Metamorphic Testing and Serverless Computing: A Basic Architecture

Yakiv Yusyn* and Tetiana Zabolotnia
E-mail: yusin.yakiv@gmail.com and tetiana.zabolotnia@gmail.com
Computer Systems Software Department
National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", Kyiv, 03056, Ukraine

Keywords: cloud computing, serverless computing, metamorphic testing, cloud architecture, Azure

Received: May 16, 2022

Automated testing of complex software systems can be a challenging task, and today there are a large number of methods for its implementation. One such method is metamorphic testing, which effectively solves the problems of usual methods and is gaining popularity. However, performing metamorphic tests can take a long time, so the question arises of their distributed running, including in the cloud. Thus, the authors of this study considered the designing of a cloud serverless architecture of software for metamorphic testing. The serverless architecture for metamorphic testing is proposed, which is based on the composition of the entire system from 5 individual components: models, data generator, software artifact under test, metamorphic relations, and serverless functions. For each of the main possible types of software artifacts, the possibility of using the serverless architecture for metamorphic testing is considered. The developed architecture is presented in the form of component, deployment, and sequence diagrams. The use of the proposed architecture in practice is shown by the example of testing two software artifacts – a class library and a web application. Performance measurements have shown that despite the additional network delay when running one test, the performance of all tests in general in the case of the serverless architecture is closer to local startup and will be faster with increasing complexity and number of tests.

Povzetek: Predstavljena je arhitektura brez strežnika za metamorfno testiranje zapletene programske opreme.

1 Introduction

Quality assurance of developed software products using automated testing methods has been and remains an urgent task for the IT sector.

The most widely used automated software testing method is the oracle-based tests that consist in a comparison of the obtained output data against the expected ones (for the specified input data) [1]. In practice, the problem of finding and determining the oracle for the software module under test is widespread and is called the "oracle problem" [2]. Most often, it is due to two causes: first, a complex logic of the software artifact to be tested that leads to difficulties with expected result identification by a human, and second, a huge capacity of the scope of possible internal states of the artifact and possible values that input parameters can get – as a result, the tests using oracles will cover only some subsets of such scope.

Metamorphic testing is one method that effectively solves the oracle problem [3]. Unlike comparing the obtained result with the expected one, the method is based on the idea of using metamorphic relations – certain relations between the input and output data characteristic for the given domain area [3]. Metamorphic relation describes how the output data should change when specific input data change. For instance, the following relation may serve as metamorphic relation for the multiplication function: if one of the multipliers is increased twice (change of input data), the result will also increase twice (output data change). As one can see, specific input or output data are not considered in this relation, thus solving the oracle problem. Accordingly, metamorphic relations form the basis of metamorphic tests that check whether the software under test fulfills such relations.

Metamorphic testing is already used successfully in some sectors, but the development of the basic architecture options for its software implementation and standardized tools for its application remains urgent. One of the tasks that we could highlight here is combining metamorphic testing with cloud and distributed computing technology to accelerate tests execution. It is due to the fact that this methodology is located in the testing pyramid closer to the integration and end-to-end tests [4], so local runs of a large number of metamorphic tests may take

* Corresponding author

Table 1: The summary table of tools from overviewed research.

Work	Type of artifact under test	MRs count	Type of run	Automated or not	Additional notes
5	Web application	4	Local	Automated	
6	Web application	4	Local	Automated	Written in JS
7	Web application	5	Local	Automated	Running hourly; some MRs achieved performance in 3000 inputs per hour
8	Desktop application	1	Local	Automated	Written in C
9	Desktop application	1	Local	Automated	Written in C
10	Desktop application	5	Local	Automated	Written in C
11	Library	6	Local	Automated	Written in MATLAB
12	Desktop application	14	Local	Manual	
13	Desktop application	3	Local	Manual	
16	Desktop application	5	Local Cloud VMs	Automated Automated	Total run time – 35 hours Total run time – 5.5 hours

much time. However, the performance of such tests can be distributed owing to their independent nature.

Thus, the purpose of this work is metamorphic testing software improvement as a whole by developing its cloud serverless architecture that would improve the obtained results in terms of the test execution speed.

In Section 2, related works are shortly described, including the use case of cloud system for metamorphic testing of bioinformatic pipeline. In Section 3, an overview of serverless computing is given, and all remaining sections are dedicated to the experiment and its results, including conclusions.

2 Related works

The idea of metamorphic testing was proposed in [3] for the first time, and since that time used successfully in several various sectors, for instance, in the development of web applications [5, 6, 7], compilers [8, 9], computer graphics [10, 11] and bioinformatics [12, 13] applications, so on.

The first meta-review of the papers involving metamorphic testing methodology is provided in [14] and expanded in [15]. Besides the description of the current status of metamorphic testing practical application, the second meta-review also describes challenges and open issues in this area, among which is the use of cloud computing to run tests.

However, most paper still focuses on applying metamorphic testing to different domains rather than metamorphic testing software, frameworks, and architectures. As of today, the only implementation of the framework for metamorphic testing using cloud technologies is described in [16]. This software uses EC2 virtual machines from the AWS cloud provider with their manual creation, control, and deletion. The developed framework was used to test the bioinformatic pipeline with one run costing $21 and taking 5.5 hours instead of 35 hours of the local run of the same tests. The following Section will show that serverless computing has certain advantages over virtual machines used in [16].

Table 1 summarizes the main points of tools from overviewed papers like the type of artifact under test, the type of run, etc.

3 Overview of serverless computing

Serverless computing is the implementation of the architecture pattern “Function-as-a-Service” (FaaS), the main idea of which consists in encapsulation of the code runtime environment control [17]. With an implementation of this architecture, the software artifact as a managed code is published to the cloud provider without relation to any dedicated or virtual server. The cloud provider automatically deploys and launches a copy of the code at any available server in response to an occurrence of a defined event – it could be an HTTP request, a message in the event bus, a scheduled time, etc. The cloud provider will automatically delete the deployed copy after the event processing.

As in the case of serverless computing it is impossible to predict the function launch location, a function must be stateless – be independent of any other running processes, files in the file system, etc.

Compared to other cloud technologies like virtual machines or managed applications, the serverless computing provides the following advantages:

– Simplification of the hardware infrastructure and operations with it: if serverless computing is used, the cloud provider automatically performs horizontal scaling of used resources. This means that the number of computation nodes always corresponds to current needs: additional computation nodes could be automatically added when it is necessary to handle a big volume of concurrent requests, and unnecessary nodes will

be removed later. This feature is important for cloud applications with uneven loads – you do not need to reserve resources that will only be used during peak loads. Besides, code deployment becomes simpler compared to traditional servers;
- Economic advantages, because in the case of serverless computing, you pay only for the de facto used resources.

To confirm the second point, let us consider one of the most popular cloud providers – Microsoft Azure which provides serverless computing services called Azure Functions [18]. In this service, two factors are rated [19]:
- Number of runs, with the first million runs (a month) for free;
- Consumed resources represented by Gb*s – memory consumed by one function run multiplied by the total runtime. 400000 Gb*s a month is for free.

Thus, in a general case, the monthly expenditures can be calculated using the equation (1), where n – a number of runs a month, t – a run time of one function (seconds), m – a memory consumed by one function (Gb), $f_{c,}\, f_n$ – free limits correspondingly, p_c – a price of one Gb*s, and p_n – a price of one run.

$$(ntm - f_c)\cdot p_c + (n - f_n)\cdot p_n \tag{1}$$

Let us assume that these monthly expenditures should be compared against an alternative – processing of all requests by N virtual machines with monthly price p_{vm} per machine. Also, let us assume that the time and memory consumed by one function are known, then it is possible to calculate n, that would show that serverless calculations are cheaper (see the equation (2))

$$\begin{aligned} &(ntm - f_c)\cdot p_c + (n - f_n)\cdot p_n \le Np_{vm} \\ &ntmp_c - f_c p_c + np_n - f_n p_n \le Np_{vm} \\ &n(tmp_c + p_n) \le Np_{vm} + f_c p_c + f_n p_n \\ &n \le \frac{Np_{vm} + f_c p_c + f_n p_n}{tmp_c + p_n} \end{aligned} \tag{2}$$

Let the theoretical function to run 1 s consuming 256 Mb of memory, and as an alternative let us consider 5 virtual machines of A1 type (1 core, 1.75 Gb RAM, Linux) with monthly rate $23 for a machine [20]. As of 01.05.2022, $p_c = \$0.000016$, $p_n = \$0.0000002$, in such case, n, at that the price for serverless architecture would be equal to the price of the used virtual machines (calculated according to (2)), would be equal to approximately 28 million function runs a month (approximately 10.5 requests per second). Wherein the serverless architecture will efficiently smooth request peaks owing to automatic horizontal scaling, virtual machines would be insufficient even at average loading (as loading of one machine would be in average 2 requests per second with runtime of one request per second and only at one available core). And in the case of $n \le f_n$ the serverless architecture would be free as the consumed Gb*s are below the free use limit.

Of course, like any other architecture pattern, the serverless architecture has some disadvantages that could be critical for some sectors [17]:
- Technical disadvantages:
  - "cold" start – when function's trigger appears, the function deploying and startup would be delayed by the cloud provider, if the function was not invocated for a long time (for Microsoft Azure – about 20 minutes [21]);
  - limitation of the function run time – most cloud providers limit the maximum time for single function execution (in Azure, the maximum available time is 10 minutes). If the code may run for more time (as, for instance, in the considered work [16]), then serverless architecture cannot be applied;
- Organizational disadvantages:
  - vendor control – in the case of serverless computing, infrastructure is controlled by the cloud provider. It could lead to uncontrolled downtimes, unexpected hidden limits, and cost changes;
  - security issues – the cloud provider has access to applications. It could increase the number of security questions if the application processes the sensitive information or implements algorithms that are trade secret;
  - vendor lock-in – if you want to change the cloud provider, you will probably need to update the application replacing the vendor-specific features and libraries.

4 Metamorphic testing and serverless architecture

As was shown in [16], metamorphic tests are independent of each other and exist as individual computing units, thus allowing their parallel runs at different virtual machines.

The possibility of applying serverless computing to run metamorphic tests depends on the nature of the software artifact under test:
- software library – easily tested as it can be deployed together with the function, for instance, using the package manager;
- web service – easily tested if provided external accessibility via the Internet because the function can do HTTP requests using various libraries;
- desktop application (CLI/GUI) – testing is impossible in the majority of cases because such software artifact would require additional deploying (violation of the function stateless principle).

A generic serverless architecture of the framework for metamorphic testing of an abstract software artifact is shown in Fig. 1 as a component diagram. The interfaces on this diagram represent the public exported classes that

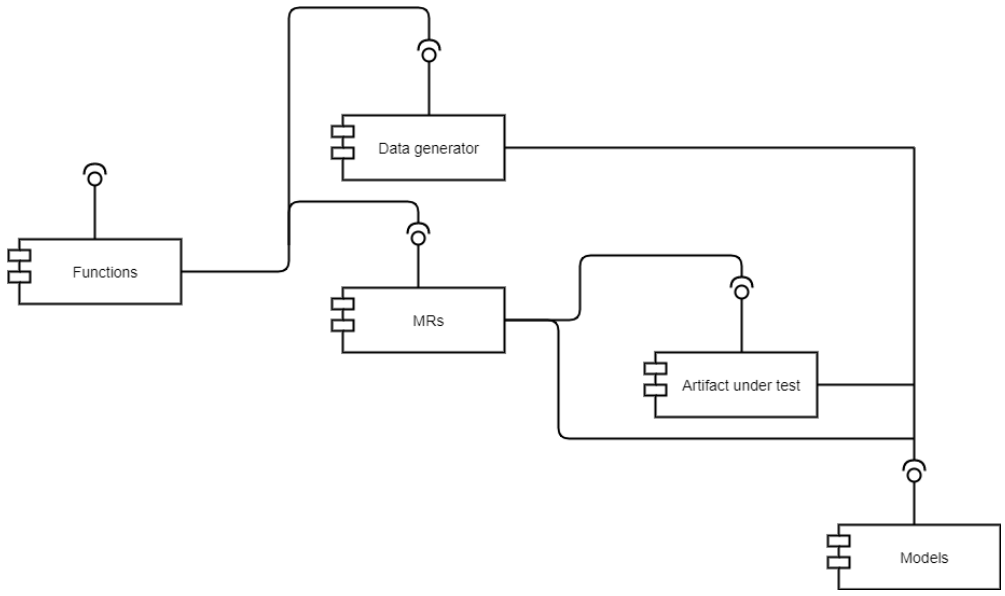

Figure 1: The generic serverless architecture: component diagram.

other components could use – these possible classes are described below.

The "Artifact under test" component in a general case corresponds to the software artifact under test. In the case of a software library, it acts in such a role itself, and in the case of a web service testing – the component encapsulates the execution of HTTP requests to the web service. This component is either connected as an independent package using the package manager or compiled and deployed along with metamorphic tests.

The "Data generator" component is responsible for obtaining input data to run metamorphic tests. Whatever input data obtaining strategies (general and specific for individual metamorphic relations) may be implemented within this component. Two main strategies may be highlighted:

- random data generation based on some passed parameters. These parameters must include a seed for the generator, which will ensure the stability and reproducibility of the tests;
- return of pre-arranged data that is either stored inside the component or at external storage (for instance, a database).

The "Models" component contains a description of data used by other architecture components. Three individual subcomponents can be identified within it: input models, output models, and data generator models.

Input models describe data supplied to the input of software artifact under test (and, correspondingly, received at the data generator output). In some cases, input models may be absent (for instance, when only primitive types or basic library types are supplied to the software artifact input) or be a part of the software artifact (if it supplies them itself). Also, if the software artifact is designed to process whatever types of data (for instance, using generics and reflection), then there will be no relation between the software artifact and input models.

Output models, correspondingly, describe data obtained at the software artifact output. Output models may be absent or embedded in the software artifact similarly to the input models.

Data generator models describe data supplied to its input. It could be both data for random generation (seed, amount of data, etc.) and, for instance, an identifier of pre-arranged data.

The "MRs" component contains an implementation of metamorphic relations that receive the input data (hence the link to the "Models" component), convert them according to the metamorphic relation, and call the software artifact under test.

The “Functions” component contains serverless functions which combine the launch of a data generator, the transfer of received data to a metamorphic relation, and the return of data to the user (hereinafter as a synonym for such functions, we will use the name “metamorphic functions”). Such function is created for each metamorphic relation.

All described components are deployed together as a serverless functions application. Metamorphic functions could use any supported launch trigger like an HTTP request or a message in a message bus. The deployment diagram of such an application (which contains N metamorphic relations) is shown in Fig. 2.

The end-user interacts with metamorphic functions using a defined user interface that calls them (using provided values for the data generator model) and processes the results (whether the test is passed or not). By "end-user" we mean not only a person but also any other software, such as CI/CD pipelines. In the case of a person, the user interface could be implemented in the form of GUI software with the fields for entry of data generator model values and selection of metamorphic functions to be called. In the case of any other software, it could be software libraries, CLI software, shell scripts, etc.

The sequence diagram of the function life cycle when the function is called is shown in Fig. 3. This diagram shows interactions between defined components (see Fig. 1), the order, and which specific models from the

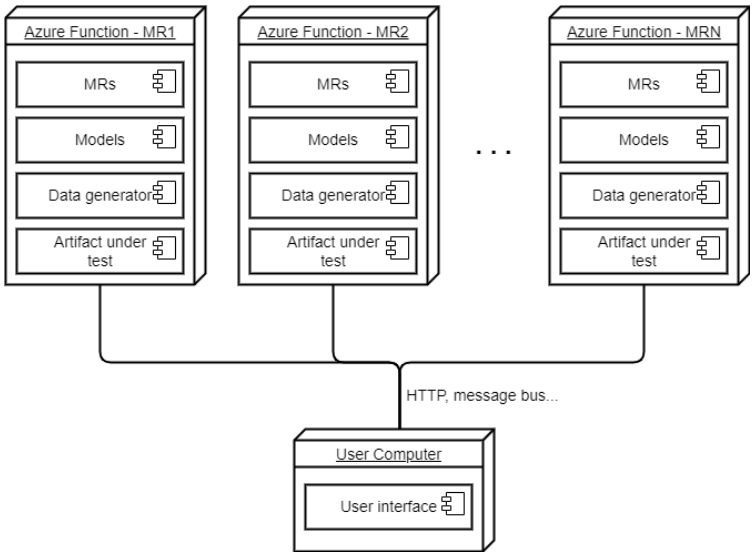

Figure 3: The generic serverless architecture: deployment diagram.

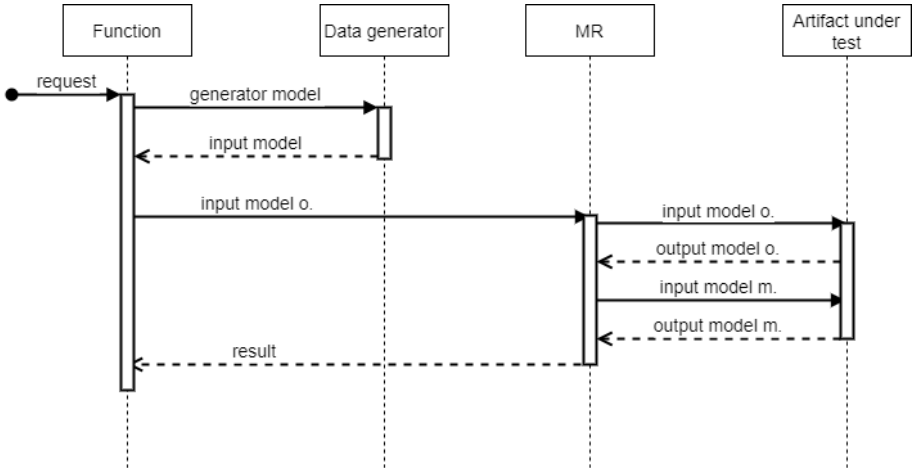

Figure 2: The generic serverless architecture: sequence diagram.

"Models" component are used for each specific interaction.

5 Experiment

5.1 Experiment design

The experimental Section aims to measure the performance of serverless metamorphic testing and define possible delays compared with local/virtual machine execution.

To achieve that, two software for metamorphic testing were implemented, corresponding to artifact types and data generator types applicable to serverless architecture (see Section 4). The first software implements metamorphic testing of a software library using a random data generator, and the second implements metamorphic testing of a web service using a data generator with constant data. Also, the software artifacts for the experiment were chosen in such a way that one metamorphic test does not take much time and is completed in a few seconds at most. This is to compensate for the possible difference in hardware when running locally and in the cloud, to focus only on the difference due to different architectures.

Each implemented software contains five metamorphic relations defined for the chosen software artifacts. So, there are ten metamorphic relations in total, each of which can be considered a separate experiment for the performance testing.

The two developed Function Apps were deployed to Azure in the "West Europe" region with Windows operating system. Each implemented metamorphic function was run in two possible modes – "warm" start and "cold" start – to define delays in both cases. To be sure that there was a "warm" start, the first three queries for the metamorphic function were not included in the statistics - that is, they were run to "warm-up" the metamorphic function. To ensure a "cold" start at each run, a delay of 30 minutes was implemented between each function call (Azure describes 20 minutes delay in the documentation, additional 10 minutes were taken for confidence).

5.2 Subject and defined MRs No.1

A public library YetAnotherConsoleTables of one of the authors was selected as a software artifact for metamorphic testing using serverless architecture [22]. The library's primary purpose is an output of the transmitted collection of objects at the console (or any other text output) as a table (see Fig. 4).

```
-------------------------
| Property1 | Field1    |
-------------------------
| 4299      | My String |
-------------------------
| 475       | My String |
-------------------------
| 4142      | My String |
-------------------------
| 239       | My String |
-------------------------
| 6547      | My String |
-------------------------
```

Figure 4: An example of an output of the transmitted collection of objects containing two fields: numeric and string.

It is obvious that each row in the table has an equal length that can be calculated using the equation (3), where $\text{length}(v_{ji})$ – length of the string representation of the field i of the object with index *j* (totally *m* objects), and $\text{col\_name}(i)$ – the string representation of the i field name.

$$w = 1 + \sum_{i=1}^{n} 3 + \max(\max(\text{length}(v_{ji})), \text{col\_name}(i)) \quad (3)$$

The total number of rows in the received table is easily calculated using the equation $h = 3 + 2m$, where 3 – the fixed number of rows in the table header, and *m* – number of objects in the collection.

For the selected software artifact there are five metamorphic relations identified that could be grouped into three groups: manipulations with the number of objects; manipulations with the number of fields; and other.

MR Group 1: Manipulations with the number of objects:

1) MR1.1. Collection reduction. Let $C_o = \{o_1, o_2, ..., o_m\}$ – collection of objects, and m_1 function deletes its last object: $m_1(C_o) = C_m = \{o_1, o_2, ..., o_{m-1}\}$. Then, $h_o - 2 = h_m$.

2) MR1.2. Collection increase. Let $C_o = \{o_1, o_2, ..., o_m\}$ – collection of objects, and m_1 function adds one object to it: $m_1(C_o) = C_m = \{o_1, o_2, ..., o_{m+1}\}$. Then, $h_o + 2 = h_m$.

MR Group 2: Manipulations with the number of fields:

3) MR2.1. Field deletion. Let the type of collection objects consists of *n* fields: $t_o = \{col_1, col_2, ..., col_n\}$, and the m_1 function creates on its basis a new type without the last field: $t_m = \{col_1, col_2, ..., col_{n-1}\}$, and maps the original collection of objects into the collection of new type objects. Then, $w_o - (3 + \max(\max(\text{length}(v_{jn})), \text{col\_name}(n))) = w_m$.

4) MR2.2. Adding a field. Let the type of collection objects consists of *n* fields: $t_o = \{col_1, col_2, ..., col_n\}$, and the m_1 function creates on its basis a new type by adding a new field: $t_m = \{col_1, col_2, ..., col_{n+1}\}$, and maps the original collection of objects into the collection of new type objects recording the constant value *const* for a new field of each object. Then, $w_o + 3 + \max(\text{length}(const), \text{col\_name}(n+1)) = w_m$.

MR Group 3: Other:

5) MR3. Change of the order of objects in the collection. Let $C_o = \{o_1, o_2, ..., o_m\}$ – a collection of objects, and the m_1 function changes the order of objects into the reverse order: $m(C_o) = C_m = \{o_m, o_{m-1}, ..., o_1\}$. Then the original table's row that corresponded to the object with index i (the index of this row can be calculated using the equation $3 + 2(i-1)$, with numbering starting with zero) corresponds to the received table's row for the element with index $m - i + 1$.

The schematic representation of defined metamorphic relations is shown in Fig. 5. The light-yellow color indicates the "Add" operation (row/object or field/column, it does not matter), the light-red color indicates the "Remove" operation, and the pair of blue-green colors are used to identify different rows.

5.3 Implemented software No.1

The software for metamorphic testing of the library YetAnotherConsoleTables was implemented using the .NET platform and C# 9.0 language and is available at https://github.com/yakivyusin/MTServerless/tree/master and corresponds to the described serverless architecture.

The software library YetAnotherConsoleTables corresponding to the "Artifact under test" component from the UML diagram (see Figure 1) is connected owing to the NuGet package manager.

The "MTServerless.Models" project corresponds to the "Models" component; it contains the description of the input model as well as the data generator model. The input model contains three fields (one of numeric type and two of string type), and the data generator model contains the initial value for the generator of pseudorandom numbers (as GUID) and the number of objects in the collection to generate: "Seed" and "Count" correspondingly.

Compared to the basic architecture, two differences could be noted:

- As the YetAnotherConsoleTables library is implemented to output collections of whatever type, there are no references between the library and the model project.
- As the output of the artifact under test is a set of rows for the console/other text output, there is no

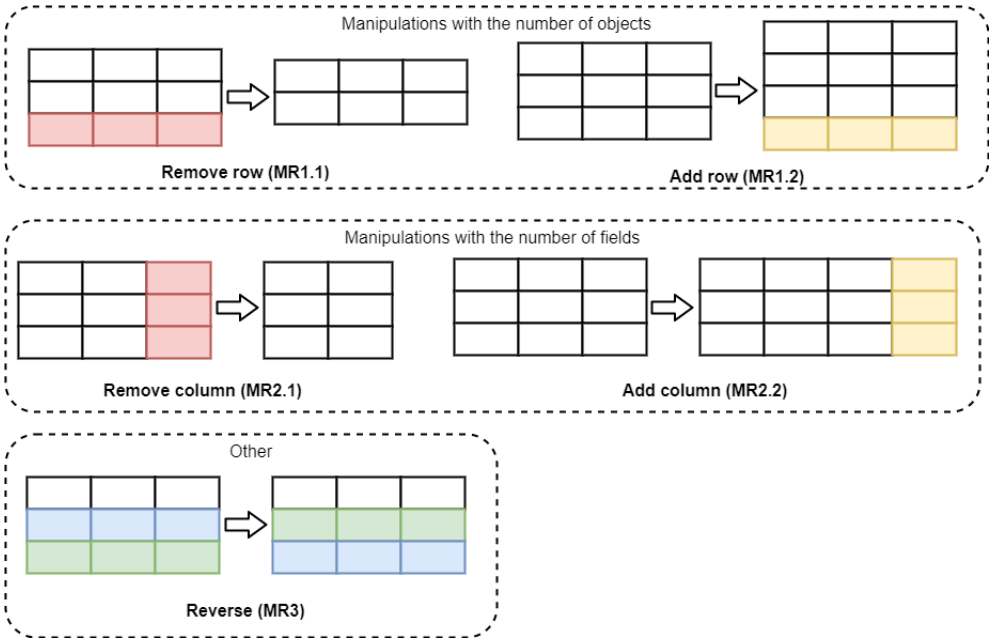

Figure 5: Defined metamorphic relations for the library of tabulated output.

need to describe the output model (types of strings, StringWriter are available at the .NET basic library).

The "MTServerless.Generator" project corresponds to the "Data generator" component that generates a collection of input objects based on the transmitted parameters. The field "Seed" of the data generator model is used for initialization of the pseudorandom number generator, which is later used for filling input model numeric and string fields.

The "MTServerless.Relations" project corresponds to the metamorphic relations component itself. Each class in this project corresponds to one metamorphic relation receiving a collection of input objects at the input and returning the Boolean value (whether the metamorphic relation is held or not). As a specific feature of such relations, it is worth mentioning the implementation of relations MR2.1 and MR2.2 that for manipulations with the number of fields create anonymous types based on the input (anonymous types were introduced in C# 3.0 [23]).

The "MTServerless" project contains a set of Azure Functions for each metamorphic relation (Functions SDK 3.0 was used for their implementation). A function is called using HTTP requests, and input parameters for the data generator are transmitted using the HTTP request query parameters: "seed" and "count" (so, the URL of a function to be called looks like https://example.com/Function?seed=b6cb1f5b-3cc5-4ed8-b75e-51c99a900a19&count=5). These parameters are used to create an instance of the data generator model type, which will be passed to the data generator.

5.4 Subject and defined MRs No.2

As an example of a web service serverless metamorphic testing case, it was decided to reproduce the software from the paper [7], which describes the metamorphic testing of web search engines.

This paper identifies five metamorphic relations grouped into two groups: "No Missing Web Page" and "Consistent Ranking."

"No Missing Web Page" MR Group:

1) MPSite. This metamorphic relation checks if some page was found using query A, then this page also should be found using query $B = A +$ page domain restriction.

2) MPTitle. This metamorphic relation checks if some page was found using query A, then this page also should be found using query $B = A +$ page title text.

3) MPReverseJD. This metamorphic relation checks that search results for queries $A = A_1 \wedge A_2 \wedge A_3 \wedge A_4$ and $A = A_4 \wedge A_3 \wedge A_2 \wedge A_1$ are similar.

"Consistent Ranking" MR Group:

4) SwapJD. This metamorphic relation checks that search results for queries $A =$ word1 word2 and $A =$ word2 word1 are similar.

5) Top1Absent. This metamorphic relation checks that the first result for search query A will also appear in search results for query $B = A +$ page domain restriction.

Instead of a random data generator, a data generator with constant values is used for this program artifact. Each metamorphic relation has its constant value chosen from examples in the original paper [7]. For metamorphic relations, which include a similarity check, the threshold 0.5 of the Jaccard coefficient was used.

It was decided to use DuckDuckGo [24] as a web search engine for metamorphic testing because this engine provides a simple HTML version that is easy to parse. Also, DuckDuckGo uses Google as an underlying engine, so the quality of results is the same.

5.5 Implemented software No.2

The software for metamorphic testing of the DuckDuckGo web search engine was implemented using the .NET

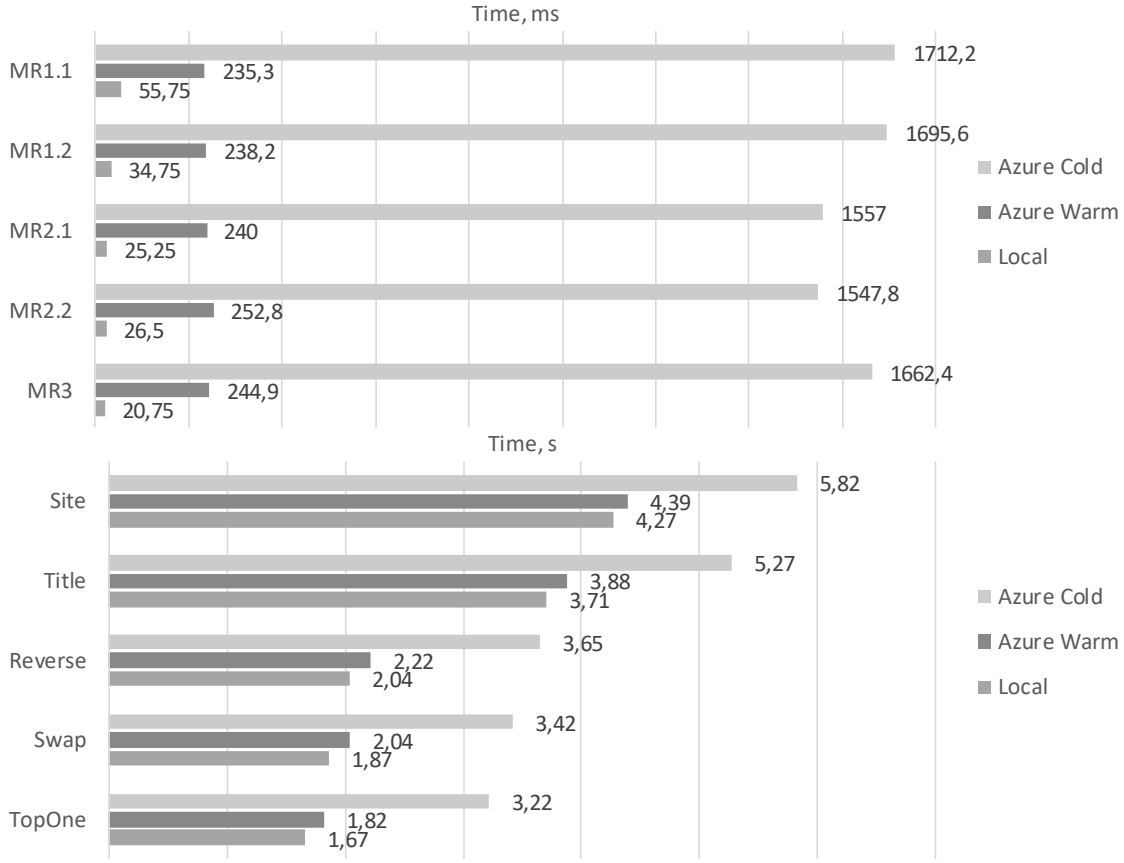

Figure 6: Metamorphic functions execution time in Azure ("warm" and “cold” start) and locally.

platform and C# 9.0 language and is available at https://github.com/yakivyusin/MTServerless/tree/search and corresponds to the described serverless architecture.

The “MTServerless.Artifact” project corresponds to the “Artifact under test” component of the basic architecture. As was described in Section 4, this project encapsulates HTTP requests to the actual artifact under test – DuckDuckGo web search engine – and parses the HTML response to return query results.

The “MTServerless.Generator” project corresponds to the “Data generator” component. This data generator accepts only one parameter – the identifier of the metamorphic relation (a simple order number) for which the generator is currently being called. Based on the passed identifier, the constant search query is returned.

The “MTServerless.Models” project corresponds to the “Models” component. In the case of the web search service, this project only describes the output model. The output model is presented as a “SearchResult” class, which contains three fields: site title, URL, and description. The input model and the data generator model are absent, because they are presented in the form of built-in .NET classes: string for the search query (the input model) and integer number for the data generator model.

The “MTServerless.Relations” project corresponds to the metamorphic relations component itself. Each class in this project corresponds to one metamorphic relation receiving a search query at the input and returning the Boolean value (whether the metamorphic relation is held or not). It should be noted that metamorphic relations, which build a follow-up query for each search result (MPSite and MPTitle), contain a 2 seconds delay before each follow-up query. This delay is implemented to avoid a temporal ban from DuckDuckGo, and all these delays will be subtracted from performance results.

The “MTServerless” project contains a set of Azure Functions for each metamorphic relation (Functions SDK 3.0 was used for their implementation). A function is called using HTTP requests without any input parameters – each function passes the hard-coded identifier to the data generator.

5.6 Results

First, the time of described metamorphic relations execution was compared in the case of the “warm” Azure start and the case of the local start. The term "execution time" in the context of these studies means the time between the HTTP request sending and receiving the server response. The obtained results are provided in Fig. 6.

As one can see, the mean execution time of metamorphic function in Azure with a "warm" startup is, on average, 200 ms longer than the mean execution time in the case of local deployment. This deceleration is caused by network delays (which may be considered a constant for each specific case) and possible delays of the cloud provider. A specific value of the network delay constant depends on the quality and speed of Internet connection and mutual geographic location of the

metamorphic testing client and selected data center of the cloud provider. Also, in the case of web application testing, there is a possible difference in network delays between the web application server and the deployment place of a testing framework. If the cloud provider's data center is closer to the web application server, testing will be faster than in the case of local deployment.

The metamorphic functions execution time in Azure was also measured for the "cold" start. The obtained results are provided in Fig. 6 too.

As one can see, the additional "cold" start delay is 1400 ms on average for the developed software. Such delay may depend on many factors, to name the main ones: the size of the artifact with metamorphic functions (because the artifact should be transferred from a storage to a server); the number of the tested artifact's external dependencies (the more dependencies you need to restore using the package manager, the longer it takes); current loading of the cloud provider's infrastructure. However, two out of the three above factors are constants for specific software, and the current loading of the infrastructure (based on analysis of the obtained data outliers) may maximum add 600 ms; therefore, the "cold" start delay may also be considered constant for defined metamorphic function.

Thus, it could be considered that the proposed serverless framework architecture for metamorphic testing increases the individual metamorphic relation execution time (compared to traditional local architectures) by the constant $C_i + \varepsilon$ in the case of a "warm" start and by the constant $C_i + C_c + \varepsilon$ in the case of a "cold" start (where C_i – the constant of network delays, C_c – the constant of software complexity, and ε – the error representing dependencies on the cloud provider). Nevertheless, in practice, such delays may be considered nonessential because executing one metamorphic function may take minutes.

The developed serverless architecture shows the best results in the case of concurrent execution of a set of metamorphic tests because in such case, all functions are executed simultaneously (degree of concurrency = number of functions) compared to the local start where the degree of concurrency is limited by the hardware/compared to the virtual machines where the count of installed VMs limits the degree of concurrency. As a result, for serverless computing, the total execution time of a test set will be close to the maximum execution time of one metamorphic function when the Amdahl's law will limit the total execution time of a test set for local/VMs.

6 Discussion

In this paper, two tools for metamorphic testing of artifacts of different types were developed using the proposed basic architecture. A total of 10 metamorphic relations were made - 5 for each artifact considered.

The developed tools differ from the tools considered in Section 2 by using cloud serverless technologies, allowing them to run locally and in the cloud without any additional changes (Azure provides tools for running serverless functions locally). The results show that despite additional network and other delays, running serverless functions in the cloud will reduce the overall metamorphic testing time for more complex artifacts or more relations. The main achievement of this paper is the idea of using serverless computing for metamorphic testing and the basic architecture of developing such software.

7 Conclusion

The study considered the possibility of using serverless computing and Function-as-a-Service pattern for metamorphic testing and developed the corresponding framework architecture.

Serverless computing provides the following advantages for metamorphic testing:

- compared to local starts – the maximum degree of concurrency that equals the number of metamorphic relations;
- compared to virtual machines – infrastructure simplification (no need to save VM images and deploy them when necessary) and expedition of calculations as functions are deploying quicker.

The developed architecture for metamorphic testing using serverless computing: represented as component, deployment, and sequence diagrams; optimized for serverless computing; ensures components isolation.

The study demonstrated the use of the proposed architecture for metamorphic testing of two different software artifacts. The obtained results show that in the case of a "warm" start, the serverless computing introduces an individual metamorphic function execution delay of ~200 ms compared to a local start. However, in the case of simultaneous start of the whole package of metamorphic tests, the serverless architecture achieves the performance of local architecture, and with an increase in the number of tests and/or their complexity, the serverless computing is much quicker. Thus, the developed serverless architecture of metamorphic testing is expedient for practical application if the disadvantages of serverless computing are not critical for this specific case. E.g., the bioinformatic pipeline from the paper [16] could not be tested using the proposed serverless architecture due to the long execution time. However, any software artifact with an execution time less than 10 minutes could be efficiently tested using the serverless framework.

Acknowledgement

Authors would like to thank: Microsoft company for providing free limits for a lot of their Azure cloud services, which were very helpful during this and other research; Pavlo Holianytskyi for his help with the English paper version; and Larysa Yusyn for her motivating of authors to finalize the research.

References

[1] Weyuker E., The Oracle Assumption of Program Testing, Proc. of the 13th International Conference

on System Sciences (ICSS), Honolulu, HI, January 1980, pp. 44-49.

[2] Barr T., Harman M., McMinn P., Shahbaz M., Yoo S., The Oracle Problem in Software Testing: A Survey, IEEE Transactions on Software Engineering, 41 (5), pp.507–525, 2015. https://doi.org/10.1109/TSE.2014.2372785

[3] Chen T., Cheung S., Yiu S., Metamorphic testing: a new approach for generating next test cases, Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong, 1998.

[4] Bourque P., Fairley R., Chapter 4: Software Testing, SWEBOK v3.0: Guide to the Software Engineering Body of Knowledge. IEEE. pp. 4–1–4–17, 2014.

[5] Zhou Z., Zhang S., Hagenbuchner M., Tse T., Kuo F.-C., Chen T., Automated functional testing of online search services, Software Testing, Verification and Reliability, 22 (4), pp.221-243, 2012. https://doi.org/10.1002/stvr.437

[6] Zhou Z., Tse T., Kuo F.-C., Chen T., Automated functional testing of web search engines in the absence of an oracle, Technical Report TR-2007-06, Department of Computer Science, The University of Hong Kong, Hong Kong, 2007.

[7] Zhou Z., Xiang S., Chen T., Metamorphic testing for software quality assessment: A study of search engines, IEEE Transactions on Software Engineering, 42 (3), pp.264-284, 2016. https://doi.org/10.1109/TSE.2015.2478001

[8] Tao Q., Wu W., Zhao C., Shen W., An automatic testing approach for compiler based on metamorphic testing technique, Proc. of the 17th Asia Pacific Software Engineering Conference (APSEC), pp.270-279, 2010. https://doi.org/10.1109/APSEC.2010.39

[9] Le V., Afshari M., Su Z., Compiler validation via equivalence modulo inputs, Proc. of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp.216-226, 2014. https://doi.org/10.1145/2666356.2594334

[10] Kuo F.-C., Liu S., Chen T., Testing a binary space partitioning algorithm with metamorphic testing, Proc. of the 2011 ACM Symposium on Applied Computing, pp.1482–1489, 2011. https://doi.org/10.1145/1982185.1982502

[11] Jameel T., Mengxiang L., Liu C., Test oracles based on metamorphic relations for image processing applications, Proc. of the 16th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), pp.1-6, 2015. https://doi.org/10.1109/SNPD.2015.7176238

[12] Pullum L.L., Ozmen O., Early results from metamorphic testing of epidemiological models, Proc. of the ASE/IEEE International Conference on BioMedical Computing (BioMedCom), pp.62-67, 2012. https://doi.org/10.1109/BioMedCom.2012.17

[13] Ramanathan A., Steed C.A., Pullum L.L., Verification of compartmental epidemiological models using metamorphic testing, model checking and visual analytics, Proc. of the ASE/IEEE International Conference on BioMedical Computing (BioMedCom), pp.68-73, 2012. https://doi.org/10.1109/BioMedCom.2012.18

[14] Segura S., Fraser G., Sanchez A., Ruiz-Cortes A., A survey on metamorphic testing, IEEE Transactions on Software Engineering, 42 (9), pp.805-824, 2016. https://doi.org/10.1109/TSE.2016.2532875

[15] Chen T., Kuo F.-C., Liu H., Poon P.-L., Towey D., Tse T., Zhou Z., Metamorphic testing: A review of challenges and opportunities, ACM Computing Surveys 51 (1), pp.1-27, 2018. https://doi.org/10.1145/3143561

[16] Troup M., Yang A., Kamali A., Giannoulatou E., Chen T., Ho J., A cloud-based framework for applying metamorphic testing to a bioinformatics pipeline, Proc. of the 1st International Workshop on Metamorphic Testing (MET '16). ACM, New York, NY, pp.33–36. https://doi.org/10.1109/MET.2016.014

[17] Roberts M., Serverless Architectures, [Online]. Available: https://martinfowler.com/articles/serverless.html (accessed October 1, 2021).

[18] Microsoft, Azure Functions – Serverless Apps and Computing, [Online]. Available: https://azure.microsoft.com/en-us/services/functions/ (accessed October 1, 2021).

[19] Microsoft, Azure Functions Pricing, [Online]. Available: https://azure.microsoft.com/en-us/pricing/details/functions/ (accessed October 1, 2021).

[20] Microsoft, Pricing Calculator, [Online]. Available: https://azure.com/e/243a23dd3327440bb5567813152d3ef0 (accessed October 1, 2021).

[21] Tresness C., Understanding serverless cold start, [Online]. Available: https://azure.microsoft.com/en-us/blog/understanding-serverless-cold-start/ (accessed October 1, 2021).

[22] Yusyn Y., YetAnotherConsoleTables: NuGet Package, [Online]. Available: https://www.nuget.org/packages/YetAnotherConsoleTables/ (accessed October 1, 2021).

[23] Richter J., CLR via C# 4th Edition, Microsoft Press, 2013.

[24] DuckDuckGo, Main Page. [Online]. Available: https://duckduckgo.com/ (accessed October 1, 2021).