Building a Pattern Language for Serverless Architectures

LEANDRO RODRIGUES DA SILVA, Universidade de São Paulo, Brazil
JOÃO FRANCISCO LINO DANIEL, Universidade de São Paulo, Brazil
ALFREDO GOLDMAN, Universidade de São Paulo, Brazil

Serverless computing is a powerful paradigm that is becoming popular in the software engineering community. However, there is still a lack of knowledge in how to design and implement robust architectures using serverless services. To assist practitioners to take better informed decisions regarding serverless, we intend to create a useful tool for reasoning about serverless architecture. In this work, we present a novel serverless pattern language, as a means to such goal. The pattern language comprehends 28 patterns, classified among 5 groups: availability, orchestration aggregation, communication, event management and authorization.

CCS Concepts: • **Software and its engineering → Software architectures**.

Additional Key Words and Phrases: architectural patterns, serverless, language pattern

ACM Reference Format:
Leandro Rodrigues da Silva, João Francisco Lino Daniel, and Alfredo Goldman. 2022. Building a Pattern Language for Serverless Architectures. In . ACM, New York, NY, USA, 13 pages.

1 INTRODUCTION

Over the years, the way we architect software has been changing. Since the problems to be solved have become more and more complex, different approaches have been adopted to design complex systems, one of them being the service-oriented architecture (SOA) [5].

SOA is an architectural approach in which developers create autonomous services that each have their own well-defined responsibilities and communicate between them to perform tasks. The modern applications based in SOA are often referred to as microservice architecture. These applications are composed of services that are small, standalone, fully independent built around a particular business purpose [5]. This model emerged as an alternative to solve the problems of monolithic architectures, aiming to make it easier to maintain or rewrite software [3].

The conventional way of building microservices is based on a server approach, where developers create servers that will be running in a data center or in the cloud. With this approach, engineers will need to be responsible for managing, patching and maintaining these servers. Also, this leads to a possible high-cost hosting strategy, where you pay for each second the application is running, no matter if it's doing a job or just waiting for requests.

Serverless services can be defined as a specific type of service where the server infrastructure is hidden from the developer and incurs costs only by usage [5]. In this model, a cloud platform is responsible for dealing with provisioning, managing and maintaining the infrastructure on behalf of the server, so that the developers are responsible for plugging the business code into this structure. However, there is still a gap in the area of understanding and formalizing serverless architectures. A useful way to structure architectural decisions is to adopt a set of patterns and build a pattern language.

Manuscript submitted to ACM

In order to help developers identify the most appropriate patterns to use, we aim to identify and characterize these patterns and its relationships in a pattern language. To analyze these patterns, we selected a set of them from different literature and analyzed its relationships using the approach suggested by Chris Richardson's pattern language for microservices [4]. The relationships proposed by Chris Richardson are: motivating/solution patterns, patterns that are solutions to the same problem, and generic/specific patterns.

2 BACKGROUND

Before the emergence of cloud computing, the most common way of building and deploying software was the **on-premises** model, meaning that the owners of the software were responsible for buying, setting, managing and maintaining the hardware that would host the application. After that, a new model arrived: **cloud computing**.

Cloud computing is the on-demand delivery of IT resources over the internet [5]. A service provider manages the infrastructure and offers access to resources on top of it over the Internet. The developers can access it through an Application Programming Interface (API). All the complexity of managing hardware and space is abstracted by the providers, such as AWS, Azure and Google Cloud. Cloud computing has several classifications, being the most popular:

- **Infrastructure as a Software (IaaS)**: Offers fundamental resources to build software, like computing, storage, network and virtual servers [6].
- **Platform as a Service (PaaS)**: Provides a platform to deploy software, with hardware, software and infrastructure, without the cost and complexity with dealing with these resources [6].
- **Software as a Service (SaaS)**: Combines the two previous classifications. Offers software and applications through the cloud. The clients do not need to deal with installing, managing or upgrading these applications [6].

Serverless is a cloud-native execution model for building and running applications without server management. In this model, the cloud platform is responsible for dealing with provisioning, managing and maintaining the infrastructure on behalf of the server, so that the developers are responsible for plugging the business code into this structure.

In this execution model, a concept commonly used is **Function as a Service (FaaS)**. FaaS is a type of cloud service that allows developers to deploy small applications (or parts of a bigger one). The cloud platform receives a package containing the code and wraps it into a stateless container, abstracting the logic of allocating resources, scheduling tasks, setting up protocols and mainly, setting up a server. In services like this - such as done by AWS Lambda - the cost of maintaining an application is tied to how many executions a function receives. In this work, we'll refer to structures that run in FaaS services simply by **functions**. Also, other built-in solutions are offered as serverless solutions, such as databases, API gateways, data lakes, queues, etc.

3 MATERIALS AND METHODS

Given the novelty aspect of Serverless and its patterns, in this work we aim to provide to practitioners a useful tool to better understanding patterns of serverless architecture. With that, there might be an improvement in the decision-making for the adoption of serverless solutions in an application's architecture.

To achieve such goal, in this work, we present a pattern language for the serverless architectural style. To craft the pattern language, first we conducted an exploration to come up with a list of serverless patterns. Such list was material for the grouping and relating that lead to a novel pattern language for serverless.

Initially, we explored either gray and white literature to create a list of patterns documented for serverless architecture. The complete list of patterns with their descriptions can be found in Appendix A. The main source of knowledge

was Taibi et al. (2020) [2], which presented a collection of 32 serverless patterns along with a classification for them: "availability", "orchestration and aggregation", "event management", "communication", and "authorization". There were other interesting gray literature resources, such as Jeremy (2018) [1]. Nonetheless, the gray literature presents patterns and practices considerably more coupled with cloud providers.

Once we gathered a comprehensive list of serverless patterns, with a few classifications, we searched for relationships between different patterns. Richardson (2018) [4] played a major role as inspiration for it, with "Motivating pattern/Solution pattern", "Solution A / Solution B", and "General / Specific". We explore further details about our adoption of these relationships in Subsec. 4.1. The classification and the relationships were the base for the creation of the proposed serverless pattern language, described in Subsec. 4.1.

The patterns were reviewed and filtered using the following criteria to disregard:

- Out-of-date: patterns that were tied to a limitation of a cloud platform that is no more a reality.
- Out-of-scope: patterns that were specified to be used by cloud platforms' back-end.
- Platform-specific: patterns that are tied to a specific cloud platform.

Using these criteria, we disregarded 1 pattern for being out-of-date, 1 for being platform-specific and 2 for being out-of-scope.

4 SERVERLESS PATTERN LANGUAGE

4.1 Language

In this section, we propose our pattern language for serverless architectures. To do so, we're going to analyze the relationships between patterns illustrated in the diagram [1]. The patterns are divided using the categories proposed by [2].

The main goal of this analysis is to relate patterns in a way that a practitioner that will apply these patterns can: **a) Observe the trade-offs**: Analyze the trade-offs regarding each decision. And **b) Combine patterns**: use both patterns in a combined way to produce a more powerful and reusable solution. Also, we categorized these patterns in subsets as the ones proposed by [2].

We'll relate the patterns in our language using the relationship types proposed by Chris Richardson [4]:

- **Motivating and solution patterns:** relates a pattern X to a pattern Y, where the solution of X motivates a new problem solved by Y.
- **Solutions to the same problem:** relates a pattern X to a pattern Y, where both patterns share a context and propose different solutions to the same problem.
- **Generic and specific patterns:** relates a pattern X to a pattern Y, where X is a specification of Y for a given context.

I. **Bulkhead as alternative to Circuit-Breaker**: These two patterns propose solutions to the problem of having a component that is a single point of failure in a system. Bulkhead does this with a replication approach, having multiple instances of the component in separate pools. Circuit-breaker uses another way to deal with that: isolating the broken component while it recovers, without replication.
Use case: Say you're building an e-commerce back-end service that fetches and returns offers from a database. To do so, it provides endpoints to serverless functions to clients to retrieve this data. Since this component can be a single point of failure for this e-commerce website, you wish it to be as much available as possible. You could choose to provide these functions using the Bulkhead pattern, meaning that you would provide different pools

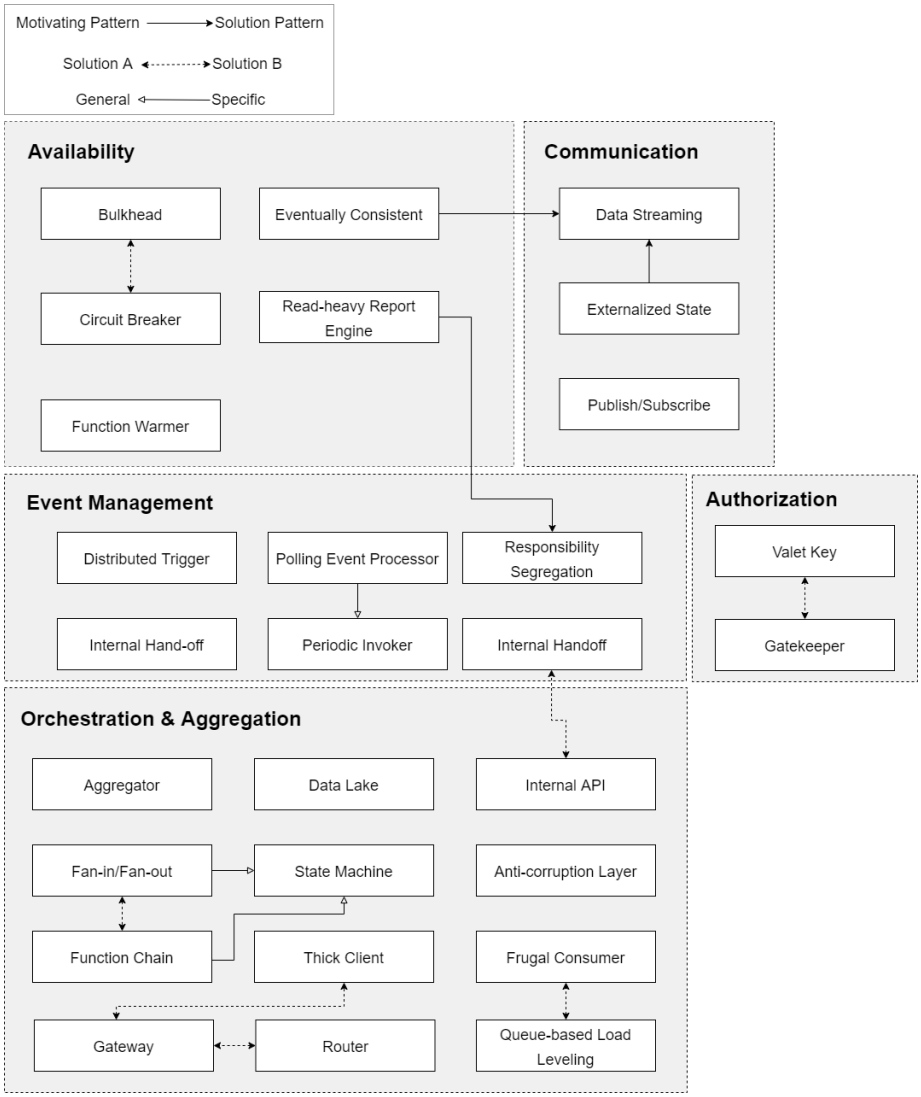

Fig. 1. Diagram containing a pattern language for serverless architectures

of functions to different clients, replicating them in different and isolated pools, so that a failed pool would not impact all the clients. On the other hand, you could implement a Circuit Breaker using a state-machine manager (e.g., AWS Step Functions, Azure Functions), so that you could prevent failing functions to be called when it's recovering from a catastrophic event. Both solutions would fit for the purpose, and could be implemented together to achieve best results. [2]

Trade-off Analysis: Bulkhead is a simpler solution to implement but leads to an increased cost, since it provides replicated functions to offer availability. Circuit-Breaker, by the other side, provides a non-trivial solution, but the cost will be only an extra function that will manage the state of the circuit.

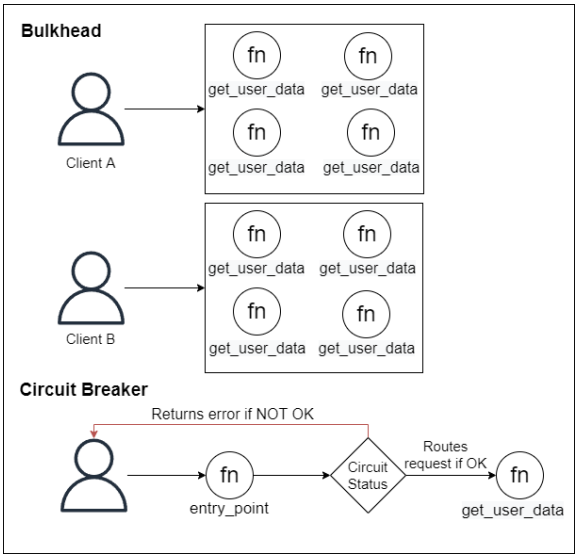

Fig. 2. Bulkhead and Circuit-Breaker patterns

II. **Read-heavy Report Engine as motivation to Responsibility Segregation**: Read-heavy Report Engine depends on Responsibility Segregation in its essence. The first one needs the read operations to be fully separated of write operations, so that it can scale each one properly to provide a powerful read engine for heavy workloads.
Use case: Imagine you're building a social network back-end service that deals with storing and providing the posts of the feed to a user. To do so, you created serverless functions that will read and write the data to a database. By following the Pareto's principle, you could implement the Read-heavy Report Engine pattern and properly scale each of the functions to execute each type of task, for example, giving more resources to read functions that to write functions. To do so, you need that this logic to be precisely separated, as proposed by Responsibility Segregation pattern, by having separated functions for read and write operations. [3]

III. **Eventually Consistent as motivation to Data Streaming**: Eventually Consistent can make usage of Data Streaming solutions as the core method to replicate data across the multiple databases, specially when dealing with large volumes of data.
Use case: Suppose you're building a social network back-end service that stores the user data, such as name and photo. This data can be used to different services, such as by the one that returns the user profile to a web client or by the message chat service. To have this data available to all the services, you could have distributed databases, one per service and replicate the data it all over this databases, applying the Eventually Consistent pattern. To do so, a smart way would be to use a streaming service (e.g., AWS DynamoDB streams) that triggers an event whenever a row of the table is updated, and then send it to a function that replicates this data across the other databases.[4]

IV. **Externalized State as motivation to Data Streaming**: Using Data Streaming is a powerful way to keep the state of Lambdas up to date in order to guarantee no inconsistent states.
Use case: Say you're building a game in which the user score needs to be shared by both, the game match and the global rank, and you have each of these components separated in two services. You probably want that the global

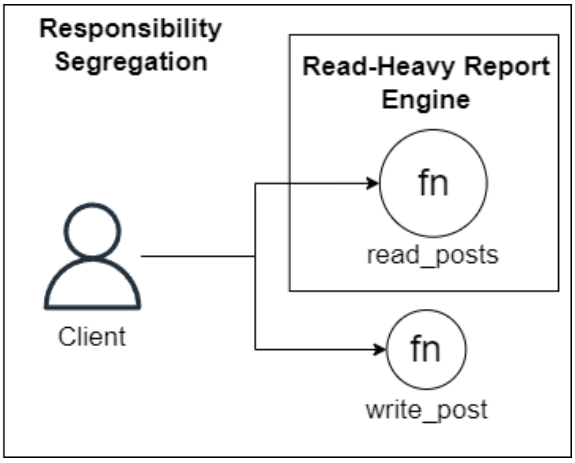

Fig. 3. Read-heavy Report engine and Responsibility Segregation patterns

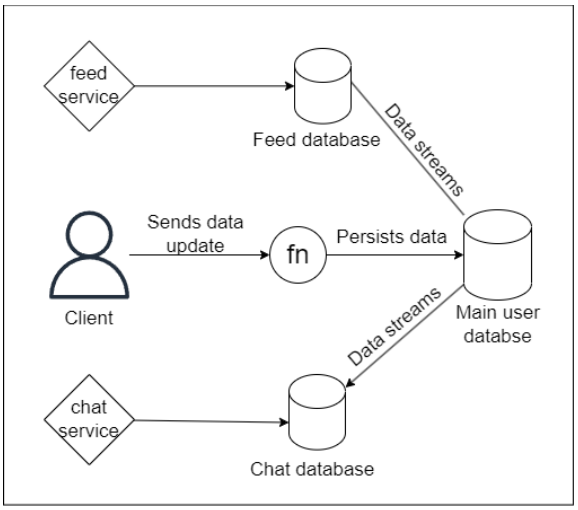

Fig. 4. Eventually Consistent and Data Streaming patterns

to be updated with the current score of the game, so you can use Data Streaming services (e.g., AWS Kinesis) to have an Externalized State of the game match execution.[5]

V. **Thick Client as alternative to Gateway** These two patterns solve the same problem: providing access to external clients to back-end services. Although, they are antagonists in the sense that Thick Client removes the abstraction that Gateway imposes with the top-layer functions that hides the endpoints of internal services.

Use Case: Imagine you're building an API to a web front-end music app. This API will have a serverless function that will fetch and return songs titles from a database. To access this API, the client can do this by calling an API Gateway endpoint or by directly invoking the function (Thick Client). In the first way, the clients will just need to know a single endpoint and the gateway will be responsible for routing the requests. In the second one, the clients

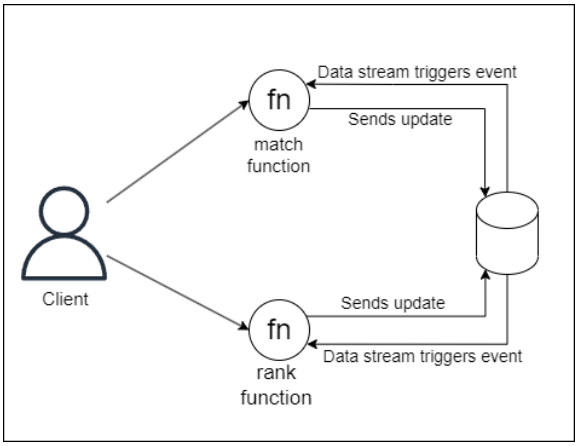

Fig. 5. Externalized state and Data Streaming patterns

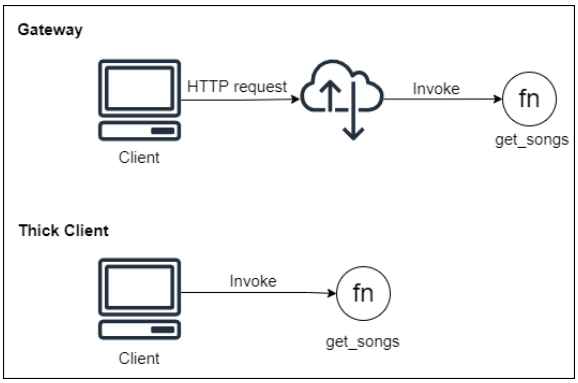

Fig. 6. Thick Client and Gateway patterns

will need to know how to properly invoke the function.[6]

Trade-off Analysis: Thick Client provides both lower latency and costs than Gateway, since it removes all intermediary layers between the client and the services, with the cost of increasing the complexity in client's code and potential security issues.

VI. **Internal API as alternative to Internal Handoff**: Internal API and Internal Handoff have similar solutions: a client calls directly the function, without an API Gateway over it. They differ in the type of request they use. Internal API proposes the usage of a synchronous HTTP request, while Internal Handoff proposes the usage of an asynchronous event.

Use Case: Suppose you're building a back-end system to serve an e-commerce. One of the tasks is to implement the payment service of the website. To do so, you can have a serverless function that receives that user's card digits and returns if the payment was concluded successfully. Since this is a complex operation that needs to call external services, such as the bank service, the user cannot wait for this task to finish, so you could call this function by using an asynchronous event, such as proposed by the Internal Handoff pattern. With this, the request is sent, the

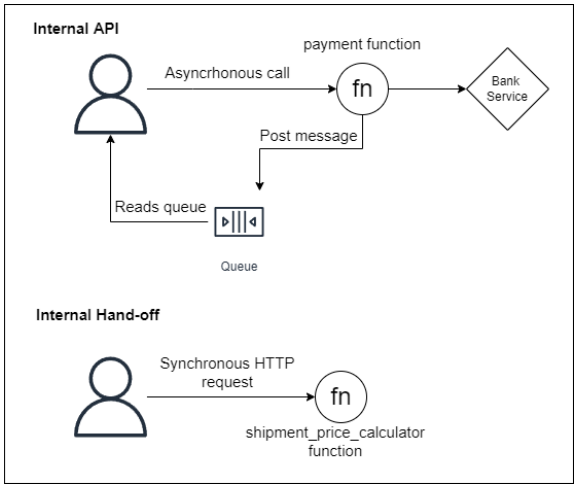

Fig. 7. Internal API and Internal Handoff patterns

payment task is executed, and the status will be returned at some point to the client through a message broker (e.g., AWS SQS, AWS SNS). On the other hand, you also need to implement a shipment price calculator. In this case, the client needs to know the price as fast as possible, so that he can finish the checkout. In this case, the usage of Internal API would fit better, since it would call the back-end shipment calculator function using a synchronous request and will return it to the front-end as soon as possible.

Trade-off Analysis: Calling synchronously a function is a better approach when the client needs a response instantly, such as in a web interface, while calling it asynchronously is a good choice if long tasks are going to be executed. Also, asynchronous calls lead to a more decoupled interaction.[7]

VII. **Fan-in/Fan-out as alternative to Function Chain**: Both patterns solves the problem of running long tasks in functions, but differ in the kind of problem they can be used to solve.

Use case: Say you're building an image processing software that improves the quality of each pixel of an image. To do so, you could break this task by sending each row of pixels of the image to a different function, and then aggregate the final result in a final image. This is a case where Fan-in/Fan-out pattern is useful. However, there are some cases where breaking the tasks in parallel executions can be hard. Imagine now that you're building an e-commerce payment service and you have two functions to: 1) Performs the payment and 2) Send the tracking ID via email. Breaking this task into these two executions in parallel could be tough and problematic, since a failure in performing the payment would need to stop the execution of the second task before it finishes. In this case, a sequential execution, such as proposed by Function Chain would be much safer.[8]

Trade-off Analysis: Fan-in/Fan-out is a good pattern to deal with tasks that can be broke in parallel subtasks, such as multiple data searched. Function Chain works for subtasks that can't be executed in parallel, when the result of a function depends on the previous execution.

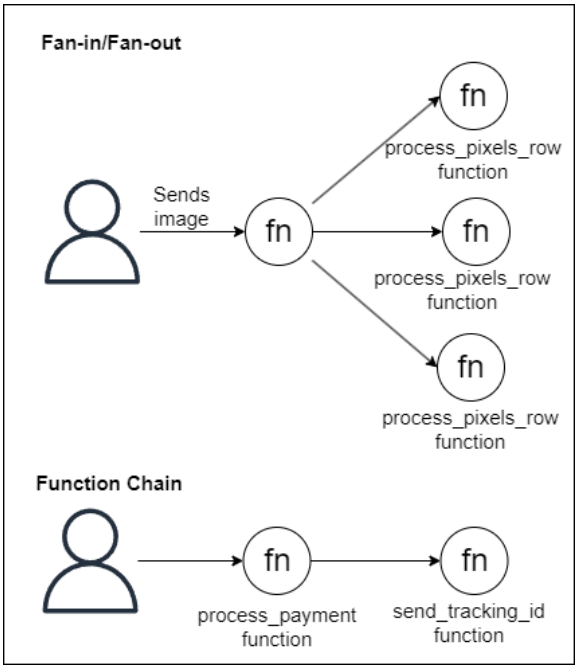

Fig. 8. Fan-in/Fan-out and Function Chain patterns

VIII. **Fan-in/Fan-out and Function Chain as specification of State Machine**: The three patterns acts on solving the problem of orchestrating long tasks. The State Machine pattern can be implemented using at least one of them, with the complexity of dealing with the state of the application, as done by AWS Step Functions.
Use case: Imagine you're building a food delivery app. The steps to order some food can be break in some sequential (Function Chain) and parallel (Fan-in/Fan-out) steps, using a state-machine to control the flow in the graph. For example, you could break the sequential steps in: 1) receiving the order and 2) executing the payment. If the payment is approved, you can break the execution in parallel steps like: 1) notifying the restaurant about the order and 2) requesting for a deliveryman to get the order in the restaurant. If the payment is denied, you can notify the client about the error. State-machine managers (such as AWS Step Functions) helps in the management of workflows, allowing to define how the data flows and how exceptions are treated during an execution.[9]

IX. **Gatekeeper as alternative to Valet Key**: Both patterns are similar and solves the same problem: dealing with authorization to securely call a restricted service. Gatekeeper does this by abstracting the credentials gathering from the client, leaving it to be a responsibility of the API Gateway. Valet Key lets to the client the responsibility of getting the authorization headers from an authorizer function and attach them to the requests.
Use case: Suppose you own a database of user data and need to expose it to other teams of your company securely. You can do this using a serverless approach using the Gatekeepr pattern, that simply exposes the database through an API Gateway and this gateway would be responsible for getting the authorization token by calling an authorizer function. Another way to do that would be to directly expose this function and let your clients be responsbile for calling it and getting the authorization token, removing the extra layer and getting some latency and cost decrease by adding some more responsibility to the client.[10]

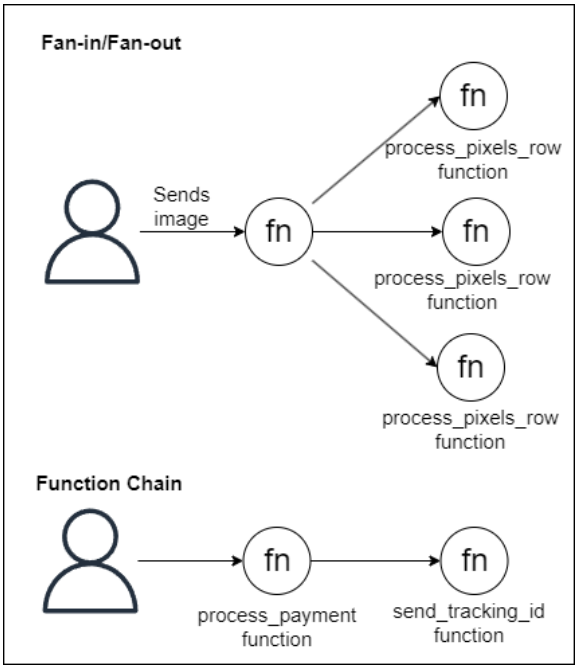

Fig. 9. Fan-in/Fan-out, Function Chain and State Machine patterns

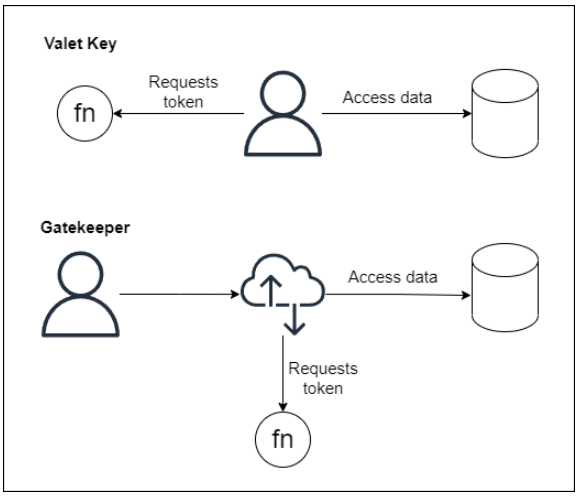

Fig. 10. Gatekeeper and Valet Key patterns

Trade-off Analysis: Calling an additional function adds one extra layer of complexity and cost, but gives more control and flexibility to the system.

X. **Frugal Consumer as alternative to Queue-based Load Leveling**: Both patterns are similar and solves the same problem: dealing with non-scalable back-ends. They differ in the way the client interacts with the throttling

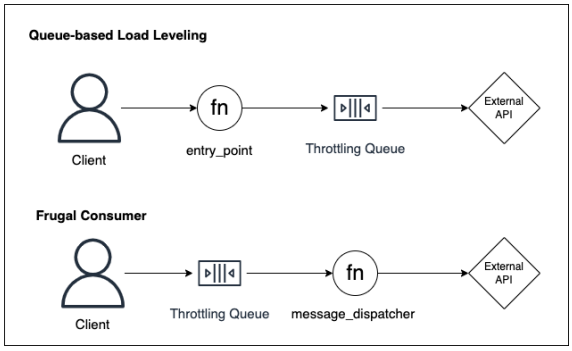

Fig. 11. Frugal Consumer and Queue-based Load Leveling patterns

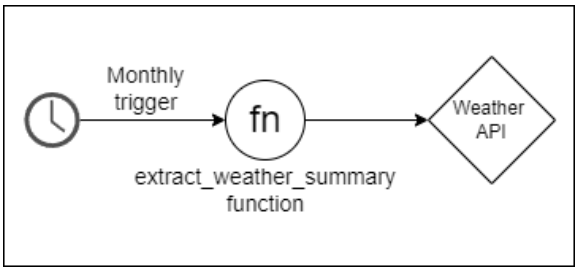

Fig. 12. Polling Event Processor and Periodic Invoker patterns

mechanism: the first one uses a function as entry point, while the second one uses the queue.

Use case: Say you're building a front-end app that needs to call a non-scalable external API that throttles high TPS requests. You could work around this scalability problem by using either Frugal Consumer or Queue-based Load Leveling patterns. Both would work fine, with the difference that the last one is based on a message delivery request, while the first one is based on a direct function call. Both patterns would work in the same way, limiting the rate of messages delivered to the limiting service.[11]

Trade-off Analysis: Calling a function adds one extra layer of complexity and cost, but makes it more general, making possible the extension to HTTP of event-based clients interactions. By the other side, removing this layer makes it specific to interactions using messages.

XI. **Polling Event Processor as specification of Periodic Invoker**: The core logic of Polling Event Processor makes use of Periodic Invoker in order to check for updates in external systems.

Use case: Imagine you're building a service that needs to summarize the data about the weather of the last month. To do so, you extract this data from a public weather API. This task is executed monthly, and it provides no event-based API, so a solution such as Polling Event Processor is required. You can define a schedule (such as offered by AWS Event Bridge) to start a function execution once in a month, fetch and extract this data. [12]

5 RELATED WORKS

Taibi et al. (2020) [2], through a multivocal literature review, selected and listed the state of the art of serverless patterns from white and gray literature. The paper lists several patterns, describing the problem solved and the proposed solution

of each one. Also, the patterns are separated into 5 categories, namely orchestration and aggregation, event management, availability, communication, and authorization. We aim to analyze how these patterns can work together to compose robust systems, and also make trade-off analysis to improve the decision-making process for engineers.

Sbarski (2022) [5] lists serverless patterns and their use cases in the industry. Also, this work shows the application of them in the context of the most used cloud-platform nowadays, Amazon Web Services (AWS), giving a highly practical approach. In this work, we plan to analyze these patterns in a more general approach.

Jeremy Daly's blog [1] describes, with intuitive diagrams, the state-of-the-art of serverless patterns being used daily by developers.

Also, Richardson (2018) [4] lists lots of microservices patterns that can be translated to the serverless paradigm. The blog posts are a fast-access source of information about pattern, being the main source for practitioners.

6 CONCLUSION

In this paper, we presented a huge list of patterns from different types of literature and built a pattern language to relate them. To describe them, we based our analysis in the **context** they apply, the **problem** they solve and **solution** they propose. After that, we could build the relationships between them based on which way they relate. We also described a trade-off analysis between patterns that are solutions to the same problem. After that, we could show a diagram that makes it easier and intuitive to use as a reference to practitioners that aim to build components using a serverless architecture.

Although we could build a pattern language, we believe that it can be improved with a deep dive into some points, such as: understanding the intersections between categories and understanding how existing microservice patterns can be translated to the serverless paradigm.

Finally, we believe that this work is useful for researchers, software engineers and software architects as a tool in the decision-making process of developing a serverless system. For future works, we intend to further specify the relationships between serverless patterns, as well as to assess the level of maturity practitioners have on the subject, and the impacts of such pattern language have on the practice.

REFERENCES

[1] Jeremy Daly. 2018. *Serverless Microservice Patterns for AWS.* https://www.jeremydaly.com/serverless-microservice-patterns-for-aws/
[2] Claus Pahl Davide Taibi, Nabil El Ioini and Jan Raphael Schmid Niederkofler. 2020. Patterns for Serverless Functions (Function-as-a-Service): A Multivocal Literature Review. (2020). https://www.scitepress.org/Papers/2020/95785/95785.pdf
[3] Sam Newman. 2015. *Building Microservices* (1st ed.). O'Reilly Media.
[4] Chris Richardson. 2018. *A pattern language for microservices.* https://microservices.io/patterns/index.html
[5] Peter Sbarski, Yan Cui, and Ajay Nair. 2022. *Serverless Architectures on AWS* (2nd ed.). Manning.
[6] Andreas Wittig and Michael Wittig. 2015. *Amazon Web Services in Action* (1st ed.). Manning.

A PATTERNS

Below we list the patterns we considered to build the relationships.

Table 1. Frequency of Special Characters

Pattern name	Brief description	Source
Read-heavy Report Engine	Usage of data caches and specialized views for read-intensive applications	[2][1][1]
Bulkhead	Divides functions in separated and isolated pools to achieve availability	[2]
Circuit-Breaker	Implements a "circuit" that isolates recovering services from new requests	[2][1]
Function Warmer	Keeps calling a function periodically to avoid cold startups and high latency	[2][1]
Eventually Consisted	Replicates data over services to keep it consistent	[2][1]
Aggregator	A function calls multiple services and returns the aggregation of its responses	[2][1]
Fan-in/Fan-out	Breaks workloads in multiple parallel functions to decrease execution time	[2][1]
Function Chain	Breaks workloads in multiple sequential functions	[2]
State Machine	Breaks workloads in a state-machine to control the flow of data through executions	[2][1]
Data Lake	Use functions to transform and persist data	[2]
API Gateway	An API Gateway routes requests based on endpoint	[2][1]
Internal API	Clients call functions directly without using an API Gateway	[2][1]
Router	A function routes requests based on payload to multiple services	[2][5][1]
Anti-corruption Layer	Creates a function that acts like a proxy to isolate legacy systems	[2][5]
Frugal Consumer	Receives and controls requests throughput using a function and queue	[2][1]
Queue-based Load Leveling	Receives and controls requests throughput using a queue	[2][1]
Thick Client	Lets clients directly access services	[2][5]
Priority Queue	Create multiple queues with different priorities to properly scale messages	[5]
Externalized State	Share state between functions using an external database	[2]
Data Streaming	Usage of streasm services	[2][5][1]
Publish/Subscribe	Usage of topics to communicate between services	[2][1]
Distributed Trigger	Usage of topics to communicate between services	[2][5][1]
Responsibility Segregation	Breaks the code logic in separated functions for read and write	[2]
Internal Handoff	Call functions with asynchronous events instead of HTTP calls	[2][1]
Periodic Invoker	Periodically triggers a function	[2]
Polling Event Processor	Use functions to poll for date periodically to check for updates	[2]
Gatekeeper	Usage of an authorizer function by API Gateway to authorize requests	[2][1]
Valet Key	Clients call an authorizer function to get a token to call downstream services	[2]