Model-based Analysis of Serverless Applications

Stefan Winzinger
Distributed Systems Group
University of Bamberg
Bamberg, Germany
stefan.winzinger@uni-bamberg.de

Guido Wirtz
Distributed Systems Group
University of Bamberg
Bamberg, Germany
guido.wirtz@uni-bamberg.de

***Abstract*—Serverless computing is a relatively new execution model where the cloud platform provider manages the allocation of resources for containerized functions dynamically. This evolving paradigm is called Function as a Service (FaaS). The statelessness of these functions enables the application to be scaled up elastically in the case of peak loads. They can be tested easily in isolation, but the behavior arising by integrating them to an application is both hard to predict and test. The parallel execution of the functions and the shift of its state to data storages can cause several workflows accessing the same data. These workflows are hard to detect, particularly for complex applications. Therefore, we suggest an approach for modelling an existing serverless application based on a specialized graph holding all relevant features.**

Our serverless-specific model can be applied during the whole life cycle of a complex application and offers a good basis for this specific class of applications. It helps to optimize an existing system by identifying hot spots, supports the generation of test cases and can be used to monitor an existing system. Furthermore, we show how the generation of the model can be automated by realizing a tool supporting Amazon's AWS Lambda.

***Index Terms*—serverless computing, FaaS, dependency graph, model-driven testing, integration testing, cloud functions**

I. Introduction

Cloud computing becomes more and more popular. This affects also serverless computing which is now offered by many big companies like Amazon [1], Google [2], IBM [3] and Microsoft [4]. But there are also open source projects like OpenLambda [5] or OpenFaaS [6]. The term serverless computing describes a programming model and architecture where small code snippets are executed in the cloud without any control over the resource on which the code runs [7]. These code snippets are usually functions written in languages like Java, JavaScript, C# or Python and can call services like functions, data storages or other platform-specific services.

Transferring an application to a serverless architecture can reduce hosting costs significantly [8] and improve the performance [9]. This is caused by the usage of stateless, mostly short-running functions, so-called serverless functions, whose degree of resource concurrency can be adjusted automatically on function level and not on application level [10]. Serverless functions have only to be paid for their execution time actually used [11] which is ideal for applications with an irregular workload.

Characteristic for a serverless application are numerous transient, short-lived, concurrent functions whose complexity requires tools which can be used to develop such complex applications [12]. Whereas the behavior of functions in isolation can be tested quite easily with unit tests, testing the behavior of a serverless application consisting of lots of serverless functions is more difficult. Because of the multitude of services involved and potential parallel executions, behavior emerges which is hard to predict if functions are tested in isolation. Therefore, a model is needed which helps visualize and analyze a serverless application. This model should enable the developer to detect hot spots and create test cases.

Many cloud modeling languages are already available [13] though there is no general model available focusing on the specific class of serverless applications and its characteristics during runtime. AWS SAM [14] is a model specialized in the definition of a serverless application on AWS which focuses mainly on the deployment. However, this model considers neither runtime aspects nor the parallelism introduced by a serverless application. A dependency graph is also used to analyze a system in [15] whereby it focuses on microservices and not on specific characteristics of a serverless application like its statelessness of functions.

Graph-based analyses are widely used, like for the analysis of microservices [15], the analysis of the evolution of software systems [16], or the analysis of event-driven applications [17]. However, the graphs used there are not applicable for serverless applications since they focus on different aspects of the system.

The contribution of this paper is the introduction of an approach to create a dependency graph from a serverless application as the basis for an advanced tool that helps developers to analyze, visualize and track the dependencies between serverless functions, data storages and other resources deployed in several stages of its life cycle. By adding relevant characteristics to the graph, developers can more easily analyze the application and detect additional errors during integration testing. Because of the complexity of the system, these errors often emerge the first time during the integration of the resources.

The paper is structured as follows. Section II introduces the basic dependency graph and possible extensions increasing its power. We discuss why the annotations are useful and how these annotations can be used for the analysis and testing of

the application. An approach for the creation of a model of an existing application is described in Section III. The feasibility of our approach is demonstrated in Section IV by applying a tool to a serverless application which consists of Amazon's AWS Lambdas and other platform-specific services. Section V gives an outlook to future work, whereas in Section VI a conclusion is drawn.

II. Definition of Serverless Dependency Graph

The core resource in a serverless application are serverless functions which are managed by the cloud provider. In this section, we show how a basic dependency graph can be constructed by focusing only on serverless functions. However, not only serverless functions are usually used to set up an application but also other resources. These resources are needed to store the data of the application, process and dispatch data and orchestrate serverless functions. We show how these resources can be used in accordance with our graph by either ignoring them or by adding them to the graph iff suitable. Adding these resources makes our graph more powerful which is useful for a deeper analysis of the application. Annotations which can be added to the graph and its extensions are discussed at the end of this section.

A. Basic serverless dependency graph

Serverless functions are the central resource of a serverless application which need to be connected with each other in order to get a running application. This can be done by invoking one function directly from another function via an event. But other resources can also connect serverless functions indirectly by processing an event of a serverless function and finally triggering another serverless function. So, if a serverless function uses a resource, this resource can trigger another serverless function (or another resource) by creating an event and thus connects these serverless functions indirectly.

Therefore, we define our basic graph G as a set of nodes N and a set of ordered pairs A which are ordered pairs of elements of N. In our model, a node $n \in N$ represents a serverless function, whereas an arc $(n_1, n_2) \in A$ represents the invocation of a serverless function $n_2 \in N$ from a serverless function $n_1 \in N$.

As a result, the structure of an application which only consists of serverless functions can be described by using this basic dependency graph. The basic dependency graph of a message pattern, which is used if a message should be passed to several serverless functions, is shown as an example in Fig. 1. The graph gives basic insights into the application and a compact overview of the functions being used and their interplay. But the graph can also be used for error detection and test case generation. A recursive call of functions would result in a cyclic graph. Therefore, identified cycles have to be analyzed in order to avoid endless, recursive function calls and their resulting enormous costs.

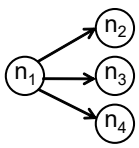

Fig. 1. Example of a basic graph representing the messaging pattern

But the graph can also be used for a general identification of workflows. If only asynchronous invocations are made, a simple depth-first search can be used to identify all paths of an acyclic graph and be used as a starting point for a coverage analysis of test cases. Of course, not all paths of this graph, be it statically or dynamically constructed, are feasible. Some paths can even be covered by a single test case, since some paths of the model can be executed in parallel if a function calls several functions in parallel and thus creates several parallel workflows.

B. Mapping of resources to the model

A serverless application usually consists not only of serverless functions but also of other resources. The statelessness of the serverless functions enables the cloud provider the dynamic parallelization of the function calls. Since the state cannot be saved within a function, the state of the application has to be saved in a data storage. But there are also often resources dispatching or processing data, orchestrating serverless functions or offering an interface to the application. Most of these resources can also trigger serverless functions. In the following, we will show that the application can be modelled as a basic graph if it contains some of these resources by either ignoring them or extending our basic graph by adding them.

1) Data storage: A serverless function does not save states over several calls. For this reason, it has to save its state to data storages which are usually databases if persistency of data is needed. Since most applications use at least one data storage, we integrate data storages into our model. If a data storage has to be modelled in our dependency graph, the graph G has to be extended by using an additional set of nodes D where a node $d \in D$ represents a data storage. The set of arcs A has to be extended by also allowing arcs (d, n) and (n, d) where $n \in N$ and $d \in D$. An arc (n, d) is used to indicate that something is written to or read from a data storage by a serverless function, whereas an arc (d, n) represents a trigger of a data storage which creates an event handled by a serverless function if something in a data storage was changed (e.g., an entry was updated, added or removed).

If only the basic dependency graph is needed, the data storages can be omitted. This can be done by adding dependencies between serverless functions if these functions are connected via a data storage (e.g., (n_1, d_1) and (d_1, n_2) where $n_1, n_2 \in N$ and $d_1 \in D$ results in an arc (n_1, n_2)) (cp. Fig. 2).

2) General approach for other resources: The same approach can be utilized for other resources independent of the platform. If a certain resource needs to be modelled, the graph can simply be extended by adding a new set of nodes representing the corresponding type of resource. The

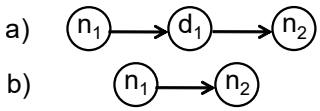

Fig. 2. Example of a graph showing a data storage (a) and its counterpart as a basic graph (b)

outcoming and incoming arcs are connected to the serverless functions invoking the resource respectively triggered by an event which was created by the corresponding resource.

However, if there are resources orchestrating the workflow of serverless functions like in Amazon's "AWS Step Functions" [18], it can be advisable not to omit all the serverless functions if only the basic model is intended to be used. Instead, all serverless functions being part of this resource can be modelled in the dependency graph representing their potential order of invocations.

Thus, our basic dependency graph can be easily extended if some resources need to be modelled. This gives new insights into the structure of the application and enables the detection and creation of architectural patterns. The additional nodes enable the creation of new test cases. Nodes and their relations can be used as a basis for coverage criteria, e.g., requiring the coverage of all nodes or all node-pairs [19], and seldom called scenarios can be identified.

C. Characteristic settings of a serverless application

Having a dependency graph and the ability to add any resource of the application to our model is not always enough to analyze the application and detect errors. Therefore, we discuss in this section some additional useful settings of the application which can be relevant for the analysis of a serverless application.

1) Read/Write access: Data storages are a central resource in a serverless application to save the state of the application. But the usage of data storages causes also errors. In a serverless application, serverless functions can be executed in parallel by the cloud provider. This can be done by invoking a function several times in parallel in one application call or by parallel calls of the application. Both results in parallel workflows. However, if workflows running in parallel access the same data and at least one of the workflow writes data, the state of the application can deviate from its intended behavior depending on the order of accesses to the data storages.

Our data storage-extended graph $G = (N, D, A)$ can add an additional information of the set $I = \{r, w, rw\}$ to arcs (n, d) where $n \in N$ and $d \in D$. This indicates if a read, write or a read/write access is executed.

This enables the developers to detect problematic scenarios like a potential race condition shown in Fig. 3 where two workflows access the same data storage. If both workflows access the same data, the data read by n_{22}, and thus the result of its workflow, depends on whether n_{12} writes its data before or after the access of n_{22}. Since calls to a serverless application can be started simultaneously several times, even write and

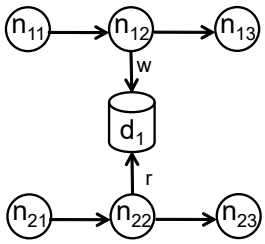

Fig. 3. Potential race condition in a graph with two workflows

read accesses of different instances of the same workflow (see Fig. 4) can be critical. Different workflows could write and read the same data and thus provoke a race condition. But also a race condition of the same workflow can occur if n_{12} is asynchronously called before the write access from n_{11} to d_1. By using such a notation, it is easier to detect hot spots and create test cases verifying that read and write access work in any order.

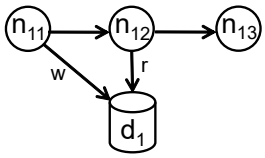

Fig. 4. Potential race condition in graph with one workflow

Not all accesses to data storages use the same data. Therefore, an annotation of the arcs adding a constraint to the data storage accesses supports the identification of the relevant accesses to data storages where race conditions might occur. If a key of a key-value database or a selection of a relational database is set as constraint, intersections of accesses can be easier detected. However, if the values constraining the access to the data storage can change depending on input parameters of the function, it is hard to detect intersections. Therefore, instead of using this annotation, a worst-case assumption has to be used.

Even more annotations for databases are applicable. E.g, an annotation of the consistency model of the database can help to detect problems in the database. If a database offers only a restricted consistency, the hot spots and their relations have to be handled more carefully.

2) Synchronous invocations: Serverless functions are invoked either asynchronously or synchronously. Asynchronous calls are based on an event-driven architectural procedure where an event is created and then is handled by another resource. This results in an independent execution of the former code having created the call. On the other side, in synchronous invocations, the caller waits until the callee is finished. In the specification of our basic graph both invocations are modelled.

However, if a synchronous invocation is made, the caller is more vulnerable for timeouts since its execution time depends on the callee whose answer the caller waits for.

But billing is also a problem since the caller is running while waiting for a response which results in double-billing [10]. Therefore, synchronous function calls are usually not the best choice for the architecture of a serverless application and are consequently worth being highlighted. Accordingly, synchronous function calls are highlighted by extending the arcs initiating such a call with an element $sync$.

Only asynchronous calls can increase the number of workflows. Therefore, the number of potential workflows which can be derived by a model can be reduced. If the focus of the analysis is more on the identification of workflows, synchronous calls can also be omitted and replaced their following asynchronous calls similar to the general approach of mapping resources to the graph.

3) Order of calls: If the order of calls to other resources is unknown, more cases than necessary can be identified as critical. As already shown in Fig. 4, potential race condition can be detected if the order of calls to resources accessing the same data storage is unclear. However, if the order of calls is known and a synchronous call is made before an asynchronous call, race conditions can be excluded for some cases. If in our example n_{11} writes its data synchronously to d_1 before n_{12}, no race condition occurs for the same workflow as long as only one instance of the workflow is used. Therefore, by annotating the order to each outcoming arc of a node facilitates the elimination of infeasible workflows.

4) Conditions of calls: The dependency graph enables the creation of potential workflows. However, the number of generated workflows can be very big because also infeasible workflows are generated. The information of calling another resource under a certain condition is not available in our basic graph. Therefore, the worst-case assumption in a node has to be used that all related resources can be called if the node is called. This can be prevented by adding constraints to invocations if they are made under a certain condition. This reduces the number of infeasible workflows generated by using the model and also helps identify workflows ending in the node since their conditions are not fulfilled.

5) Multiple asynchronous calls: In order to analyze for parallel workflows or bottlenecks, dependencies are interesting where a method is not called only once asynchronously but several times. By annotating the number of calls to an arc made by the caller to the callee, parallel workflows can be identified more easily. Additionally, this enables the analysis of potential bottlenecks where the number of parallel calls can be throttled due to server limits or an increase of execution time could be expected due to the heavy demand of parallel containers.

6) Access rights: In a serverless application each resource has access rights assigned describing the rights it has to use other resources. By assigning only the rights needed for each resource, unnecessary security breaches can be prevented. The dependency graph gives a good insight in the rights being needed for a function. Therefore, if access rights are displayed with its nodes, nodes with missing or unnecessary rights can be identified and the security of the application be improved.

7) Platform settings: Some other resource-specific settings made to the platform can also be assigned to the nodes of our graph. The timeout limit indicating the maximum time a function is allowed to run before an error is thrown can be set. The information about the assigned hardware to a function, like memory and computational power, can also be stored into a node. However, this is only relevant if data are available giving information about the profile of a running application as described in the following.

D. Runtime data

The previous section has shown how a serverless application can be modelled and which static attributes can be added to the model to support the analysis. This section shows that also some runtime information which can be collected during test runs or production runs can be used in accordance with our model, and thus support the detection of errors.

1) Time settings: If data are available providing runtime information, also usage statistics can be annotated to the nodes indicating the time a function or another resource took to be executed. If workflows are generated by using the model, their runtime can be estimated more easily by using the annotated time. This enables a better estimation of runtimes and the identification of long-running tasks. The cold start time [20] needed could also be added to enable a more intensive analysis.

By comparing these values to the timeout limits of the functions, a risk analysis can be made making the application more reliable.

2) Hardware settings: Similar to the annotation of time settings, a serverless function can get the usage statistics of its hardware needed for its executions. This helps identify functions with a huge demand for hardware, like memory or CPU power, and might be worth improving. If the actual assigned hardware setting is assigned to the node as described in the previous section, the model can be evaluated to identify critical nodes which might be prone to an exception of not having enough memory or CPU available. Also nodes having too much hardware capabilities assigned can be identified which can help save money if performance is not an issue.

3) Billing: Having functions which are only paid if they are actually used is a key characteristic of serverless computing. Therefore, in order to get an impression of the costs and to have the possibility to improve them, the costs of runs can be assigned to nodes. This helps to identify expensive functions. This information correlates with the data of time taken for a function execution and the hardware assigned to it.

4) Call history: the call history of the resources describes the number of calls or even the call chains [12]. This makes it possible to create a usage profile and identify more efficiently spots where improvement can be made or resources which are seldom used.

III. Creation of Model

We defined a basic dependency graph specialized for serverless applications and identified several information in the

previous section which can be modeled depending on the the aim of the analysis. Some of this information can be created automatically by using the source code of the application, some can be added manually, whereas other information needs to be generated during runtime. The latter one is the basis of the model for monitoring an application. In the following, we describe how a basic model can be generated and how its annotations and monitoring information can be added to the graph.

A. Structural model

1) Basic model: The basic model can be created by reading a file describing the infrastructure of the application (e.g., Amazon's AWS CloudFormation). This infrastructure file has to contain all the resources which are deployed. Additionally, the source code of the serverless functions is needed. If the infrastructure file or the source code is not available, the model can still be created manually by making worst-case assumptions.

All the resources being used in the application can be identified by parsing the infrastructure file. Even some of the relationships can be identified, e.g., if a database triggers a serverless function.

In order to get the invocations made by serverless functions, their source code must be analyzed. This can be done by using a parser identifying the relevant code snippets and has to be done for all serverless functions. However, it is hard to decipher an invocation call if it is not hard-coded and even impossible to get the invocation call if it depends on a parameter given to the function. If so, possible functions have to be added manually. If the source code of the functions is not available, the relations of the functions have to be constructed manually.

A graph containing all resources and their relations can be constructed first if the application contains other resources than serverless functions. Afterwards, all other unwanted resources can be removed and their predecessors and successors be connected.

2) Data storage: Data storages being used should already be defined in the file describing the infrastructure. The invocations made to data stores, as read and write accesses, can be identified by analyzing the source code of the serverless functions. However, like for the identification of serverless function calls, if the parameters are not hard-coded, it cannot be created automatically. Hence, manual annotations can be useful. Constraints can also be identified by analyzing the source code of serverless functions with the same drawback.

Invocations made by data storages can be read in the infrastructure file where the configuration of the data store is defined.

3) General resources: Depending on the resource, its dependencies can be generated by the infrastructure file of the serverless application. The dependencies of gateways, queues etc. are usually described there. The structure of a state graph, like used in Amazon's step functions, is also described in the infrastructure file. By analyzing the states of the state charts, the dependencies of the serverless functions used within the state chart can be constructed. If the connection of a resource to another resource or a serverless function is not described in the infrastructure file, this information has to be saved somewhere else which can then be used for the generation of the dependency.

B. Annotations of the model

Not only the structure of the model but also the annotation of the model can mostly be created automatically which is discussed here.

Some settings of the platform can simply be read from the infrastructure file. Usually, if they are not set, the standard values of the platform provider are used which have to be known. Typical settings are access rights, timeouts, throttling limits but also, depending on the platform and resource, synchronous and asynchronous calls can be read from the infrastructure file.

Some other settings have to be read from the source code files of the serverless functions. The order of calls, conditions and the identification of multiple calls has to be constructed by parsing the source code file. If external factors influence the control flow or if the structure of the source code is hard to parse, it is quite difficult to calculate the orders correctly. Usually, it is easy to identify if a call is made synchronously or asynchronously by analyzing the command doing the invocation.

However, the quality of all annotations which can be created automatically depends strongly on the quality of the parser interpreting the source code and has to be set manually if needed.

C. Monitoring

There are several metrics which can be monitored and assigned to the nodes and arcs of our model. In order to assign the runtime data to the nodes, a mapping of the resources of our model to the runtime data constructed of the application is needed. If such a mapping is done, the runtime information, e.g., log information, can be evaluated and the relevant information be assigned to the corresponding node or arc of the graph. If, for example, a node is mapped to a certain log stream, the log stream can be analyzed and its runtimes, number of invocations and costs be saved in the node and be evaluated dynamically in the context of the graph.

D. Dynamic generation of graph

The same kind of model can also be generated by evaluating the log data produced by test cases or during production. This leads to a dynamic generation of the model where different approaches are applicable. Instead of creating the graph statically, the whole graph can be created dynamically. This requires an instrumentation of the application which produces meaningful enough log data to capture the resources of the application, their relations and annotations. Additionally, the application has to be executed with a profile covering as many cases as possible to create the log data at all. However,

it is difficult to create enough cases which cover all resources, relations and annotations which are needed to produce the whole graph. Therefore, some relevant information can easily be missed.

A dynamic approach can also be combined with the static one. The graph can be created statically and be expanded by relations and annotations which are hard or impossible to create statically. Additionally, worst-case assumptions can be reduced by evaluating the application dynamically if enough cases are available for the corresponding component.

IV. Proof of concept

In order to show the feasibility of our approach we wrote a tool[1] being able to construct our model of an application run on AWS by using lambda functions which is the implementation of serverless functions for this platform. We decided to support serverless functions and Amazon's NoSQL database service "DynamoDB" as services, and write/read accesses, order of calls and synchronous calls as annotation in order to demonstrate their benefits. The application we evaluate consists of a data storage which triggers a lambda function as soon as a file is added to the data storage. The lambda function has to compress the uploaded file and save it to the data storage. Additionally, hash values of the original file and of the new file are created. These values are saved and sent to the user. The lambda functions were written in Java for AWS and use a S3 data store and a DynamoDB database.

A. Model creation tool

Our tool creates the graph with its relations statically by using the infrastructure file and the source code files of the serverless functions. By reading the AWS SAM file describing the infrastructure of AWS, our basic model is created. This model is expanded by additional relations which can be read from the source code files of the functions.

A parser for the source code files is applied which is able to identify the relations of the resources and their annotations. If more complex information about the invocations is required, a more sophisticated parser being able to identify them is needed. Our parser supports some basic detection mechanisms for java files having an adapted structure. After this, a basic model is available. This model can be expanded by statistic data which can be read from a log file. If a simpler version of the graph is needed, this can be done afterwards by removing the unnecessary nodes and annotations. The concept of the tool can be transferred to other platforms by adapting the parser for the input files for the corresponding platform.

B. Evaluation of example application

In this section, the graph generated by our model creation tool for our test application is evaluated. A visual representation of the generated graph of this application can be seen in Fig. 5. The visual representation of the AWS SAM file constructed by Amazon's AWS designer is depicted in Fig. 6. The graph has less information and does not represent potential

[1]https://github.com/snwinz/ServerlessApplicationTool

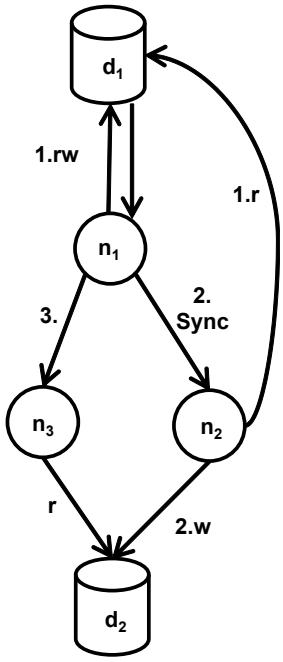

Fig. 5. Basic graph with some annotation of application to be modelled

runtime behavior between the components since this file is mainly used for deployment and does not evaluate the source code of the serverless functions. Even when our example

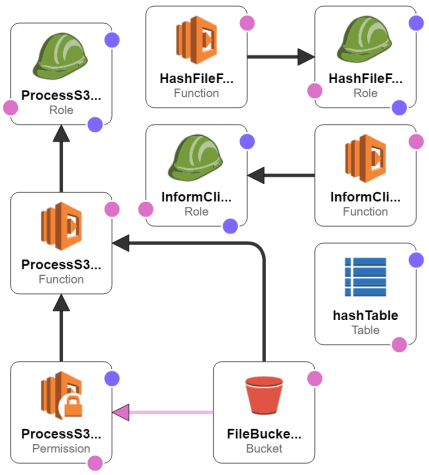

Fig. 6. Basic graph with some annotation of application to be modelled

application is very small, there are several insights which can be detected by analyzing our graph which are harder to detect otherwise:

- A subgraph of nodes n_1 and d_1 builds a cyclic graph which could create an endless recursive call. So, if a file is added to the data storage d_1, n_1 is informed, reads the data, and uploads the compressed data again to the data storage. This would create another event handled by n_1 and so on. Therefore, settings have to be made for the

data storage that n_1 is only triggered for a certain kind of files or n_1 has to ignore compressed files.
- Another cycle can be found in the subgraph d_1, n_1 and n_2. Data of d_1 is read by n_2. However, this is not a problem since d_1 does not trigger for read accesses for S3 data storages on this platform.
- Both n_1 and n_2 access d_1 whereas n_1 also writes data. However, since the calls made by n_1 are ordered, we see that the read access of n_2 is always after n_1 since the write access of n_1 is done synchronously (not shown in model since this is the default for data stores operations).
- n_1 reads and writes to a data store. Since there is no order given for this operation, it has to be ensured that these operations either are done in a fixed order or use different data and the application behaves correctly if n_1 is called in parallel.
- d_1 is read and written in the application. It has to be ensured that the data are consistent if the application triggers n_1 in parallel after several files were uploaded at once.
- d_2 is written by n_2 and read by n_3. There is no race condition since n_2 is called synchronously by n_1. However, it has to be checked that the application behaves correctly if it is called in parallel.
- n_1 makes a synchronous call to n_2. Since n_1 has to wait for an answer (which can take some time since n_2 makes some data storage accesses), it has to be paid unnecessarily. Therefore, depending on its usage profile, it might be worth restructuring the application.

The example showed that it is possible to create the graph to some degree automatically and that the model introduced helps identify hot spots.The example shows the most important typical flaws and potential problems that can be identified by creating and analyzing the graph for even such a simple application.

V. Future work

For our future work we plan to extend our tool by adding an automatic analysis of the graph which can be applied for real life applications. Additionally, an automatic test case generation based on the model is planned fulfilling coverage criteria tailored to the characteristics of serverless applications. Furthermore, the application will be adapted to support a dynamic graph creation and a live analysis of an application.

VI. Conclusion

We introduced an approach for the creation of a model which is designed for serverless applications supporting the analysis of the application. Furthermore, we showed how the graph can be created automatically.

The annotations which can be added depending on the purpose of the analysis focus on the characteristics of serverless applications and enable the detection of several hot spots.

Our serverless-specific model is a good foundation for the specific class of serverless applications and can be applied during the whole life cycle of complex application. It helps to optimize an existing system by identifying hot spots, supports the generation of test cases and can be used to monitor an existing system.

By using our approach cloud providers could support developers in several development phases and help them produce systems with a better quality.

References

[1] "AWS Lambda," accessed January 30, 2019. [Online]. Available: https://aws.amazon.com/lambda/

[2] "Google Cloud Functions," accessed January 30, 2019. [Online]. Available: https://cloud.google.com/functions/

[3] "IBM Cloud Functions," accessed January 30, 2019. [Online]. Available: https://www.ibm.com/cloud/functions/

[4] "Azure Functions," accessed January 30, 2019. [Online]. Available: https://azure.microsoft.com/en-us/services/functions/

[5] "OpenLambda," accessed January 30, 2019. [Online]. Available: https://github.com/open-lambda

[6] "OpenFaaS," accessed January 30, 2019. [Online]. Available: https://github.com/openfaas

[7] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter, "Serverless computing: Current trends and open problems," in *Research Advances in Cloud Computing*. Springer Singapore, 2017, pp. 1–20.

[8] G. Adzic and R. Chatley, "Serverless computing: economic and architectural impact," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017*. ACM Press, 2017.

[9] G. McGrath and P. R. Brenner, "Serverless computing: Design, implementation, and performance," in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE, Jun. 2017.

[10] I. Baldini, P. Cheng, S. J. Fink, N. Mitchell, V. Muthusamy, R. Rabbah, P. Suter, and O. Tardieu, "The serverless trilemma: function composition for serverless computing," in *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software - Onward! 2017*. ACM Press, 2017.

[11] E. van Eyk, A. Iosup, S. Seif, and M. Thömmes, "The SPEC cloud group's research vision on FaaS and serverless architectures," in *Proceedings of the 2nd International Workshop on Serverless Computing - WoSC '17*. ACM Press, 2017.

[12] W.-T. Lin, C. Krintz, and R. Wolski, "Tracing function dependencies across clouds," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, Jul. 2018.

[13] A. Bergmayr, M. Wimmer, G. Kappel, and M. Grossniklaus, "Cloud modeling languages by example," in *2014 IEEE 7th International Conference on Service-Oriented Computing and Applications*. IEEE, Nov. 2014.

[14] "AWS SAM," accessed January 30, 2019. [Online]. Available: https://github.com/awslabs/serverless-application-model

[15] S.-P. Ma, C.-Y. Fan, Y. Chuang, W.-T. Lee, S.-J. Lee, and N.-L. Hsueh, "Using service dependency graph to analyze and test microservices," in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*. IEEE, Jul. 2018.

[16] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos, "Graph-based analysis and prediction for software evolution," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, jun 2012.

[17] M. Madsen, F. Tip, and O. Lhoták, "Static analysis of event-driven node.js JavaScript applications," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications - OOPSLA 2015*. ACM Press, 2015.

[18] "AWS Step Functions," accessed January 30, 2019. [Online]. Available: https://aws.amazon.com/step-functions/

[19] U. Linnenkugel and M. Mullerburg, "Test data selection criteria for (software) integration testing," in *Systems Integration '90. Proceedings of the First International Conference on Systems Integration*. IEEE Comput. Soc. Press, 1990.

[20] J. Manner, M. Endreß, T. Heckel, and G. Wirtz, "Cold start influencing factors in function as a service," in *Fourth International Workshop on Serverless Computing (WoSC4)*, Zurich, Switzerland, 2018.