Patterns for Serverless Functions (Function-as-a-Service): A Multivocal Literature Review

Conference Paper · May 2020

DOI: 10.5220/0009578501810192

Davide Taibi
Tampere University
215 PUBLICATIONS **3,829** CITATIONS
SEE PROFILE

Nabil El Ioini
Free University of Bozen-Bolzano
112 PUBLICATIONS **1,253** CITATIONS
SEE PROFILE

Claus Pahl
Free University of Bozen-Bolzano
426 PUBLICATIONS **8,210** CITATIONS
SEE PROFILE

Patterns for Serverless Functions (Function-as-a-Service): A Multivocal Literature Review

Davide Taibi[1][a], Nabil El Ioini[2][b], Claus Pahl[2][c], Jan Raphael Schmid Niederkofler[2]

[1] *Tampere University, Tampere, Finland*

[2] *Free Univeristy of Bozen-Bolzano, Bozen-Bolzano, Italy*

davide.taibi@tuni.fi, nabil.elioini@unibz.it, claus.pahl@unibz.it, Jan.SchmidNiederkofler@stud-inf.unibz.it

Keywords: Serverless, Function as a service, Serverless Functions, Cloud

Abstract: [Context] Serverless is a recent technology that enables companies to reduce the overhead for provisioning, scaling and in general managing the infrastructure. Companies are increasingly adopting Serverless, by migrating existing applications to this new paradigm. Different practitioners proposed patterns for composing and managing serverless functions. However, some of these patterns offer different solutions to solve the same problem, which makes it hard to select the most suitable solution for each problem. [Goal] In this work, we aim at supporting practitioners in understanding the different patterns, by classifying them and reporting possible benefits and issues. [Method] We adopted a multivocal literature review process, surveying peer-reviewed and grey literature and classifying patterns (common solutions to solve common problems), together with benefits and issues. [Results] Among 24 selected works, we identified 32 patterns that we classified as orchestration, aggregation, event-management, availability, communication, and authorization. [Conclusion] Practitioners proposed a list of fairly consistent patterns, even if a small number of patterns proposed different solutions to similar problems. Some patterns emerged to circumvent some serverless limitations, while others for some classical technical problems (e.g. publisher/subscriber).

1 Introduction

Serverless computing is a new paradigm that allows companies to efficiently develop and deploy applications without having to manage any underlying infrastructure (Lloyd et al., 2018). Different serverless computing platforms such as AWS Lambda, Azure Functions, and Google Cloud Functions have been proposed to the market. Such platforms enable developers to focus only on the business logic, leaving all the overhead of monitoring, provisioning, scaling and managing the infrastructure to the cloud service providers [S1].

The most prominent implementation of serverless computing is Function-as-a-Service (FaaS) (also called ”serverless functions”). When using FaaS, developers only need to deploy the source code of short-running functions, and define triggers for executing them. The FaaS provider then, on-demand, executes and bills functions as isolated instances and scales their execution. In the remainder of this work, we refer to FaaS with the term ”Serverless Functions”.

One of the main issues with serverless functions is the lack of patterns for composing different functions in order to create a complete application or part of it. Previous works also highlight the lack of patterns in serverless functions (Leitner et al., 2019)[S1], and practitioners are commonly pointing out this problem in technical talks, often proposing patterns that are discordant.

Therefore, in order to help practitioners and researchers to map the current state of the art on serverless patterns, in this work, we propose a review and a classification of patterns proposed both by practitioners and by researchers.

We adopted a Multivocal Literature Review process (Garousi et al., 2019), to capture both the state of the art and the state of practice in the field, including both white literature (i.e., peer-reviewed papers) and grey literature (i.e., blog posts, industrial whitepapers, books and practitioner’s talks). We selected 24 studies, published from 2017 to the end of January 2019. Then, we proposed a taxonomy of architectural patterns aimed at solving a set of common problems, to-

[a] https://orcid.org/0000-0002-3210-3990
[b] https://orcid.org/0000-0002-1288-1082
[c] https://orcid.org/0000-0002-9049-212X

gether with their benefits and issues.

The contributions of this work, mainly targeted to practitioners and researchers, are two-folded:

- 32 serverless patterns, classified into five categories, that practitioners can adopt in their work.
- Benefits and issues of the patterns reported in the literature and by practitioners.
- Identification of trends and open issues on serverless patterns.

The remainder of this paper is structured as follows. In Section 2 we compare our work with the existing literature reviews and we present related works. In Section 3 we describe the study process we adopted while in Section 4 we report the results. Section 5 discusses the results and threats to validity. Finally, Section 6 draws conclusions.

2 Related Work

Despite the very recent introduction of the serverless technology, the research in this field is very active. As reported by the Serverless Literature Dataset[1](Al-Ameen and Spillner, 2018), almost 200 papers have been published between 2017 and 2019.

In this Section, we present the literature reviews already conducted on this field, and we introduce the related work that investigated serverless patterns.

2.1 Literature Reviews on Serverless

Five different works already surveyed the literature on Serverless.

- Lynn et al. (Lynn et al., 2017) compared the features of seven serverless computing platforms identifying benefits and issues of each of them.
- Alqaryouti and Siyamba (Alqaryouti et al., 2018) conducted a holistic literature review classifying the various proposals related to scheduling tasks in clouds. They identified approaches to minimize execution time. Differently from our work, they highlighted issues in scheduling but not investigated patterns.
- Kuhlenkamp and Werner (Kuhlenkamp and Werner, 2018) conducted a systematic literature review to map the empirical evidence on serverless functions. They identified nine studies that conducted a total of 26 experiments on five qualities such as performance, scalability, and availability. Also in this case, their goal was not targeted to the identification of architectural patterns.
- Sadaqat et al. (Sadaqat et al., 2018) proposed a non-peer reviewed multivocal literature review to identify the core technological components of serverless computing, the key benefits and the challenges. Also in this case, they did not focus on patterns.
- Leitner et al. [S10] performed a mixed-method study, which brings value to this new field by looking at both peer-reviewed and gray literature on serverless patterns. Their goal was to identify the type of applications that mostly benefit from serverless technology, and the patterns that have been used to implement them. As a result, 5 patterns have been identified when implementing serverless-based services. The authors also discuss the drawbacks in terms of development overhead, configuration and performance that each of the pattern can cause. Differently than in our work, they elicited the patterns from a survey, while we aim at understanding the patterns reported in the literature. While an empirical evaluation of the patterns can be of very high value, we especially aim at classifying the patterns proposed by the practitioner's talks, that might highly influence other practitioners in the adoption of the patterns they will use in their applications.

2.2 Serverless Patterns

Six peer-reviewed works already investigated serverless patterns while one work investigated anti-patterns (Nupponen and Taibi, 2020).

In [S7], the authors have focused on serverless-based design patterns to improve the security of cloud-based services. They have presented six design patterns that can be combined in order to build a threat intelligent services such as intrusion detection and virus scanning. Rabbah et al. [S24] published a patent where they proposed patterns dealing with composing serverless functions to build new applications. Bernstein et al. [S6] present an example of a possible pattern that can be built relying on serverless services. The goal is to process event-streams using single-threaded virtual actors, which increases performance and parallel execution of tasks. Gadepalli et al. [S15] outlined an architectural pattern dealing with the possible applications of serverless technology in combination with Edge Computing (Pahl et al., 2019). Nupponen and Taibi (Nupponen and Taibi, 2020) presented an industrial survey where they

[1]The Serverless Literature Dataset http://serverless.research-output.org/

distilled a set of anti-patterns from common problems experienced by practitioners while developing serverless-based applications.

As for non-peer reviewed literature, a recently published book [S8] provides a more in-depth explanation of serverless computing and the different design patterns to consider when architecting an application. In particular, the book discusses four major pattern classes, namely web application patterns, extract transform load (ETL) patterns, big data patterns, and automation and deployment patterns. In [S13] the author presented guidelines to help companies embrace serverless technology by showing concrete use cases, architectural designs, and patterns. The book details five specific patterns and how they can be implemented. We note that all the five patterns are not unique to serverless technology, however, implementing them using serverless components and services can be considered a good contribution.

Our work differs from the existing literature not only by focusing on patterns from a specific domain (e.g., security) or to solve a particular problem (e.g., streaming data), rather, we applied a systematic approach identifying all existing patterns in the state of the art including both peer-reviewed and non-peer reviewed works.

3 Study Design

Architectural patterns exist for several architectural styles. There are patterns for object-oriented monolithic systems (Gamma et al., 1994), patterns for service-oriented architectures (Erl, 2008), patterns for microservices (Taibi et al., 2018)(Taibi et al., 2019a)(Taibi et al., 2019b) and patterns for several other architectural styles. Practitioners recently started to discuss patterns in serverless functions.

In order to elicit the existing patterns for serverless functions, we conducted a Multivocal Literature Review (Garousi et al., 2019). In particular, we focus on answering the following questions:

RQ1 What are the different serverless-based architectural patterns?

RQ2 What benefits and issues have been highlighted for these patterns?

The first question revolves around defining the concept of Serverless technology and how researchers and practitioners were able to define specific patterns to build Serverless based services. RQ2 explores the pros and cons of each of the patterns taking in consideration two main points of views, namely researchers who have been exploiting the new technology and practitioners who have used it in real settings.

3.1 Multivocal Literature Review

Multivocal Literature Review (MLR) is a type of literature review that considers not only academic and peer-reviewed literature, rather, it includes also practitioners' opinions and literature produced by industrial professionals. MLR is a double-sided blade in that, differently from the classical systematic literature review (SLR), it does not rely solely on peer-reviewed studies. This could bring high value to answer questions from an industrial and practical point of view. However, it could also lead to biased conclusions if taking into account non-founded claims and views.

MLR relies on what is called gray literature (GL), which could take many forms including industrial, government and organizational reports, as well as publicly available resources such as blogs and online articles. When new fields are still not well established, it is more common to find more GL on the topic that its peer-reviewed counterpart, this is mainly due to the fact that academic studies rely on a more systematic process that requires longer time (e.g., conference or journal submissions, notifications, and publication). In our context, given the fact that serverless technology is still relatively new, only a few studies have touched on the topic of serverless patterns. Therefore, including GL in our study enables us to combine academic studies with the state of the practice given by the GL.

In order to minimize the bias in the section process, we follow the MLR protocol proposed by Garousi et al. (Garousi et al., 2019).

3.2 The Search Process

The MLR protocol is divided into three main steps *i)* define the search process, *ii)* source selection, and *iii)* literature assessment, as illustrated in Figure 1.

Based on our RQs, the following search string was generated

(”serverless” OR ”function as a service” OR ”function-as-a-service” OR ”IaaS” OR ”Lambda” OR ”Function”) AND (”pattern” OR ”architecture” OR ”best practice” OR ”smell” OR ”issue” OR ”problem” OR compos OR ”anti-pattern” OR ”anti pattern”)*

The search string was used both in google search engine as well as a set of digital libraries. Google search was used to track grey literature, while digital libraries targeted peer-reviewed studies. The digital libraries we considered were IEEExplore, SpringerLink, Google Scholar, ACM, Science Direct and Scopus. The search process yielded 127 results. Due to

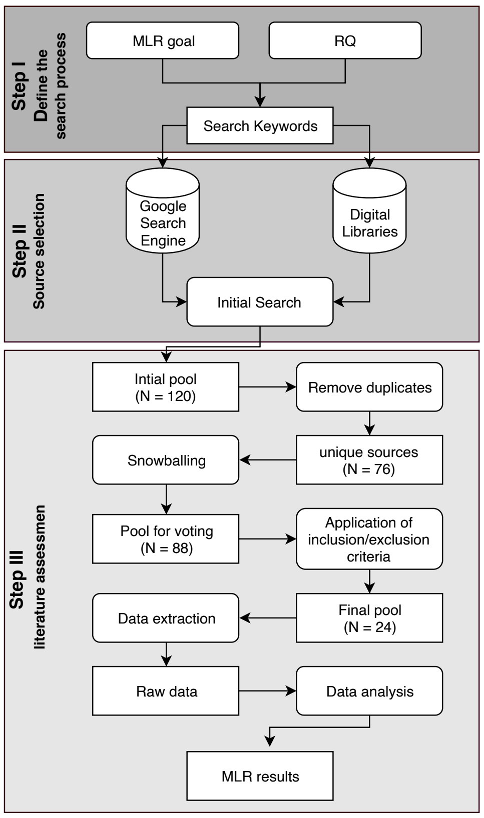


Figure 1: MLR search process

the significant number of unrelated results in google search, we limited the results to the first 10 pages. Afterward, we filtered the results to remove any duplicates. This process resulted in 76 unique sources. 65 practitioners blogs, 7 peer-reviewed studies, 2 videos, 1 patent, and 1 Powerpoint presentation. From the application of the snowballing process on the references and links contained in the initial set of results, we included 12 more blogs obtaining a total of 88 sources.

To select the relevant sources for our process, we developed a set of inclusion and exclusion criteria. To test the applicability of the criteria, the authors applied the criteria independently to the full content of 10 GLs randomly selected from the result pool. Based on some disagreements and discussions regarding the inclusion of three GLs, the criteria has been reformulated to be more precise. Using these inclusion and exclusion criteria, we were able to capture most of the sources that represent the state of the art in the area of serverless patterns. The final list of inclusion and exclusion criteria is listed in Table 1.

All the authors have evaluated the sources independently by indicating whether each source *i)* satisfies, or *ii)* does not satisfy the criteria. In case of conflicts, all the authors were involved in clarifying any doubts about the sources in question.

Table 1: inclusion and exclusion criteria. (All of the following criteria must be satisfied by the selected sources)

Inclusion Criteria
GLs on serverless functions describing at least one pattern
Exclusion Criteria
GLs NOT in English
GLs using the term 'FaaS' or "serverless" for other purposes (e.g. in the food service industry)
GLs mentioning a pattern but not describing it
Duplicated GLs
GLs only summarizing other GLs
GLs selling a third party product
GLs reporting only anti-patterns, as they are out of scope of this work

Table 2: MLR extraction Form

Attribute	Description
Source	The publication source
Publication date	when the source was published
Authors	The author or list of authors
Title	The source title
Pattern Name	The given name of the pattern
Pattern Description	A description of the pattern
Problem it solves	The specific problem the pattern solves
Advantages	Advantages of implementing this pattern
Limitations	Limitations (e.g. platform dependent)

Finally, we applied the quality assessment adopting the criteria proposed by Garousi et al. (Garousi et al., 2019). All the sources passed the quality criteria and, as a result, we obtained 24 unique sources from which we extracted the patterns. The attributes of interest have been extracted based on the extraction form defined in Table 2.

From the extracted data, we grouped the patterns based on problems they are aimed to solve. In case the selected sources adopted different names for the same pattern, we opt for the most common name, while in case of equal number of sources reporting the same pattern, but with different names, we discussed and selected the name that was more clear to understand.

4 The Patterns Proposed by Practitioners

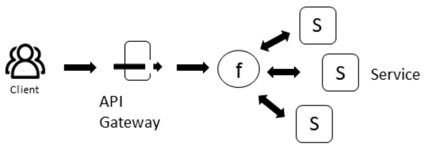
From the selected GLs we identified 32 patterns that we classified into five categories, namely *1)* orchestration and aggregation, *2)* event management, *3)* availability, *4)* communication, and *5)* authorization.

In order to ease the reading, especially from the practitioners' point of view, we answer our RQs for each pattern independently. For each pattern, we first describe the problem it is aimed to solve (RQ1), and then benefits and issues, when reported by the GLs (RQ2). All the patterns are listed in Table 3 together with the alternative names adopted in the GLs.

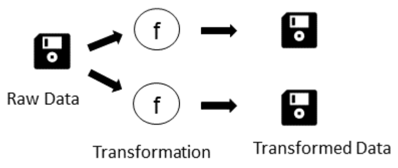
4.1 Orchestration and Aggregation

These patterns can be used to compose serverless functions or to orchestrate their execution creating more complex functions or microservices.

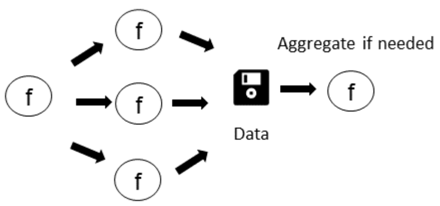
Aggregator [S12][S2][S1] (also known as "Durable functions" [S3]):
Problem: Exposing a single endpoint for several APIs.
Solution: A function calls APIs separately, then aggregates the results and exposes them as a singular endpoint.



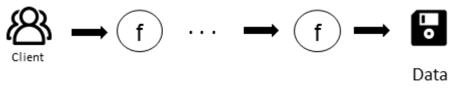
Data Lake [S4] [S5]: *Problem:* Keeping up with evolving requirements of data transformation and processing can be a hassle.
Solution: The data lake is a physical storage for raw data where data is processed and deleted the least possible. Organizing it with sensible metadata naming as times is a must for keeping order.
Benefits: The data remains always the same independently from the needs of the moment. It can be transformed just in time as necessary.



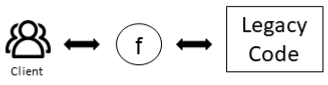
Fan-In/Fan-Out [S12][S8](McGrath et al., 2016)[S2][S4] (also known as "Virtual Actors" [S6], "Data transformation" [S7], "Processor" [S9], "Fire triggers and transformations" [S3]):
Problem: Enable the execution of long tasks that exceed the maximum execution time (similar to Function Chain).
Solution: Split the work parallel tasks and aggregate the results in the end. The parallel execution leads to faster completion.
Issues: Strong coupling between the chained functions. As for function chain, splitting the tasks between functions can be complex [S10].



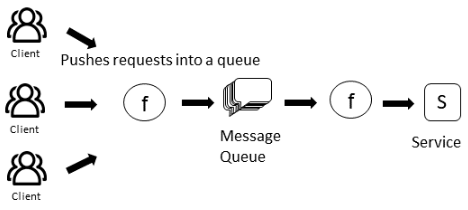
Function chain [S10][S3]:
Problem: Enable to execute long tasks that exceed the maximum execution time (e.g., longer than 15 minutes in Lambda). *Solution:* Combine functions in a chain. An initial function starts the computation while keeping track of the remaining execution time. Before reaching the maximum execution time, the function invokes another function asynchronously, passing each parameter needed to continue the computation. The initial function can then terminated without affecting the next function in the chain. *Issues:* Strong coupling between the chained functions, increased number of functions. Splitting the tasks between functions can be complex [S10].



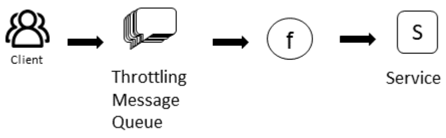
Proxy [S11] (also known as "Command pattern" [S13], "Anti-Corruption Layer" [S3]): *Problem:* Integration of functions with a legacy system. *Solution:* Create a function that acts as a proxy for another service, handling any necessary protocol or data format translation. *Benefit:* Clean and easy-to-access API for clients.



Queue-Based Load Leveling [S8][S2][S3] (also known as "The Scalable Webhook) [S12], "The Throttler" [S1]): *Problem* Building scalable webhooks with non-scalable back-ends. Webhooks enable to augment or alter the behavior of a web page, or web application, with custom callbacks. *Solution:* Similar to the frugal consumer, queue service to trigger a function can be used, which allows queueing the requests under heavy load.



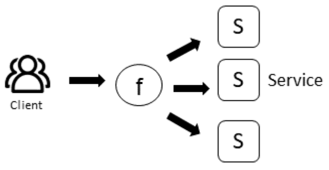
The Frugal Consumer [S12]: *Problem:* Increase scalability of non-scalable backends. *Solution:* A function that processes the requests of multiple services (or functions) that post messages directly to a message queue.



The Internal API [S12]: *Problem:* Accessing microservices that are only accessed within the cloud infrastructure. *Solution:* Leave the API Gateway and call the functions HTTP directly using a Invocation Type. *Benefits:* Increased security as services are not accessible from outside.

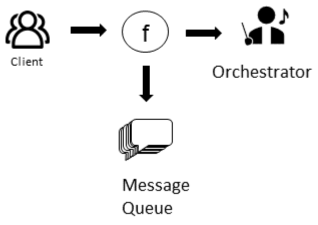
The Robust API [S2][S12] (also known as "The Gateway" [S3]): *Problem:* Sometimes clients know which services in the back-end they want to use. *Solution:* Use an API Gateway to grant access for clients to selected services. *Benefits:* Allows handling more individually clients. *Issues:* Increases complexity.

The Router [S12] (also known as "Routing Function" [S13], "Decoupled Messaging" [S2], "Data probing" [S9]): *Problem:* distribute the execution based on payload, without paying the extra cost of orchestration systems adopted in the state machine pattern. *Solution:* Create a function that acts as a router, receiving the requests and invokes the related functions based on payload. *Benefit:* Easy implementation. *Issues:* the routing function needs to be maintained. Moreover, it can introduce performance bottlenecks and be a single point of failure. Double billing, since the routing function needs to wait until the target function terminates the execution.



Thick Client [S13]: *Problem:* Any intermediary layer between client and service increases costs and latency. *Solution:* Allow clients to directly access services and orchestrate workflows. *Benefits:* Increased performances, reduced cost at server-side, increased separation of concerns (Roberts, 2016).

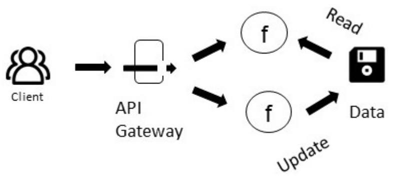
The State Machine [S7] [S2] [S12]: *Problem:* Orchestration and coordination of functions. *Solution:* Adoption of serverless orchestration system such as AWS Step Functions[2], Azure Durable Functions[3], or IBM Composer[4] to orchestrate complex tasks. *Issue:* The complexity of the system increases, as well we the development effort.



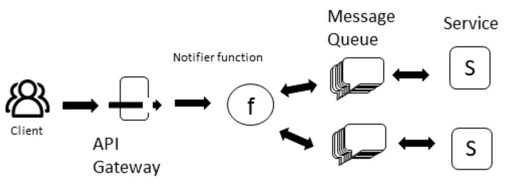
4.2 Event-Management

These patterns help to solve communication problems.

Responsibility Segregation [S3]: *Problem:* When the same functions are used for queries and data updates, it increases the risk of becoming inflexible. *Solution:* Segregate functions that update and read from data sources. Use "Commands and Queries" for the appropriate function to avoid this congestion.



Distributed Trigger [S12] (also known as "Event Broadcast" [S13]): *Problem:* Coupling a message queue topic only with its own service. *Solution:* Couple multiple services into a single notification function, possible via message queues. This setup works well if the topics have only a single purpose and don't need any data outside of it's micro-service. *Issues:* The subscriptions to the queue topic remains the responsibility of the individual services.



FIFO [S12], [S13]: *Problem:* Create a FIFO Queue for serverless functions. Several messages do not work with a FIFO approach (first in, first out). *Solution:* Use a crontab such as AWS Cloudwatch that periodically invokes the function asynchronously. Then, set the function's concurrency to 1 so that there are no attempts to run competing requests in parallel.

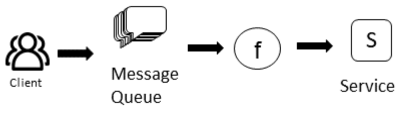
[2]AWS Step Functions https://tinyurl.com/ycu84so5

[3]Azure Durable Functions https://tinyurl.com/uscla27

[4]IBM Composer https://tinyurl.com/s6b6rbl

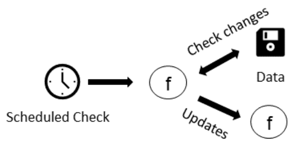
The function polls the queue for (up to 10) ordered messages and does whatever processing it needs to do. Once the processing is complete, the function removes the messages from the queue and then invokes itself again (asynchronously). This process will repeat until all the items have been removed from the queue. *Benefits:* Simple sequentialization. *Issues:* Cascading effect [S12]: if the function is busy processing other messages, the cronjob will fail because of the concurrency setting. If the self-invocation is blocked, the retry will continue the cascade.

The Internal Hand-off [S12]: *Problem:* Use invocation Type (Event) for an asynchronous event. *Solution:* Function stops automatically when the execution is finished and automatically retries when it needs to. Using a message queue to attach a Dead Letter Queue enables to capture failures.



Periodic Invoker [S7]: *Problem:* Execute tasks periodically. *Solution:* Subscribe the function to a scheduler, such as AWS Cloud Watch, Google Cloud Scheduler, or Azure Scheduler. *Benefits:* Run functions periodically without the need to keep them permanently alive.

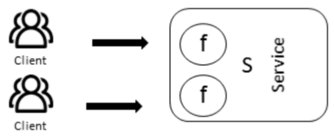
Polling Event Processor [S11] (also known as "Polling consumer" [S22]): *Problem:* React to changes of states of external systems that do not publish events *Solution:* Use the Periodic Invoker pattern to check the state of the service *Benefits:* Run functions periodically without the need to keep a function permanently alive as a listener.



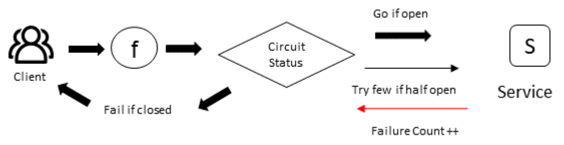
4.3 Availability Patterns

This group of patterns helps to solve availability problems, reducing the warm-up time, and possible failures.

Bulkhead [S20], [S3]: *Problem:* When a crucial, maybe load heavy, function fails, the complete system risks being compromised. *Solution:* Partitioning workloads into different pools. These pools can be created on the base of consumer load or availability. *Benefits:* This process isolates failure and reduces risks of a chain reaction of failures.



Circuit breaker [S12][S14]: *Problem:* Keeps track of failed or slow API calls. *Solution:* When the number of failures reaches a certain threshold, "open" the circuit sends errors back to the calling client immediately without even trying to call the API. After a short timeout, the system "half open" the circuit, sending just a few requests through to see if the API is finally responding correctly. All other requests receive an error. If the sample requests are successful, the system "close" the circuit and start letting all traffic through. However, if some or all of those requests fail, the circuit is opened again.*Benefits:* Cost saving for synchronous requests.

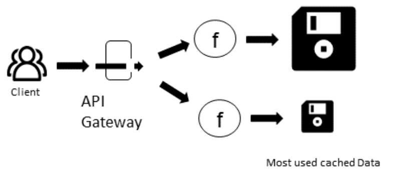


Compiled Functions [S15]: *Problem:* Serverless cloud computing would be a perfect fit for IoT especially at Edge would it not be so heavy-weight in memory footprint and invocation time. *Solution:* A high level specialized ahead of time compiled Serverless language can reduce the memory footprint and invocation time. This might make Edge technology in the cloud viable.

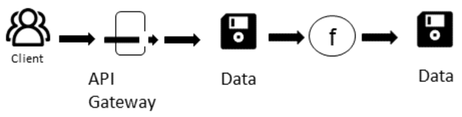
Function warmer[S12][S16][S23] (also known as "Function Pinging" [S10], "Warmer service" [S9], "Cold Start" [S21], "Keeping Functions Warm" [S8], "keep-alive" [S3]): *Problem:* Reduction of cold start time, the delay between the execution of a function after someone invokes it. Serverless functions are executed in containers that encapsulate and execute them. When they are invoked, the container keeps on running only for a certain time period after the execution of the function (warm) and if another request comes in before the shutdown, the request is served instantaneously. Cold start takes between 1 and 3 seconds [S2][S23]. For example, AWS (Shilkov, 19 b) and Azure (Shilkov., 19 a) recycle idle function instances after a fixed period of 10 and 20 minutes respectively. *Solution:* Ping the function periodically to keep it warm. *Benefit:* Reduction of response times from 3 seconds to 200 milliseconds. *Issues:* Increased cost, even if limited to only one call every 10-15 minutes.

Oversized Function [S10]: *Problem:* In Serverless it is not possible to choose on which CPU runs. *Solution:* Asking for bigger memory grants a faster virtual machine too, even if no more memory is required.

Read-heavy report engine [S12][S2]: *Problem:* Overcome the limits of downstream limits of read-intensive applications. *Solution:* Usage of data caches and the creation of specialized views of the data most frequently queried *Benefits* Increased performances.



The Eventually Consistent [S12]: *Problem:* Replicate data between services to keep them consistent. *Solution:* Use database stream services (e.g. DynamoDB stream) to trigger events made on the database from previous functions and use the data again for whatever needed.

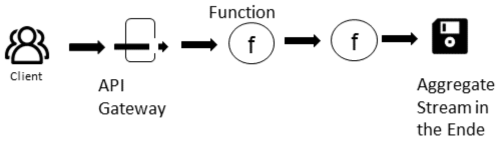


Timeout [S17]: *Problem:* The timeout time for API Gateway is 29 seconds. Which is a long time for a user and makes for a bad experience with the service. *Solution:* Reduce the timeout to a shorter span, preferably around 3-6 seconds.

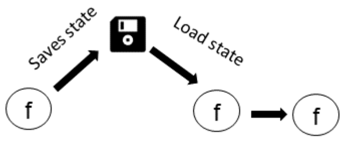
4.4 Communication Patterns

Here, we describe patterns to communicate between functions.

Data Streaming (also known as "Stream and pipeline" [S2], "I am a streamer" [S12] and "Event Processor" [S13][S22], "Streaming Data Ingestion" [S4], "Stream processing" [S3]): *Problem:* Manage continuous stream of data. *Solution:* Serverless platforms offer possibilities like Kinesis(AWS) to handle and distribute large streams of data to services. *Issues:*Data streams can be expensive in Serverless. Working outside the platforms eco system can be difficult too.



Externalized state [S10] [S3] (also known as "Share State" [S21]). *Problem:* In some cases, it is needed to share the state between functions. *Solution:* Share the state saving it into an external database. *Issues:* High coupling between the functions, latency overhead [S10], additional programming effort. *Solution:* "*Apply a continuous stream processor that captures large volumes of events or data, and distributes them to different services or data stores as fast as they come.*" [S2] Romero [S2] and Dali [S12] propose an AWS specific example, using the AWS API Gateway as a Kinesis proxy[5]. In this way, it is possible to use any number of services to pipe data to a Kinesis stream. Finally, Kinesis can be used to aggregate the results.

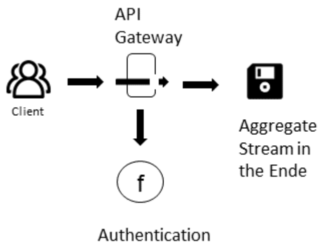


Publish/Subscribe [S2][S3][S18] (also known as "The Notifier"[S12]): *Problem:* Forward data for internal services (or APIs). *Solution:* Use a standalone topic in the message queue to distribute internal notifications for internal services.

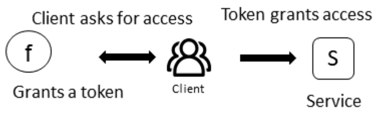
4.5 Authorization Patterns

These deal with user authorization problems.

The Gatekeeper [S3][S12]: *Problem:* Authorize Functions. *Solution:* Use a Gateway to create an authorizer function that processes the authorization header and returns the authorization policy.



Valet key [S19]: *Problem:* Authorization without routing all the traffic through a gatekeeper process. *Solution:* By requesting first access from a special authorizer serverless function it is granted a token which is valid for a certain period of time and access rights.



[5]Create a REST API as an Amazon Kinesis Proxy in API Gateway https://docs.aws.amazon.com/apigateway/latest/developerguide/integrating-api-with-aws-services-kinesis.html

Table 3: The patterns identified

Pattern	Peer Reviewed studies	Gray Literature
Orchestration and Aggregation		
Aggregator		[S1], [S2]. Also named: Durable functions [S3]
Data Lake		[S4] [S5]
Fan in/Fan out	Also named: Virtual Actors [S6], Data transformation [S7]	[S2] [S4][S8]. Also named: Processor [S9], Fire triggers and transformations [S3]
Function chain	[S10]	[S3]
Proxy		[S11]. Also named: Command pattern [S13][S24]
Queue-Based Load Leveling		[S2][S3]
The Frugal consumer		[S12]
The Internal API		[S12]
The Robust API		[S2][S12]
The Router	Also named: Routing Function [S10]	[S12]. Also named: Decoupled Messaging [S2], Data probing [S9], Routing function [S13][S11]
The State machine		[S12]
Thick Client		[S13]
Event Management		
Responsibility Segregation		[S3]
Distributed Trigger		[S12]
FIFO		[S12][S3]
Internal Handoff		[S12]
Periodic Invoker	[S7]	
Polling Event Processor		[S11]. Also named: Polling consumer [S22]
Availability		
Bulkhead		[S20][S3]
Circuit Breaker		[S12], [S14]
Compiled Functions	[S15]	
Function Warmer	Also named: Function pinging [S10][S16]	[S12] [S23]. Also named: Warmer service [S9], Cold Start [S21], Keeping Functions Warm [S8], keep-alive [S3]
Oversized Function	[S10]	
Read-heavy report engine		[S12] [S2]
The eventually consistent		[S12]
Timeout		[S17]
Communication		
Data streaming		They Say I am A Streamer [S12], Streams and Pipelines [S2] Streaming Data Ingestion [S4], Stream processing [S3]
Externalized state	[S10]	Share State [S21][S3]
Pub/Sub		[S2] [S18][S3] Also named: Notifier [S12]
Authorization		
The Gatekeeper		[S12], [S3]
The Gateway		[S12][S3]
Valet key	[S19]	

5 Discussion

The results of this work highlight that serverless patterns are still not clear. Different practitioners propose similar and often discordant patterns, to solve similar problems.

Some patterns are mentioned only by one source while others are mentioned by different sources, confirming the importance given by the practitioners.

While the continuous evolution of serverless platforms, such as the introduction of new tools, can be beneficial for practitioners, it probably confuses practitioners to understand which solution is more beneficial to adopt. As an example, AWS lambda adapted their queue service (SQS) to enable FIFO messages and therefore it is no longer necessary to manage your re-drive policies (Frugal Consumer pattern). However, up to now, FIFO messages still need to be manually managed in Azure. Other tools have been introduced in the last years, and will probably continue to be introduced in the future, increasing the decision complexity when deciding which pattern to adopt.

It is interesting to notice that several patterns have been created to work around serverless limitations. As an example, the function warmer has been created to circumvent the long startup problem of functions. Other patterns are inherited from other cloud-domains or monolithic systems. As an example, the gatekeeper and valet key are also used in microservices.

Thanks to this work, practitioners will be able to access the description of all patterns in one report, together with an analysis of the perceived usefulness of each pattern, and possible problems that can raise using them. Researchers can further validate the patterns, considering their usefulness or extending them.

5.1 Threats to Validity

The results of a MLR may be subject to validity threats, mainly concerning the correctness and completeness of the survey, and the subjectivity of interpretations. Moreover, the fast-evolving domain of serverless, might also influence the results of this work.

- *Evolution Pace:* Because of the fast evolution of the serverless topic, the list of patterns might become outdated very quickly. Some patterns might be implemented directly by tools provided the cloud providers, while other patterns might be proposed in the future.
- *Correctness and Completeness* A first issue is a possible bias in the selection of GL's. To limit this bias, we followed the guidelines for Multivocal Literature Reviews proposed by Garousi et al. (Garousi et al., 2019). A first measure to limit the bias in the selection of GL's regarded the search for relevant sources. To this end,
  - We perform the search for grey literature in Google and for scientific literature with automatic searches in multiple digital libraries.
  - We applied a broad search string, which resulted in a large set of results. The fact that finally less than one-tenth of the retrieved GLs were selected for the survey demonstrates that the search string allowed for quite a broad search.
  - We applied the snowballing process to identify GLs that may have not detected by the automatic search.

  To increase the confidence that all relevant sources were identified, we did not rely exclusively on titles and abstracts to determine whether the article reported serverless patterns. Instead, three authors carefully read the full text independently. In case of disagreements, the GLs were reviewed collaboratively.
- *Subjectivity* In general, assessing the relevance of GL's in a review is partly subjective. In our case, the main subjectivity issue concerns the identification of the patterns. As already mentioned, different GL's address different patterns or investigate how to solve different problems. Therefore, we faced the problem of deciding what GL's could be considered as solving the same problem at some level of abstraction.
- *Pattern Classification* is based on our experience with cloud-native systems. A different classification might have resulted in a different number of patterns. However, due to similarity in tools and techniques between serverless and cloud-native, some transferability can be assumed.

5.2 Trends and Open Issues

Serverless moves us towards continuous development and delivery. An important observation is that, differently than in Microservices, serverless-based applications do not require to develop a full application stack. That means that their infrastructure resources like data stores and networks don't need to be managed, as they are under the responsibility of the cloud provider. Resulting from our study, but also discussions in Leitner et al. [S10], we can identify the following emerging issues:

- Comparison between microservices and serverless functions. A microservice can be composed of one or more serverless functions. However, how to combine functions into a complete application or into a microservice is still not clear. Researchers might support practitioners by proposing the adoption of previously developed techniques for aggregating distributed systems or basic software engineering techniques.

- Lack of stable tools. The actual state of tools for supporting serverless development is still limited. Different tools are continuously proposed on the market increasing the complexity of decisions for long-term development.
- Reuse of Functions. What happens once a growing system has thousands or millions of functions is still not clear. Will it be possible to have a good system understandability with such a complex system? Grouping functions into isolated microservices might help, but at the moment it is still not clear how to proceed.
- Negative Results. In which contexts do serverless, and in particular some specific patterns turn out to be counterproductive? Are there anti-patterns? All the aforementioned points require more experience reports and empirical investigations.

6 Conclusion

As the number of companies developing and deploying serverless applications continues to rise, a natural question that everyone joining this direction asks is what are the best practices or the design patterns to follow to make the best of this novel technology. So far, most companies have been exploring the new territory by applying known methods and patterns coming from well-established technologies (e.g., microservices, web services). However, as serverless technology is becoming mainstream, two classes of patterns start to emerge. The first class represents existing patterns that have been adapted from existing technologies to fit the serverless paradigm, others instead have been developed specifically to address serverless implementation needs.

Practitioners proposed several patterns, and the community is already aware of several of them. We can observe here a community in an early adoption stage. However, existing reports highlight the value of these patterns, although complexities introduced by serverless are only slowly emerging. In any way, the pattern catalog that we extracted from the literature and has been confirmed by the survey provides a valuable basis for practitioners and researchers alike.

In this fast-evolving technology context, more tools will be developed, that might require either new patterns or turn those that are work-around for limitations obsolete. Here, a continuous update of any pattern catalog is necessary.

Future work includes the validation of the actual usefulness of these patterns in more concrete application contexts as well as the identification of anti-patterns.

REFERENCES

Al-Ameen, M. and Spillner, J. (2018). Systematic and open exploration of faas and serverless computing research. In *European Symposium on Serverless Computing and Applications*.

Alqaryouti, O., Siyam, N., et al. (2018). Serverless computing and scheduling tasks on cloud: A review. *American Scientific Research Journal for Engineering, Technology, and Sciences (ASRJETS)*, 40(1):235–247.

Erl, T. (2008). *SOA Design Patterns (paperback)*. Pearson Education.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. M. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition.

Garousi, V., Felderer, M., and Mäntylä, M. V. (2019). Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. *Inf. and Software Technology*, 106:101 – 121.

Kuhlenkamp, J. and Werner, S. (2018). Benchmarking faas platforms: Call for community participation. In *Int. Conf. on Utility and Cloud Computing*.

Leitner, P., Wittern, E., Spillner, J., and Hummer, W. (2019). A mixed-method empirical study of function-as-a-service software development in industrial practice. *Journal of Systems and Software*, 149:340 – 359.

Lloyd, W., Ramesh, S., Chinthalapati, S., Ly, L., and Pallickara, S. (2018). Serverless computing: An investigation of factors influencing microservice performance. In *Int. Conf. on Cloud Engineering (IC2E)*, pages 159–169.

Lynn, T., Rosati, P., Lejeune, A., and Emeakaroha, V. (2017). A preliminary review of enterprise serverless cloud computing (function-as-a-service) platforms. In *Int. Conf. on Cloud Computing Technology and Science (CloudCom)*, pages 162–169.

McGrath, Garrett, M., Short, J., Ennis, S., Judson, B., and Brenner, P. (2016). Cloud event programming paradigms: Applications and analysis. *IEEE Computer Society*.

Nupponen, J. and Taibi, D. (2020). Serverless: What it is,what to do and what not to do. In *International Conference on Software Architecture (ICSA 2020)*.

Pahl, C., El Ioini, N., et al. (2019). Blockchain based service continuity in mobile edge computing. In *2019 Sixth Int. Conf. on Internet of Things: Systems, Management and Security (IOTSMS)*, pages 136–141. IEEE.

Roberts, M. (2016). Serverless architectures. https://martinfowler.com/articles/serverless.html.

Sadaqat, M., Colomo-Palacios, R., and Knudsen, L. E. S. (2018). Serverless computing: a multivocal literature review. *Nokobit*.

Shilkov., M. (2019 a). When does cold start happen on azure functions? https://mikhail.io/serverless/coldstarts/azure/intervals/.

Shilkov, M. (2019 b). When does cold start happen on aws lambda? https://mikhail.io/serverless/coldstarts/aws/intervals/.

Taibi, D., Lenarduzzi, V., and Pahl, C. (2018). Architectural patterns for microservices: a systematic mapping study. *Int. Conf. on Cloud Computing and Services Science (CLOSER2018)*.

Taibi, D., Lenarduzzi, V., and Pahl, C. (2019a). Microservices anti-patterns: A taxonomy. *Microservices - Science and Engineering. Springer. 2019*.

Taibi, D., Lenarduzzi, V., and Pahl, C. (2019b). Microservices architectural, code and organizational anti-patterns. *Communications in Computer and Information Science*, pages 126–151.

References: The Selected Papers

[S1] Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A., and Suter, P. (2017). Serverless Computing: Current Trends and Open Problems.

[S2] Romero, E. (2019). Serverless microservice patterns for aws. https://medium:com/@eduardoromero/serverless-architectural-patterns-261d8743020/.

[S3] Likness, J. (2018). Serverless apps: Architecture, patterns, and Azure implementation. Microsoft Developer Division, .NET, and Visual Studio product teams.

[S4] Serverless architectural patterns and best practices (arc305-r2) - aws re:invent 2018. https://www.slideshare.net/AmazonWebServices/serverless-architectural-patterns-and-best-practices-arc305r2-aws-reinvent-2018.

[S5] Benghiat, G. (2017). The data lake is a design pattern. https://medium.com/data-ops/the-data-lake-is-a-design-pattern-888323323c66/.

[S6] Bernstein, P. A., Porter, T., Potharaju, R., Tomsic, A. Z., Venkataraman, S., and Wu, W. (2019). Serverless event-stream processing over virtual actors. In CIDR.

[S7] Hong, S., Srivastava, A., Shambrook, W., and Dumitras, T.(2018). Go serverless: securing cloud via serverless design patterns. In 10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18).

[S8] Zambrano, B. (2018). Serverless Design Patterns and Best Practices: Build, secure, and deploy enterprise ready serverless applications with AWS to improve developer productivity. Packt Publishing Ltd.

[S9] Shafiei, H., Khonsari, A., and Mousavi, P. (2020). Serverless computing: A survey of opportunities, challenges and applications.

[S10] Leitner, P., Wittern, E., Spillner, J., and Hummer, W. (2019). A mixed-method empirical study of function-as-a-service software development in industrial practice. Journal of Systems and Software, 149:340 – 359.

[S11] Pekkala, A. (2019). Migrating a web application to serverless architecture. Master's Thesis in Information Technology, University of Jyväskylä.

[S12] Daly, J. (2019). Serverless microservice patterns for aws. https://www.jeremydaly.com/serverless-microservice-patterns-for-aws.

[S13] Sbarski, P. (2017). Serverless Architectures on AWS . Manning.

[S14] Nygard, M. T. (2007). Release It!: Design and Deploy Production-Ready Software (Pragmatic Programmers). 1 edition.

[S15] Gadepalli, P. K., Peach, G., Cherkasova, L., Aitken, R., and Parmer, G. Challenges and opportunities for efficient serverless computing at the edge.

[S16] Bardsley, D., Ryan, L. M., and Howard, J. (2018). Serverless performance and optimization strategies. 2018 IEEE International Conference on Smart Cloud (SmartCloud).

[S17] Lumigo (2019). Aws lambda timeout best practices. https://lumigo:io/blog/aws-lambdatimeout-best-practices/.

[S18] Pirtle, J. (2019). 10 things serverless architects should know. https://aws:amazon:com/blogs/architecture/ten-things-serverlessarchitects-should-know/.

[S19] Adzic, G. and Chatley, R. (2017). Serverless computing: Economic and architectural impact. In Joint Meeting on Foundations of Software Engineering , ESEC/FSE 2017, page 884–889.

[S20] AWS, A. (2018). Serverless application lens aws. https://d1.awsstatic.com/whitepapers/architecture/AWS-Serverless-Applications-Lens.pdf.

[S21] Group, I. S. (2019). Aws lambda serverless coding best practices.https://www:intentsg:com/awslambda-serverless-coding-best-practices/.

[S22] Gregor, H. and Woolf, B. (2004). Enterprise integration patterns: Designing, building, and deploying messaging solutions. 1 edition.

[S23] Bhojwani, R. (2019). Aws lambda timeout best practices. https://lumigo:io/blog/aws-lambdatimeout-best-practices/.

[S24] Rabbah, R., Mitchell, N. M., Fink, S., and Tardieu, O. L.(2019). Serverless composition of functions into applications. US Patent App. 15/725,756

ACKNOWLEDGEMENTS

This research was partially supported by the grant ”SHAPIT” awarded by the Ulla Tuominen Foundation (Finland)