Computing and Informatics, Vol. 39, 2020, 678–694, doi: 10.31577/cai_2020_4_678

COOKERY: A FRAMEWORK FOR CREATING DATA PROCESSING PIPELINE USING ONLINE SERVICES

Mikołaj Baranowski

Multiscale Networked Systems (MNS), Institute of Informatics
University of Amsterdam
Amsterdam, Netherlands

Adam Belloum

Multiscale Networked Systems (MNS), Institute of Informatics
University of Amsterdam
Amsterdam, Netherlands
&
Netherlands eScience Center
Science Park 140, 1098 XG Amsterdam, The Netherlands
e-mail: a.s.z.belloum@uva.nl

Reginald Cushing, Onno Valkering

Multiscale Networked Systems (MNS), Institute of Informatics
University of Amsterdam
Amsterdam, Netherlands
e-mail: {r.s.cushing, o.a.b.valkering}@uva.nl

Abstract. With the increasing amount of data the importance of data analysis in various scientific domains has grown. A large amount of the scientific data has shifted to cloud based storage. The cloud offers storage and computation power. The Cookery framework is a tool developed to build scientific applications using cloud services. In this paper we present the Cookery systems and how they can be used to authenticate and use standard online third party services to easily create data analytic pipelines. Cookery framework is not limited to work with standard

web services; it can also integrate and work with the emerging AWS Lambda which is part of a new computing paradigm, collectively, known as serverless computing. The combination of AWS Lambda and Cookery, which makes it possible for users in many scientific domains, who do not have any program experience, to create data processing pipelines using cloud services in a short time.

Keywords: Function-as-a-service (FaaS), serverless computing, AWS Lambda, domain specific languages (DSL)

1 INTRODUCTION

Cloud computing is getting a more and more prominent factor in life. People use cloud services every day, sometimes without even knowing. Cloud services come in a variety types like storage services, computation services, video streaming services and much more. Dropbox [1] and Google Drive [2] are well-known examples of cloud storage services that are widely used, with Dropbox claiming to have 600 million users in 2020 [3]. On the other side, more people tend to use cloud services for business purposes. Using cloud services, it is easier to collaborate across geographically distributed locations. In the Harvard Business Review: Cloud Computing Comes of Age [4], it is stated that cloud software greatly reduces the implementation time and it does not need a big up-front investment. It is also stated that a cloud provider could have an application up and running in five weeks, contrary to the 18 months that it would take according to the IT-business. On the other hand, the same review also shows that security of these cloud services is still the biggest barrier. For a number of potential cloud service users from various scientific disciplines like life sciences, health research, humanities, social sciences, environmental science, or earth science taking advantage of these cloud services has become more difficult and complex due to the many programming languages available and the documentations that accompany the APIs (Application Programming Interfaces) being rather technical. Besides not all scientists can afford to setup their own server; hence the cloud approach provides a very cost effective solution. They can buy some computing time or storage somewhere in the cloud, which will be cheaper at the beginning as opposed to buying a server. However, running a service on cloud resources comes with some additional problems. There is a need to build a software infrastructure to handle all the requests your application will receive. This is a long and tedious job that not everybody can do. In order to make programming using cloud services easier, Cookery has been developed in the context of Ph.D. research work at the University of Amsterdam [6].

Cookery framework allows users to develop applications that can connect features of multiple cloud service providers together. The Cookery approach is focused on programming distributed systems through a domain specific language. This approach is similar in terms of functionality to IFTTT (If This Then That) [5].

IFTTT is a web-based service that allows its users to create chains of conditional statements [25]. These chains contain triggers from cloud services that can invoke actions in other cloud services. Thus, IFTTT can be used to automate a web-application task. It will be useful to have the ability to extract, combine and exploit the best features of their favorite online services. With this ability users can develop applications that can automate their tasks, extend usability or simply give the user easier access to their data. In order to achieve this goal, a proof of concept will be developed in which the Cookery architecture is designed for optimal combination of functionalities of multiple cloud service providers. Important features of this architecture are the cloud service provider APIs, a server to handle authentication, Jupyter Notebook for user interaction and of course the Cookery kernel implementations itself.

Cookery enables scientists to combine multiple cloud applications in an easy way. On top of that, Cookery uses its own Domain Specific Language (DSL) to make it more accessible for users without any programming experience. The Cookery system comes with a couple handy features to make using cloud application seamless and easy to use, among interesting:

1. Cookery is integrated with the Jupyter notebook,
2. it implements the OAuth 2.0 protocol for a convenient authorization, and
3. it supports the use of AWS Lambda functions.

2 COOKERY

Cookery is a framework to make programming with other cloud applications easy [6]. Cookery makes it possible to combine cloud services using a high-level language. This high-level language, or Cookery language, has the same syntax as English, which makes it possible for users without any programming experience to easily create applications pipeline. A closer look at Cookery shows that it is actually composed of three layers, which can be seen in Figure 1.

- The first layer (Layer 1) is used by a user to create Cookery applications using the Cookery DSL language. In this layer data processing pipeline can be defined and modified, in this paper we will refer to the data processing tasks as "activities". This is the highest abstraction level, which requires the least programming skills.
- The second layer (Layer 2) is for developers and, as opposed to the previous layer, this layer makes use of the Cookery Domain Specific Language (DSL). Layer 2 is used for defining and modifying actions, subjects and conditions. This layer abstracts the infrastructure details from application domain programmers.
- The third and last layer (Layer 3) is the Cookery backend and is also intended for developers. At this level, developers can implement protocols, which are for the activities and data, and communication with execution environments. This is the most lower layer offered by the Cookery framework.

Programming a Cookery layer is composed of simple statements called activities that will be executed by the framework. These activities consist of other elements, namely variables, actions, subjects and conditions. These elements all have their own role within the Cookery language. Variables are optional. They help to keep track of results of activities that could be referenced later using the labels; this feature helps to create data flows. An action refers to its implementation in the Cookery DSL and it represents remote operations. Subjects represent the input or output of an application, also known as remote data. They can, for example, be used for retrieving data from a cloud service. Both actions and subjects can be followed by arguments and both implementations are divided between all three levels. Conditions are used with keywords, like if or with, to separate them from the rest of an activity. Those are routines defined in Cookery DSL and are meant to transform data before it is passed to an action. This data can be retrieved in different ways, including from a remote location in a subject or from a variable. Layer 2 helps to develop the Cookery DSL, which allows Layer 1 users to use appropriate actions, subjects and conditions (Cookery elements). These elements all consist of a name, a regular expression and a Python routine. Cookery comes with a toolkit in order to make things easy. The toolkit enables a user to execute Cookery applications, generate new projects and evaluate expressions. More details about the Cookery language can be found in [6].

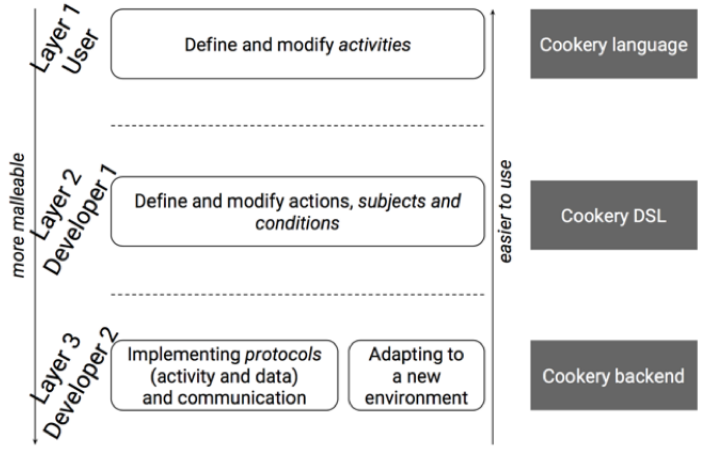


Figure 1. User roles and layers in Cookery [6]. The first layer is for developing applications, with Cookery language, the second layer is for defining actions, subjects and conditions using the Cookery DSL and the third layer is the back-end where protocols are implemented.

It can also provide more sophisticated features as code completion and transformation of rich objects (e.g., graphs). For languages that have Python bindings

(such as Cookery) the implementation of a kernel is much simplified using a kernel wrapper [26] which reuses the kernel machinery in IPython to create new kernels.

In the next subsection we present how Cookery systems implement the authentication and enable the use of AWS Lambda functions.

2.1 Authentication

Because Cookery Layer 1 users create data pipelines that can connect features of multiple cloud service providers together, these services often require third party access and involve payment. One of the main principles behind Cookery is that the user of Cookery application pays for its execution, not the application developer. While it makes a fair model for ensuring application reusability, it requires addressing a non-trivial issue of providing someone's resources to already existing application. Among the commonly used authentication methods OAuth 2.0 is one of the best option to enable third party access. The OAuth protocol provides a generic framework to let a resource owner authorize a third party that wants to access to its resources [14]. The widely used OAuth protocol is not yet fully without flows [15], however it does provide the user with the best combination of convenience and safety. OAuth does not expose the user to third party root access-right threats, and keeps the user in control over granted permissions. At the same time it brings a very convenient experience to the user because a minimal action from the end-users is required. Cookery takes a role of an OAuth client, it requests authorization from remote services and stores granted token for future use. A token is a cryptographically signed piece of text that is handed by the service to the client. Any future client to service communication can be done with the token. The service validates the token on requests which authorizes the service call. These limited time token can be given by the user to third party services so that they can access your resources on your behalf. The user and the service can decide to revoke the token thus blocking third party access. It can be successfully used in very different use-cases presented in Section 3. We decided not to make OAuth an essential nor required part of Cookery; it can be optionally used by layer 3 developers. Figure 2 shows how Cookery achieves OAuth authentication, Jupyter Notebook interacts with programming languages through the kernel. It acts as a middleman; it executes code given by Jupyter Notebook (provided by a user), executes it and handles results by transforming it to the form required by the target service.

2.2 Serverless Computing – AWS Lambda

Amazon Lambda is a service that provides serverless application deployment, many cloud service providers such as Google Cloud Functions [27] and Azure Functions [28] have FaaS in their portfolio. However, due to the programming language limitations of the other two (Google to JavaScript and Microsoft to JavaScript, *C#*, *F#*, and

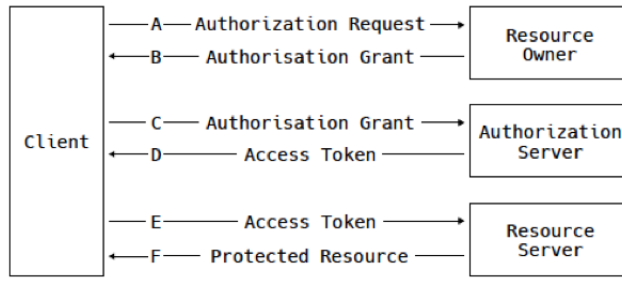


Figure 2. In this diagram of OAuth protocol flow [4], Cookery plays a role of a client regardless how it is deployed (Jupyter notebook, local script or on serverless computing platform such as Amazon Lambda). Resource owner can be understood as Cookery user while, from technical perspective, Resource owner can be the same entity as Authorization server. This, in turn, is a cloud service provider such as Google Cloud, Amazon AWS, etc. – usually sharing the same API as Resource Server.

scripting options: PHP, Bash, Batch, and PowerShell) we decided to experiment first with Amazon Lambda with Python 3.6. Our aim is to provide IFTTT [2] inspired functionality:

1. Deploy Cookery applications to Amazon Lambda with a single command. Deployment requires gathering all application sources including Cookery application, Cookery framework sources and all its dependencies within one zip file and upload it together with Lambda function specification (more details are given in Section 2).
2. Allow specifying periodic invocations. Periodic invocations require a trigger. Amazon Lambda supports several ways of triggering Lambda functions, we chose to use Amazon Cloud Watch Events [8]. It allows scheduling Lambda invocations with an arbitrary interval (more details are given in Section 2).
3. Enable connections to third-party cloud services. In order to enable connection to third-party cloud services, Cookery application has to be authenticated. If Cookery is used in an interactive mode – after invoking the code, user waits for results – all the missing authorization can be asked on demand. In case of serverless deployment, there are two components that have to be authenticated at different time:

    - Amazon Lambda service for application in order to deploy Cookery application in Amazon Lambda.
    - Protocols used in Cookery application during application execution on Amazon Lambda.

2.3 Scheduling

We will use AWS CloudWatch to schedule AWS Lambda functions, by creating rules. The scheduling function needs fewer parameters than the deployment function, which makes it easier to implement. A user will need to specify the period with which the function needs to be invoked. This period consists of a number and a period specifier like minutes, hours or days. Cookery adds some constraints to make it easier to work with a period. On top of that Cookery can also easily check if a rule already exists. The Cookery interface for scheduling Lambda functions first checks the given period and, with the rule name, checks whether this rule already exists. When this is the case, it just adds the Lambda function as a target to the rule. Because AWS regulates a limit of 5 targets per rule and a maximum of 100 rules, it is always possible to add every Lambda function to a rule. Cookery has to explicitly add permission for the rule to invoke the Lambda function; otherwise AWS will give an error. To add permission, the name of the rule is needed to acquire its ARN. A simple Cookery API Call allows the users to deploy and schedule a Lambda function using one API call, as shown in Listing 1.

```
cookery deploy
    --function_name=...
    --file_name=...
    --handler=...
    --every=... COOKERY_PROJECT_PATH
Options:
    --function_name TEXT Name of the Function
    --file_name TEXT Name of the file where the handler function can be found,
    --handler TEXT name of the handler function
    --every TEXT the interval to invoke the function m like 1,m, 1hm or 1d,
    --help show this message and exit
```

Listing 1. Cookery command line API Call to deploy and schedule AWS Lambda functions

3 APPLICATION

To illustrate how Cookery authenticate to remote services, we will use a very simple proof of concept use case which consists in implementing a GitHub monitor that takes a repository property (in this case commits) and monitors it for changes (Figure 3). Whenever a change is detected, it triggers an action (email notification). The proof of concept application works with the Jupyter notebook, which is a web application itself. With this proof of concept application users can easily automate a GitHub related notification process.

The access to the repository is achieved by the implementation of an OAuth authentication, which is explained in detail in Figure 2. This implementation gives the user a convenient and secure authentication process. The monitoring works via periodic calls by a recursive function. This function compares the newest

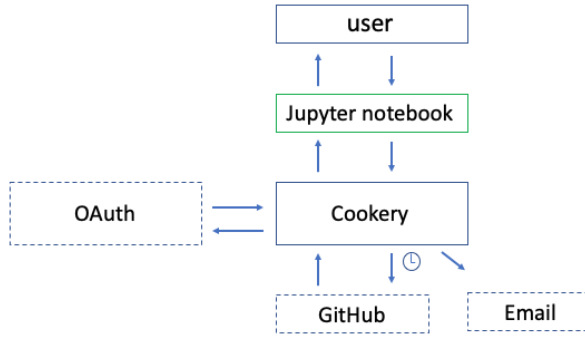


Figure 3. Scheme of the proof of the concept application GitHub monitor. The complete code implementing the GitHub monitor in GitHub [30]

commit with the latest monitored commit. If there is a difference between these two commits, an action is triggered. Because of the interval base of the monitoring mechanism, it could happen that two commits are made in one interval. When this happens only one notification gets send. The chance of this happening can be reduced by decreasing the interval. However, this will increase the server load.

Another variant of the GitHub monitor has been implemented using AWS Lambda [10]. This version of GitHub monitor application makes use of the GitHub REST API to get all the information while being authenticated with a GitHub personal access token. The scopes of this token can be configured by the user, like giving access to all private repositories, but not being able to delete them or to change user data. Making authenticated requests gives the opportunity to send 5 000 requests per hour to the API, while making unauthenticated requests only gives us 60 requests per hour. This program first checks if the authenticated user has access to the given repository. This is done by sending a request using Python library *urllib*. The request is also needed to authenticate to GitHub by adding a header with the personal access token to it, which can be seen in Listing 2.

```
cookerydef
make_request (url):
  request = urllib.request.Request(url)
  request.add_header("Authorization", "token"+github_token)
  response = urllib.request.urlopen(request)
  data = response.read().decode("utf−8" )
  return data
```

Listing 2. The function to make requests to GitHub with authentication using urllib

Implementation of the GitHub monitor with using AWS Lambda functions. The complete code implementing the GitHub monitor using AWS Lambda is available

```
cookery deploy
        -- function_name = "github_monitor"
        -- file_name = "github_monitor"
        -- handler = "github_monitor.handle"
        -- every = 5mn "https://github.com/mikolajb/cookerylambda/blob/master/
           github_monitor/github_monitor.py"
```

Listing 3. Cookery call to deploy and schedule the GitHub monitor AWS Lambda Function

in GitHub [31]. When a GET request is sent to `https://api.github.com/user/repos`, a response in JSON format is sent back, which can easily be deserialized to a Python object, containing the list of accessible repositories. This way we can examine every repository and check if the target repository is among them. A second more specific request is send to get the list of all the commits, sorted by time, with the last commit first. In the response, we can see how many additions and deletions are made, and what the changes are in every file. This information will then be put into the body of an email, which is sent with Gmail using the Python SMTP library. The authentication of Gmail is done with an application key, just like with GitHub.

As mentioned before, the Lambda functions are stateless and terminate after 5 minutes. This means that we cannot use any variables after termination, except when we save them to a database or communicate them back. The way the GitHub monitor works is thus the simplest. We have the same problem with giving the repository to check. This information will be lost after termination and needs to be given every time the function gets invoked. This is where the environmental variables come in. They can be given to a Lambda function as key-value pair and they will be the same for every invocation when deployed. This makes it also possible to reuse the GitHub monitor to check another repository, without changing the code.

3.1 Finding Life Trends Using Data Analysis in Cookery

The use case shows how to create a data pipeline in Cookery [11]. The use case is about analyzing Gmail in order to visualize life trends. As pointed out in Section 2, Cookery framework offers a separation between the creating of general function by developers (Layer 2) and the creation of user-specific programs by the end-user (Layer 1). The layers reduce the amount of code that the end-user has to write, but still allow flexibility. Layer 1 is where the end-user will be interacting. The user will combine already defined activities to create a Cookery program. The use case aims to create in Cookery a data processing pipeline similar to Stephen Wolfram "The Personal Analytics of My Life" [12]. In our case the personal analytics is limited to analyzing the emails of one of the authors. The data analytics pipeline combines a number of online services providing the initial data Gmail service, and Google

analytics service. A visual representation of the data flow for the project is shown in Figure 4. This means that Cookery and the data pipeline within the need to handle data from multiple sources, some of which need to be stored to a file or be kept in the program memory, and give visual results back to the user.

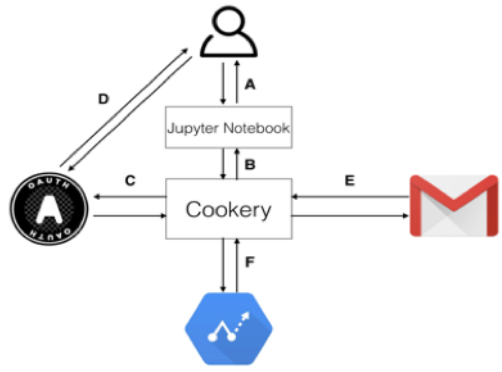


Figure 4. Finding life trends using data analysis in the Cookery data processing pipeline which is composed of two third-party services (Gmail, Google analytics)

The data pipeline can be broken into three steps:

1. The authentication step using OAuth protocol (Figure 4 – C): The first step is to register the application at Google in order to obtain the credentials. After the registration, it is possible to download the "client secret", which is stored on client side; this part is similar to the previous case using the GitHub monitor. Now that the application is known at the OAuth authentication server, we can start writing code to interact with the servers. In this case the application is limited to read emails and access the prediction API. The credentials are stored, so this verification needs to be done only once. Unless the scopes get changed, the credentials expire or the user retracts the permission.
2. The retrieval of individual emails from the server (Figure 4 – E): In this phase, it is possible to specify the characteristics of the request to read the emails, an example could be the label "INBOX" or "SENT". In the list of arguments of the request are the IDs of individual messages that were sent, which can be retrieved by their ID. For the purpose of this example, we are interested in the body of the email, the time and the date and the list of recipients, which are formatted as "Alias – Email address" and grouped by their address field (TO, CC, BCC).
3. Implement the machine learning API (Figure 4 – F): This was done in a similar way as with the Gmail API. We focused on the pre-trained models of the API. These models are used to analyze and predict the language of the emails. Google made the models based on their data set. A body of a message is sent to the servers, analyzed and returned to the Cookery system. The possible outcomes

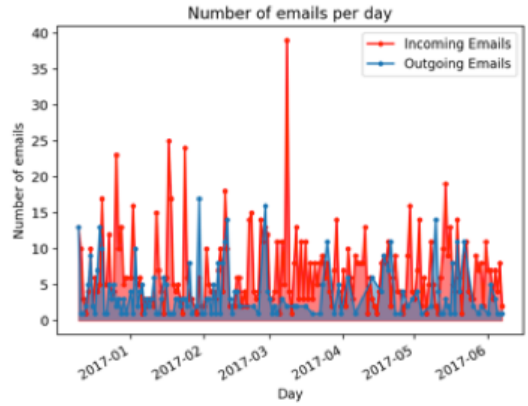
of the semantics analysis are positive, negative or neutral. The text of each email is categorized as either English, Spanish or French. The complete code implementing the data analysis workflow for the email analysis case is available in GitHub. The user can specify how many months of email they would like to analyze. All the incoming and outgoing emails are analyzed. For the incoming emails we are only looking at the date and time and the number of emails. For the outgoing emails we are interested in the date and time, the participants, number of emails, semantics and languages. All this information was stored as an object per email in a dictionary.

After Cookery finished analyzing emails, we visualized the dictionary of analyzed emails. The visualization was done outside Cookery using Matplotlib [13]. The plot is based on the last six month of author's personal emails which contain 470 outgoing and 1 200 incoming messages. The results based on the time stamp of the email and the number of emails are shown in Figure 5 a), and the results of the sentiment analysis are in Figure 5 b).

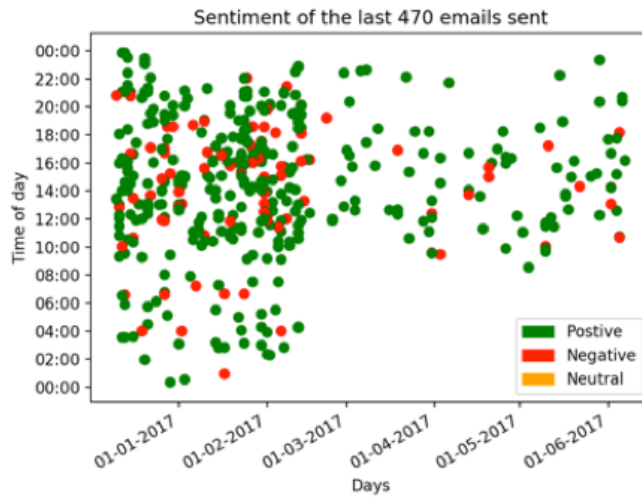
4 RELATED WORK

This research is focused on the combination of multiple systems; it has a lot of similarity with IFTTT (If This Then That) [14]. IFTTT is a web-based service that allows its users to create chains of conditional statements. These chains contain triggers from cloud services that can invoke actions in other cloud services. Thus, IFTTT can be used to automate web-application tasks. Examples are uploading photos to Dropbox that were received by email or automatically uploading the same content to multiple social media streams. IFTTT is focused on simplicity and therefore only supports one trigger action pair. Other IFTTT alternatives Zapier [16], and Microsoft Flow [17] differ from that approach and offer support for longer chains of trigger/actions pairs. Apart from offering longer chains, Zapier and Microsoft Flow use the same concept for workflow automation as IFTTT. Cookery differs from these services by its ability to implement any data transformation, because it can be extended with all the functionalities that Python has to offer.

In the article "Serverless Computation with OpenLambda" [18] the authors show that we have reached a new stage in the sharing model with FaaS. Technology has progressed from only sharing the hardware, which is done with virtual machines like VMWare, sharing the hardware and the operating system, as seen with containers like Docker, to sharing the runtime of a system. The handler is started in a container, which can only be used by the handler itself. Although multiple containers run in the same run-time, communication between containers is not possible. Other functions would then be able to intercept your functions and gain access to valuable information. On the other hand, a user will have to recognize some places where performance issues can arise. The readiness latency, the time it takes to start, restart or resume a container, can have consequences for the overall performance [18]. And there are more like the number of containers per memory (container density), pack-



a) Number of emails per day



b) Sentiment of the last 470 emails sent

Figure 5. Outcome of the sentiment analysis of the inbox of the email account of one of the authors

age support, cookies and sessions. A study has compared the cost, performance and response time of different implementation architectures such as monolithic architecture, micro-service architecture operated by the cloud customer and micro-service operated by AWS Lambda. With the micro-service architecture a developer will try to develop an application as a suite of small services [19], which all run their own process. The results of this study show that a micro-service operated by AWS Lambda is up to 77.08 % cheaper per million requests than the other two methods, while

the response times are faster than the cloud customer operated micro-service architecture and about the same as the monolithic architecture [20]. There are multiple online platforms that offer FaaS. Some examples are Google Cloud Function [21], Microsoft Azure, AWS Lambda and IBM OpenWhisk [22]. AWS Lambda is chosen for this project, because it is the only one to offer the service in combination with Python.

5 CONCLUSIONS AND FUTURE WORK

The goal of the Cookery framework is to enable scientists with little programming background to define a data analysis pipeline and execute it on online services provided by multiple cloud providers. To achieve this goal Cookery had to simplify process of the authentication and authorization, allowing the users to focus only on creating the data processing pipeline. In this paper, we described extension to the Cookery systems that allow for connections with cloud services using the OAuth 2.0 protocol. We used the "life trends" example to demonstrate how to create a data processing pipeline in Cookery using online services namely the Gmail service, and Google analytics. When developing the "life trends" example using cloud services, the only limitations we faced were related to the fact that we used a free version of the Google analytics, the machine learning API has a "User rate limit", which might have an impact on the performance of the time critical data processing pipeline. In the "life trends" example, we only made use of services from a single service provider namely Google.

Our next goal is to build data processing pipeline using service from different providers like AWS Amazon and Microsoft Azure. The first step toward this goal is described in the paper; we have developed an extension to use AWS Lambda services. In the future Cookery will be extended with more services from different cloud providers, to create a broader framework and to enable more developers to create applications. For example, an interesting extension would be with AWS DynamoDB [23] or AWS RDS (Relational Database Service) [24] to make it easier to create and manage databases using Cookery. When we combine databases with the functions of AWS Lambda, we can create more complicated applications and deploy them using Cookery. This also means that the toolkit of Cookery can be extended with more services and functionalities in future projects.

An important extension, we are currently investigating as a potential easy to develop light web application is the recently established Function-as-a-Service (FaaS). In the FaaS approach a user only runs a function on an external server. This is also called serverless computing, because you do not need a server for your application anymore. You basically have the function running on a server in the cloud. A user of the application provides the input and the function returns the output. This makes it possible for smaller businesses to develop their own application without buying servers. Developers also do not need a system administrator to maintain the servers, they do not need to write a complete infrastructure that can scale with the

demands of the applications and they do not need to handle all the administration. So basically, applications can scale up rapidly without needing to start new servers.

Inline with the FaaS concept we are looking at integrating this work with the idea of micro-infrastructures [29]. A micro-infrastructure is a dedicated network of application specific containers that are hosted on Kubernetes clusters. The containers expose functionality as FaaS so it is a natural coupling with Cookery. The added benefit of a micro-infrastructure is to define your own complex routines, package them in containers and expose them as FaaS. These can be placed closer to the data, in cases where the data is not on the public cloud but on HPC resources.

This can be developed further into a full-stack approach. An ecosystem of development tools would be used to capture isolated functionality at each level of the technical stack. The orchestration of the entire stack, composed of these isolated functionalities, can then be expressed using the DSL. Including, but not limited to, setting up (private) networks, data transfers, running compute routines in the cloud or on HPC clusters, and invoking remote web services. In the spirit of Cookery, each tool would provide a different abstraction level, and targets a specific level of the technical stack and/or user role. This will not only make the life easier of the scientists, operating at the highest level of the stack, but also supports efforts by engineers at the underlying levels of the stack. The low to non-existing coupling between isolated functionalities also facilitates reusability and maintainability. Moreover, functionality can easily be swapped out for a different version or an alternative implementation. To ensure the valid coherence of a application, composed of isolated functionality, the DSL can be extended with a type system. The type system will validate the application composition before it is executed, identifying interoperability issues at compile time. This prevents unnecessary costs, i.e. development time and resource usage expenses, caused by running invalid applications.

Acknowledgment

The authors thank Michael van Mill, Timo Dobber, and Dennis Kruidenberg – students at the University of Amsterdam who helped in implementing the three extensions of the Cookery framework presented in this paper.

REFERENCES

[1] Dropbox. `https://www.dropbox.com/`.

[2] Google Drive. `https://www.google.com/drive/`.

[3] Dropbox Statistics, Users, Growth and Facts for 2020. `https://saasscout.com/dropbox-statistics/`.

[4] Harvard Business Review Analytic Services. Cloud Computing Comes of Age. 2015.

[5] Ur, B.—Ho, M. P. Y.—Brawner, S.—Lee, J.—Mennicken, S.—Picard, N.—Schulze, D.—Littman, M. L.: Trigger-Action Programming in the Wild: An Analysis of 200 000 IFTTT Recipes. Proceedings of the 2016 CHI Conference on

Human Factors in Computing Systems (CHI '16), ACM, 2016, pp. 3227–3231, doi: 10.1145/2858036.2858556.

[6] Baranowski, M.—Belloum, A.—Bubak, M.: Cookery: A Framework for Developing Cloud Applications. Proceedings of IEEE International Conference on High Performance Computing and Simulation (HPCS), 2015, pp. 635–638, doi: 10.1109/HPCSim.2015.7237105.

[7] van Mill, M.: A Cookery Extension to Simplify Cloud Service Integrations. Bachelor Thesis. University of Amsterdam, 2017, https://esc.fnwi.uva.nlthesis/centraal/files/f704994072.pdf.

[8] Low-Level Clients. http://boto3.readthedocs.io/en/latest/guide/clients.html, visited on 04/20/2017.

[9] https://aws.amazon.com/s3/.

[10] Dobber, T.: Cookery in AWS Lambda. Bachelor Thesis. University of Amsterdam, 2017, https://esc.fnwi.uva.nl/thesis/centraal/files/f274795790.pdf.

[11] Kruidenberg, D.: Finding Life Trends Using Data Analysis in Cookery. Bachelor Thesis. University of Amsterdam, 2017, https://esc.fnwi.uva.nl/thesis/centraal/files/f628826658.pdf.

[12] Wolfram, S.: The Personal Analytics of My Life. 2012, https://writings.stephenwolfram.com/2012/03/the-personal-analytics-of-my-life/.

[13] Python Package Matplotlib. https://matplotlib.org.

[14] Yang, F.—Manoharan, S.: A Security Analysis of the OAuth Protocol. 2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM), IEEE, 2013, pp. 271–276, doi: 10.1109/PACRIM.2013.6625487.

[15] Lodderstedt, T.—McGloin, M.—Hunt, P.: OAuth 2.0 Threat Model and Security Considerations. Internet Engineering Task Force (IETF), 2013.

[16] https://zapier.com/developer/documentation/v2/#what-is-zapier.

[17] Getting Started with Microsoft Flow. https://www.windowscentral.com/getting-started-microsoft-flow.

[18] Hendrickson, S.—Sturdevant, S.—Harter, T.—Venkataramani, V.—Arpaci-Dusseau, A. C.—Arpaci-Dusseau, R. H.: Serverless Computation with OpenLambda. Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing (HotCloud '16), 2016, pp. 33–39.

[19] Fowler, M.—Lewis, J.: Microservices: A Definition of This New Architectural Term. 2014, https://martinfowler.com/articles/microservices.html.

[20] Villamizar, M.—Garcés, O.—Ochoa, L.—Castro, H.—Salamanca, L.—Verano, M.—Casallas, R.—Gil, S.—Valencia, C.—Zambrano, A.—Lang, M.: Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures. 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), 2016, pp. 179–182, doi: 10.1109/CCGrid.2016.37.

[21] Google Cloud Functions. https://Cloud.google.com/functions/.

[22] IBM OpenWhisk. https://developer.ibm.com/openwhisk/.

[23] AWS DynamoDB. https://aws.amazon.com/dynamodb/.

[24] AWS RDS. https://aws.amazon.com/rds/.

[25] IFTTT: About IFTTT. https://ifttt.com/about.

[26] Making Simple Python Wrapper Kernels. http://ipython.readthedocs.io/en/stable/development/wrapperkernels.html.

[27] Google Cloud Functions Documentation. 2017, accessed: 11-Dec-2017, available at: https://Cloud.google.com/functions/docs/.

[28] Microsoft Azure: Azure Functions. 2017, accessed: 11-Dec-2017, available at: https://azure.microsoft.com/en-us/services/functions/.

[29] Cushing, R.—Valkering, O.—Belloum, A.—de Laat, C.: Towards a New Paradigm for Programming Scientific Workflows. 2019 15th International Conference on eScience (eScience), San Diego, USA, 2019, pp. 604–608, doi: 10.1109/eScience.2019.00083.

[30] https://github.com/mikolajb/cookery.

[31] https://github.com/mikolajb/cookery/tree/master/cookery.

Mikołaj Baranowski is Ph.D. student at the University of Amsterdam in the Multiscale Networked Systems (MNS) group. His research fields are new paradigms in computer language abstractions.

Adam Belloum is Senior Researcher at the Computer Science Department of the University of Amsterdam and the technology lead working on optimized data handling at Dutch National eScience Center. He received his M.Sc. and Ph.D. degrees from the Compiegne University of Technology, France.

Reginald Cushing is PostDoc at the University of Amsterdam in the Multiscale Networked Systems (MNS) group. His research fields are in distributed systems with a focus and data processing, federation, and scientific workflows.

Onno Valkering is Scientific Programmer in the Multiscale Networked Systems (MNS) research group, at the University of Amsterdam (UvA). His interests are distributed data processing, domain-specific languages, and privacy-preserving techniques.