A self-adaptive batch request aggregation pattern for improving resource management, response time and costs in microservice and serverless environments

George Kousiouris
Department of Informatics and Telematics
Harokopio University of Athens
Athens, Greece
gkousiou@hua.gr

***Abstract*— Addressing increased demand in cloud environments is typically achieved through auto-scaling the use of further resources allocated. However this comes at the trade-off of increased back-end resource stress as well as runtime costs. In this paper a request batch aggregation and management pattern is proposed and implemented, especially for cases of computational load that needs a heavyweight environment setup (such as performance prediction and AI services). The pattern acts as a preprocessing layer and accumulates requests, withholding their forwarding to the backend based on the conditions of execution (frequency of requests), in order to regulate back-end resource stress without adding extra resources. Different controlling logics are tried out for batch request release, ranging from simpler ones (batch size static regulation, time out interval) to more elaborate ones (ANN model predicting the desired batch size) for adapting the pattern configuration during runtime. Results indicate severe reduction of back-end stress through significantly reducing needed containers (therefore costs as well), enhanced response time as well as avoidance of system breakdown under heavy load. The pattern can be applied in both microservices and serverless environments, especially in Edge cases where resources are constrained.**

Keywords—design patterns, resource management, performance, cost, function as a service, microservices

I. INTRODUCTION

A. Motivation

In many application domains, such as Artificial Intelligence, performance estimation, resource management etc., service implementations are required to serve incoming requests by leveraging heavyweight computational environments and libraries. These are used in order to apply created models to desired request inputs and respond with the according prediction.

When these functionalities are offered as a service, through typical threadpool server implementations, this implies that for each incoming request a relevant environment needs to be spawned in order to process it. This creates extreme stress on a server for mainly two reasons. Firstly, the initialization time of such an environment thread is typically non negligible. In [1], a performance analysis on such a server implementation using the Octave environment [33] indicated that above 70% of the total serving time (1.2 seconds) for a single client request was due to raising the respective Octave environment thread. Only 10 milliseconds of computation time were needed for the actual computation, indicating an extreme overhead of preparation in relation to the actual computation time.

Secondly, when multiple simultaneous requests are sent towards the server, an according number of threads need to be concurrently run and compete for resources such as memory and CPU, while interfering with peripheral elements such as cache memories. The aforementioned competition creates a further increase of.the response times, while concurrency overheads can reach levels of 400% of performance degradation [10].

Similar overheads apply for more modern computing models such as serverless and Function as a Service environments (FaaS), in which containers with designated functions are executed in order to respond to an incoming request endpoint. In this case, the process itself of raising the containerized environment is even worse than raising another thread on an existing server. Overhead inserted by this computing model is heralded as one of the main challenges of the domain [2] and several approaches have been applied at a resource management layer in order to measure it or mitigate it. Examples include intra and inter-container interference effects [9], cold start reduction [13], pre-warm container strategy in FaaS, sharing of container runtimes between functions [3], sharing libraries [7] etc. Optimizing such aspects in an autonomic manner is key in order to support the creation and runtime management of cloud-native applications [29].

B. Contribution of this work

The aim of this work is to propose the design of a request preprocessing layer that targets to improve server performance and responsiveness to varying conditions of traffic without having to increase used resources. The foreseen mechanism aggregates incoming requests into groups and forwards them to the service backend based on specific traffic criteria and controlling logic (Figure 1). In light traffic conditions, each request could result to spawning a new computational instance (thread or container), however in heavy traffic, the mechanism will create batches of inputs and request the serving of the input array from one computing instance.

A direct implication of this strategy is to significantly reduce the number of containers needed for serving the specific set of requests. If on a private cloud, this regulation will aid in reducing the stress of the backend infrastructure and orchestration system, especially for edge resource clusters with reduced computational capacity. If on public cloud setups, it will reduce costs since equivalently the number of function invocations (a main billing factor of FaaS environments) or load balanced service replicas will be reduced.

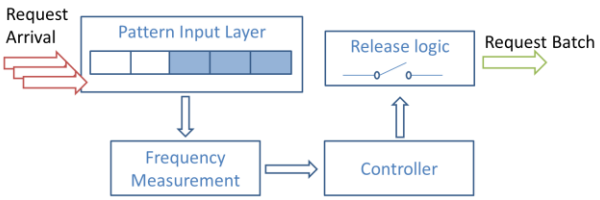

Figure 1: General Overview of the Mechanism

Further contributions of this work relate to the examination of deciding logic strategies through which the specific pattern is adapting to varying frequencies of request arrivals. These include simple rules such as a general time-out rule release as well as adaptation throughArtificial Neural Network (ANN) model investigation. The deciding models investigate the quantitative link between the batch size after which the accumulated requests need to be released, the observed frequency of incoming requests and the system response time.

A prototype implementation is created and experimented with, in order to demonstrate the benefits of the approach, indicating lower response times and container numbers in medium traffic conditions. In heavy traffic conditions the main benefit is the increased robustness of the mechanism to avoid server stalling (in cases of sudden peaks) or the significant improvement of response times through reduction of resource contention (in cases of high but gradually increasing load).

The paper proceeds as follows. In Section 2 related work with relation to scaling, cost and structural patterns in the area of microservices and FaaS is investigated in order to highlight differences and complementary points to this work. Section 3 presents the proposed pattern design and rationale, prototype implementation and requirements as well as limitations for its application. Furthermore it includes the variations of the controlling logic used for dynamically setting the operational point of the pattern. Section 4 contains the experimentation in order to observe the pattern's effect on response times, number of active containers and cost parameters while section 5 concludes the paper.

II. Related work

In terms of related work, different categories can be identified as related to the concepts presented in this work. Initially, a general performance strategies overview is presented and positioned with relation to this work. Following, different types of scaling patterns in typical FaaS environments are investigated, in order to detect differences with the proposed approach. Related structural or functional patterns are also identified and linked to the approach, followed by typical cost models in FaaS in order to highlight the key billing factors and focus on characteristics that would minimize them.

A. Performance Overview of cloud environments

The issue of performance management in cloud environments has been extensively analyzed in recent years. Approaches around auto-scaling, through direct Proportional Integral Derivative (PID) controllers [14], ANN based application prediction models [15], feedback loop-based Quality of Service (QoS) prediction through optimization techniques such as Particle Swarm Optimization [16] focus on increasing application related resources in order to enhance Key Performance Indicators (KPIs). Other works combine issues of benchmarking, modelling and real time analysis [34]. Increasing resources is of course a valid choice, however it is results to extended costs. Approaches such as the pattern described in this work target primarily at better utilizing existing resources in order to improve application performance, without having to increase them.

Other works examine interference caused by concurrency, either at the Virtual Machine (VM) [10] or the container level [9] (including intra or inter container overheads) as well as the introduction of microVMs [11]. With the minimization of container instances needed, following the application of the proposed pattern, the according interference effects are also anticipated to be reduced.

Alternative approaches try strategies such as dependency packages aware affinity scheduling [8] or library sharing [7]. These approaches target a lower level of management. Therefore they can be applied in parallel to the presented work, in order to further enhance the runtime execution of the individual containers created after the batch release.

B. Scaling patterns in FaaS

In [4], a thorough analysis is presented on the various scaling patterns available in serverless offerings such as AWS Lambda, Google and IBM Cloud Functions as well as Knative and Cloud Run. Based on the findings, two main modes of auto-scaling connecting the number of containers to function requests are identified, in addition to a third one based on node metrics. Per-request scaling (Figure 2a) raising a separate container per function request (most typical case) and concurrency value scaling (Figure 2b), enabling the concurrent execution of multiple functions in one container. The proposed pattern in this paper appears in Figure 2c. In this case we propose the aggregation or incoming requests before submitting them to the back-end. Release logic can be on batch size, timeout threshold or a combination of both. This would enable reduction of total containers running in the system, compared to case (a), while removing function concurrency overheads that can occur in case (b), especially for computationally intensive workloads.

One aspect to notice with relation to serverless environments is the term "*concurrency*". One variation of the term relates to the total number of the function activations in a namespace that execute across the cluster (in different containers) and is more related to throttling aspects, so that a specific namespace and its functions does not overwhelm the overall quota of function executions available to a user [22]. This needs to be distinguished in terminology compared to the case of "*function concurrency scaling factor*" that is presented in Figure 2b. An example documentation of the second case appears in [21].

C. Structural and Functional Patterns in Cloud environments

Patterns, or architectural styles, are considered a key element of cloud software application development in order to increase code reuse and ensure a number of issues such as fault tolerance or performance in distributed environments [17]. A very interesting survey on existing patterns appears in [3]. From the mentioned ones, the queue based load

levelling resembles the rational of the function input aggregator, in the sense that it does not allow the requests to reach the backend in order not to create unpredictable peaks in load. For this reason, it accumulates messages in a queue, from which the consuming function/service retrieves them at its own pace. The main difference here is that the queue based load levelling pattern does not aggregate function inputs into one function execution in order to optimize costs and improve response times, but it only acts as a queue from which the back-end can retrieve tasks at its own pace. Thus significant waiting time is added to the requests. The Aggregator pattern is also mentioned, however this refers to aggregation of functionalities from multiple services and offering them through a single endpoint, thus it has no actual resemblance, other than a partially similar name.

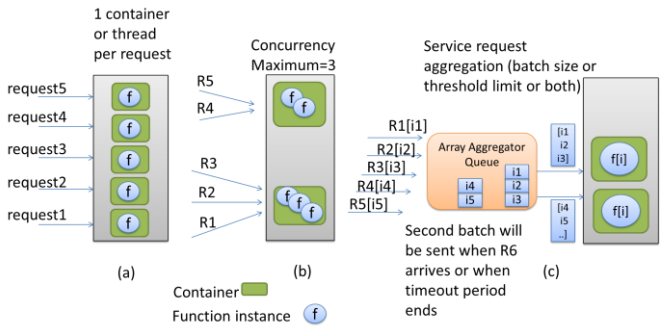

Figure 2: Scaling Patterns (a),(b) from [4] compared to the proposed pattern (c)

In some cases, the existence of a pattern formalism for a common action such as retrying a failed request can be seen as an overkill for such a simplistic concept. The same could be argued for the concept of batching requests together. However one needs to take under consideration the overall context of execution, what parameters can be set, effects of various settings (e.g. extensive retrying can further stress a problematic server), what strategy to follow etc. An example of how even such a simplistic pattern such as the Retry pattern can evolve to a more complex, self-regulating and generic software component is the Circuit Breaker [5]. This pattern is primarily intended for handling errors in calls in a dynamic manner and at the client side. The pattern foresees three states, the Closed state (in which all calls go through to the target), the Open state (no call goes through since previous errors have indicated that the target experiences issues) and the Half-open state (some of the calls are tried out in order to check the state of the target service). Thresholds (either for successful or unsuccessful calls in a period) regulate the transition between states as well as timers for remaining in a state. In this case, the use of the pattern (fault tolerance) is of course different, but the interesting feature is the self-regulation rationale, with which the pattern can decide which mode to apply, based on current conditions of executions. Further more sophisticated logic has also been proposed on top of the typical Circuit Breaker pattern [32].The use of the Circuit Breaker pattern is proposed alongside the pattern proposed in this paper as will be highlighted in Section IV.D.

Self-adaptation, e.g. based on models, is considered key for ensuring interactivity with the environment, i.e. a static configuration may not adapt in varying workload conditions, as anticipated in elastic cloud environments. Flexibility-based patterns, based on resource increase or replacement at the workflow level are studied in [18].The presented work complies with these principles, both in the usage of performance model driven variations of the pattern as well as the exploitation of runtime information such as request frequency in order to adapt the pattern rationale, as will be presented in Section III.E.

A very detailed list of numerous cloud design patterns, along with the benefits and shortcomings of each one, as well as means of mitigating weaknesses through pattern combinations, is included in [28]. No similar pattern to the one described in this paper has been identified.

D. Cost models in FaaS

FaaS cost models charge customers based on performance metrics such as execution time, memory or number of invocations [25]. This implies that there is a combined need as well as incentive to tackle the performance-cost tradeoff, especially for runtime management ([2],[12]). The proposed pattern reduces the number of invocations, as well as the execution time (given the lower need for environment initialization), and thus is expected to aid towards minimizing cost aspects.

Approaches including analytical modelling can help improve prediction and estimation of running costs [19] based on used resources and could be used to regulate the batch size in this work, although analytical modelling may need adaptations per service case of investigation. Other works that can be combined include the investigation of trade-off after which a switch to serverless mode is beneficial for the customer [26]. Predicting accurately costs [20] can also be used in combination with this approach. This can be helpful in cases where one needs to regulate/minimize cost directly and not for example based on frequency, if performance is considered less important.

III. Pattern Design, Parameters and Implementation

A. Batch Request Aggregator Pattern Definition

As mentioned in [30], there is no common ground as to what a pattern consists of. Numerous definitions are included in the aforementioned work from existing literature. The most complete and generic seems to be the one provided by [31], which mentions that a pattern is *"a proven series of activities which are supposed to overcome a recurring problem in a certain context, particular objective, and specific initial condition"*. In the particular case the according fields are defined as follows:

- Context: Server-based (threadpool) or serverless environments that rely on heavyweight runtimes and libraries (e.g. AI, numerical computation etc.)
- Recurring problem: even a small number of requests can create performance bottlenecks due to high environment initialization time and concurrency overheads
- Objective: improve response time without having to increase resources, adapt to varying traffic conditions
- Initial condition: ability to measure frequency of request arrival, ability to process batch requests in groups

- Series of activities: the structure defined in Figure 1, which is further analyzed in the following sections.

Analysis of the pattern limitations is also performed, as commonly found in relevant documentation [28]. Proof of the benefits will be provided in Section IV.

B. Pattern Mechanism Rationale and Design

The main purpose of the mechanism is to gather the incoming requests, create arrays of inputs (batches) and launch one processing instance for each batch. This is expected to reduce the number of computational instances for serving the requests and therefore the stress towards the back-end server. In order to achieve the functionality, the pattern needs supporting logic that is presented in Figure 3.

The pattern consists of the following main building blocks:

- A submission endpoint accepting external requests and assigning a message id to each one
- A request accumulation layer that stores the requests in a queue, including a map for indicating which external requests have been included in the batch, based on their message id.
- Release logic that launches the request batch once specific criteria are met (see Section E for details). This logic may include the calculation of request arrival frequency over a time window.
- A response creation layer in which the original incoming messages wait until the response from the main operation is available. When the overall response is available, it is broken down based on the id map, the individual messages are completed and returned to the clients.

A functional programming framework based on function workflows and message passing between function nodes (such as node.js) is a good candidate in order to implement such a logic.

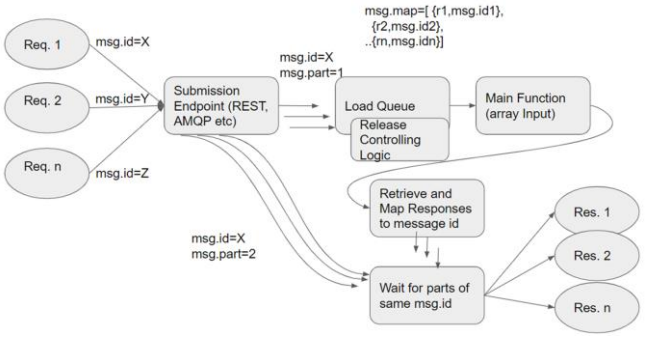

Figure 3: Batch Request Aggregator Pattern Structure

C. Pattern Requirements

In order to apply the respective pattern, the main code modification of the typical function (or microservice) includes its ability to accept an array of the necessary input argument type (regardless of the argument type i.e. integer, double, array, object etc). Then for each element of this array it should process each request, while returning an array of responses in the same order the inputs where received. In case of asynchronous implementations, potential further actions may be required primarily at the pattern side (inclusion of an existing id along with the input). Then the function can include that id in the response in order for the pattern to filter outputs based on input id and recreate the proper responses. The pseudo code for the function modification appears in Figure 4.

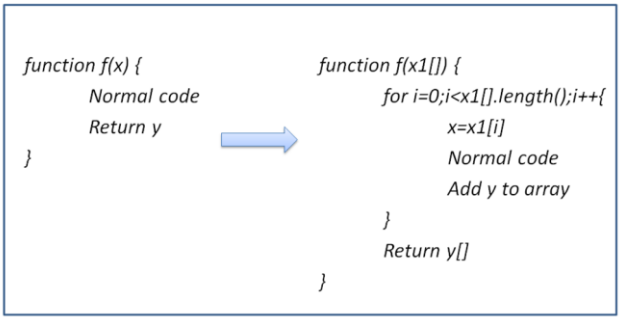

Figure 4: Necessary code modification for pattern application (support for array arguments)

D. Pattern Limitations

One limitation of the pattern is in case some information or state needs to be retrieved from an external repository. Such a case would be for example a function flow in FaaS that retrieves an AI model from object storage. Given that the array input now is executed in sequence, rather than in parallel in the case of multiple function containers, the model retrieval is also executed in sequence which may produce large response time delays. In this case alternate configurations or adjustments may be made such as:

- Batching together only requests that target the same model ID
- Keeping model repositories in volumes that are attached dynamically to the containers

Furthermore, global context usage can not be guaranteed and multitenancy issues may exist (if batching requests from different users). However the latter may be mitigated if presigned URLs (or the Valet Key pattern [3]) are used in order to retrieve the tenant's model from a private repository. Another limitation refers to the fact that the pattern needs to be applied against requests targeting the same function (or service endpoint).

E. Control Releasing Logic/Self regulation

The release logic may determine the conditions under which the aggregated request batch can be forwarded as one message in the backend layer. It is evident that the frequency of request arrivals plays a key role in this decision. If for example requests are sparse, having a large batch size will imply that the first requests need to wait for a considerable time until the batch is complete, resulting in higher waiting times and overall response time. On the other hand, if requests are very frequent, having a small batch size will lead to higher container numbers.

Thus one needs to regulate the parameters of the pattern based on the current conditions of execution in order to optimize the overall result. Indicative approaches for this case can include:

- a simple timeout period, during which the requests are batched. Each arriving request checks the elapsed period from the previous checkpoint and, if complete, it populates a msg.complete field that alerts the next layer to release the batch. Setting the

timeout period as a percentage of the typical response time in single request scenarios can help predict the final response time. For example, a 10%*(averageResponseTime) timeout period will result in only 10% according increase in the final experienced response time, compared to the response time of the service under sparse load.

- a set batch size that needs to be completed before the batch is released. However in this case the batch size needs to be dynamically regulated during runtime, so that it is set to 1 if the request frequency is low or higher if the frequency is high. The control of the batch size can be based on various methods (already investigated in Section II.B), such as PID controllers or other modelling techniques (based e.g. on neural networks),as investigated in the related work. These need to create a model of the response time, that will be fed with the current conditions of the system (i.e. request arrival frequency) and direct the needed metric (e.g. batch size or timeout interval) (as indicated in Figure 1). An example form of such an ANN model appears in Figure 5. Having the response time as the output is needed primarily for validation reasons.

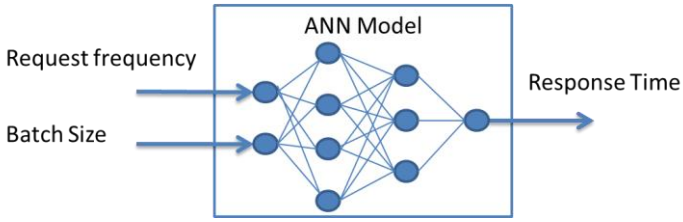

Figure 5: Model Structure for deciding batch size with relation to expected response time

F. Pattern Prototype Implementation

The prototype implementation is built on the Node-RED environment [6]. Node-RED is a visual programming environment for event driven applications, built on top of node.js. It provides a large number of readymade function blocks as well as the ability to include ad-hoc code in functions inside a workflow. To implement the logic of Figure 3, the function flow of Figure 6 was created. Each node receives a triggering message (in our case the initial message is the incoming request for the service), applies the node function logic and passes it to the next, coordinated by the runtime engine. Supporting flows were also created in order to aid in management decisions, such as frequency measurement of requests from the system scope as well as requests towards the controlling logic decision function. Monitoring subflows for logging running container numbers is also included. The pattern and supporting flows implementation is available as open source in [24].

G. Target Application Use Case

For the measurements of Section IV, a service created in the context of [23] was used, that appears in Figure 7. This is a performance estimation service that has created ANN-based performance models, stored in a container volume, and enables the enquiry of these models through a REST endpoint. Whenever a request is received, an equivalent container is launched in order to serve the request, run the environment and give back the prediction. The environment is based on the GNU Octave numerical computation tool, an open source equivalent of Matlab. Based on the structure and operation, this service is a hybrid between serverless and microservice environments (REST endpoint and equivalent container launch for serving a request). It is also the same environment in which the model for correlating batch size to input frequency and response time was created (Figure 5).

In principle, any service that includes large preparation overheads compared to the actual useful computation is a target. Such example services are typically AI-based ones (such as model inference) or approaches that create a large number of requests (e.g. Monte Carlo methods) and are based on heavyweight libraries or frameworks. Given the fact that the pattern acts as a preprocessing layer, it can be applied in both typical microservice, threadpool based servers (in order to reduce the number of concurrent threads in the server) as well as in front of the function invocation gateway (in the case of serverless architectures).

IV. Experimentation

In order to test the performance and benefits of the pattern, a series of experiments were performed, in order to stress the system and observe the altered behavior. The application used is the one described in Section III.G and the orchestration platform was Docker Swarm with a single VM node (4 CPUs, 10GB RAM). No extra resources were made available to the service load in order to evaluate the effect of the pattern without the need to increase the resources. As the main system performance metric, the response time of the service to the various request scenarios and frequencies was considered. Details on these experiments are presented in the subsequent sections.

A. Initial Investigation of Batch Size and Request Period on Response time

Initially a set of separate measurements is performed in order to investigate the effect of the batch size on response times. For this reason a number of different periods (every 0.01, 1 and 5 seconds) of incoming requests and an according diverse batch size (1, 5, 10, 50) is investigated in order to observe their effect on the average response time. The results appear in Figure 8. The timeout is set to 120000 milliseconds, so the respective values in the graph indicate an unresponsive system due to resource contention and high container numbers. The typical mode is 1 container per request without the usage of the pattern. The aggregated mode with a batch size of 1 applies again 1 container per request but this time through the pattern, in order to investigate delays inserted by the pre and post processing layers.

From the measurements it can be portrayed that the pattern includes some delay due to the request management layer (when comparing typical versus aggregated mode of 1 batch size) in the area of ~ 1 second (or <10% of the total response time). But also the benefit from the pattern application is portrayed, in the case of high frequency requests (inter-arrival period of 0.01 seconds). In this case the typical system that was previously unresponsive, when applying the aggregation mode with a sufficiently large batch size (~50) manages to stabilize its behavior as well as

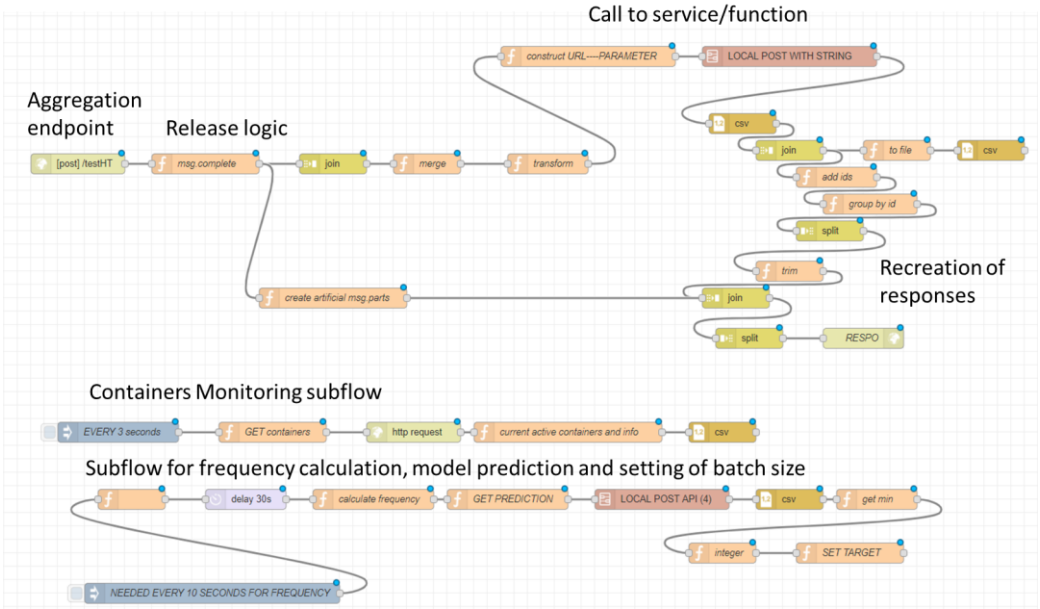

Figure 6: Node-RED flow implementation of the pattern (implementation of Design described in Figure 3)

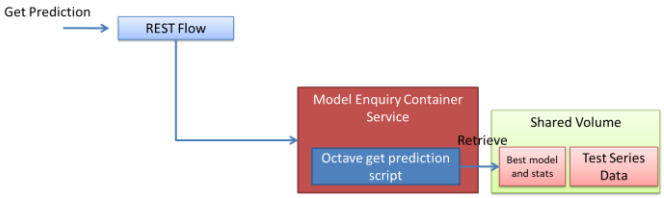

Figure 7: Target service use case (AI based model prediction) for pattern experimentation

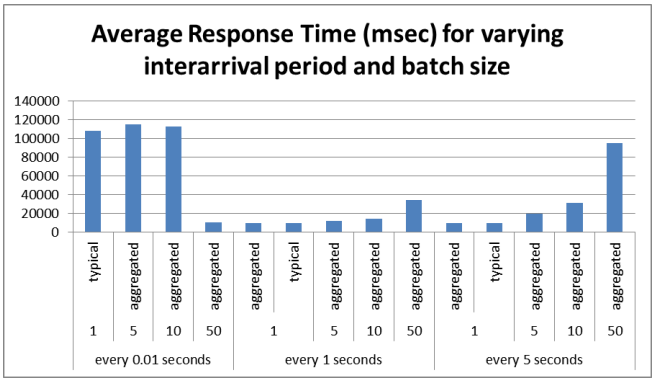

Figure 8: Investigation of batch size and inter-arrival period on the response time average

maintain an average response time that is very close to the average of a single request (10.763 seconds of average response in a period of 0.01 seconds and batch size of 50 versus 10.046 seconds of a typical mode for 1 request every 5 seconds). On the other hand, once the period between requests starts to get higher, the drawback of the pattern is portrayed, given that it needs to wait a large amount of time until the batch size is complete.

B. Runtime Container Numbers for gradually increasing request frequency

In order to investigate the overall performance of the pattern, a series of measurements was performed, applying the timeout variation of the release logic mentioned in Section III.D. Overall the pattern was tested under a scenario in which the request inter-arrival period was gradually reduced, starting from 10 seconds and being reduced to 5,1,0.5, 0.1 and 0.01 seconds of inter-arrival. Each setup was set to run for 15 minutes, unless the platform started to get saturated from container numbers in which case the experiment was halted.

The two main variations that were tested were static delays of 500 and 1000 milliseconds before releasing the batch, which is approximately 5% and 10% of the average response time in a single request. During this time the framework accumulated requests coming in. The results of the experiment are presented in the following figures and are compared to the typical, no-batch implementation. It is indicative that as the experiment progresses and frequencies increase, the batch size (accumulated messages during the timeout interval) increases (Figure 9). This enables the pattern to maintain a constant rate (~20 active containers) of container generation (Figure 10) even at high frequencies (as indicated by Figure 11), in the 10% timeout case (1000 milliseconds of waiting for gathering and batching requests). The no-batch implementation quickly gets saturated (at around sample ~1500 of the experiment), as indicated also from the response times that reach the request timeout limit (Figure 12), reaching very quickly the saturation point of around 80 active containers. This is mapped to a frequency of around 2 messages per second (Figure 11). The 5% timeout is still not sufficient, although it gets saturated a bit later (around sample 5000 of the experiment), achieving a frequency of around 10 messages per second.

C. ANN Model Static Accuracy

The results from the previous runs were subsequently used in order to train an ANN model predictor, based on the description of Figure 5, for getting the expected average response delay (output) for different frequencies (input 1) and different batch sizes applied (input 2). ANNs were chosen since they represent black-box universal approximators and can be applied based on an available dataset, without further knowledge of the internals of the system. In contrast, PID controllers (examined in the Related Work and Section III.E) require a more tailored approach in order to determine the relevant metrics of the system as well as controlled entity, as well as a calibration step. In this case, the main point of interest is whether the pattern can be functionally enriched with an adaptive approach, as well as a comparison of the batch effect compared to the no batch case. Inclusion of comparisons between different adaptive approaches will be pursued in the future.

The ANN model was optimized based on the process defined in [1], using a genetic algorithm in order to determine the main model structural characteristics (number of layers, type of neurons per layer and number of neurons for hidden layers). The network type is based on back-propagation, feed forward architecture and the resulting model was selected from a pool of approximately 450 candidate networks that portrayed an intermediate validation error of less than 20%, based on its performance on a test set. The finally selected network consisted of 5 layers (3 hidden). The first 4 were configured with the tansig transfer function while the output layer with the purelin function.

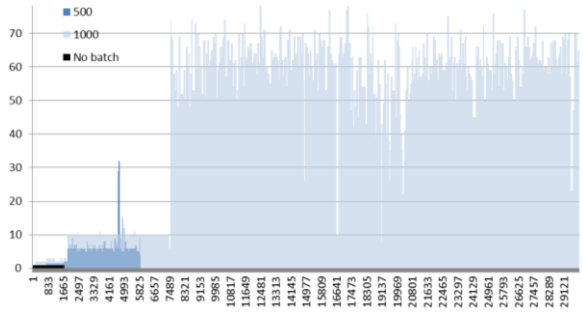

Figure 9: Average batch size of requests (Y axis) in the system as the experiment progresses (X axis:sample number of experiment)

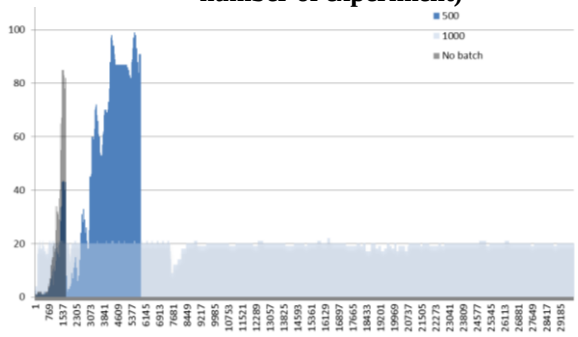

Figure 10: Average number of containers (Y axis) in the system as the experiment progresses (X axis:sample number of experiment)

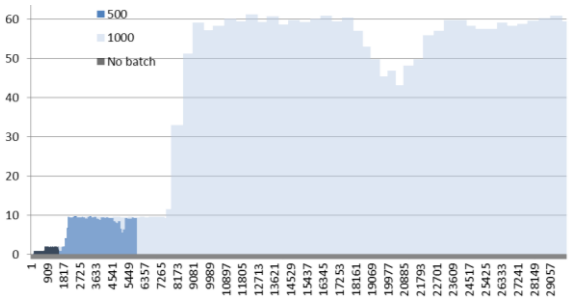

Figure 11: Average frequency of requests (Y axis) in the system as the experiment progresses (X axis:sample number of experiment)

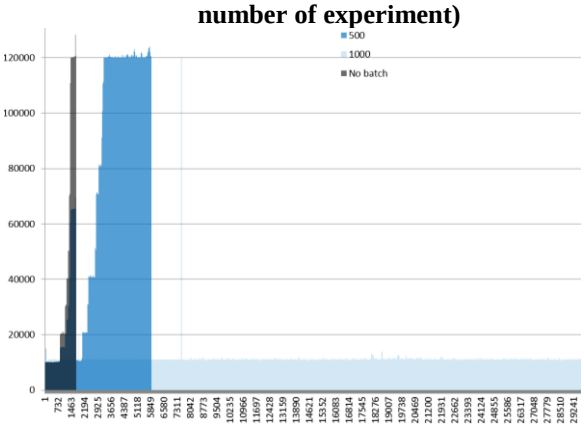

Figure 12: Response time of requests (Y axis) in milliseconds (timeout at 120000) as the experiment progresses (X axis:sample number of experiment)

After the selection of the best model, a further acceptance test was performed to simulate runtime usage. From the available 30k values, ~7k of them were reserved for this purpose. These values were not utilized in whatever manner during the training or optimization process of the model design. The Mean Absolute Percentage Error of 11.73% was achieved (Figure 13), which is considered usable.

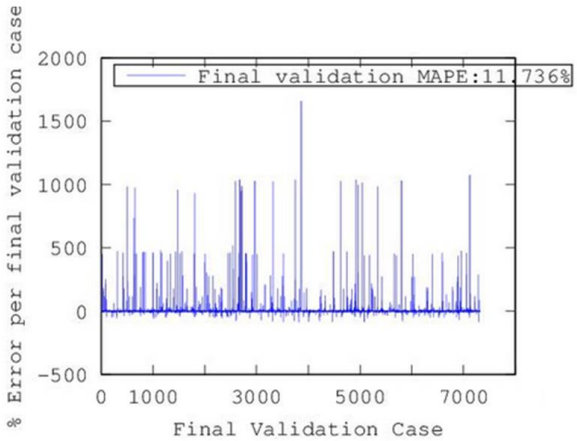

Figure 13: Model Error in final validation cases

The surface plot of the resulting model appears in Figure 14, in the model normalized range. Input 1 is the frequency, input 2 the set batch size and the predicted output is the request response time.

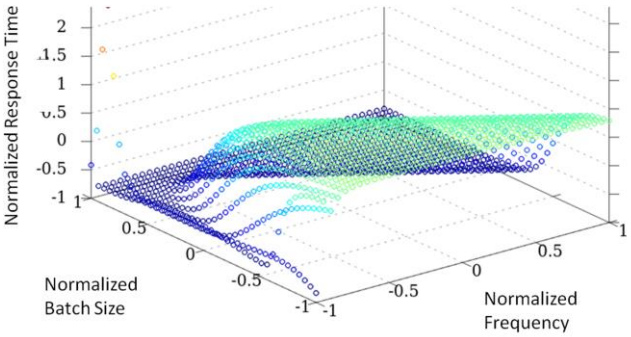

Figure 14: Surface Plot of Model Predictor in the normalized range (Input1: Frequency, Input2: Batch Size, Output: Response Time)

D. Dynamic Adaptation based on the ANN model

In order to test the model's response during runtime, and especially in bursts of load, an experiment was performed with varying loads including sudden peaks (Figure 15). This was performed in order to observe also the effect of the delay in getting the response and how the system gets affected by it in actual conditions, despite the good accuracy of the model in the static validation. In order to decouple this delay from the conditions in the system, the pattern adaptation model was running on a separate node to ensure timely reaction to sudden peaks of load.

The model was queried at a regular interval (every 10 seconds) based on the current conditions of execution (frequency). Given that the model takes approximately 10 seconds to respond, this results in a decision update for the batch size every 20 seconds. Predictions were obtained for different batch sizes (ranging from 1 to 70 with a step of 10). The one achieving the lowest predicted response time was selected and used to configure the pattern. Given that there is a direct relation between frequency and anticipated batch completion, one can also utilize this information in order to configure the time out interval. One of the lessons learned during the first runs was the fact that the model query process needed to run on a different server than the main framework. If not, then due to the saturation caused by the sudden peaks of containers, the model querying process delayed significantly, which led to very delayed responses and server stalling, before being able to adapt.

The results are portrayed in Figure 15 and Figure 16 from the client side (for each individual request sample) and Figure 17, Figure 18 and Figure 19 from inside the system, by taking samples every 1 second for processed frequency, active container numbers and batch size (set and measured). The set is obtained from querying the external model service. From the figures it can be seen that when the increase is gradual (e.g. from 0.2 messages/sec to 4) the system has the time to adapt (Figure 16 in terms of response times and Figure 18 in terms of active containers). Although there is an initial increase in response times, this is fixed by the increased batch size (Figure 19). A further increase from 4 to 8 messages per second (around sample 2000 of Figure 15) does not influence the system, given that the batch size is already high. In this period we also observe further decrease in response times, as would be anticipated by the average times portrayed in Figure 8.

When the load peak is more sudden (from 0.2 to 20 messages/sec), the delay in adaptation leads to a very high response time (in essence reaching the 120000 milliseconds timeout) due to the high number of accumulated messages and according containers. However one major benefit of the pattern in this case is the fact that the system does not stall (like in the case of Figure 10), it only experiences a periodic denial of service. This could be handled either by larger timeout times or by applying in combination with this pattern the Circuit Breaker pattern to refrain the client from generating more requests temporarily.

Another interesting conclusion is that there are sudden peaks in container numbers in the transition time from high to low frequency, not justified by the amount of traffic at that time (tailing of container numbers in Figure 18 following the drop of the peak around sample 400 of Figure 17). This indicates that there are a number of lingering requests, already measured in the frequency, in node.js queues, that start to request resources. However when the frequency is detected as reduced, the system returns to the 1 batch size, making these lingering requests raise one container each. This is an indication that a potentially improved predictor for the ANN would be the number of active containers and not the request frequency. It is also an indication that in other types of adapters (e.g. PID controllers) the past values element (i.e. Integral factor) should be strengthened in order to achieve a graceful, gradual reduction in batch set sizes.

E. Cost Comparison for FaaS cost models

As mentioned in the related work section, typical cost models of FaaS are dependent on the execution duration. Hence in this case a series of measurements was performed in order to investigate what is the overall execution time for a prediction request with multiple input lines, compared to running the model for only one input row for estimation.

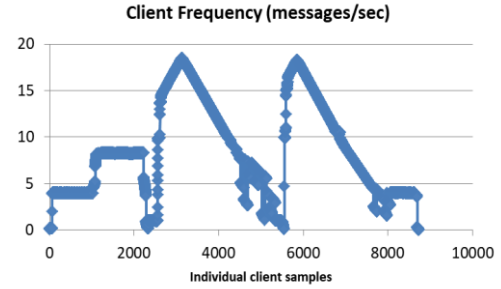

Figure 15: Client side request generation frequency measured per sample

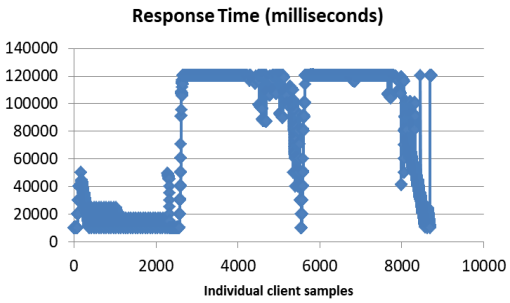

Figure 16: Client Side Response Times per sample

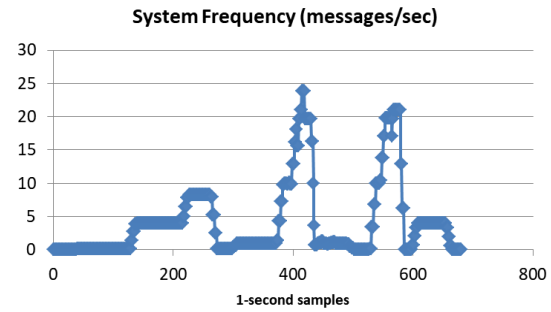

Figure 17: System side measured frequency per 1 second interval

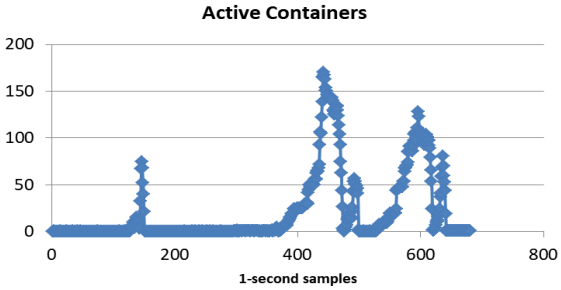

Figure 18: Active (running) containers in the system

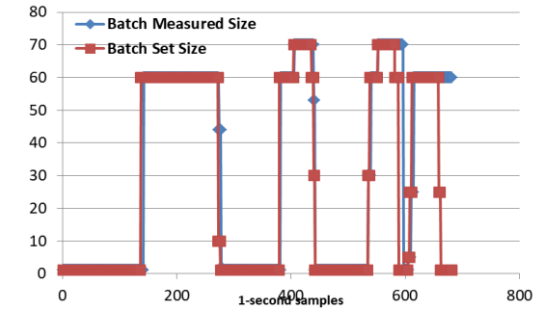

Figure 19: Batch Size Setting by ANN model and according system side application of the setting

The performed requests include an interval of 15 seconds between them, in order to investigate the baseline execution time when only 1 container is active in the system. Input size variations included 1, 100 and 450 input rows for which predictions were needed. Calls to the service were performed for a duration of 10 minutes per case. The results appear in Table 1 and are associated to the estimated costs. The estimation is based on a number of assumptions. All calls are considered as cold-start ones and include also the time to raise the container (which is not billed in actual FaaS offerings). However they are intended as a rough overview of potential benefits.

As a baseline, the cost model of AWS Lambda (US East-Ohio) [27] is used. The containers were allowed to use the overall memory available (10GB). Thus the price of 0.0000001667 dollars per 1 msec of runtime was used and multiplied by the runtime duration, while the cost per input row also appears in the table. This depicts significant difference, which is also evident when taking under consideration the other billing factor (number of requests set at 0.2$ per 1 million of requests) that is also affected by the reduced overall number of calls needed for one to obtain 1 million predictions.

Table 1: Cost Estimation of GB-second based on AWS Lambda pricing model

Input Rows	Average Response Time Delay (msec)	Estimated GB-Second Cost Per Input Row	Cost of Requests for 1M predictions
1	10051.25	0.001675	0.2
100	10082.62	0.000016807	0.002
450	10228.29	0.000003789	0.000444

V. Conclusions and Future Work

The work presented in this paper formulates as a pattern an initially simplistic approach such as request batching, augmented with relevant self-adaptation means and methods. In terms of controlling logic, different approaches have been tried out, starting from simpler ones like static waiting intervals to more complex ones such as training neural networks to undertake the role of pattern parameters setting during runtime. The application of the pattern has indicated significant benefits in terms of improved application performance with the same amount of resources, enhanced stability and responsiveness as well as adaptation to dynamic conditions of execution. Quantification of the benefits has also been measured. Average client response times are stable even in high request frequencies, compared to a high increase in the case of no pattern application. This happens because needed containers are significantly reduced due to the pattern application. Thus the ability to serve more requests with the same number of nodes and the same response time is a clear indication of an improved system performance, especially when the comparison to the no pattern application case is overwhelming. This applies specifically to cases where resource pools are smaller in size and can not be easily scaled (e.g. edge environments).

The requirements for applying the pattern include interface modifications, while relevant limitations have been highlighted. A weakness of the pattern in sudden load peaks has been identified, leading to client timeouts but not a complete system halting. This weakness can be complemented by the Circuit Breaker pattern. This is a common case in pattern design, in which a designed pattern may fit a subset of purposes, while its coupling with other patterns enables the mitigation of its weak points. An alternative mean of mitigation is the coupling with a time series prediction mechanism that will detect upcoming bursts and will be able to set a priori the increased batch size. An application agnostic pattern prototype has been developed in Node-RED as a functional flow, that can be applied in front of any service endpoint to provide the benefits of the pattern and has been provided as open source.

For the future, aspects to be investigated for extending this work refer to including more experimentation regarding controller technology, including PID or control theory approaches. Furthermore, variations of the model creation by taking under consideration active containers directly as the main metric will be investigated. Finally, the porting of the prototype implementation to a more native functional framework (such as native node.js) may be pursued.

Acknowledgment

The research leading to the results presented in this paper has received funding from the European Union's Project H2020 PHYSICS (GA 101017047).

REFERENCES

[1] G. Kousiouris et al., "Parametric Design and Performance Analysis of a Decoupled Service-Oriented Prediction Framework Based on Embedded Numerical Software," in IEEE Transactions on Services Computing, vol. 6, no. 4, pp. 511-524, Oct.-Dec. 2013, doi: 10.1109/TSC.2012.21.

[2] Van Eyk, E., Iosup, A., Seif, S. and Thömmes, M., 2017, December. The SPEC cloud group's research vision on FaaS and serverless architectures. In Proceedings of the 2nd International Workshop on Serverless Computing (pp. 1-4).

[3] Taibi, D., El Ioini, N., Pahl, C. and Niederkofler, J.R.S., 2020. Patterns for Serverless Functions (Function-as-a-Service): A Multivocal Literature Review. In CLOSER (pp. 181-192).

[4] Mahmoudi, N. and Khazaei, H., 2021. SimFaaS: A Performance Simulator for Serverless Computing Platforms, to be published in "The 11th IEEE International Conference on Cloud Computing and Services Science (CLOSER 2021)".

[5] Vergara, S., González, L. and Ruggia, R., 2020, March. Towards formalizing microservices architectural patterns with Event-B. In 2020 IEEE International Conference on Software Architecture Companion (ICSA-C) (pp. 71-74). IEEE.

[6] Node-RED event driven programming environment, available at: https://nodered.org/

[7] Ferreira, J.B., Cello, M. and Iglesias, J.O., 2017, August. More sharing, more benefits? A study of library sharing in container-based infrastructures. In European Conference on Parallel Processing (pp. 358-371). Springer, Cham.

[8] G. Aumala, E. Boza, L. Ortiz-Avilés, G. Totoy and C. Abad, "Beyond Load Balancing: Package-Aware Scheduling for Serverless Platforms," 2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), 2019, pp. 282-291, doi: 10.1109/CCGRID.2019.00042.

[9] D. N. Jha, S. Garg, P. P. Jayaraman, R. Buyya, Z. Li and R. Ranjan, "A Holistic Evaluation of Docker Containers for Interfering Microservices," 2018 IEEE International Conference on Services Computing (SCC), 2018, pp. 33-40, doi: 10.1109/SCC.2018.00012.

[10] Kousiouris, G., Cucinotta, T. and Varvarigou, T., 2011. The effects of scheduling, workload type and consolidation scenarios on virtual machine performance and their prediction through optimized artificial neural networks. Journal of Systems and Software, 84(8), pp.1270-1291.

[11] J. Park, H. Kim and K. Lee, "Evaluating Concurrent Executions of Multiple Function-as-a-Service Runtimes with MicroVM," 2020 IEEE 13th International Conference on Cloud Computing (CLOUD), 2020, pp. 532-536, doi: 10.1109/CLOUD49709.2020.00080.

[12] Eivy, A. and Weinman, J., 2017. Be wary of the economics of" Serverless" Cloud Computing. IEEE Cloud Computing, 4(2), pp.6-12.

[13] Mohan, A., Sane, H., Doshi, K., Edupuganti, S., Nayak, N. and Sukhomlinov, V., 2019. Agile cold starts for scalable serverless. In 11th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 19).

[14] da Silva, R.F., Filgueira, R., Deelman, E., Pairo-Castineira, E., Overton, I.M. and Atkinson, M.P., 2019. Using simple pid-inspired controllers for online resilient resource management of distributed scientific workflows. Future Generation Computer Systems, 95, pp.615-628

[15] Bouras, I., Aisopos, F., Violos, J., Kousiouris, G., Psychas, A., Varvarigou, T.A., Xydas, G., Charilas, D. and Stavroulas, Y., 2019. Mapping of Quality of Service Requirements to Resource Demands for IaaS. In CLOSER (pp. 263-270).

[16] Xing Chen, Haijiang Wang, Yun Ma, Xianghan Zheng, Longkun Guo, Self-adaptive resource allocation for cloud-based software services based on iterative QoS prediction model, Future Generation Computer Systems, Volume 105, 2020, Pages 287-296, ISSN 0167-739X, https://doi.org/10.1016/j.future.2019.12.005.

[17] Claus Pahl, Pooyan Jamshidi, and Olaf Zimmermann. 2018. Architectural Principles for Cloud Software. ACM Trans. Internet Technol. 18, 2, Article 17 (March 2018), 23 pages. DOI:https://doi.org/10.1145/3104028

[18] I. B. Fraj, Y. B. Hlaoui and L. BenAyed, "A control system for managing the flexibility in BPMN models of cloud service workflows," 2020 IEEE 13th International Conference on Cloud Computing (CLOUD), 2020, pp. 537-543, doi: 10.1109/CLOUD49709.2020.00081.

[19] C. Lin and H. Khazaei, "Modeling and Optimization of Performance and Cost of Serverless Applications," in IEEE Transactions on Parallel and Distributed Systems, vol. 32, no. 3, pp. 615-632, 1 March 2021, doi: 10.1109/TPDS.2020.3028841.

[20] R. Cordingly, W. Shu and W. J. Lloyd, "Predicting Performance and Cost of Serverless Computing Functions with SAAF," 2020 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech), 2020, pp. 640-649, doi: 10.1109/DASC-PICom-CBDCom-CyberSciTech49142.2020.00111.

[21] Function Concurrency in Openwhisk containers, available at: https://github.com/apache/openwhisk/blob/master/docs/concurrency.md

[22] Function Concurrent activation across namespace, available at: https://github.com/apache/openwhisk/blob/master/docs/reference.md

[23] George Kousiouris and Dimosthenis Kyriazis. 2021. Enabling Containerized, Parametric and Distributed Database Deployment and Benchmarking as a Service. In Companion of the ACM/SPEC International Conference on Performance Engineering (ICPE '21). Association for Computing Machinery, New York, NY, USA, 77–80. DOI:https://doi.org/10.1145/3447545.3451188

[24] Pattern implementation prototype, available at : http://bigdatastack-tasks.ds.unipi.gr/gkousiou/adw/-/blob/master/adwdocker/patterns/request_aggregation_pattern

[25] Grogan, J., Mulready, C., McDermott, J., Urbanavicius, M., Yilmaz, M., Abgaz, Y., McCarren, A., MacMahon, S.T., Garousi, V., Elger, P. and Clarke, P., 2020, September. A Multivocal Literature Review of Function-as-a-Service (FaaS) Infrastructures and Implications for Software Developers. In European Conference on Software Process Improvement (pp. 58-75). Springer, Cham.

[26] Lin, X.C., Gonzalez, J.E. and Hellerstein, J.M., 2020. Serverless Boom or Bust? An Analysis of Economic Incentives. In 12th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 20).

[27] AWS Pricing model, available at: https://aws.amazon.com/lambda/pricing

[28] Microsoft Cloud Design Patterns Catalogue and documentation, available at: https://docs.microsoft.com/en-us/azure/architecture/patterns/index-patterns

[29] Kosińska, J. and Zieliński, K., 2020. Autonomic Management Framework for Cloud-Native Applications. Journal of Grid Computing, 18(4), pp.779-796.

[30] Asma Hachemi. 2020. Software Development Process Modeling with Patterns. In Proceedings of the 2020 The 2nd World Symposium on Software Engineering (WSSE 2020). Association for Computing Machinery, New York, NY, USA, 37–41. DOI:https://doi.org/10.1145/3425329.3425339

[31] Agung Wahyudi, George Kuk and Marijn Janssen. A Process Pattern Model for Tackling and Improving Big Data Quality. Information Systems Frontiers (2018) 20:457–469. Springer. https://doi.org/10.1007/s10796-017-9822-7

[32] Mohammad Reza Saleh Sedghpour, Cristian Klein, and Johan Tordsson. 2021. Service mesh circuit breaker: From panic button to performance management tool. In Proceedings of the 1st Workshop on High Availability and Observability of Cloud Systems (HAOC '21). Association for Computing Machinery, New York, NY, USA, 4–10. DOI:https://doi.org/10.1145/3447851.3458740

[33] GNU Octave Scientific Programming Tool, available at: https://www.gnu.org/software/octave/index

[34] T. Cucinotta et al., Virtualised e-learning on the IRMOS real-time cloud, Service Orient. Comput. Appl. (2011). doi: 10.1007/s11761-011-0089-4 (pages 116).