Computing and Informatics, Vol. 42, 2023, 878–895, doi: 10.31577/cai_2023_4_878

ARCHITECTURE OF A FUNCTION-AS-A-SERVICE APPLICATION

Ondrej Habala*, Martin Bobák, Martin Šeleng,
Viet Tran, Ladislav Hluchý

Institute of Informatics
Slovak Academy of Sciences
Dúbravská cesta 9
845 07 Bratislava, Slovakia
e-mail: {ondrej.habala, martin.bobak, martin.seleng, viet.tran, ladislav.hluchy}@savba.sk

Abstract. Serverless computing and Function-as-a-Service (FaaS) are programming paradigms that have many advantages for modern, distributed and highly modular applications. However, the process of transforming a legacy, monolithic application into a set of functions suitable for a FaaS environment can be a complex task. It may be questionable whether the obvious advantages received from such a transformation outweigh the effort and resources spent on it. In this paper we present our continuing research aimed at the transformation of legacy applications into the FaaS paradigm. Our test subject is an airport visibility system, a sub-class of the meteorological services required for airport operations. We have chosen to modularize the application, divide it into parts that can be implemented as functions in the FaaS paradigm, and provide it with a simple cloud-based data management layer. The tools that we are using are Apache OpenWhisk for FaaS and Airflow for workflow management, Apache Airflow for workflow management and NextCloud for cloud storage. Only a part of the original application has been transformed, but it already allows us to draw some conclusions and especially start forming a generalized picture of a Function-as-a-Service application.

Keywords: FaaS, serverless computing, cloud computing

Mathematics Subject Classification 2010: 68-U35

* Corresponding author

1 INTRODUCTION

In this paper we present the initial stages of the construction of an airport visibility meteorological application based on the Function-as-a-Service paradigm [1]. The application has been in use by its developer and operator (MicroStep-MIS) for several years now at multiple airports, albeit as a monolithic system, without the use of cloud or serverless computing. Now it is being attempted to transform it into a serverless application. It is not necessary for the application's work, but it is seen by the developer as an investment into the commercial future of the product, as it will:

- allow the developer to modularize the application, potentially offering several different deployments with different requirements, functionality and pricing,
- allow to deploy the application without requiring the customer to invest into hardware and maintenance,
- allow to develop the application into a completely different functionality and even domains,
- give the developer critical know-how based on modern computing paradigms, which will allow it to stay current with its products, as more and more customers move from dedicated hardware to cloud computing and even serverless computing.

The re-development of the application in the FaaS paradigm is done in cooperation with the Institute of Informatics of the Slovak Academy of Sciences – the research partner. For the research partner the development is of interest because of its long-standing research in distributed computing, leading to cloud computing and now serverless computing. While the application's parameters may not be ideal for the serverless paradigm, it is still very well usable, and will benefit from the application of FaaS concepts.

The development of the serverless version of the application is still in its initial, exploratory stages, and the contents of this article reflects this fact. In the following chapters we present the general overview of what is Function-as-a-Service, as well as of the tools we are currently using – OpenWhisk, OSCAR, OpenFaaS and Airflow. We also describe the general architecture and workings of the application as well as the challenges it faces during the process of moving towards a serverless architecture based on FaaS. In the final chapter of the paper we present our future plans for the application and its further transformation.

2 THE FUNCTION AS A SERVICE PARADIGM

Function as a service (FaaS) is a subset of serverless computing [2] that provides a platform allowing developers to write and deploy applications without building and maintaining the underlying infrastructure. Typical tasks related to infrastructure management like resource provisioning, maintenance and regular update of base

operating systems are on the responsible Cloud provider. Developers may focus only on application codes and their logics.

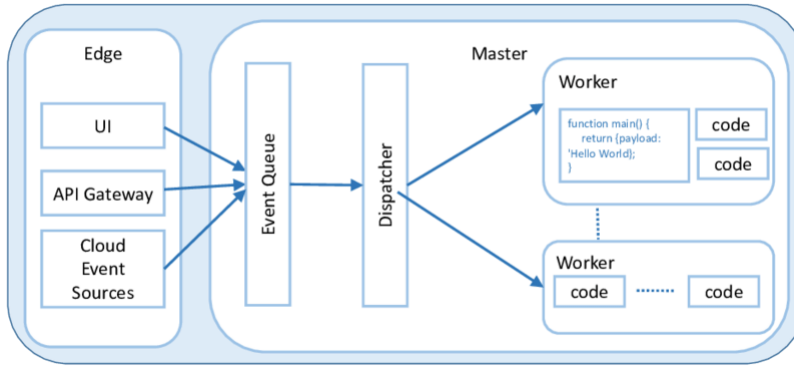

Figure 1. Serverless platform architecture [2]

The architecture of FaaS is shown in Figure 1. The core component of the FaaS platform is the event dispatcher that receives events from different sources: users' requests sent to API gateways, mouse clicks on GUI, time alarms, data arrivals on cloud storages and so on. For each event, application developers can define the corresponding code that will be executed by the dispatcher on worker nodes when the event occurs. An event queue is often placed in front of the event dispatcher for handling high event rates, when all worker nodes are busy in processing other events.

Whole system, including the dispatcher and worker nodes are managed by the FaaS provider so application developers may focus only on the codes that are called by the event dispatcher when an event occurs. The codes corresponding to the events are typically implemented as functions: stateless small pieces of code, therefore the platform is called "Function as a Service". As the code is stateless, dispatchers can execute multiple instances of the code in parallel with very low overheads, so the application is scaled easily, and practically limited only by the computation capacity of the FaaS providers. The billing system is based on the actual resource usage by the functions, if there is no function call, no cost occurs.

Functions have to be implemented in a programming language supported by the FaaS platforms and using libraries provided by the providers. That limits portability of the code and makes potential vendor lock-in. Some FaaS platforms, e.g. OpenWhisk, can support Docker containers as functions so developers can implement the functions in any programming language and with custom libraries. Loading Docker containers at each execution causes much higher costs than function calls, so the containers are often cached and used repeatedly. The FaaS paradigm is also well suited for low-code [3] programming, as the serverless concept frees the developer of a whole class of details (code and data location, connections, etc.).

In summary, the main advantages of FaaS:

Automatic scaling: With FaaS, functions are scaled automatically, independently, and instantaneously according to the actual demands. That will relieve developers from concerns of high traffic or heavy use. The cloud providers will handle all issues related to scaling, e.g. allocating computing resources, replicating the codes and so on.

Cost efficiency: Users have to pay only for the computing resources they really use, not for idle resources that are often reserved for handling possible high demands in the typical IaaS (Infrastructure as a Service). As mentioned above, functions are scaled up automatically on high demands and scaled back down on low demands. If there are no demands or events, no costs are incurred.

Quick development: As developers do not need to manage infrastructure, they can focus only on the code, reducing the cost of development and the time to market.

FaaS also have some disadvantages:

Potential vendor lock-in: The application codes are built on the top of a concrete FaaS platform and difficult to port to another vendor.

Difficulties for testing: The codes are running on the top of a FaaS platform, it may make difficulties for creating local test environments for applications.

FaaS is very suitable for applications that have dynamic or volatile loads as it can scale easily and handle very high demands without big issues, and also has no cost when the application is idle. For applications with constant loads, the cost of FaaS may be higher than typical IaaS solutions.

The first commercial provider offering FaaS is Amazon AWS with AWS Lambda platform [4], followed by Google with Google Cloud Functions [5], Microsoft with Azure Functions [6]. In this paper, we will focus on the open-source FaaS platform OpenWhisk from IBM (described in Section 4).

3 THE APPLICATION – AIRPORT VISIBILITY USING CAMERAS

An important element in the safety of all kinds of transport is good visibility. Poor visibility can cause fatal accidents [7]. Visibility measurement is therefore a relevant issue for air transport during the whole flight but especially when the aircraft is maneuvering on or near the ground [8]. Aircraft accidents due to bad weather conditions comprise almost 50 % of all cases and the main cause of weather-related accidents is reduced visibility [9]. Good visibility information can significantly decrease the risk of accidents, number of redirected flights, save fuel and decrease negative economic consequences.

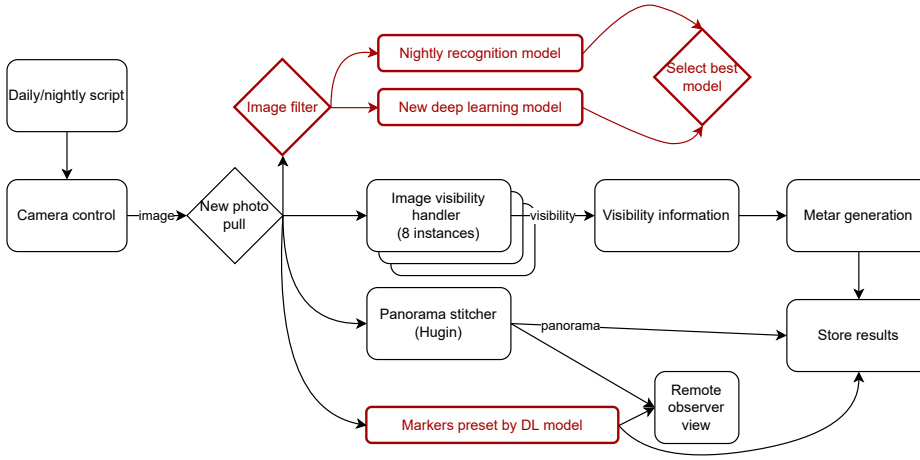

Figure 2. Architecture of the airport visibility application

There exist dedicated sensors for measuring visibility at airports [10], but:

1. they are usually costly to obtain,
2. their coverage is often insufficient,
3. they can only measure visibility at an exact measuring point.

Another approach to determine visibility is based on the research of remote and automatic visibility observation using camera images [11, 12]. This approach addresses all above mentioned problems, as:

1. the required cameras are cheaper than dedicated sensors,
2. cameras can cover larger environment,
3. cameras are often already present at airports and can be re-tasked for visibility measurements.

Visibility measurement using cameras includes instant processing of large number of images from various sources, with image recognition and automatization of modeling processes as well as the application of multiple parallel functionalities and checking mechanisms. Therefore a more sophisticated and highly flexible information infrastructure is necessary.

In our system for visibility determination, the camera images are the basis for a remote observer to estimate visibility with the help of reference points. In the automatic version images taken during good visibility are used to construct reference objects. Then during measurement the system determines which reference objects are visible and which are not, and from that information it can make the decision on visibility conditions. This system can monitor visibility according to aviation

rules for human observers, which is preferable to other arbitrary solutions. Since the responsibility of the system is depending on the capacity of the underlying infrastructure, it can be increased by expanding this infrastructure. It is possible to achieve frequency of visibility measurements of one minute or even less for critical situations – and by altering the frequency requirements, resources and costs can be saved when high frequency is not necessary.

The processes of our visibility measurement application are depicted in Figure 2. Camera configurations for image capture during a day and during a night differ, as there are differences in visibility reporting process by meteorological observers during day and night. Overall the cameras mimic manual observations. Images covering full horizon are taken and processed by the Image visibility handler module. This module finds visible reference points by comparison with the database of available reference points for every image, using edge detection as poor visibility presents itself with loss of contrast and thus loss of edges in the obtained images. Then the Visibility information module determines the minimal and prevailing visibility using recognized reference points. Prevailing visibility is the maximum distance visible throughout at least half of the horizon. Then information about prevailing visibility is displayed as in the METAR[1] reporting tool and results are stored in a database. In parallel runs the process of stitching panorama for remote observers to have better conditions for visibility observations remotely is run and outputs from these observations are also fitted in a METAR report and stored in the results database. We also plan to create new deep learning visibility model that could run in parallel to Image visibility handler module so we can evaluate this method.

The technologies used in the original application have to be modified to fit Function-as-a-Service infrastructure in order to be executed using OpenWhisk or OSCAR frameworks. Image data are acquired using the ONVIF standard. The user interface is provided through a web application server integrated within the IMS [13] software. The UI for IMS is built using industry proven standards: HTTP/HTTPS protocol, HTML and XML formats, JavaScript and AJAX technologies, which makes them well suitable for cloud deployment. The backend implementation uses Java. The UI, since it is HTTP based, is ready for access over the network. The IMS system is also accessible through a web browser. The basic IMS server software also allows for both edge and cloud deployment.

4 APACHE OPENWHISK

OpenWhisk is a free and open implementation of the Function-as-a-Service paradigm described in Section 2. The history of OpenWhisk is tied to the Amazon Web Services' Lambda service, first presented in November 2014. According to Rodric Rabbah, then working at IBM Research, his research group quickly realized the

[1] Standardized message for reporting meteorological conditions obligatorily emitted every half an hour by airports and meteorological institutions.

Lambda's value and the overall value of the serverless computing premise. He saw it as a transformation of computing native to cloud, and the future of the architectures of cloud applications [14]. Therefore, IBM Research started working on a competing product called Whisk. Whisk was later renamed OpenWhisk, open-sourced and transferred to the Apache Software Foundation Incubator [15]. Nowadays it is an open-source alternative to AWS Lambda [4], Microsoft Azure Functions [6] and Google Cloud Functions [5].

4.1 The OpenWhisk Programming Model

OpenWhisk is an event-driven system, in which events from an event source feed into triggers, and via rules cause the execution of actions (functions). The production of event can be done via a command-line tool, via a HTTP call to the OpenWhisk installation, or from numerous existing plugins for various tools, messaging services, and software systems that produce events. One OpenWhisk installation can serve several independent systems of actions, rules and triggers, each in its own namespace. The schema of this process is shown in Figure 3.

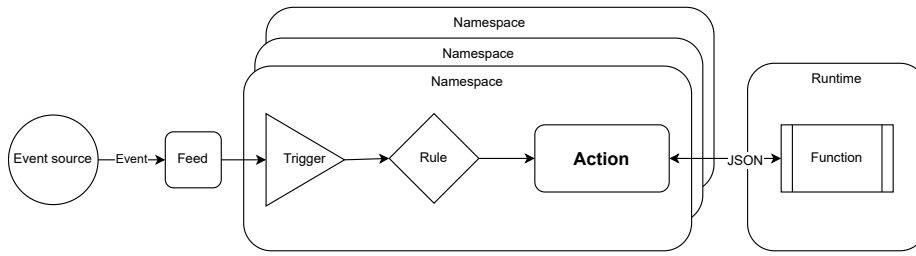

Figure 3. The event-driven programming model of OpenWhisk

The action itself (the principal function that is to be executed as a result of an event occurring) is provided by the developer of the system which uses OpenWhisk. It is a self-contained section of programming code, written in one of several supported programming languages:

1. Java,
2. Javascript,
3. Python,
4. PHP,
5. Go,
6. .NET,
7. Ruby,
8. Swift.

These languages are supported by their own runtime environments, implemented as docker images. Apart from support for functions implemented in these languages, a developer can also provide an action designed as a black box, via providing a specifically constructed docker image based on a provided Docker Runtime[2] image.

As we can infer from this description, each action is instantiated and executed in its own docker container, which contains the necessary software to compile and execute code in the programming language in which the action is written. In the case of a black-box docker action, the docker image itself contains the function, and the OpenWhisk execution system does not provide support with its compilation, it only calls the code implementing the action via an established mechanism.

Independently of which language (and, consequently, which runtime) is used, the process of execution of the action's code is similar:

- The runtime (docker image) contains a REST-enabled server.
- This server receives a JSON[3] structure with input data and the code of the action itself.
- The code of the action is compiled (if necessary) and executed, with the received input data provided to it.
- If the code of the action finishes in certain pre-set time, its result is packaged in JSON, and returned as the result of the initial call to the REST server.
- If the code of the action takes more time, the REST server returns a result indicating that the action is still executing. In that case, the result of the action can be queried later via the action's ID.

In the case of the black box action based on Docker Runtime, no compilation of code is done in the 3rd step and the input data is fed directly to an executable place already present in the docker image.

4.2 Installation and Use of OpenWhisk

OpenWhisk can be installed in several ways, as detailed in OpenWhisk documentation[4]. We have chosen to use Kubernetes installation. OpenWhisk is already available as a Helm chart, so deployment in Kubeapps is very straightforward. Additionally a local installation of the wsk command-line tool is required on the machine, from which the OpenWhisk installation is to be used. The installation and configuration of this tool is also described in OpenWhisk documentation[5].

[2] Apache OpenWhisk runtimes for docker, `https://github.com/apache/openwhisk-runtime-docker#readme`

[3] What is JSON, `https://www.w3schools.com/whatis/whatis_json.asp`

[4] OpenWhisk documentation: deployment options, `https://openwhisk.apache.org/documentation.html#openwhisk_deployment`

[5] OpenWhisk documentation: OpenWhisk CLI (wsk), `https://openwhisk.apache.org/documentation.html#wsk-cli`

4.3 Using OpenWhisk for Airport Meteorology

In the course of our work, we will implement all the modules of the application as shown in Figure 2 as OpenWhisk actions, to be managed and executed either programatically, or via a workflow orchestration system (see Section 5 for one such system which we intend to use).

So far, we have implemented 3 functions:

1. ImageVisibilityHander, as a Java action,
2. Visibility info from 8 xmls, also as a Java action,
3. Panorama stitch, as a black-box docker action, since we use a third party software (Hugin[6]).

In the case of the Panorama stitch action, we have created a specific docker image, based on the above described Docker Runtime image. Our image contains also the Hugin software and a program, which:

1. receives the JSON input structure from the Docker Runtime proxy via the HTTP protocol,
2. decodes it into a set of parameters, including input file URLs,
3. downloads the input files,
4. executes panorama stitching on the downloaded input files using the Hugin software's capabilities,
5. uploads the resulting stitched panorama to cloud storage,
6. indicates success or failure of the panorama stitching operation on output.

The details of the modification of the original Docker Runtime from OpenWhisk are detailed by J. Thomas in [16].

In the case of the two Java-based actions from the above list, we also had to perform additional steps. The software that is necessary to execute the actions requires several third party libraries, including OpenCV[7], which are quite large. The maximum size of an action in OpenWhisk is 48 MB, and this is not sufficient to transfer the software and all the required libraries. Therefore, we had to create our own specific Java Runtime for OpenWhisk actions, based on the standard OpenWhisk Java Runtime[8] provided by the OpenWhisk project. The building of the docker image of all OpenWhisk runtimes is managed by the Gradle[9] build tool. The libraries can be added from an external repository, as shown in [17], or as local files, according to the rules of Gradle build files. In our case we have used local JAR files

[6] Hugin – Panorama Photo Stitcher, `https://hugin.sourceforge.io/`

[7] The Open Source Computer Vision Library, `https://opencv.org/`

[8] OpenWhisk Java Runtime, `https://github.com/apache/openwhisk-runtime-java#readme`

[9] Gradle build tool, `https://gradle.org/`

provided by the developers of the application. The resulting source code for our application-specific Java Runtime is available in our GitHub repository[10].

5 SERVERLESS FUNCTIONS ORCHESTRATION WITH APACHE AIRFLOW

Many scientific applications are so complex that they are often described by complex workflows requiring efficient management and coordination of jobs. They process and analyze large volumes of data through numerous interconnected tasks. The management and organization of the tasks are challenging due to their complexity and their need for scalability and reliability.

Function as a Service (see Section 2) changes the development of applications, allowing developers to focus on code instead of infrastructure management. Apache AirFlow [18] is an open source[11] framework for the management of lightweight serverless functions. It amalgamates individual tasks into a workflow that is expressed as a directed acyclic graph (DAG) representing the workflow structure and dependencies between tasks. Each task within the DAG can be associated with a serverless function, enabling the integration of the FaaS paradigm into workflows.

5.1 AirFlow Architecture

Apache Airflow is composed of the following components (see Figure 4):

Scheduler: orchestrates the execution of tasks defined by a DAG. Execution order is defined by their dependencies, and tasks are triggered according to them.

Workers: execute tasks within a specific environment (e.g. local machine, cluster, or cloud).

Web server: provides a user interface for AirFlow. Users can view and manage workflows, monitor task execution, logs, and metadata of workflow runs.

Metadata database: stores and manages information related to DAGs, tasks, workflows and their executions. It maintains the state and history of workflows.

5.2 Using AirFlow for Airport Meteorology

In our case, the tasks are serverless functions, since the requirements for AirFlow tasks are atomicity and no resource sharing, which the serverless functions meet. The resulting workflow characterizes the relations between its tasks and also defines their execution order.

The paper presents the Airflow workflow (see Figure 5) for the airport meteorology application (see Section 3). The whole application is divided into individual

[10] `https://github.com/IISAS/openwhisk-runtime-java`

[11] Under Apache-2.0 license.

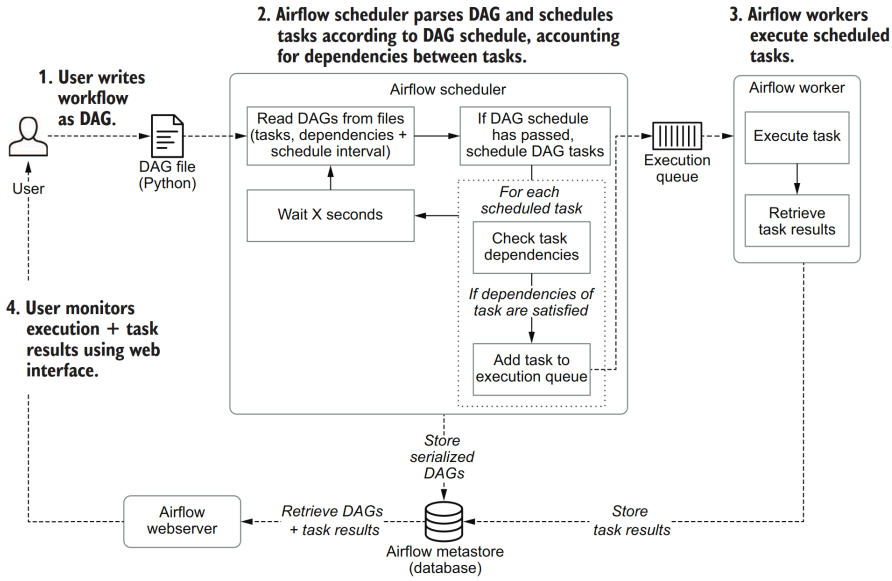

Figure 4. Schematic overview of the process involved in developing and executing pipelines as DAGs using Airflow [19]

tasks which are implemented as Docker containers. The workflow is divided into the following tasks:

1. **day_or_night**: determines whether the captured images fall within the day or night period. This task plays a crucial role in optimizing operational procedures and decision making in aviation workflows.
2. **image_from_camera**: retrieves airport visibility data from selected airport camera data sources. The task integrates real-time visual data into the workflow, allowing further analysis, processing, or decision-making based on the captured images. The images are sent to the following tasks: DL_model_markers, Image_filter_1, Panorama_stitch, and Image_Visibility_Handler.
3. **DL_model_markers**: applies deep learning (DL) models to detect and analyze markers or specific features within the images.
4. **Image_filter_1**: applies a specific image filter (in this case filter #1) to enhance or modify the images from the camera. This task enables the integration of image processing algorithms to manipulate them.
5. **Panorama_stitch**: is responsible for stitching multiple overlapping images together to create a seamless panoramic image. This task uses image processing algorithms and computer vision techniques to align and blend the input images, producing a panoramic view of the airport.

6. **Image_Visibility_Handler**: analyzes and assesses the visibility conditions in images covering the full 360° horizon. This task takes advantage of image processing techniques and visibility assessment algorithms to determine the level of visibility or clarity in the input images.

7. **Visibility_informations**: gathers and provides relevant visibility-related information to observers. This task collects data, generates insights, and presents visibility metrics or indicators derived from the camera images.

8. **Metar_generation**: produces a METAR (Meteorological Aerodrome Report) data based on relevant meteorological parameters. This task gathers weather information, processes it, and generates a standardized METAR report that provides essential weather observations for an airport.

9. **Remote_Observer_View**: enables remote monitoring and observation of an airport area using visual data captured from remote cameras. This task facilitates real-time or near-real-time viewing of the airport, allowing monitoring, anomaly detection, and gathering information without physically being present at the site.

10. **Store_results**: task within the workflow orchestration system is responsible for storing and persisting the outputs generated by previous tasks within the workflow.

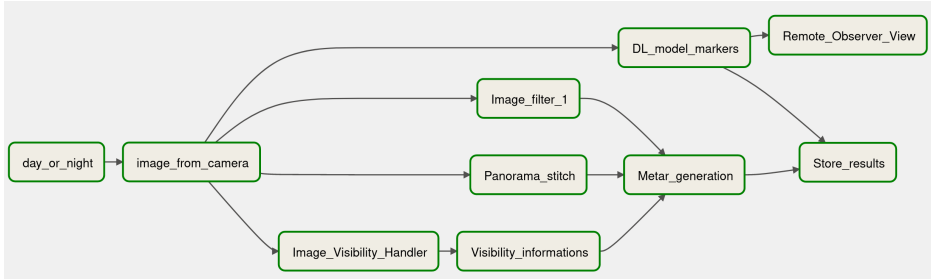

Figure 5. Workflow for the airport meteorology application in AirFlow

One of the key advantages of our Airflow-based workflow orchestration system is the flexibility between data-driven and computation-driven serverless functions. This allows the workflow to efficiently scale the functions based on their utilization, which is a vital aspect of such applications. By dynamically allocating resources to functions as needed, Airflow ensures optimal performance and resource utilization within the workflow.

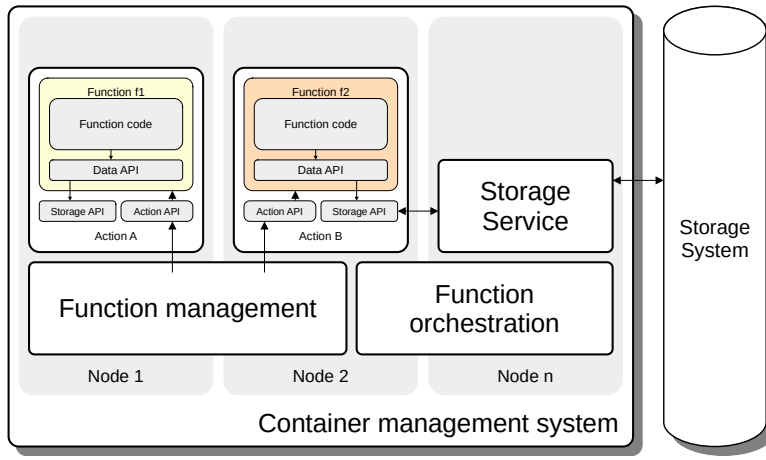

Figure 6. Generic architecture of a Function-as-a-Service application

6 THE ARCHITECTURE OF A FUNCTION-AS-A-SERVICE APPLICATION

Based on our experiments with adapting the airport visibility application to FaaS concepts, we have acquired enough experience to be able to start formulating a generic architecture of a Function-as-a-Service application – see Figure 6.

The whole application is sitting in a cluster of nodes managed by a container management system. We are using the Kubernetes containerization system, but any other system able to manage a cluster of Docker (or others) containers can be used instead of Kubernetes. Inside this cluster is deployed a function management system, able to create parametrized function instances, execute them, provide them with input parameters and communicate their output back to the function caller. In our case we are using the Apache OpenWhisk as the function manager, but OpenFaaS [20] could be also used, for example. To actually create an application out of a group of functions, they need to be orchestrated. For this the architecture containes a function orchestration component, which is able to create workflows of functions, provide them with inputs and store or display their outputs. In our case, we are using the Apache Airflow. Additionally, the cluster contains a deployment of a storage management system. In our case we are using the NextCloud system and its WebDAV server, but this can be replaced by any other cloud storage management, providing other data access protocols – FTP, NFS, SMB and others. The choice of a concrete cloud storage management is dependent on the requirements of the application itself.

Above this software infrastructure are the functions themselves. In the terminology of Apache OpenWhisk, which we have adopted in our research, a prepared

function is called an action. An action can be activated, thus executing the function which it contains on concrete input data. The whole function is executed inside a runtime environment – in the case of Apache OpenWhisk, and any other FaaS framework using containerization, a runtime environment is a pre-prepared Docker container, containing all the necessary software to execute the function code. So the concrete runtime environment is selected based on the characteristics of the function. A generic Apache OpenWhisk installation distinguishes functions only by the programming language in which they are written, and provides runtime environments – Docker images – for functions written in Java, JavaScript, PHP, GO, Python, .NET, Swift and Ruby. It also provides a "black-box" runtime environment for functions not written in a managed programming language. However, this Docker image is just a template, and has to be modified for every such a function.

Each runtime environment provides certain APIs and interfaces which allow the function to interact with the user, the orchestration system, and the storage management system. Most important of these is a REST API receiving the function code and input data, compiling, executing the function code, and returning its result. In the case of Apache OpenWhisk this REST API is provided by a pre-prepared server, which is part of each runtime environment.

Another important part of the function infrastructure is the access to storage management. In our architecture, we have divided the storage management infrastructure available to functions into two layers. The lower layer, named Storage API in Figure 6, is pre-configured with the actual storage end-point information (the server data, so to say). The higher layer, through which the function code accesses data, called Data API in Figure 6, is server-agnostic, or *serverless*. Its structure and method signatures depend on the application's domain, not on the actual underlying storage technology. For example in the case of our application, dealing mainly with large collections of time-coded images, the data are identified by date, time, image origin and image resolution – all metadata. The resulting data structure created in the Storage System and its translation from/to the metadata coordinate system is the responsibility of the Data API, and the function itself is unaware of it.

The runtime environment may contain other APIs or interfaces, depending on the requirements of the application. For example an application which uses a camera system may require that the runtime environment provides an abstract API for accessing these cameras. An application using distributed networked sensors may require an IoT API, and an application using artificial intelligence and machine learning methods will require APIs which will give it access to specialized hardware. All these APIs should follow the pattern used by the storage API – a lower level API configured for the specific external environment (camera network, IoT network, GPU cluster), and a higher-level API used by the function itself, agnostic of the hardware specifics and using only metadata to express task parameters. This will allow the function developer to actually work in a serverless environment.

7 SUMMARY AND FUTURE WORK

In this paper we have described the process of transforming a legacy application into the Function-as-a-Service paradigm. The application is used for measuring visibility at airports. We have described how it has been divided into modules, and how these modules are being transformed into serverless cloud functions. We have also shown the use of OpenWhisk to manage and run those functions (actions in OpenWhisk terminology), and how to manage the actions using Apache AirFlow.

Most importantly, we have been able to derive from this work a partial architecture of such a serverless cloud application. The architecture includes data access, which is not being handled by OpenWhisk or any of the other FaaS tools we use. The architecture of the data access subsystem adheres to the serverless principle – the action code accesses data by their metadata, not by their location.

In our future work, we will continue to transfer more of the application's blocks into the serverless cloud as OpenWhisk functions. We will also extend the generic architecture, to encompass more of the application's components, and if possible include other methods of data storage, like NoSQL [21]. We will try to extract a generalized methodology for the transformation of legacy applications into the serverless cloud paradigm, and use it to transfer another application to which we have access, for example [22].

Another goal of our future work is to try and apply a cloud risk assessment method to the application [23].

Acknowledgements

This publication is the result of the project implementation: Research on the application of artificial intelligence tools in the analysis and classification of hyperspectral sensing data (ITMS: NFP313011BWC9) supported by the Operational Programme Integrated Infrastructure (OPII) funded by the ERDF. It is also supported by APVV grant No. APVV-20-0571 and VEGA grant No. 2/0131/23.

REFERENCES

[1] Chowhan, K.: Hands-on Serverless Computing: Build, Run and Orchestrate Serverless Applications Using AWS Lambda, Microsoft Azure Functions, and Google Cloud Functions. Packt Publishing Ltd., 2018.

[2] Baldini, I.—Castro, P.—Chang, K.—Cheng, P.—Fink, S.—Ishakian, V.—Mitchell, N.—Muthusamy, V.—Rabbah, R.—Slominski, A.—Suter, P.: Serverless Computing: Current Trends and Open Problems. In: Chaudhary, S., Somani, G., Buyya, R. (Eds.): Research Advances in Cloud Computing. Springer, Singapore, 2017, pp. 1–20, doi: 10.1007/978-981-10-5026-8_1.

[3] Marek, K.—Śmiałek, M.—Rybiński, K.—Roszczyk, R.—Wdowiak, M.: BalticLSC: Low-Code Software Development Platform for Large Scale Compu-

tations. Computing and Informatics, Vol. 40, 2021, No. 4, pp. 734–753, doi: 10.31577/cai_2021_4_734.

[4] Sbarski, P.—Cui, Y.—Nair, A.: Serverless Architectures on AWS, Second Edition. Manning, 2020.

[5] Rose, R.: Hands-on Serverless Computing with Google Cloud: Build, Deploy, and Containerize Apps Using Cloud Functions, Cloud Run, and Cloud-Native Technologies. Packt Publishing Ltd., 2020.

[6] Kumar, V.—Agnihotri, K.: Serverless Computing Using Azure Functions: Build, Deploy, Automate, and Secure Serverless Application Development with Azure Functions. BPB Publications, 2021.

[7] Abdel-Aty, M.—Ekram, A. A.—Huang, H.—Choi, K.: A Study on Crashes Related to Visibility Obstruction Due to Fog and Smoke. Accident Analysis and Prevention, Vol. 43, 2011, No. 5, pp. 1730–1737, doi: 10.1016/j.aap.2011.04.003.

[8] Özdemir, E. T.—Deniz, A.—Sezen, İ.—Menteş, Ş. S.—Yavuz, V.: Fog Analysis at Istanbul Ataturk International Airport. Weather, Vol. 71, 2016, pp. 279–284, doi: 10.1002/wea.2747.

[9] Bueso, J.—Rojas Gregorio, J.—Lozano, M.—Pino Gonzalez, D.—Prats, X.—Miglietta, M.: Influence of Meteorological Phenomena on Worldwide Aircraft Accidents in the Period 1967–2010. Meteorological Applications, Vol. 25, 2018, pp. 236–245, doi: 10.1002/met.1686.

[10] Burnham, D. C.—Spitzer, E. A.—Carty, T. C.—Lucas, D. B.: United States Experience Using Forward Scattermeters for Runway Visual Range. U.S. Department of Transportation, Federal Aviation Administration, 1997.

[11] Bartok, J.—Ivica, L.—Gaál, L.—Bartoková, I.—Kelemen, M.: A Novel Camera-Based Approach to Increase the Quality, Objectivity and Efficiency of Aeronautical Meteorological Observations. Applied Sciences, Vol. 12, 2022, No. 6, Art. No. 2925, doi: 10.3390/app12062925.

[12] Bartok, J.—Šišan, P.—Ivica, L.—Bartoková, I.—Malkin Ondík, I.—Gaál, L.: Machine Learning-Based Fog Nowcasting for Aviation with the Aid of Camera Observations. Atmosphere, Vol. 13, 2022, No. 10, Art. No. 1684, doi: 10.3390/atmos13101684.

[13] MicroStep-MIS: IMS4 Remote Observer. https://www.microstep-mis.com/drupal/web/sites/default/files/datasheets/IMS4%20Remote%20Observer_product%20sheet.pdf.

[14] Sciabarra, M.: Learning Apache OpenWhisk. O'Reilly Media, Inc., 2019.

[15] Foundation, T. A. S.: OpenWhisk Incubation Status - Apache Incubator. https://incubator.apache.org/projects/openwhisk.html.

[16] Thomas, J.: OpenWhisk Docker Actions. 2017, https://jamesthom.as/2017/01/openwhisk-docker-actions/.

[17] Thomas, J.: Large (Java) Applications on Apache OpenWhisk. 2019, https://jamesthom.as/2019/02/large-java-applications-on-apache-openwhisk/.

[18] Haines, S.: Workflow Orchestration with Apache Airflow. Modern Data Engineering with Apache Spark: A Hands-on Guide for Building Mission-Critical Streaming

Applications, Apress, Berkeley, CA, 2022, pp. 255–295, doi: 10.1007/978-1-4842-7452-1_8.

[19] Harenslak, B. P.—de Ruiter, J. R.: Data Pipelines with Apache Airflow. Manning, 2021.

[20] Bobák, M.—Hluchý, L.—Tran, V.: Tailored Platforms as Cloud Service. 2015 IEEE 13th International Symposium on Intelligent Systems and Informatics (SISY), 2015, pp. 43–48, doi: 10.1109/SISY.2015.7325408.

[21] Faridoon, A.—Imran, M.: Big Data Storage Tools Using NoSQL Databases and Their Applications in Various Domains: A Systematic Review. Computing and Informatics, Vol. 40, 2021, No. 3, pp. 489–521, doi: 10.31577/cai_2021_3_489.

[22] Krammer, P.—Kvassay, M.—Forgáč, R.—Očkay, M.—Skovajsová, L.—Hluchý, L.—Skurčák, L.—Pavlov, L.: Regression Analysis and Modeling of Local Environmental Pollution Levels for the Electric Power Industry Needs. Computing and Informatics, Vol. 41, 2022, No. 3, pp. 861–884, doi: 10.31577/cai_2022_3_861.

[23] Zbořil, M.: Risk Assessment Method of Cloud Environment. Computing and Informatics, Vol. 41, 2022, No. 5, pp. 1186–1206, doi: 10.31577/cai_2022_5_1186.

Ondrej Habala is a researcher at the Institute of Informatics of the Slovak Academy of Sciences. He works mainly with distributed computing systems and cloud systems, applying them towards solving domain-specific problems mainly in meteorology and hydrology. He has participated in more than 10 national and international research projects, including EU FP5, FP6, FP7, H2020 and HE projects. He is the author of more than 80 publications in his research field.

Martin Bobák works as a researcher at the Institute of Informatics of the Slovak Academy of Sciences. His focus is mainly in cloud computing and cloud architectures. He has participated in several research projects, including FP7, H2020 and HE European research programs. He is the autor of more than 25 scientific publications in the areas of cloud computing, artificial intelligence and data science.

Martin Šeleng is a researcher at the Institute of Informatics of the Slovak Academy of Sciences. He specializes in research infrastructures, cloud computing, and machine learning. He has participated in several research projects, including in the FP5-FP7, H2020 and HE European research programs. He is the author of over 50 scientific publications.

Viet Tran is a senior researcher at the Institute of Informatics of the Slovak Academy of Sciences. His main work is in cloud computing and research infrastructures. He is a senior team leader at II SAS, and has led II SAS researchers in several H2020 and HE European research projects. He is the author of approximately 100 scientific publications.

Ladislav Hluchý is a senior researcher and the head of the Department of Parallel and Distributed Information Processing at the Institute of Informatics of the Slovak Academy of Sciences. He has been active in European research programs since FP4, and has led II SAS team in dozens of research projects in FP4, FP5, FP6, FP7, H2020 and HE. Over his research career he has been the author of over 150 scientific publications. His specialization is distributed information processing, cloud computing and data science.