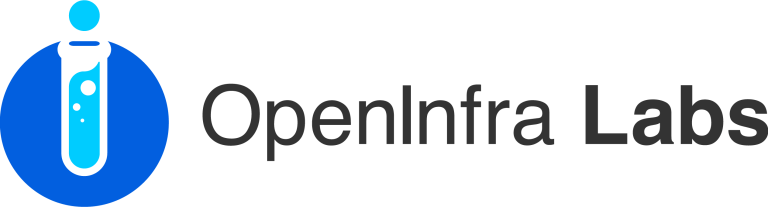
****



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

**System Design Document**

**Version 1.0**

**February 28, 2021**

*Yong Wang*

Table of Contents

[1 Introduction 5](#_Toc65406617)

[1.1 Purpose of Document 5](#_Toc65406618)

[1.2 Scope 5](#_Toc65406619)

[1.2.1 Overview 5](#_Toc65406620)

[1.2.2 Goals 7](#_Toc65406621)

[1.2.3 Competitive Analysis 7](#_Toc65406622)

[1.2.4 Benefits 7](#_Toc65406623)

[1.2.5 Major Features 8](#_Toc65406624)

[1.3 References 8](#_Toc65406625)

[1.4 Definitions and Acronyms 8](#_Toc65406626)

[2 Design Considerations 8](#_Toc65406627)

[2.1 Assumptions and Dependencies 8](#_Toc65406628)

[2.2 Goals and Guidelines 9](#_Toc65406629)

[2.3 Architectural Strategies 9](#_Toc65406630)

[3 System Architecture 9](#_Toc65406631)

[3.1 Execution Sequences for Major Workflows 11](#_Toc65406632)

[3.1.1 Programmable Storage – Event-Driven Storage UDF Workflow (Serverless) 11](#_Toc65406633)

[3.1.2 Programmable Storage – Direct Invocation of UDF Workflow (Serverless) 12](#_Toc65406634)

[3.1.3 Programmable Storage – Event-Driven Storage UDF Workflow (Standalone container, 2021 Q3 item) 13](#_Toc65406635)

[3.1.4 Programmable Storage – Direct Invocation of UDF Workflow (Standalone container, 2021 Q3 item) 14](#_Toc65406636)

[3.1.5 UDF Pushdown By Compute Platform – Spark SQL UDF pushdown (2021 Q2-Q3 item) 14](#_Toc65406637)

[3.2 Deployment Topology: Physical/Virtual Resources 14](#_Toc65406638)

[3.3 Network Architecture 15](#_Toc65406639)

[3.4 Data Architecture 15](#_Toc65406640)

[4 Design Decomposition 16](#_Toc65406641)

[4.1 Caerus NDP Service 16](#_Toc65406642)

[4.1.1 Inputs 16](#_Toc65406643)

[4.1.2 Outputs 16](#_Toc65406644)

[4.1.3 Internals 16](#_Toc65406645)

[4.2 Caerus UDF Service 19](#_Toc65406646)

[4.2.1 Inputs 19](#_Toc65406647)

[4.2.2 Outputs 19](#_Toc65406648)

[4.2.3 Internals 19](#_Toc65406649)

[4.3 Caerus Event Listener Service 20](#_Toc65406650)

[4.3.1 Inputs 20](#_Toc65406651)

[4.3.2 Outputs 20](#_Toc65406652)

[4.3.3 Internals 20](#_Toc65406653)

[4.4 Caerus UDF Registry Service 21](#_Toc65406654)

[4.4.1 Inputs 21](#_Toc65406655)

[4.4.2 Outputs 21](#_Toc65406656)

[4.4.3 Internals 21](#_Toc65406657)

[4.5 Caerus UDF Registry 23](#_Toc65406658)

[4.6 Caerus Faas (Function-As-A-Service) 23](#_Toc65406659)

[4.7 Caerus S3 CLI with UDF Support 24](#_Toc65406660)

[4.8 Caerus Sample UDF Functions 24](#_Toc65406661)

[5 Software Architecture Quality Attributes 24](#_Toc65406662)

[5.1 DevOps 24](#_Toc65406663)

[5.2 Security and Privacy 24](#_Toc65406664)

[6 Testing: Unit Tests, System Tests and Security Tests 25](#_Toc65406665)

[7 Future Enhancements 25](#_Toc65406666)

[8 Open Issues 25](#_Toc65406667)

List of Figures

[Fig. 1. Caerus NDP-UDF Architecture 10](#_Toc65406782)

[Fig. 2. Automatic Event-Driven UDF Invocation 12](#_Toc65406783)

[Fig. 3. Direct Invocation of UDF 13](#_Toc65406784)

[Fig. 4. Caerus NDP Service 17](#_Toc65406785)

[Fig. 5. Caerus NDP Service Class Diagram 18](#_Toc65406786)

[Fig. 6. Caerus UDF Service Class Diagram 19](#_Toc65406787)

[Fig. 7. Caerus Event Listener Service 21](#_Toc65406788)

[Fig. 8. Caerus UDF Registry Service 22](#_Toc65406789)

[Fig. 9. Caerus OpenFaas Client 23](#_Toc65406790)

List of Tables

[Table 1 - Acronyms 8](#_Toc65406773)

**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 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

# 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.), 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.

* + 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** in Caerus can be described as follows:

1. Although SQL UDF Pushdown is a must-have feature, there are more and more use cases can be explored if Programmable Storage UDF feature can be supported.
2. 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.

Here we only give one concrete example on how customers can potentially take advantage of Caerus NDP-UDF support in terms of Programmable Storage category to show how powerful this feature can bring to the storage systems, but many more examples can be provided if needed:

In biomedical field, there are a lot of large image files (medical imaging files including tissues, x-ray, MRI etc.), which are normally 5-10 GB per file in size, imagine we can support UDF pushdown to allow physicians to easily do ROI (Region of Interest) Extractions, Cell Detection for possible cancer/abnormal cells, and Cohort Identification.

In Cohort Identification case, A researcher would like to get a cohort of images with similar features from as multiple patients as possible. In the traditional business process, a hospital assistant would need to download all images from the storage infrastructure to a staging compute system. Then, the assistant would apply Image Similarity module to create a cohort, followed by a De-identication module to comply with HIPAA. Finally, the de-identied cohort is sent to the researcher. With Caerus NDP-UDF, the Image Similarity UDF will run in the cluster of machines within the storage, this UDF can even take advantage GPU resources provided in storage layer. Then, another Caerus NDP-UDF for Biomedical Processes De-identication will de-identify the data within the storage and send to the researcher. This approach is much easier, faster (eliminates large file transfer, and potentially take advantage of compute resources like GPU in storage for in-place data processing), and is more secure than the traditional method as it spares the need to move clear data to the staging system.

### 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

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

1. **AWS 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, 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. **AWS 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
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. Highly portable architecture that can be easily added to any storage system or cloud storage backend without the need to change storage systems
3. First to support storage-side serverless architecture option that is easy to deploy UDFs, lower cost, better scalability, and improved latency
4. First to support fully automated event driven UDF invocations in area of Programming Storage
5. Work with any workflows, compute platforms, and programming languages in BDA and AI
6. 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)
7. Has the same customer benefits as general Near Data Processing:
   * Significantly reduce network traffic between compute and storage layers
   * 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

## 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 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 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 1.



Fig. 1. 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), 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 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
6. **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.
7. **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.
8. **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.
   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

## 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.

1. **Event Registration**

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.

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. **UDF Invocation**

If matched UDF invocation condition is found, those UDFs will be invoked via OpenFaas APIs by the Caerus UDF Service. 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. 2.



Fig. 2. 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**

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.

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. 3.



Fig. 3. Direct Invocation of UDF

### Programmable Storage – Event-Driven Storage UDF Workflow (Standalone container, 2021 Q3 item)

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. **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 6.

### Programmable Storage – Direct Invocation of UDF Workflow (Standalone container, 2021 Q3 item)

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.

### UDF Pushdown By Compute Platform – Spark SQL UDF pushdown (2021 Q2-Q3 item)

Based on the investigation, here are the changes needed for this support:

1. **Spark Client Side - Catalyst**: the only way for Spark client to pushdown its UDFs is via changing UDFs into Spark expressions, then the UDFs can be part of Spark Catalyst optimization process, and optimizations like predicate pushdown become free. We did POC to manually implement UDF into a custom expression, and proved that indeed Spark client pushed down the UDF. However, this POC needs a private build of Spark core (SQL), which is not ideal. The next investigation will focus on finding way to automatically generate Spark expressions based on Spark UDFs, similar to this Spark-SQL-Macros approach: <https://github.com/hbutani/spark-sql-macros>. Longer term, Spark community (majorly via Databricks’ Spark Compiler team) has been developing new Spark auto translation mechanism of UDFs into Spark expressions, we will keep track that effort and might join in if proper work can be identified.
2. **Spark Client Side - Caerus Data Source**: integration effort is needed for Caerus Spark V2 Data Source implementation.
3. **Storage Side**: the pushed down UDFs will be apply to Caerus SQL engine via its native UDF support.

More details will be added in the future based on ongoing investigation.

## 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)**

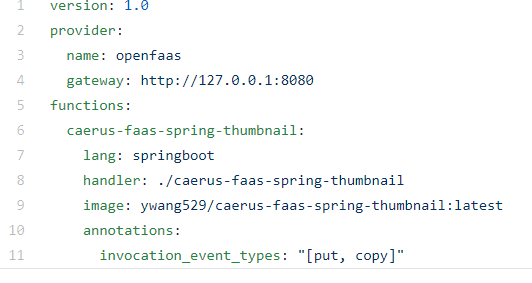
These services, including **Caerus Registry (Redis Cluster)** and **OpenFaas Framework** (K8S etc.) are only one running instance per storage system. Caerus NDP-UDF Backend Services on each storage node behave as multiple client services for these common services.

## Network Architecture

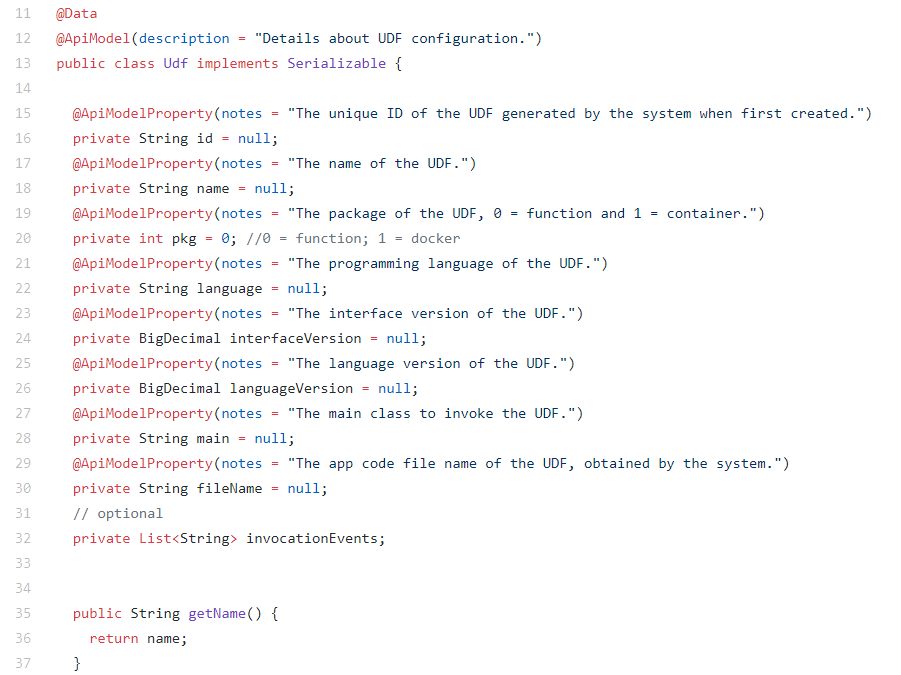
The network port numbers for any microservice are configurable via application configuration files under each service source code. A detail list of the port numbers can be found in the Caerus github project source code location.

## 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

## 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 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 CLI) 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 CLI) 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
  + 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. 4.)

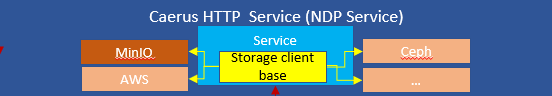
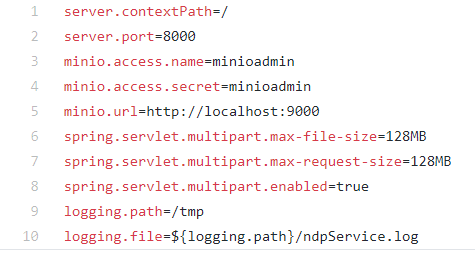


Fig. 4. Caerus NDP Service

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. 5. More detail class APIs and definition can be referenced by the API document (JavaDoc and Swagger API document in the source code).

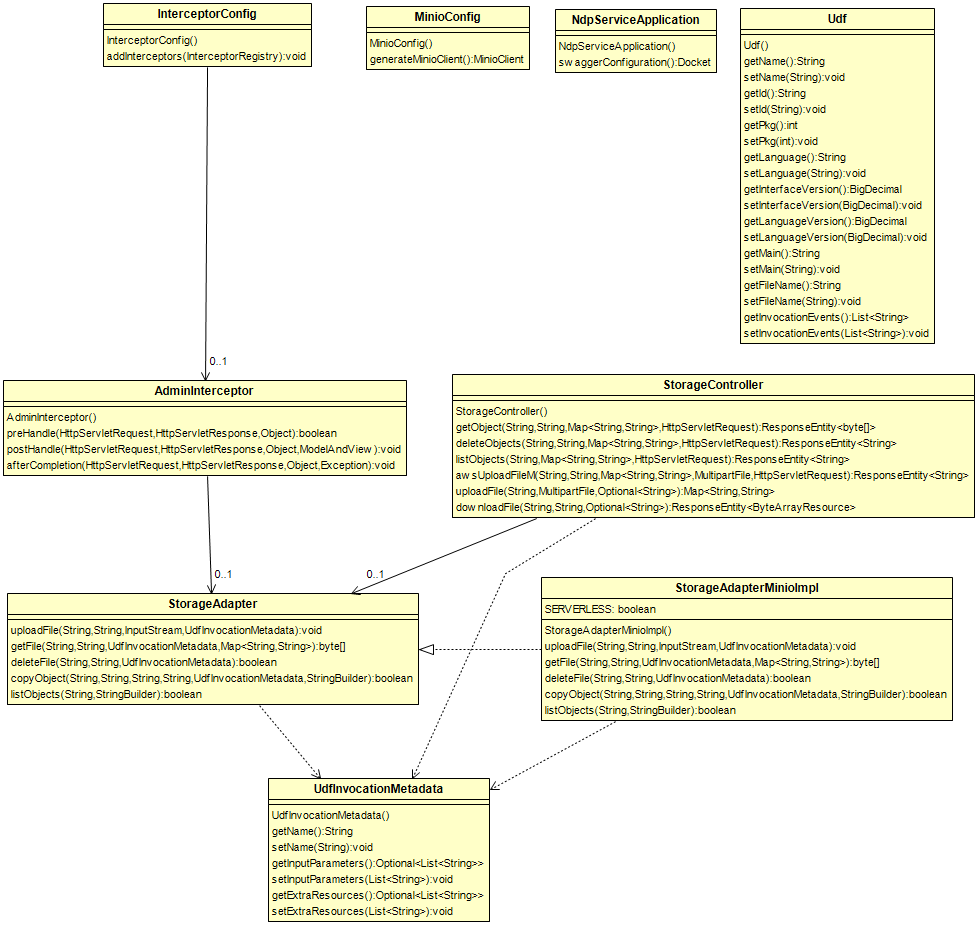


Fig. 5. Caerus NDP Service Class Diagram

## 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. 6. More detail class APIs and definition can be referenced by the API document (JavaDoc and Swagger API document in the source code).



Fig. 6. 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. 7. More detail class APIs and definition can be referenced by the API document (JavaDoc and Swagger API document in the source code).

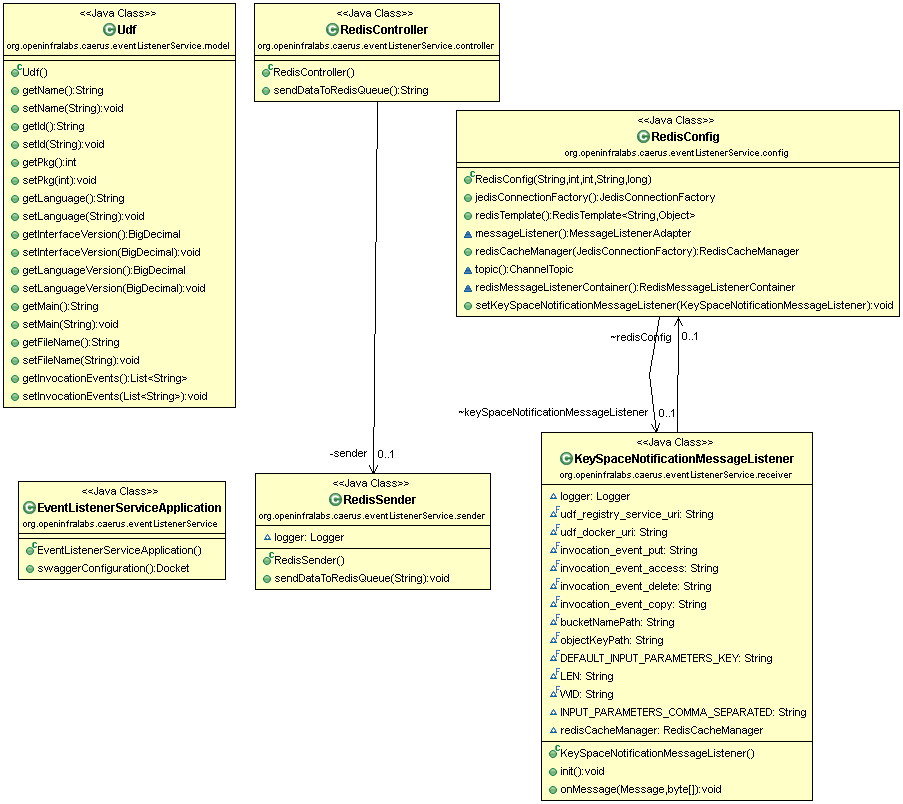


Fig. 7. Caerus Event Listener Service

## 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. 8. More detail class APIs and definition can be referenced by the API document (JavaDoc and Swagger API document in the source code).

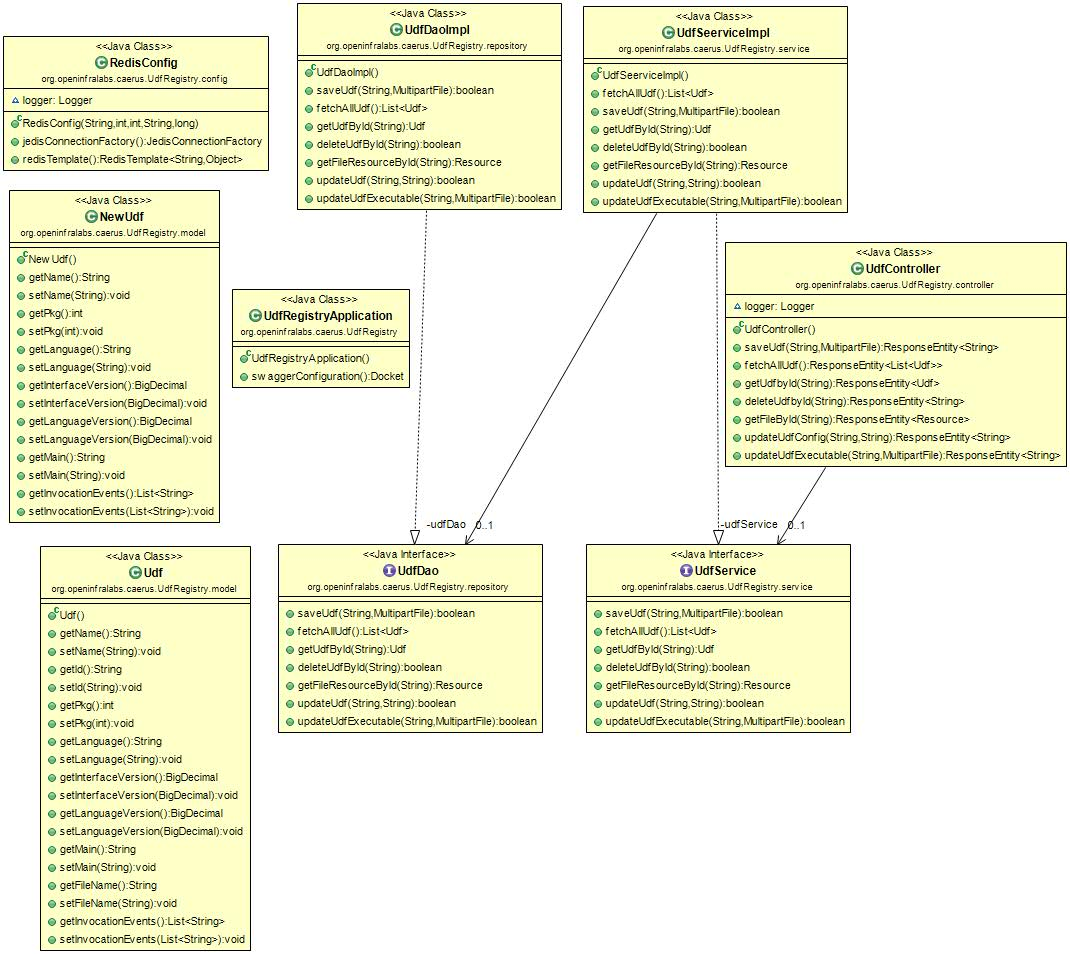


Fig. 8. Caerus UDF Registry Service

## 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. 9. 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).

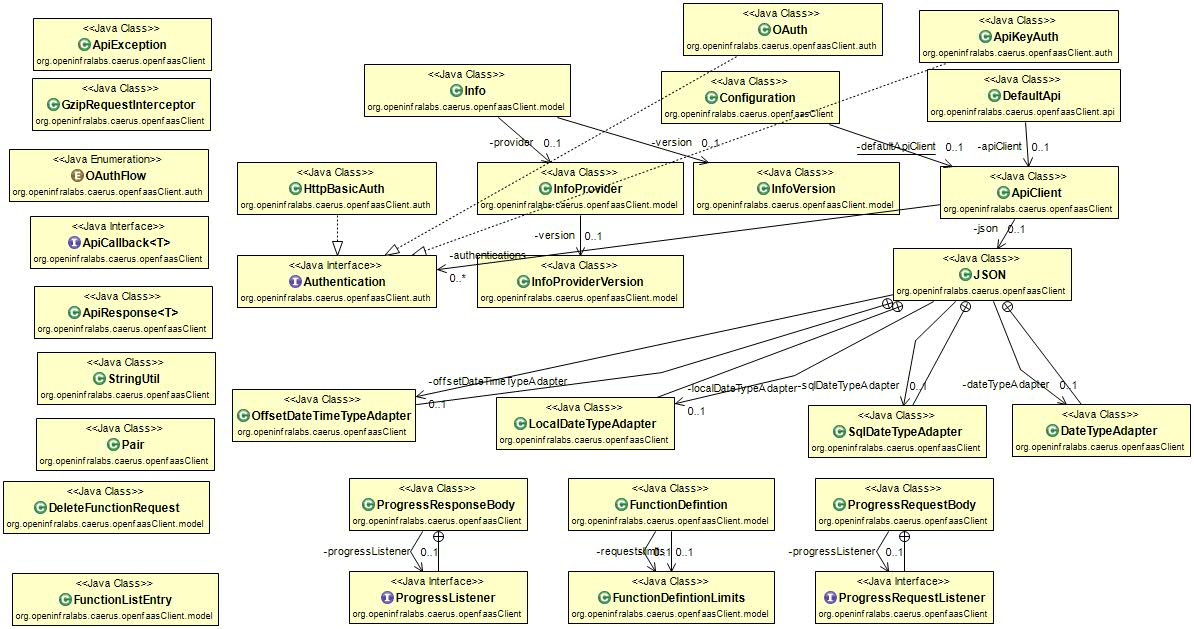
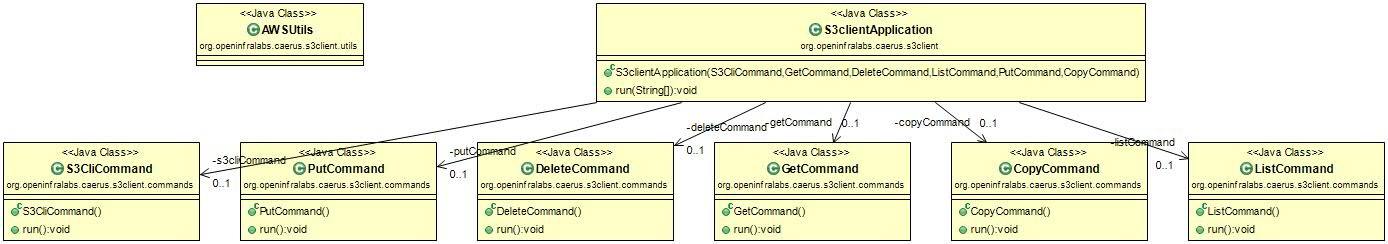


Fig. 9. Caerus OpenFaas Client

## Caerus S3 CLI with UDF Support

Caerus S3 CLI 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. In the future, HDFS etc. client support can be added.

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).



## 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. 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. 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**/”

# 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