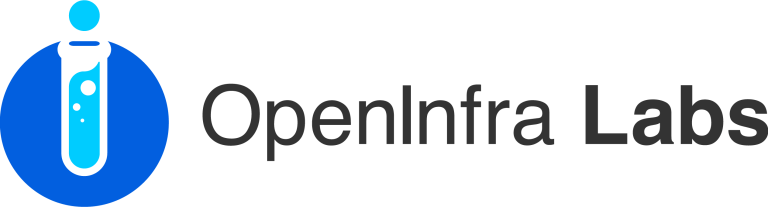
****



**Caerus Near Data Processing – User Defined Function Support**

**System Design Document**

**Version 5.1**

**November 08, 2021**

*Yong Wang*

Table of Contents

[1 Introduction 6](#_Toc87256073)

[1.1 Purpose of Document 6](#_Toc87256074)

[1.2 Scope 6](#_Toc87256075)

[1.2.1 Overview 6](#_Toc87256076)

[1.2.2 Goals 9](#_Toc87256077)

[1.2.3 Competitive Analysis 9](#_Toc87256078)

[1.2.4 Benefits 11](#_Toc87256079)

[1.2.5 Major Features 11](#_Toc87256080)

[1.3 Organization of This Document 12](#_Toc87256081)

[1.4 References 12](#_Toc87256082)

[1.5 Definitions and Acronyms 12](#_Toc87256083)

[2 Design Considerations 12](#_Toc87256084)

[2.1 Assumptions and Dependencies 12](#_Toc87256085)

[2.2 Goals and Guidelines 13](#_Toc87256086)

[2.3 Architectural Strategies 13](#_Toc87256087)

[3 System Architecture 13](#_Toc87256088)

[3.1 Storage Protocols and Events to Support UDF Pushdown 15](#_Toc87256089)

[3.1.1 Object Storage (S3 Protocol) 15](#_Toc87256090)

[3.1.2 HDFS (WebHDFS Protocol) 17](#_Toc87256091)

[3.2 Execution Sequences for Major Workflows 18](#_Toc87256092)

[3.2.1 Programmable Storage – Event-Driven Storage UDF Workflow (Serverless) 18](#_Toc87256093)

[3.2.2 Programmable Storage – Direct Invocation of UDF Workflow (Serverless) 22](#_Toc87256094)

[3.2.3 Programmable Storage – Event-Driven Storage UDF Workflow (Standalone container) 23](#_Toc87256095)

[3.2.4 Programmable Storage – Direct Invocation of UDF Workflow (Standalone container) 24](#_Toc87256096)

[3.3 Deployment Topology: Physical/Virtual Resources 24](#_Toc87256097)

[3.4 Network Architecture 24](#_Toc87256098)

[3.5 Data Architecture 24](#_Toc87256099)

[4 Design Decomposition 25](#_Toc87256100)

[4.1 Caerus NDP Service 26](#_Toc87256101)

[4.1.1 Inputs 26](#_Toc87256102)

[4.1.2 Outputs 26](#_Toc87256103)

[4.1.3 Internals 26](#_Toc87256104)

[4.2 Caerus UDF Service 29](#_Toc87256105)

[4.2.1 Inputs 29](#_Toc87256106)

[4.2.2 Outputs 29](#_Toc87256107)

[4.2.3 Internals 29](#_Toc87256108)

[4.3 Caerus Event Listener Service 30](#_Toc87256109)

[4.3.1 Inputs 30](#_Toc87256110)

[4.3.2 Outputs 30](#_Toc87256111)

[4.3.3 Internals 30](#_Toc87256112)

[4.4 Caerus UDF Registry Service 31](#_Toc87256113)

[4.4.1 Inputs 31](#_Toc87256114)

[4.4.2 Outputs 31](#_Toc87256115)

[4.4.3 Internals 31](#_Toc87256116)

[4.5 Caerus UDF Registry 33](#_Toc87256117)

[4.6 Caerus Faas (Function-As-A-Service) 33](#_Toc87256118)

[4.7 Caerus S3 CLI with UDF Support 34](#_Toc87256119)

[4.8 Caerus HDFS CLI with UDF Support 34](#_Toc87256120)

[4.9 Caerus Sample UDF Functions 34](#_Toc87256121)

[5 UDF Pushdown by Compute Platforms 35](#_Toc87256122)

[5.1 Spark 35](#_Toc87256123)

[5.1.1 Introduction 35](#_Toc87256124)

[5.1.2 Partial UDF Pushdown – Translatable UDF 36](#_Toc87256125)

[5.1.2.1 Options 36](#_Toc87256126)

[5.1.2.2 Performance Measurement and Results 41](#_Toc87256127)

[5.1.3 Whole UDF Pushdown (TBD) 44](#_Toc87256128)

[6 Software Architecture Quality Attributes 45](#_Toc87256129)

[6.1 DevOps 45](#_Toc87256130)

[6.2 Security and Privacy 45](#_Toc87256131)

[7 Testing: Unit Tests, System Tests and Security Tests 45](#_Toc87256132)

[8 Future Enhancements 45](#_Toc87256133)

[9 Open Issues 45](#_Toc87256134)

List of Figures

[Figure 1. Caerus DNP-UDF Use Case Example 8](#_Toc87256135)

[Figure 2. Caerus NDP-UDF Architecture 14](#_Toc87256136)

[Figure 3.. Automatic Event-Driven UDF Invocation 22](#_Toc87256137)

[Figure 4. Direct Invocation of UDF 23](#_Toc87256138)

[Figure 5. Caerus NDP Service 27](#_Toc87256139)

[Figure 6. Caerus NDP Service Class Diagram 28](#_Toc87256140)

[Figure 7. Caerus UDF Service Class Diagram 29](#_Toc87256141)

[Figure 8. Caerus Event Listener Service 31](#_Toc87256142)

[Figure 9. Caerus UDF Registry Service 32](#_Toc87256143)

[Figure 10. Caerus OpenFaas Client 33](#_Toc87256144)

[Figure 11. Caerus S3 CLI 34](#_Toc87256145)

List of Tables

[Table 1 - Acronyms 12](#_Toc87256146)

**Revision History**

|  |  |  |  |
| --- | --- | --- | --- |
| Date | Version | Author(s) | Change Summary |
| 02/17/2021 | 1.0 | Yong Wang | Initial draft of the document: focus on Programmable Storage use cases |
| 03/17/2021 | 2.0 | Yong Wang | Incorporated review comments |
| 04/01/2021 | 3.0 | Yong Wang | Modified Competitive Analysis section with the latest released AWS S3 Object Lambda feature |
| 08/01/2021 | 4.0 | Yong Wang | Adding HDFS UDF support |
| 08/06/2021 | 4.1 | Yong Wang | Adding events details for S3 and HDFS |
| 08/28/2021 | 4.2 | Yong Wang | Adding Spark UDF pushdown: Caerus UDF Compiler |
| 10/11/2021 | 5.0 | Yong Wang | Finalizing Spark UDF (compiler) pushdown including performance measurement |
| 11/08/2021 | 5.1 | Yong Wang | Adding I/O reduction portion for Spark UDF (compiler) pushdown |
|  |  |  |  |

# Introduction

## Purpose of Document

This software design document describes the architecture and system design of Caerus NDP-UDF support.

## Scope

### Overview

Modern BDA/AI platforms, such as Spark, Presto, Hive (w/ Tez) and TensorFlow etc., widely used disaggregated architecture and compute-side in-memory computations. Although they have high performance in most of the case and scale very well, they do have following problems:

1. **Disaggregated architecture**: the computation and storage are disaggregated, so they can scale independently. The side effect of the disaggregated architecture is that the networks connecting the computation and storage layers can be a major performance bottleneck.
2. **In-memory Computation**: huge memory resources on compute side allow applications to have very good performance via distributed processing, however, with Big Data become bigger, compute platform often run into problem that compute memory can’t hold all the data (often refers to ‘too big to eat” problem), so techniques like shuffle are often used, this not only further increases storage I/O, network traffic, but also decreases performance significantly.

Near Data Processing is a technique that functionality is pushed down as close to storage as possible, so that unnecessary data transportation between storage and compute are eliminated, thus improve BDA/AI performance and efficiency. Furthermore, it is also more economic to throw more processing power into storage systems rather than invest in high bandwidth network between compute and storage.

To make NDP work, cloud storage backend and enterprise storage systems must be able to accept computation requests, but traditionally this fells out of the scope of storage systems, because they normally only manage storage objects and handle storage related requests. Some cloud and storage vendors started to support “query-able” or ‘computable” storage via features like AWS S3 Select, but it can only support very few standard operations/functions to be pushed down, and still lacks many features, among them, NDP-UDF support is a big ask from customers and no vendor has supported this feature yet.

UDFs are widely used in BDA/AI fields due to following reasons:

1. **Filling Function Gaps**: Although compute platforms like Spark have provided many standard function support, but significant amount of non-standard “functions” needed for different data science use cases (e.g. Kmeans) are still yet to implemented, UDFs are the solution.
2. **Support of New AI Workflows**: The explosion of AI requires simplification of the entire ML/DL process, e.g., we can support inference via a UDF of “predict” as center of a simple SQL statement, or support complex feature engineering operations via a few UDFs. For example, in Spark world, the Spark UDFs are extensively used in ML support either via native [MLlib](https://databricks.com/spark/getting-started-with-apache-spark/machine-learning) or other framework like [MLflow](https://www.mlflow.org/docs/latest/python_api/mlflow.spark.html).
3. **Porting UDFs Across Different Compute Platforms**: Many organizations have developed UDFs with different programming languages for different compute platforms over the years and hope to port or migrate such UDFs into different platforms in native and high performance form.

Unfortunately, UDFs are notoriously known for bad performance. The key reason is that UDFs are normally black-boxes to compute platform optimization, for example, in Spark, UDFs are not part of Catalyst optimization, so UDFs are missing out optimizations like Predicate Pushdown (NDP) etc. Due to this reason, UDFs usage, although strongly desired by customers, are largely limited.

UDFs traditionally refers to the ones supported by different compute platforms (like database, Spark, Presto and Hive etc.) in SQL context, most of them can only run on the compute side and cannot be pushed down to storage side, so NDP-UDF or UDF Pushdown is a new concept. Furthermore, there is strong customer need to let storage system to support any type of UDFs besides the UDFs in SQL:

* + SQL predicate pushdown is the main use case, it is a must-have feature. However, there are more and more use cases can be explored if UDF-like feature can be supported
  + The smartness of the storage layer should not be single purpose. Conversely, the challenge is to enable a storage system to execute general-purpose code close to the data. Such code should be easily deployed to extend the functionalities of the system for handling new offloaded tasks.

In Caerus project scope, UDF (Pushdown) has broader meaning, it covers ANY UDFs that can be pushed to storage side and execute, it can be looked at as in two categories:

1. **Programmable Storage** (or refers to **Rich-Active Storage**): Cloud storage backend and storage system can accept user defined functions *from the user directly* to allow storage to execute to do “in-place” computation within storage network without raw data ever leaving the storage.
2. **UDF Pushdown By Compute Platform** (orrefers to **SQL UDF Pushdown**): This will require that compute platform to be able to pushdown UDFs, and cloud storage backend and storage system can accept user defined functions *from the compute platforms* to allow storage to do “in-place” computation within storage network without raw data ever leaving the storage.

Note: some of the UDF Pushdown By Compute Platform cases might take advantage of Programmable Storage workflow directly, for example, Presto’s UDFs are normally written in Java, are very similar to normal functions, and the compiled jar file can be pushed to the storage side and registered by the user, the UDFs can then be called during run time as part of query, this workflow is almost identical as the Programmable Storage workflow, this workflow can also be confirmed by comparing with cloud vendors’ support of UDFs, for example, Amazon Athena UDF (currently it is Preview stage). However, in Spark case, the Spark UDFs are very specific to Spark internals, and majorly operated at row level that are similar to the lambda functions, and the registration and invocation of UDFs are also internal and specific to Spark, thus the UDF Pushdown by Compute Platform for Spark might have a different workflow than Programmable Storage, but some of the backend portions can still be reused.

The importance to support **Programmable Storage** as described in Caerus can be described as follows:

1. Storage system or cloud storage backend traditionally only handle storage operations, but seldom helping customer to explore value of the stored data, we hope Caerus project can open doors for storage system to become smart storage that can accept any form of user define functions which can do in-place computation within storage layer, allow customers to get more values from their data in a more efficient way.
2. There are more and more use cases can be explored if Programmable Storage UDF feature can be supported.

In terms of NDP-UDF invocation options, traditional thinking and the first option is to via direct invocation which UDF is explicitly called by a caller, this fits a lot of use cases such as both SQL UDF in data analytics and some Programmable Storage use cases, for example, we can send storage request like PUT, COPY etc. with a metadata tag of UDF, so that storage system can invoke this UDF directly while it is doing the storage request. One simple example is that we can upload a large image file via a storage request with a thumbnail UDF metadata, we will create a thumbnail from that large image file and store in the storage system by directly invoking this thumbnail UDF.

However, the second NDP-UDF invocation option, we called it **event-driven UDF invocation**, might provide more powerful use cases and bring out more smartness from storage system that can change people’ perception on storage system. In modern storage systems and cloud storage backend, many of them start to implement a feature called “bucket notification”, vendors and open sources like Amazon S3, GCP Storage (Google Cloud Storage), IBM Cloud Object Storage, Ceph, MinIO etc. all have this support now. This notification feature enables you to receive notifications when certain events (such as PUT, GET, COPY, DELTE etc.) happen in your bucket. Currently people use this feature majorly for alerting etc. lightweight use cases, with Caerus storage-side UDF pushdown, now we can support use cases even they are very data intensive in an fully automatic fashion.

Related to “bucket notification”, there are two characteristics:

1. It is currently only supported in object storage systems/cloud object storage, we haven’t seen it in file system (like HDFS) or block systems, but this doesn’t say that we can’t add such support to file/block storage, in matter of fact, people do ask such feature and have some experimental implementation (for example, people hook up HDFS with Apache Nifi and event target like Redis/Kafka to support HDFS version of this feature). We can integrate this feature into HDFS as needed.
2. When people use bucket notification feature, they normally take advantage the rich feature of “bucket” configurations (normally it is called bucket policy), see examples of Amazon S3 bucket policy) to create specialized bucket. For example, a bucket normally only contains certain type of object types (images file only), or a bucket only belongs to certain group/user/organization etc.

Amazon S3 bucket policy examples:

1. <https://aws.amazon.com/premiumsupport/knowledge-center/s3-allow-certain-file-types/>
2. <https://docs.aws.amazon.com/AmazonS3/latest/userguide/example-bucket-policies.html>

Here we only give few concrete examples on how customers can potentially take advantage of Caerus NDP-UDF support in terms of Programmable Storage category, **especially in combine our unique automated event-driven storage-side serverless UDF support**, to show how powerful this feature can bring to the storage systems, but many more examples can be provided if needed:

Use case #1 – Potential Use of Caerus Automated Event-Driven of UDFs in Health Care Industry:

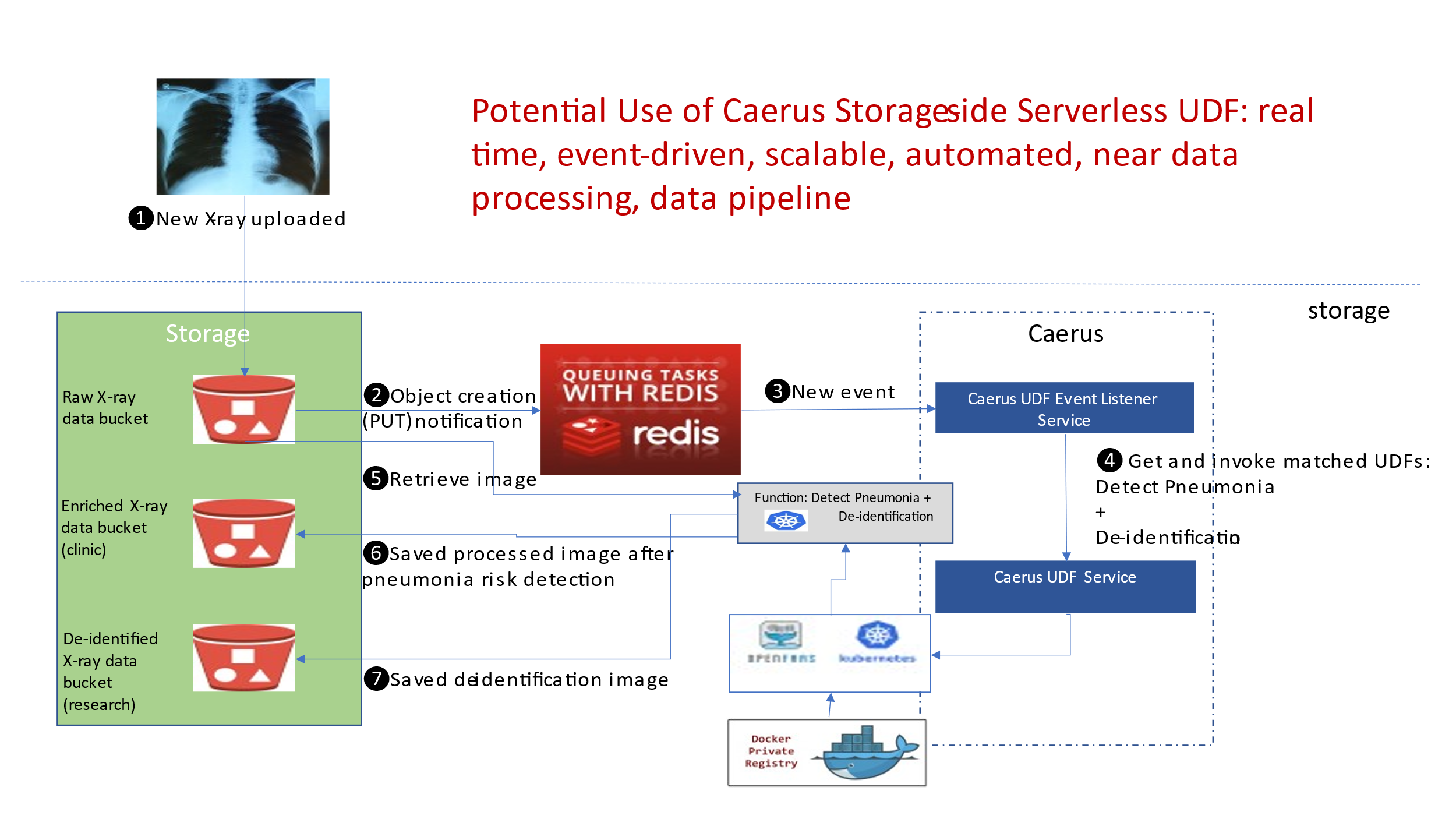


Figure 1. Caerus DNP-UDF Use Case Example

When a patient’s X-ray is uploaded to storage raw X-ray data bucket, the bucket notification on this bucket will fire an event to the target such as Redis or Kafka etc. The Caerus UDF Event Listener Service will get the event, then to look up and invoke the matched UDFs via Caerus UDF Service, which will call it serverless platform (OpenFass with Kubernetes), a serverless function container will be deployed automatically under the orchestration of Kubernetes. This function can then retrieve the newly uploaded image, and execute following functionalities:

1. To detect pneumonia risk from the raw X-ray image via image recognition or Deep Learning models, and then enrich the image buy adding metadata or tags. This processed image can then be saved to another data bucket that has all the enriched X-ray data, so that physicians can use it for clinical usage.
2. To de-identification the raw X-ray data, so that when data is used in research, patients’ privacy can be properly protected and be complying with HIPAA. These de-identification images data can be stored into another bucket that are used by researchers.

In the traditional business process, a hospital assistant or data scientist would need to download image from the storage infrastructure to a staging compute system. Then, the assistant would apply Image Detection module to evaluate the risk followed by a De-identication module to comply with HIPAA. Finally, the de-identied is sent to the researchers, and risk detection results to physicians. The entire process is manual, error prone, not scale, and inefficient and unnecessarily duplicate data (need transfer large image files back-and-forth between storage and compute).

With Caerus NDP-UDF support, the entire process becomes a data pipeline that is secure, efficient, and fully automated.

Use case #2 – Potential Use of Caerus Automated Event-Driven of UDFs in IoTs:

Suppose a smart grid utility company has many IoT devices like home energy monitoring devices, it is building an analytics application and storing raw data in object storage system or cloud. The application allows each user to display consumption history, real time comparison with neighbors and analytics about consumption behavior. When a new IoT data is uploaded into a dedicated bucket that has bucket notification enabled, the event target can call back to the Caerus UDF Listener Service. In this case, the event data provides the object key, bucket name, event name (such as PUT), and other relevant details. You can write a UDF function to generate custom metrics by aggregating raw data to supply processed data to the real time application.

### Goals

The goal of Caerus NDP-UDF support is to provide an end-to-end solution, including software architecture, design, and interfaces to support both Programmable Storage and UDF Pushdown By Compute Platform.

### Competitive Analysis

**Amazon S3 Object Lambda**:

On March 18, 2021 (after we completed design and major implementation of our Caerus NDP-UDF), AWS published a news blog to release a new feature called Amazon S3 Object Lambda:

* [**Amazon S3 Object Lambda**](https://aws.amazon.com/blogs/aws/introducing-amazon-s3-object-lambda-use-your-code-to-process-data-as-it-is-being-retrieved-from-s3/)

Amazon hasn’t included too much details on how the Amazon S3 Object Lambda internals work and nor it is open source software, some preliminary conclusions can only be made based on the information provided. From the information we gathered, this feature has some similarities with Caerus NDP-UDF, while it has a lot of fundamental differences with Caerus NDP-UDF. For the completeness of this design document, it was decided to include the comparison section here:

* **Introduction of Amazon S3 Object Lambda**

“With S3 Object Lambda you can add your own code to **Amazon S3 GET requests** to modify and process data as it is returned to an application. You can use custom code to modify the data returned by standard S3 GET requests to **filter rows, dynamically resize images, redact confidential data, and more**. Powered by AWS Lambda functions, your code runs on infrastructure that is fully managed by AWS, **eliminating the need to create and store derivative copies of your data or to run proxies**, all with no changes required to applications.”. “In addition to this list, S3 Object Lambda access points do not support [POST Object](https://docs.aws.amazon.com/AmazonS3/latest/API/RESTObjectPOST.html), [Copy](https://docs.aws.amazon.com/AmazonS3/latest/API/API_CopyObject.html) (as the source), or [Select Object Content](https://docs.aws.amazon.com/AmazonS3/latest/API/API_SelectObjectContent.html).”

* **Similarities between Amazon S3 Object Lambda and Caerus NDP-UDF**

1. ***Programmable Storage***: Both support Programmable Storage paradigm that customer can write custom code or user defined function to transform storage objects.
2. ***UDF pushdown***: Both support some level of UDF pushdown to storage (although it is not clear if Amazon Lambda if fully operated within the storage network in their cloud backend to have the full near data processing effect, while Caerus guarantees that function is running within storage network).
3. ***Serverless***: Both support serverless framework (Amazon Lambda vs Caerus OpenFaas) for ease-of-use to end users so that they don’t need to worry infrastructure deployment and runtime management of UDF running environment.

* **Differences between Amazon S3 Object Lambda and Caerus NDP-UDF**
  + ***Supported Storage Operations***: Amazon currently only supports GET (and LIST) request, while Caerus support full set of storage operations like GET, LIST, PUT, POST, COPY, and DELETE, as well as S3 Select. So Caerus can support more customers’ use cases.
  + ***Fully-automated Event-Driven UDF Invocation***: Amazon currently doesn’t support this feature, while Caerus does, and Caerus can also support the events from any storage operations.
  + ***Supported Storage System Types***: Amazon’s feature currently only targets to S3 object storage, while it is not clear how many code and architecture can be reused when applying to other storage system (both cloud and enterprise) types like file and block etc. The Caerus NDP-UDF is architected and implemented with portability as a major design concern, it can be easily ported to any cloud and enterprise storage system types, such as S3, Ceph, and HDFS etc.
  + ***(Caerus Future) Storage Location Affinity and GPU Affinity Support***: Amazon hasn’t mentioned any of this support. But Caerus has a plan to support storage object location and special hardware (e.g. GPU) affinity support, so that Caerus can run UDFs on the storage nodes that are the closest to the requested objects and have the compute resources like GPU that the UDFs need.
  + ***Productization***: Amazon have many features that Caerus hasn’t implemented yet, like full-blown security (IAM), ease-of-use (rich Lambda templates, and function libraries), management and debugging tools (centralized cloud GUI portal etc.). A lot of engineering effort will be needed for Caerus to reach to the production level.

Here are the other products or researches that tried to solve similar problems:

1. **Amazon S3 Select**: it can only support simple SQL pushdowns like predicate, and projection. UDF pushdown is not supported. Programmable Storage is not supported.
2. **MinIO S3 Select**: similar to AWS S3 Select, UDF pushdown and Programmable Storage are not supported
3. **OpenStack Storlets**: OpenStack specific UDF framework implementation, not portable to other storage systems, standalone container UDF runner only without container orchestration, the resource allocation for UDF runner are complicated and manual, no serverless support. It only supports simple SQL pushdown, while SQL UDF pushdown is not supported.
4. **Ceph SkyhookDM**: Ceph specific UDF framework implementation, not portable to other storage systems, standalone VM-based UDF runner only, no container or serverless support. It only supports one database implementation based on ProsgreSQL, no other compute platforms are supported.
5. **Amazon Athena UDF**: private code base, only support Amazon Athena (based on Presto), not portable to other storage systems. It has compute-side serverless architecture, UDFs cannot be pushdown.

We haven’t seen following features in any of the products or researches related to NDP-UDF support, more details on how we are going to support these features are described in the design sections of this document:

1. **Portable NDP-UDF architecture** that can be added to any storage systems
2. **Storage-side serverless framework** for UDF pushdown of full set of storage operations including S3 Select
3. **Automatic event-driven storage-side UDF invocation**
4. **Compute platform UDF pushdown** to storage side, will work on Spark first, then move on to other storage platforms
5. **Hardware/software acceleration** of storage side UDFs (future support)

### Benefits

1. Highly portable architecture that can be easily added to any storage system or cloud storage backend without the need to change storage systems
2. First to support storage-side serverless architecture option that is easy to deploy UDFs, lower cost, better scalability, and improved latency
3. First to support fully automated event driven UDF invocations in area of Programming Storage
4. Work with any workflows, compute platforms, and programming languages in BDA and AI
5. Have the potential for further UDF acceleration (future TODO) by taking advantage storage-side hardware (CPU, GPU, FPGA, Smart SSD etc.) and software (caching and indexing)
6. Has the same customer benefits as general Near Data Processing:
   * Significantly reduce network traffic between compute and storage layers
     + In an apple-to-apple comparison, if the available compute resources are the same between compute and storage layers, the NDP (including NDP-UDF) approach will be faster because of the reduction of unnecessary network traffic
     + If the storage system doesn’t have available resources for NDP computation, it actually can slow down the application by forcibly pushdown UDFs. Caerus will address this issue by implementing a wholistic orchestration of NDP based on the runtime telemetry data collection, statistic and ML predictions to decide where is the best place for computations to run. See more details in related design document.
   * Reduce storage I/O in most of the cases
   * Speed up overall processing time
   * Mitigate the “too big to eat” problem
   * Take full advantage of storage system resources
   * Reduce cost
   * Improve in data privacy and regulation

### Major Features

1. Support options to run UDF as serverless (using Openfaas framework, Q1) or standalone containers (Q3)
2. Support both fully automated event-driven and direct invocation of UDFs
3. Support UDF invocation upon any storage operations like Get/Access, Put, Copy, and Delete
4. Support Spark UDF SQL integration (Q2)
5. Support any storage systems (Integration of Minio for Q1 as an example, Ceph and HDFS in Q2 and beyond)
6. Ability to support any programming language implementations of UDFs

## Organization of This Document

As mentioned, in Caerus project scope, UDF (Pushdown) has broader meaning, it covers two integrated aspects:

* **Programmable Storage** (or refers to **Rich-Active Storage**): Section 3-4
* **UDF Pushdown By Compute Platform** (orrefers to **SQL UDF Pushdown**): Section 5

## References

List any documents, if any, which were used as sources of information for the test plan.

## Definitions and Acronyms

Table 1 - Acronyms

| Acronym | Literal Translation |
| --- | --- |
| NDP | Near Data Processing |
| UDF | User Defined Function |
| BDA | Big Data Analytics |

# Design Considerations

To follow Agile software development model, this design document is a running document that more details will be added to some sections that are planned to be implemented in later quarterly releases.

## Assumptions and Dependencies

Although Caerus NDP-UDF can theoretically support any storage systems and any compute platforms, doesn’t have hard dependency on those systems and platforms, the integration of Caerus NDP-UDF with these systems and platforms still needs some effort to show end-to-end benefits.

We develop Caerus NDP-UDF in different stages to support different storage system types and compute platforms:

1. Stage 1: Adding support for Programmable Storage
   1. Build up different infrastructure components needed for supporting both Programmable Storage and Compute UDF Pushdown, components include serverless framework, direct invocation framework, event registration/notification framework, NDP and UDF frontend and backend services and APIs
   2. Support at least one storage system types (choosing Object Storage, and pick MinIO as the first integration example).
2. Stage 2: Adding support for Compute UDF Pushdown, and extend Programmable Storage
   1. Support simple and well-formed compute UDFs, such as row-base input parameter only, on at least one compute platform (choosing Spark), adding UDF pushdown mechanism in both Spark and storage side, try to reuse the infrastructure pieces built up during Stage 1.
   2. Extend Programmable Storage to more storage system types such as HDFS. If needed Ceph support can be added.
3. Stage 3: Extend Compute UDF Pushdown
   1. Support pushdown of complex UDFs, such as UDAF, object-level input parameters etc. with at least one compute platform (choosing Spark), adding pushdown mechanism in both Spark and storage side
   2. Extend to other compute platforms if needed.

## Goals and Guidelines

The Caerus NDP-UDF design should have following goals:

1. No hard dependency on storage system internals: this will make sure that Caerus NDP-UDF support is highly portable and can be easily integrated into different types of storage system.
2. Trying to use existing storage protocol as much as we can: this will make sure that no significant changes are needed for existing customers who want to use Caerus support
3. Any changes in compute platform should be either in standard build or use plugin mechanism, no private build of compute platform is needed

## Architectural Strategies

Because of the design goals highlighted above, trade-offs are made in following areas:

1. One can argue that because Caerus NDP-UDF is highly portable, it might not perform as well as deep integration with storage internals. We made the decision to take portability as higher priority, while we don’t prevent people to adapt based on our product to do deep integration with their storage system internals.
2. Same argument can be raised on compute platform side, the plugin approach might not perform and easy to use as native support. But open source nature of these compute platforms requires very long period of time to add new support, we will try to raise requirements and get involved in development in open source community, meanwhile we don’t have to wait the more elegant native solution before we can release Caerus.

# System Architecture

The overall architecture of the Caerus NDP-UDF is described in Figure 2.



Figure 2. Caerus NDP-UDF Architecture

The software components of Caerus NDP-UDF support are listed as follows:

1. **Caerus NDP Service**: a storage-side HTTP service that can accept and process common storage requests by complying standard protocols (e.g. AWS S3 storage protocol and HDFS WebHDFS protocol etc.), the major difference of this service comparing with other similar service is that we have the ability to process UDF request as part of storage requests for direct invocation of UDFs.
2. **Caerus UDF Service**: a storage-side REST service that allows validate and invoke UDFs with the option of using serverless or standalone containers.
3. **Caerus UDF Registry Service**: (for standalone container option only) a storage-side REST service that provides REST APIs to manage UDF Registry which is implemented based on Redis and underlining storage.
4. **Caerus Event Listener Service**: a storage-side REST service that listens to registered streaming sources (Redis for now, can add other sources like Kafka, RMQ etc. if needed). Upon event, it reacts and automatically invokes related UDFs upon certain storage actions.
5. **Caerus Registry**:
   1. **Redis Cluster**: it is a storage-side service (dockers cluster) that plays two roles. First, it acts as a streaming target for storage events, this is the common part for both serverless and standalone options. Second, in standalone mode, it acts as a repository for UDFs (this can be migrated to Docker Hub is needed in the future).
   2. **Docker Hub**: In serverless mode, we will use Openfaas scheme which uses Docker Hub (public and private) as UDFs repository

**Note**: For UDF repository implementation choice, we chose Redis as our first implementation target, because it provides comprehensive caching mechanism that can speed up UDFs execution. Redis also provides nice event queueing mechanism that we can use it for streaming target of storage events. But this doesn’t mean this is the only option, other event target like Kafka can be implemented as well. In next release, we will further consolidate the UDF repository implementation into one single docker registry (docker hub is a cloud-based docker registry), we will continue to use Redis, not as UDF repository, but as cache for docker registry, as well as event target.

1. **Caerus Faas (Function-As-A-Service)**:
   1. **Caerus Faas Client**: A modified version of Openfaas client library (from a public github source) that is part of the Caerus UDF Service, allow it to send request to Openfaas framework in serverless mode. Our major contributions are adding authentication support, updating code and depend libraries (e.g. from okhttp to okhttp3 etc.).
   2. **Openfaas Server-side Framework**: A set of commands, configurations and instructions to set up Openfaas platform for Caerus UDF support.
2. **Caerus S3 CLI (with UDF support)**: A CLI built based on AWS S3 SDK that can support standard storage operations by using standard AWS S3 protocols, PUT, GET, DELETE, COPY and LIST with UDF support. The major difference of this CLI comparing with other similar product is that we have the ability to process UDF request as part of storage requests for direct invocation of UDFs.
3. **Caerus HDFS CLI (with UDF support)**: Similar to Caerus S3 CLI, it is a CLI built based on WebHDFS that can support standard storage operations by using standard WebHDFS protocols, PUT, GET, DELETE, COPY and LIST with UDF support. The major difference of this CLI comparing with other similar product is that we have the ability to process UDF request as part of storage requests for direct invocation of UDFs.
4. **Caerus UDF Functions**:
   1. A complete **serverless UDF** example that compiles, publishes and deploys UDF as an Openfaas serverless function that combines user defined function and common boilerplate code. It will read/write to storage directly via storage client.
      1. A serverless UDF that is targeting to Object Storage (MinIO)
      2. A serverless UDF that is targeting to HDFS
   2. A complete **standalone UDF** example that compiles, publishes and deploys UDF docker that combines user defined function and common boilerplate code. It will read/write to storage directly via storage client

## Storage Protocols and Events to Support UDF Pushdown

Since UDF pushdown is a new concept, most of storage protocols natively don’t support such pushdown. In order to support UDF pushdown to storage systems, standard storage protocols might need small changes or extensions. Here are the examples in several different storage protocols:

### Object Storage (S3 Protocol)

Amazon Object Storage S3 protocol is the most common object storage protocol supported by most of the enterprise storage systems and cloud storage, although it starts to support NDP features like [S3 Select](https://docs.aws.amazon.com/AmazonS3/latest/API/API_SelectObjectContent.html) to pushdown SQL query into storage, but currently it doesn’t support UDF pushdown natively. Fortunately, it supports custom metadata tag that can be embedded in the S3 protocol, we will just use this custom metadata for our UDF metadata information, so we don’t need any storage protocol changes.

Amazon S3 SDK is used in our project to compose standard S3 requests:



Most of the S3 requests are following the same flow, here we just use “[PutObject](https://docs.aws.amazon.com/AmazonS3/latest/API/API_PutObject.html)” as an example to describe UDF pushdown extension we need to add to the original S3 protocol (the green text part is the new extension to existing S3 protocol):

PUT /Key+ HTTP/1.1

Host: localhost

x-amz-acl: ACL

Cache-Control: CacheControl

Content-Disposition: ContentDisposition

Content-Encoding: ContentEncoding

Content-Language: ContentLanguage

Content-Length: ContentLength

Content-MD5: ContentMD5

Content-Type: ContentType

Expires: Expires

x-amz-grant-full-control: GrantFullControl

x-amz-grant-read: GrantRead

x-amz-grant-read-acp: GrantReadACP

x-amz-grant-write-acp: GrantWriteACP

x-amz-server-side-encryption: ServerSideEncryption

x-amz-storage-class: StorageClass

x-amz-website-redirect-location: WebsiteRedirectLocation

x-amz-server-side-encryption-customer-algorithm: SSECustomerAlgorithm

x-amz-server-side-encryption-customer-key: SSECustomerKey

x-amz-server-side-encryption-customer-key-MD5: SSECustomerKeyMD5

x-amz-server-side-encryption-aws-kms-key-id: SSEKMSKeyId

x-amz-server-side-encryption-context: SSEKMSEncryptionContext

x-amz-server-side-encryption-bucket-key-enabled: BucketKeyEnabled

x-amz-request-payer: RequestPayer

x-amz-tagging: Tagging

x-amz-object-lock-mode: ObjectLockMode

x-amz-object-lock-retain-until-date: ObjectLockRetainUntilDate

x-amz-object-lock-legal-hold: ObjectLockLegalHoldStatus

x-amz-expected-bucket-owner: ExpectedBucketOwner

***x-amz-meta-function\_name: caerus-faas-spring-thumbnai***

***x-amz-meta-function\_inputParameters: “400, 600”***

Body

Once Caerus backend gets these custom tags (“x-amz-meta-” are the standard custom tag prefix supported by S3 protocol) from the request, it will do followings:

1. The backend will store the Caerus UDF metadata information from the header, strip the these custom tags out of the request
2. Submit the request to this S3 storage node for storage operation (PUT in this case)
3. If the storage operation is successful, the UDF will be invoked on the storage objects that were stored by above 2) request. In failure case of storage operation, an error will be returned without the step of invocation of UDF

### HDFS (WebHDFS Protocol)

WebHDFS currently doesn’t support UDF pushdown, a protocol change has to be made. This change should be communicated to the HDFS community for UDF pushdown standardization in the future, if approved, then certain WebHDFS REST interfaces and SDK can be modified for ease of use.

Most of the WebHDFS requests are following the same flow, here we just use “[Create and Write to a File](https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/WebHDFS.html#Create_and_Write_to_a_File)” as an example to describe normal process without UDF pushdown:

Step 1: Submit a HTTP PUT request without automatically following redirects and without sending the file data, the <HOST><PORT> refers to HDFS namenode:

curl -i -X PUT "http://<HOST>:<PORT>/webhdfs/v1/<PATH>?op=CREATE

[&overwrite=<true |false>][&blocksize=<LONG>][&replication=<SHORT>]

[&permission=<OCTAL>][&buffersize=<INT>][&noredirect=<true|false>]"

Namenode will return information of a datanode where the file data is to be written:

HTTP/1.1 307 TEMPORARY\_REDIRECT

Location: http://<DATANODE>:<PORT>/webhdfs/v1/<PATH>?op=CREATE...

Content-Length: 0

Step 2: Submit another HTTP PUT request using the URL in the Location header with the file data to be written.

curl -i -X PUT -T <LOCAL\_FILE> "http://<DATANODE>:<PORT>/webhdfs/v1/<PATH>?op=CREATE..."

**In UDF pushdown case**, the Step 1 keeps the same, we will make changes in Step 2:

**Change #1**: we will replace <DATANODE>:<PORT> with <DATANODE>:<**NDP\_PORT**>, where NDP\_PORT is the Caerus NDP Service port. In deployment, each datanode of the HDFS has one set of Caerus backend components (NdpService, UDF Service, Event Listener Service etc.) running, these components will accept/intercept storage request, invoke UDFs if needed.

**Change #2**: we will embed UDF metadata information as an xml in the HTTP request header (the green text part is the new addition to existing WebHDFS protocol):

…

Url: http://localhost:8000/webhdfs/v1/input/sample0.jpg<br/>

Uri: /webhdfs/v1/input/sample0.jpg<br/>

Scheme: http<br/>

Server Name: localhost<br/>

Port: 8000<br/>

Context Path: <br/>

Servlet Path: /webhdfs/v1/input/sample0.jpg<br/>

Path Info: null<br/>

Query: op=CREATE&namenoderpcaddress=namenode:9000&createflag=&createparent=true&overwrite=true

Header: Basic cm9vdDpwYXNzd29yZA==

**Header: <?xml version="1.0" encoding="UTF-8" standalone="yes"?> <caerusudf id ="1"> <function\_name>caerus-faas-spring-thumbnail-hdfs</function\_name><function\_inputParameters>400, 600</function\_inputParameters></caerusudf>**

Header: http://datanode:9864/webhdfs/v1/input/sample0.jpg?op=CREATE&namenoderpcaddress=namenode:9000&createflag=&createparent=true&overwrite=true

Header: Java/1.8.0\_292

Header: localhost:8000

….

**Change #3**: This is not related protocol change, but it is related to how we execute the next step after Caerus backend gets this changed protocol (new header):

1. The backend will get the Caerus UDF metadata information from the header, strip the new header out of the request
2. Submit the request to this datanode’s WebHDFS service for storage operation (PUT in this case)
3. If the storage operation is successful, the UDF will be invoked on the storage object/file that were stored by above 2) request. In failure case of storage operation, an error will be returned without the step of invocation of UDF

## Execution Sequences for Major Workflows

We will describe detail sequences for Programmable Storage use case, while a lot of steps are the same in the use case of UDF Pushdown By Compute Platforms, and we have some ongoing investigation work related to different compute platforms, such as figure out what are the best way for Spark to pushdown UDFs within the SQL context. Therefore, we will only give the highlight for the later use case, and more details will be added in later months.

### Programmable Storage – Event-Driven Storage UDF Workflow (Serverless)

1. **Caerus NDP-UDF Initialization**

All Caerus UDF related services, including OpenFaas serverless framework and event publishing target like Redis clusters are initialized and running. The Caerus UDF Event Listener Service, in particular, will register an onMessage() function callback to the targets, Redis etc., so any event happens to the bucket/folder, a callback will invoke to this service.

1. **UDF Registration**

User builds and deploys UDF container to the Docker Hub, UDF can be written in:

* 1. any programming languages (Java, Scala, Python etc.)
  2. any language version (Java 8, 9, 11 etc.)
  3. any build systems (Maven, Gradle etc.)

In the UDF metadata section of each UDF there will be information related to invocation conditions, for example, it will be only invoked when a copy operation happens.

For UDF registration security, we will rely on docker registry (including docker hub) security, unauthorized user will not be able to access the repositories with in docker registry/docker hub account. For UDF runtime security, we will rely on storage system object/bucket security.

1. **Event Registration**

* **Object Storage**

User registers notification events via APIs provided by storage vendors. Many modern storage systems now support event notification system, for example, AWS S3, Ceph and MinIO all support storage bucket notification, this feature allows user to register storage events, such as put, get, delete, copy etc. on specific storage bucket or folder, anytime a registered event happens, such as an object has been put into the registered bucket, the event can be published to targets like Redis, Kafka, Webhooks, AMQP, ElasticSearch, ProstgreSQL and NATs etc. Storage system will normally provide APIs, e.g. MinIO CLI commands, for user to register event. For storage systems that don’t have this notification yet, it is not that difficult to add such support since target systems like Redis, Kafka etc. normally provide very comprehensive APIs support that be easily integration into storage system.

Here are the examples of such event notification:

* Amazon S3: <https://docs.aws.amazon.com/AmazonS3/latest/userguide/NotificationHowTo.html>
* MinIO S3: <https://docs.min.io/docs/minio-bucket-notification-guide.html>

The event message structure is normally a json format with the following fields, they include information related to bucket name, object id, event name (what storage operation is) etc.

Text

Description automatically generated

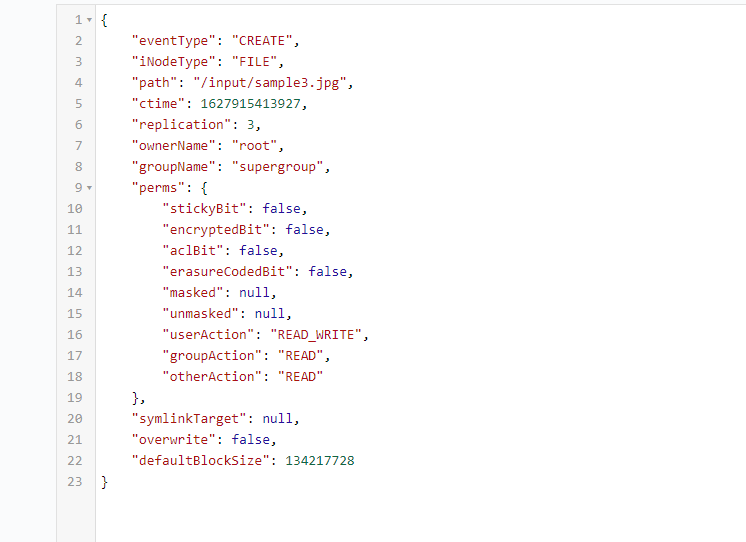
* **HDFS**

HDFS natively doesn’t support event registration to different target systems like Redis and Kafka etc. however, it does support an event API SDK called [iNotify](https://hadoop.apache.org/docs/current/api/org/apache/hadoop/hdfs/inotify/Event.html). The integration with these popular target systems can be implemented via this iNotify interface.

To save development time, it is decided to look for an existing integration solution with these target systems, the most popular one people are using is through [Apache Nifi](https://nifi.apache.org/), which is scalable directed graphs of data routing, transformation, and system mediation logic. There is Nifi processor called [GetHDFSEvents](https://nifi.apache.org/docs/nifi-docs/components/nifi-docs/components/org.apache.nifi/nifi-hadoop-nar/1.9.0/org.apache.nifi.processors.hadoop.inotify.GetHDFSEvents/index.html) that can most of the storage events on HDFS including append, close, create, metadata, rename and unlink. Redis is also chosen as the target system via another [Nifi PutDistributedMapCache processor and FetchDistributedMapCache processor](https://bryanbende.com/development/2017/10/09/apache-nifi-redis-integration).

Once Nifi processors run, it functions as a live event monitor, any registered HDFS storage operations like create/append a file etc. will cause any event firing to the target system (Redis), then Caerus backend that was registered to listen these events will get automatic notification (callback) from the Redis, then it can invoke matched serverless UDFs on the HDFS storage side.

Here is an example of event message structure (also uses json format as S3 event) from HDFS iNotify:



1. **Storage Operation**

User uses any storage client (CLI/GUI/compute platform/custom/curl) to do a storage operation like put, get, delete, copy etc. As long as such operation(s) is registered, notification(s) will be sent from storage system to the event target, e.g. Redis. The Caerus UDF Event Listener Service will get a onMessage() function callback, it will check any UDF in the Docker Hub has the matched invocation condition, for example, if a put notification occurs, the Caerus will check if any UDF invocation condition has put action.

1. **Event Notification and Caerus Event Listening Service**

When a registered storage event happens, event target like Redis will send notification to Caerus Event Listening Service via the callback function (OnMessage function). The Caerus Event Listening Service will first get all the registered UDFs that have event-driven invocation conditions, each of such UDFs has a metadata section that defines what storage operation this UDF will act on, for example, A UDF can have PUT and COPY as invocation condition, so anytime a PUT or a copy happens, this UDF will be invoked.

The Caerus Event Listening Service will then find matched UDFs by compare event object and UDF metadata invocation conditions. If matched UDF invocation condition is found, The Caerus Event Listening Service will explicitly call the Caerus UDF Service to invoke matched UDFs.

1. **UDF Invocation**

The Caerus UDF Service will based on the deployment environmental variables to decide either invoke serverless or standalone UDF function. The storage APIs are also used to obtain objects from storage systems before it can invoke UDFs to operate on these storage objects.

1. **Exception Handling**

Any exception and critical error during the entire process will be logged into logs of related Caerus services.

An animated sequence diagram is described in Fig. 3.



Figure 3.. Automatic Event-Driven UDF Invocation

### Programmable Storage – Direct Invocation of UDF Workflow (Serverless)

1. **Caerus NDP-UDF Initialization**

Same as 3.1.1 step 1.

1. **UDF Registration**

Almost same as 3.1.1 step 2. In the UDF metadata section of each UDF there might or might not have information related to invocation conditions.

1. **Storage Operation**

* **Object Storage**

User uses Caerus S3 CLI (with UDF support) to issue storage operation command like put, copy, delete, list and get. The UDF info including UDF unique identifiers, function input parameters etc. can be supplied via CLI switches.

.

* **HDFS**

User uses Caerus HDFS CLI (with UDF support) to issue storage operation command like put, copy, delete, list and get. The UDF info including UDF unique identifiers, function input parameters etc. can be supplied via CLI switches

1. **UDF Invocation**

If a UDF information is supplied, the UDF will be invoked via OpenFaas APIs by the Caerus UDF Service after the storage operation is successfully finished.

1. **Exception Handling**

Any exception and critical error during the entire process will be logged into logs of related Caerus services. UDF will not be invoked if the storage operation failed. Proper error message will be returned to user via CLI if an error occurs during the process.

An animated sequence diagram is described in Fig. 4.



Figure 4. Direct Invocation of UDF

### Programmable Storage – Event-Driven Storage UDF Workflow (Standalone container)

1. **Caerus NDP-UDF Initialization**

Same as 3.1.1 step 1.

1. **UDF Registration**

Similar to 3.1.1 step 2, instead of using Docker Hub in serverless mode, we use Redis as UDF Registry (repository) inside storage network. In future, certain consolidation can be made to use the same UDF Registry (either Docker Hub or Redis) for both serverless mode and standalone mode.

A set of REST APIs are provided from the Caerus Registry Service to allow users to manage (including list, upload, download, delete, and register etc.) UDFs with metadata information, such as input parameters, invocation conditions etc., in the Caerus UDF Registry.

1. **Event Registration**

Same as 3.1.1 step 3.

1. **Storage Operation**

Same as 3.1.1 step 4.

1. **Event Notification and Caerus Event Listening Service**

Same as 3.1.1 step 5.

1. **UDF Invocation**

If matched UDF invocation condition is found, those UDFs will be invoked by the Caerus UDF Service, internally it will call Redis APIs to obtain, load, and invoke the UDF.

1. **Exception Handling**

Same as 3.1.1 step 7.

### Programmable Storage – Direct Invocation of UDF Workflow (Standalone container)

1. **Caerus NDP-UDF Initialization**

Same as 3.1.2 step 1.

1. **UDF Registration**

Same as 3.1.3 step 2.

1. **Storage Operation**

Same as 3.1.2 step 3.

1. **UDF Invocation**

If a UDF information is supplied, the UDF will be invoked via Redis APIs by the Caerus UDF Service after the storage operation is successfully finished.

1. **Exception Handling**

Same as 3.1.2 step 5.

## Deployment Topology: Physical/Virtual Resources

1. **Caerus NDP-UDF Backend Services (One instance per storage node)**

Currently all major Caerus NDP-UDF related storage-side backend services, including **Caerus NDP Service, Caerus UDF Service, Caerus UDF Registry Service, and Caerus Event Listener Service**, are microservice-based REST services, in future if needed, containerization can be accomplished easily. They, along with Caerus Query Service, will be deployed into a Caerus NDP HTTP Server component which can be either a VM or a container. We envision in distributed storage systems, this Server component will be deployed into each storage server/controller node or outside storage node, but it has 1x1 mapping to each storage node. The storage request to each storage node will be intercepted by this Server component, if it is a query or UDF related request, Caerus components will process the request and send query/UDF results back to the client. Otherwise, it behaves like an in-place storage client, does normal storage operations by calling storage APIs to the associated storage server/controller.

1. **Common Caerus NDP-UDF Services (One instance per storage system)**

These services, including **Caerus Registry (Redis Cluster)** and **OpenFaas Framework** (K8S etc.) are only one running instance per storage system. Caerus NDP-UDF Backend Services on each storage node behave as multiple client services for these common services.

1. **Docker-compose version of “one-click” deployment**

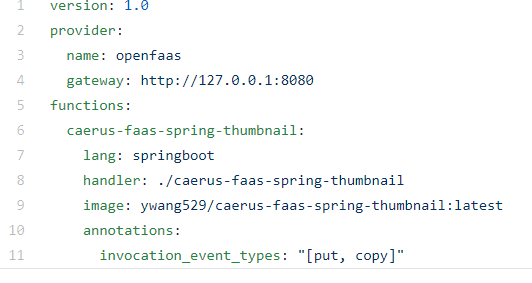
Both S3 and HDFS deployment are also supporting one-click docker-compose deployment for ease of demo etc. See github repo [“deployment” README](https://github.com/open-infrastructure-labs/caerus-udf/tree/master/deployment) for guidance.

## Network Architecture

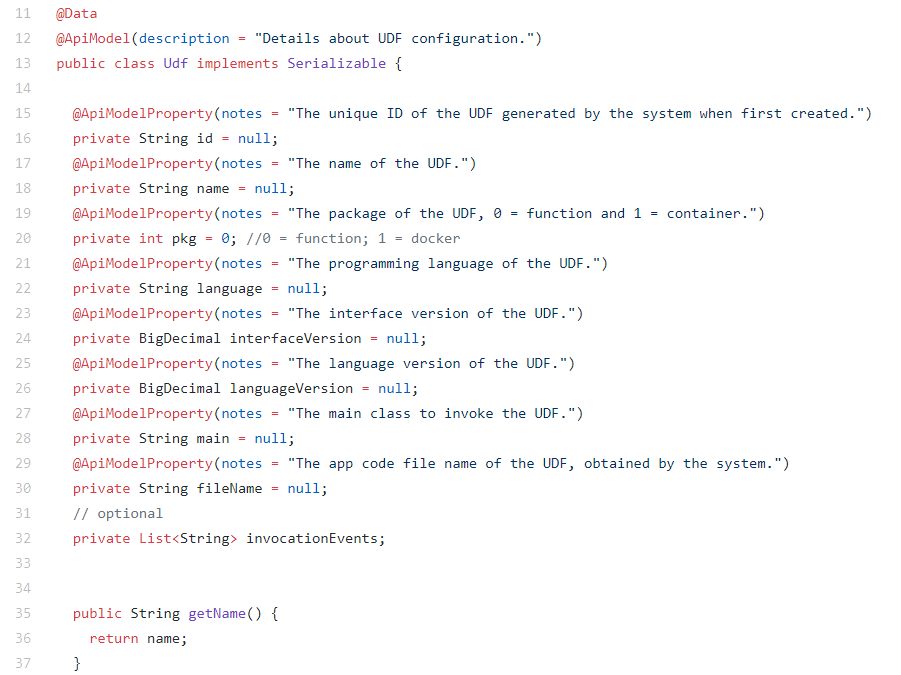
The network port numbers for any microservice are configurable via application configuration files under each service source code. A detail list of the port numbers can be found in the Caerus github project source code location.

## Data Architecture

* In the serverless mode, all the UDF images and metadata are stored in Docker Hub with proper user credentials. The metadata information such as UDF invocation condition are stored in the “Annotations” field of each UDF in Docker Hub. An example is listed as [follows](https://github.com/futurewei-cloud/caerus/blob/99f55a37c4215ce060b1ebb4756e33df19758a27/ndp/udf/examples/thumbnail_serverless/caerus-faas-spring-thumbnail.yml):



* In the standalone mode, the UDF images and metadata are stored in Redis with data schema defined in [this class](https://github.com/futurewei-cloud/caerus/blob/master/ndp/udf/registry/src/main/java/org/openinfralabs/caerus/UdfRegistry/model/Udf.java).



# Design Decomposition

Following software services are responsible for Caerus NDP-UDF support, the organization of this section most contains:

1. Inputs: service APIs details
2. Outputs: interaction details with external and other internal services
3. Internals: implementation details

Note: for detail code API documents, we use Javadoc and Swagger, both are generated from code directly, to generate these documents, just simply follow the procedure in Caerus github udf top level by calling the build script, it will generate documents for all the Caerus UDF related services.

## Caerus NDP Service

Caerus NDP Service is a storage-side HTTP REST service and the entry point of all storage related requests. Note this service can be integrated into storage system native HTTP service or other similar service if needed.

### Inputs

The service can take standard AWS S3 protocol and WebHDFS as input HTTP request, it can support common standard storage operations such as PUT (upload), GET (download), DELETE, LIST and COPY, more importantly, it can process UDF request as part of storage requests using the metadata portion of the storage protocol (see UDF request details in Caerus S3 and HDFS CLIs) for direct invocation of UDFs. In the future, we will extend this support for HDFS requests.

### Outputs

* The service can accept (PUT) object stream from the client (e.g. Caerus S3 or HDFS CLIs) and return (GET) object stream to the client. Extra features like multi-part file support can be added in the future.
* It can communicate with underlining storage systems via respective storage system APIs to serve the storage requests.
* After the storage request is served successfully, this service will communicate with Caerus UDF Service via its service REST API to invoke UDF.
* The error message will be sent back to the client and logged in the service log if any error occurs during runtime.

### Internals

* Communication mechanism to underlining storage systems:

The service contains a storage client interface class (StorageAdaptor) that can be instantiated into different storage client classes like MinIO, Ceph, HDFS, S3 etc., so that Caerus NDP-UDF support can be easily ported into different storage systems. This service will be responsible for executing storage requests via these client implementations. Inside the client implementation, storage SDK (e.g. MinIO Java SDK, HDFS Java Client, Ceph SDK etc.) is used to issue storage requests to the storage system. The UDF metadata information is also tagged with the storage object using standard storage protocols, for example, in MinIO case, we use metadata tags like followings, these metadata information will be stored along with storage object in the storage system:



* Software package and class organization:

The common MVC (model-view-controller) Design Pattern is used in this service implementation. It contains following software packages under “org.openinfralabs.caerus.ndpService” namespace:

* + config: create storage system client entry point by taking parameters (IP, port, credentials etc.) from separate resource files
    - Note: user credentials are passed in from client as part of the storage operation, such as from CaerusS3 CLI, these can be used to create storage system entry point object. More fine-turned security can be implemented in the future if needed.
  + controller: HTTP request handler
  + interceptor: for debug purpose only to intercept raw standard incoming HTTP requests (e.g. S3, HDFS etc.)
  + model: UDF definition classes
  + service: StorageAdaptor interface class and its implementation to different storage systems, e.g. MinIO etc. (see Fig. 5.)

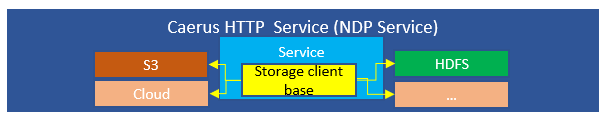
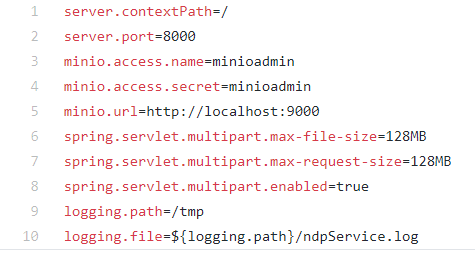


Figure 5. Caerus NDP Service

Typical resource file will look like this:



The generated class diagram by Eclipse ObjectAid plugin (note: can be replaced if we can find better generator) is listed in Fig. 6. More detail class APIs and definition can be referenced by the API document (JavaDoc and Swagger API document in the source code).

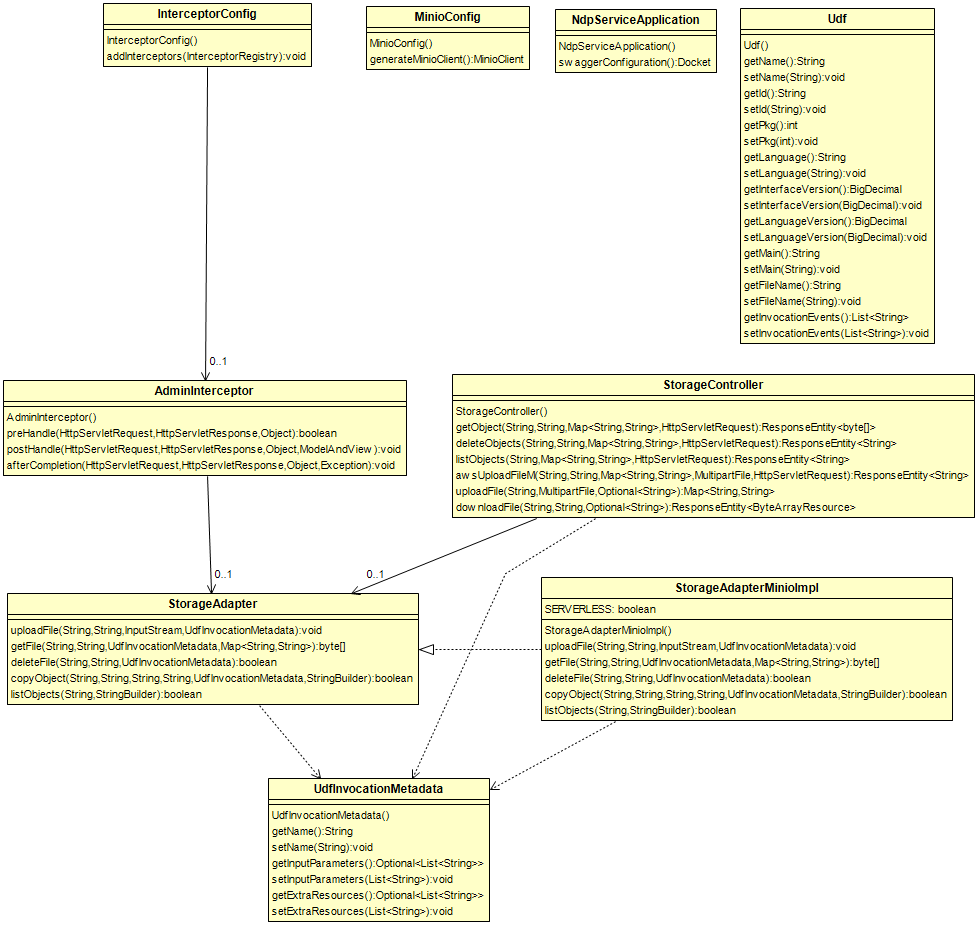


Figure 6. Caerus NDP Service Class Diagram

## Caerus UDF Service

Caerus UDF Service is a simple storage-side HTTP REST service that supports validation and invocation of serverless and Standalone UDFs, its main responsibility is to accomplish proper separation the caller (NDP Service) from UDF management details (register, validate, invoke, serverless vs standalone vs. future webassembly etc.).

### Inputs

The service can take standard input HTTP request, it takes object info like bucket/folder key and object key/name, as well as UDF metadata information in name-value pairs.

### Outputs

* In Serverless mode, it will communicate with OpenFaas via Caerus OpenFaas Java Client SDK for validation, and invocation of UDFs.
* In Standalone mode, it will communicate with Caerus UDF Registry Service via its REST APIs for validation, and invocation of UDFs

### Internals

It uses simplified MVC design pattern, under the namespace of “org.openinfralabs.caerus.udfService”, there are two modules: the controller module is a REST service request handler that can accept HTTP (GET) request with storage object information and UDF metadata as input; while the model module contains UDF definitions. The generated class diagram by Eclipse ObjectAid plugin is listed in Fig. 7. More detail class APIs and definition can be referenced by the API document (JavaDoc and Swagger API document in the source code).



Figure 7. Caerus UDF Service Class Diagram

## Caerus Event Listener Service

Caerus Event Listener Service is a storage-side HTTP REST service that listens to registered streaming sources (Redis for now, can add other sources like Kafka, RMQ etc. if needed). Upon event, it reacts and automatically invokes related UDFs upon certain storage actions.

### Inputs

Currently it is a “passive” service that doesn’t need to take any input, an OnMessage() function is implemented for listening event from event target such as Redis. However, for completeness. It does have a controller that can accept REST requests and then send request to the event target, this can be use for test purpose or there is special need to modify certain event in the target.

### Outputs

* In Serverless mode, it will communicate with OpenFaas via Caerus OpenFaas Java Client SDK for validation, and invocation of UDFs.
* In Standalone mode, it will communicate with Caerus UDF Registry Service via its REST APIs for validation, and invocation of UDFs

### Internals

* Communication mechanism to event target (e.g. Redis):

We are using the exact same object definitions as the most storage systems when they publish the events, for example, in MinIO and AWS implementation, they all use AWS S3 SDK “com.amazonaws.services.s3.event.S3EventNotification” package, we obtain the event object including storage object information (bucket/folder and object key) as well as storage operation type (PUT, GET. DELETE, COPY, ACCESS etc.). The service will look up all the registered UDFs to find the matched storage operation type. Once the match is found, UDF is then invoked.

* Software package and class organization:

It contains following software packages under “org.openinfralabs.caerus.eventListenerService” namespace:

* + config: create target such as Redis, Kafka entry point
  + controller: HTTP request handler. Currently it is for test purpose only, but it can be expanded. It will call sender classes below
  + model: UDF definition classes
  + receiver: OnMessage function implementation (see above communication mechanism)
  + sender: classes to handle different targets like Redis, Kafka. Currently it is for test purpose only, but it can be expanded.

The generated class diagram by Eclipse ObjectAid plugin is listed in Fig. 8. More detail class APIs and definition can be referenced by the API document (JavaDoc and Swagger API document in the source code).

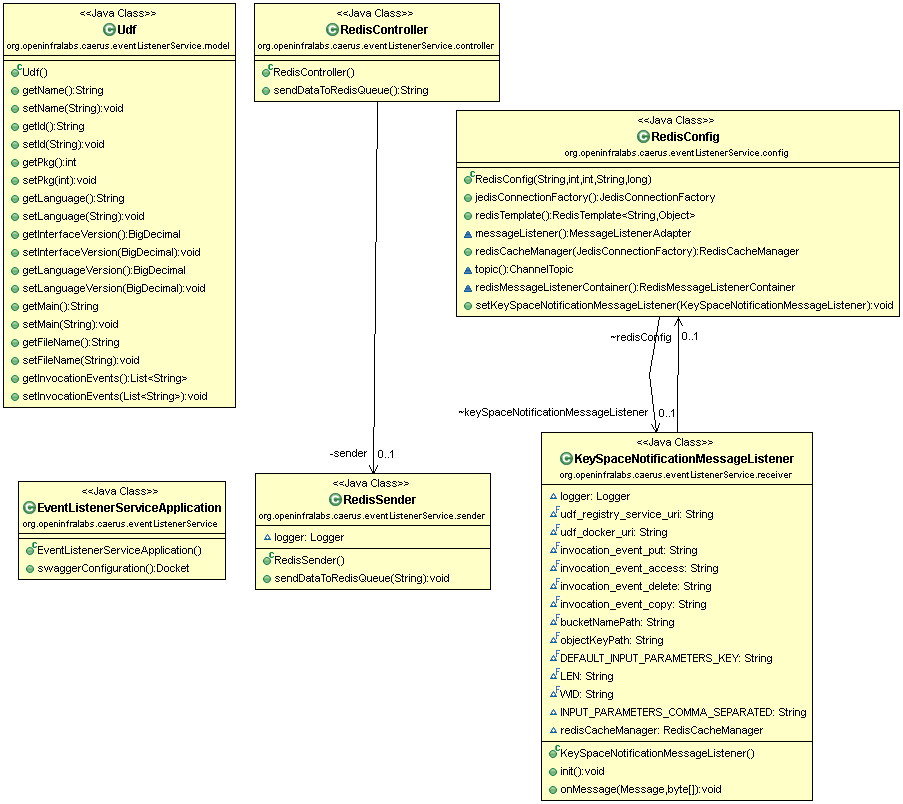


Figure 8. Caerus Event Listener Service

## Caerus UDF Registry Service

Caerus UDF Registry Service is a storage-side REST service that serves requests to store/create, retrieve, modify and delete UDF configurations and its app code (e.g. jar file). It uses Redis (mount on any storage) as backend. Currently it is used in the Standalone option only, where in serverless mode, the registration of UDFs is handled via OpenFaas SDK and framework.

### Inputs

Currently it serves REST APIs for common actions such as POST/GET/Delete/PUT etc.

### Outputs

It will communicate with Caerus Registry, currently implemented using Redis via Redis SDK, to manage UDFs.

### Internals

It uses MVC and DAO (data-access-object) design patterns, it contains following software packages under “org.openinfralabs.caerus.UdfRegistry” namespace:

* + config: create target Redis entry point
  + controller: HTTP request handler. It serves requests to store/create, retrieve, modify and delete UDF
  + model: UDF definition classes
  + repository: UDF DAO definitions
  + service: interface and implementation of different registry service including fetchAllUdfs, getUdfById, deleteUdfById, updateUdfInfo, updateUdfExecutable etc.

The generated class diagram by Eclipse ObjectAid plugin is listed in Fig. 9. More detail class APIs and definition can be referenced by the API document (JavaDoc and Swagger API document in the source code).

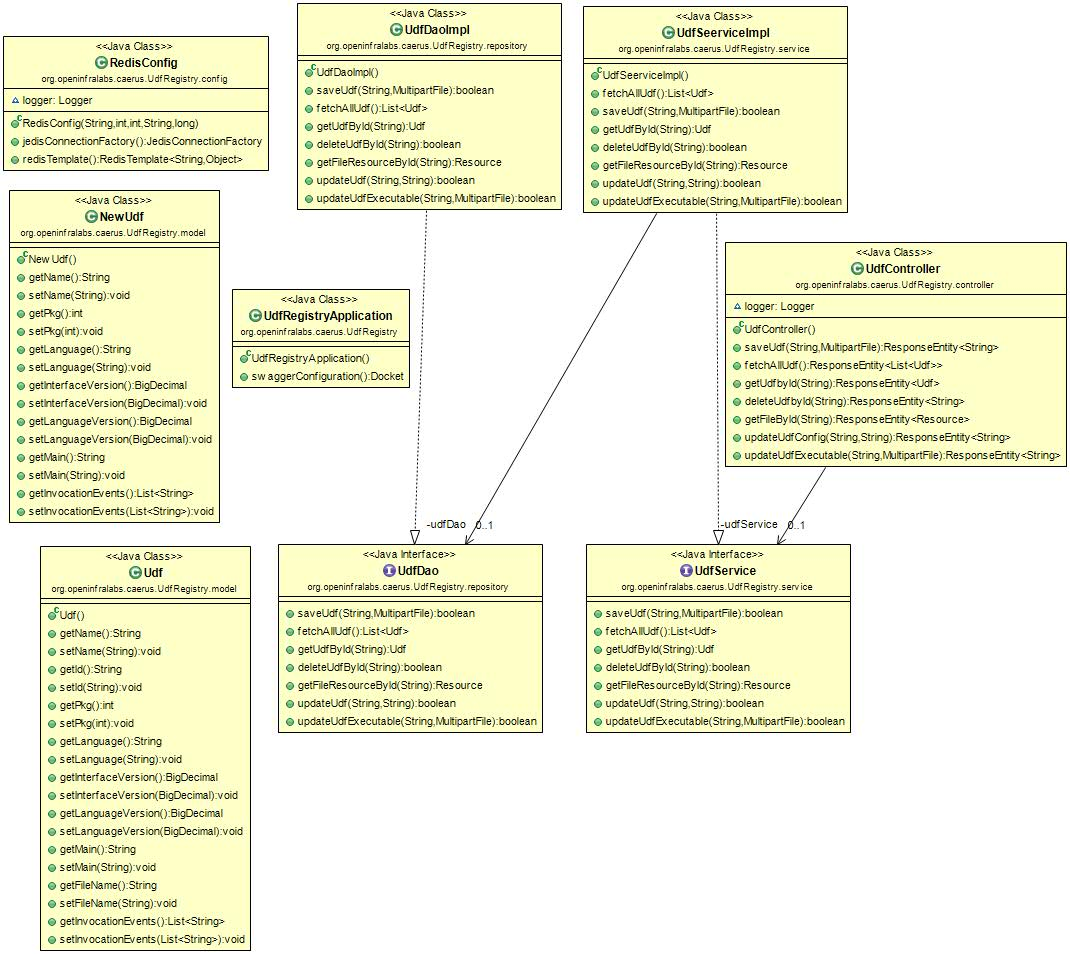


Figure 9. Caerus UDF Registry Service

## Caerus UDF Registry

It only contains set of scripts, commands and dockers files, written procedures etc. for setting up following registry framework support:

1. **Redis Cluster**: it is a storage-side service (dockers cluster) that plays two roles. First, it acts as a streaming source for storage events, this is the common part for both serverless and standalone options. Second, in standalone mode, it acts as a repository for UDFs (this can be migrated to Docker Hub is needed in the future).
2. **Docker Hub**: In serverless mode, we will use Openfaas scheme which uses Docker Hub (public and private) as UDFs repository

## Caerus Faas (Function-As-A-Service)

It contains following software components:

1. **Caerus Faas Client**: A modified version of Openfaas client library (from a public github source) that is part of the Caerus UDF Service, allow it to send request to Openfaas framework in serverless mode. Our major contributions are adding authentication support, updating code and depend libraries (e.g. from okhttp to okhttp3 etc.).
2. **Openfaas Server-side Framework**: A set of commands, configurations and instructions to set up Openfaas platform for Caerus UDF support.

The generated class diagram by Eclipse ObjectAid plugin is listed in Fig. 10. More detail class APIs and definition of Caerus Faas Client can be referenced by the API document (JavaDoc and Swagger API document in the source code).

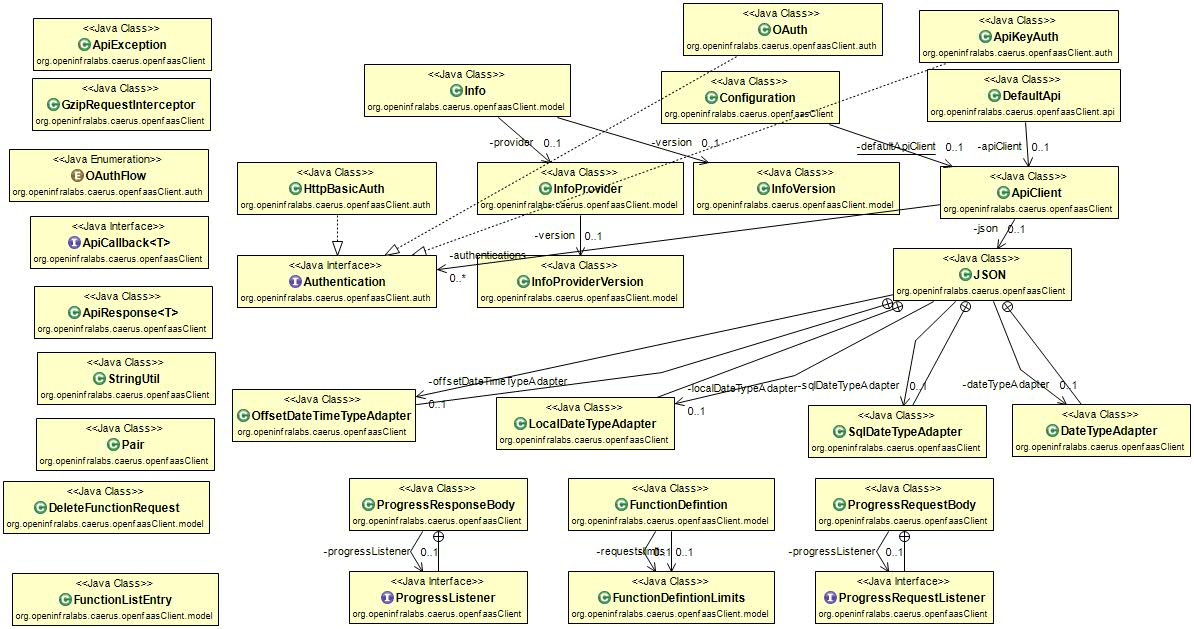


Figure 10. Caerus OpenFaas Client

## Caerus S3 CLI with UDF Support

Caerus S3 CLI is a CLI is built based on AWS S3 SDK that can support standard storage operations by using standard AWS S3 protocols, PUT, GET, DELETE, COPY and LIST with UDF support. The major difference of this CLI comparing with other similar product is that we have the ability to process UDF request as part of storage requests for direct invocation of UDFs.

The generated class diagram by Eclipse ObjectAid plugin is listed in Fig. 11. More detail class APIs and definition of Caerus Faas Client can be referenced by the API document (JavaDoc and Swagger API document in the source code).

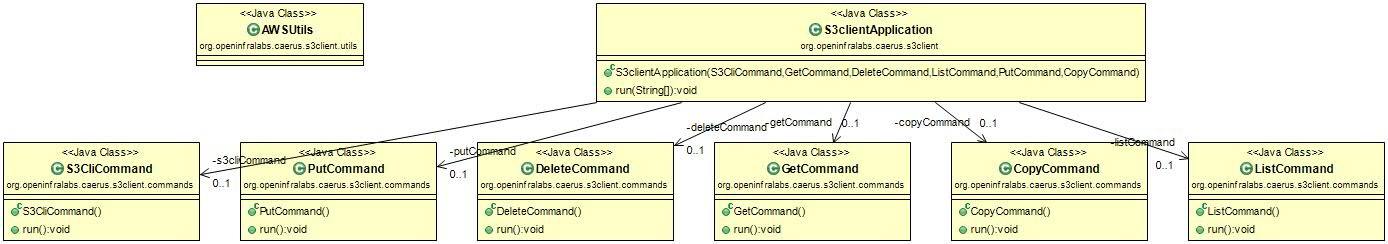


Figure 11. Caerus S3 CLI

## Caerus HDFS CLI with UDF Support

Caerus HDFS CLI is a CLI is built based on WebHDFS REST APIs that can support standard storage operations by using standard WebHDFS storage protocols, PUT, GET, DELETE, COPY and LIST with UDF support. The major difference of this CLI comparing with other similar product is that we have the ability to process UDF request as part of storage requests for direct invocation of UDFs.

## Caerus Sample UDF Functions

It only contains a UDF example that can create custmiziable thumbnails (different size, watermarks etc.) from large storage objects (image files). The implementation is using Java and Maven build. It has two separate sample functions:

1. Object Storage UDF example: a complete **serverless UDF** example that compiles, publishes and deploys UDF as an Openfaas serverless function that combines user defined function and common boilerplate code. It will read/write to storage directly via storage client. The code path is under “[caerus](https://github.com/futurewei-cloud/caerus)/[ndp](https://github.com/futurewei-cloud/caerus/tree/master/ndp)/[udf](https://github.com/futurewei-cloud/caerus/tree/master/ndp/udf)/[examples](https://github.com/futurewei-cloud/caerus/tree/master/ndp/udf/examples)/**thumbnail\_serverless**/”
2. HDFS UDF example: a complete **serverless UDF** example that compiles, publishes and deploys UDF as an Openfaas serverless function that combines user defined function and common boilerplate code. It will read/write to storage directly via storage client. The code path is under “[caerus](https://github.com/futurewei-cloud/caerus)/[ndp](https://github.com/futurewei-cloud/caerus/tree/master/ndp)/[udf](https://github.com/futurewei-cloud/caerus/tree/master/ndp/udf)/[examples](https://github.com/futurewei-cloud/caerus/tree/master/ndp/udf/examples)/**thumbnail\_serverless\_hdfs**/”
3. A complete **standalone UDF** example that compiles, publishes and deploys UDF docker that combines user defined function and common boilerplate code. It will read/write to storage directly via storage client. The code path is under “[caerus](https://github.com/futurewei-cloud/caerus)/[ndp](https://github.com/futurewei-cloud/caerus/tree/master/ndp)/[udf](https://github.com/futurewei-cloud/caerus/tree/master/ndp/udf)/[examples](https://github.com/futurewei-cloud/caerus/tree/master/ndp/udf/examples)/[java](https://github.com/futurewei-cloud/caerus/tree/master/ndp/udf/examples/java)/**thumbnail**/”

Note: The goal is to find as much as common ground (including code and procedure reuse) between these two options, currently we only support serverless option, the standalone details are yet to hashed out (standalone delivered date is targeting to Q3), this section will be updated once we have more details on standalone option.

# UDF Pushdown by Compute Platforms

## Spark

### Introduction

Spark SQL UDF supports different programming languages:

* Scala
* Java
* Panda UDF (Python)
* Python native

Among these programming languages, [Scala has the best performance](https://towardsdatascience.com/performance-in-apache-spark-benchmark-9-different-techniques-955d3cc93266). But they are still not as good as Spark native functions, which are natively implemented using Spark Catalyst expressions, such expression can then be highly optimized into SQL execution plans to contain features like predict pushdown, code generation (codegen), and serialization etc.

Unlike Spark native functions, Spark UDF doesn’t support pushdown natively with the exception of certain JDBC/database data source implementations. Spark UDF is forced to run in a black box (opaque) on compute-side, and Spark Catalyst by design can't optimize (codegen, predict pushdown etc.) UDF. Thus Spark UDFs are often performance bottlenecks.

Here is a simple example of native Spark UDF:

A custom function is registered, in the following way. Consider a very simple function that adds 2 its argument.

spark.udf.register("intUDF", (i: Int) => {

val j = 2

i + j

})

Spark plan containing a Spark UDF:

val udfResult = spark.sql("SELECT \* FROM people\_with\_schema WHERE intUDF(age) > 15")

scala> udfResult.explain(true)

== Parsed Logical Plan ==

'Project [\*]

+- 'Filter ('intUDF('age) > 15)

+- 'UnresolvedRelation [people\_with\_schema], [], false

== Analyzed Logical Plan ==

name: string, age: int

Project [name#0, age#1]

+- Filter (if (isnull(age#1)) null else intUDF(knownnotnull(age#1)) > 15)

+- SubqueryAlias people\_with\_schema

+- Relation[name#0,age#1] json

== Optimized Logical Plan ==

Filter (if (isnull(age#1)) null else intUDF(knownnotnull(age#1)) > 15)

+- Relation[name#0,age#1] json

== Physical Plan ==

\*(1) Filter (if (isnull(age#1)) null else intUDF(knownnotnull(age#1)) > 15)

+- FileScan json [name#0,age#1] Batched: false, DataFilters: [(if (isnull(age#1)) null else intUDF(knownnotnull(age#1)) > 15)], Format: JSON, Location: InMemoryFileIndex[file:/data/source/people.json], PartitionFilters: [], PushedFilters: [], ReadSchema: struct<name:string,age:int>

To solve the performance problems of the Spark UDFs, especially in the context of UDF pushdown for NDP, we think there are two fundamental approaches:

* 1. ***Partial UDF Pushdown – Translatable UDF***: this is especially the case when UDFs are used in the SQL, since Spark has a strong SQL optimization engine (Catalyst) that can optimize all SQL related actions. This option can take full advantage of Spark Catalyst expressions, the UDF can somehow be translated into native expressions, so they can be automatically optimized by the Catalyst. This case will cover most of the use cases for Spark SQL UDFs, since most of them are just simple lambda functions with math, string, time and relational operations, which can be translated into expressions. In this case, the UDF itself, as a whole, is not pushed down, instead, the “**translatable**” portion of the UDF is pushed down to storage.
  2. ***Whole UDF Pushdown***: For cases like data intensive operations like ETL and ML/DL etc. They cannot, and cannot easily, be translated into expressions, a generic mechanism should be provided to allow UDF to be pushed down to storage, so that it can take advantage of NDP. But such UDFs should be used carefully, especially in SQL UDFs, to make sure it doesn’t not interfere the Spark SQL optimization.

### Partial UDF Pushdown – Translatable UDF

#### Options

Even when most of UDFs used in Spark SQL contain the contents, such as math operations, string manipulation, date and time, and relational operations etc. that can be translated into Spark Catalyst expression, but because UDFs are run in an opaque box, such contents inside the UDF cannot be detected. Thus this forces users to choose one of the two following options:

* Implement UDFs as standard Spark UDFs to enjoy the flexibility and convenience of UDF, but the UDFs’ performance will suffer
* Implement UDFS as Spark expressions directly to enjoy high performance, but the user must understand Spark internals well, and implementation logic will be much more complicated

One of the examples of such options in Spark UDF implementation can be found in the TPCH Spark test programs:

* Standard Spark UDFs: <https://github.com/ssavvides/tpch-spark/blob/master/src/main/scala/Q01.scala>
* Spark Expression implementation of UDFs: <https://github.com/databricks/spark-sql-perf/blob/master/src/main/resources/tpch/queries/1.sql>

It is our goal to find a solution that has following features:

* Allow user to implement standard Spark UDFs
* No code change is required for existing Spark UDFs
* The translatable portion of the UDFs can be pushed down to storage to have performance advantages

Based on the investigation, there are three open-source solutions that we can get inspiration from:

##### Nvidia Spark Rapids UDF Compiler

Nvidia spark-rapids udf-compiler (<https://github.com/NVIDIA/spark-rapids>) uses JVM reflection (Scala UDF Compiler) to analyze bytecode, it then attempts to translate UDF logic to Catalyst expressions:

* Common math operations
* Type casts
* Conditional (if, case)
* Common string operations
* Data and time
* …

The translation is automatic, and requires no code changes for existing Spark application code, the translation can be explained in the following graph:

Diagram

Description automatically generated with medium confidence

The feature set of the "udf-compiler" solution is still under developed, the feature gap examples of the "udf-compiler" solution are listed as follows:

* It doesn't support tuple, map and collections
* It has less DateTime support than the Macros solution: monthsBetween, getDayInYear, getDayOfWeek etc.
* It doesn't support complex UDFs like recursive UDFs

The full supported features can be found in the following document: [udf-compiler](https://github.com/NVIDIA/spark-rapids/blob/branch-21.10/docs/additional-functionality/udf-to-catalyst-expressions.md)

One of the issues of the "udf-compiler" is that it has dependency on GPU setting, it requires user to install many cuda related drivers to the system, and it might have runtime issues when system doesn't have the GPU hardware. This will limit the usage of "udf-compiler", especially for our UDF data source/storage pushdown (Near Data Processing) use cases.

In order to address this issue, we developed the ***Caerus Spark UDF Compiler***, which is based on spark-raids, but certain modifications are made to remove GPU dependency. Users can follow instructions below to deploy and use udf-compiler in NDP use cases. See more detail code changes in this forked repo: <https://github.com/open-infrastructure-labs/caerus-spark-udf-compiler-from-rapids/blob/branch-21.10/README.md>

The result example of the ***Caerus Spark UDF Compiler*** is listed as follows:

root@ubuntu1804:/home/ubuntu/openinfralabs/caerus-spark-udf-compiler-from-rapids# spark-shell --driver-class-path udf-compiler/target/rapids-4-spark-udf\_2.12-21.10.0-SNAPSHOT.jar --conf "spark.sql.extensions"="com.nvidia.spark.udf.Plugin"

SLF4J: Class path contains multiple SLF4J bindings.

...

Spark context Web UI available at http://10.124.62.103:4040

Spark context available as 'sc' (master = local[\*], app id = local-1629810836057).

Spark session available as 'spark'.

Welcome to

\_\_\_\_ \_\_

/ \_\_/\_\_ \_\_\_ \_\_\_\_\_/ /\_\_

\_\ \/ \_ \/ \_ `/ \_\_/ '\_/

/\_\_\_/ .\_\_/\\_,\_/\_/ /\_/\\_\ version 3.1.1

/\_/

Using Scala version 2.12.10 (OpenJDK 64-Bit Server VM, Java 1.8.0\_292)

Type in expressions to have them evaluated.

Type :help for more information.

scala> import org.apache.spark.sql.types.{IntegerType, StringType, StructType}

import org.apache.spark.sql.types.{IntegerType, StringType, StructType}

scala> val schema = new StructType().add("name", StringType, true).add("age", IntegerType, true)

schema: org.apache.spark.sql.types.StructType = StructType(StructField(name,StringType,true), StructField(age,IntegerType,true))

scala> val df\_with\_schema = spark.read.schema(schema).json("file:///data/source/people.json")

df\_with\_schema: org.apache.spark.sql.DataFrame = [name: string, age: int]

scala> df\_with\_schema.createOrReplaceTempView("people\_with\_schema")

scala> spark.udf.register("intUDF", (i: Int) => {

| val j = 2

| i + j

| })

res1: org.apache.spark.sql.expressions.UserDefinedFunction = SparkUserDefinedFunction($Lambda$2424/1272194712@5fa7cb3,IntegerType,List(Some(class[value[0]: int])),Some(class[value[0]: int]),Some(intUDF),false,true)

scala> val udfResult = spark.sql("SELECT \* FROM people\_with\_schema WHERE intUDF(age) > 15")

udfResult: org.apache.spark.sql.DataFrame = [name: string, age: int]

scala> udfResult.explain(true)

21/08/24 09:15:42 INFO FileSourceStrategy: Pushed Filters: IsNotNull(age)

21/08/24 09:15:42 INFO FileSourceStrategy: Post-Scan Filters: isnotnull(age#1),((age#1 + 2) > 15)

21/08/24 09:15:42 INFO FileSourceStrategy: Output Data Schema: struct<name: string, age: int>

== Parsed Logical Plan ==

'Project [\*]

+- 'Filter ('intUDF('age) > 15)

+- 'UnresolvedRelation [people\_with\_schema], [], false

== Analyzed Logical Plan ==

name: string, age: int

Project [name#0, age#1]

+- Filter ((age#1 + 2) > 15)

+- SubqueryAlias people\_with\_schema

+- Relation[name#0,age#1] json

== Optimized Logical Plan ==

Filter (isnotnull(age#1) AND ((age#1 + 2) > 15))

+- Relation[name#0,age#1] json

== Physical Plan ==

\*(1) Filter (isnotnull(age#1) AND ((age#1 + 2) > 15))

+- FileScan json [name#0,age#1] Batched: false, DataFilters: [isnotnull(age#1), ((age#1 + 2) > 15)], Format: JSON, Location: InMemoryFileIndex[file:/data/source/people.json], PartitionFilters: [],

PushedFilters: [IsNotNull(age)],

ReadSchema: struct<name:string,age:int>

scala>

The result of the ***Caerus Spark UDF Compiler*** indicated that it does fit all the requirements for an ideal Partial UDF Pushdown solution, with no code change, and just some simple configuration setting changes in Spark, users can just achieve Spark UDF pushdown automatically.

More features can be implemented into this solution to fill the features gaps list above.

##### Spark SQL Macros

Spark-SQL-Macros (<https://github.com/hbutani/spark-sql-macros>) leverages the **Macro mechanics of the Scala Compiler** to analyze the Scala AST of the function body and try to generate **an equivalent Catalyst Expression for the function body**. With Spark SQL Macros you can register the function as a macro like this:

import org.apache.spark.sql.defineMacros.\_

spark.registerMacro("intUDM", spark.udm((i: Int) => {

val j = 2

i + j

}))

scala> val udmResult = spark.sql("SELECT \* FROM people\_with\_schema WHERE intUDM(age) > 15")

udmResult: org.apache.spark.sql.DataFrame = [name: string, age: int]

scala> udmResult.explain(true)

== Parsed Logical Plan ==

'Project [\*]

+- 'Filter ('intUDM('age) > 15)

+- 'UnresolvedRelation [people\_with\_schema], [], false

== Analyzed Logical Plan ==

name: string, age: int

Project [name#0, age#1]

+- Filter ((age#1 + 2) > 15)

+- SubqueryAlias people\_with\_schema

+- Relation[name#0,age#1] json

== Optimized Logical Plan ==

Filter (isnotnull(age#1) AND ((age#1 + 2) > 15))

+- Relation[name#0,age#1] json

== Physical Plan ==

\*(1) Filter (isnotnull(age#1) AND ((age#1 + 2) > 15))

+- FileScan json [name#0,age#1] Batched: false, DataFilters: [isnotnull(age#1), ((age#1 + 2) > 15)], Format: JSON, Location: InMemoryFileIndex[file:/data/source/people.json], PartitionFilters: [], PushedFilters: [IsNotNull(age)], ReadSchema: struct<name:string,age:int>

Spark SQL Macros API:

|  |
| --- |
|  |

**Spark SQL Macros Problem Statement**:

For customers who have the existing Spark applications, they must change their application code:

Vanilla Spark UDF:

spark.udf.register("intUDF", (i: Int) => {

val j = 2

i + j

})

val udfResult = spark.sql("SELECT \* FROM people\_with\_schema WHERE intUDF(age) > 15")

Spark SQL Macros:

spark.registerMacro("intUDM", spark.udm((i: Int) => {

val j = 2

i + j

}))

val udmResult = spark.sql("SELECT \* FROM people\_with\_schema WHERE intUDM(age) > 15")

Although Scala Macro mechanism can be a very powerful tool to solve the translation problem, the current implementation of Spark SQL Macros doesn’t have the full automation we are looking for, especially for users who have existing Spark UDFs, or portability of the UDFs are very important to you, this solution might not be a good choice. But this solution does have much more feature sets, for users who develop new Spark UDFs and you don’t care that much on UDF’s portability, it can still be a good solution.

See more details in these investigation reports:

* + <https://github.com/open-infrastructure-labs/caerus-udf/blob/master/docs/README_SPARK_UDF.md>
* <https://github.com/open-infrastructure-labs/caerus-udf/blob/master/docs/README_SPARK_UDF_DATA_SOURCE.md>
* <https://github.com/open-infrastructure-labs/caerus-udf/blob/master/docs/README_SPARK_UDF_SQL_MACROS_INVESTIGATION.md>
* <https://github.com/open-infrastructure-labs/caerus-udf/blob/master/docs/README_SPARK_UDF_SQL_MACROS_USER_MANUAL.md>

##### Informatica manual implementation Spark UDF as Spark native function

Manual implementation of UDF as Spark native function: <https://databricks.com/session_eu20/optimizing-apache-spark-udfs>, provides another way to improve Spark UDF performance, however, it will need to rebuild Spark core code, this forces user to use custom Spark core, which is not an ideal solution, but it did provide another way for translation.

#### Performance Measurement and Results

##### UDF performance benchmarks

Since all above solutions attempt to translate opaque UDFs into Spark expressions, their performance gains should be very similar, they are majorly from following areas:

* Operations (e.g. predicate) Pushdown
* Catalyst related optimizations
  + WholeStageCodegen: optimization in serialization/de-serialization
  + Null optimization
  + Other Catalyst optimizations

For translatable UDF, since all above solutions need a lot additional work (see above restrictions/functional gaps of each solution) to have full production-level support, it is difficult to identify test benchmark for performance measurement.For example, udf-compiler doesn't support string equal sign "=" comparison yet, so any existing customers' UDFs that contain such string comparison will either fall back to native Spark UDF calculation (note: currently there is a defect that any exception will fail the entire UDF in UDF-compiler), or need to be rewritten to use supported expression (String.equal() function is supported).

For Spark UDF performance measurement, it can be divided into two categories:

1. “standard” benchmarks – especially from tpc series

* [tpcx-bb](http://tpc.org/tpcx-bb/default5.asp): UDFs are clearly defined and separated, and normally take 1/3 of the total query (total 30 queries) time. However,
  + It has strong dependency on Cloudera products
  + It has many assumptions like Spark worker nodes coexist with storage HDFS data nodes (Spark and storage use the same yarn for control etc.), this might limit our NDP measurement
  + The UDFs in current form are not translatable by the ufd-compiler
  + It is still valuable in future test
* [Tpch](http://tpc.org/tpch/default5.asp): it has some UDFs, but they are optional, e.g. [Databricks’s queries replace all UDFs with expressions](https://github.com/databricks/spark-sql-perf), we need explore more in the future.

1. “custom” benchmarks – based on real custom use cases, for example,

* [SQL Macros tax and discount calculation UDF with retails](SQL%20Macros%20tax%20and%20discount%20calculation%20UDF%20with%20retails)
* [Facebook hive UDFs migration](NDhttps://www.youtube.com/watch?v=wnZlLRMsY)
* [Informatica Expression Language Functions](https://databricks.com/session_eu20/optimizing-apache-spark-udfs)

Since it seems too early to measure the translation solution against “standard benchmarks, considering solution like udf-compiler is still under development, it cannot support most of existing UDFs as they are in existing “standard” benchmarks, it has been decided to use ‘custom’ benchmark first to show the performance improvement of UDF translation. The SQL Macros tax and discount calculation in retails is used (adapted to support udf-compiler) in this experiment because of its availability.

##### Preliminary UDF performance results

**System setup**

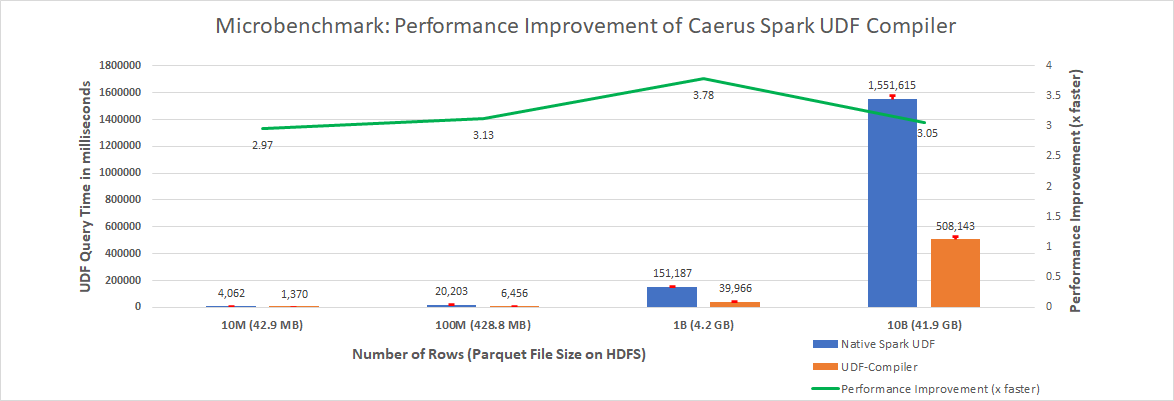
In our experiment, we set up:

* a compute cluster to include 3 Spark VMs nodes (one master and two workers) on one server
* a storage cluster to include 3 HDFS VMs nodes (one name node that has yarn, two data nodes) on another server
* the data generation tool (modified from the SQL Macros project)
* the data generation tool can generate any scale factors as needed, the data sizes (parquet file on HDFS) are as follows:
* 10 million rows: 42.9 MB
* 100 million rows: 428.8 MB
* 1 billion rows: 4.2 GB
* 10 billion rows: 41.9 GB

**Results**

1. Query time improvement

**Caerus UDF-compiler translation is average 3x faster than Spark native UDF**



1. Network/Storage I/O Reduction

For certain data sources that Spark natively support, like Parquet (columnar data with stats metadata), they can be optimized for Spark QL query performance improvements by only transporting needed data from storage to compute. The three major categories are:

* **Predicate/filter pushdown**: Spark can evaluate filtering predicates in the query against metadata stored in the columnar (Parquet) files, to skip reading chunks of data if the provided filter predicate value in the query is outside the range of values stored for a given column.
* **Partition pruning**: when data is partitioned by certain patterns (like based on zip code of IoT data etc.), Spark can read data from a list of partitions, based on a filter on the partition key, skipping the rest.
* **Column projection**: Spark can just read the data for columns that the query needs to process and skip the rest of the data.

Among these three, when Spark UDFs are involved in the query, **Partition pruning** and **Column projection** can always work, because the final query plan to include the UDF calculation will ask for certain partition or/adn columns, Spark can naturally take advantage these features.

However, this is not the case for predicate/filter pushdown of UDFs, when a native Spark UDF (blackbox or opaque) is used in the predicate (Where clause in SQL), Spark Catalyst can't natively translate the UDFs, thus in the final physical plan of a query will have empty PushedFilters (to tell datasource for pushdown) list. As a result, Spark compute side has to pull all the data, then filter on the compute side, this causes unnecessary I/Os (read data from storage/drives and transport data via network from storage to compute).

Via UDF compiler translation of UDFs, we hope the PushedFilters list will be formed naturally in the physical plan, so that Spark can use the stats in the Parquet file for predicate pushdown. This should speed up query in time spent and I/O reduction.

Other than UDF translation, predicate pushdown also has special requirements on the source data and NDP capability in storage (see below).

To summarize, in order for Spark to pushdown predicate/filter, it must meet following conditions

* Physical plan must contain PushedFilters
* Data source must support pushdown – for native data source, only columnar type data sources like parquet support this operation
* Data source must be ‘filterable’ --- min-max etc, see parquet-cli tool results , data transformation (ETL) might be needed before query
* Any calculations beyond basic stats in the data source cannot be pushed down to storage, unless a custom datasource is implemented on Spark side and a NDP is implemented on the storage side. e.g. Caerus NDP, and certain database engines.

For generic introduction of Spark UDF performance, please see [Caerus Spark UDF Performance Improvement](https://github.com/open-infrastructure-labs/caerus-udf/blob/master/examples/spark-udf/README_SPARK_UDF_PERFORMANCE.md) for more details

**Caerus UDF-compiler translation has up to 3.3x Network I/O Reduction than Spark native UDF**

Spark measurement (based on the ordered data with 10 billion rows, see detail SQL query, data transformation etc. in the source code):

* Before (Spark native UDF): 6475 ms (+/- 363) query time, 7978193 bytes read
* After (UDF Compilation): 1991 ms (+/- 127) query time, 2428777 bytes read
* **I/O Reduction: 7978193/2428777 = 3.3x**
* **Query time Speed up: 6475/1991 = 3.2x**

### Whole UDF Pushdown (TBD)

**Spark Client Side – Advanced Use Cases like data intensive ETL, DL, MLlib (ML), graph, streaming, complex queries etc.**

For UDFs that are either not part of query or it is part of complex query, such as, in many ML/DL cases, UDF predict(..) or train(..) functions can be implemented as UDFs, and they can be either part of query or standalone calculation in Spark, such UDFs will be very hard to be optimized using the macro approach by above simple query case, the UDF pushdown to storage systems will have great performance gain, especially those UDFs are data-intensive operations.

More concrete use cases and standard dataset need to be investigated, and performance need to be measured to compare UDF pushdown and native Spark UDF.

One inspiration could be from Nvidia spark-rapids RapidsUDF extension: <https://github.com/NVIDIA/spark-rapids/blob/branch-21.10/docs/additional-functionality/rapids-udfs.md>

This Nvidia solution is to provide an interface class, RapidsUDF, so users can implement their own UDFs to implement RapidsUDF, so that when such UDF is called, it will automatically produce UDF that can run on GPU.

Graphical user interface, text, application, email

Description automatically generated

We can borrow this idea to implement our own interface class CaerusUDF for pushdown, any UDFs that implements such CaerusUDF can be pushdown entirely to storage. More details will be added in the future based on ongoing investigation.

# Software Architecture Quality Attributes

## DevOps

Details are to be filled.

## Security and Privacy

This section is planned, but yet to be implemented yet, it can be implemented as requested.

The Caerus UDF ACL (Access Control List) support:

It is basically an extension of underlining storage system ACL that allow to give user access through a UDF (either in serverless or standalone mode). If a user will get access violation error if the UDF is trying to access the object (including bucket) that this user doesn’t have the correct role.

# Testing: Unit Tests, System Tests and Security Tests

Details are to be filled

# Future Enhancements

Details are to be filled:

1. WebAssembly serverless support
2. Hardware (GPU) acceleration on UDFs

# Open Issues

N/A