M2FaaS: Transparent and fault tolerant FaaSification of Node.js monolith code blocks

Stefan Pedratscher *, Sasko Ristov, Thomas Fahringer

University of Innsbruck, Technikerstraße 21a, A-6020 Innsbruck, Austria

1. Introduction

Function-as-a-Service (FaaS) became a popular technology in recent years to develop and run cloud applications [1]. Recently, AWS started to offer a pay-as-you-go pricing model with a granularity of 1 ms. FaaS’ popularity grows even more because developers build and deploy their codes as serverless functions and the entire underlying platform and infrastructure is completely managed by FaaS providers [2]. Afterwards, serverless functions are seen as a black-box and usually accept JSON-formatted input and produce JSON-formatted output.

Larrucea [3] and Mazlami [4] reported that the distribution of a monolith into several microservices improves the performance. Moreover, offloading parts of the monolithic application to the cloud reveals several additional benefits through the use of the FaaS paradigm. The use of serverless functions might additionally improve the performance by providing more resources for a specific part of the monolith and exploiting the benefits of FaaS elasticity. This results in more granular elasticity to scale the resources only for a part of the code that is resource demanding rather than overprovision the virtual machine during the entire runtime.

The process of converting a monolith into a serverless application is known as FaaSification [5]. In the paper we use the prefix “FaaS” when we refer to names and “faas” to represent operations or verbs. With FaaSification, a monolith is converted into a hybrid application, usually comprising the faasified monolith and the extracted serverless functions.

FaaSification may be a tedious operation because developers need to assure that all interdependencies should be included recursively, within each serverless function. Due to the stateless nature of serverless functions, developers must work around many limitations [6] including limited function duration, code size, size of data inputs and outputs, or even the number of concurrent invocations. In order to exploit the benefits of the FaaS paradigm, developers are challenged to convert existing legacy applications (monoliths) to the FaaS architectural style [7]. In the monolithic architectural style, developers assume that all required files and packages are in the scope of the monolith.
and therefore accessible to all methods of the monolith. This approach, however, is not suitable for serverless architectural style, which requires serverless functions to be isolated from each other, because every function has been isolated in its own runtime environment including its required data.

In a cohesive application like monoliths, elements of the code and dependencies are highly functionally related. In our initial investigation of monolithic architectural style, we identified three classes of dependencies that should be considered for monolith FaaSification. If any of these dependencies are not resolved, the faasified serverless functions may not work properly. The first class contains package dependencies, which require several activities by developers to ensure that extracted serverless functions contain all needed packages to run properly. For instance, in Node.js monoliths, the required packages need to be declared in the code of the serverless function, included in the deployment package, and zipped with the extracted code that is deployed as a serverless function. Another class of dependencies are the code dependencies which represent the inter-method dependencies. Code dependencies appear frequently in GitHub projects [8]. These dependencies are for example methods from external files, created by developers. Similar as package dependencies, code dependencies must be resolved. This process assumes that all callees that are declared in external files, must be bundled with the deployment package, as well. This is a recursive process since the callee methods may depend on other packages, files, or variables. With FaaSification, the cohesiveness of the monolith is broken, which affects the scope of the variables including their values. The cohesiveness additionally leads to the third class of dependencies, which we refer to as data dependencies. Developers need to consider two types of data dependencies. Firstly, it must be ensured that all variables in the code of the serverless function will be in the correct scope, declared properly, and the values from the last update in the faasified monolith will be passed to the serverless function. Analogously, values of all updated variables in the serverless function should be passed back to the faasified monolith if they are used afterwards in order to continue to work properly. Mainly, the recent FaaSifier DAF [9] partially tackles the package and code dependencies in order to ensure that extracted serverless functions run properly. Another recent FaaSifier named Node2FaaS [10] assumes an isolated function. However, to the best of our knowledge, this paper is the first one that handles data dependencies, which ensures that not only the serverless functions runs properly, but also the entire faasified monolith.

Even after handling all dependencies, still FaaSification may raise other challenges for the hybrid application in terms of fault tolerance. Extracted serverless functions may increase the failure rate because additional failures may be caused by networking issues, throttling, various FaaS provider limitations, or API invoke errors. The increased failure rate requires additional measures in the hybrid application.

To manage all above-mentioned dependencies and fault-tolerance challenges, this paper introduces M2FaaS, which automates FaaSification of a monolith. M2FaaS offers FaaSification in two stages: annotation and automatic hybrid generation. In the annotation stage, the developer analyzes the code and determines which arbitrary lines of the monolithic code (code blocks) will benefit from offloading as a serverless function. Further on, the developer annotates the code blocks by specifying variables that should be forwarded to functions, dependencies that should be included, and variables that should be returned back to the monolith. In the second stage, M2FaaS automatically generates a hybrid version of the monolith by offloading the annotated code blocks as serverless functions, by resolving all dependencies and creating bidirectional interfaces between the monolith and each function to pass all needed variables to the serverless functions and the faasified hybrid.

Moreover, M2FaaS tackles fault tolerance and allows developers to annotate multiple alternative functions that are deployed on multiple regions and cloud providers, which are invoked in case of a failure. Alternative functions refer to multiple deployments of the same code block on different cloud providers or regions. In contrast to fault tolerance techniques of cloud providers, in which functions are simply retried within the same region only after a failure, M2FaaS increases reliability of the hybrid. For instance, if the FaaS provider refuses further requests beyond the limit or a request failed due to the networking issues, M2FaaS may run an alternative function on another FaaS provider. Invocations of alternatives are transparent to the user. With these innovative features for automatized FaaSification, M2FaaS may reduce the development effort by at least 18.12% and a higher throughput in case of failures of at least 18.5% compared to the state-of-the-art FaaSifiers. Moreover, the latter improvement is achieved by completely tolerating failures with failure rate of 27.18%. Related work generated a higher failure rate of 40.29% all of which were not tolerated.

This paper introduces several contributions:

- Publicly available FaaSifier that will help practitioners to faasify Node.js monoliths with complex dependencies;
- Automated handling of code, package, and data dependencies with simple annotations;
- Automated deployment across multiple FaaS providers with simple annotations to specify provider, region, and assigned memory;
- Injecting fault tolerance in the faasified monolith;

This paper is organized in several sections. We motivate the benefits of FaaSification, define three phases of FaaSification’s life-cycle in Section 2 and evaluate which steps are supported by the state-of-the-art FaaSifiers. Section 3 presents the system architecture of M2FaaS and shows examples how to faasify a code block of a Node.js monolith. We present the results of the M2FaaS evaluation with several realistic applications in Section 4. Section 5 discusses the related work and how M2FaaS advances beyond the state-of-the-art FaaSifiers, along with M2FaaS’s limitations. Finally, we conclude our work and present plans for the future work in Section 6.

2. Motivation

In this section we motivate the benefits of FaaSification and the needs for fault tolerance and transparency. Further on, we present the entire life-cycle of FaaSification to focus not only on faasified serverless functions, but also on the resulting faasified monolith. Finally, we evaluate the support that the existing FaaSifiers offer for each step of the FaaSification life-cycle.

2.1. Need for FaaSification

FaaS is gaining more and more traction, and there is a real need for smarter tools to facilitate the work of developers in migrating their legacy programs to this new paradigm. FaaS properties are very appealing for many monoliths because the resulting hybrid application will gain several benefits from FaaSification. Firstly, FaaSification may provide enormous speedup of the embarrassingly parallel monoliths because users can spawn numerous functions simultaneously. Even monoliths with internal dependencies may benefit from FaaSification, although they are not scalable. Another class of monoliths that may gain from

---

1. https://github.com/Apollo-Tools/M2FaaS.
FaaSification are burstable ones, whose computing requirements of some code block is burstable for a short period of time and low flat load for the rest of the code. Running them in a VM or as a container would often cause either much higher costs due to overprovisioning or worse performance or even failures in case of underprovisioning. For example, many applications load data from external storage or data repository, perform complex computing over data in a code block and store the results back to a storage. In such case, the monolith requires high amount of computing resources for the middle part only because the other parts perform mainly I/O operations. Fig. 1 presents an example of a burstable monolith and its possible memory consumption over time. Without losing generality, we show that the monolith has a single burstable code block. We also show only a few options to run the monolith using t2 family.

The safest solution for the user is to overprovision the resources and run the burstable monolith on t2.xlarge VM. However, this approach generates the highest cost, which is not justified. A cheaper solution would be to use t2.large, in which case the burst code block may struggle for resources. Even cheaper solutions would be to use t2.medium or the instance type t2.small. However, underprovisioning for the burst code block may lead to a failure. On the other side, offloading the burst code block as a serverless function may both reduce the costs and avoid failures. The blue dashed line shows ideal resource provisioning for the hybrid. Namely, we could run the monolith on t2.micro and deploy the burst code block as a serverless function with 10 GB memory, which is supported by AWS Lambda. Even though, every second on a t2.micro instance has similar cost as 0.2 s AWS Lambda execution with the same memory assignment and t2.large instances cost around 10% of an AWS lambda function execution with the same memory assignment, the cost can be decreased by combining EC2 instances and cloud functions. We compared the execution cost of (i) the overprovisioning case, or running the whole application shown in Fig. 1 on a t2.xlarge instance and (ii) the hybrid case, or running the monolith on a t2.micro instance and the burst code block as a serverless function with 10 GB. The results showed that the hybrid case is cheaper if the burstable code block is up to 22% of the time. If, for instance, the burstable code duration is only 5% of the time, then the hybrid case reduces the costs by 78%.

2.2. Need for fault tolerance

If we neglect the interfaces to other services that a monolith may use, its code is cohesive and the probability of a failure may be minimized with proper software testing. However, FaaSification breaks the cohesiveness of the monolith by distributing the extracted code in the cloud in a form of stateless and serverless functions. Such distribution may increase the failure rate because the serverless function deployment may fail due to code size limitations or its execution may fail due to the limited size of data input or output. Moreover, the serverless function execution may fail due to network connection, or due to some FaaS provider resource limitations such as maximum assigned memory or maximum duration. Many of these failures cannot be determined with static code analysis because they may happen occasionally based on specific values of input data and be known during runtime. Many FaaS providers state in their Service Level Agreements (SLAs) that they reimburse the users if availability is lower than 90%, 95%, or even below 99.95%. Still, serverless functions may fail due to various reasons that are not raised within the monolith execution. Carvalho et al. [11] reported that various types of serverless functions (CPU, memory, or I/O intensive), which were extracted with Node2FaaS from a monolith, fail. The failure rate was extremely high for memory intensive serverless functions, although the monolith success rate was 100%.

Most public cloud providers offer built-in fault tolerance mechanisms for their serverless functions. However, these mechanisms mainly include retrying the function several times in case of a failure. On the other hand, FaaS providers simply return an error message if a user reaches limitations set by the FaaS providers. For instance, when a synchronously invoked function runs longer than the duration limit or when the size of the data input or output is larger than the limit. If these potential failures are not properly handled with exceptions in the faasified monolith, it will fail. Indeed, many FaaS providers offer try-catch mechanisms in their serverless workflow management systems, such as AWS Step Functions, but developers need to rewrite the entire monolith to be compatible with the individual systems, e.g. a state machine for AWS Step Functions.

2.3. Need for transparency and freedom in configuration

The inclusion of the dependencies from the monolithic architectural style into serverless functions is another challenge that needs to be tackled in order to benefit from the serverless technology. The resulting hybrid application should retain all interfaces, while clients should still be able to interact with the application in a transparent way [12]. The location where the computing is performed and who is serving the request should be irrelevant for the client [13]. However, many parameters need to be specified in order to deploy a function. To begin with, a developer needs to decide and specify FaaS providers and regions in which functions should be deployed. Further on, function names, execution rights, timeout, and the amount of memory need to be assigned to the function individually. In order to keep the same interfaces to the users, the existing FaaSifiers use default settings for faasified serverless functions. This approach restricts the user to offload all functions only on a single region of a single provider, which are specified in the credentials file. Further on, specifying a fixed value for

---

2 t2 instances are one of the cheapest Amazon EC2 general purpose instances. Very popular is t2.micro, which contains 1 GB RAM and is charged 0.0058 $/h. The charge increases linearly up to 0.3712 $/h for t2.xlarge instance, which contains 32 GB RAM. https://aws.amazon.com/ec2/instance-types/t2/.
properly as a serverless function. However, problems for developers do not stop here. Namely, as presented in Fig. 2, after the extracted code block, the monolith may use variables that were updated in the extracted serverless function. In the given example in Fig. 2, the value of the variable `local` may be changed by the serverless function from 1 to 0, which directly affects the control flow of the monolith when using this variable `local` within the `if` condition. Even worse, if the variable `local` is not declared or initialized with value 1 before the code block, and then used later by the monolith after the code block `(if (local))`, the faasified monolith may crash with an error “ReferenceError: local is not defined” after replacement of the code block with an API call to the serverless function because the variable `local` is not declared. In order to solve these two challenges, FaaSification requires a function to monolith interface (F2M), which passes the values of all updated variables from the serverless function back to the monolith. In this case, the variable `local` is updated in the serverless function and its new value must be returned back to the monolith variable `local`. Regardless whether the variable `local`, which is updated in the extracted code block is declared or not before the annotated code block, the assignment of the returned value solves both the (re)declaration and assignment of the correct value. The F2M interface also requires two adaptations, both in the extracted serverless function and the faasified monolith. In the serverless function, a developer needs to prepare data outputs (usually required in JSON format), while in the faasified monolith, the developer needs to add code to read the JSON-formatted output data from the serverless function output and pass the values back to the faasified monolith.

The above-mentioned example shows that even for a simple Node.js monolith, FaaSification requires substantial development activities in order to faasify even a single serverless function. The development effort linearly grows with the number of code blocks that should be faasified and the dependencies they have. For a real-world application, this manual process is a very costly operation. GitHub projects have on average of 200 package dependencies, reaching 1000 and above in some cases [14]. Moreover, some of these dependencies need to be resolved in multiple serverless functions.

Based on the above analysis we introduce three phases of the FaaSification. We assume that before FaaSification, developers determined which code blocks of the monolith should be faasified. For this pre-phase we refer the readers to approaches to decompose monoliths proposed by related work and some of which are presented in Section 5.1.

2.4.1. Phase 1: Create the serverless function
The goal of the first phase is to adapt the extracted code in order to run as a serverless function. This involves several steps: declare all requirements (code and package dependencies), read the inputs and assign their values to the corresponding variables that are used within the serverless function code, and finally, prepare and return the output of the serverless function by returning all variables that are needed in the monolith.

2.4.2. Phase 2: Deploy the serverless function
The goal of the second phase is to create a deployment package (e.g., zip or jar) and deploy it on the target cloud provider, which also comprises several steps, i.e., adapt the entry point of the function code for the target FaaS system, include all npm packages, zip all external files including the main, and deploy it.

2.4.3. Phase 3: Adapt the monolith
Finally, the goal of the third phase is to adapt the faasified monolith code accordingly. This phase requires to include code that prepares the JSON input for the serverless function, replaces the code block with API call to the serverless function, adds code for monitoring and alternative functions, and reads back the output from the serverless function.

Table 1
Table 1 evaluates the support of the phases defined in the FaaSification life-cycle of several FaaSifiers that we found in the literature, which faasify parts of the monolith as serverless functions. We covered the latest FaaSifiers for the three programming languages, Python, Node.js, and Java, which are most popular among AWS Lambda users. Node2FaaS supports Python monoliths. Node2FaaS and DAF support Node.js monoliths, Termite creates Java functions, while Lambada faasifies Python monoliths. Node2FaaS supports neither package nor code dependencies. If a monolith contains any of these dependencies, the resulting hybrid terminates with a ReferenceError. DAF and M2FaaS are able to handle annotated package and code dependencies. Lambada supplies a dependency traversal functionality, while Termite relies on an external build tool (Maven) to solve such dependencies. Node2FaaS replaces the access of the input values by an access to the corresponding JSON object of the serverless function. Global variables used in the method are not handled by Node2FaaS and result in failures. Also Lambada and Termite correctly transfer input values of the method, but ignore global variables used within the method. DAF does not offer annotation to transfer input variables to the extracted serverless function, while M2FaaS allows to specify such global variables, as well as input variables of the method. All FaaSifiers support a single return value of the serverless function. However, only M2FaaS supports to specify several return values which are automatically merged in the serverless function output.

Several steps are required to deploy the extracted serverless function. All FaaSifiers except Node2FaaS manage to correctly include package and code dependencies in the deployment package. FaaSifiers are able to create and successfully deploy the generated serverless function to the corresponding cloud provider automatically.

Since all FaaSifiers except M2FaaS build serverless functions out of entire monolithic methods, the input values for the serverless function are already prepared as input to the corresponding method which should be ported. Anyway, the input preparation for global variables is missing and if such variables are updated in the serverless function, then the resulting hybrid application is semantically incorrect. DAF is not capable of handling any
input from the monolith. M2FaaS on the other hand manages to prepare all needed inputs to the serverless function. The reading of the serverless function output in JSON format is handled by all FaaSifiers. Anyway, only M2FaaS allows to correctly assign multiple specified output values of a code block without additional development effort. Node2FaaS and DAF do not allow any deployment configuration like maximum function duration or memory assignment of the serverless function. Instead, predefined, hard coded default values are used. Lambda, Termite and M2FaaS allow to add additional configuration. Only M2FaaS supports a fault tolerant execution of the hybrid application, as well as the porting of an arbitrary code block. All other FaaSifiers support only a single serverless function invocation and are able to port only entire methods.

3. **M2FaaS FaaSification**

This section describes the system architecture of M2FaaS including details for the supported annotations to faasifying the Node.js monoliths. With a real example code, we also show how the developer can annotate a code block of the monolith and present the resulting faasified monolith and the serverless function, created by M2FaaS.

### 3.1. M2FaaS system architecture

The system architecture of M2FaaS is presented in Fig. 3. M2FaaS requires a pre-phase for FaaSification where developers need to annotate code blocks that should be extracted as serverless functions. We present an example where two code blocks are faasified on AWS Lambda as functions \( f_1 \) and \( f_2 \). For each annotated code block, the developer may specify the target FaaS provider and the region where the code should be deployed, including function name, timeout, runtime, and assigned memory. Moreover, the developer may annotate a sequence of alternative functions that should be deployed and invoked in case of failure. For instance, Fig. 3 shows an example that the second annotated block should be additionally deployed as function \( f_2^* \) on IBM, which is invoked if the function \( f_2 \) fails. If \( f_2^* \) fails as well, or if network is completely down and no provider can be reached, M2FaaS would run the original monolithic code locally. Such an annotated monolith is then used as input to M2FaaS, which does the FaaSification process.

M2FaaS consists of four FaaSification modules. The parser analyzes the annotated monolith and creates a request to the builder for each annotated code block. The builder then determines all dependencies, adapts the extracted code, and creates a deployment package for each annotated code block for the specified cloud provider. Once all deployment packages are created, the deployer deploys all serverless functions to the specified regions and cloud providers. The current prototype of M2FaaS supports two cloud providers, i.e., AWS Lambda and IBM Cloud Functions. After deployment of serverless functions including the alternatives, the adaptor updates the code of the monolith. This adaptation includes code (i) to pass values of variables used by the serverless function, (ii) to monitor the execution of the serverless function and subsequent execution of alternatives in case of a failure, and (iii) to return the values of variables that were updated by the serverless function and are used later in the monolith. The monitoring and invocation of the resulting serverless functions is managed by the m2faasInvoker file which is included in the faasified monolith. Finally, the adaptor updates the annotated code with the API call to the specified serverless function.

The output of the adaptor is the hybrid application which may be executed in the same runtime environment as the original monolith, but runs the annotated code as serverless functions which is transparent to the user. Also the invocations of alternative serverless functions are transparent to the user that runs the hybrid application.

![Fig. 3. M2FaaS system architecture. In the given example, the developer annotated two code blocks to be faasified as serverless functions \( f_1 \) (red circle) and \( f_2 \) (blue circles) on AWS Lambda, specifying also an alternative serverless function \( f_2^* \) on IBM, which is invoked if \( f_2 \) fails. After the annotation, M2FaaS automatically parses the annotations and for each code block builds and deploys the equivalent serverless functions \( f_1 \) and \( f_2 \) on AWS, while \( f_2^* \) on IBM. Additionally, M2FaaS adapts the annotated code blocks with the API call to the corresponding serverless function (marked with squares).](image-url)

### 3.2. M2FaaS annotation constructs

M2FaaS offers a rich set of annotations to developers to automatically extract, build, and deploy serverless functions from a Node.js monolith and adapt the monolith to run the extracted serverless functions transparently. Introduced annotations automate all phases of FaaSification presented in Section 2. M2FaaS introduces annotations per code block. All annotations are specified as a comment, which does not change the semantics of the annotated code blocks.

We classify the annotations into three groups, i.e., (i) **M2F interface**, which specifies all necessary dependencies that the extracted serverless function needs to run properly, (ii) **F2M interface**, which specifies all dependencies that the faasified monolith needs back from the serverless function, and (iii) **Deployment and Runtime**, which configures the parameters for deployment and monitoring and runs alternative serverless functions in case of a failure.

**Table 2** describes the constructs that the current prototype of M2FaaS supports for automated FaaSification of a code block. The first two constructs (/ / cfun and / / cfunend) are used to annotate the start and the end of the code block that should be extracted as a serverless function. The entire code in between will be extracted as a serverless function and replaced with the API call for the deployed function.

#### 3.2.1. Annotation constructs for the M2F interface

The M2F interface is managed with three annotation constructs described below. By specifying a comma separated list of variables in `vars()`, the developer configures the variables that are used in the annotated code. With this annotation construct, M2FaaS passes the values of all annotated variables as data input to the API call of the serverless function and configures the code of the serverless function to read them in the corresponding variable names. M2FaaS uses the same annotation for both, local and global variables. Package dependencies are managed with the construct `install()` with which the developer may specify the modules that need to be automatically included in the package.json. To add them also in the deployment package (in the corresponding node_modules folder), the developer need to add a comma separated list of package dependencies within the construct `require()`. Code dependency is managed also with
the annotation construct `require()`, in which the developer appends a comma separated list of the external files that should be included in the deployment package. All of these dependencies are compressed with the serverless function code which simplifies deployment. We are aware that the manual specification of annotations may be a very error-prone and time-consuming manual task for the developer and we discuss it in Section 5.4 and specified it as our future work.

3.2.2. Annotation constructs for the F2M interface

The `assign()` annotation construct supports the automation of the F2M interface. The developer needs to specify a comma separated list of all variables whose values are updated within the annotated code block and are used later in the monolith, regardless whether they are declared or not before the annotated code block. M2FaaS automatically returns their values as an output of the serverless function and assigns their values in the corresponding variables in the faasified monolith after the API call of the serverless function. Unlike related work, which mainly supports a single variable as an output of the serverless function, M2FaaS supports an arbitrary number of output variables with arbitrary variable names. With this approach, to the best of our knowledge, M2FaaS is the first FaaSifier that focuses not only on the serverless function to run properly, but also on the faasified monolith after the serverless function finishes and returns its output.

3.2.3. Annotation constructs for deployment and runtime

Finally, the developer may use the `deploy()` annotation construct to configure the characteristics for deployment of the extracted code block. Inside this construct, the developer can use an array to specify where and how the extracted code will be deployed. Unlike the related work, where the default settings are used for deployment, M2FaaS offers two innovations. Firstly, in order to support fault tolerance, the developer may specify multiple deployments of the code block, including the specification of a region of a cloud provider (AWS or IBM). For example, a serverless function may fail and return “Internal server error” or, if there are too many requests, “Too Many Requests Exception: Rate Exceeded” and the FaaS provider does not accept additional requests before some of the existing ones finishes. M2FaaS configures the faasified monolith to run the first specified alternative serverless function in the construct `deploy()`. If that serverless function fails, the next alternative serverless function of the array is executed. We selected this iterative approach to invoke alternative functions because invoking all alternatives concurrently would cause higher costs. This process is repeated until one function finishes successfully or until all functions from the list fail. Secondly, for each alternative serverless function, the developer may configure various characteristics for deployment, such as `memorySize`, `runtime`, and `timeout`. All annotations constructs of the Deployment and Runtime class are described in Table 3.

3.3. M2FaaS FaaSification example

With a real example that contains code, package, and data dependencies, we show how a Node.js monolith should be annotated and present the resulting hybrid application comprising faasified monolith and the extracted serverless function.

3.3.1. Before FaaSification

To start with FaaSification, a developer needs to annotate the code block that should be extracted as a serverless function. We use the example presented in Listing 1 to show the annotation of a concrete real-world application used in industry. Within this application, the code block between lines 22 and 24 is faasified, while the corresponding annotation is at line 21. Due to the package dependency `@cloudant/cloudant` at line 22, both annotation constructs `install()` and `require()` are needed in order to include the package cloudant in the deployment package and make it accessible by the serverless function. The code block accesses the global variable named `PASSENGER_DB` and two local variables `id` and `area`. Such usage of variables declared outside of the annotated code block is represented within the `vars()` annotation construct. The accessibility of the external files `utils.js` and `config.js` (code dependency) is specified with the `require()` annotation construct. The area and passenger variables are accessed at line 29 after the annotated code block, which requires the `assign()` annotation construct to return their values back from the serverless function. As specified in the `deploy()` annotation, the code block will be deployed on AWS region `us-east-1` as a serverless function `readGPSAWS` and permissions are granted according to the provided AWS role.

---

**Table 2**

<table>
<thead>
<tr>
<th>M2FaaS annotation constructs to specify the code block and its properties for automated FaaSification.</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>// cfun</code></td>
</tr>
<tr>
<td><code>// cfunend</code></td>
</tr>
<tr>
<td><code>vars()</code></td>
</tr>
<tr>
<td><code>require()</code></td>
</tr>
<tr>
<td><code>install()</code></td>
</tr>
<tr>
<td><code>assign()</code></td>
</tr>
<tr>
<td><code>deploy()</code></td>
</tr>
</tbody>
</table>

**Table 3**

<table>
<thead>
<tr>
<th>Annotation constructs to specify the configuration of deployment and fault tolerance. The developer may specify multiple providers, regions, and assigned memory for a code block that will be deployed and invoked subsequently in case of a failure. The annotations for name and provider, as well as role for AWS Lambda are mandatory, while <code>M2FaaS</code> uses default values of <code>128MB</code>, <code>30s</code>, and <code>Node.js</code> runtime (version <code>14</code> for AWS and <code>12</code> for IBM) for parameters <code>memory</code>, <code>timeout</code>, and <code>runtime</code>, respectively. For the parameter region, the default settings on the local machine are used for the corresponding FaaS provider.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>Provider</td>
</tr>
<tr>
<td>Region</td>
</tr>
<tr>
<td>MemorySize</td>
</tr>
<tr>
<td>Runtime</td>
</tr>
<tr>
<td>Timeout</td>
</tr>
<tr>
<td>Role</td>
</tr>
</tbody>
</table>
The developer does not specify, the function `readGPSAWS` will be deployed with a memory assignment of 128 MB, a timeout of 30 s and Node.js runtime with version 14. In addition, the extracted code is deployed as an alternative function on IBM’s region eu-gb. Unlike the related FaaSifiers, which use default values to deploy functions on a single FaaS provider at a time, we offer dynamic configuration for deployment. Nevertheless, the developer may omit region, memorySize, runtime, and timeout, in which case `M2FaaS` will use default values. All other annotations are needed to automatize FaaSification. Otherwise, as described in Section 2, the serverless function and the faasified monolith may fail.

3.3.2. After FaaSification

After FaaSification, the generated serverless function is presented in Listing 2. The lines 22–24 from the annotated monolith are adapted as lines 5–16 in Listing 3. The rest of the monolith stays exactly the same as before the transformation.

All code and package dependencies annotated with the constructs `require()` and `install()` are included in the resulting serverless function `readGPSAWS` of Listing 2 at lines 1–3, while the global and local variables are made accessible at lines 6–8. The variable area must be either declared in the serverless function, or made as an input to the serverless function. Otherwise, the assignment of the variable at line 12 would declare a new variable in the scope of the `if`-construct, which would make the return invalid. The actually ported code block is at lines 10–12 and the values which should be accessible after the annotated code block specified with the `assign` annotation, are returned at lines 14–17.

The hybrid application, presented in Listing 3, replaces the annotated code block with an invocation to the corresponding serverless functions. Listing 3 presents only the replaced code. Line 6 invokes the serverless function, while providing the input. `m2faasInvoker` handles the fault tolerant invocation of the provided serverless function. In this example, the hybrid application runs the serverless function `readGPSAWS` on AWS region `us-east-1` and if it fails, then the serverless function `readGPS` on IBM region `eu-gb` is invoked. If this function fails as well, i.e. if all alternative functions fail, `M2FaaS` executes the monolithic code (line 8–15). The resulting variables specified in the `assign` annotation are updated or declared at lines 18 and 19.

4. Evaluation

We conducted several experiments to evaluate and compare FaaSification of `M2FaaS` with two state-of-the-art Node.js FaaSifiers `Node2FaaS` and DAF. The evaluation considered the performance of the hybrid application and development effort needed for FaaSification. In this section we elaborate the testing methodology in detail and discuss the results of the evaluation.

4.1. Testing methodology

This section details the conducted experiments and used benchmark applications and environment setup in order to make our evaluation reproducible.

4.1.1. Test data/experiments

We conduct two classes of experiments to evaluate `M2FaaS` with the related work. The first class evaluates the development effort needed to faasify a monolith. For this purpose, we measure
S. Pedratscher, S. Ristov and T. Fahringer Future Generation Computer Systems 135 (2022) 57–71

4.1.2. Benchmark applications

The second class of experiments investigates performance of the resulting hybrid application generated with all three FaaSifiers. In this class of experiments, we measure the average submission time and round trip time (RTT), throughput, and failure rate when the hybrid application invokes a load of 1500 concurrent serverless functions.

4.1.3. Testing environment

The generated hybrid applications of the benchmark applications were hosted on a t2.large instance of AWS Elastic Computing (EC2) in us-east-1 with 8 GB RAM, 2 vCPUs and Amazon Linux 2 AMI (HVM) as operating system. Each faasified hybrid monolith runs on an EC2 instance, which invokes the corresponding serverless functions. The GCA hybrid application is executed once on the same t2.large instance in order to confirm its correctness. We repeated each experiment with CPU, memory, and I/O intensive tasks 5 times and considered average values in the evaluation, similar as done in recent work [18].

All generated serverless functions of both classes of experiments were deployed in AWS North Virginia region (us-east-1). In addition, M2FaaS was configured to deploy alternative serverless functions on IBM London (eu-gh) and AWS Frankfurt (eu-central-1) for the intensive tasks. In all cases the memory assignment of 128 MB and a default timeout of 30 s was set to be consistent with the maximum duration of Node2FaaS serverless functions which were invoked with HTTP GET requests. In order to simulate a world-wide distributed application on three continents, we deployed the database to gather information about the passengers of the GCA monolith on IBM Cloudant in Sydney.

4.1.4. Fair comparison

Due to some specific requirements in the related work FaaSifiers, we had to perform some manual adaptations in the original monolith, faasified monolith, and the generated code of serverless functions in order to be able to run FaaSification successfully
and generate a hybrid application that runs properly. For a fair comparison regarding the development effort, we neglected the time to analyze and conduct these changes in the evaluation. The following paragraphs describe these changes.

Node2FaaS aims to automatically detect which serverless function is eligible for the conversion. The process of detecting the eligibility is simple: either the method in the monolith exceeds the size of 2200 characters or the method contains a loop including the method input parameter. Since with this approach Node2FaaS does not faasify all methods, we decided to manually add comments in both benchmark applications to exceed the limit of 2200 characters. Another issue with Node2FaaS which resulted in a failure “SyntaxError: Unexpected identifier” was the async keyword. A manual movement of the async keyword to the correct spot solved the issue. Finally, Node2FaaS requires in addition a variable named result which represents the return value of the serverless function. Since the monolith does not contain a variable result, we manually added this variable in the monolith to prevent a failure of the generated serverless function (“result is not defined”).

Similar adaptations were needed to run Faasification with DAF. Due to the “/” character in the cloudant package (see @cloudant/cloudant in line 2 in Listing 1), DAF was unable to automatically add the package in the deployment package of the serverless function, although DAF supports package dependency. DAF failed with an error message “no such file or directory”, because the slash was interpreted as path to a local internal code dependency file. In order to solve this issue, we removed the package dependency, generated the serverless function and manually added the package dependency in the node_modules and index.js.

### 4.2. Development effort evaluation

In Table 1, we evaluated five Faasifiers including M2FaaS for all three phases. However, only DAF and Node2FaaS can faasify Node.js monoliths, while Termite is a Faasifier for monoliths written in Java, while Lambda for Python monoliths. Therefore, we were able to evaluate M2FaaS with the Node.js Faasifiers DAF and Node2FaaS. Table 4 presents the development effort to faasify the realistic GCA monolith. Presented time for Node2FaaS was spent mainly for manual development and code and package dependency preparation, for DAF partially for manual development and annotation, and for M2FaaS, developers needed to spend time on annotation only.

The automatic definition and inclusion of the @cloudant package in the serverless function is supported by DAF and M2FaaS. Even if DAF failed in this specific scenario, in general the inclusion of such dependencies is supported. In both cases, only the according annotation was required, which took approximately 2 min to detect the error and add the dependency. In the serverless function ported by Node2FaaS the definition of the package dependency needed to be done manually, which took approximately 5 min. In addition, the insertion of the module in the deployment package and redeployment of the serverless function took additional 10 min. The code block of the GCA monolith that should be faasified contains two code dependencies. DAF and M2FaaS support the definition and automatic inclusion of both dependencies with a development effort of 2 min per code dependency annotation, while the serverless function ported by Node2FaaS required to manually add the definition and code and package dependency files, which took approximately 10 min and 20 min, respectively. The undefined global variable used in the code block was manually added to the serverless functions ported by Node2FaaS and DAF. In both cases the searching, inclusion and redeployment took approximately 5 min. M2FaaS automatically included the global variable with an appropriate annotation and needed approximately 2 min to annotate it. Local variables are automatically ported by Node2FaaS and do not require any additional development effort. In the serverless function ported by DAF both variables have to be added manually, which took 4 min, while Node2FaaS needed also 4 min to find and annotate the variables for M2FaaS. To correctly return and reuse both variables needed after the code block, merging of both values into a single JSON object was required for Node2FaaS and DAF. In addition, Node2FaaS required the renaming of the result variable in order to be recognizable, which took approximately 10 min. With M2FaaS it was sufficient to add an annotation including both return values, which took approximately 4 min for both variables. Based on the overall time for successful Faasification of the GCA monolith, we observe that M2FaaS reduces the development effort by more than forty minutes or 73.3% compared to Node2FaaS and by 48.4% compared to DAF.

The development effort to deploy serverless functions on additional locations is not considered within Table 4 because this feature is only supported by M2FaaS. In general such additional deployment requires changes in the hybrid application in order to execute alternative serverless functions on failures, as well as redeployment of serverless functions on other regions or cloud providers. The evaluation from Table 4 reported significant reduction in development effort for the realistic but still simple GCA monolith that contains a few dependencies of each type (code, package, and data). Also, the code block that is appropriate to be extracted as a serverless function comprises several lines of code with a few variables.

In general, Faasification needs to handle the following challenges: package dependencies, code dependencies, and data dependencies represented as input variables in the M2F interface and return variables in the F2M interface. Based on this, in (1) we define the Total Faasification Time TFT as a sum of three times to solve all three challenges.

\[
TFT = n_p \cdot p + n_v \cdot v + n_r \cdot r
\]

where \(n_p\), \(n_v\), and \(n_r\) denote the number of package and code dependencies, input and global variables, and return variables, respectively. All these three variables are affected by the monolith. The corresponding multipliers \(p\), \(v\), and \(r\) on the other side, denote the average time to handle each corresponding challenge and they are determined by the Faasifier that is used.

Let us analyze Eq. (1) for real life applications.

Based on our evaluation in Table 4, we observed that code and package dependencies are the most complex ones since they require two activities, to define each dependency and to insert it in the deployment package of the function. Node2FaaS does not offer support for these dependencies and developers have to solve them manually. Based on Table 4 we observe that, for Node2FaaS, \(p_{n2f} = 15\) (5 for definition + 10 for insertion) and \(v_{n2f} = r_{n2f} = 5\).
In addition, we observe that global, local, and return variables are the most challenging ones for DAF, leading to $\nu_{\text{def}} = \tau_{\text{def}} = 5$, while $p_{\text{def}} = 2$. For M2FaaS we observed $p_{\text{def}} = \nu_{\text{def}} = \tau_{\text{def}} = 2$.

On average, a real project has $n_p = 200$ package dependencies, reaching 1000 and above in some cases [14]. For $n_v$ and $n_r$, we use the work from Grechanik et al. [19] who inspected over two thousand applications. They reported that a project has on average 97 classes and each class on average 2.5 methods, giving a total of 242.5 methods per project. If we assume that only 10% of the methods are eligible to be faasified, then a total of 24 methods should be ported as serverless functions. From these methods 42% do not take any argument, while for the others on average 1.5 arguments per method are used, resulting in $n_v = 21$. From all methods 56% have a return value, resulting in $n_r = 13.6$. Based on the above analysis, we can estimate $TFT_{\text{def}} = 52.9$ h and $TFT_{\text{def}} = 9.59$ h. However, if one uses M2FaaS, then $TFT_{\text{def}} = 7.82$ h significantly reduces. This means that the automated FaaSification with M2FaaS reduces the development effort by 85.21% compared to N2F and 18.12% compared to DAF. In general the percentages for improvement represent a lower bound of possible development effort gains, since we did not consider the annotations of arbitrary code blocks, which introduces the complexity of variables defined outside and used after the code block. Especially the development effort of DAF would be strongly worsened.

The evaluation in Table 4 assumes that the developers posses expertise in code, data, and package dependencies. Further on, developers need to understand all three phases of the FaaSification. Non-experienced developers, on the other side, would need not only more time for each activity, but also time to repeat multiple times the deployment and running the generated functions. Moreover, the development effort may be affected by the lines of code of the monolith, especially to determine which variables should be passed to the functions and which variables should the function return. Nevertheless, this process can be further automatized, which we set as our future work.

### 4.3. Performance evaluation

Further on, we evaluated the performance of the three FaaSifiers (DAF, Node2FaaS, and M2FaaS). In particular, we investigated submission time delay, finish time, and throughput, as well as support for fault tolerance and the trade-off in terms of higher RTT.

#### 4.3.1. Submission time

Figs. 4–6 represents the submission time of 1500 function invocations for the CPU, memory and I/O intensive tasks, respectively. Despite the concurrent invocation from the same region, we observe that all faasified monoliths need considerable time in seconds to invoke 1500 serverless functions. M2FaaS and DAF needed similar amount of up to 8.2 s for all three benchmark tasks, while we observed unexpected behavior of Node2FaaS. Firstly, it needed considerable longer time period compared to M2FaaS and DAF to invoke 1500 functions. Secondly, although submission time should not depend on the task, Node2FaaS needed variable time period of 10.5 s for the I/O intensive task up to 18.4 s for the CPU intensive task. Another observation is the linear increase in the delay for the invocation for DAF’s and M2FaaS’ monoliths by 5.5 ms, while the invocation delay of Node2FaaS faasified monolith is unstable and on average is up to 12.3 ms for the CPU intensive task. After revision of the resulting monoliths, we determined that Node2FaaS adapts the faasified monolith to invoke serverless functions by using HTTP GET requests, while M2FaaS and DAF invoke serverless functions via the AWS SDK. We believe that AWS optimized its AWS SDK compared to the HTTP GET requests. Further investigation of this insight is beyond the scope of this manuscript.

#### 4.3.2. Fault tolerance

Once a serverless function is invoked, it can either successfully return or fail. The main reason to fail is the limitation of AWS Lambda for the maximum number of concurrent invocations. Based on the specification of the inputs for all three intensive tasks, the reported results were mainly as expected. Table 5 shows the average number of failures for all three intensive tasks. For the CPU intensive task we observe that all three FaaSifiers reported a huge number of failed serverless functions because their duration is longer than the time needed to submit the limitation of 1000 serverless functions. Node2FaaS generated even more than expected 500 failures, while DAF’s hybrid application reported similar number of failures as expected. For the CPU intensive task, M2FaaS generated failure rate of 27.18% which is by 14.9% lower compared to DAF and 34.4% smaller than Node2FaaS. More important, the failures for the CPU intensive task caused the hybrid application generated by Node2FaaS and DAF to fail.

On the other side, due to the support for fault tolerance and multiple alternatives, M2FaaS runs on average 422.5 alternative serverless functions deployed on IBM London, which finish all since the concurrency limit of IBM is also 1000. With this support
Table 5
Average amount of failures of each state-of-the-art faasifier for various intensive tasks.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Tool</th>
<th>Failures</th>
<th>Tolerated failures</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>M2FaaS</td>
<td>422.5</td>
<td>422.5</td>
</tr>
<tr>
<td></td>
<td>DAF</td>
<td>465.6</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>N2F</td>
<td>604.4</td>
<td>0</td>
</tr>
<tr>
<td>MEMORY</td>
<td>N2F</td>
<td>219</td>
<td>0</td>
</tr>
<tr>
<td>I/O</td>
<td>N2F</td>
<td>45.2</td>
<td>0</td>
</tr>
</tbody>
</table>

Fig. 7. Average Finish Time for CPU, memory, and I/O intensive tasks faasified with M2FaaS, DAF, and Node2FaaS.

for fault tolerance, all 422.5 failures are tolerated and handled without crashing the hybrid application, without the need to run the functions locally. DAF and M2FaaS do not report any failure for the other two intensive tasks (memory and I/O), but Node2FaaS does. We explain this with the same reason as for the submission time, i.e., the HTTP GET requests sent by Node2FaaS, compared to AWS SDK used by M2FaaS and DAF.

4.3.3. Finish time and throughput
Fig. 7 represents the average finish time of all 1500 serverless functions for the CPU, memory and I/O intensive tasks. The finish time represents the time period from the invocation of the first function until receiving the responses of all 1500 functions. In all cases the finish time of Node2FaaS is the highest due to the longer submit time and the largest RTT of the serverless functions. For the CPU intensive task Node2FaaS hybrid application take 3.09 times more time than M2FaaS to finish the serverless functions, while DAF’s hybrid shows the fastest finish time of 28.14s, 18% faster than M2FaaS. The submission times of DAF and M2FaaS are nearly identical, but the average RTT of M2FaaS is higher than the one from DAF. The reason for this is the alternative invocation of failed serverless functions, as shown in Section 4.3.2.

Nevertheless, as shown in Fig. 8, M2FaaS achieved 18.5% higher throughput compared to DAF, because DAF does not invoke alternatives for failed serverless functions. Node2FaaS achieved 5.18 times lower throughput compared to M2FaaS in the CPU intensive task. We observed similar results in finish time and throughput for M2FaaS and DAF in the memory and I/O intensive tasks. All results are within 4% and no serverless function failed for these two Faasifiers, while in Node2FaaS we observed a reduction of the throughput by 457% and 273% compared to M2FaaS for the memory and I/O intensive task.

5. Discussion and related work
This section discusses the related work in terms of decomposition of monoliths to microservices and various existing Faasifiers in various programming languages. We also discuss other programming languages that use annotations. Finally, we discuss several limitations of M2FaaS.

5.1. Code offloading decision
One approach to offload parts of the code is to create a service, e.g., a microservice, deploy the code, and simply call it when needed. Several works exist to decompose monoliths into a set
of microservices. The work from Eski et al. [20] proposes an approach to transform existing applications into microservices using code repositories. Evolutionary and static code coupling information, as well as the graph clustering methodology is used to automatically extract microservices from monoliths. The accuracy of the extraction results could be increased with a more advanced graph clustering algorithm. Chen et al. [21] presents a semi-automatic dataflow-driven decomposition algorithm in order to identify parts of the monolith suitable for the conversion to microservices. The mechanism requires a manual construction of a data flow diagram in order to illustrate the detailed dataflow of the application. This diagram is then automatically converted to a decomposable diagram and finally microservice candidates are identified from that. This process requires manual effort for the generation of the data flow diagram and additionally the obtained microservice candidates could still need expert judgement before being deployed in practice.

The above-mentioned techniques help practitioners to decompose monoliths into a set of microservices. However, this approach may improve the overall performance, but generates costs for running the server on which the service is hosted. With FaaSification, costs may be significantly reduced [22] because users are charged only when the function is actually executed.

Node2FaaS [11] analyzes the code length of methods and presence of input variables in loops in order to offload it as a separate serverless function. Node2FaaS does not check if a selected method is self-contained and this may lead to a failure during runtime causing the entire hybrid application to fail.

5.2. FaaSifiers (decompose monoliths to serverless functions)

Several researchers introduced various FaaSifiers to decompose monoliths to serverless functions. Due to specifics of each programming language, each FaaSifier support a single programming language only. Node2FaaS [11] is a recent FaaSifier that automates the conversion of entire Node.js monolith methods to serverless functions, while the body of each faasified method is replaced with a corresponding API call to the target cloud provider. However, Node2FaaS exhibits several weaknesses. One downside is the incapability of resolving code and package dependencies, meaning that only fully self-contained methods which do not access any outer method, library or variable are faasified correctly. Methods which are not self-contained would have to be checked for correctness and manually re-structured in order to make the resulting serverless function valid. This step takes usually a lot of time and is not viable in most cases, but it is required for the resulting hybrid application to run without failures.

Another approach to faasify Node.js monoliths is with the Dependency-Aware FaaSifier (DAF), introduced by Ristov et al. [9]. DAF mainly focuses on resolving code and package dependencies and automatically includes them in resulting serverless functions. Still, DAF does not automate data dependencies, which may cause that the resulting hybrid application fails. Yussupov et al. [23] introduced another method for FaaSification. Instead of reengineering the monolith as in M2FaaS, they introduced the concept of serverless parachutes, which duplicates the main functionalities of the monolith. Finally, after FaaSification, a proxy, which is decoupled from the monolith, may offload the load to the serverless functions that are extracted from the annotated code. However, similar to Node2FaaS, serverless parachutes are faasified from the whole methods or services and their approach is unable to faasify arbitrary code blocks.

FaaSification of a single Python method is supported by LambdaTermite [16] port existing Java monolith methods as serverless functions. However, multiple restrictions appear on the input Java Project. The above-mentioned FaaSifiers are limited in terms of references and namespaces, as well as supporting only one cloud provider to deploy the serverless function on. PyWren [26] is another approach to port Python code as serverless functions. PyWren does not deploy the method code directly, but both, the code and the data input are fetched from the storage by a single executor function. Additional developer effort is required in order to handle multiple dependencies and additional runtime overhead is generated for fetching the code and the input from the executor function. FuncX [27] runs Python functions within containers, which improves the portability and eliminates cloud provider specific limitations like maximum function duration. However, FuncX limits users in terms of multiple allocations and it suffers from higher startup latency compared to widely used cloud providers.

To the best of our knowledge, there is no FaaSifier that supports FaaSification of a code block and a fault tolerant execution of the resulting hybrid application. While many cloud providers offer a reactive fault tolerance by retrying serverless functions that fail and return exceptions with try-catch, none of them supports running alternative serverless functions on another region of the same or even another cloud provider.

5.3. Code block offloading with annotations

Annotations are widely used in HPC to specify sections that should be parallelized. For example, OpenMP [28] uses pragma to annotate parallel sections or parallel loops of a C program which will be executed across multiple threads. Further on, OpenACC [29] uses similar annotation to specify the parallelization to be executed on GPUs. Moreover, annotations may be used to help the compiler how to optimize the data distribution between the CPUs and GPUs, e.g., to avoid copying data back to the CPUs which is not needed. Very similar as our approach to annotate variables that are needed for the rest of the monolith, OpenMP and OpenACC use annotations to specify reduction operations that the should be performed after parallelization so that the parallel execution returns the same output as the sequential one. Programming languages like Java [30] offer annotations to suppress warnings, to override methods or to represent other custom package annotations, e.g., from Jackson [31], which are suitable for data-processing.

Recently, several works used annotations for serverless functions. DAF [9] uses annotations to mark the code and package dependencies that should be zipped when building the deployment package for the resulting serverless function. Yussupov et al. [23] allow developers to annotate functionalities that should be created as serverless functions.

5.4. M2FaaS limitations

Similar as all related FaaSifiers, M2FaaS is limited to a single programming language because each programming language handles dependencies and builds deployment packages differently. In our evaluation, we consider the concurrency limitation. However, there are different other limitations, such as zipped and unzipped code size of serverless functions, as well as their input and output data size. The current M2FaaS prototype assumes that the generated serverless functions satisfy at least AWS Lambda or IBM limitations. Nevertheless, M2FaaS fault tolerance tolerates failures of a FaaS provider.

Despite the automated FaaSification, M2FaaS requires some developer effort to annotate the code block and supported dependencies. The annotation need to be performed manually by
the developer. In general, many of the annotations could be automatically generated with various code transformation tools. For example, the needed dependencies could be derived from the source code. Spoon [32] is able to detect various dependencies, code structure and return statements of Java files. Esprima [33] allows to tokenize and parse Node.js source code and extract information like variable declarations or dependency definitions. Integrating such tools into M2FaaS was beyond the scope of this paper, which focuses on the automation of the steps of FaaSification.

M2FaaS offers dynamic configuration of memory that should be assigned to the serverless functions. However, developers are usually unaware of the memory requirements for the annotated code block because memory requirements may depend on input data during runtime. On the other side, some FaaS providers claim that they scale CPUs together with memory. As a consequence, developers may need to test multiple memory settings on various FaaS providers to determine the optimal memory settings for each function.

In monolithic architectural style, accesses to global variables are coordinated by using shared memory within the application. However, FaaS architectural style differs because each function has isolated runtime environment and data segment. In order to manage data dependencies, the current M2FaaS prototype passes global variables by values to the serverless functions without support for synchronization. Therefore, the monolith first needs to be decompiled, e.g., using some approaches of microservices presented in Section 5.1, after which, FaaSification will be straightforward.

6. Conclusion and future work

This paper introduces M2FaaS, a novel FaaSifier for monolithic Node.js applications that automates FaaSification of arbitrary code blocks using simple annotation constructs that do not change the semantics of the monolith. M2FaaS is the first FaaSifier that introduces FaaSification to extract a code block as a serverless function considering not only all package and code dependencies, but also the data dependencies. With the latter, M2FaaS’s FaaSification offers support for the semantic correctness of the entire hybrid application after FaaSification, that is, both for the serverless functions and the faasified monolith. M2FaaS can be used to deploy the extracted code as serverless functions on multiple regions for different providers with various configuration parameters. An alternative serverless function is invoked if the previous serverless function fails.

Experiments that compare M2FaaS with two recent FaaSifiers for Node.js monoliths demonstrated that M2FaaS reduces the number of failed serverless functions by 14.9% and tolerates all failures which is fully transparent to the user that runs the hybrid application. The trade-off for an improved fault tolerance behavior is an increase of the average round trip time of functions by 25.6% when 1/3 of the functions failed and only up to 4% when all functions finish successfully.

M2FaaS deploys annotated code assuming that the serverless functions are not deployed. We will extend M2FaaS to allow versioning of created serverless functions. In addition, we will research methods for how to automate the first stage in order to automatically fill and generate annotations which can be extracted from the source code.

Another issue that has potential for extension is fault tolerance. We did the initial step with fault tolerance by offering higher level of fault tolerance than each provider individually. Anyway this could be extended with more advanced mechanisms, such as simultaneous invocation of multiple alternative functions to achieve higher reliability.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References


Stefan Pedratscher received his Master in computer science in 2021 from the University of Innsbruck, Austria. He is a Ph.D. student and his main research interests include serverless computing in edge-cloud continuum.

Sasko Ristov received his Ph.D. in computer science in 2012 from University of Ss. Cyril and Methodius, Skopje, Republic of North Macedonia (UKIM) and was Assistant Professor at UKIM until 2017. Since 2016 he is a postdoctoral university assistant at University of Innsbruck. Dr. Ristov authored more than 100 research articles in the areas of performance modeling, optimization and parallel and distributed systems. He received the UKIM’s Best Scientist award in 2012 and an IEEE conference best paper award in 2013.

Thomas Fahringer received his Ph.D. degree from the Vienna University of Technology in 1993. Since 2003, he has been a full professor of computer science at the Institute of Computer Science, University of Innsbruck, Austria. His main research interests include software architectures, programming paradigms, compiler technology, performance analysis, and prediction for parallel and distributed systems. He is a member of the IEEE.