University of São Paulo
Institute of Mathematics and Statistics
Bachelor of Computer Science

# Building a Pattern Language for Serverless Architectures

Leandro Rodrigues da Silva

Final Essay

mac 499 — Capstone Project

Supervisor: Prof. Dr. Alfredo Goldman
Co-supervisor: M.e. João Francisco Daniel

São Paulo
2022

# Resumo

Leandro Rodrigues da Silva. **Building a Pattern Language for Serverless Architectures**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2022.

Computação *serverless* é um poderoso paradigma que vem se tornando popular na comunidade de engenharia de software. Porém, ainda há uma falta de conhecimento em como desenhar e implementar arquiteturas roubstas usando serviços *serverless*. Para ajudar praticantes a tomar decisões mais concisas nesse tópico, nós pretendemos criar uma ferramenta útil para utilizar arquiteturas desse tipo. Nesse trabalho, nós apresentamos uma linguagem de padrões como forma de atingir este objetivo. A linguagem de padrões compreende 28 padrñoes, classificados em 5 categorias: disponibilidade, orquestração e agregação, comunicação, gerenciamento de eventos e autorização.

**Palavras-chave:** Padrões de arquitetura. Computação sem servidor. Linguagem de padrões.

# Abstract

Leandro Rodrigues da Silva. **Building a Pattern Language for Serverless Architectures**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2022.

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 this topic, we intend to create a useful tool for reasoning about this king of 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 and aggregation, communication, event management and authorization.

**Keywords:** Architectural patterns. Serverless. Pattern language.

# Contents

# Chapter 1

# Introduction

## 1.1 Context

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) [7].

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 [7]. This model emerged as an alternative to solve the problems of monolithic architectures, aiming to make it easier to maintain or rewrite software [5].

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 [7]. 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.

## 1.2 Methodology

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 [6]. The relationships proposed by Chris Richardson are: motivating/-solution patterns, patterns that are solutions to the same problem, and generic/specific patterns.

To do so, we filtered patterns we found, both in gray and white literature, chose the relationships to be used and analyzed which and how the patterns relate. After performing these steps, we had as result our pattern language and also created a website to expose it to practitioners.

## 1.3 Structure of this document

This work is structured as follows: Chapter 2 explains the context of the problem as well as the required concepts to understand the problem and the solution; Chapter 3 explains the methodology we used to achieve our main goals; Chapter 4 presents our language, as well as applicable scenarios, use cases, trade-off analysis and the website we created to help in the diffusion of this knowledge; Chapter 5 explains about SugarLoaf PLoP[4] conference in which we presented our work; Chapter 6 concludes the work, talks about results, related works and future improvements.

# Chapter 2

# Background

Christopher Alexander [1] defined the term pattern as a problem that occurs over and over again in a given context. Each pattern can be described with three main elements: context, problem and solution. In software engineering, patterns are widely used to describe common situations that can occur during software development or designing and propose a scaffold to achieve the final goal.

Alexander also defined the term "pattern language" in the book The Timeless Way of Building [1] as a set of patterns that can have constraints and relationships between them. For example, patterns can relate to each other as "different solutions to the same problem" or "patterns that can be combined to achieve the solution". A pattern language can be used by developers in the process of decision making during the planning of an application, giving insights about which are the best patterns that can be used to achieve a goal and the trade-offs between them. A common way to describe a pattern language is to define a diagram that relates a set of patterns for a given context.

Another important concept that we'll be using in this work is "microservices". As defined by Sam Newman [5], microservices are small, autonomous services that work together. Microservices emerged as a solution to monolithic services, that have centralized responsibilities, strong coupling and tough maintainability. A single microservice has its own codebase and might be deployed as an isolated service. Multiple microservices can communicate with each other to perform a bigger responsibility, leading to a Service Oriented Architecture (SOA).

When talking about microservices, we have some strategies on how to make these services available over the Internet. 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 [7]. 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 [8].
- **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 [8].
- **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 [8].

Inside the cloud computing world, we have serverless. 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.

# Chapter 3

# Methodology

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

To achieve this work's goal, 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 both gray and white literature to create a list of patterns documented for serverless architecture. We searched for articles about this topic in Google Scholar [1] and in common gray literature sources (e.g. Medium, Dev.to, GitHub, etc.). With that, we could select a list of sources that made more sense to the work we wanted to develop. The complete list of patterns with their descriptions can be found in Section 4.1.

The main source of knowledge was Taibi et al. (2020) [3], which presented a collection of 32 serverless patterns along with a classification for them:

i. **Availability**: patterns that act on reducing possible failures or high latency issues
ii. **Orchestration and Aggregation**: patterns that structures how functions can be composed to perform complex tasks
iii. **Event management**: patterns that show different types of triggering by events
iv. **Communication**: patterns that show different types of communication between functions or services
v. **Authorization**: patterns that act on properly authorizing user or services, providing more access security

To achieve this list, Taibi et.al [3] also performed this search in gray and white literature, coming with a list of patterns from multiple sources.

[1] https://scholar.google.com.br/

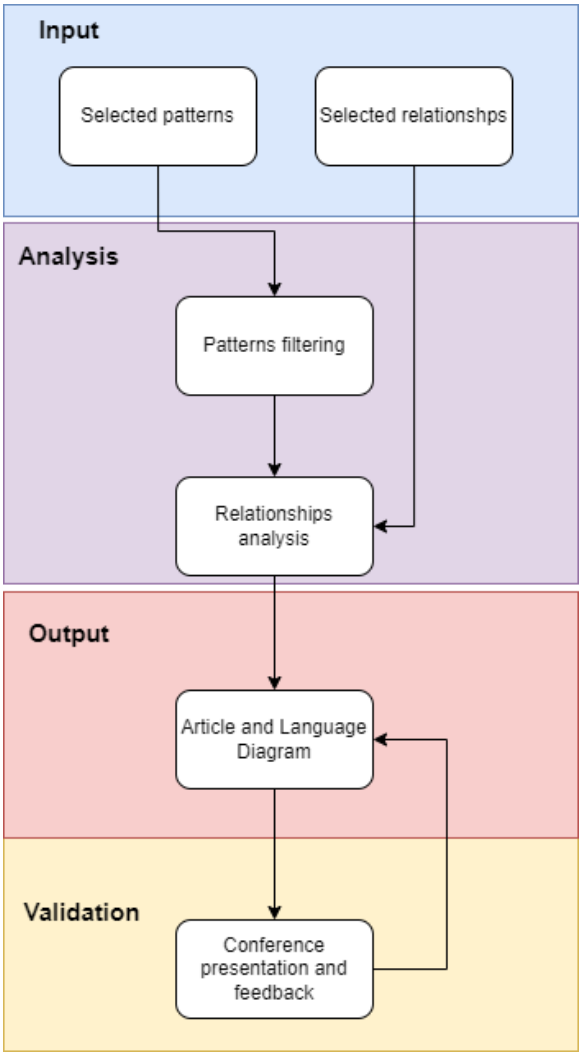


**Figure 3.1:** *Methodology used to produce the language*

Sbarski (2022) [7] 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 [2] describes, with intuitive diagrams, the state-of-the-art of serverless patterns being used daily by developers.

Also, Richardson (2018) [6] 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.

There were other interesting gray literature resources, such as Jeremy (2018) [2], InfoQ[1] and Medium[2] posts. Nonetheless, some of the gray literature sources presents patterns and practices considerably more coupled with cloud providers.

[1] https://www.infoq.com/articles/design-patterns-for-serverless-systems/

[2] https://medium.com/

Once we gathered a comprehensive list of serverless patterns, with a few classifications, we searched for relationships between different patterns. Richardson [6] is a reputed software architecture that wrote lots of works in the area of Microservices, such as Microservices.io[3], that centralizes a huge list of contents about microservices, including the pattern language created by Richardson. We used this source as reference for our work since the microservice world is too attached to the serverless world. Richardson (2018) [6] played a major role as inspiration for it, with the relationships here used in his language: "Motivating pattern/Solution pattern", "Solution A / Solution B", and "General / Specific". We explore further details about our adoption of these relationships in Chapter 4.2.

With that sources selected, we could start the steps to end up with our language. First of all, we reviewed and filtered the patterns 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.

Then we analyzed the possible relationships each pattern could have with the others, based on the relationships described by Richardson and described real use case scenarios where the relationships could be used. Finally, we got our resulting language that we described in both a diagram and a website.

Having these results, we could present our work in the SugarLoaf PLoP, the premier event in the area of patterns. The conference allowed us to receive feedback to improve the language and the paper and end up with a more robust work. We restructured our paper to make it more practical and become clearer who is the target reader of our work, adding hypothetical examples of applications to each of the relationships we have in our language. Also, we could give more focus in our language instead of the pattern listing, since that's the main goal of our work.

[3] https://microservice.io/

# Chapter 4

# Serverless Pattern Language

## 4.1 Patterns

In this section, we provide a list of patterns to build our pattern language. Patterns are reusable solutions to problems in software design. According to Christopher Alexander [1], a pattern is a three-part construct: context (Ct), problem (Pr) and solution (So). They're used to addressing common problems found in software development, serving as a powerful communication tool for developers working together in a solution [7].

**Read-heavy Report Engine** [3][2][2]

Ct - Read-intensive applications

Pr - Improve performance in read operations

So - Usage of data caches and specialized views for most frequently queried data.

**Bulkhead** [3]

Ct - Functions with high TPS.

Pr - If a function fails, the whole system can be compromised.

So - Partition functions in multiple pools, separating each pool for a set of clients. If a single pool becomes down, just a subset of the clients will be compromised.

**Circuit-Breaker** [3][2]

Ct - Downstream services being a point of failure.

Pr - If a downstream service is down, sending requests to it will cost resources and cause more impact.

So - Create a "circuit" where whenever a high amount of failures is detected, the circuit opens and the requests are not sent to downstream services. When they are recovered, the circuit closes again and the requests are sent normally.

**Function Warmer** [3][2]

Ct - Latency-sensitive applications with low TPS.

Pr - Functions have “cold-starts”, meaning that a function that is not used frequently will have a startup time when a request arrives.

So - Have a cron job that calls the function periodically to keep it always warm.

**Eventually Consisted** [3][2]

Ct - Applications that need data replicated over multiple regions/services.

Pr - Keep incoming data consistent over multiple services.

So - Use stream services to trigger events made on the database to publish it to the multiple bases.

**Aggregator** [3][2]

Ct - Services that need to perform many operations or read data from multiple sources.

Pr - Overcome complexity of calling multiple services and aggregating the responses.

So - Create a single function that calls the downstream services, aggregates the data and exposes it through an API.

**Fan-in/Fan-out** [3][2]

Ct - Execution of long tasks.

Pr - Functions usually have a timeout if it keeps executing for a long time (> 15 minutes). Long tasks would not finish and the connection with the caller will be closed without a response.

So - Split the work into multiple parallel tasks, each part being executed by a function. At the end, the final result is aggregated.

**Function Chain** [3]

Ct - Execution of long tasks.

Pr - Functions usually have a timeout if it keeps executing for a long time ( 15 minutes). Long tasks would not finish and the connection with the caller will be closed without a response.

So - Break the job into multiple sequential functions.

**State Machine** [3][2]

Ct - Orchestration of multiple functions with state.

Pr - Orchestrating a workflow of functions with different states is complex and can lead to unexpected results.

So - Use services such as AWS Step Functions to coordinate the execution of state machines.

**Data Lake** [3]

Ct - Persistence of large volumes of data.

Pr - Storage of a large database with multiple transformations can be complex.

So - Trigger functions for updating events in the database, process the raw data and persist it with the transformed data.

**API Gateway** [3][2]

Ct - Exposing access to back-end services.

Pr - Exposing multiple endpoints to clients can be complex.

So - Use an API Gateway to grant clients access to specific services.

**Internal API** [3][2]

Ct - Dealing with access to back-end services.

Pr - Using an API Gateway would expose the services publicly, leading to possible security issues.

So - Call internal functions directly using events.

**Router** [3][7][2]

Ct - Dealing with access to back-end services.

Pr - Clients do not always know about downstream services and its endpoints.

So - Use a "router function" that acts like a router, invoking services based on the payload content of the request.

**Anti-corruption Layer** [3][7]

Ct - Dealing with legacy systems.

Pr - Sometimes, a service needs to call a legacy system that has old protocols or inconsistent behavior.

So - Create a function that acts like a proxy, abstracting the calls to the legacy service.

**Frugal Consumer** [3][2]

Ct - Dealing with non-scalable services.

Pr - Sometimes a non-scalable back-end service can be the bottleneck of the system.

So - Use a queue that acts as a throttling queue that receives the requests from clients, and a function that processes the messages asynchronously and sends to the non-scalable service. With this function, it's possible to limit the rate of requests that goes to the back-end.

**Queue-based Load Leveling** [3][2]

Ct - Dealing with non-scalable services.

Pr - Use an entry-point function that receives the requests and posts them to a queue that acts as a throttling queue and sends to the non-scalable service. With this queue, it's possible to limit the rate of message deliveries that goes to the back-end.

So - Create multiple queues with different priorities that triggers different functions, with resources allocated according to its prioritization.

**Thick Client** [3][7]

Ct - Decrease of costs and latency.

Pr - The Gateway pattern adds an intermediary layer to abstract downstream services, increasing costs and latency.

So - Allowing clients to call services directly can reduce both.

**Priority Queue** [7]

Ct - Scale message processing with different priorities.

Pr - Although serverless services handle scalability under the hood, sometimes it is necessary to control how messages are dealt by the system.

So - Create multiple queues with different priorities that triggers different functions with resources allocated according to its prioritization

**Externalized State** [3]

Ct - Dealing with state in a serverless architecture.

Pr - Functions are stateless, but sometimes state between them is necessary.

So - To overcome this, an external database can be used to keep state across functions.

**Data Streaming** [3][7][2]

Ct - Applications that need to deal with continuous incoming data.

Pr - Managing continuous incoming data can be complex, especially if it will b+ replicated through services.

So - Use data streams, such as those offered by AWS DynamoDB, that can trigger events to functions whenever an update is made in a table.

**Publish/Subscribe** [3] [2]

Ct - Dealing with notifications to internal services.

Pr - Forward data for internal services.

So - Use a publish/subscribe service, such as AWS SNS to propagate notifications from clients to backends.

**Responsibility Segregation** [3]

Ct - Scaling different operations accordingly to the application requirements.

Pr - Generally, reads and write operations occur with different frequencies. If a function performs both read and write operations, it could be difficult to scale it properly.

So - Segregate the responsibility of reads and writes in different functions.

**Distributed Trigger** [3][7][2]

Ct - Broadcasting messages to multiple services.

Pr - Notify multiple services with a new message.

So - Create a single notification topic inside the notifier service and subscribe the downstream services to it.

**Internal Handoff** [3][2]

Ct - Calling functions asynchronously.

Pr - Syncrhonous calls to functions will hold the connection open until a response is received from the function.

So - Use "InvocationType" event, that sends an asynchronous message to functions and closes the connections. The message then can be processed and if it fails, can be sent to a dead-letter queue.

**Periodic Invoker** [3]

Ct - Dealing with periodic tasks.

Pr - Execute tasks periodically.

So - Configure a periodic event, such as offered by AWS EventBridge, that periodically calls a function.

**Polling Event Processor** [3]

Ct - Dealing with periodic tasks.

Pr - Execute tasks periodically.

So - Configure a periodic event, such as offered by AWS EventBridge, that periodically calls a function.

**Gatekeeper** [3][2]

Ct - Dealing with authorization to call services.

Pr - Grant proper access to services.

So - Use an API Gateway with an authorizer function that receives the authorization header and returns the correct authorization policy.

**Valet Key** [3]

Ct - Dealing with authorization to call services.

Pr - Grant proper access to services.

So - Here the client calls a function that returns the authorization headers to be included in the requests to the restricted service.

## 4.2 Language

In this chapter, we present our pattern language for serverless architectures. To do so, we're going to analyze the relationships between patterns illustrated in the diagram [4.1]. The patterns are divided using the categories proposed by Taibi et. al.[3].

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.

We'll relate the patterns in our language using the relationship types proposed by Chris Richardson [6]:

- **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.

Below we present the relationships in our language with: (i) a brief description on how they relate, (ii) a use case exemplifying this relationship and (iii) a trade-off analysis in the case of patterns that are alternative solutions to the same problem:

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 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. [4.2]
**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-

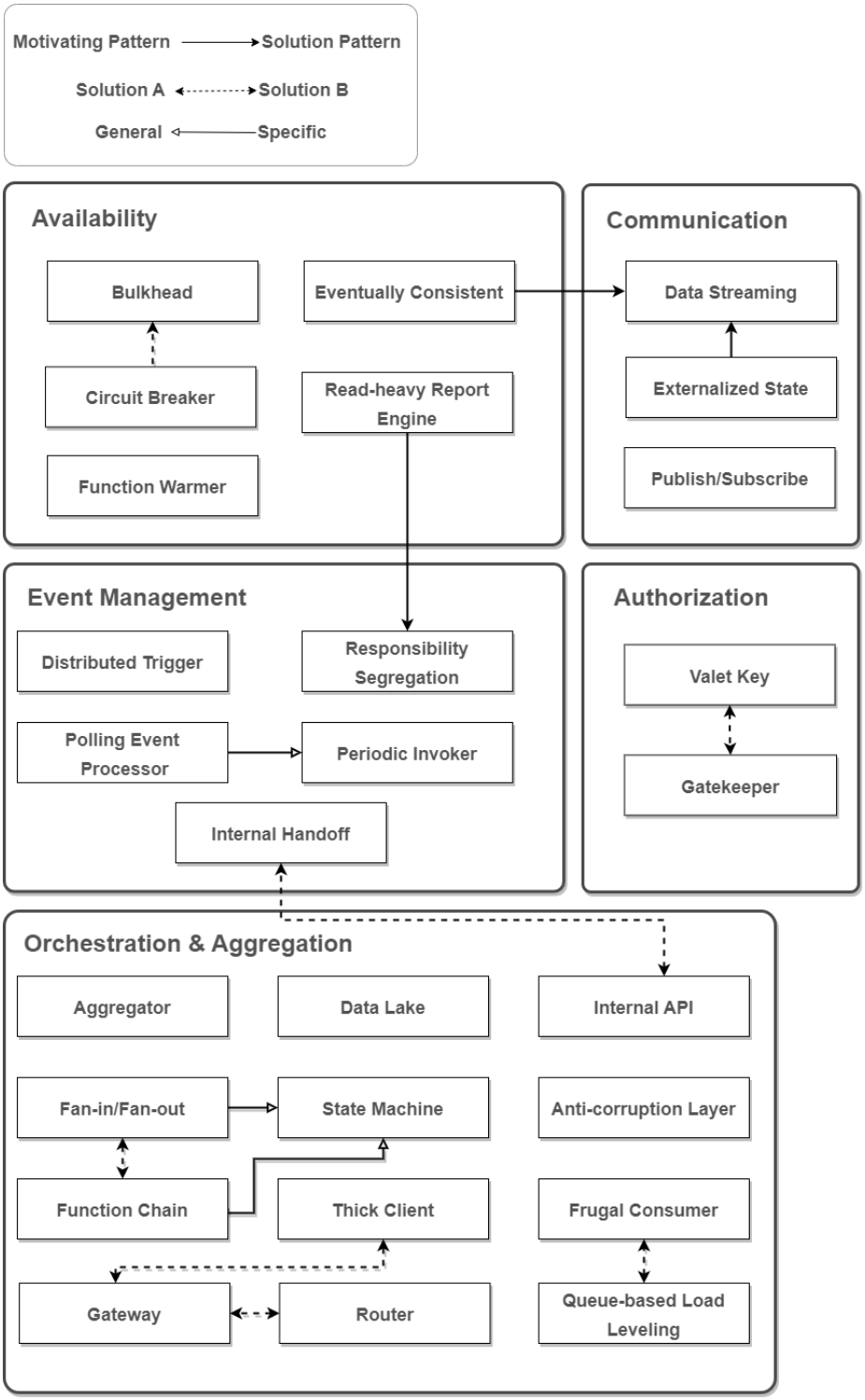


**Figure 4.1:** *Diagram containing a pattern language for serverless architectures*

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.

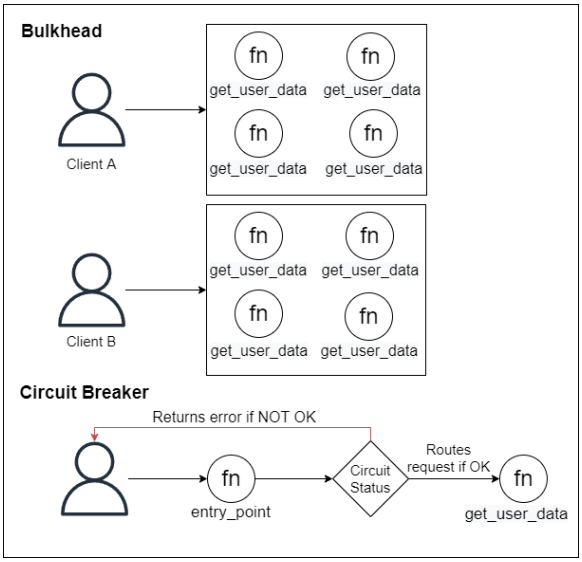


**Figure 4.2:** *Bulkhead and Circuit-Breaker patterns.*

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 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. [4.3]

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 from 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 all over these 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.[4.4]

IV **Externalized State as motivation to Data Streaming**: Using Data Streaming is

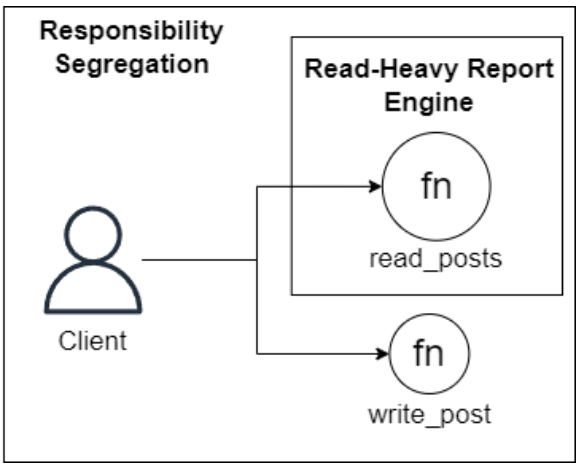


**Figure 4.3:** *Read-heavy Report engine and Responsibility Segregation patterns.*

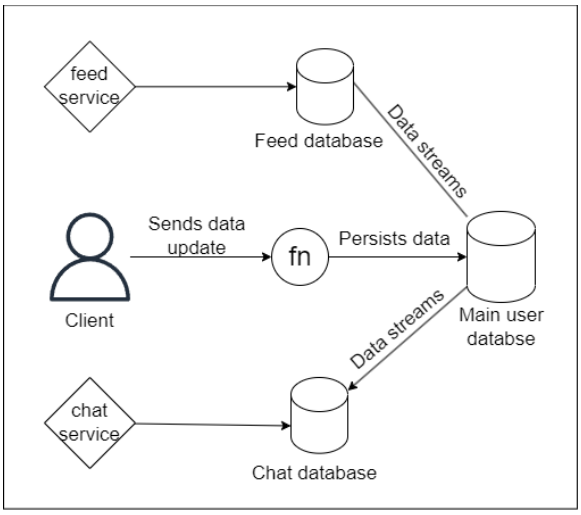


**Figure 4.4:** *Eventually Consistent and Data Streaming patterns.*

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 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.[4.5]

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

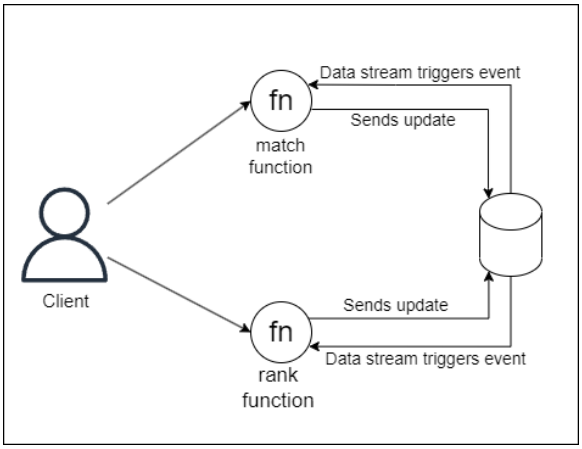


**Figure 4.5:** *Externalized state and Data Streaming patterns.*

need to know a single endpoint and the gateway will be responsible for routing the requests. In the second one, the clients will need to know how to properly invoke the function.[4.6]

**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.

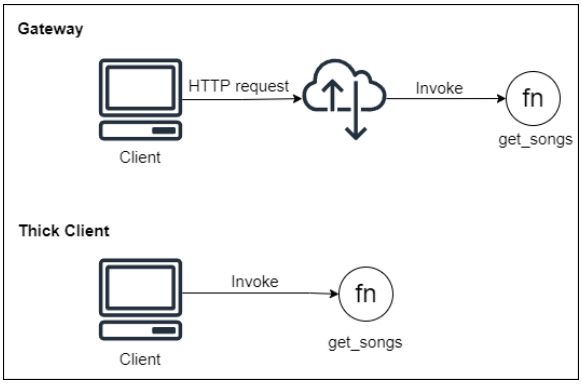


**Figure 4.6:** *Thick Client and Gateway patterns.*

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 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.[4.7]

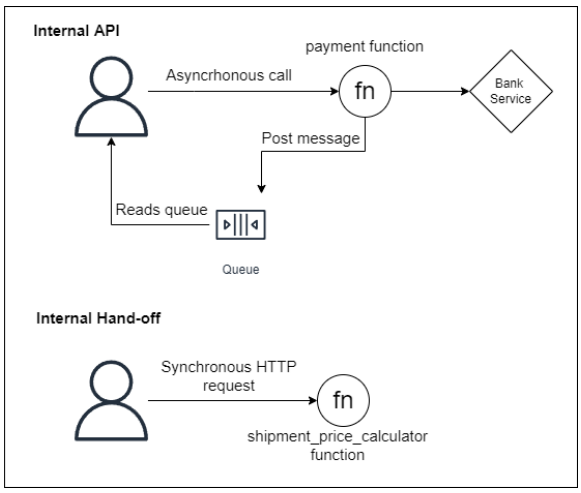


**Figure 4.7:** *Internal API and Internal Handoff patterns.*

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.[4.8]
**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.

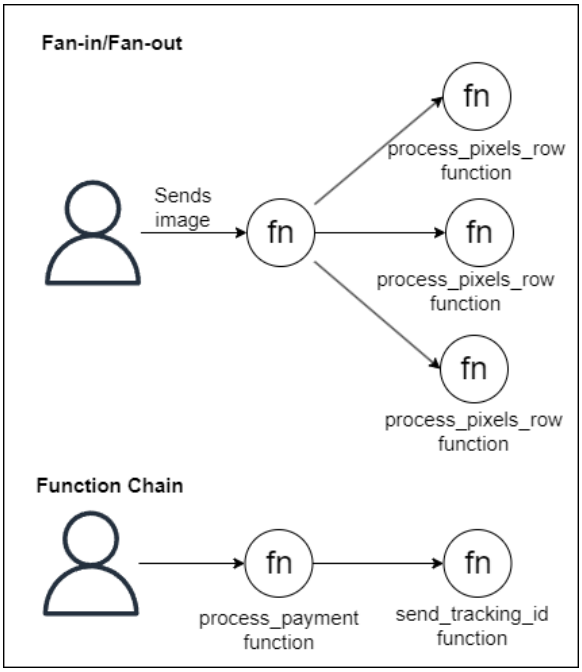


**Figure 4.8:** *Fan-in/Fan-out and Function Chain patterns.*

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.[4.9]

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 Gatekeeper 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

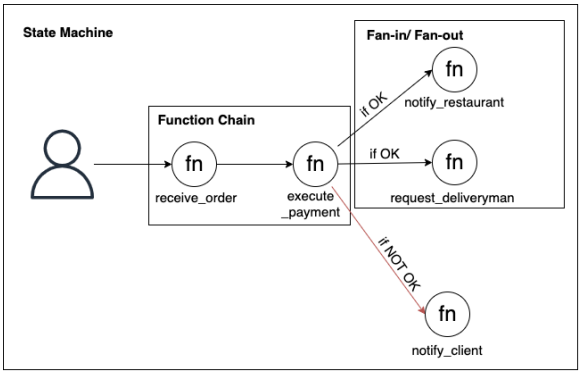


**Figure 4.9:** *Fan-in/Fan-out, Function Chain and State Machine patterns.*

function and let your clients be responsible 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.[4.10]
**Trade-off Analysis**: Calling an additional function adds one extra layer of complexity and cost, but gives more control and flexibility to the system.

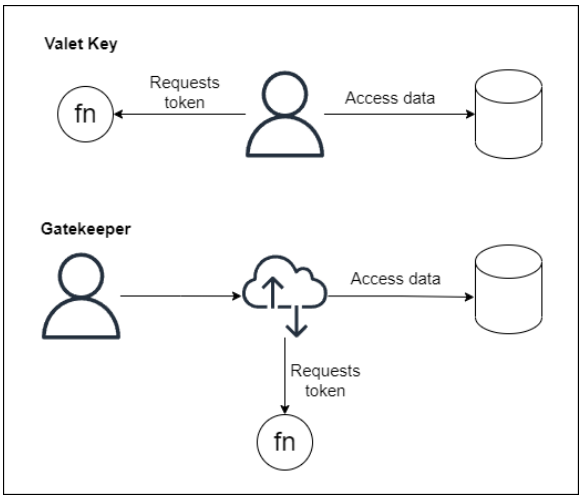


**Figure 4.10:** *Gatekeeper and Valet Key patterns.*

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 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 a high amount of transactions per second. 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.[4.11]
**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.

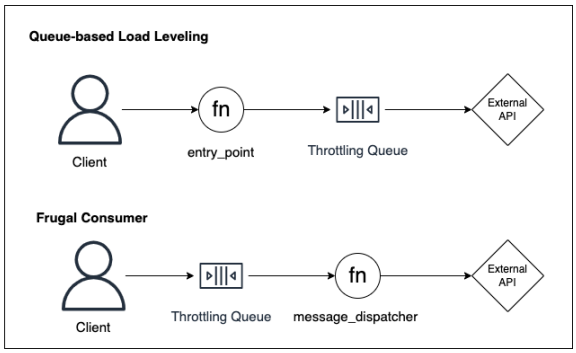


**Figure 4.11:** *Frugal Consumer and Queue-based Load Leveling patterns.*

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. [4.12]

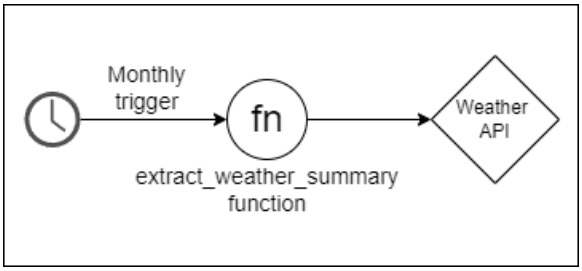


**Figure 4.12:** *Polling Event Processor and Periodic Invoker patterns.*

## 4.3 Website

During the presentation of our work in the SugarLoaf PLoP conference, we received the feedback to expose the work we made to a website. We planned to create a website to display the results of our work. The idea was to put the knowledge to be available in the gray literature, so that the engineers and architects could have a centralized source of information about the language, the patterns and how to apply them in their daily job.

We based our idea in well known websites such as Refactoring Guru[1] and Microser-

[1] https://refactoring.guru/

vices.io[2], that provide this kind of content for their respective topics. In our website, the user will have information about the patterns we selected, the diagram we built and the relationships we described.

### 4.3.1 Structure and repository

When entering in our website, the user will face the landing page with a brief description to the solution we built and a link to get more details. the user will face: (i) a link to the page containing a brief description of each pattern 4.13 and the link to the pages with the descriptions of each of the relationship as shown in 4.14; and (ii) a link to the page containing the final version of our article, containing all the details about the construction of our language.

The website was made using VuePress[1], a JavaScript framework to create documentations using Markdown. The site is available at https://serverlesspatternlanguage.surge.sh/ and the repository containing the source code can be found at GitHub[3].

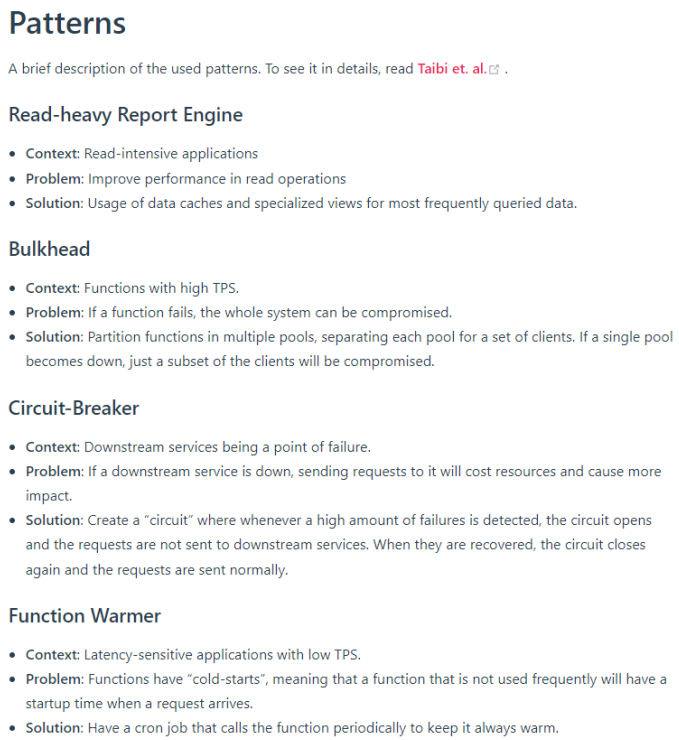


**Figure 4.13:** *Part of the patterns in the website.*

[2] https://microservices.io/

[1] https://vuepress.vuejs.org/

[3] https://github.com/Leandrigues/serverless-pattern-language

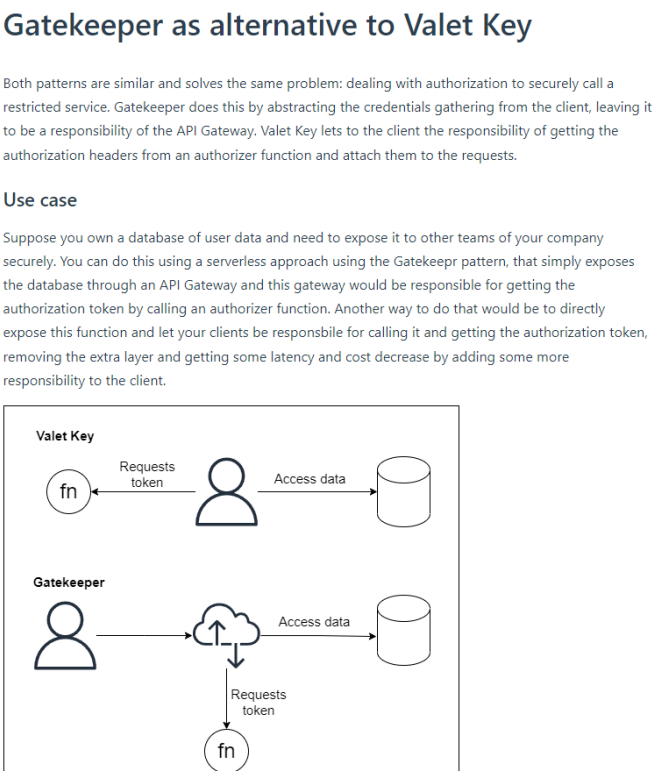


**Figure 4.14:** *Example of a relationship in the website.*

# Chapter 5

# SugarLoaf PLoP Conferece

As part of the validation of this work, we presented the paper in the SugarLoaf Pattern Languages of Programs 2022. SugarLoaf PLoP is the premier event for pattern authors and enthusiasts to discuss patterns and aims to improve authors' papers quality by gathering feedback from other participants authors.

The conference consisted of three parts:

- **Papers selection** - Submission of the first version of the paper to be accepted or denied by the organizers,
- **Shepherding** - Assignment of a tutor to the papers that has huge knowledge in the area of patterns to help to improve the paper to the final version.
- **Presentation** - A week of presentation of the papers to gather feedback from other authors in a "fly on the wall" discussion pattern in order to improve the paper quality.

## 5.1 Shepherding Phase

During the shepherding process, our paper was assigned to Fernando Lyardet[1], an experienced author of papers related to patterns in software engineering. For almost one month, Fernando helped us in refining the content and structure of the paper.

The first version of the paper had some issues, like defining who was our target reader, since the paper mixed contents related to white literature and gray literature. Using the shepherding feedback, we restructured the paper to be consumed by software builders and designers, instead of academic researchers. To do so, we introduced the use cases with visual schemes, so that we could get a more practical vision of the relationships. The idea is that someone that is acting on decision making process of an application could read a brief description of the problem and also an exemplification of the solution.

Other issue pointed by our shepherd was that the main content of our article was not being properly highlighted. Although the most important value we're bringing to

[1] https://www.f6s.com/fernandolyardet

developers is the construction of a pattern language for serverless architecture, the first version of the paper was giving a lot of space to listing and describing the selected patterns, what is already done by Taibi et. al. [3]. With that in mind, we removed the pattern listing and moved it to an Appendix, leaving the highlight of the paper to be the language we constructed. Although, we added this list back to the final essay since it makes sense to give more context to the whole work.

By the end of the shepherding phase, we had a new version of the paper with a clear scope, focus and target reader. With that done, we could take it to the conference to get the feedback of the readers to keep improving the paper.

## 5.2 Conference Presentation

With the new version of the article, we could take it to present for other authors to read and discuss it in the SugarLoaf PLoP 2022. The conference has a presentation model called "fly on the wall", which participants are put in small groups of authors and then each day, one paper is selected and the other authors will read the selected article. The author of the selected paper will be like a "fly on the wall", listening the participants discuss his paper without intervening. This process is helpful to understand the strong points and gaps of the paper and get the proper feedback to improve the final version.

Our article was presented on October 18th and was read and discussed by other four authors and one conference organizer. Some authors claimed that the source of the patterns were not clear. To fix this, we added sections to emphasize the sources and how they were filtered. Also, some of them asked us to keep the patterns description, since it makes the article self-contained and does not require the reader to read other works to understand the language. Finally, another tip to improve the visibility of the work to developers was to create a website listing the patterns and their relationship. We implemented this solution and described it in Chapter 4.3.

With this feedback, we could improve our article both in content and structure, finishing the event with a more robust and refined paper. In the next year, some of the papers will be selected to be published by ACM[2].

[2] https://www.acm.org/

# Chapter 6

# Conclusion

## 6.1 Contributions

The lack of content in the area of patterns in serverless is huge. The sources of knowledge are spread over multiple work from white literature that are not used as a consulting source as part of the daily job of software engineers and architects.

In this paper, we presented a selection of papers extracted from other sources. 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. Also, we exported this knowledge to a website to make it easier to find and to consume, giving to our target reader a simple tool to consult whenever necessary.

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 serverless systems. We provided both a formal article that was improved and validated in SugarLoaf PLoP Conference, that is a well respected event in the area of patterns, and also a platform to expose the knowledge we built.

## 6.2 Related Works

Taibi et al. (2020) [3], 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 analyzed how these patterns can work together to compose robust systems, and also make trade-off analysis to improve the decision-making process for engineers. We also consulted and based our work in other authors such as Christopher Alexander [1] that made a great work in creating

a pattern language for microservices, and [7] that created a book that lists patterns and good practices for building complex servlerless systems.

## 6.3 Future improvements

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, creating new relationships or constraints and understanding how existing microservices patterns can be translated to the serverless paradigm. Also, we plan to keep improving the website to make it a centralized source of information for serverless patterns and how to apply them, becoming a central point for builders that want to understand better about this topic.

# Bibliography


[1] Christopher Alexander. *The Timeless Way of Building*. 1979 (cit. on pp. 3, 9, 27).

[2] Jeremy Daly. *Serverless Microservice Patterns for AWS*. 2018. URL: https://www.jeremydaly.com/serverless-microservice-patterns-for-aws/ (cit. on pp. 6, 9–13).

[3] Claus Pahl Davide Taibi Nabil El Ioini and Jan Raphael Schmid Niederkofler. "Patterns for Serverless Functions (Function-as-a-Service): A Multivocal Literature Review". In: (2020). URL: https://www.scitepress.org/Papers/2020/95785/95785.pdf (cit. on pp. 5, 9–14, 26, 27).

[4] Hillside Group. *SugarLoaf PLoP*. https://www.hillside.net/plop/2022/index.php?nav=sl-ploppaperscfp. Accessed: 2012-11-06. 2022 (cit. on p. 2).

[5] Sam Newman. *Building Microservices*. 1st. O'Reilly Media, 2015 (cit. on pp. 1, 3).

[6] Chris Richardson. *A pattern language for microservices*. 2018. URL: https://microservices.io/patterns/index.html (cit. on pp. 2, 6, 7, 14).

[7] Peter Sbarski, Yan Cui, and Ajay Nair. *Serverless Architectures on AWS*. 2nd. Manning, 2022 (cit. on pp. 1, 3, 6, 9, 11–13, 28).

[8] Andreas Wittig and Michael Wittig. *Amazon Web Services in Action*. 1st. Manning, 2015 (cit. on p. 4).