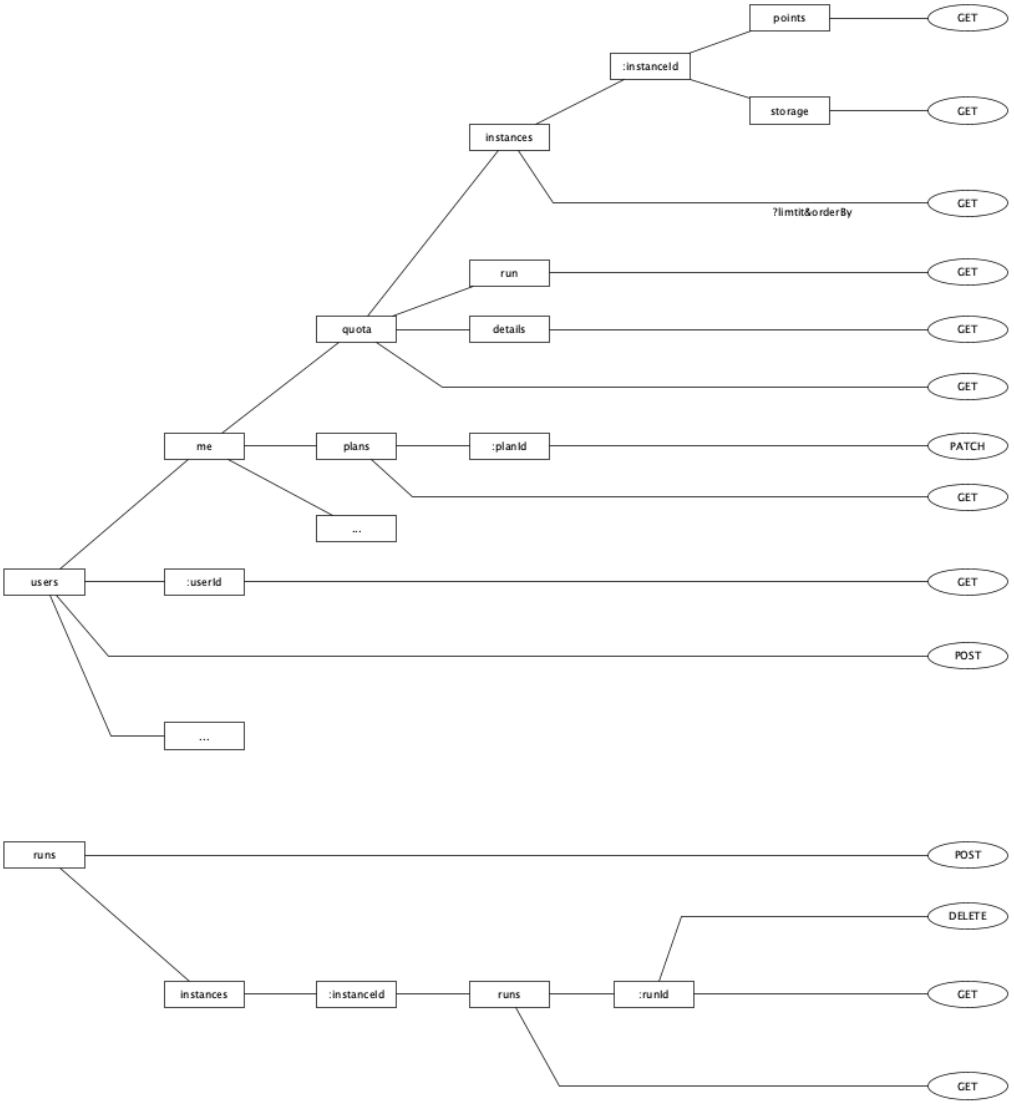


**Abbildung 6.1:** Allgemeine REST API

Hierbei setzen wir zum einen bei *users* und zum anderen bei *runs* an.

**U: *users*:** Die Endpunkte unter *users* beziehen sich auf Informationen über Nutzer.

**U1: *users* (POST):** Dieser Endpunkt dient zum Erstellen eines neuen Nutzers und sollte bereits existieren.
**U2: *users/:userId* (GET):** Dieser Endpunkt gibt Informationen zum Nutzer mit der übergebenen *User Id* zurück. Auch dieser Endpunkt sollte bereits existieren.

**UM: *users/me*:** Unter *users/me* befinden sich alle Endpunkte, die sich auf Informationen zum aktuell eingeloggten Nutzer beziehen.
**UMP: *users/me/plans*:** Unter *users/me/plans* befinden sich alle Endpunkte, die sich um den Plan bzw. die wählbaren Pläne eines Nutzers kümmern.
**UMP1: *users/me/plans* (GET):** Dieser Endpunkt gibt alle Pläne mit den dazugehörigen Informationen zurück, die für den aktuell angemeldeten Nutzer wählbar sind.
Hierbei gibt es eine kleine Besonderheit, denn dieser Endpunkt ist auch ansprechbar, sollte kein Nutzer angemeldet sein. Für diesen Fall werden alle allgemein wählbaren Pläne zurückgegeben.
**UMP2: *users/me/plans/:planId* (PATCH):** Über diesen Endpunkt lässt sich der aktuell gewählte Plan des eingeloggten Nutzers zu dem Plan mit der übergebenen *Plan Id* ändern.

**UMQ: *users/me/quota*:** Unter *users/me/quota* sind alle Endpunkte eingeordnet, welche Informationen zur Quota Nutzung des eingeloggten Nutzers liefern. Hierbei handelt es sich somit nur um GET-Endpunkte.
**UMQ1: *users/me/quota/run* (GET):** Dieser Endpunkt wird angesprochen, um zu überprüfen, ob der angemeldete Nutzer einen neuen Run starten kann. Er gibt ein Objekt zurück, aus welchem abgeleitet werden kann, ob der Nutzer weitere Runs starten kann oder ob sein Quota Limit bereits erreicht ist.
**UMQ2: *users/me/quota* (GET):** Dieser Endpunkt gibt ein Objekt zurück, welches Informationen zur Quota Nutzung des eingeloggten Nutzers enthält. Hierbei handelt es sich nur um die genutzten Punkte, den belegten Speicherplatz, sowie den abonnierten Plan. Diese Informationen werden beispielsweise für die Anzeige der Quota Informationen auf der Frontend Komponente **Quota Card** benötigt. Für andere Frontend Komponenten sind noch weitere Informationen zur Quota Nutzung nötig, welche zum Teil deutlich rechenaufwändiger zu bestimmen sind. Aufgrund dessen gibt es hierfür eigene Endpunkte, sodass dieser Rechenaufwand nur betrieben wird, wenn diese Informationen benötigt werden.
**UMQ3: *users/me/quota/details* (GET):** Dieser Endpunkt gibt ebenso ein Objekt mit Informationen zur Quota Nutzung des eingeloggten Nutzers zurück, jedoch deutlich detaillierter, wie sie für die Frontend Komponente **Punkte Diagramm** des **Quota Dashboards** nötig sind.

**UMQI: *users/me/quota/instances*:** Unter *users/me/quota/instances* befin-

den sich alle Endpunkte, welche Quota Informationen bezogen auf Instanzen zurückliefern.
**UMQI1: *users/me/quota/instances* (GET):** Dieser Endpunkt gibt Quota Informationen zu Instanzen zurück. Hierbei gibt es zwei Query Parameter. Der Query Parameter *orderBy* kann entweder den Wert *‚points‘* oder *‚storage‘* haben. Sortiert werden die zurückgegebenen Instanzen dann absteigend je nach Wert des *orderBy* Parameters nach der Punktenutzung bzw. der Speicherplatznutzung. Der Default Wert für diesen Query Parameter ist *‚storage‘*.
Der zweite Query Parameter ist *limit* und gibt an, zu wie vielen Instanzen die Quota Informationen zurückgegeben werden sollen. Ist er undefiniert, so werden zu allen Instanzen Quota Informationen zurückgegeben.
**UMQI2: *users/me/quota/instances/:instanceId/points* (GET):** Dieser Endpunkt gibt die Anzahl an Punkten zurück, die Runs der Instanz mit der übergebenen *Instance Id* in diesem Monat bereits benötigten.
**UMQI3: *users/me/quota/instances/:instanceId/storage* (GET):** Dieser Endpunkt gibt die Menge an Speicherplatz zurück, die die Resultate der Runs der Instanz mit der übergebenen *Instance Id* insgesamt belegen.

**R: *runs*:** Die Endpunkte unter Runs dienen zum Starten von neuen Runs, sowie der Verwaltung von Runs.
**R1: *runs* (POST):** Dieser Endpunkt dient zum Starten eines neuen Runs und sollte bereits existieren.
**R2: *runs/instances/:instanceId/runs* (GET):** Dieser Endpunkt gibt alle Runs, die der Instanz mit der übergebenen *Instance Id* zugeordnet sind, zurück.
**R3: *runs/instances/:instanceId/runs/:runId* (GET):** Dieser Endpunkt gibt den Run, mit der übergebenen *Run Id* zurück und wird für die Frontend Komponente **Quota Informationen für Runs** benötigt. Denn im Run, den dieser Endpunkt zurückgibt, sind auch die anzuzeigenden Quota Informationen enthalten.
**R4: *runs/instances/:instanceId/runs/:runId* (DELETE):** Dieser Endpunkt dient zur Löschung des Runs mit der mitgegebenen *Run Id*. Die Möglichkeit Runs zu löschen, ist zwingend notwendig, da ansonsten, die Speicherplatzbelegung des Nutzers immer weiter ansteigen würde, ohne dass diesem die Möglichkeit gegeben wird, Runs zu löschen, um wieder Speicherplatz zur Verfügung zu bekommen.

### 6.1.3 Komponenten

Wir nehmen an, dass bereits eine *Run Komponente* existiert, welche sich um das Starten der Runs, sowie die Verwaltung der Resultate derer kümmert. Die Verwaltung der Resultate oder andere Teilschritte können auch in andere oder eigene Komponenten ausgelagert werden. Verallgemeinert gehen wir jedoch davon aus, dass dies in einer einzelnen Komponente geschieht.

Zudem gehen wir davon aus, dass bereits eine Komponente zur Verwaltung des Codes, welcher in den Runs ausgeführt wird, existiert. Die Runs können entweder direkt mit einem Projekt verknüpft sein oder mit einer Instanz eines Projektes. In diesem Fall nehmen wir an, dass die Runs Instanzen zugeordnet sind. Die Komponente dazu nennen wir im folgenden *Instanzen Komponente*.
Wir gehen ebenso davon aus, dass bereits eine *Nutzer Komponente* existiert, welche sich um die Verwaltung der Nutzer kümmert.
Allgemein können die verschiedenen Komponenten auch in verschiedenen Services liegen, die per *REST API* miteinander kommunizieren. Ein Beispiel hierfür befindet sich im Abschnitt **Architektur für JValue**.
In dieser allgemeinen Beschreibung der Architektur nehmen wir jedoch an, dass alle Komponenten in einem Service liegen und es lediglich *Backend* und *Frontend* gibt.
Zu den bereits existierenden Komponenten fügen wir die *Quota Komponente* hinzu, welche sich um die Verwaltung der Quota Informationen der Nutzer und die Verwaltung der Pläne kümmert. Dazu gehört das Ausliefern der Informationen zur Quota Nutzung, das Ändern dieser Informationen bei Abschluss oder Löschung eines Runs, sowie das Ändern des aktuell ausgewählten Plans eines Nutzers und das Ausliefern von Informationen über die wählbaren Pläne.

## 6.2 Architektur bei JValue

Bei JValue ist die Architektur deutlich komplizierter aufgebaut, da wir hier unter anderem verschiedene Services haben, welche untereinander per *REST API* kommunizieren.
Ziel hierbei ist es jedoch, möglichst wenig Datenverkehr zwischen den Services zu haben.
Im Folgenden wollen wir die eben erarbeitete allgemeine Quota Architektur auf JValue anwenden.

### 6.2.1 Datenbankstruktur bei JValue

#### Speicherung der Nutzerdaten

Wie auch schon im allgemeinen Modell beschrieben, fügen wir zur User Tabelle die Spalte *plan* hinzu, in welcher die *Id* des Plans, den der jeweilige Nutzer abonniert hat, gespeichert wird. Daraus resultiert folgende *User* Tabelle:

| *id* | *username* | *password* | *salt* | *createdAt* | *updatedAt* | *plan* |
|---|---|---|---|---|---|---|

#### Speicherung der Quotadaten

Wie auch bereits im Kapitel zur allgemeinen Datenstruktur beschrieben, benötigen wir nun noch die beiden Tabellen zur Speicherung der Quota Nutzung.

**Punkte Quota** Die Tabelle zur Speicherung der *Punkte* Quota nennen wir *Computation Quota* und sie besteht aus den Spalten *id*, *usedPoints*, *month*, *year* und *userId*.

| *id* | *usedPoints* | *month* | *year* | *userId* |
|---|---|---|---|---|

**Speicherplatz Quota** Die Tabelle zur Speicherung der *Speicherplatz* Quota nennen wir *Storage Quota* und sie besteht aus den Spalten *id*, *usedStorage*, *autoDelete* und *userId*.

| *id* | *usedStorage* | *autoDelete* | *userId* |
|---|---|---|---|

**Pläne** Für die Speicherung der Pläne haben wir uns zunächst für die zweite Option entschieden, also der Speicherung der Pläne in einer Datei im Repository. Dies liegt vor allem daran, dass die Speicherung in einer Datenbanktabelle, wie bereits erwähnt, auch eine Frontend Oberfläche zum Bearbeiten der Pläne benötigen würde. Das wäre zum einen für diese Arbeit von zu großem Umfang, als auch zum aktuellen Zeitpunkt aufgrund fehlender Admin Account Funktionalität nicht umsetzbar.

#### Speicherung der Rundaten

Wie bereits im **Kapitel Quota Modell bei JValue** beschrieben, wollen wir in die Berechnung unseres Run Scores die CPU-Time und den genutzten Arbeitsspeicher miteinbeziehen. Wir benötigen somit 4 zusätzliche Spalten in der Run Tabelle für *CPU-Time*, *Speicherplatz*, *Punkte* und *Arbeitsspeicher*.

| *runId* | *instanceId* | ... | *deletedAt* | *cpuTime* | *storage* | *points* | *ram* |
|---|---|---|---|---|---|---|---|

### 6.2.2 REST API bei JValue

Wie bereits erwähnt, existiert bei JValue eine Aufteilung in verschiedene Services. Hiervon benötigen wir für die Umsetzung des Quota Systems das *Hub-Web*, das *Hub-Backend*, den *Pipeline-Service* und die *Runtime-Simple*.
Das Hub-Web benötigen wir zum Anzeigen der Quota Informationen und der Pläne.
Im *Hub-Backend* werden die Projekte, die Nutzer, sowie nun auch die Pläne verwaltet.

Im *Pipeline Service* findet die Verwaltung der Instanzen, der Runs und nun auch der Quota Informationen statt.
Die *Runtime-Simple* kümmert sich nur um die Ausführung der Runs, wobei nun auch die Resourcenverwendung gemessen wird. Sie sendet die Ergebnisse dann zurück an den *Pipeline-Service*.
Das *Hub-Web* kommuniziert lediglich mit dem *Hub-Backend*, welches auch als *API-Gateway* zum *Pipeline-Service* und dieser wiederum zur *Runtime-Simple* fungiert.
Im Folgenden werden wir nun die API-Endpunkte der einzelnen Services vorstellen, die wir zur Umsetzung von Quota bei JValue benötigen.
Zudem gibt es noch einige weitere bereits existierende Endpunkte, welche wir jedoch zur Umsetzung des Quota Systems nicht benötigen.

### Hub-Backend

Dadurch, dass das *Hub-Backend* als *API-Gateway* fungiert, lässt sich das Schaubild zur API aus dem allgemeinen Teil komplett auf das *Hub-Backend* übertragen. Zwar findet davon nicht die gesamte Logik im *Hub-Backend* statt, jedoch müssen dieselben API-Endpunkte hier aufgrund des *API-Gateways* dennoch existieren.

### Pipeline-Service

Im *Pipeline-Service* benötigen wir die Endpunkte, bei denen es um Informationen zur Quota Nutzung geht.
Also somit fast alle, die unter *users/me/quota* hängen. Hierbei werden wir das *users/me* vom Anfang des Pfades weglassen, da der *Pipeline-Service* kein Wissen über *User* hat. Er speichert nur bei den Instanzen und bei Quota deren *User Id*, welche er vom *Hub-Backend* mit übergeben bekommt.
Ebenso benötigen wir die Endpunkte, die unter *run* hängen.
Zusätzlich brauchen wir den Endpunkt, über welchen die *Runtime-Simple* mit dem *Pipeline-Service* kommuniziert. Hierzu dient der *results/:runId* POST Endpunkt, über welchen die Informationen zu abgeschlossenen Runs von der *Runtime-Simple* empfangen werden.
Daraus resultiert schließlich folgendes API-Schaubild des *Pipeline-Service*:

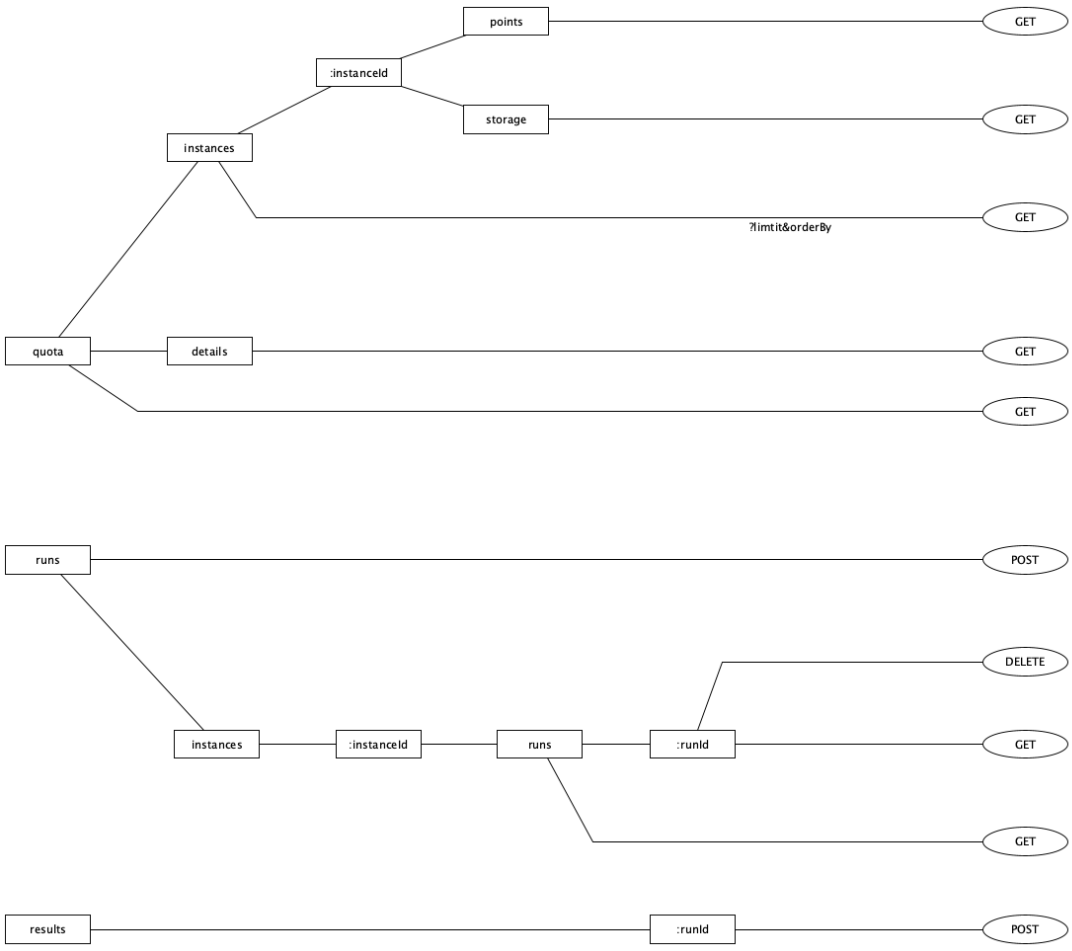


**Abbildung 6.2:** Pipeline-Service REST API

Die Funktionalität, der bereits im allgemeinen Kapitel zur REST API beschriebenen Endpunkte, bleibt die Gleiche.
Hinzu kommt lediglich der *result/:runId* POST Endpunkt, welcher die Informationen zu den abgeschlossenen Runs von der *Runtime-Simple* entgegennimmt, inklusive der Daten zum Run, wie beispielsweise auch der Quota Informationen. Diese werden anschließend an die später noch beschriebene *Run Komponente* weitergegeben, welche die Daten zum Run in der Datenbanktabelle *Run* speichert.

### Runtime-Simple

Die *Runtime-Simple* bietet lediglich zwei verschiedene POST-Endpunkte an, Beide zum Starten eines neuen Runs.


**Abbildung 6.3:** Runtime-Simple REST API

Der *runs/sync* Endpunkt startet einen Run, wartet daraufhin bis dieser Run fertig ausgeführt wurde und gibt dann Informationen dazu zurück. Dazu gehören beispielsweise Informationen zum Speicherort der Resultate der Runs, sowie zur Ressourcenverwendung des Runs. Dieser Endpunkt wird für *Quick-Runs*, wie beispielsweise bei der Ausführung im Editor verwendet.

### 6.2.3 Komponenten bei JValue

Wenn wir uns bei JValue die Komponenten anschauen, so gibt es hier deutlich mehr, welche zudem auf verschiedene Services verteilt sind, was das Ganze deutlich komplizierter macht.

#### Hub-Backend

Im *Hub-Backend* benötigen wir die *Runs Komponente*, welche sich um das Starten der Runs, das Löschen von Runs, sowie das Erhalten von Informationen zu den Runs kümmert.
Hierbei werden jedoch die meisten Daten aus dem *Pipeline-Service* geholt, was wiederum über die *Pipeline-Service* Komponente geschieht. Das *Hub-Backend* fungiert hierbei lediglich als *API-Gateway*.
Ebenso existiert bereits eine *User Komponente*, welche sich um die Verwaltung der Nutzer kümmert. Dies geschieht alles im *Hub-Backend*. Kein anderer Service hat Wissen über die Nutzer, sondern speichert lediglich teilweise die *User Id* des Nutzers.
Zusätzlich benötigen wir im *Hub-Backend* nun noch die neue Komponente Quota. Diese kümmert sich um das Erhalten von Informationen über die Quota Nutzung, sowie die Verwaltung der Pläne.
Die Verwaltung der Quota Informationen der Nutzer geschieht jedoch im *Pipeline-Service*, sodass das *Hub-Backend* hierbei wieder nur als *API-Gateway* dient. Die Pläne werden jedoch im *Hub-Backend* verwaltet, und die anderen Services haben kein Wissen über deren Existenz.

### Pipeline-Service

Der *Pipeline-Service* hat vier für uns wichtige Komponenten. Die *Run Komponente*, die *Instance Komponente*, die *Results Komponente* und die neu erstellte *Quota Komponente*.
Über die *Run Komponente* werden Runs gestartet, gelöscht, Informationen dazu aktualisiert, sowie bereitgestellt. Die *Runs Komponente* speichert die Informationen zu den Runs in der *Run* Tabelle in der Datenbank, und somit auch die Quota Informationen zu den jeweiligen Runs, also die CPU-Zeit, den genutzten Arbeitsspeicher, den belegten Speicherplatz der Resultate der Runs, sowie die Punktzahlen der Runs.
Die Ausführung der Runs geschieht dann jedoch in der *Runtime-Simple*.
Die *Instance Komponente* verwaltet die Instanzen und hat dafür eine entsprechende *Instance* Tabelle in der Datenbank, worin zum Beispiel auch der auszuführende Code der jeweiligen Instanz gespeichert wird.
In der neu erstellten *Quota Komponente*, werden die Quota Informationen eines Nutzers aktualisiert und abgefragt. Hierfür werden auch teilweise Methoden der anderen Komponenten benötigt.
Die *Results Komponente* nimmt Informationen zu abgeschlossenen Runs von der *Runtime-Simple* entgegen und übergibt diese der Run Komponente zur Persistierung dieser Informationen.

### Runtime-Simple

Die *Runtime-Simple* besteht lediglich aus einer *Run Komponente*, welche die Runs ausführt und deren Resultate dann zur Speicherung an den *File-Service* schickt. Die Informationen zum ausgeführten Run werden entweder zurückgeliefert oder per API Request bei Abschluss des Runs an die *Results Komponente* des *Pipeline-Service* geschickt.

# 7 Implementierung

Die Implementierung des Quota Modells kann je nach Programmiersprache, verwendeter Frameworks und vieler anderer Faktoren sehr unterschiedlich ausfallen. Aufgrund dessen werden wir keine allgemeine Implementierung vorstellen, sondern lediglich die Implementierung für JValue, welche als Referenzimplementierung für andere Projekte dienen kann.

## 7.1 Technologische Rahmenbedingungen bei JValue

Die Rahmenbedingung der Architektur können dem **Kapitel 6.2 Architektur bei JValue** entnommen werden.
Hier wollen wir nun noch knapp auf die restlichen gegebenen Rahmenbedingungen, wie verwendete Frameworks, Programmiersprache etc. eingehen.
Für das gesamte Projekt nutzen wir *TypeScript*[1], eine zu *JavaScript* kompilierende Programmiersprache mit statischer Typisierung.
Für das Frontend wird *React*[2] genutzt, eine komponentenbasierte *JavaScript* Bibliothek zum Erstellen von webbasierten Benutzeroberflächen. Für die meisten Frontend Komponenten wird entweder *react-md*[3] oder das *JValue Design System*[4] verwendet.
Für die Interaktion des *Frontends* mit den *Backends* via API nutzt das *Frontend* *RTK Query*[5], was zudem einen sehr nützlichen Cachingmechanismus mitbringt. Die API wird hierbei im *Frontend* automatisch anhand der *Controller* Klassen des *Hub-Backends* generiert. Hierfür dient der *openapi-generator*[6].
Für die restlichen Services wird das *NestJs*[7] Framework für *NodeJs*[8] verwendet,

[1] https://www.typescriptlang.org
[2] https://react.dev
[3] https://react-md.dev
[4] https://github.com/jvalue/hub/blob/main/libs/design-system/README.md
[5] https://redux-toolkit.js.org/rtk-query/overview
[6] https://openapi-generator.tech
[7] https://nestjs.com
[8] https://nodejs.org/

um die entsprechenden serverseitigen Services umzusetzen.
Hierbei besteht eine Komponente immer aus *Module*, *Controller* und *Service* Klasse, wobei das *Module* nur zum Importieren und Exportieren dient, und hierbei keinerlei Logik stattfindet. Der *Controller* implementiert die API-Endpunkte und ruft die entsprechende Methode der *Service* Klasse auf, in welcher dann die Logik stattfindet.

## 7.2 Hub-Web (Frontend)

Im Folgenden werden wir die Implementierung des Frontends vorstellen, welche wir anhand der vorgestellten Frontend Komponenten aus dem **Kapitel Design der Nutzeroberfläche** strukturieren wollen.
Wir werden hierbei jedoch nur grob die Implementierung vorstellen und nur auf besondere Stellen genauer eingehen.

### 7.2.1 Quota Card

Wie bereits beschrieben, ist die **Quota Card** eine Komponente auf der *Profile Page*, welche Informationen zur Quota Nutzung des eingeloggten Nutzers anzeigt und gleichzeitig als Button, welcher zum *Quota Dashboard* führt, dient.
Die *Profile Page* zeigt eine Liste von *Cards* an.
Dafür gibt es ein Array aus Elementen des Typs *LinkCardProps*, welche jeweils die Informationen zum Icon der *Card*, deren Titel, sowie den Link, zu welcher Seite die *Card* führen soll, angeben.
Zu diesem Array können wir unsere *Quota Card* jedoch nicht hinzufügen, da wir zusätzlich noch die Informationen zur Quota Nutzung anzeigen wollen. Aufgrund dessen erstellen wir eine separate *Quota Link Card* Komponente, welche zusätzlich zu den im Array enthaltenen *Cards* angezeigt wird.
Diese erhält die Quota Informationen durch API-Request an den *users/me/quota* GET-Endpunkt des *Hub-Backends*.
Die beiden Leisten, die die Quota Nutzung anzeigen, implementieren wir jeweils durch die *LinearProgress* Komponente von *react-md*.
Der Inhalt der *Quota Card* wird zudem an die Farbe des ausgewählten Plans des Nutzers angepasst. Dazu überprüfen wir, ob es sich beim Wert für die Farbe des übergebenen Plans um eine *HEX* Farbe im gültigen Format handelt. Sollte dies nicht der Fall sein, so wird ein standardmäßiger Wert für die Farbe des Plans verwendet.

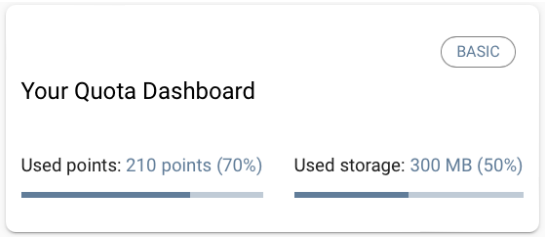


**Abbildung 7.1:** Quota Card

### 7.2.2 Quota Dashboard

Für das **Quota Dashboard** erstellen wir eine eigene Seite, die *Quota Dashboard Page*, welche über */quota-dashboard* erreichbar ist.
Sollte der Nutzer nicht eingeloggt sein, wird hier lediglich ein Hinweis angezeigt, dass das *Quota Dashboard* nur für angemeldete Nutzer verfügbar ist.
Die Quota Informationen für die *Quota Dashboard Page* bekommen wir per GET API-Request an *users/me/quota/details*. Hierbei reicht der *users/me/quota* Get-Endpunkt nicht aus, da dieser nicht die detaillierten Quota Informationen liefert, welche für die *Quota Diagramm* Komponente notwendig sind.
Da wir auch hier wieder die Farbe des Plans verwenden, überprüfen wir zunächst, ob es sich beim Wert für die Farbe des übergebenen Plans um eine *HEX* Farbe im gültigen Format handelt. Sollte dies nicht der Fall sein, so wird ein standardmäßiger Wert für die Farbe des Plans verwendet. Die im Kapitel Design Frontend vorgestellte *Auto Delete* Komponente des *Quota Dashboards* implementieren wir nicht, da die *Auto Delete* Funktion nicht implementiert wurde (siehe **Kapitel 9.1**).

#### Quota Diagramm

Für das **Punkte Diagramm** erstellen wir eine eigene Komponente, das *Quota Diagram Element*, welches durch die *Quota Dashboard Page* eingebunden wird und hierbei auch die benötigten Quota Informationen übergeben bekommt. Das *Quota Diagram Element* selbst sendet somit keinerlei API-Requests.
Für die Umsetzung des Diagramms verwenden wir *Apache ECharts*[9]. Um *Apache ECharts* für *React* verfügbar zu machen, wird die *ReactECharts* Komponente aus folgendem Workaround benötigt (Goyal, 2021).
Übergeben bekommt die Komponente ein Objekt des Typs *ReactEChartsProps*, über welches das Aussehen, sowie die Daten für das Diagramm definiert werden. Für die Daten wird ein 2-dimensionales Array benötigt. Die erste Dimension enthält hierbei die x-Werte. Die zweite Dimension ist ein Array aus den zugehörigen y-Werte der anzuzeigenden Punkte.
Die erste Dimension des Arrays steht in unserem Fall für die Tage des Monats,

[9]https://echarts.apache.org/en/index.html

die zweite Dimension für die y-Werte zum jeweiligen Tag. Wir wollen hierbei sowohl die kumulierte Punktenutzung des Anwenders bis zu diesem Tag anzeigen, als auch das Punktelimit des Plans, sofern dies kein unlimitierter Plan ist.
Zudem wollen wir die zukünftige Punktenutzung des Nutzers für den restlichen Monat approximieren. Aufgrund dessen wird die durchschnittliche bisherige Punktenutzung pro Tag dieses Monats berechnet. Dieser Betrag wird abgerundet und für die noch kommenden Tage des Monats dazu addiert, und so im Diagramm dargestellt. Die Schätzung der Nutzung für die restlichen Tage des Monats wollen wir zur besseren Anschaulichkeit in einer blassen hellblauen Farbe darstellen, wofür wir die *visualMap* Funktion von *Apache ECharts* zur Unterteilung des Graphen in zwei Abschnitte verwenden. Der erste Abschnitt besteht aus den Daten für die bisherigen Tage des Monats und wird in der Farbe des Plans des Nutzers gefärbt. Der zweite Abschnitt für die noch folgenden Tage, für die die Werte nur Schätzungen sind, wird in der blassen hellblauen Farbe gefärbt.

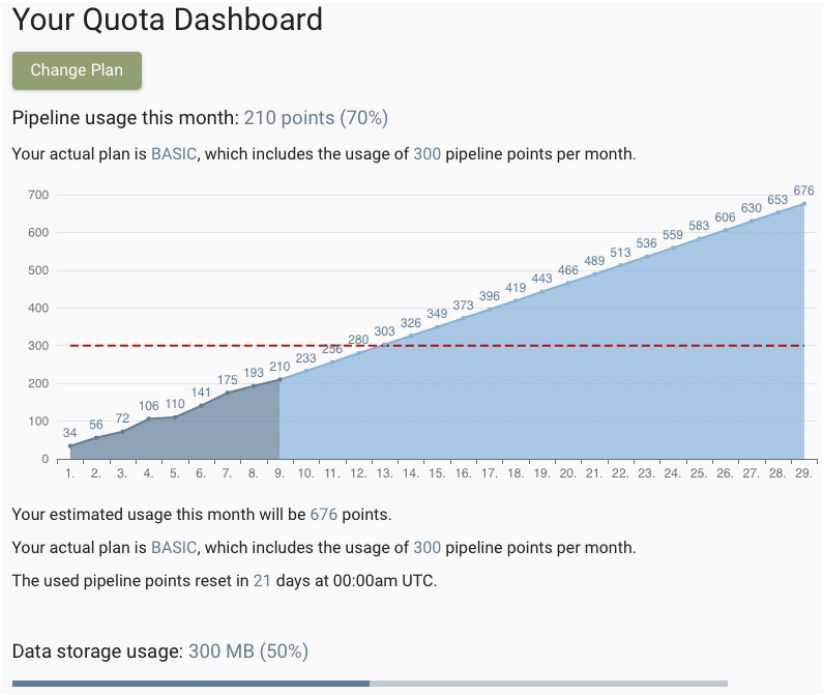


**Abbildung 7.2:** Quota Dashboard

### Speicherplatz Quota

Wie bereits im Kapitel Frontend Design beschrieben, zeigen wir zum Speicherplatz Quota lediglich die Speicherplatzbelegung sowohl absolut, als auch relativ zum zur Verfügung stehenden Speicherplatz an.
Anschaulich visualisiert wird dies durch eine Fortschrittsleiste, welche durch die *LinearProgress* Komponente von *react-md* implementiert wird.

### Instanzen mit höchster Punktenutzung bzw. Speicherplatzbelegung

Wir zeigen zunächst die Instanzen mit höchster Punktenutzung in diesem Monat und anschließend die Instanzen mit höchster Speicherplatzbelegung insgesamt an. Hierfür verwenden wir jeweils dieselbe Komponente, die *Top Instances Element* Komponente. Hierfür werden zunächst per API-Requests an *users/me/quota/instances?limit=5&orderBy=points* bzw. *users/me/quota/instances?limit=5&orderBy=storage* die Informationen zu den Instanzen mit höchster Punktenutzung in diesem Monat, bzw. höchster Speicherplatzbelegung insgesamt geholt.
Hierbei gibt die *5* im API-Request an, dass wir nur die Informationen für die *5* Instanzen mit der höchsten Nutzung haben wollen, falls vorhanden. Dieser Wert ist als Konstante gespeichert und kann jederzeit geändert werden, sodass mehr Instanzen angezeigt werden. Ebenso kann man den Query Parameter auch weglassen, sodass die Informationen zu allen Instanzen angezeigt werden.

**Top Instances Element** Diesem Element wird ein Array mit den Informationen zu den Instanzen, die Quota Informationen des Nutzers, sowie ein *boolean* Parameter, welcher angibt, ob es sich hierbei um die Instanzen mit höchster Punktenutzung oder höchster Speicherplatzbelegung handelt, übergeben.
Wir zeigen hierzu eine Tabelle an, in welcher jede Instanz eine eigene Zeile bekommt, und dort den Namen der Instanz, den Namen des zugehörigen Projektes, sowie die Speicherplatzbelegung und Punktenutzung der Instanz anzeigt. Die Speicherplatzbelegung und die Punktenutzung werden hierbei sowohl absolut, als auch relativ zur insgesamten Nutzung dieses Kunden angezeigt. Bei Klick auf den Namen der Instanz wird man zur *Pipeline Page* der Instanz geleitet.

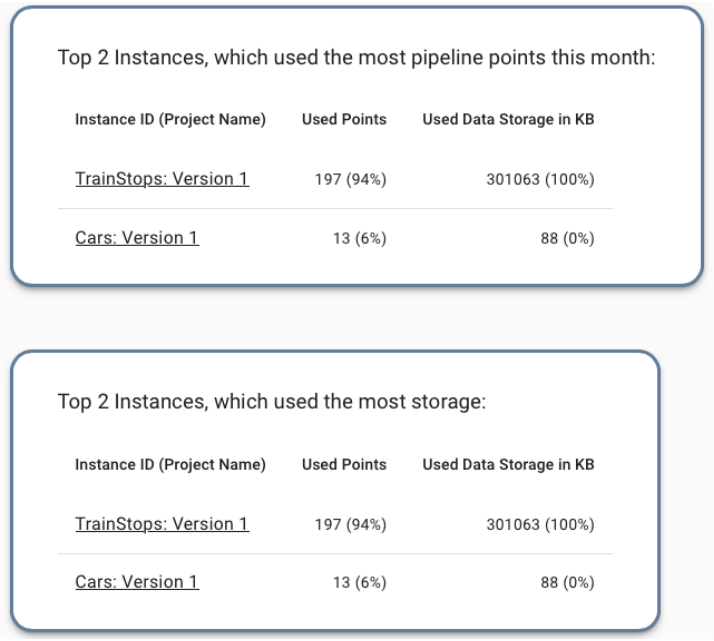


**Abbildung 7.3:** Top Instances Element

### 7.2.3 Change Plan Page

Für die *Change Plan Page* erstellen wir ebenso eine eigene Seite.
Auch hier überprüfen wir zunächst, ob die Farbe des aktuellen Plans des Nutzers eine Farbe im gültigen *HEX* Format ist, und verwenden sonst einen standardmäßigen Wert für die Farbe.
Für den Fall, dass ein Nutzer eingeloggt ist, zeigen wir oben Informationen zur aktuellen Quota Nutzung des Kunden, sowie den Namen seines aktuell gewählten Plans an.
Darunter zeigen wir die wählbaren Pläne in einem *Grid* mit drei Spalten an. Der *Query Wrapper* mappt die einzelnen Pläne jeweils auf ein *Plan Card Element*, welches wir gleich noch vorstellen werden. Übergeben wird diesem Element hierbei als *key* die *Id* des Plans und zusätzlich noch Informationen über den jeweiligen Plan, sowie ob es sich bei diesem um den aktuell ausgewählten Plan des Nutzers handelt.

#### Plan Card Element

Das *Plan Card Element* wird durch die *Change Plan Page* eingebunden. Wie bereits beschrieben, bekommt diese Komponente die Informationen über den darzustellenden Plan übergeben und benötigt somit keinen API-Request.
Da wir uns dafür entschieden haben, jedem Plan eine spezifische Farbe zuzuweisen, wird in dieser Komponente zunächst überprüft, ob es sich beim übergebenen

Wert für die Farbe des übergebenen Plans um eine *HEX* Farbe im gültigen Format handelt. Sollte dies nicht der Fall sein, so wird ein standardmäßiger Wert für die Farbe des Plans verwendet.
Wir verwenden nun die *Card* Komponente des *@jvalue/design-system* und versehen diese mit *style* Parametern, um die gewünschten abgerundeten Ecken darzustellen. Sollte es sich bei diesem Plan um den aktuell gewählten Plan des Nutzers handeln, so bekommt die *Card* einen grauen dicken Rand, welcher ebenso über die *style* Parameter definiert wird.
Die in der *Card* darzustellenden Informationen werden in der eben schon erwähnten Farbe abgebildet.
Unten auf der *Card* befindet sich noch ein Button, über welchen sich der Plan wählen lässt, sollte es sich dabei nicht bereits um den aktuell gewählten Plan des Nutzers handeln.
Bei Klick auf den *Button* öffnet sich ein *Dialog*. Der *Button* und der *Dialog* werden durch die *Change Plan Dialog* Komponente implementiert, welche an dieser Stelle eingebunden wird.

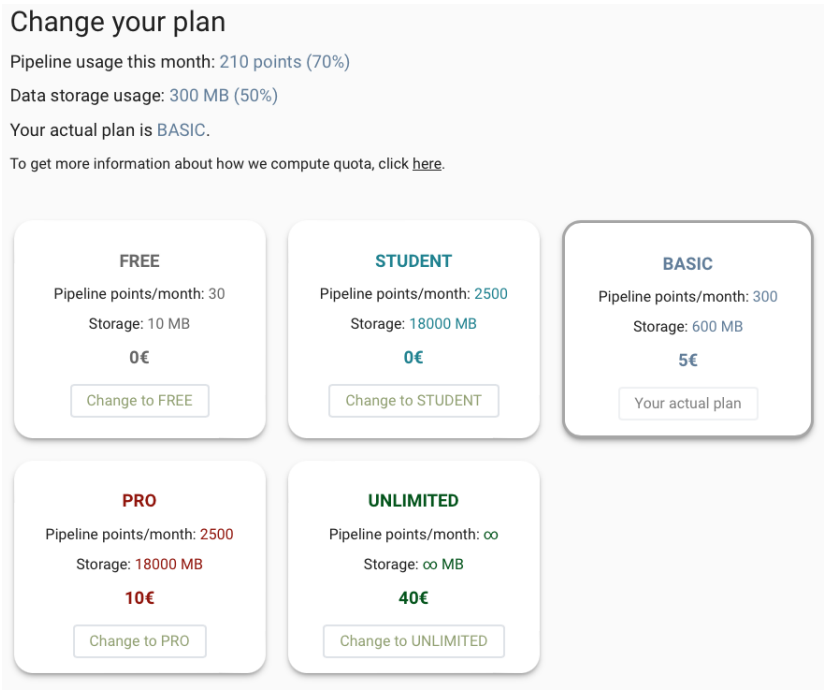


**Abbildung 7.4:** Change Plan Page

**Change Plan Dialog** Diese Komponente wird durch die *Plan Card Element* Komponente eingebunden und erhält darüber auch alle benötigten Informationen zum jeweiligen Plan, sowie, ob es sich hierbei um den aktuell ausgewählten Plan des Nutzers handelt.
Angezeigt wird zunächst nur ein *Button*, welchen der Nutzer betätigen muss, um

den Plan auszuwählen. Der *Button* ist jedoch deaktiviert, sollte es sich hierbei bereits um den aktuell abonnierten Plan handeln.
Bei Klick auf den *Button* öffnet sich ein *Dialog*, in welchem um eine Bestätigung zur Änderung des Plans gebeten wird. Hier kann der Nutzer entweder auf einen *Cancel Button* klicken, um abzubrechen und den Dialog zu schließen, oder auf den *Change Plan Button*, durch welchen ein API-Request ans *Hub-Backend* geschickt wird, um den Plan des Nutzers zu ändern.
Aktuell ist dieser *Dialog* noch ziemlich leer. In Zukunft könnte hier jedoch die Abwicklung der Zahlung stattfinden.
Zur Implementierung der *Dialog* Funktion wird die *useToggle* Funktion von *React* genutzt. Die Variable *visible* gibt an, ob der *Dialog* angezeigt wird. Durch Klick auf den *Change to <Name des Plans> Button* wird die Funktion *enable* aufgerufen und der *Dialog* wird sichtbar. Durch Klick auf den *Change Plan* oder *Cancel Button* im nun geöffneten Dialog wird die *disable* Funktion aufgerufen, durch welche der *Dialog* wieder unsichtbar wird.

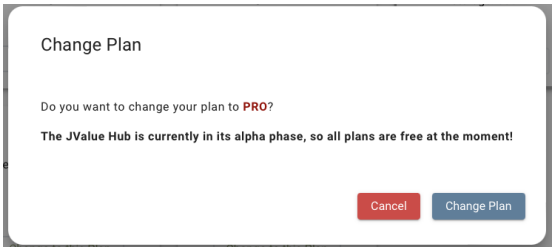


**Abbildung 7.5:** Change Plan Dialog

### 7.2.4 Quota Informationen auf der Pipeline Page

Die *Pipeline Page* zeigt Informationen der zugehörigen Instanz an und listet darunter alle ihr zugehörigen Runs mit Informationen dazu auf.
Oben auf der *Pipeline Page* befindet sich eine *Card* mit Informationen zur Instanz. Hier fügen wir auch die Informationen zur Speicherplatzbelegung der Resultate der Runs, sowie zur Punktenutzung dieses Monats ein.
Die Quota Informationen hierzu bekommt die Komponente per API-Request an die GET-Endpunkte
*users/me/quota/instances/:instanceId/points* und
*users/me/quota/instances/:instanceId/storage* des *Hub-Backends*.
Sollte ein Nutzer eingeloggt sein und es sich dabei um den Eigentümer der Instanz handeln, so wird, sollte der Nutzer noch kein Quota Limit erreicht haben, ein *Create Run Button* zum Starten von neuen Runs angeboten. Ein Sequenzdiagramm, welches die Interaktionen von der Erstellung einer Pipeline Instanz mit anschließendem Starten eines Runs bis hin zur Beendigung des Runs darstellt, befindet sich in **Anhang B**.

Beim Laden der Seite wird zunächst ein API-Request an *user/me/quota/run* geschickt, um zu erfahren, ob und wenn ja welches Quota Limit des Nutzers bereits erreicht ist.
Sollte ein Quota Limit bereits erreicht sein, so wird anstatt des *Create Run Buttons* ein *Upgrade Plan Button* angezeigt, welcher zur *Change Plan Page* leitet.
Hier müsste bei Einführung der *Auto Delete* Funktion noch überprüft werden, ob der Nutzer diese Funktion aktiviert hat, denn dann müsste der *Create Run Button* trotz erreichtem Speicherplatz Quota Limit angezeigt werden, da bei Start eines neuen Runs automatisch genug alte Runs zur Freigabe von Speicherplatz gelöscht würden.
Zusätzlich wird dem Nutzer noch die Möglichkeit zur Auswahl der Arbeitsspeichergröße für die Ausführung des Runs geboten. Dies dient als Art Ersatz für die Ressource RAM, welche in der Implementierung nicht berücksichtigt werden konnte. Mehr dazu siehe **Kapitel 7.3.4**. Dies wird durch die *Select* Komponente von *react-md* umgesetzt, welche die Auswahlmöglichkeiten aus der erstellten *ReservedRamOptions* Datei entnimmt, in welcher auch der zugehörige *default* Wert gespeichert wird.

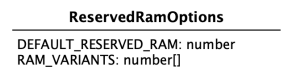


**Abbildung 7.6:** Reserved Ram Options

Bei Auswahl einer Arbeitsspeichergröße wird ein PATCH API-Request an *pipelines/:instanceId/reservedRamMB* zur Änderung der ausgewählten Arbeitsspeichergröße der Instanz gesendet. Zukünftige Runs werden dann mit der ausgewählten Arbeitsspeichergröße gestartet, die jedoch jederzeit änderbar ist.
Unter der *Card* befindet sich eine Tabelle, welche die Runs der Instanz anzeigt. Hierfür wird die *Table* Komponente von *react-md* verwendet.
Wie bereits beschrieben, wollen wir zu dieser Tabelle zwei weitere Spalten hinzufügen.
Eine Spalte, welche die *Pipeline-Size* und *Pipeline-Points* des Runs anzeigt und eine Spalte, welche die Speicherplatzbelegung der Resultate des Runs anzeigt.
Hierzu fügen wir zunächst zwei *Table-Cell* Elemente in die *Table-Row* des *Table-Headers* hinzu.
Der *Table-Body* besteht aus *RunRow* Elementen. Für jeden Run eine Instanz des *RunRow* Elements.

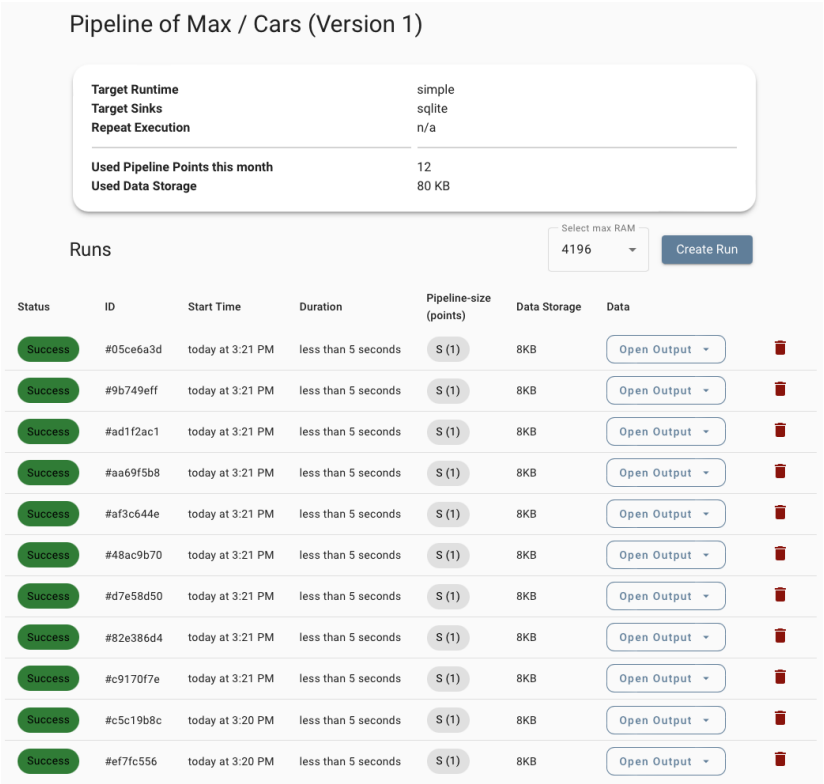


**Abbildung 7.7:** Pipeline Page

### RunRow Komponente

Übergeben wird diesem Element unter anderem die *Run Id* und die *Instance Id* des anzuzeigenden Runs. Mithilfe dieser Informationen wird ein GET API-Request an *runs/instances/:instanceId/runs/:runId* gesendet, wodurch wir die Informationen zum Run bekommen.
Wir fügen zu den bereits existierenden *Table-Cells* zwei weitere hinzu. Eine, welche die Speicherplatzbelegung des jeweiligen Runs anzeigt und eine, welche die Komponente *ResourcesTooltip* einbindet. Diese besteht aus einer *Chip* Komponente von *react-md*, in welcher die *Pipeline-Size*, sowie die *Pipeline-Points* dieses Runs angezeigt werden, und einer *Tooltip* Komponente, welche die Informationen zu Ressourcennutzung durch Hovern über die Tabellenzelle anzeigt.
Zudem fügen wir noch einen *Button* zum Löschen des Runs ein, bei dessen Nutzung ein DELETE API-Request an *runs/instances/:instanceId/runs/:runId* geschickt wird.

## 7.3 Hub-Backend

Das *Hub-Backend* kümmert sich unter anderem um die Verwaltung von Nutzern und Projekten. Zudem dient es als API-Gateway zum *Pipeline-Service*.

### 7.3.1 Quota Komponente

Wie bereits im Kapitel Architektur beschrieben, findet die Quota Verwaltung im *Pipeline-Service* statt und lediglich die Verwaltung der Pläne im *Hub-Backend*.

#### Quota Controller

| **ID** | Endpunkt (Typ) | Zugehörige Methode des *Quota Service* |
|---|---|---|
| **H-QC-E1** | *users/me/quota (GET)* | *getBasicQuotaInformation* |
| **H-QC-E2** | *users/me/quota/details (GET)* | *getDetailedQuotaInformation* |
| **H-QC-E3** | *users/me/quota/run (GET)* | *getQuotaReachedInformation* |
| **H-QC-E4** | *users/me/plans (GET)* | *getAllAvailablePlans* oder *getAllAvailablePlansForUser* |
| **H-QC-E5** | *users/me/plans/:plan (PATCH)* | *changePlan* |
| **H-QC-E6** | *users/me/instances?limit=:limit &orderBy=:orderBy (GET)* | *getInstances-QuotaInformation* |
| **H-QC-E7** | *users/instances/:instanceId/points (GET)* | *getPointsForInstance* |
| **H-QC-E8** | *users/instances/:instanceId/storage (GET)* | *getStorageForInstance* |

**Tabelle 7.1:** Hub-Backend Quota Controller

**H-QC-E4:** Eine Besonderheit stellt hierbei der GET *users/me/plans* Endpunkt dar. Dieser kann aufgerufen werden, sowohl falls ein Nutzer eingeloggt ist, als auch wenn kein Nutzer eingeloggt ist. Über die Hilfsmethode *extractUserFrom-Request* des Controllers wird hierfür der *User* aus dem *Request* bestimmt, falls existent. Die Hilfsmethode gibt entweder den *User* zurück oder *null*.
Sollte der Rückgabewert *null* sein, so wird die Methode *getAllAvailablePlans* des *Quota Service* aufgerufen, welche alle allgemein verfügbaren Pläne zurückgibt.
Sollte jedoch der Rückgabewert ein *User* sein, so wird mit dessen *User Id* die Methode *getAllAvailablePlansForUser* des *Quota Service* aufgerufen, welche alle für den übergebenen *User* verfügbaren Pläne zurückgibt.
Der Rückgabewert dieser zwei Methoden, kann durchaus unterschiedlich sein, falls der Nutzer beispielsweise noch einen Altvertrag hat, welcher nicht mehr wählbar ist.

Der Rückgabewert der aufgerufenen Methode des *Quota Service* wird anschließend zurückgesendet.

### Quota Service

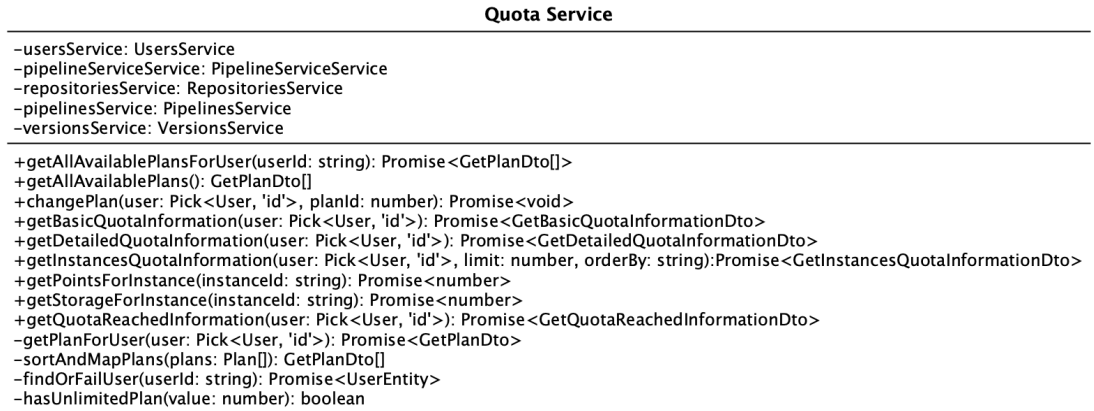


**Abbildung 7.8:** Hub-Backend Quota Service Klassendiagramm

Zunächst bekommen wir als Attribute per *Dependency Injection* die Instanz des *Users Service*, des *PipelineService Service*, des *Repositories Service*, des *Pipelines Service* und des *Versions Service*.
Der *PipelineService Service* bietet die Schnittstelle zum *Pipeline-Service* und ist daher wichtig, da die Verwaltung der Quota Informationen dort stattfindet. Viele der im Folgenden vorgestellten Methoden rufen daher lediglich die gleichnamigen Methoden des *PipelineService Service* auf, welche die entsprechenden API-Endpunkte im *Pipeline-Service* ansprechen. Diese Methoden werden in der folgenden Tabelle durch die eingeklammerte Kurzbeschreibung dargestellt.

| ID | Methode | Kurzbeschreibung |
|---|---|---|
| **H-QS-M1** | *getAvailable-PlansForUser* | Liefert alle verfügbaren Pläne des Nutzers zurück |
| **H-QS-M2** | *getAllAvailablePlans* | Liefert alle allgemein verfügbaren Pläne zurück |
| **H-QS-M3** | *changePlan* | Ändert den Plan des Nutzers auf den Plan mit der übergebenen *Plan Id* |
| **H-QS-M4** | *getBasic-QuotaInformation* | Liefert Informationen zur Quota Nutzung des Nutzers zurück |
| **H-QS-M5** | *getDetailed-QuotaInformation* | Liefert detaillierte Informationen zu Quota Nutzung des Nutzers zurück |
| **H-QS-M6** | *getInstances-QuotaInformation* | Liefert Informationen zur Quota Nutzung der Instanzen zurück |
| **H-QS-M7** | *getPointsForInstance* | (Gibt die Punktenutzung des aktuellen Monats zur übergebenen Instanz zurück) |
| **H-QS-M8** | *getStorageForInstance* | (Gibt die Speicherplatzbelegung der übergebenen Instanz zurück) |
| **H-QS-M9** | *getQuota-ReachedInformation* | Gibt Informationen darüber zurück, ob der Nutzer bereits ein Quota Limit erreicht hat |
| **H-QS-M10** | *getPlanForUser* | Gibt den Plan des Nutzers zurück |
| **H-QS-M11** | *sortAndMapPlans* | Sortiert die übergebenen Pläne nach Preis und mappt die Pläne auf die für das Frontend wichtigen Werte |
| **H-QS-M12** | *findOrFailUser* | Gibt Informationen zum übergebenen Nutzer zurück, falls dieser existiert oder wirft eine *NotFoundException*. Wird von allen Methoden, die einen Nutzer übergeben bekommen verwendet |
| **H-QS-M13** | *hasUnlimitedPlan* | Überprüft, ob es sich beim übergebenen Wert um einen Wert eines unlimitierten Planes handelt, also ob der Wert -1 ist |

**Tabelle 7.2:** Hub-Backend Quota Service

**H-QS-M1:** Wie bereits im Kapitel Architektur beschrieben, sind die Pläne als *Append-Only Liste* in einer gesonderten Datei gespeichert und haben einen *available* Parameter, welcher angibt, ob der jeweilige Plan aktuell wählbar ist. Das heißt also, die Liste soll nur erweitert und keinerlei Pläne gelöscht werden. Das Löschen von Plänen wird quasi durch Setzen des *available* Parameters auf *false* ersetzt. Diese Methode filtert alle Pläne heraus, die entweder verfügbar oder gleich dem

aktuell gewählten Plan des Nutzers sind und gibt diese anschließend zurück.
Mit dieser Liste wird dann die Hilfsmethode *sortAndMapPlans* aufgerufen und deren Rückgabewert, ein Array aus *GetPlanDto* Objekten, zurückgegeben.

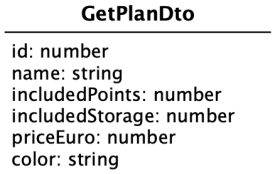


**Abbildung 7.9:** Get Plan Data transfer object (DTO)

**H-QS-M2:** Die Methode *getAllAvailablePlans* ist sehr ähnlich zur *getAllAvailablePlansForUser* Methode, mit dem einzigen Unterschied, dass diese für den Fall genutzt wird, dass kein Nutzer angemeldet ist, und somit nur die Pläne zurückgibt, deren *available* Parameter den Wert *true* hat. Diese Methode nutzt ebenfalls die Hilfsmethode *sortAndMapPlans*.
**H-QS-M3:** Zunächst wird überprüft, ob ein wählbarer Plan mit der übergebenen *Plan Id* existiert.
Falls dies der Fall ist, wird die *changePlan* Methode des *Users Service* aufgerufen, ansonsten eine *NotFoundException* geworfen. Eine Überprüfung, ob der übergebene Nutzer existiert, findet hierbei erst im *Users Service* statt.
**H-QS-M4:** Diese Methode gibt ein Objekt des Typs *GetBasicQuotaInformationDto* zurück, welches wie folgt aufgebaut ist:

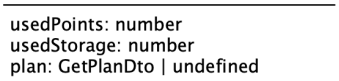


**Abbildung 7.10:** Get Basic Quota Information DTO

Die Informationen für die Parameter *usedPoints* und *usedStorage* werden per Methodenaufruf der gleichnamigen Methode des *PipelineService Service* vom *Pipeline-Service* geholt. Für die Informationen zum Plan des Nutzers wird die Hilfsmethode *getPlanForUser* aufgerufen.
**H-QS-M5:** Diese Methode gibt ein Objekt des Typs *GetDetailedQuotaInformationDto* zurück, welches wie folgt aufgebaut ist:

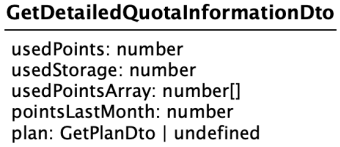


**Abbildung 7.11:** Get Detailed Quota Information DTO

Die Informationen für die ersten vier Parameter werden per Aufruf der entsprechenden Methode des *PipelineService Service* und somit vom *Pipeline-Service*

geholt.
Für die Informationen zum Plan des Nutzers wird hier ebenfalls die Hilfsmethode *getPlanForUser* aufgerufen.
**H-QS-M6:** Die Methode *getInstancesQuotaInformation* bekommt drei Parameter übergeben, die *User Id* des Nutzers für den die Informationen zurückgegeben werden, sowie zwei optionale Parameter.
Der optionale Parameter *limit* bestimmt die Anzahl an Instanzen, für die die Quota Informationen zurückgegeben werden sollen.
Der optionale Parameter *orderBy* kann entweder den Wert *points* oder *storage* haben.
Im Falle von *points* werden die Quota Informationen für die Instanzen mit der höchsten Punktenutzung zurückgegeben, bei *storage*, die mit der höchsten Speicherplatzbelegung. Hierbei wird ein Array der Länge *limit* mit dem Datentyp *GetInstancesQuotaInformationDto* zurückgegeben.

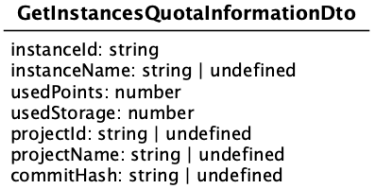


**Abbildung 7.12:** Get Instances Quota Information DTO

Hierzu rufen wir die *getInstancesQuotaInformation* Methode des *PipelineService Service* auf und übergeben hierbei alle drei Parameter, woraufhin wir ein Array von *GetInstancesQuotaInformationDto* zurückbekommen, welche wir nun noch mit den optionalen Parametern bestücken müssen.
Zusätzlich zu den Informationen aus dem *Pipeline-Service* benötigen wir noch die Informationen, die für den Link zur *Pipeline Page* notwendig sind.
Dafür benötigen wir zum einen die *Project Id* des zugehörigen Projekts, zum anderen den *Commit Hash* zu dieser Instanz, sowie ebenso noch den Namen des zugehörigen Projektes, sowie den Versionsnamen der Instanz.
Hierzu rufen wir zunächst die *getPipelineInformation* Methode des *Pipelines Service* mit der *Instance Id* auf, um aus dem Rückgabewert den *Commit Hash*, sowie die *Repository Id* zu extrahieren. Durch den Aufruf der *findOne* Methode des *Repositories Service* mit der *Repository Id* können wir nun noch aus deren Rückgabewert die *Project Id*, sowie den *Project Name* auslesen. Zuletzt wird noch die *find* Methode des *Versions Service* mit der *Repository Id* und dem *Commit Hash* aufgerufen, aus deren Rückgabewert wir nun noch den Namen der Version dieser Instanz erhalten.
**H-QS-M9:** Die Methode *getQuotaReachedInformation* gibt für den übergebenen *User* die *GetQuotaReachedInformationDto* zurück.

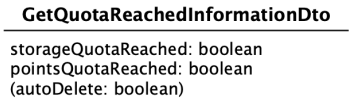


**Abbildung 7.13:** Get Quota Reached Information DTO

Hierfür kann ein *SOFT_LIMIT* Parameter hardgecoded definiert werden, durch welchen angegeben wird, um wie viel Prozent die Nutzer ihre Quota Limits überschreiten dürfen.
Dabei wird überprüft, ob der belegte Speicherplatz den im Plan enthaltenen um mehr als das Soft Limit übersteigt. Falls ja, wird der Parameter *storageQuotaReached* auf *true* gesetzt.
Selbiges wird für die genutzten Punkte durchgeführt, und dementsprechend der Parameter *pointsQuotaReached* gesetzt.
Die Informationen für die Überprüfungen werden durch Aufruf der *getBasicQuotaInformation* Methode erlangt.
Bei den Überprüfungen muss jeweils darauf geachtet werden, ob es sich beim Plan des Nutzers um einen unlimitierten Plan handelt. Dafür wird die Methode *hasUnlimitedPlan* mit dem jeweiligen Punkte- bzw. Speicherplatzwert des Plans aufgerufen.
Für die *Auto Delete* Funktion würden wir hierzu einen zusätzlichen Parameter benötigen, welcher angibt, ob der jeweilige Nutzer die *Auto Delete* Funktion aktiviert hat.

### 7.3.2 Users Komponente

Die *User Komponente* existiert bereits und muss lediglich um Funktionalität erweitert werden.

#### User Entity

Zuerst müssen wir das *User* Entity um den Parameter *plan* erweitern, welcher die *Plan Id* des gewählten Plans speichert und per default gleich der *Plan Id* des kostenlosen Plans ist. Dieser Parameter darf nicht *null* sein, da jedem Nutzer ein Plan zugeordnet sein muss.

#### Users Service

Weitere Änderungen sind nur in der *Users Service* Klasse notwendig. Zusätzlich zu den bereits existierenden Methoden wird hier noch eine Methode zum Ändern des Plans eines Nutzers benötigt. Zudem muss die Methode zum Erstellen eines neuen Nutzers erweitert werden.
Folgendes Klassendiagramm zeigt lediglich die für unser Quota System genutzten Methoden des *Users Service*:

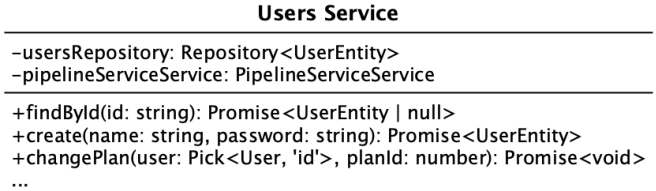


**Abbildung 7.14:** Hub-Backend Users Service Klassendiagramm

Als globale Variablen benötigt der *Users Service* das *Users Repository* zur Speicherung und zumAbruf der Daten zu den Nutzern, sowie nun neu die Instanz des *PipelineService Service* zur Kommunikation mit dem *Pipeline-Service*.

| **ID** | Methode | Kurzbeschreibung |
|---|---|---|
| **H-US-M1** | *findById* | Liefert den Nutzer zur übergebenen *User Id* oder null zurück |
| **H-US-M2** | *create* | Erstellt einen neuen Nutzer |
| **H-US-M3** | *changePlan* | Ändert den Plan des übergebenen Nutzers auf den mit der übergebenen *Plan Id* |

**Tabelle 7.3:** Hub-Backend Users Service

**H-US-M2:** Die Implementierung hierfür existiert bereits, muss jedoch erweitert werden. Um im *Pipeline-Service* bei gleichzeitigem Start mehrerer Runs eines neuen Nutzers eine *Race Condition* zu verhindern, rufen wir beim Erstellen eines Nutzers die *createEmptyStorageQuotaForUser* Methode des *PipelineService Service* auf, wodurch im *Pipeline-Service* ein leeres *Storage Quota Entity* für diesen Nutzer angelegt wird.
**H-US-M3:** Die Methode *changePlan* wird durch den *Quota Service* aufgerufen, sobald ein API-Request an den entsprechenden Endpunkt im *Quota Controller* geschickt wird. Die Methode überprüft zunächst, ob der übergebene Nutzer wirklich existiert. Falls ja, wird die *Plan Id* im Eintrag des Nutzers in der Datenbank auf die übergebene *Plan Id* geändert.
Die Überprüfung, ob ein Plan mit dieser *Plan Id* überhaupt existiert und auch wählbar ist, findet bereits in der *changePlan* Methode des *Quota Service* statt.

### 7.3.3 Runs Komponente

In der *Runs Komponente* müssen wir die *Delete* Funktion für Runs implementieren. Denn diese existiert noch nicht und ohne die Möglichkeit, Runs löschen zu können, würde ein Quota Limit auf belegten Speicherplatz keinen Sinn machen.

#### Runs Controller

Dem *Runs Controller* fügen wir hierfür den folgenden Endpunkt hinzu:

| ID | Endpunkt (Typ) | Zugehörige Methode des *Runs Service* |
|---|---|---|
| **H-RC-E1** | *instances/:instanceId/runs/:runId (DELETE)* | *deleteRun* |

**Tabelle 7.4:** Hub-Backend Runs Controller

### Runs Service

| ID | Methode | Kurzbeschreibung |
|---|---|---|
| **H-RS-M1** | *deleteRun* | *Soft Delete* des Runs |

**Tabelle 7.5:** Hub-Backend Runs Service

**H-RS-M1:** Die *deleteRun* Methode des *Runs Service* bekommt zum einen eine *GetRunDto* übergeben, welche die *Instance Id* und die *Run Id* des zu löschenden Runs enthält, zum anderen die *User Id* des Nutzers, der diesen Run löschen möchte. Zurückgegeben wird der gelöschte Run.
Zunächst werden hierfür die Informationen zur Pipeline Instanz durch Methodenaufruf der *getPipelineInformation* Methode des *PipelineService Service* geholt, woraufhin überprüft wird, ob die *User Id* des *Pipeline Owners* mit der *User Id* des Nutzers, der den Run löschen möchte, übereinstimmt. Falls nicht, wird eine *ForbiddenException* geworfen, da nur *Pipeline Owner* Runs löschen dürfen. Hierzu sollte es jedoch eigentlich gar nicht erst kommen, da der Button zum Löschen eines Runs im Frontend nur berechtigten Nutzern angezeigt wird.
Sollte der Nutzer berechtigt sein den Run zu löschen, so löschen wir den Run per *Soft Delete* aus dem *Run Information Repository* und rufen zudem die *deleteRun* Methode des *PipelineService Service* auf, wodurch der Run im *Pipeline-Service* per *Soft-Delete* gelöscht wird.

### Quota-Reached-Guard

Zum Starten eines Runs gibt es bereits einen API-Endpunkt im *Runs Controller* und eine zugehörige Methode im *Runs Service*.
Den API-Endpunkt im *Runs Controller* wollen wir jedoch um einen *Guard* erweitern, welcher den *HTTP Response Code 402 (Payment Required)* zurückliefert, sollte mindestens ein Quota Limit des Nutzers bereits erreicht sein. Denn dann soll dieser Nutzer keine Runs mehr starten können.
Hierfür implementieren wir den *Quota Reached Guard*, welcher per *Annotation* an den entsprechenden API-Endpunkt gebunden wird.
Der *Quota Reached Guard* extrahiert zunächst den *User* aus dem *Request* und ruft damit die *getQuotaReached* Methode des *Quota Service* auf.
Sollte nun mindestens einer der beiden Parameter des Rückgabewerts *true* sein,

so ist mindestens ein Quota Limit bereits erreicht. Dann kann kein weiterer Run mehr gestartet werden, weshalb auf den API-Request die Antwort *402 Payment Required* geschickt wird.
Dieser *HTTP Response Code* bedeutet „Zum Abrufen dieser Ressource ist eine Zahlung erforderlich“ (https://http-status-code.de/402/).
Wir haben uns für diesen *HTTP Response Code* entschieden, da wir ihn zurückliefern, falls ein Quota Limit des Nutzers erreicht ist und dieser somit keine weiteren Runs mehr starten kann. Lösen kann der Nutzer dieses Problem, indem er durch Erhöhung seines Plans und damit verbundener Zahlung sein Quota Limit erhöht.
Sollten jedoch beide Parameter *false* sein, so geben wir *true* zurück und der Aufruf der *initiateRun* Methode des *Runs Service* wird dadurch erlaubt.
Für die *Auto Delete* Funktion müssten wir hier bei erreichtem Storage Quota Limit zunächst noch überprüfen, ob der Nutzer die *Auto Delete* Funktion aktiviert hat, denn dann darf er dennoch Runs starten, obwohl sein Storage Quota Limit bereits erreicht ist.

### 7.3.4 Pipelines Komponente

Die Messung der Arbeitsspeichernutzung hat sich als problematisch herausgestellt, da diese vom Inneren des Programms heraus nur sehr unpräzise messbar ist. Dies liegt an der Ausführung des *Garbage Collectors*, welche nicht performant steuerbar ist, woraus unpräzise Werte für die Arbeitsspeichernutzung resultieren würden.
Aufgrund dessen haben wir uns dazu entschieden, die Ressourcenkomponente RAM abzuändern.
Der Nutzer bekommt nun die Möglichkeit zu wählen, welche Arbeitsspeichergröße er dem Run zur Ausführung zur Verfügung stellen möchte. Die gewählte Arbeitsspeichergröße wird mit der Dauer des Runs multipliziert und ergibt so die RAM Komponente für die Formel zur Ermittlung der Punkte des Runs.
Diese Berechnung findet jedoch in der *Runtime-Simple* statt.
Die Ressource RAM wird so auch bei *AWS-Lambda* (AWS Team, n. d. c) und *Google Dataflow* (Google Dataflow Team, n. d.) berechnet, sodass dies auch schon einigen Entwicklern bekannt sein sollte.
Hierbei verschiedene Arbeitsspeichergrößen zur Verfügung zu stellen, wäre im Rahmen der Arbeit nicht umsetzbar gewesen, weshalb es zum aktuellen Zeitpunkt lediglich die Arbeitsspeichergröße *4 GB* zur Auswahl gibt.

#### Pipelines Controller

Aufgrund dieser Alternativlösung für die Ressource RAM und dem damit verbundenen Parameter *ReservedRamMB* der Instanzen, welcher die gewählte Arbeitsspeichergröße speichert, benötigen wir nun einen API-Endpunkt, um den *ReservedRamMB* Parameter zu ändern.

| ID | Endpunkt (Typ) | Zugehörige Methode des *Pipelines Service* |
|---|---|---|
| **H-PC-E1** | *pipelines/:instanceId/reservedRamMB (PATCH)* | *updateReservedRamMB* |

**Tabelle 7.6:** Hub-Backend Pipelines Controller

**Pipelines Service**

| ID | Methode | Kurzbeschreibung |
|---|---|---|
| **H-PS-M1** | *updateReservedRamMB* | Ruft die updateInstance Methode des *PipelineService Service* auf, welche den entsprechenden API-Request an den *Pipeline-Service* schickt |

**Tabelle 7.7:** Hub-Backend Pipelines Service

### 7.3.5 PipelineService Komponente

Die *PipelineService Komponente* dient zur Kommunikation des *Hub-Backends* mit dem *Pipeline-Service*.
Diese Kommunikation findet nur *unidirektional* statt, das heißt also, das *Hub-Backend* schickt zwar API-Requests an den *Pipeline-Service*, jedoch nicht umgekehrt.
Hierbei mussten wir lediglich in der *PipelineService Service* Klasse Änderungen vornehmen.

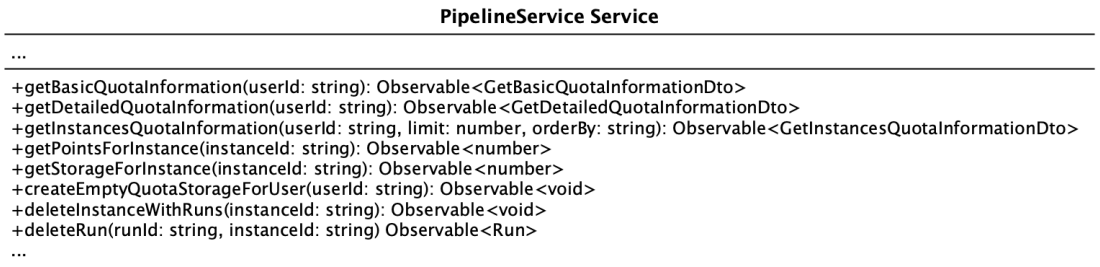


**Abbildung 7.15:** Hub-Backend PipelineService Service Klassendiagramm

Zu den bestehenden Methoden werden die im Klassendiagramm dargestellten hinzugefügt, welche jeweils den entsprechenden API-Endpunkt im *Pipeline-Service* ansprechen und die dafür notwendigen Parameter übergeben bekommen.

## 7.4 Pipeline-Service

Der *Pipeline-Service* kümmert sich um die Verwaltung der Instanzen, der Runs, sowie nun auch der Quota Informationen.

### 7.4.1 Quota Komponente

#### Quota Entities

Wir benötigen hier zunächst noch die beiden Quota Entities der zwei Datenbanktabellen, das *Computation Quota Entity* und das *Storage Quota Entity*.
Beide enthalten, die im Kapitel Architektur, beschriebenen Parameter.

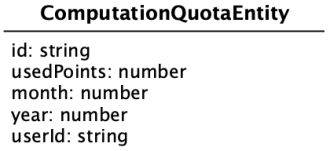


**Abbildung 7.16:** Computation Quota Entity

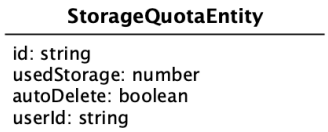


**Abbildung 7.17:** Storage Quota Entity

#### Quota Controller

| **ID** | Endpunkt (Typ) | Zugehörige Methode des *Quota Service* |
|---|---|---|
| **P-QC-E1** | *quota/:userId (GET)* | *getBasic-QuotaInformation* |
| **P-QC-E2** | *quota/:userId/details (GET)* | *getDetailed-QuotaInformation* |
| **P-QC-E3** | *quota/:userId/instances?limit=:limit &orderBy=:orderBy (GET)* | *getInstances-QuotaInformation* |
| **P-QC-E4** | *quota/instances/:instanceId/points (GET)* | *getPointsForInstance* |
| **P-QC-E5** | *quota/instances/:instanceId/storage (GET)* | *getStorageForInstance* |
| **P-QC-E6** | *quota/:userId (POST)* | *createEmpty-StorageQuotaForUser* |

**Tabelle 7.8:** Pipeline-Service Quota Controller

### Quota Service

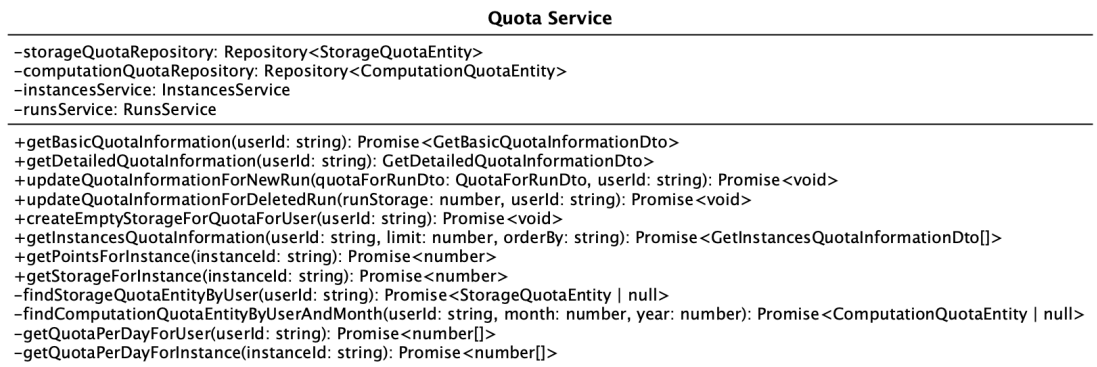


**Abbildung 7.18:** Pipeline-Service Quota Service Klassendiagramm

Der *Quota Service* benötigt zunächst die beiden Repositories als globale Variablen zum Zugriff auf die zwei Datenbanktabellen. Zusätzlich bekommen wir per *Dependency Injection* die Instanz des *Instances Service* und des *Runs Service*.

| ID | Methode | Kurzbeschreibung |
|---|---|---|
| **P-QS-M1** | *getBasicQuotaInformation* | Liefert Informationen zur Quota Nutzung des Nutzers zurück |
| **P-QS-M2** | *getDetailed-QuotaInformation* | Liefert detaillierte Informationen zur Quota Nutzung des Nutzers zurück |
| **P-QS-M3** | *updateQuota-InformationForNewRun* | Aktualisiert die Quota Informationen des Nutzers nach Abschluss seines Runs |
| **P-QS-M4** | *updateQuotaInformation-ForDeletedRun* | Aktualisiert die Quota Informationen des Nutzers nach Löschung seines Runs |
| **P-QS-M5** | *createEmpty-StorageQuotaForUser* | Erstellt einen Datenbankeintrag in der Storage Quota Tabelle für den Nutzer, was zur Verhinderung einer Race Condition dient |
| **P-QS-M6** | *getInstances-QuotaInformation* | Liefert Informationen zur Quota Nutzung der Instanzen des Nutzers zurück |
| **P-QS-M7** | *getPointsForInstance* | Gibt die Punktenutzung des aktuellen Monats zur übergebenen Instanz zurück |
| **P-QS-M8** | *getStorageForInstance* | Gibt die Speicherplatzbelegung der übergebenen Instanz zurück |
| **P-QS-M9** | *findStorageQuota-EntityByUser* | Liefert das Storage Quota Entity des Nutzers zurück oder null |
| **P-QS-M10** | *findComputationQuota-EntityByUserAndMonth* | Liefert das Computation Quota Entity des Nutzers für den angegebenen Monat zurück oder null |
| **P-QS-M11** | *getQuotaPerDay-ForUserAndMonth* | Liefert die tagesgenaue Quota Nutzung des Nutzers im angegebenen Monat zurück |
| **P-QS-M12** | *getQuotaPerDay-ForInstanceAndMonth* | Liefert die tagesgenaue Quota Nutzung der Instanz im angegebenen Monat zurück |

**Tabelle 7.9:** Pipeline-Service Quota Service

**P-QS-M1:** Die Methode *getBasicQuotaInformation* ruft zunächst die Hilfsmethode *findStorageQuotaEntityByUser* mit der übergebenen *User Id* und *findComputationQuotaEntityByUserAndMonth* mit der übergebenen *User Id*, dem aktuellen Monat und dem aktuellen Jahr auf.
Die Rückgabewerte der Hilfsmethoden werden dann zur Bestückung der *GetBa-*

*sicQuotaInformationDto* genutzt, welche anschließend zurückgegeben wird.
**P-QS-M2:** Die Methode *getDetailedQuotaInformation* ruft zunächst die *getBasicQuotaInformation* Methode mit der übergebenen *User Id* auf, um mit deren Rückgabewert bereits die Parameter *usedPoints* und *usedStorage* der *GetDetailedQuotaInformationDto* zu befüllen.
Nun fehlt noch der Wert für den Parameter *pointsLastMonth*, also den genutzten Punkten im letzten Monat. Hierfür nutzen wir die Hilfsmethode *findComputationQuotaEntityByUserAndMonth*, welcher wir hier den letzten Monat übergeben.
Zuletzt benötigen wir dann noch den Wert für den Parameter *usedPointsArray*, welcher ein Array aus Elementen des Typs *number* ist und für jeden Tag des aktuellen Monats die genutzten Punkte des Nutzers enthält.
Dieses Array liefert uns die Hilfsmethode *getQuotaPerDayForUserAndMonth*, der wir die *User Id* des zugehörigen Nutzers übergeben.
Nun geben wir die befüllte *GetDetailedQuotaInformationDto* zurück.
**P-QS-M11:** Die eben schon genutzte Hilfsmethode *getQuotaPerDayForUserAndMonth* bekommt die *User Id*, sowie Monat und Jahr übergeben.
Zunächst rufen wir mit der übergebenen *User Id* die *findForUser* Methode des *Instances Service* auf, welche alle Instanzen des Nutzers zurückliefert.
Dann erstellen wir ein Array in der Länge der Anzahl der Tage des aktuellen Monats. Dieses nennen wir im Folgenden Summenarray.
Anschließend iterieren wir über die zuvor erhaltenen Instanzen und rufen für jede die Methode *getQuotaPerDayForInstanceAndMonth* auf, welche die *Instance Id* der jeweiligen Instanz, den übergebenen Monat und das übergebene Jahr übergeben bekommt und für diese Instanz die tagesgenaue Punktenutzung im angegebenen Monat des angegebenen Jahres zurückliefert.
Anschließend iterieren wir über das zurückgegebene Array und addieren den Wert an der jeweiligen Stelle zum Wert an derselben Stelle im vorher definierten Summenarray.
Ist dies für jede Instanz geschehen, so geben wir das resultierende Array zurück, welches nun die Summe der Punktenutzung der Instanzen für jeden Tag des übergebenen Monats im übergebenen Jahr enthält.
**P-QS-M12:** Die Hilfsmethode *getQuotaPerDayForInstanceAndMonth* bekommt die *Instance Id* der Instanz übergeben, sowie Monat und Jahr.
Anschließend wird die Methode *getAllRunsForInstanceInclDeletedForTimePeriod* des *Runs Service* aufgerufen, welcher die *Instance Id*, sowie ein Start- und ein Endzeitpunkt übergeben werden. Von dieser Methode bekommen wir nun alle Runs zurück, die der Instanz mit der übergebenen *Instance Id* zugehörig sind und zwischen Start- und Endzeitpunkt gestartet wurden.
Als Startzeitpunkt wählen wir den ersten Tag um 00:00 des übergebenen Monats im übergebenen Jahr und als Endzeitpunkt den letzten Tag des übergebenen Monats im übergebenen Jahr um 23:59:59.
Nun erstellen wir ein Array mit Elementen des Typs *number* in der Länge der Anzahl der Tage des aktuellen Monats.

Über die Runs aus dem Rückgabewert der vorher aufgerufenen Methode wird nun iteriert. Die Punkte des jeweiligen Runs werden zum Wert des eben erstellten Punktearrays an der Stelle des jeweiligen Tages addiert, welches anschließend zurückgegeben wird.
**P-QS-M3:** Aufgerufen wird die Methode *updateQuotaInformationForNewRun* einmalig durch die *update* Methode des *Runs Service*, sobald die Resultate eines Runs vorliegen.
Übergeben bekommt die Methode eine *QuotaForRunDto* und die *User Id* des Nutzers, der den Run gestartet hat.
Die *QuotaForRunDto* enthält alle notwendigen Informationen zur Quota Nutzung eines Runs, die benötigt werden, um die Quota Informationen des Nutzers zu aktualisieren.

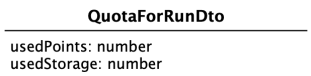


**Abbildung 7.19:** Quota For Run DTO

Zunächst wollen wir den Eintrag des Nutzers in der *Storage Quota* Datenbanktabelle aktualisieren.
Hierzu rufen wir die Hilfsmethode *findStorageQuotaEntityByUser* auf, welche uns das *Storage Quota Entity* des Nutzers aus der Datenbanktabelle liefert.
Sollte der Rückgabewert jedoch *null* sein, so existiert noch kein Eintrag in der *Storage Quota* Datenbanktabelle für diesen Nutzer. Dieser Fall kann nur bei alten Nutzern auftreten, die bereits einen Account erstellt hatten, bevor das Quota System eingeführt wurde. Denn seit Einführung des Quota Systems erfolgt direkt beim Anlegen eines neuen Nutzers ein *Storage Quota* Eintrag für ihn in der Datenbank (siehe *createEmptyStorageQuotaForUser*). Für diesen Fall erstellen wir einen neuen Eintrag in der Datenbank und wählen hierzu für den Parameter *usedStorage* den Wert aus der *QuotaForRunDto*.
Sollte jedoch bereits ein Eintrag in der Datenbank zu dieser *User Id* existieren, so nutzen wir die *increment* Funktion des Repositories, um den *usedStorage* Wert um den in der *QuotaForRunDto* gespeicherten Wert zu inkrementieren. Hierbei ist es wichtig, die *increment* Funktion des Repositories zu nutzen und die Inkrementierung nicht durch Abfrage des Wertes, Addition und anschließender Speicherung vorzunehmen. Denn hierbei kann es sonst zu einer *Race Condition* kommen, wodurch falsche Werte in die Datenbank gelangen könnten. Die *increment* Funktion der Datenbank verhindert dies, da dort das Abfragen, Addieren und Speichern in einer gemeinsamen Transaktion stattfindet und somit nicht unterbrochen werden kann.

Für die *Computation Quota* Datenbanktabelle gehen wir ähnlich vor.
Zunächst wird hier die Hilfsmethode *findComputationQuotaEntityByUserAndMonth* mit der übergebenen *User Id* und dem aktuellen Monat des aktuellen

Jahrs aufgerufen und deren Rückgabewert gespeichert.
Sollte dieser *null* sein, so existiert für diesen Nutzer im angegebenen Monat noch kein Eintrag in der Datenbanktabelle *Computation Quota*. Für diesen Fall wird eine neue Instanz des *Computation Quota Entities* für diese *User Id*, diesen Monat und dieses Jahr erstellt, welche für den *usedPoints* Parameter den entsprechenden Wert aus der *QuotaForRunDto* zugewiesen bekommt. Diese Instanz wird dann in der Datenbanktabelle *Computation Quota* gespeichert. Hierbei kann es zu einer *Race Condition* kommen, falls ein Nutzer, für den für diesen Monat des aktuellen Jahrs noch kein Eintrag in der Datenbank existiert, in sehr kurzem Abstand, quasi gleichzeitig, mehrere Runs startet. Dabei kann es zu einem *Schreib-Lese-Konflikt* kommen, indem der eine Aufruf dieser Methode sein neu erstelltes *Computation Quota Entity* noch nicht in der Datenbank gespeichert hat, während ein anderer Aufruf dieser Methode mit derselben *User Id* aber schon die Hilfsmethode *findComputationQuotaEntityByUserAndMonth* aufgerufen und *null* zurückbekommen hat, wodurch dieser ebenso ein neues *Computation Quota Entity* erstellt. Dies würde dazu führen, dass nun zwei *Computation Quota Entities* für denselben Nutzer für den aktuellen Monat existieren würden. Dadurch würde jedoch lediglich die Quota Nutzung eines Runs nicht mit zur Quota Nutzung des Nutzers für diesen Monat dazuzählen. Denn eines der beiden *Computation Quota Entities* würde für den restlichen Monat für das Abrufen und Aktualisieren von Quota Informationen des Nutzers genutzt werden. Das andere, welches weiter hinten in der Datenbank liegt, würde ignoriert werden. Dieses Risiko ist jedoch als sehr gering anzusehen, weshalb wir es in Kauf nehmen. Eine Alternative wäre ein Programm zu schreiben, welches automatisch am Anfang jeden Monats für jeden Nutzer einen Eintrag in der Datenbanktabelle *Computation Quota* erstellen würde.
Sollte der Rückgabewert der aufgerufenen Hilfsmethode *findComputationQuotaEntityByUserAndMonth* jedoch nicht *null* sein, so existiert bereits ein Eintrag in der Datenbanktabelle für diese *User Id* für diesen Monat und diesem Jahr. Dann wird der Wert von *usedPoints* dieses Eintrags um den Wert aus der *QuotaForRunDto* inkrementiert. Auch hierbei ist es wieder wichtig, die *increment* Funktion des Repositories zu verwenden, um eine *Race Condition* auszuschließen.
**P-QS-M4:** Die Methode *updateQuotaInformationForDeletedRun* wird vom *Runs Service* aufgerufen, wenn dort ein Run gelöscht wird. Denn daraufhin muss auch der genutzte Speicherplatz des Nutzers, dem die Instanz gehört und welcher somit auch den Run gestartet hat, dementsprechend dekrementiert werden.
Hierfür bekommt die Methode die *User Id* des Nutzers übergeben, sowie die Anzahl an KB um die der *usedStorage* Parameter des *Storage Quota Entities* des Nutzers dekrementiert werden soll, also die Anzahl an KB, die die Resultate des Runs belegt haben.
Hierzu wird über die *findStorageQuotaEntityByUser* Methode das *Storage Quota Entity* des Nutzers geholt. Sollte hier kein Eintrag in der Datenbanktabelle *Stora-*

*ge Quota* existieren, so hat dieser Nutzer seit Einführung des Quota Systems noch keinen Run gestartet. Somit hat dieser Run auch noch nicht zum Speicherplatz Quota des Nutzers gezählt und die Quota Informationen müssen nicht aktualisiert werden. Die Methode wird dann an dieser Stelle beendet.
Wenn jedoch bereits ein Eintrag in der Datenbanktabelle zu diesem Nutzer existiert, so wird die *decrement* Funktion des Repositories aufgerufen, welche den *usedStorage* Wert des Eintrags in der Datenbank um den übergebenen Wert dekrementiert.
Hier ist es besonders wichtig, die *decrement* Funktion des Repositories zu verwenden, da eine *Race Condition* sonst sehr realistisch wäre, da beispielsweise beim Löschen einer Instanz viele Runs desselben Nutzers auf einmal gelöscht werden und diese Methode somit in kürzester Zeit sehr oft aufgerufen wird.
**P-QS-M7:** Die Methode *getPointsForInstance* nutzt zunächst die *getAllRunsInclDeletedForUserAndMonth* Methode des *Runs Service*, welche für die Instanz mit der übergebenen *Instance Id* alle zugehörigen Runs im übergebenen Zeitraum zurückliefert. Als Zeitraum übergeben wir hier den aktuellen Monat. Anschließend wird über die zurückgelieferten Runs iteriert und die Summe über den *points* Parameter gebildet und anschließend zurückgegeben.
**P-QS-M8:** Die Methode *getStorageForInstance* ruft mit der übergebenen *Instance Id* die *getAllRuns* Methode des *Runs Service* auf. Diese Methode gibt alle Runs, die zur übergebenen *Instance Id* gehören, zurück.
Im Gegensatz zur *getPointsForInstance* Methode benötigen wir hierfür alle Runs und nicht nur die eines bestimmten Zeitraums, da es sich beim Speicherplatz um eine fortlaufende Ressource handelt (siehe **Kapitel Quota Modell**). Hierbei benötigen wir jedoch nicht die gelöschten Runs, da die Resultate derer auch gelöscht wurden und somit keinen Speicherplatz mehr belegen, weswegen dieser auch nicht zum Storage Quota dazuzählt.
Über diese Runs wird nun iteriert und die Summe über deren *storage* Parameter gebildet, welche anschließend zurückgegeben wird.
**P-QS-M6:** Die Methode *getInstancesQuotaInformation* bekommt drei Parameter übergeben.
Die *User Id*, für welche die Quota Informationen zurückgegeben werden sollen, sowie zwei optionale Parameter.
Der optionale Parameter *limit* bestimmt die Anzahl an Instanzen, für die die Quota Informationen zurückgegeben werden sollen.
Der optionale Parameter *orderBy* kann entweder den Wert *points* oder *storage* haben.
Im Falle von *points* werden die Quota Informationen für die Instanzen mit der höchsten Punktenutzung zurückgegeben, bei *storage*, die mit der höchsten Speicherplatzbelegung.
Hierbei wird ein Array der Länge *limit* mit dem Datentyp *GetInstancesQuotaInformationDto* zurückgegeben. Zunächst wird die Methode *findForUser* des *Instances Service* mit der übergebenen *User Id* aufgerufen. Der Rückgabewert

ist ein Array aus allen Instanzen zu dieser *User Id*.
Über diese Instanzen wird nun iteriert und pro Instanz ein Objekt aus *Instance Id* der Instanz, Punktenutzung der Instanz in diesem Monat, sowie Speicherplatzbelegung der Resultate der, der Instanz zugehörigen, Runs gebildet.
Die Informationen zur Punkte- und Speichernutzung der Instanz bekommen wir durch Aufruf der Methoden *getPointsForInstance* bzw. *getStorageForInstance* mit der jeweiligen *Instance Id*.
Anschließend werden diese Objekte noch, abhängig vom Parameter *orderBy*, nach Punktenutzung bzw. Speicherplatzbelegung sortiert und falls der *limit* Parameter definiert ist, das Array abgeschnitten, sodass nur noch die ersten *limit* Elemente übrigbleiben. Dieses Array wird anschließend zurückgegeben.
Die optionalen Parameter werden erst im *Hub-Backend* gesetzt.

### 7.4.2 Instances Komponente

#### Instances Entity

Zunächst muss dem *Instance Entity* noch der Parameter *ReservedRamMB* hinzugefügt werden, welcher die gewählte Arbeitsspeichergröße für die Ausführung von Runs dieser Instanz angibt. Default Wert hierfür ist *4196*.

#### Instances Controller

Im *Instances Controller* wird für das Quota System lediglich der folgende Endpunkt zum Aktualisieren des *reservedRamMB* Parameters benötigt.

| **ID** | Endpunkt (Typ) | Zugehörige Methode des *Instances Service* |
|---|---|---|
| **P-IC-E1** | *instances/:instanceId (PATCH)* | *update* |

**Tabelle 7.10:** Pipeline-Service Instances Controller

#### Instances Service

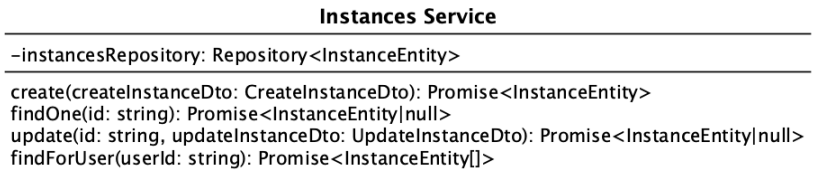


**Abbildung 7.20:** Pipeline-Service Instances Service Klassendiagramm

| ID | Methode | Kurzbeschreibung |
|---|---|---|
| **P-IS-M1** | *create* | Erstellt eine neue Instanz in der Datenbanktabelle Instances |
| **P-IS-M2** | *update* | Aktualisiert die Informationen zu einer Instanz in der Datenbanktabelle Instances |
| **P-IS-M3** | *findForUser* | Gibt alle Instanzen des Nutzers zurück |

**Tabelle 7.11:** Pipeline-Service Instances Service

**P-IS-M1:** In der create Methode des *Instances Service* muss zudem noch eine Zuweisung des Wertes für *reservedRamMB* stattfinden.

### 7.4.3 Runs Komponente

#### Run Entity

Zunächst müssen dem *Run Entity* ein paar Parameter hinzugefügt werden, sodass Informationen zur Ressourcenverwendung des Runs in der Datenbank gespeichert werden können.

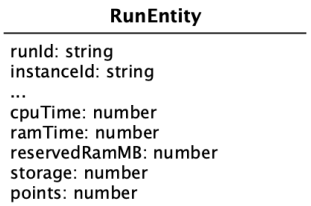


**Abbildung 7.21:** Run Entity

Der Parameter *cpuTime* speichert die CPU-Zeit in ms, *ramTime* die insgesamte Ausführungszeit in ms, *storage* den Speicherplatz in KB, den die Resultate des Runs belegen, und *points* die Punkte, die für den Run anhand der Ressourcenverwendung berechnet wurden.
*ReservedRamMB* speichert den zur Verfügung gestellten Arbeitsspeicher des Runs in MB. Dieser Wert wird zwar auch im *Instance Entity* der zugehörigen Instanz gespeichert, jedoch kann dieser vom Nutzer geändert werden, weswegen wir im *Run Entity* den Wert für *ReservedRamMB* speichern, der zum Start des jeweiligen Runs ausgewählt war und somit auch zur Verfügung gestellt wurde. Default Wert für alle neuen Parameter ist *0*.

#### Runs Controller

Im *Runs Controller* führen wir zwei API-Endpunkte zum Löschen von Runs ein. Wir haben uns dazu entschieden, für die Metadaten der Runs und Instanzen nur *Soft Deletes* durchzuführen, also die Runs bzw. Instanzen nicht wirklich zu löschen, sondern nur das *delete Flag* im Falle einer Löschung zu setzen. Die Logik

hierfür übernimmt das *Typeorm*[10] *Repository*.
Grund für die Entscheidung zum *Soft Delete* ist, dass die Metadaten der Runs und Instanzen im Vergleich zu deren Resultaten kaum Speicherplatz belegen. Die Metadaten der Runs dürften wir sowieso erst zu Beginn eines neuen Monats löschen, da sonst die tagesgenaue Anzeige der Punktenutzung nicht mehr umsetzbar wäre, weil wir diese anhand der Metadaten der Runs erstellen.
Dadurch, dass wir die Metadaten nur per *Soft Delete* löschen, können wir sie weiter für statistische Zwecke nutzen. Ein Beispiel hierfür wäre, die tagesgenaue Punktenutzung auch für vergangene Monate anzuzeigen, inklusive der Runs, die eigentlich schon gelöscht wurden. Die Resultate der Runs wollen wir jedoch per *Hard Delete* löschen, da diese den Großteil des Speicherplatzes belegen. Den *Hard Delete* der Run Resultate haben wir noch nicht implementiert, mehr dazu im **Kapitel 9.2.2**.

### Runs Controller

| **ID** | Endpunkt (Typ) | Zugehörige Methode des *Runs Service* |
|---|---|---|
| **P-RC-E1** | *instances/:instanceId/runs/:runId (DELETE)* | *softDeleteForRun* |
| **P-RC-E2** | *instances/:instanceId/runs (DELETE)* | *softDeleteForInstance* |
| **P-RC-E3** | *instances/:instanceId/runs (POST)* | *create* |

**Tabelle 7.12:** Pipeline-Service Runs Controller

**P-RC-E3:** Der POST instances/:instanceId/runs Endpunkt existiert bereits. Wir stellen ihn trotzdem zum besseren Verständnis hier kurz vor.
Der Endpunkt dient zum Starten eines neuen Runs der Instanz mit der mitgegebenen *Instance Id*.
Zunächst wird hierbei überprüft, ob überhaupt eine Instanz mit dieser *Instance Id* existiert. Falls nicht wird eine *NotFoundException* geworfen.
Anschließend wird der Speicherort für die Resultate des Runs vorbereitet und dann die *create* Methode des *Runs Service* aufgerufen, welche den Run in der Datenbanktabelle *Run* anlegt.
Daraufhin wird die Hilfsmethode *startRun* des *Runs Controllers* aufgerufen. Diese ruft die *callRuntime* Methode des *Runs Service* mit allen notwendigen Parametern, inklusive des *reservedRamMB* Parameters, welcher im *InitiateRunsParams* Objekt gespeichert ist, auf.
Das zurückgegebene Egebnis werten wir aus, um herauszufinden, ob beim Start des Runs ein Fehler auftrat.
Für diesen Fall, wird die *update* Methode des *Runs Service* aufgerufen, um die Informationen zum Fehler in der *Run* Datenbanktabelle zu speichern.

[10] https://typeorm.io

### Runs Service

In der Service Klasse der *Runs Komponente* benötigen wir einige Anpassungen und Ergänzungen.

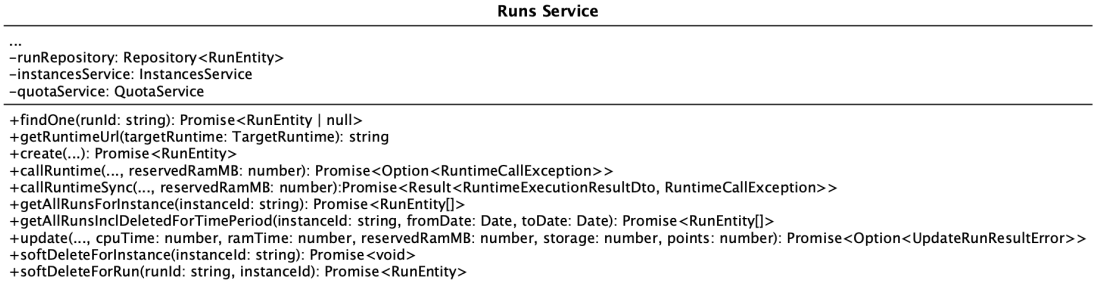


**Abbildung 7.22:** Pipeline-Service Runs Service Klassendiagramm

Als globale Variable wird hier das *Run Repository* für den Zugriff auf die Run Datenbanktabelle benötigt. Zudem werden per *Dependency Injection* noch die Instanz des *Instances Service* und des *Quota Service* bereitgestellt.

| ID | Methode | Kurzbeschreibung |
|---|---|---|
| **P-RS-M1** | *findOne* | Liefert das *Run Entity* zur übergebenen *Run Id* zurück oder null |
| **P-RS-M2** | *getRuntimeURL* | Liefert die *URL* zur übergebenen *Runtime* zurück |
| **P-RS-M3** | *create* | Erstellt einen Datenbankeintrag in der *Run* Tabelle für einen neuen Run |
| **P-RS-M4** | *callRuntime* | Startet einen Run durch API-Request an den *runs* POST-Endpunkt der *Runtime* |
| **P-RS-M5** | *callRuntimeSync* | Startet einen Run durch API-Request an den *runs/sync* POST-Endpunkt der *Runtime* und wartet auf dessen Beendigung |
| **P-RS-M6** | *getAllRuns-ForInstance* | Gibt alle nicht gelöschten Runs zur übergebenen *Instance Id* zurück |
| **P-RS-M7** | *getAllRunsFor-InstanceInclDeleted-ForTimePeriod* | Gibt alle Runs zur übergebenen *Instance Id* im angegebenen Zeitraum zurück |
| **P-RS-M8** | *update* | Aktualisiert die Informationen zu einem Run in der Datenbank |
| **P-RS-M9** | *softDelete-ForInstance* | Soft Delete der Instanz mit der übergebenen *Instance Id* und deren Runs. Ruft für jeden Run die *updateQuotaInformationForDeletedRun* Methode des *Quota Service* auf |
| **P-RS-M10** | *softDeleteForRun* | Soft Delete des Runs mit der übergebenen *Run Id* und ruft die *updateQuotaInformationForDeletedRun* Methode des *Quota Service* auf |

**Tabelle 7.13:** Pipeline-Service Runs Service

**P-RS-M8:** Die Methode *update* bekommt Informationen zu einem Run übergeben und aktualisiert dazu die Daten in der Datenbank.
Hierbei wird nun auch der *Quota Service* benötigt, denn wir rufen hier die Methode *updateQuotaInformationForNewRun* auf, sollten die übergebenen Parameter *cpuTime*, *ramTime*, *reservedRamMB*, *storage* und *points* hierzu definiert sein.
Durch den Aufruf der Methode *updateQuotaInformationForNewRun* des *Quota Service* werden die Quota Informationen des Nutzers aktualisiert. Der Nutzer wird zuvor über die dem Run zugehörige Instanz ermittelt.
Hierbei ist es wichtig, dass die *update* Methode des *Runs Service* nur einmal pro Run mit den Quota Informationen aufgerufen wird, da sonst das *Computation Quota* und das *Storage Quota* des Nutzers mehrfach erhöht werden würde.

### 7.4.4 Results Komponente

Die *Results Komponente* dient lediglich als API-Schnittstelle zur *Runtime-Simple* und hat auch keine Service Klasse.

#### Results Controller

| **ID** | Endpunkt (Typ) | Zugehörige Methode des *Runs Service* |
|---|---|---|
| **P-REC-E1** | *results/:runId (POST)* | *update* |

**Tabelle 7.14:** Pipeline-Service Results Controller

**P-REC-E1:** Der *Results Controller* bietet lediglich einen *POST* API-Endpunkt an, über welchen Daten eines abgeschlossenen Runs empfangen werden.
Daraufhin wird überprüft, ob der Run erfolgreich war oder ein Fehler aufgetreten ist. Anschließend wird die bereits vorgestellte *update* Methode des *Runs Service* aufgerufen, wodurch die Informationen zum Run in der Datenbanktabelle *Run* gespeichert werden.

## 7.5 Runtime-Simple

Die *Runtime-Simple* kümmert sich ausschließlich um die Ausführung der Runs. Hierfür gibt es auch lediglich die *Runs Komponente*.

### 7.5.1 Runs Komponente

#### Runs Controller

Der *Runs Controller* bietet zwei *POST* API-Endpunkte zum Starten von Runs an.

| **ID** | Endpunkt (Typ) | Zugehörige Methode des *Runs Service* |
|---|---|---|
| **R-RC-E1** | *runs (POST)* | *interpretModel* |
| **R-RC-E2** | *runs/sync (POST)* | *interpretModel* |

**Tabelle 7.15:** Runtime-Simple Runs Controller

Beide starten durch Aufruf der *interpretModel* Methode des *Runs Service* einen Run.
Der einzige Unterschied besteht darin, dass der Endpunkt **R-RC-E2** auf die Fertigstellung des Runs wartet und anschließend die Informationen zum abgeschlossenen Run, inklusive der Informationen zum Speicherort der Resultate, zurückliefert, während der Endpunkt **R-RC-E1** nicht auf die Beendigung des Runs

wartet. In diesem Fall werden die Informationen zum Run nach dessen Abschluss über die Methode *performCallback* des *Runs Service* per *POST* API-Request an die übergebene *callbackUrl* gesendet, also in der Regel an den *Results Controller* des *Pipeline-Service*.

### Runs Service

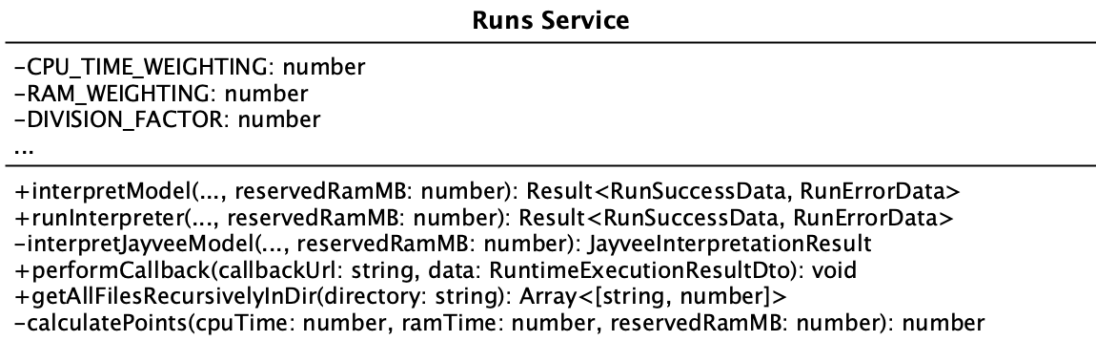


**Abbildung 7.23:** Runtime-Simple Runs Service Klassendiagramm

Der *Runs Service* kümmert sich um die tatsächliche Ausführung der Runs. Hierbei dient die Methode *interpretModel* als *Entrypoint*.

| ID | Methode | Kurzbeschreibung |
|---|---|---|
| **R-RS-M1** | *interpretModel* | Bereitet einen temporären Ordner für Run Resultate vor und gibt den Rückgabewert der aufgerufenen runInterpreter Methode zurück |
| **R-RS-M2** | *runInterpreter* | Bereitet alles für die Ausführung des Runs vor und ruft die Methode interpretJayveeModel auf, deren Rückgabewert ausgewertet und anschließend die Informationen zum Run zurückgegeben werden |
| **R-RS-M3** | *interpret-JayveeModel* | Ruft den Interpreter zur Ausführung des Runs auf und misst dabei die Ressourcenverwendung |
| **R-RS-M4** | *performCallback* | Sendet Informationen zum abgeschlossenen Run per POST API-Request an die übergebene URL (meist der POST-Endpunkt der Results Komponente des *Pipeline-Service*). Hierzu gehören auch die Quota Informationen und Informationen zum Speicherort der Run Resultate |
| **R-RS-M5** | *getAllFiles-RecursivelyInDir* | Gibt die Pfade zu allen Dateien im übergebenen Verzeichnis und deren Unterverzeichnissen zurück, inklusive deren Speichergröße |
| **R-RS-M6** | *calculatePoints* | Berechnet anhand der übergebenen Parameter und der Konstanten den Pipeline-Score und anschließend die Pipeline-Points des Runs |

**Tabelle 7.16:** Runtime-Simple Runs Service

**R-RS-M2:** Die Methode *runInterpreter* bereitet nun alles zum Ausführen des Runs vor und ruft die Methode *interpretJayveeModel* auf, welche sich dann um die wirkliche Ausführung des Runs kümmert.
Hierbei wird auf den Rückgabewert dieser Methode gewartet und im Erfolgsfall die Methode *getAllFilesRecursivelyInDir* mit dem Pfad des Ordners, in dem die Run Resultate gespeichert sind, aufgerufen. Diese Methode gibt dann alle Pfade zu den Dateien der Resultate des Runs zurück, sowie deren Größe.
Anschließend werden all diese zu speichernden Dateien bzw. Resultate des Runs per API-Request an den *File Service* zur Speicherung geschickt.
Die insgesamte Speichergröße der Runs wird aus der Summe der einzelnen Speichergrößen gebildet.
Die Informationen zu *cpuTime* und *ramTime* sind direkt im Rückgabewert des Typs *JayveeInterpretationResult* gespeichert. Die Punkte des Runs werden durch die Methode *calculatePoints* bestimmt.
Zurückgegeben wird nun ein *Results* Objekt, was entweder bei Erfolg des Runs

ein Objekt des Typs *RunSuccessData* zurückgibt:

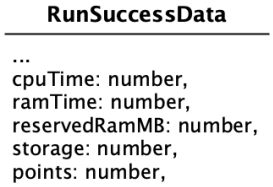


**Abbildung 7.24:** Run Success Data

Oder im Fehlerfall ein Objekt des Typs *RunErrorData*:

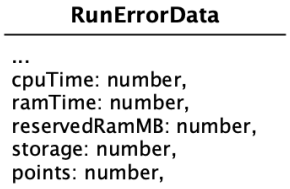


**Abbildung 7.25:** Run Error Data

**R-RS-M3:** Die Methode *interpretJayveeModel* führt dann den Run, durch Aufruf der *interpretModel* Methode des *JavyeeInterpreters*, aus.
Dabei wird die Dauer der Ausführung als Wert für die *ramTime* gemessen.
Ebenso wird die *cpuTime* gemessen, indem vor dem Methodenaufruf die Methode *process.cpuUsage* aufgerufen wird und ebenso direkt danach nochmal mit dem vorherigen Rückgabewert der Methode als Parameter.
Bei der Ausführung von mehreren Runs gleichzeitig auf demselben *Runner* können aktuell noch Ungenauigkeiten auftreten. Aktuell wird die Verteilung der Runs auf die *Runner* jedoch sowieso überarbeitet, sodass in Zukunft auf einem *Runner* immer nur **ein** Run gleichzeitig ausgeführt wird, sodass diese Ungenauigkeiten nicht mehr auftreten können.
Nach der Messung wird die Summe aus dem *User cpuTime* Wert und dem *System cpuTime* Wert gebildet und diese durch *1000* geteilt, um die *cpuTime* in Millisekunden zu bekommen.
Anschließend wird im Erfolgsfall des Runs ein Objekt des Typs *JayveeInterpretationResult* zurückgeben, welches die *cpuTime* und die *ramTime* enthält, sowie die *runLogs*, die wir vom *jayveeLogger* als Ergebnis bekommen:

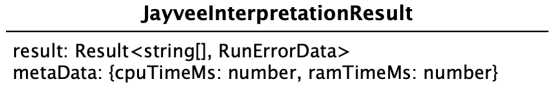


**Abbildung 7.26:** Jayvee Interpretation Result

Im Fehlerfall wird ein *Error* Objekt zurückgegeben.

# 8 Evaluation

In diesem Kapitel wollen wir darstellen, welche der Anforderungen aus **Kapitel 3** umgesetzt wurden. Wie dort bereits beschrieben, beziehen sich die vorgestellten Anforderungen auf ein Quota System für JValue und nur teilweise auf diese Arbeit. Als Anforderungen für diese Arbeit können die der Priorität **A** als MVP gesehen werden und die der Priorität **B** als Stretch Goals.
Im Folgenden gehen wir nur auf die Anforderungen der Priorität **A** und **B**, sowie auf die erreichten und teilweise erreichten Anforderungen der Prioritäten **C** und **D** ein. Die nicht erreichten Anforderungen aller Prioritäten werden im **Kapitel Future Work** vorgestellt, sowie Ansätze für deren Umsetzung erarbeitet. Zusätzlich wird dargestellt, warum diese nicht umgesetzt wurden.

## 8.1 A-Anforderungen

**A-Anforderungen** haben wir als Anforderungen definiert, welche umgesetzt werden müssen, um ein voll funktionsfähiges Quota System bereitzustellen.

**A1.1: Die Anzahl an Pipeline Runs pro Projekt und Nutzer müssen aufgezeichnet werden (wurde ersetzt)**
Dies wäre über die *Run* Tabelle im *Pipeline-Service* leicht umzusetzen, da die Runs nur per *Soft Delete* gelöscht werden. Jedoch wurde die Anforderung durch die Erfüllung von *B1.1* ersetzt.

**A1.2: Die Ausführung von Pipeline Runs pro Monat und Nutzer muss begrenzt werden (wurde erfüllt)**
Diese Anforderung wurde durch die Einführung des Punkte Quota Limits erfüllt, indem jedem Run eine gewisse Anzahl an Punkten zugeordnet wird und der Nutzer pro Monat, je nach gewähltem Plan, nur eine gewisse Anzahl an Punkten zur Verfügung hat.

**A2: Es muss ein Dashboard geben, in welchem dem Nutzer folgende Daten angezeigt werden (wurde teilweise erfüllt)**

Ein **Quota Dashboard** zur Anzeige der Quota Nutzung haben wir eingeführt.

**A2.1 Anzahl Pipeline Runs pro Nutzer und Monat (wurde erfüllt)**
Hier wurde abstrahiert, sodass im **Quota Dashboard** die Anzahl an genutzten Punkten angezeigt wird und nicht die Anzahl an Pipeline Runs, da diese in der Abrechnung, bzw. im Abo des Nutzers keine direkte Rolle spielen.

**A2.2 Anzahl Pipeline Runs pro Projekt (wurde nicht erfüllt)**
Die Anzeige pro Projekt auf dem Quota Dashboard war aus architektonischen Gründen nicht performant umsetzbar. Anstatt dessen zeigen wir dort die Instanzen mit der höchsten Punktenutzung bzw. Speicherplatzbelegung an. Für mehr Informationen hierzu siehe **Kapitel Future Work**.

**A3: Es muss mindestens 2 Abo-Stufen geben, durch welche das Kontingent an Pipeline Runs pro Nutzer und Monat vorgegeben wird (wurde erfüllt)**
Wir haben mehrere Pläne bzw. Abo-Stufen eingeführt.

**A3.1 Eine mit limitierter Anzahl an Pipeline Runs pro Monat (wurde erfüllt)**
Wir haben mehrere Pläne eingeführt, die die Punktenutzung pro Monat und somit auch die Anzahl an ausführbaren Pipeline Runs begrenzen.

**A3.2 Eine mit unlimitierter Anzahl an Pipeline Runs pro Monat (wurde erfüllt)**
Wir haben ebenso einen unlimitierten Plan eingeführt. Dieser wird jedoch in Zukunft durch die Abrechnung per Einheitspreis abgelöst.

## 8.2 B-Anforderungen

**B-Anforderungen** haben wir als Solche definiert, die umgesetzt werden sollen, jedoch für die Grundfunktionalität nicht zwangsläufig notwendig sind.

**B1.1: Die von einem Pipeline Run benötigten Ressourcen (CPU-Zeit, Arbeitsspeicher) sollen gemessen und dargestellt werden (wurde erfüllt)**
Wir messen bei der Ausführung eines Runs die CPU-Millisekunden. Die Ressource Arbeitsspeicher mussten wir aufgrund des Problems mit dem Garbage Collector abändern (siehe **Kapitel 7.3.4 Pipelines Komponente**). Dafür haben wir die Variante von AWS Lambda und Google Dataflow adaptiert.
Die gemessenen Werte, werden auf der **Pipeline Page** angezeigt.

**B1.2: Die Pipeline Runs sollen anhand des genutzten Arbeitsspeichers und der genutzten CPU-Minuten in verschiedene Kategorien mit unterschiedlicher Bepreisung unterteilt werden und anhand dessen begrenzt werden (wurde erfüllt)**
Die Runs werden anhand derer Ressourcennutzung in verschiedene Größen unterteilt, für welche wiederum eine gewisse Anzahl an Punkten bei Ausführung fällig wird. Die Nutzung dieser Punkte ist pro Monat und Nutzer über dessen gewählten Plan begrenzt.

**B2.1: Der genutzte Speicher für die Bereitstellung der Ergebnisse eines Pipeline Runs soll gemessen und dargestellt werden (wurde erfüllt)**
Der Speicherplatz, den die Resultate eines Runs belegen, wird gemessen und mit in der Datenbanktabelle Run gespeichert. Dargestellt wird der genutzte Speicherplatz pro Run auf der **Pipeline Page** und pro Nutzer auf dem **Quota Dashboard**.

**B2.2: Die Nutzung von Speicherplatz soll pro Nutzer begrenzt werden (wurde erfüllt)**
Die Nutzung von Speicherplatz wird auf die im Abo enthaltene Menge begrenzt.

**B3: Es muss mindestens 2 Abo-Stufen geben, durch welche das Speicherplatzkontingent pro Nutzer vorgegeben wird (wurde erfüllt)**
Wir haben mehrere Pläne bzw. Abo-Stufen eingeführt.

**B3.1 Eine mit limitierter Menge an Speicherplatz insgesamt (wurde erfüllt)**
In den Plänen ist auch ein Kontingent an Speicherplatz enthalten.

**B3.2 Eine mit unlimitierter Menge an Speicherplatz insgesamt (wurde erfüllt)**
Im unlimitierten Plan ist eine unlimitierte Menge an Speicherplatz enthalten.

## 8.3 C-Anforderungen

**C-Anforderungen** stellen Anforderungen dar, welche erfüllt werden könnten, jedoch nicht für die allgemeine Funktion notwendig sind.

**C1: Es könnte mehr als 2 Abo-Stufen geben (wurde erfüllt)**
Wir haben eine Liste an Plänen erstellt, die jederzeit erweitert werden kann.

**C2: Es könnte ein „Soft Limit“ geben, sodass Nutzer die Quota Limits um einen vordefinierten Prozentsatz überschreiten dürfen (wurde erfüllt)**
Wir haben implementiert, dass Nutzer ihre Quota Limits um einen definierten Prozentsatz überschreiten dürfen. Hierfür haben wir derzeit 5% gewählt.

**C3: Es könnte die Option zur Auswahl geben, dass bei Erreichen des Speicher Limits automatisch die ältesten Runs mit deren Resultaten gelöscht werden (wurde nur konzipiert, jedoch nicht erfüllt)**
Dies ist die *Auto Delete* Funktion, welche wir vorgestellt und auch bereits in der Implementierung teilweise berücksichtigt haben.
Eine komplette Umsetzung hat sich aufgrund architektonischer Gründe ohne große Änderungen nicht umsetzen lassen. Auf die Gründe hierfür werden wir im **Kapitel Future Work** genauer eingehen.

**C5: Ad hoc Ausführungen könnten nicht zur monatlichen Nutzung zählen (wurde erfüllt)**
Sogenannte *Quick Runs*, wie beispielsweise Runs, die durch den Editor gestartet werden, zählen nicht zur Quota Nutzung des Kunden dazu.

## 8.4 D-Anforderungen

**D-Anforderungen** definierten wir als diejenigen Anforderungen, welche für die Zukunft eine Option zur Umsetzung darstellen, jedoch auch teilweise mit dem aktuellen Stand von JValue noch gar nicht umsetzbar sind. Sie sind eher Ideen für die Zukunft, welche noch genauer ausgearbeitet werden müssen.

**D2: Es wäre eine Option, den Nutzern die Wahl zu geben, ob bei Erreichung des Quota Limits das Ausführen weiterer Runs blockiert werden soll, oder ob ab Erreichung des Limits nach Einheitspreis, mit Angabe eines maximalen Limits, abgerechnet werden soll. (wurde vorgestellt, jedoch nicht erfüllt)**
Diese Funktion wurde aus Zeitgründen nicht implementiert, sondern lediglich im **Kapitel Quota Modell** vorgestellt.

**D4: Es wäre eine Option, im Dashboard eine Schätzung für den Verbrauch bis Monatsende anzuzeigen (wurde erfüllt)**
Die Hochrechnung zur Quota Nutzung eines Anwenders zeigen wir in der **Komponente Punkte-Diagramm** an.

# 9 Optionen zur Optimierung und Erweiterung des Quota Systems bei JValue

In diesem Kapitel werden wir auf Optimierungs- und Erweiterungsmöglichkeiten des Quota Systems und seiner Komponenten eingehen.

## 9.1 Optimierbare oder nicht erfüllte Anforderungen

Zunächst wollen wir hierfür auf die nicht erfüllten Anforderungen eingehen.

### A2.2: Anzeige Quota pro Projekt auf Quota Dashboard

Aus konzeptionellen Gründen nicht umgesetzt wurde die Anzeige der Quota Nutzung der einzelnen Projekte auf dem *Quota Dashboard*.
Das Problem hierbei ist, dass der *Pipeline-Service* keinerlei Wissen über die Zuordnung zwischen Instanzen und deren Projekte hat, sodass für diese Funktion zunächst die Quota Nutzung für alle Instanzen vom *Pipeline-Service* beschafft werden müsste, um dann im *Hub-Backend* die Zuordnung stattfinden zu lassen. Dies wäre jedoch bei Nutzern mit vielen Projekten ein zu hoher Datentransfer. Anstatt dessen zeigen wir die Quota Nutzung der Instanzen mit der höchsten Punktenutzung bzw. der höchsten Speicherplatzbelegung an.
Zum einen verringert dies den Datentransfer, da nur die Instanzen mit der höchsten Quota Nutzung an das *Frontend* geschickt werden, und zum anderen kann die Bestimmung dieser Instanzen komplett im *Pipeline-Service* stattfinden.

### B1.1 Messung der Ressourcennutzung bei Ausführung von Runs

Es ist hier lediglich nicht gelungen den genutzten Arbeitsspeicher zu messen, weswegen ein Ersatz für die Ressource Arbeitsspeicher eingeführt wurde. Hierbei hat

der Nutzer die Möglichkeit die Arbeitsspeichergröße für seinen auszuführenden Run selbst zu wählen. Aktuell gibt es nur *4GB* zur Auswahl. In Zukunft wäre es notwendig hier mehr Auswahlmöglichkeiten zur Verfügung zu stellen.

### C2: Soft Limit

In der Anforderung C2 wird vorgeschlagen ein *Soft Limit* einzuführen, sodass Nutzer ihr monatliches Quota Limit um eine gewisse Prozentzahl überschreiten dürfen.
Dies haben wir zum Teil implementiert, indem in der Methode *getQuotaReached-Information* eine Überprüfung stattfindet, ob die im Plan enthaltenen Punkte bzw. der enthaltene Speicherplatz um mehr als die definierte Prozentzahl des *Soft Limits* überschritten wurde.
Wünschenswert wäre es, hierbei die Überschreitung nur dann zu erlauben, wenn diese keine Regelmäßigkeit bei diesem Nutzer darstellt. Zur Überprüfung müsste dessen Quota Nutzung der letzten Monate abgefragt werden.

### C3: Auto Delete Funktion

Die *Auto Delete* Funktion konnte aus architektonischen Gründen nicht performant umgesetzt werden.
Da in der *Run* Datenbanktabelle im *Pipeline-Service* keine Informationen zum Nutzer gespeichert werden, ist es sehr kompliziert, die ältesten Runs eines Nutzers zu finden.
Der Nutzer zu einem Run lässt sich lediglich über die Instanz, der dieser Run zugeordnet ist, herausfinden. Denn dort findet eine Zuordnung zum Nutzer statt. Man müsste somit zunächst alle Runs aller Instanzen eines Nutzers zusammentragen, um daraus die ältesten herauszufiltern. Dieses Vorgehen wäre sehr unperformant und stellt somit keine Option zur Umsetzung dar.
Die sinnvollste Lösung wäre es, in der Datenbanktabelle *Run* im *Pipeline-Service* auch den Nutzer zu den Runs zu speichern. Genauer wird hierauf im **Kapitel 9.2.1 Speicherung der User Id bei Runs** eingegangen.

### C4: Schutz vor „schlechten Pipelines“

*Endlosschleifen* oder *Deadlocks* in den Runs zu erkennen, ist aufgrund des *Halteproblems* nicht möglich.
Die einfachste Option zum Schutz vor *Endlosschleifen* und *Deadlocks* wäre ein *Timeout*, durch welchen ein Run nach einer definierten Zeit abgebrochen wird. Das Problem ist jedoch, einen passenden Wert zu finden, da es auch sehr lange Runs geben kann. Möglich wäre es jedoch, den Nutzer selbst optional einen Wert für den *Timeout* wählen zu lassen. Dies würde ihn vor Kosten durch einen Run in Endlosschleife schützen.

Ein solcher *Timeout* wäre generell wichtig, da der Nutzer bisher keine Möglichkeit hat, Runs abzubrechen.

### D1: Quota Limits für Projekte

In dieser Anforderung geht es darum, dem Nutzer die Möglichkeit zu bieten, selbst Limits für die Quota Nutzung auf Projekt-Ebene festzulegen.
Die Umsetzung dieser Anforderung wäre mit größerem Aufwand verbunden.
Zunächst bräuchte es eine entsprechende Komponente zur Auswahl dieses Limits im *Frontend*.
Nun stoßen wir jedoch auf ein Problem, da die Quota Verwaltung eigentlich komplett im *Pipeline-Service* stattfinden soll, dieser jedoch kein Wissen über Projekte hat, sondern nur über Instanzen. Die Zuordnung von Instanzen zu Projekten findet im *Hub-Backend* statt. Um dieses Problem zu lösen, gäbe es verschiedene Ansätze, von denen jedoch keiner für uns zufriedenstellend ist.

**Option 1:** Wir erstellen im *Pipeline-Service* eine Projekt Komponente mit zugehöriger Datenbanktabelle, sodass der *Pipeline-Service* auch über Projekte Bescheid weiß.
Diese Möglichkeit stellt für uns jedoch keine Option dar, denn das würde zu einer deutlichen Erhöhung der Komplexität im *Pipeline-Service* führen wie auch zu redundanten Daten aufgrund der doppelten Projekt Tabelle in der Datenbank, sowie zu deutlich mehr Datentransfer zwischen *Hub-Backend* und *Pipeline-Service*. Denn der *Pipeline-Service* müsste über neue Projekte informiert werden.

**Option 2:** Die *Project* Datenbanktabelle des *Hub-Backends* wird um die Spalten *pointsLimit* und *storageLimit* ergänzt, welche die Quota Limits des Projektes speichern.
Dies hätte zur Folge, dass das *Hub-Backend* sich nun zusätzlich um die Quota Verwaltung kümmern würde, was eigentlich logisch getrennt im *Pipeline-Service* stattfinden sollte.
Zudem bräuchten wir in der *Project* Datenbanktabelle zwei weitere Spalten, in welchen die Quota Nutzung des Projekts gespeichert wird. Denn sonst müsste beim Starten eines Runs zunächst für alle Runs aller Instanzen dieses Projektes, die Punktenutzung und Speicherplatzbelegung summiert werden, um so zu überprüfen, ob eines der Quota Limits des Projektes erreicht ist.
Die Quota Nutzung der Projekte im *Hub-Backend* zu speichern, würde jedoch zur Folge haben, dass der *Pipeline-Service* nach Abschluss eines Runs dessen Quota Informationen an das *Hub-Backend* schicken muss, um die Informationen zur Quota Nutzung des Projektes zu aktualisieren.
Bisher ist die Kommunikation zwischen *Hub-Backend* und *Pipeline-Service unidirektional*. Das würden wir mit dieser Option ändern, weswegen sie für uns ebenso nicht tragbar ist.

Angenommen, wir könnten dieses Problem lösen, so würden wir den Vorschlag aus der Literatur aus **Kapitel 2.2** nutzen und diesen für unseren Zweck entsprechend abwandeln. Wir würden somit für jede Ressource, welche limitierbar sein soll, dem Tupel ein Limit hinzufügen.
Das dort vorgestellte System könnten wir dabei umkehren und nicht mit Ausnahmen, sondern mit Verschärfungen arbeiten.
Jede Ebene der Begrenzung bekäme ihr eigenes Tupel. *Ebene 0* würde hierbei die Beschränkung auf Nutzerebene darstellen, deren Beschränkungen durch den Plan des Nutzers vorgegeben sind und in jedem Falle gelten und nicht durch eine andere Ebene überschrieben werden können.
Die nächste Ebene wäre dann *Ebene 1*, welche Beschränkungen pro Projekt darstellt, die lediglich strenger als diejenigen von *Ebene 0* sein können.
Wir ändern das Quota System aus **Kapitel 2.2** also so ab, dass auf der jeweiligen Ebene auch immer die Beschränkungen aus niedrigeren Ebenen gelten und diese jeweils nur verschärft werden können je höher eine Ebene liegt.

### D2: Abrechnung nach Einheitspreis bei Überschreitung des Quota Limits

Die Abrechnung nach Einheitspreis bei Überschreitung des Quota Limits würde den *UNLIMITED* Plan ersetzen. Für die Umsetzung dieser Funktion würden wir eine weitere Spalte in der Datenbanktabelle *User* benötigen, welche angibt, welches Limit der Nutzer bei der Abrechnung nach Einheitspreis angegeben hat. Der *Wert 0* würde hierbei bedeuten, dass der Nutzer diese Funktion nicht aktiviert hat.
Diese Funktion käme komplett ohne Erweiterungen im *Pipeline-Service* zurecht, da das *Hub-Backend* lediglich sicherstellen muss, dass das Einheitspreislimit nicht überschritten wird. Sie muss den Preis nur anhand der genutzten Punkte, bzw. des belegten Speicherplatzes mit den Quota Limits des Plans abgleichen.

### D3: Andere Bepreisung auf privater Infrastruktur

Die Ausführung von Runs auf privater Infrastruktur kann logischerweise nicht zur Quota Nutzung des jeweiligen Nutzers dazuzählen, da hierbei nicht die JValue Infrastruktur genutzt wird. Die Möglichkeit Runs auf privater Infrastruktur auszuführen, gibt es bei JValue derzeit noch nicht. Dennoch wollen wir kurz darauf eingehen, wie eine Bepreisung hierfür ausschauen könnte.
Wenn ein Nutzer seine Runs auf privater Infrastruktur ausführen möchte, so bieten wir hierfür einen gesonderten Plan an, welcher keine Quota Nutzung oder lediglich die des kostenlosen Planes enthält.
Nutzer dieses Plans können nun beliebig viele Runs auf ihrer Infrastruktur ausführen und zahlen nur einen monatlichen Preis für die Verwaltung der Runs, wie beispielsweise auf deren geplante Ausführung.

Der Preis für solch einen Plan würde wahrscheinlich individuell ausgehandelt werden, da die Ausführung auf privater Infrastruktur in der Regel bei Firmen stattfindet.

### D5: Erinnerungsmail bei Quota Limit fast erreicht

Die Implementierung der Funktion, dem Nutzer eine Mail zu schicken, sollte sein Quota Limit fast erreicht sein, stellt sich als nicht sonderlich aufwendig dar.
Bei Aktualisierung der Quota Informationen eines Nutzers muss überprüft werden, ob der vordefinierte Schwellenwert erreicht wurde. In diesem Falle wird die Erinnerungsmail geschickt. Hierbei wäre es sinnvoll zu speichern, ob diesen Monat bereits eine Erinnerungsmail verschickt wurde, damit ein Nutzer ab Erreichung des Schwellenwertes nicht nach Ausführung jedes neuen Runs eine Erinnerungsmail bekommt.
Grundvoraussetzung für diese Funktion wäre selbstverständlich die Speicherung der Mailadressen der Nutzer.
Diese Funktion ist besonders sinnvoll, sobald JValue die automatisch geplante Ausführung von Runs ermöglicht, da in diesem Falle der Nutzer die Runs nicht mehr manuell starten muss. Somit muss er dafür die JValue Website nicht mehr aufrufen und sieht daher seine Quota Nutzung nicht mehr.

### D6 Firmenaccounts

Die Anforderung D6 beschreibt die Möglichkeit, Firmenaccounts zu erstellen. Hierbei gäbe es einen Firmenaccount, auf welchen die Abrechnung bzw. das Abo laufen würde und mit diesem könnten mehrere normale Nutzeraccounts verbunden werden.
Wir würden hierfür die komplette Nutzer Verwaltung, wie sie bisher bei JValue existiert, abändern. Zunächst führen wir sogenannte *Billing User* ein und erstellen dabei eine eigene Datenbanktabelle, in welcher alle dazu notwendigen Informationen, inklusive der Zahlungsinformationen, gespeichert werden. Jeder normale Nutzer ist dann einem *Billing User* zugeordnet, sodass wir nun auch die Informationen zu den Plänen der Nutzer im *Billing User* speichern. Hierbei wäre es auch möglich, dem Nutzer das Teilen seines Plans und somit auch seiner Quota Kontingente zu ermöglichen, indem die anderen Nutzer dann mit seinem *Billing User* verknüpft würden. Sollte es einen *UNLIMITED* Plan zur Auswahl geben, muss hierbei die Möglichkeit zur Teilung des Plans begrenzt werden.
Bei den Quota Informationen könnte man sich nun überlegen, ob diese für den aktuell eingeloggten Nutzer angezeigt werden sollen oder übergreifend für alle Nutzer, die dem *Billing User* zugeordnet sind.
Je nachdem würden in den Quota Datenbanktabellen nur die übergreifenden Quota Informationen gespeichert oder, falls man die Quota Informationen auch einzeln für den normalen Nutzer anzeigen möchte, die *Billing User* Quota Infor-

mationen aus denen der einzelnen Nutzer addiert werden.

### D7 Admins können Quota Limits von Nutzern ändern

Um Admins die Möglichkeit zu geben, die Quota Limits eines Nutzers ändern zu können, müsste es zunächst Admin-Accounts geben. Die *User* Datenbanktabelle müsste somit um eine Spalte erweitert werden, welche angibt, ob der Nutzer Admin-Rechte besitzt. Ebenso müsste man eine *Frontend* Komponente entwickeln, in welcher Admin-Nutzer andere Nutzer suchen und deren Quota Limits ändern können.
Die Funktionalität zur Suche von Nutzern nach Nutzername existiert bereits in der *Search Komponente*.
Da wir jedoch mit Plänen arbeiten und daraus das Quota Limit lesen, lassen sich die Quota Limits eines Nutzers nicht einzeln ändern, sondern lediglich sein Plan. Hierfür benötigen wir im *Hub-Backend* einen *API-Endpunkt*, welcher die *User Id* des Nutzers, dessen Plan geändert werden soll, sowie die *Plan Id*, auf welchen Plan geändert werden soll, übergeben bekommt. Hierbei muss dann auch nochmal überprüft werden, ob der Nutzer, der den *API-Request* geschickt hat, Admin Rechte besitzt.

### D8: Priorisierung der Runs von Nutzern mit höherer Abo-Stufe

Die Funktion, Runs von Nutzern mit höherer Abo-Stufe zu priorisieren, würde eine Warteschlange mit Priorisierung im *Pipeline-Service* oder der *Runtime-Simple* benötigen. Hierbei würden wir jedem Run eine Zahl zuweisen, welche die Priorisierung des Runs angibt und nach welcher die Warteschlange sortiert wird.
Dabei ist es jedoch wichtig, darauf zu achten, dass auch niedriger priorisierte Runs an die Reihe kommen. Eine Möglichkeit wäre, die Prioriätszahl aus bisheriger Wartezeit und Priorität des Runs, also Priorität der Abo-Stufe des zugeordneten Nutzers, zu bilden.
Da jedoch aktuell die Kommunikation zwischen *Pipeline-Service* und *Runtime-Simple* überarbeitet wird, konnten wir dieses Feature zum jetzigen Zeitpunkt nicht umsetzen.

### D9 Andere Priorisierung bei ad hoc Ausführungen

Die Funktion ad hoc Ausführungen, wie beispielsweise im Editor, anders zu priorisieren, könnte gemeinsam mit der zuvor vorgestellten Anforderung D8 kombiniert werden, indem ad hoc Ausführungen ein vordefinierter Wert für die Priorisierung zugeordnet wird.

### D10 Button für fehlgeschlagene Runs

Hierbei geht es darum, ob fehlgeschlagene Runs auch zur Quota Nutzung eines Nutzers zählen sollten oder, ob der Nutzer hierfür einen Button bekommen sollte, mit welchem er die Punkte von fehlgeschlagenen Runs zurückfordern könnte. Die Speicherplatzbelegung eines fehlgeschlagenen Runs ist hierbei so gering, dass wir diese hier nicht berücksichtigen. Zudem könnte der Nutzer diesen Run auch löschen, um den Speicherplatz wieder freizugeben.
Bisher zählen nur fehlgeschlagene Runs zur Quota Nutzung, bei welchen der Fehler beim Ausführen des Pipeline Codes auftritt.
Für einen Button zum Zurückfordern der Punkte eines fehlgeschlagenen Runs benötigen wir zunächst einen Button im *Frontend*, der bei fehlgeschlagenen Runs angezeigt wird, sowie im *Hub-Backend* und im *Pipeline-Service* jeweils einen *API-Endpunkt*, über welchen das Punkte Quota des Nutzers dekrementiert wird.
Zudem benötigen wir eine zusätzliche Spalte in der Run Tabelle, welche angibt, ob beim jeweiligen Run die Punkte bereits zurückgefordert wurden, denn für diesen Fall darf dieser Run bei der tagesgenauen Berechnung der Punktenutzung in der Anzeige im Diagramm nicht mehr berücksichtigt werden.
Nun stellt sich noch die Frage, inwiefern man den Missbrauch des Buttons verhindern kann. Denn dieser soll vom Nutzer nur genutzt werden, wenn dieser sich sicher ist, dass der Fehler nicht auf seiner Seite, also beispielsweise nicht an seinem Code, liegt.
Eine Option hierfür wäre es, den Button zunächst anzubieten, und nach einer gewissen Zeit dessen Nutzung zu analysieren. Sollte sich dabei herausstellen, dass der Button deutlich zu oft verwendet wurde, muss zunächst analysiert werden, ob es ein allgemeines Problem bei der Ausführung von Runs gibt, weswegen dieser Button so oft betätigt wurde, oder ob er von den Anwendern ausgenutzt wurde, obwohl der Fehler auf ihrer Seite lag. Im letzteren Fall muss der Button wieder entfernt werden oder die im Folgenden vorgestellte Option genutzt werden.
Eine andere, aber aufwändigere, Option stellt die Speicherung der Nutzung des Buttons pro Nutzer dar. Hierbei würden wir in der *Computation Quota* Datenbanktabelle zwei Spalten hinzufügen, eine, die die gesamte Anzahl an fehlgeschlagenen Runs eines Nutzers pro Monat speichert, sowie eine, die die Häufigkeit der Nutzung des Buttons durch diesen Nutzer in diesem Monat speichert. Hieraus kann schließlich ein Verhältnis gebildet und beispielweise der Button nur angezeigt werden, wenn dieses Verhältnis die letzten drei Monate nicht über einem gewissen Schwellenwert lag.
Die Einführung dieser Funktion bietet auch noch einen weiteren großen Vorteil, denn ein Anstieg bei der Nutzung des Buttons kann auf ein allgemeines Problem bei der Ausführung von Runs hindeuten. So können mögliche Probleme frühzeitiger entdeckt und behoben werden.

**D11: Pipeline Runs in Development (DEV)-Phase**

Pipeline Runs in der *DEV Phase* nicht zu bepreisen, ist der Kerngedanke der Anforderung D11. Es gibt jedoch bei JValue bisher keinerlei Unterteilung in verschiedene Phasen. Deshalb ist diese Anforderung aktuell nicht umsetzbar. Wie dies umzusetzen ist, sollte es in JValue in Zukunft die Unterteilung in Phasen geben, hängt stark von der Implementierung der Phasen ab. Daher können wir hier keine Details zu einer möglichen Umsetzung darstellen.

## 9.2 Weitere Möglichkeiten zur Optimierung des Quota Systems

### 9.2.1 Speicherung der User Id bei Runs

Aktuell werden in der *Run* Datenbanktabelle keine Informationen zum Nutzer, der den Run gestartet hat, gespeichert. Die *User Id* wird in der Tabelle der Instanzen gespeichert. In der *Run* Datenbanktabelle wird wiederum die *Id* der zugehörigen Instanz abgelegt.
Bisher war das auch passend, da der Nutzer eines Runs nicht von der *Runs Komponente* im *Pipeline-Service* benötigt wurde.
Nun benötigen wir jedoch die *User Id* beim Speichern der Quota Informationen eines neuen Runs, sowie auch beim Löschen eines Runs, zur Aktualisierung der Quota Informationen. Ebenso muss die *Quota Komponente* beispielsweise für die tagesgenaue Bestimmung der Punktenutzung zunächst alle Instanzen eines Nutzers abfragen und dazu dann alle Runs. Um den Nutzer zu einem Run oder alle Runs eines Nutzers zu erhalten, muss zunächst immer erst der Weg über die Instanz gegangen werden.
Aufgrund dessen schlagen wir vor, der *Run* Datenbanktabelle die Spalte *User Id* hinzuzufügen, sodass nun der Nutzer zum jeweiligen Run gespeichert wird. Zwar führt dies zu Redundanz innerhalb des *Pipeline-Service*, da der Nutzer bereits in der Tabelle der Instanzen gespeichert wird. Die Redundanz bringt jedoch zum einen einen Performanz Vorteil durch die wegfallenden Methodenaufrufe an den *Instances Service*, um die Instanz und somit den Nutzer des Runs zu erhalten. Zum anderen wird die *Runs Komponente* und vor allem die *Quota Komponente* deutlich weniger komplex und somit einfacher zu warten.
Zudem könnte mit dieser Änderung die *Auto Delete* Funktion nun einfach und performant umgesetzt werden, da hierfür nun nicht mehr alle Runs aller Instanzen eines Nutzers geholt werden müssten, um hiervon die Ältesten zu bestimmen. Anstatt dessen können die ältesten Runs per Abfrage auf die *Run* Datenbanktabelle in der Datenbank bestimmt werden, da diese nun die *User Id* speichert.

### 9.2.2 Hard Delete der Run Resultate

Wie bereits im **Kapitel Implementierung** erwähnt, haben wir den *Hard Delete* der Run Resultate noch nicht implementiert.
Dies ist jedoch eine wichtige Funktion, um die Speicherplatzbelegung von JValue und die damit verbundenen Kosten möglichst niedrig zu halten.
Die optimale Lösung zur Löschung der Run Resultate wäre eine Art Papierkorb, wie dies aus den meisten Dateisystemen der gängigen Betriebssysteme bekannt ist. Das heißt also, der Nutzer bekäme die Möglichkeit, Runs mit deren Resultaten zu löschen. Daraufhin werden Diese per *Soft Delete* gelöscht, jedoch in einer *Frontend Komponente* namens „Deleted runs of the last 30 days“ weiterhin angezeigt, sodass der Nutzer die Möglichkeit hat, diese innerhalb von 30 Tagen nach Löschung wiederherzustellen. Jedoch nur, wenn dem Nutzer genug Speicherplatz für die Wiederherstellung zur Verfügung steht. Nach 30 Tagen werden die Run Resultate endgültig gelöscht und nur die Metadaten zum Run bleiben erhalten.
Um dies umzusetzen, benötigen wir eine **Frontendkomponente**, welche die gelöschten Runs der letzten 30 Tage anzeigt, sowie zwei *API-Endpunkte* im *Hub-Backend* und im *Pipeline-Service*. Einen zum Wiederherstellen eines gelöschten Runs, sowie Einen, welcher alle Runs der letzten 30 Tage zurückgibt, die vom jeweiligen Nutzer gelöscht wurden. Zudem benötigen wir die Funktionalität zum *Hard Delete* der Run Resultate, sowie einen *Scheduler*, der dies 30 Tage nach *Soft Delete* erledigt.

### 9.2.3 Anzeige der Quota Informationen pro Projekt

Für die Anzeige der Quota Informationen im Projekt benötigen wir zunächst eine entsprechende *Frontend* Komponente für die *Project Page*, sowie einen *API-Endpunkt* im *Hub-Backend*, welcher die Informationen hierfür liefert.
Bei Aufruf dieses API-Endpunkts müssten zunächst alle Instanzen zu diesem Projekt bestimmt werden und für jede Instanz die *getPointsForInstance*, sowie die *getStorageForInstance* Methode des *Quota Service* aufgerufen werden. Die Rückgabewerte der *getPointsForInstance* Methodenaufrufe müssten jeweils addiert werden. Selbiges gilt für die Rückgabewerte der *getStorageForInstance* Aufrufe. Die beiden Summen sind dann die Quota Informationen dieses Projekts und können an das *Frontend*, welches den *API Request* geschickt hat, zurückgeliefert werden.

### 9.2.4 Erweiterung der einbezogenen Ressourcen

Bisher werden der Pipeline Score und so auch die Punkte für einen Run anhand der Nutzung der Ressourcen CPU und Arbeitsspeicher berechnet. Wie bereits im Kapitel **Kapitel Quota Modell** erwähnt, können in die Berechnung des Pipeline Scores beliebig viele weitere Ressourcen miteinbezogen werden. Ein Beispiel

hierfür wäre der Netzwerkverkehr. Die Daten, welche bei der Ausführung eines Runs verarbeitet werden, werden in der Regel aus dem Internet geladen, sodass dafür Netzwerkkapazitäten zur Verfügung gestellt werden müssen, wodurch Kosten für JValue entstehen. Aufgrund dessen wäre es sinnvoll den Netzwerkverkehr bei Ausführung eines Runs zu messen und diesen mit in den Pipeline Score einzubeziehen.

# 10 Schlussfolgerung

Ziel dieser Arbeit war es, ein Quota Modell zur Limitierung der Verwendung von Ressourcen bei der Ausführung von Code auf Cloud Infrastruktur zu entwerfen und dieses anschließend für den JValue Hub zu implementieren.
Hierfür haben wir zunächst die Lösungsansätze anderer Tools, sowie aus der Literatur verglichen.
Anschließend haben wir Anforderungen an ein Quota Modell für JValue und für dessen Umsetzung definiert.
Die Erkenntnisse aus dem Vergleich verschiedener Lösungsansätze, sowie der definierten Anforderungen haben wir verwendet, um ein eigenes Quota Modell zu entwerfen, welches dem Nutzer volle Flexibilität bietet und dennoch die Bereitstellung der Infrastruktur für den Anbieter leicht planbar macht.
Anhand der erarbeiteten Anforderungen, sowie des vorgestellten Quota Modells haben wir anschauliche Frontend Komponenten erstellt, welche die Transparenz des Quota Modells, sowie dessen Verständnis, verbessern.
Daraufhin haben wir zur Umsetzung des Quota Modells eine allgemeine Architektur inklusive der nötigen Komponenten und der benötigten REST API erarbeitet und diese ebenso für JValue angepasst.
Danach folgt eine Referenzimplementierung der Frontendkomponenten, sowie der notwendigen Komponenten im Backend. Die Referenzimplementierung beschreibt, wie wir das Quota Modell bei JValue umgesetzt haben, kann jedoch auch als Vorlage für die Implementierung des Modells in andere Tools dienen.
Anschließend haben wir auf die definierten Anforderungen zurückgeblickt. Hierbei wurden von den *A-Anforderungen* lediglich eine aus architektonischen Gründen nicht erfüllt, von den *B-Anforderungen* alle erfüllt, von den *C-Anforderungen* 3 von 5 erfüllt, sowie von den *D-Anforderungen* eine erfüllt. Die Ziele dieser Arbeit wurden somit, abgesehen von der architektonisch nicht umsetzbaren Anforderung, mehr als erreicht.
Zu den nicht erfüllten Anforderungen haben wir Lösungsvorschläge für eine zukünftige Umsetzung erstellt.
Ebenso haben wir noch Möglichkeiten zur Erweiterung des implementierten Quota Modells vorgestellt, sowie eine mögliche Umsetzung erarbeitet.
Insgesamt haben wir ein allgemeines Quota Modell erstellt, welches für Nutzer als

auch Anbieter der Infrastruktur möglichst positiv gestaltet ist. Dieses Modell haben wir für JValue implementiert, sodass JValue nun die Nutzung der von ihnen bereitgestellten Cloud Infrastruktur begrenzen, aber auch deren Bereitstellung besser planen kann und in Zukunft dafür sorgt, das JValue Projekt profitabel zu machen.

# Anhänge

# A Bestimmung von Konstanten

## A.1 Pipeline Score Formel

Die Gewichtung für die *Pipeline Score* Formel ergibt sich aus den unterschiedlichen Kosten für die Bereitstellung der Ressourcen. Hierbei nutzen wir die Werte von Google *Dataflow* für *Batch Pipelines*, da diese *ETL-Pipelines* am ähnlichsten sind.
Das Verhältnis zwischen genutzten CPU-Stunden und Arbeitsspeicherstunden pro bereitgestelltem GB an Arbeitsspeicher ist dort circa *16* zu *1*. CPU-Stunden sind somit 16 mal teuerer als Arbeitsspeicher Stunden pro bereitgestelltem GB. (Google Dataflow Team, n. d.)
Die Zeit messen wir zwar in Millisekunden, da wir das jedoch sowohl bei CPU als auch Arbeitsspeicher so machen, bleibt das Verhältnis gleich. Lediglich den bereitgestellten Arbeitsspeicher geben wir in MB an, weswegen wir unseren Wert für den Arbeitsspeicher zunächst noch durch *1000* teilen müssen. Hieraus resultiert dann zunächst folgende Formel:

$$s(16, 1, cpu, ram, reservedRam, d) = \frac{16 * cpu + ram * \frac{reservedRam}{1000}}{d}$$

Nun müssen wir noch den Divisionsfaktor $d$ definieren.
Wir nehmen hierfür das *Cars*[1] Beispiel von *Jayvee*, dessen Pipeline wir als Pipeline der Größe $S$ definieren. Wir wollen somit erreichen, dass das *Cars*[1] Beispiel einen *Pipeline Score* von circa *1* erreicht. Als Durchschnittswerte für Pipeline Runs der *Cars* Pipeline bei *10-maliger* Ausführung haben wir bei Arbeitsspeichergröße *4196 MB* eine CPU Zeit von circa *308 ms* und eine RAM Zeit von circa *146 ms* gemessen. Um hieraus nun einen *Pipeline Score* von *1* zu errechnen, müssen wir den Divisionsfaktor $d$ gleich *5540* wählen. Der Pipeline Score wird nun anhand folgender Formel berechnet:

$$s(16, 1, cpu, ram, reservedRam, 5540) = \frac{16 * cpu + ram * \frac{reservedRam}{1000}}{5540}$$

Da sich die Punkte aus den genutzten Ressourcen ergeben, macht es hierbei Sinn, die Preise von *Google Dataflow* zur Ressourcenverwendung zu vergleichen. Jedoch muss dabei beachtet werden, dass die RAM-Zeit und die CPU-Zeit von der genutzten CPU abhängen. Zum groben Vergleich können die Werte jedoch genutzt werden.
Die Ausführung eines Runs, welcher einen Pipeline Punkt benötigt, kostet *JValue*,

[1] https://jvalue.github.io/jayvee/docs/0.3.0/user/examples/cars

wenn man die Werte von *Google Dataflow* (Google Dataflow Team, n. d.) nutzt circa *0,00058 Cent* für die CPU-Zeit und für die RAM-Zeit circa *0,000074 Cent*. Ein *Pipeline-Punkt* kostet *JValue* somit circa *0,000654 Cent*. Für die Umrechnung von *US-Dollar* in *Euro* verwenden wir einen Wechselkurs von *0,93*[2].

## A.2 Kontingente für Pläne

Ebenso müssen wir noch die Werte für die enthaltenen Punkte, sowie den enthaltenen Speicherplatz der jeweiligen Pläne definieren.
Die Implementierung ist so gestaltet, dass flexibel Pläne an den Business Plan angepasst werden.
Folgende Überlegungen dienen lediglich der Demonstration, wie Punktekontingente für verschiedene Nutzergruppen abgeleitet werden können. Marktforschung, die Erstellung eines Businessplans und Nutzer-Feedback sind unerlässlich für eine finale Strukturierung der Pläne, allerdings nicht Gegenstand dieser Arbeit.

### Punktekontingent

Schauen wir uns zunächst das monatliche Punktekontingent an.
Der kostenlose **FREE** Plan soll zum Ausprobieren von *JValue* dienen und den Nutzer von einem Abo eines kostenpflichtigen Planes überzeugen. Somit sollte der kostenlose Plan für *JValue* auch keine großen Kosten verursachen. Den kostenlosen Plan wollen wir somit so ansetzen, dass dadurch eine Ausführung pro Tag einer Pipeline mit Größe $S$ möglich ist. Dies ergäbe dann im Schnitt *30* Punkte. Aufgrund des *Soft Limits* ist auch in Monaten mit 31 Tagen eine Ausführung pro Tag möglich. Wenn man den in **Anhang A.1** berechneten Preis pro Punkt nutzt, so kosten die *30* Punkte des kostenlosen Plans *JValue* circa *0,02 Cent*.
Den **BASIC** Plan setzen wir als Plan für Hobby Entwickler an. Ermöglichen wollen wir mit diesem Plan die Ausführung einer Pipeline der Größe $L$ dreimal am Tag. Dies ergibt dann ein Punktekontingent von *270* Punkten pro Monat, welches wir auf *300* Punkte aufrunden. Für diese *300* Punkte ergibt das Kosten von circa *0,2 Cent* für *JValue* Der **PRO** Plan soll für mittelgroße Anwendungen dienen und hierfür die stündliche Ausführung einer Pipeline mit Größe $L$ ermöglichen. Daraus folgt ein monatliches Punktekontingent von *2160* Punkten, welches wir auf *2500* Punkte aufrunden. Diese *2500* Punkte kosten *JValue* circa *1,63 Cent*.
Nun bleibt noch der **UNLIMITED** Plan mit unendlichem Punkte- und Speicherplatzkontingent. Hierbei ist *JValue* vor keinerlei Kosten geschützt, sodass hierfür ein entsprechend hoher monatlicher Preis gefordert werden muss.

[2] https://www.finanzen.net/waehrungsrechner/us-dollar_euro (abgerufen am 18.02.2024)

**Speicherplatzkontingent**

Nun zum enthaltenen Speicherplatz in den jeweiligen Plänen.
Speicherplatz ist keine teuere Ressource, sodass wir mit dieser großzügig umgehen können. Bei *Google Cloud* beispielsweise kostet *1 GB Standardspeicher* pro Monat *0,020$* und somit *0,0186€* (Google Cloud Team, n. d.).
Wie bereits beschrieben, soll der **FREE** Plan zum Ausprobieren von *JValue* dienen. Aufgrund dessen wollen wir auch hier nicht viel Speicherplatz inkludieren. Wir gewähren hierbei lediglich *10 MB* Speicherplatz, weil der Nutzer zum Ausprobieren von *JValue* nicht viel Speicherplatz benötigt, da er jederzeit Runs löschen kann, um wieder Speicherplatz freizugeben. Hierdurch entstehen *JValue* Kosten von circa *0,02 Cent*. Der kostenlose Plan ist somit in Bezug auf die Speicherplatzbelegung sehr günstig.
Der **BASIC** Plan wiederum soll zur Hobby-Nutzung dienen. Bei den Punkten haben wir uns an ein 3-mal tägliches Ausführen einer *L-Pipeline* gehalten. Wenn wir großzügig von einer Speicherplatzbelegung von *1 MB* pro Run ausgehen, so würden für die Speicherung der Resultate von einem Monat mit 30 Tagen *90 MB* nötig sein. Da Speicherplatz jedoch keine teuere Ressource darstellt, wollen wir dem Nutzer eine Speicherung der Resultate eines halben Jahres ermöglichen, was zu einem Speicherplatzkontingent von *540 MB* führt, welches wir auf *600 MB* aufrunden. Wenn wir für die Speicherung der Daten die Google Cloud verwenden würden, so würde die Speicherbereitstellung für den **BASIC** Plan circa *1,1 Cent* pro Monat kosten.
Für den **PRO** Plan müssen wir schon mehr Speicherplatz zur Verfügung stellen. Hierbei sind wir von einer stündlichen Ausführung einer *L-Pipeline* ausgegangen, sodass bei einer Speicherplatzbelegung von *1 MB* pro Run circa *720 MB* nötig wären. Da wir bei diesem Plan von einem professionellen Nutzer ausgehen, wollen wir diesem die Speicherung seiner Daten für *2* Jahre ermöglichen, was zu einem Speicherplatzkontingent von *17.280 MB* führt, welches wir auf *18 GB* aufrunden. Hierdurch entstehen für *JValue* Kosten von circa *34,5 Cent* pro Monat.
Beim **UNLIMITED** Plan ist das Speicherplatzkontingent nicht begrenzt.

Wir kommen somit für den **FREE** Plan auf monatliche Kosten von circa *0,04 Cent* für *JValue*.
Beim **BASIC** Plan belaufen sich die monatliche Kosten auf circa *1,2 Cent*.
Der **PRO** Plan kostet *JValue* monatlich circa *36 Cent*.

# B Interaktionen bei Start eines Runs

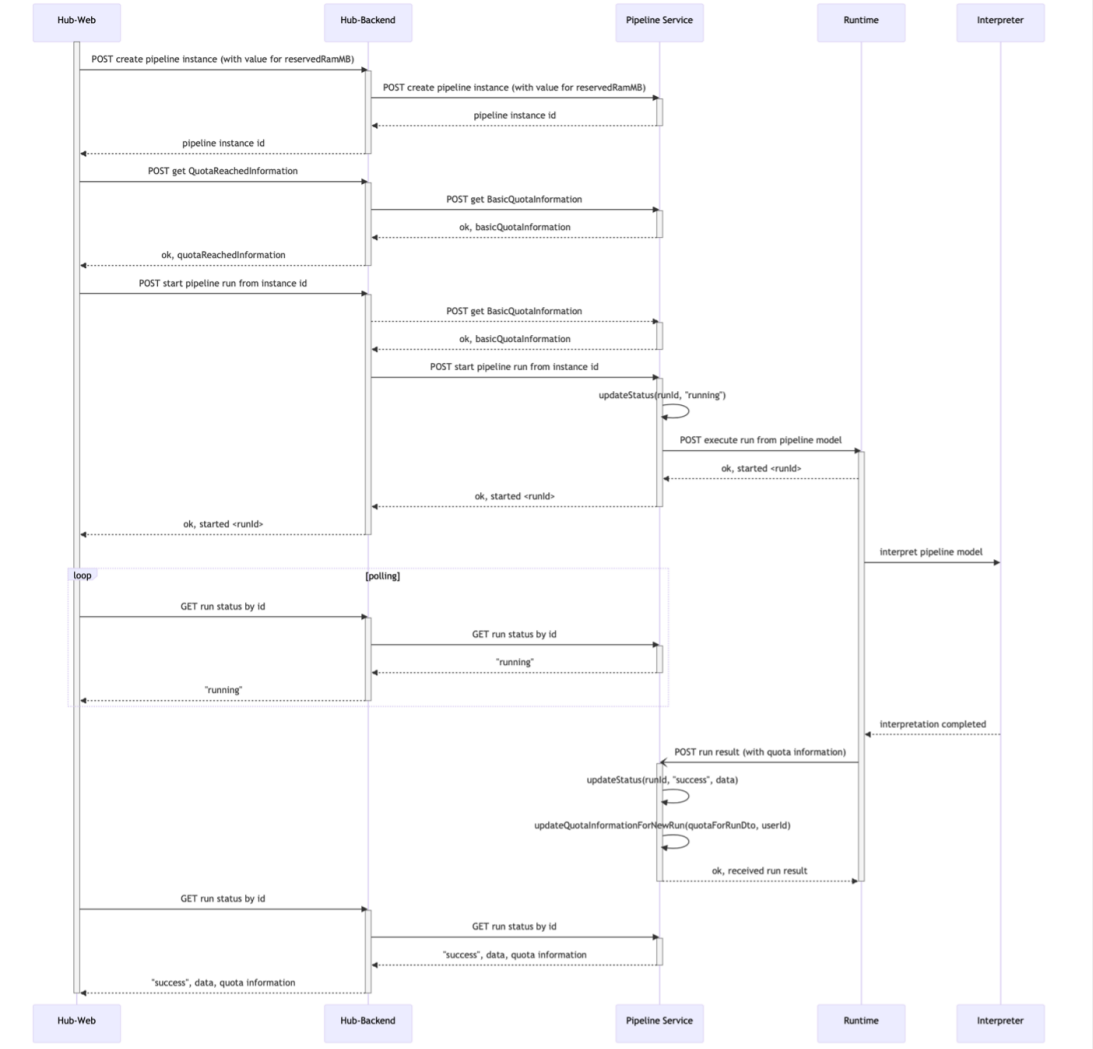


**Abbildung 1:** Sequenz Diagramm zum Starten eines Runs auf der Pipeline Page

Im Diagramm wird nur dargestellt, was zum Starten eines Runs auf der *Pipeline Page* wichtig ist. Natürlich werden beim Öffnen der *Pipeline Page* beispielsweise auch Informationen zur Instanz geladen. Darauf wollen wir aber nicht eingehen. Wir gehen hierbei auch nur vom positiven Fall aus, also dass der Nutzer noch kein Quota Limit erreicht hat.
Zunächst muss auf der *Project Page* eine Pipeline Instanz erstellt werden.
Anschließend, bei Aufruf der *Pipeline Page*, werden die *QuotaReachedInformationen* vom *Hub-Backend* abgefragt, um daraus auszulesen, ob der *Create Run Button* oder der *Change Plan Button* angezeigt werden soll. Das *Hub-Backend*

holt dafür die *BasicQuotaInformation* vom *Pipeline Service*.
Daraufhin kann im Falle der Anzeige des *Create Run Buttons* darüber ein neuer Run gestartet werden. Hierzu wird ein *POST Request* an den *Run Endpunkt* des *Hub-Backends* geschickt, welcher auch nochmal überprüft, ob der Nutzer schon ein Quota Limit erreicht hat.
Falls nicht, wird ein *POST API-Request* an den *Run Endpunkt* des *Pipeline-Service* geschickt, wo der Run in der Datenbanktabelle *Run* im *Pipeline-Service* gespeichert und hierbei der Status *Running* angegeben wird.
Anschließend wird ein *POST API-Request* an die *Runtime-Simple* geschickt, woraufhin der Run gestartet und vom *Interpreter* interpretiert wird.
Währenddessen fragt das *Hub-Web* in regelmäßigen Abständen per *GET API-Request* an den *Run Endpunkt* des *Hub-Backends* den Status des Runs ab, welcher wiederum einen *GET API-Request* an den *Run Endpunkt* des *Pipeline-Service* sendet, um ebenfalls den *Run Status* abzufragen.
Sobald der Run abgeschlossen ist, wird ein *POST API-Request* an den *Results Endpunkt* des *Pipeline-Service* geschickt. Dieser ruft die *update* Methode des *Runs Service* auf, woraufhin der Eintrag zu diesem Run in der Datenbank entsprechend aktualisiert und dessen Status auch auf *success* geändert wird.
Ebenso werden die Quota Informationen des Nutzers, der den Run gestartet hat, per Aufruf einer Methode des *Quota Service* aktualisiert.
Wenn das *Hub-Web* nun das nächste Mal den *GET API-Request* ans *Hub-Backend* sendet, bekommt es die Informationen zum abgeschlossenen Run zurück, die das *Hub-Backend* wiederum per *GET API-Request* vom *Pipeline-Service* erhält.

# Literaturverzeichnis


AWS Team. (n. d. a). Amazon Managed Workflows for Apache Airflow Preise [Zuletzt aufgerufen am 12.02.2024]. https://aws.amazon.com/de/managed-workflows-for-apache-airflow/pricing

AWS Team. (n. d. b). AWS Data Pipeline – Preise [Zuletzt aufgerufen am 14.02.2024]. https://web.archive.org/web/20220826190324/https://aws.amazon.com/de/datapipeline/pricing/

AWS Team. (n. d. c). Memory and computing power [Zuletzt aufgerufen am 12.02.2024]. https://docs.aws.amazon.com/lambda/latest/operatorguide/computing-power.html

Azure Data Factory Team. (n. d.). Data Pipeline pricing [Zuletzt aufgerufen am 12.02.2024]. https://azure.microsoft.com/en-us/pricing/details/data-factory/data-pipeline/

Barisits, M. (2015). Resource control in ATLAS distributed data management: Rucio Accounting and Quotas. *Journal of Physics, Conference series*, *664*(6).

GitHub Team. (n. d.). Informationen zur Abrechnung für GitHub Actions [Zuletzt aufgerufen am 12.02.2024]. https://docs.github.com/de/billing/managing-billing-for-github-actions/about-billing-for-github-actions

Google Cloud Team. (n. d.). Cloud Storage – Preise [Zuletzt aufgerufen am 12.02.2024]. https://cloud.google.com/storage/pricing?hl=de

Google Dataflow Team. (n. d.). Dataflow – Preise [Zuletzt aufgerufen am 12.02.2024]. https://cloud.google.com/dataflow/pricing?hl=de

Goyal, M. (2021). Using Apache ECharts with React and TypeScript [Zuletzt aufgerufen am 12.02.2024]. https://dev.to/manufac/using-apache-echarts-with-react-and-typescript-353k

Mell, P., & Grance, T. (2011). *The NIST Definition of Cloud Computing* (Techn. Ber.). National Institute of Standards und Technology. https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-145.pdf