Received: 5 September 2022 | Revised: 12 July 2023 | Accepted: 19 August 2023

DOI: 10.1002/spe.3263

RESEARCH ARTICLE

WILEY

Migrating monolithic applications to function as a service

Hendrik M. Würz[1,2] | Michel Krämer[1,2] | Marvin Kaster[1,2] | Arjan Kuijper[1,2]

[1]Fraunhofer Institute for Computer Graphics Research IGD, Darmstadt, Germany

[2]Department of Computer Science, Technical University of Darmstadt, Darmstadt, Germany

Correspondence
Hendrik M. Würz, Fraunhofer Institute for Computer Graphics Research IGD, Fraunhoferstr. 5, 64283 Darmstadt, Germany.
Email: hendrik.martin.wuerz@igd.fraunhofer.de

Summary

Function as a service (FaaS) promises low operating costs, reduced complexity, and good application performance. However, it is still an open question how to migrate monolithic applications to FaaS. In this paper, we present a guideline for software designers to split monolithic applications into smaller functions that can be executed in a FaaS environment. This enables independent scaling of individual parts of the application. Our approach consists of three steps: We first identify the main tasks (and their subtasks) of the application to split. Then, we define the program flow to be able to tell which application tasks can be converted to functions and how they interact with each other. In the final step, we specify actual functions and possibly merge those that are too small and which would produce too much communication overhead or maintenance effort. Compared to existing work, our approach applies to applications of any size and results in functions that are small enough—but not too small—for efficient execution in a FaaS environment. We evaluate the usefulness of our approach by applying it to a real-world application for the storage of geospatial data. We describe the experiences made and finish the paper with a discussion, conclusions, and ideas for future work.

KEYWORDS

architectural design, cloud computing, design patterns, function as a service, migration

1 | INTRODUCTION

In recent years, the global amount of data has increased dramatically. Images have become larger, models became finer and updates are made more frequently. All of these data has to be processed and the underlying software must grow with its requirements.

Just a few years ago, companies operated their own servers. This gave them exclusive access to the machines but was not flexible enough to handle high variations in load. As a result, applications were moved to the cloud. Today, the large amount of data requires even more flexible processing. Applications become more powerful, support more operations, and have to run faster. At the same time, they should be easy to maintain and inexpensive to operate.

Abbreviations: EJB, Enterprise Java Beans; FaaS, Function as a Service; IoT, Internet of Things; WMS, Workflow Management Systems.

These requirements have led cloud providers to introduce a new processing concept called *Function as a Service (FaaS)*. In contrast to a monolithic software architecture, FaaS comprises many small functions that act in concert as a common distributed application. Developers only write function bodies and do not have to deal with the operation of servers, virtual machines, or even containers as this is managed by the cloud provider. They can focus on a single function without the need to look at the whole system. This reduces complexity and may lead to increased maintainability, testability, and reusability.

For cloud providers, each function is independent. They can scale it when required and execute it wherever capacity is available. As the run time of a function is short, they can quickly change the data center if capacity becomes tight. This enables the cloud provider to increase resource utilization and offer lower prices. If no data needs to be processed, no function is running and therefore no costs incur.

Therefore, Function as a Service promises to provide benefits that make it an interesting concept for modern applications. However, there are many monolithic applications in practical use. They are not written for FaaS and have to be migrated.

As a first step, a monolithic application can be split into only a few functions. This requires less adjustments, and larger parts of the existing code can be reused. However, this might reduce the benefits offered by FaaS. Larger functions cannot be scaled as flexible as smaller ones and might require a longer run time than allowed by the cloud providers.

On the other hand, many fine-granular functions can lead to more communication overhead. Especially if large data is processed, the network can become the bottleneck of the whole application. The problem gets even bigger, if a function fails for any reason and has to be invoked again. If the data is included in the function call, it has to be sent again which results in high traffic. This has to be avoided as it affects the overall performance.

In the end, a good function design is a trade-off. Functions should be small enough to benefit from the advantages of FaaS but not too small to avoid communication overhead. The resulting problem is not easy to solve and requires a reasonable planning.

The thoughts that need to be made when migrating an application to FaaS can be summarized in the following three challenges:

1. Which parts of the application could be functions?
2. How are these parts related to each other?
3. When should two parts of the application be covered in a common function and when in separate ones?

In this paper, we present an approach that solves these challenges. With our contribution we provide a guideline for software designers on how to migrate their applications to FaaS. Our contribution consists of three steps: identifying the tasks of an application, defining the program flow, and finally specifying actual functions. Compared to existing work, our approach applies to applications of any size and results in functions that are small enough (but not too small) to be efficiently executed in a FaaS environment.

The remainder of the paper is structured as follows: We first review existing works related to FaaS in general and how existing applications can be transformed (Section 2). Then, we present our main contribution in Section 3. In Section 4, we apply our approach to an existing application to evaluate its usefulness. Finally, we discuss benefits and drawbacks (Section 5) and finish the paper with conclusions and ideas for future work (Section 6).

2 | RELATED WORK

Leitner et al. analyzed when Function as a Service (FaaS) is used in practice or for what reasons it is not.[1] According to them, FaaS has not found its way into large applications yet. It is mainly used for small applications with up to 10 functions. This includes data or event-driven processing and "glue functions" between larger applications. Well-known examples include the Internet of Things (IoT) or Chatbots.[2] Another field of application is prototyping. Since functions can be written and deployed quickly, they are ideal for trying things out. However, the step towards larger applications is taken very hesitantly. Leitner et al. refer to limitations in appropriate programming models as a reason and point out technical difficulties with FaaS.

There are publications that address these limitations and describe ways to circumvent them. For example, FaaS typically requires stateless function execution but a shared memory can overcome this Reference 3. Another problem is the limited run time, but this limit has increased in recent years and most cloud providers now allow run times between 9 and 15 min. If this is still not enough, Soltani et al. describe an approach to transfer the execution from one function to another, which results in a new time contingent.[4]

Cold starts are another problem but they can be handled quite well too. There are approaches to warm up instances by predicting future function executions[5] or keeping instances alive.[6] Some frameworks have the possibility to keep generic instances running and only inject the specific function code when a request is made.[7]

In summary, many of the technical limitations can be reduced or overcome. However, one major problem remains: Existing applications can not simply be executed as functions in a FaaS setup. They have to be migrated, which leads to the question of how to split them up.

A similar question was addressed by Wu et al.[8] They investigated the use of microservices in the context of edge computing. In contrast to FaaS, other criteria such as energy consumption play a much greater role here. Nevertheless, their work demonstrates the importance of matching software and execution environment.

Modularity is an important part of software development.[9] However, its focus is not on the execution in isolated units but on better maintainability. Even though the code is modular, it is still assembled into a single application for operation. Evans formalizes a modular application architecture in his book domain-driven design.[10] He describes an application as a set of subsystems with their own contexts. The exact modeling of a system should not influence other systems. For this, Evans introduces anti-corruption layers. They translate between two systems, enabling isolation. While Evan's work provides an important contribution to the overall architecture of an application, it is too general to be applied in the context of FaaS.

However, Balalaie et al. use it to define microservices.[11] The individual services should be cut out of an existing application using the context bounds of domain-driven design. They pointed out that a good decomposition can lead to incompatibilities with the existing system. Nevertheless, a perfect setup can only be found if such problems are ignored at first. This is similar to our approach. In Section 3.1, we split the application into as many parts as possible, ignoring the associated effort. The final design is determined later, which means that no possibilities are excluded in advance.

The size of a perfect microservice has been discussed in literature many times. Newman says they should be "small enough and no smaller"[12] and that developers normally have a good sense of what is too big. Once a microservice is no longer perceived as too big, it is good, he says. Another definition by Richardson suggests that microservices should be so large that they can be "developed by a small team".[13] The individual teams should work together as little as possible to ensure a loose coupling of the microservices. Both definitions do not specify a fixed size, but try to steer the development towards isolated units. This goal is different from FaaS, where a short and stateless execution of the individual functions is aimed. As a result, migration rules for microservices can only be used in a limited way for FaaS. Often, the services are still too coarse and must be further subdivided. For example, Newman proposes to leave components that work on the same database table in one microservice. This way, the table can be isolated better, but in FaaS, this would result in functions that are too large. Nevertheless, the concepts for migrating applications to microservices can be a useful step on the way to FaaS functions.

Gysel et al. introduce a rule system for the creation of microservices.[14] Using 16 rules, they recommend which components of an application should be part of a microservice. They introduce a system that processes a given entity relationship model and a definition of use cases and analyzes the dependencies. As with Newman, these rules are designed for microservices and not for short-running functions in FaaS.

Fritzsch et al. compare ten different decomposition methods for microservices and evaluate their suitability in different situations.[15] For this, they classify different approaches according to the required input information. The groupings range from pure source code to UML diagrams or communication measurements. According to them, the aforementioned approach of Gysel et al. should be used if no measurements or version control history are available. However, all evaluated approaches result in an architecture for microservices, which is not suitable for FaaS.

To obtain FaaS functions finer splits must be made. Hamzeh presents a tool that calculates a score for each method in an application as to how suitable it is for outsourcing to a FaaS function.[16] The decision is based on typical FaaS functions that are compared to the code in the application. For example, resetting a password was identified as a typical use case for FaaS functions. The tool scans for keywords fitting to this use case and recommends modeling the associated method as a function.

Spillner et al. convert Java programs directly to FaaS functions.[17] Each class becomes one or multiple functions while the instance variables are moved to function arguments. As long as no extended language features such as dynamic class loading are used, this approach transforms any Java application to FaaS. However, the resulting setup is much slower than the original application. It is only appropriate for small applications. Spillner wrote a similar program to transform Python code into functions.[18] In comparison to the Java version, only selected parts of the application are moved to FaaS. The main execution remains local, and whenever an outsourced part is reached, the function is called. However, similar to the first approach, this one is only suitable for moderately complex Python applications and is accompanied by a significant drop in performance. Both approaches have the advantage that the transformation from code to FaaS is done automatically. However, they also show the limitations of automated approaches: they do not work well for large applications.

To summarize, there are approaches to split large applications, but they result in microservices that are too big for FaaS. Other approaches lead to small functions but are only suitable for small applications. In our work, we fill the gap between these approaches and transform a large application into small functions.

3 | APPROACH

Our approach to migrate a monolithic application to FaaS consists of three steps. First, the *tasks* (and *subtasks*) of the application have to be identified (Section 3.1). After that, the *program flow* through the identified tasks has to be specified (Section 3.2). In a FaaS setup, there is no master component that controls all the others. Instead, the functions are responsible for calling appropriate successors. The program flow can be used to derive which function calls which other one. Finally, the actual functions need to be specified. For this purpose, each task is first considered as an independent function. In some cases, it makes sense to merge two functions together. In other cases, it does not. The rules for which functions should be combined, are presented in Section 3.3.

3.1 | Identifying application tasks

Each application has a reason for existing: it solves one or more tasks. To identify these tasks, we propose a top-down approach. For this, a graph like the one in Figure 1 can be used.

We start with a root node called “application”. Most applications have some kind of main tasks (e.g., “copy a file” or “search a database”). The main tasks are the children of the root node.

Each main task can be further divided. For example, searching for data might require parsing a query, accessing (multiple) data sources, and finally merging the results into a response. These three steps can be considered subtasks.

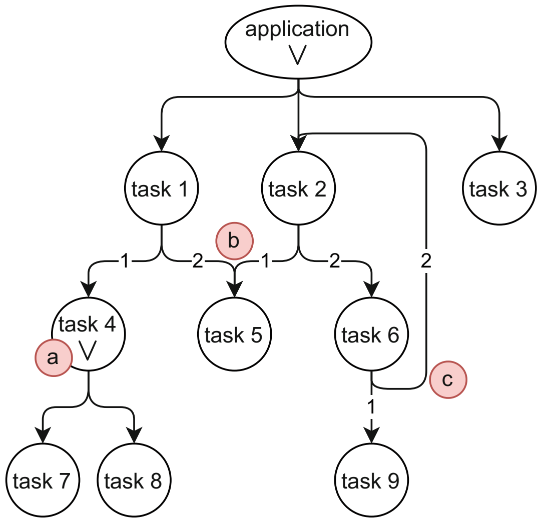

FIGURE 1 An application consists of multiple tasks. They can be further divided and form a graph. To convert an application to FaaS, the individual tasks of the application have to be identified as thoroughly as possible.

The subtasks are ordered. In the example above, parsing the query has to be done before the data sources are accessed. In Figure 1, this order is given by the small numbers on the edges between parent and child nodes.

To complete one task, only the functionalities of the subtasks are required. A subtask can include further subtasks. Sometimes, which subtask is actually required depends on the processed data. To model this behavior, we use a special OR-node in our graph. In Figure 1, it is marked with ∨ (see ⓐ). Depending on the data, exactly one subtask is required to fulfil the parent task. For this reason, the outgoing edges of OR-nodes have no numbering. Note that the root node "application" is an OR-node as well because the main tasks are alternatives to each other.

One task can be a subtask of multiple other tasks (see ⓑ in Figure 1). This is often the case when using libraries. The provided functionality is accessed at many places in the application, so that the corresponding task is a subtask of several parent tasks.

Finally, a task can have its own parents as subtasks (see ⓒ in Figure 1). This is required by recursive or iterative algorithms.

As long as the implementation of a task includes multiple classes or consists of several longer methods, the task should be divided into further subtasks. Finer tasks should be preferred over coarser ones. A coarse task modeling prematurely limits the function design in the following. On the other hand, tasks that are too fine can be merged again later (see Section 3.3).

It is difficult to define a rule when a task is fine enough. This depends on the actual application as well as on the programming language, support from libraries, or implementation details. In object-oriented languages, individual methods can provide a good orientation, but in any case, it is helpful to ask for the semantic meaning of a code snippet. For high-level tasks, this question is usually easy to answer. Tasks such as "The code processes an uploaded image" or "It renders a chart for the requested time range" have a purpose that is easy to understand. They should be further divided step by step into subtasks. Eventually the descriptions become more technical: "The pixel (x|y) from the image is converted from RGB to HSV color space" or "Missing data is interpolated by neighboring values". Even without seeing the associated code, most developers know how to implement tasks like this in a few lines. This can be a good indication for reaching a sufficiently fine granularity.

By defining the tasks, we answered the first question from the introduction: "Which parts of the application could be functions?" The identified tasks are at least candidates to be used later for function design.

3.2 | Defining the program flow

In the previous section, we created a task graph. It contains all tasks with their respective subtasks. In this section, the *program flow* is defined based on this graph. It defines which task calls which other task.

The program flow is an ordered list of triples that may split into multiple branches. Each triple contains the calling task, the called task and the calling type. For example, one entry could be `(Task A, Task B, asynchronous)`. A second entry could be `(Task B, Task C, synchronous)`. If these two entries are one after the other in the program flow list, this means that first task *A* calls task *B* with an asynchronous call. Then, task *B* calls task *C* synchronously.

Section 3.2.1 provides an overview of the two calling types *synchronous* and *asynchronous* along with their strengths and limitations. Which one to use depends on whether a return value is required or not.

The program flow is initialized in Section 3.2.2. Based on the root node of the task graph the first entries are inserted into the program flow list.

Section 3.2.3 presents a priority list that is used to determine the further program flow. It operates on the task graph defined in Section 3.1 and fills the program flow list with triples.

3.2.1 | Calling types

Asynchronous calls. If task *A* does not require a return value from task *B*, *B* can be called asynchronously. In this case, *A* instructs the FaaS framework to execute *B*. This has the benefit that *A* is terminated without waiting for *B* to finish. Such a behavior is desirable in view of limited run time and low resource consumption. In addition, failed calls can be automatically retried at a later time.

Synchronous calls. In synchronous execution, task *A* calls task *B* directly. While *B* is being executed, *A* continues to run and waits for the result of *B*. There is no option to pause *A*. If it has to wait for data, the task runs idly and causes costs.

Once the result is available, it can be used in the further execution of *A*. Such behavior is necessary if *A* has to return a result to its caller based on the data of *B*, or if it needs the results from *B* for further calculations.

3.2.2 | Initialization

In this section, we initialize the program flow with the first triples. Only the root node of the task graph and its children are considered here. The further program flow is defined in Section 3.2.3.

The root node of the task graph is an OR-node. This means that its children are alternatives, which lead to independent branches in the program flow.

For each branch, one entry has to be created in the program flow list. This entry represents a call from the root node to the corresponding child node. The calling type is application-specific. In most cases, the client wants to get feedback when using the application, which requires a synchronous call. If a response is not necessary, an asynchronous call can be used.

For the task graph in Figure 1, this leads to three branches of the program flow:

```
B1: [(application, task1, synchronous)]
B2: [(application, task2, synchronous)]
B3: [(application, task3, synchronous)]
```

The root node has three children, therefore three branches were created. Each branch consists of a list containing a single triple for calling the appropriate task. It was assumed that all calls are synchronous.

3.2.3 | Priority list

Based on the initialization in Section 3.2.2, the further program flow can be defined. For this, we describe a priority list. It analyzes the last entry in the program flow together with the task graph from Section 3.1 and suggests the next entry.

The priority list assumes that all branches of the program flow are considered individually. The first case that properly describes the last entry and the position of the current task in the task graph should be applied. The other cases are ignored.

Assume that the last entry in the program flow is `(previous, current, callingType)` where `previous` and `current` are tasks from the task graph and `callingType` is either synchronous or asynchronous.

In the following priority list, task *X* is mentioned. It is a variable that should be initialized with `current`.

1. If *X* is an OR-node, split the program flow into as many branches as there are children. In each branch, add a call to the child: `(current, child, callingType)`. This calls the child in the same way as the current task was called. Then continue to trace each branch separately. This way, independent execution paths of the program are defined. This occurs, for example, if different database backends can be configured in the application, but only one is used at run time. By separating into different branches, each database backend is handled individually and the architecture remains flexible.
2. If *X* has children and these children have not yet been added to the program flow, take the first of them. Resulting from the construction of the task graph, all child nodes are needed to provide the functionality of the parent node (i.e., X). We start with the first child node.

   If `current` needs the results of the selected child node (either for its own return value or for further program execution), then call the task synchronously and add the corresponding triple to the program flow. Continue the program flow for the selected child. When no more entries are added, `current` must call another successor task. To do this, run through this priority list again and apply the first matching situation. This is necessary because `current` is the only task still running at this time. Therefore, it must ensure the further execution of the program.

   Otherwise, if the result of the called task is not required, an asynchronous invocation is possible. In this case, the called task takes care of the continuation of the program and `current` does not need an additional successor.
3. If `callingType` is “synchronous”, there is no successor. Do not add anything to the program flow. Note that there cannot be child nodes that have not yet been added to the program flow at this point. Otherwise, the second case of this priority list would have been applied.

4. If X has sibling nodes that have not been added to the program flow yet, call the next sibling asynchronously: `(current, sibling, asynchronous)` This call can be asynchronous because `current` does not need results from sibling nodes by definition. If this were the case, it would not be a sibling but a child node.
5. If X has a parent, set this parent as the new value for X and go through this priority list again. This way, the task that takes care of the further program execution is searched.
6. If none of the previously described cases apply, there is no further call necessary and the current branch in the program flow is finished.

We explain this priority list using the example in Figure 2. We assume that there is only one entry in the program flow: `(A, t, asynchronous)`. The current task is t and we want to identify the next entry for the program flow.

If possible, the next child node of t is called (case 2). In the example, t has two child nodes: B and C. The first one that has not been added to the program flow yet is taken. In this case, it is B. The task has to be called synchronously if t requires its results for further execution. Otherwise, the task can be executed asynchronously. When a task is called asynchronously, it has to take care of invoking a subsequent task. With a synchronous call, in contrast, the calling task must ensure the continuation of the program flow. Let us assume t requires results from B and add a synchronous call to the program flow.

Since B has no child nodes, it will not add any entries to the program flow (case 3). Instead, t calls another task (case 2). This time, the first child that is not already added to the program flow is C. We assume an asynchronous call and add a corresponding entry to the program flow.

C has no child nodes, was not called synchronously and has no sibling nodes that are not already added to the program flow. According to case 5, we look for the successor of the parent task t. Task t has a sibling task D under the common parent A. Case 4 specifies that this is the successor.

D itself has no successor and neither does its parent node A. This terminates the program flow (case 6). The final list looks as follows:

```
[
 (A, t, asynchronous),
 (t, B, synchronous),
 (t, C, asynchronous),
 (C, D, asynchronous)
]
```

Since it is difficult to read such a list, we use a graphical representation in the following. Asynchronous calls are represented by a single arrow, synchronous calls by a double one. The above program flow is visualized in Figure 3.

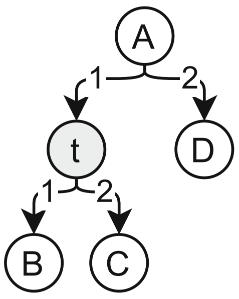

FIGURE 2 Example task graph to explain the definition of the program flow.

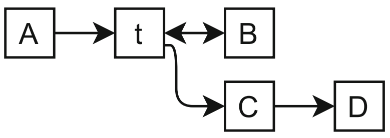

FIGURE 3 Graphical representation of the program flow through the example task graph from Figure 2.

This example is simple and only intended to illustrate the use of the priorities list. For a more complex application, we refer to the evaluation in Section 4.2.

By identifying the program flow on the tasks, we answered the second question from the introduction: "How are these parts related to each other?" The program flow defines which tasks have a connection and what kind of connection it is.

3.3 | Specifying actual functions

In the previous section, we defined the program flow through the application. It specifies which tasks call which other ones and whether the calls are synchronous or asynchronous. Now we assume that each task is a separate function. This gives us a first function setup for the application.

However, the functions are very fine-granular. Sometimes, it is useful to merge two functions into one. In other situations, it is better to keep them separate. The decision is a trade-off. On the one hand, a fine function setup offers many benefits, such as better scalability or more precise resource requests. On the other hand, a finer setup requires more effort during migration and increases communication overhead.

The candidates for merging can be derived from the program flow. If one function calls another, then these two are merge candidates. Another possibility is when two functions are called after each other by a common parent function. In this case, the two child functions are also candidates to merge.

Two functions should only be kept separate if the added value justifies the drawbacks. In the following, we describe six rules to identify situations in which separation makes sense. However, it is always necessary to weigh up how large drawbacks really are and how much gain there is in return. This answers the third question from introduction: "When should two parts of the application be covered in a common function and when in separate ones?"

3.3.1 | Changing number of elements

If one function produces exactly one output and another function consumes exactly this one output, then these functions should be merged. Nothing would be gained from keeping these functions separate. Only overhead would be created. The situation is different if the first function produces n outputs and the second function has to be called for each of them. In particular, if the execution of the second function should be parallelized to optimize performance, the functions must be kept separate.

In Figure 4, function A produces several outputs, each processed by B. The functions should be separated so that multiple instances of B can be called at the same time.

3.3.2 | Alternative functions resulting from OR-nodes

Functions based on OR-nodes should also be separated. Depending on the input, a different successor function is called. Merging these functions would lead to a function containing logic that is not used in every call. This wastes resources as the unneeded parts require memory and have to be initialized.

In Figure 5, A, B and C should form their own functions. Depending on the input, the program flow goes through either B or C, but not both. If other alternatives to B and C could be added in the future, it makes even more sense to separate them. This way, the system remains extensible.

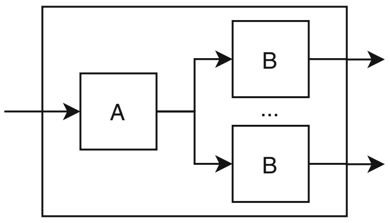

FIGURE 4 If one call of A leads to many calls of B, the functions should be kept separate. This way, the execution can be parallelized.

If separating into three functions causes too much overhead, an intermediate solution can be chosen. In this case, *A* is duplicated and merged with *B* as well as with *C*, resulting in two functions. As a consequence, the case discrimination moves up to the caller of *A*. This variant is helpful if *A* is called from a place where all the information to select the correct function is given anyway. However, it makes maintainability more difficult, because two functions have to be adapted each time *A* is updated.

3.3.3 | Multiple uses

If a function is part of several branches of the program flow, it should be kept separate. In Figure 6, *C* is required by both *A* and *B*. Without separation, all three functions would have to be merged into a common function. This should be avoided because *A* and *B* are alternatives.

Like in Section 3.3.2, there is an intermediate solution. If three functions cause too much overhead, *C* can be duplicated and merged with *A* and with *B*. Like before, this may cause issues in maintenance because an update of *C* requires a redeployment of *A* and *B*.

3.3.4 | Varying resource requirements

If the resource requirements vary greatly between two functions, they should be separated. An example could be a short-term, very high memory requirement. This is shown in Figure 7 by the enlarged box of *B*. A function containing *A* and *B* must be started with a correspondingly high memory limit but only uses this for a fraction of the run time. As a result, resources are wasted.

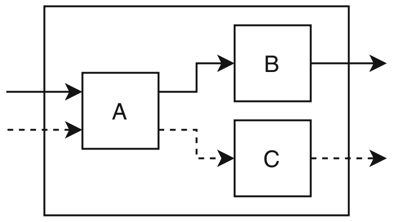

FIGURE 5 If only a subset of the functionality is used, the functions should be separated. In this case, three individual functions would be better than a big one.

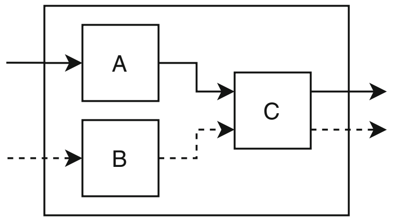

FIGURE 6 If *C* is needed at several places, it should be a function of its own.

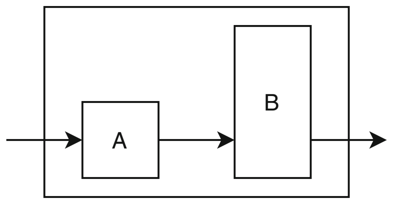

FIGURE 7 If one function requires much less resources than another function, they should be kept separate.

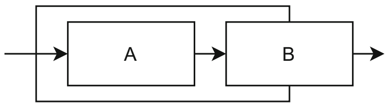

FIGURE 8 If the two functions A and B would be merged together, their total run time is longer than allowed by the cloud provider.

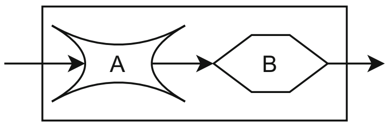

FIGURE 9 *A* and *B* are too different and cannot be merged.

3.3.5 | Long-running functions

If the expected run time of a function is already close to the limit defined by the cloud provider it should not be merged with another function. The total run time could exceed the limit and lead to an abortion (Figure 8).

3.3.6 | Too different functions

Sometimes, two functions cannot be merged because they are too different. This can be the case if they are maintained by two development teams that work independently of each other. Another reason might be that two different programming languages with different requirements for the environment are used (Figure 9).

4 | EVALUATION

We applied our approach to a real-world application called *GeoRocket*.[19] GeoRocket is a data store for very large geospatial objects. It creates an index for the data that can be used for queries. For example, a client can import a whole city model and query only the downtown area. To do this, GeoRocket breaks down each data set into small *chunks*. A chunk can be a simple geometry in a set of polygons or even a house in a 3D city model. All chunks are assigned an ID and put into a store. The metadata for each chunk is saved in an index. When querying, the required chunks are identified via the index and then retrieved from the store. More information regarding the internal architecture can be found in the paper by Krämer.[19]

Since a monolithic implementation already exists, GeoRocket is a good example to evaluate our approach. The aim of this section is to create a plan for migrating GeoRocket to FaaS.

4.1 | Task analysis

GeoRocket implements the CRUD operations *create*, *read*, *update*, and *delete*. Note that geospatial objects can only be created, read, and deleted. They are immutable. An object that should be updated has to be deleted, and the new version has to be imported again. Nevertheless, GeoRocket offers the possibility to attach tags and metadata properties to existing objects, which can be updated later on.

Figure 10 shows an overview of the tasks performed by GeoRocket. From top to bottom, the tasks become more and more detailed. While the second row contains only the four CRUD operations, the third row already contains many sub-tasks. The application task *GeoRocket* as well as the tasks *Split*, *Parse* and *Merge* are OR-nodes. Only one of their children has to be executed to complete the task. Which one depends on the content of the request. For example, GeoRocket can handle XML as well as JSON data, and the internal logic has to distinguish between them.

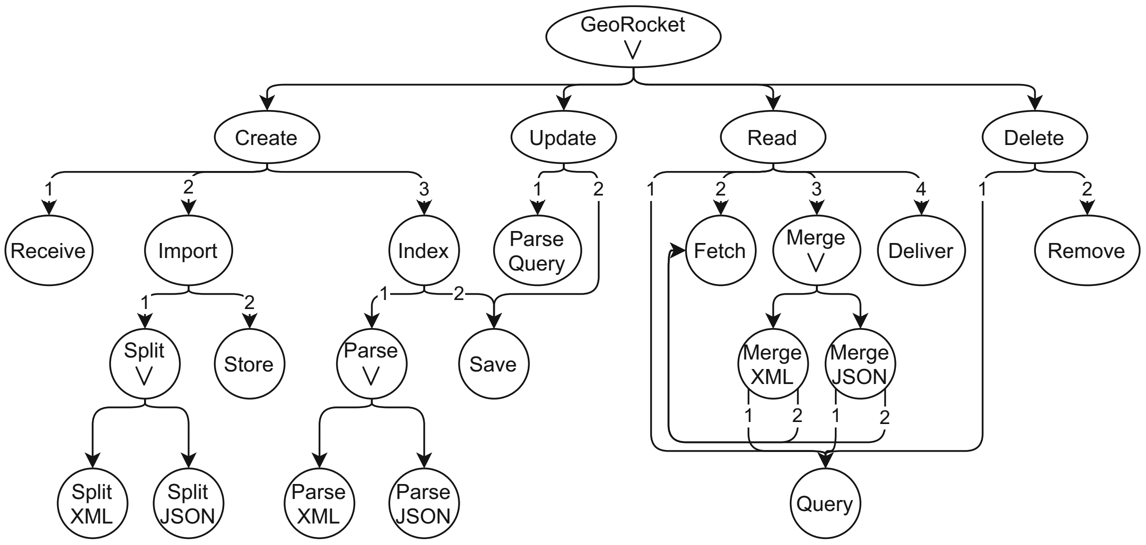

FIGURE 10 Tasks graph for GeoRocket. The root node represents the whole application and can be divided into several subtasks.

To add new data (see *Create* node in Figure 10), the input file is first *received* through an HTTP endpoint. Then, each file is *imported*. This includes *splitting* the file into chunks and *storing* each of them. Finally, each chunk is *indexed* by *parsing* it and *saving* the results in an index. Since the import of new data takes some time, GeoRocket does not provide a direct feedback to the client at this point. The request is terminated before all data has been processed (i.e., it is an asynchronous operation).

Reading data always also includes searching for it in the index. The index returns one or more chunk IDs matching a given query. The actual search takes place in the index and is not part of the application. The obtained chunk IDs can be used to *fetch* the chunks from the storage. The chunks are then *merged* into one file and *delivered* to the client. It has to be noted that these steps might need to be repeated for large responses. If the queried data contains too many chunks, GeoRocket starts merging the first ones and streams the result to the client. Afterwards, it queries the next chunk IDs and continues merging them into the output stream.

Deleting is very similar to reading data. GeoRocket runs a *query* against the index and receives the IDs of the chunks to be deleted. Afterwards, these IDs are used to *remove* the chunks from the store.

Updating data is a special case in GeoRocket, which also becomes apparent in the division of this task. As mentioned above, the data stored in GeoRocket is immutable, but metadata properties and tags can be attached to chunks and modified later. Tags and properties are only saved in the index. If they are changed, only the corresponding entries in the index have to be updated. There is no need to access the data store with the actual chunks.

4.2 | Program flow

The root node *GeoRocket* in Figure 10 is an OR-node that has four children. For each of them, we initialize the program flow with one branch (see Section 3.2.2). In addition, there are two more OR-nodes below *Create* and one below *Read*, each with two children. At all these points, the program flow splits (see case 1 in Section 3.2.3) so a total of eight different branches exist at the end. For the sake of conciseness, we only explain the branch for adding XML data in detail. The other branches can be defined in the same way.

Figure 11 illustrates the final branch for adding XML data. A single arrow represents an asynchronous call in which the caller does not expect a response from the called task. A double arrow indicates a synchronous call.

When the client adds data to GeoRocket, the *Create* task is used. To find the successor of *Create* using the priority list in Section 3.2.3, the first step is to check if child nodes exist that are not already added to the program flow. This is the case and *Receive* is the first of them. It is selected as the successor. The client should get feedback about whether

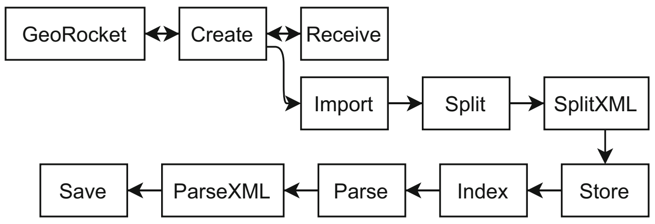

FIGURE 11 One branch of the program flow through the application.

the transfer was successful or not. For this reason, *Create* must wait for a success message from *Receive*. So this call is synchronous.

Receive has no successor because no children exist and it was called synchronously. However, a successor for *Create* has yet to be identified. The next child node that is not already added to the program flow is *Import*. In this step, the received data is divided into small chunks and stored afterwards. Since this can take some time, GeoRocket does not include its result in the response to the client. Instead, after new data is successfully received, the request is terminated. *Create* therefore does not need a response from *Import* and can call the task asynchronously.

The successor of *Import* is its first child: *Split*. The call can be asynchronous again, since *Import* does not need to process the results of *Split*. *Split* is an OR-node, so its children are alternatives. In the branch of the program flow illustrated in Figure 11, the child *Split XML* was considered. Since *Split* was called asynchronously, the request to *Split XML* can be asynchronous too.

According to Section 3.2.3, the successor of *Split XML* is *Store*. *Split XML* has no children, was not called synchronously and has no siblings—as a reminder, all children of OR-nodes are considered in different branches of the program flow. Therefore, its successor is the successor of the parent node *Split*. *Split* also has no other children, was also not called synchronously, but has a sibling node *Store*. The call can be asynchronous again, since *Split XML* does not expect a response from *Store*. *Store* takes the chunks generated by *Split XML* and puts them into the store.

Store has no children itself, has not been called synchronously, and has no siblings. Therefore, its successor is the successor of *Import*, namely *Index*. *Index* parses each chunk and extracts metadata. This information is then saved in an index and can be used later to query chunks.

Finding successors at this point is equivalent to *Import*. The last task is *Save*. It has no successor, so the branch of the program flow ends here.

The other branches can be defined in the same way. It should be noted that asynchronous calls cannot be used for the *Update*, *Delete*, and *Read* branches, since GeoRocket uses all results from the subtasks for the response to the client. The final program flow including all branches is shown in Figure 12.

4.3 | Function definition

In the previous section, we defined the program flow. Now we consider each task (illustrated as a box in Figure 12) as a function candidate. This section examines which of them can be merged. For this, we analyze which of the rules from Section 3.3 can be applied to the program flow. For the sake of brevity, we focus on adding new data to GeoRocket, which is the most complex branch. The other branches can be treated analogously.

The corresponding merge candidates have been marked in Figure 12 with red circles. In the following list, we explain which functions should be merged into one and which should not. For this, we identify matching rules from Section 3.3 and estimate the possible gain. In addition, we estimate how large the overhead is for a separation.

① *GeoRocket ↔ Create*
Benefits when separating: Medium
Rule 3.3.2: *GeoRocket* is an OR-node. *Create*, *Update*, *Delete* and *Read* are alternatives. Merging *GeoRocket* and *Create* would lead to unnecessary logic, for example when *Update* is called. However, the overhead is not particularly high because the main work is done in the sub-functions.
Benefits when merging: High

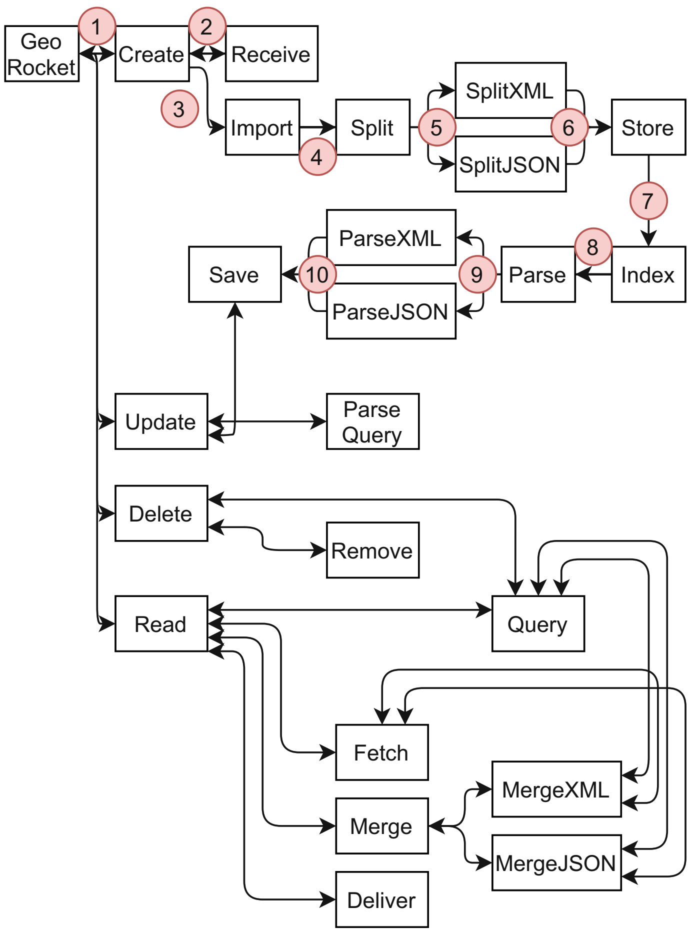

FIGURE 12 The program flow in the entire application. The merge candidates when adding new data to GeoRocket are marked with red circles. They are explained in Section 4.3.

Often, new data is large. If it has to be sent from one function to the next, this causes a high communication overhead that can be avoided by merging.
Result: Merge

② *Create ↔ Receive*
Benefits when separating: Low
No rule can be applied.
Benefits when merging: High
Same argumentation as in ①.
Result: Merge

③ *Create ↔ Import*
Benefits when separating: High
Rule 3.3.1: When data is added to GeoRocket, it might consist of multiple files. Each file has to be imported. So, one call of *Create* leads to several calls of *Import*. To enable parallel processing, the functions should be kept separate. Rule 3.3.5: If *Create* and *Import* were merged into one function, the maximum run time could easily be exceeded as it scales with the number of files to be imported.
Benefits when merging: Medium
In total, the entire new data set must be transferred. However, the amount of data is distributed over all calls to *Import*, so that a single function invocation is no longer particularly large.
Result: Separate

④ *Import ↔ Split*
Benefits when separating: Low
No rule can be applied.

Benefits when merging: Medium
A separation would lead to an additional transmission of each new file.
Result: Merge

⑤ *Split ↔ Split XML / Split JSON*
Benefits when separating: Medium
Rule 3.3.2: *Split* is an OR-node and should be separated from its children.
Benefits when merging: Medium
A separation would lead to an additional transmission of each new file.
Result: Merge but distinguish between JSON and XML before ③

⑥ *Split XML / Split JSON ↔ Store*
Benefits when separating: Medium
Rule 3.3.1: The split functions generate multiple chunks. Each of them must be stored so that one call of *Split XML* or *Split JSON* lead to multiple calls of *Store* Rule 3.3.3: Both semantic split functions call *Store*.
Benefits when merging: High
Store is a pure I/O operation that contains very little logic of its own. The resulting communication overhead when separating is therefore disproportionate to the gain.
Result: Merge

⑦ *Store ↔ Index*
Benefits when separating: High
Rule 3.3.1: Since the *Store* function was merged with the split functions in ⑥, one invocation of *Store* leads to many invocations of *Index*. Rule 3.3.3: Since *Store* was merged, there are still two functions that are calling *Index*. Rule 3.3.5: If *Store* and *Index* were merged into one function, the maximum run time could easily be exceeded as it scales with the number of extracted chunks. If a file contains many chunks, the total run time might be longer than allowed.
Benefits when merging: Low
Only the ID of the chunk has to be transmitted.
Result: Separate

⑧ *Index ↔ Parse*
Benefits when separating: Low
No rule can be applied.
Benefits when merging: Low
Only one additional function call with the chunk ID would be necessary in the case of a separation.
Result: Merge

⑨ *Parse ↔ Parse XML / Parse JSON*
Benefits when separating: Medium
Rule 3.3.2: *Parse* is an OR-node and should be separated from its children.
Benefits when merging: Medium
If these functions were separated from each other, the function containing *Index* and *Parse* would be very small and only a glue function between the chunk creation and parsing.
Result: Merge but distinguish between JSON and XML before

⑩ *Parse XML / Parse JSON ↔ Save*
Benefits when separating: Medium
Rule 3.3.1: The parse functions generate multiple results. All of them have to be saved. Rule 3.3.3: Both semantic parse functions as well as the *Update* function call *Store*.
Benefits when merging: Medium
Save is a pure I/O operation that contains very little logic of its own. The resulting communication overhead when separating is therefore disproportionate to the gain.
Result: Merge

Figure 13 shows the resulting grouping of the functions. Like before, a synchronous call is represented by a double arrow and an asynchronous call by a single arrow. This results in a total of ten functions, with *GeoRocket* being the entry function that calls all the others. In comparison to Figure 12, many function candidates were combined to reduce communication overhead and the distinction between data types was shifted forward. This resulted in some duplicated

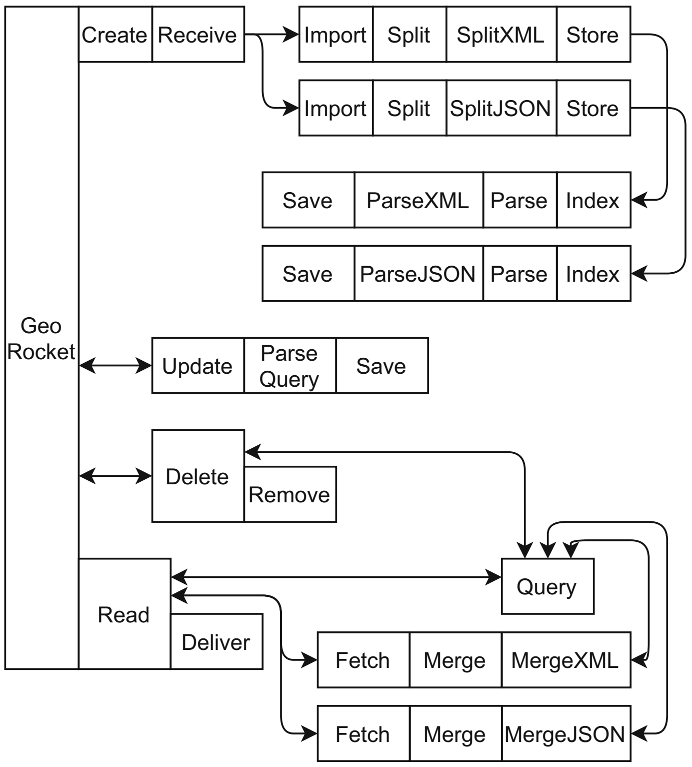

FIGURE 13 The final function design after merging.

tasks for adding new data as well as for reading them. However, these duplications made it possible to reduce the number of function calls, enabling a more efficient processing.

4.4 | Performance

We implemented the presented concept and run it with the FaaS framework OpenFaaS[20] (helm chart version 8.0.4) on top of Kubernetes[21] (version 1.22.5). Kubernetes was installed on three virtual machines with 4 cores and 16 GB of RAM each. For comparison, we installed the existing monolithic version of GeoRocket (version 1.3.0) on three other instances with the same hardware configuration. For the monolithic evaluation, incoming requests were distributed to one of these instances by an nginx proxy using round robin. We imported a public LoD 2 model of the city of Cologne (Germany)[22] with the FaaS version as well as with the monolithic implementation. The dataset consists of 479 CityGML files with a total size of 7.7 GB. The monolithic implementation took 59 min while our approach with OpenFaaS is 19% faster and needs only 48 min (see Figure 14).

If a new data set is imported via the FaaS implementation, the split function gets called for each included file. Because of the asynchronous invocation, the request is stored in a queue until it can be processed. Figure 15A visualizes the queue length (orange line). Right at the beginning, the number of queued split calls increases fast. The reason is that not enough instances of the split function are started yet. This changes a few minutes after start as shown in Figure 15B. In response to the increased queue length, the split function is automatically scaled up to the configured maximum of 10 instances. The queue continues to grow, but at a much slower rate than at the beginning. The blue line in Figure 15 shows the queue length and the number of function instances for indexing. After a file was split into chunks, the indexer function analyses each of them. It is an asynchronous call again, so the requests are stored in a queue. This is where the flexibility of FaaS becomes apparent: sometimes the queue is a bit longer, in which case more function instances are started. As soon as the queue is shorter again, these additional instances are terminated. In the end, this flexibility enables a faster run time in comparison to the monolithic implementation.

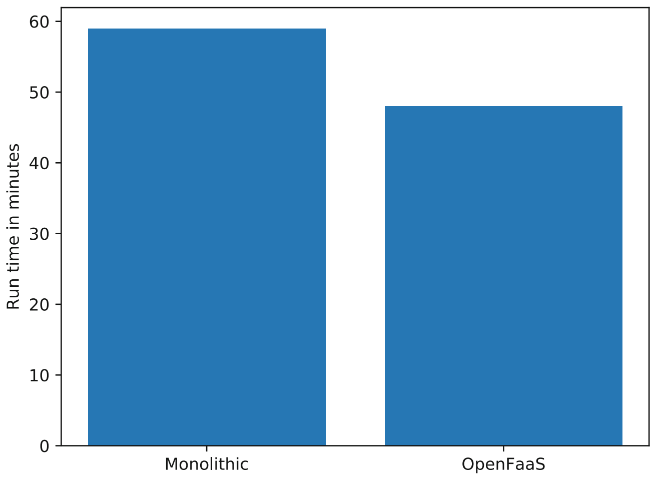

FIGURE 14 Comparison between the monolithic version of GeoRocket with our OpenFaaS implementation when importing a city model. Both setups use 3 VMs with 16 GB of RAM each.

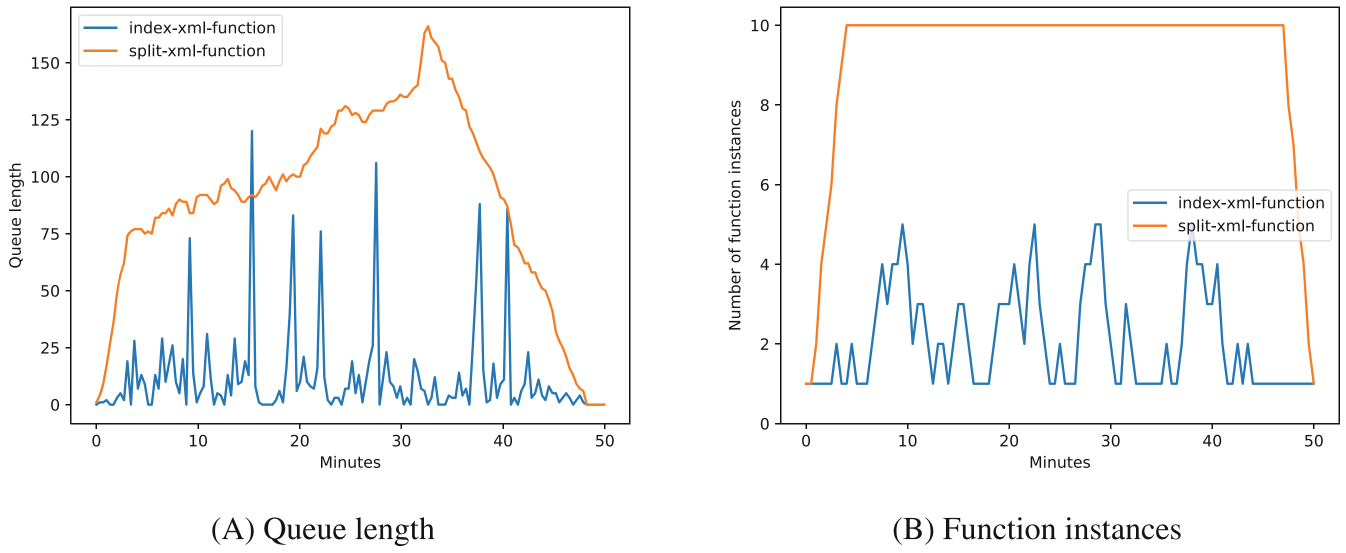

FIGURE 15 Queue length of pending asynchronous function calls and number of split / index function instances during the import of a city model.

5 | DISCUSSION

In this section, we critically examine our approach and identify potential limitations. Section 4 showed that the approach is well suited for the migration of GeoRocket. Nevertheless, the question arises whether this applies to all kinds of applications.

The first difficulty is that the developer has to know the application very well. Otherwise, the definition of the tasks and the program flow would not be possible. The only solution to solve this problem is automated analysis of the source code. This has already been investigated by other works (see Section 2) and is not suitable for larger applications. This is also due to the fact that an estimation of the communication overhead is necessary when deciding between merging and separating of functions. For this, the developer must know which data is processed, how large it is and how frequent a function is executed. This is hard to do automatically. A developer can use the presented approach to analyze applications of any size. Nevertheless, nothing is for free. It should be noted that the larger the application, the more effort is needed to identify tasks, assess impacts, and derive the final functional design. This inherent problem will remain, but the developer can be supported by the presented approach.

The second difficulty is that people can make mistakes. When deciding to separate a function, a trade-off must be made between the resulting benefits and drawbacks. Sometimes, this decision is clear, sometimes it is not. For example, for merge candidate ⑤, the decision was made to perform a case discrimination between XML and JSON data before ③ and then merge the semantic split functions into the previous functions. This resulted in many duplicated tasks. Alternatively, *Split XML* and *Split JSON* could have been merged into one big function. This would have avoided duplicating three other tasks. There is no right or wrong at this point, only pros and cons that need to be weighed up against each other. The approach presented in this paper helps developers to make reasonable decisions in this respect. To verify the decision, the resulting setup can be benchmarked in comparison to the monolithic version as other works suggest for microservices[23] or large distributed systems.[24]

Other works exploit the structure of the source code during the migration process. For example, Escobar et al. use Enterprise Java Beans (EJB) for a breakdown into microservices.[25] This enables more knowledge to be gained from the underlying programming language, but it limits the approach to applications written in that language. Our approach is independent of the implementation but requires more effort in defining the program flow. Nevertheless, the program flow indirectly depends on the implementation. If the monolith is poorly written, then it can be more difficult to identify the individual tasks. For example, an application in Java provides a good indication of possible tasks through its class structure. An old procedural PHP implementation makes this process much harder.

There are some applications that are unsuitable for running in a FaaS environment. This applies, for example, to applications that have tightly coupled components. In such applications, a lot of communication overhead would be incurred if they were split into functions, since a large amount of information would have to be exchanged. Another example are applications that have to read or write a lot of data. Here, an execution as FaaS is also associated with a high communication overhead. The presented approach will not lead to a good FaaS setup in these cases. However, it will become apparent while attempting to define functions that it is best to merge all of the individual tasks. If a developer observes such a behavior, then consideration should be given to whether running as FaaS is actually the right way to go.

Finally, it should be considered that parallelization is only supported in the form of loops (see Section 3.3.1). If a task produces multiple outputs and each of them is processed by a different function, then this can be done in parallel. On the other hand, if two processes are started independently, then this behavior cannot be modeled with the presented approach. This decision was made to keep the approach concise. However, if needed, another node type for parallelization could be added in the task graph. Its children would all be executed in parallel. They have no order (as in OR nodes), but would all have to be executed (as in normal nodes). This raises interesting questions regarding how to synchronize the execution at the end of the parallel part, how to call subsequent tasks after parallelization, and who is responsible for that. We will address this important topic in a future work.

6 | CONCLUSIONS AND FUTURE WORK

In this paper, we presented an approach to decompose an existing monolithic application into functions for Function as a Service. We used three steps that correspond to our original questions in the introduction Section 1: First, we proposed to identify the main tasks of an application and further refine them to obtain a task graph (Section 3.1). Subsequently, we presented a priority list that can be used to identify the program flow through the task graph (Section 3.2). This results in a list of triples, each indicating which task calls which other task and whether this call is synchronous or asynchronous. Finally, we presented six rules that can be used to decide whether two tasks should be combined into one function or whether two separate functions are better (Section 3.3).

This solved the challenges from the beginning. The identified tasks of an application can be considered as a single function. They are related via the program flow and can be called either synchronously or asynchronously. The rules from Section 3.3 provide suggestions for situations when two functions should be merged and when they should be kept separate.

The suitability of our approach was demonstrated using GeoRocket as an example. The tasks of GeoRocket resulted from a stepwise refinement of the four main tasks *Create*, *Read*, *Update*, and *Delete*. Subsequently, the program flow through the application was determined. This took into account which information is required to respond to a request and when asynchronous function execution is possible. Finally, the functions were merged until the communication overhead no longer outweighed the benefits of FaaS execution. The result was an application with ten functions that together represent the functional scope of the monolithic application.

In the future, we will investigate whether the speed of the FaaS version can be further increased. As addressed in Section 2, cold start is a problem of FaaS that can be solved by prewarming containers. While existing approaches rely on a prediction of future function calls, we can make a more confident statement by using the program flow. This includes not only which function will be called next, but also how often. We want to exploit this additional information and enable the FaaS framework to scale the functions in a more efficient way. This can reduce latency for function invocations, leading to faster processing.

Additionally, we will investigate the connections with workflow management systems (WMS). In WMS, the processing of data is structured in workflows. They consist of different services that receive data, process it and produce output. This output is used as input for subsequent services. A future work should investigate to what extent the services in WMS can be replaced by functions in FaaS. In this way, a monolithic application can first be broken down into functions, which then become services in a WMS.

AUTHOR CONTRIBUTIONS:

Hendrik M. Würz: Conceptualization, Investigation, Methodology, Project Administration, Software, Validation, Visualization, Writing – Original Draft Preparation. **Michel Krämer**: Conceptualization, Methodology, Project Administration, Supervision, Writing – Review & Editing. **Marvin Kaster**: Investigation, Methodology, Software, Validation, Visualization, Writing – Original Draft Preparation. **Arjan Kuijper**: Project Administration, Supervision, Writing – Review & Editing.

ACKNOWLEDGMENT

Open Access funding enabled and organized by Projekt DEAL.

FUNDING INFORMATION:

Open Access funding enabled and organized by Projekt DEAL

DATA AVAILABILITY STATEMENT

Data sharing is not applicable to this article as no new data were created or analyzed in this study.

ORCID

Hendrik M. Würz https://orcid.org/0000-0002-4664-953X
Michel Krämer https://orcid.org/0000-0003-2775-5844
Arjan Kuijper https://orcid.org/0000-0002-6413-0061

REFERENCES

1. Leitner P, Wittern E, Spillner J, Hummer W. A mixed-method empirical study of function-as-a-service software development in industrial practice. *J Syst Softw*. 2019;149:340-359.
2. Fox GC, Ishakian V, Muthusamy V, Slominski A. Status of serverless computing and function-as-a-service (faas) in industry and research. arXiv preprint arXiv:1708.08028 2017.
3. Barcelona-Pons D, Sánchez-Artigas M, París G, Sutra P, García-López P. On the FaaS track: building stateful distributed applications with serverless architectures. Paper presented at: Middleware'19. Association for Computing Machinery, New York, NY, USA. 2019:41-54.
4. Soltani B, Ghenai A, Zeghib N. A migration-based approach to execute long-duration multi-cloud serverless functions. Proceedings of the 3rd International Conference on Advanced Aspects of Software Engineering, ICAASE 2018, Constantine, Algeria, December 1-2, 2018. 2326 of CEUR Workshop Proceedings. 2018:42-50.
5. Bermbach D, Karakaya A, Buchholz S. Using application knowledge to reduce cold starts in FaaS services. Paper presented at: SAC'20: the 35th ACM/SIGAPP Symposium on Applied Computing, Online Event, [Brno. Czech Republic], March 30–April 3, 2020, ACM. 2020:134-143.
6. Lloyd W, Vu M, Zhang B, David O, Leavesley G. Improving application migration to serverless computing platforms: latency mitigation with keep-alive workloads. Paper presented at: 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), IEEE; Los Alamitos, CA, USA. 2018:195-200.
7. van Eyk E. Four Techniques Serverless Platforms Use to Balance Performance and Cost. 2019. https://www.infoq.com/articles/serverless-performance-cost/

8. Wu J, Zhang J, Zhang Y, Wen Y. Constraint-aware and multi-objective optimization for micro-service composition in mobile edge computing. *Softw: Pract Exper*. 2023:1-25. doi:10.1002/spe.3217
9. Laan EM, Broekhuis M, Offenbeek M, Ahaus K. Service decomposition: a conceptual analysis of modularizing services. *Int J Oper Prod Manag*. 2016;36(3):308-331. doi:10.1108/IJOPM-06-2015-0370
10. Evans E. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional; 2004.
11. Balalaie A, Heydarnoori A, Jamshidi P, Tamburri DA, Lynn T. Microservices migration patterns. *Softw: Pract Exper*. 2018;48(11):2019-2042.
12. Newman S. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, Inc.; 2015.
13. Richardson C. *Microservices Patterns*. Manning Publications Company; 2018.
14. Gysel M, Kölbener L, Giersche W, Zimmermann O. Service cutter: a systematic approach to service decomposition. In: Aiello M, Johnsen EB, Dustdar S, Georgievski I, eds. *Service-Oriented and Cloud Computing–5th IFIP WG 2.14 European Conference, ESOCC 2016, Vienna, Austria, September 5-7, 2016, Proceedings*. 9846 of Lecture Notes in Computer Science. Springer; 2016:185-200.
15. Fritzsch J, Bogner J, Zimmermann A, Wagner S. From monolith to microservices: a classification of refactoring approaches. In: Bruel J, Mazzara M, Meyer B, eds. *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment–First International Workshop, DEVOPS 2018, Chateau de Villebrumier, France, March 5-6, 2018, Revised Selected Papers*. 11350 of Lecture Notes in Computer Science. Springer; 2018:128-141.
16. Hamzeh R. Decision support for migrating application's functionality to FaaS. Master's Thesis 2019.
17. Spillner J, Dorodko S. Java code analysis and transformation into AWS lambda functions. CoRR. 2017 abs/1702.05510.
18. Spillner J. Transformation of python applications into function-as-a-service deployments. CoRR. 2017 abs/1705.08169.
19. Krämer M. GeoRocket: a scalable and cloud-based data store for big geospatial files. *SoftwareX*. 2020;11:100409. doi:10.1016/j.softx.2020.100409
20. OpenFaaS Ltd. Home | OpenFaaS–Serverless Functions Made Simple; 2022. https://www.openfaas.com/
21. Kubernetes Authors. Kubernetes; 2022. https://kubernetes.io/
22. Landesbetrieb Information und Technik Nordrhein-Westfalen. Köln EPSG25832 CityGML. 2020 https://www.opengeodata.nrw.de/produkte/geobasis/3dg/lod2_gml/lod2_gml_paketiert/3d-gm_lod2_05315000_KC3B6ln_EPSG25832_CityGML.zip
23. Bjørndal N, Bucchiarone A, Mazzara M, Dragoni N, Dustdar S. Migration from Monolith to Microservices: Benchmarking a Case Study. 2020. doi:10.13140/RG.2.2.27715.14883
24. Denys PF, Fournier Q, Dagenais MR. Distributed computation of the critical path from execution traces. *Softw: Pract Exper*. 2023;53(8):1722-1737. doi:10.1002/spe.3210
25. Escobar D, Cardenas D, Amarillo R, et al. Towards the understanding and evolution of monolithic applications as microservices. XLII Latin American Computing Conference, CLEI 2016, Valparaíso, Chile. 2016:1-11. doi:10.1109/CLEI.2016.7833410

How to cite this article: Würz HM, Krämer M, Kaster M, Kuijper A. Migrating monolithic applications to function as a service. *Softw: Pract Exper*. 2024;54(2):149-167. doi: 10.1002/spe.3263