WasteLess: An Optimal Provisioner for Self-Adaptive Second-Generation Serverless Applications

Emilio Incerto
Roberto Pizziol
{emilio.incerto,roberto.pizziol}@imtlucca.it
IMT School for Advanced Studies
Lucca, Italy

Gabriele Russo Russo
russo.russo@ing.uniroma2.it
University of Rome Tor Vergata
Rome, Italy

Mirco Tribastone
mirco.tribastone@imtlucca.it
IMT School for Advanced Studies
Lucca, Italy

***Abstract*—Function-as-a-service (FaaS) is enjoying widespread adoption in the cloud thanks to the ability of dynamically provisioning function instances (typically containers) to seamlessly handle invocation bursts. However, in practice, this mechanism may result in resource overprovisioning and increased operational costs when function resources are not configured properly. Besides memory and CPU cores, so-called *second-generation* FaaS systems let users specify the concurrency limit, enabling the consolidation of multiple requests into a single function instance to improve latency and cost. However, this introduces yet another configurable parameter, which is hard to tune manually. We present WasteLess, a resource provisioning framework designed to automate and optimize the deployment of modern FaaS self-adaptive applications. By leveraging a predictive queuing network model, WasteLess computes configurations to maximize application performance while minimizing costs. Our evaluation demonstrates the effectiveness of WasteLess in achieving average cost savings of about 40% without performance degradation with respect to a state-of-the-art research baseline and Google Cloud Run's self-adaptive strategy for second-generation FaaS.**

***Index Terms*—Serverless computing, Function-as-a-Service, resource allocation, performance optimization**

I. Introduction

Market analysts unanimously acknowledge the robust market potential of serverless computing, forecasting compound annual growth rates ranging from 21% to 28% in the coming years. Its appeal lies in anticipated cost reductions, reduced operational overhead, and inherent scalability, making it an ideal choice for building distributed self-adaptive applications with stringent performance requirements [1].

Function-as-a-Service (FaaS) has gained significant traction as a key enabler of serverless computing and is now a core component of all major cloud offerings (e.g., AWS Lambda[1], Google Cloud Functions[2], and Azure Functions[3]). With FaaS, developers define loosely coupled code units (i.e., functions) executed in an event-driven fashion. These functions run within ephemeral containers (i.e., *instances*) that are dynamically spawned upon service invocation. In this setting, users are billed based on the number of active function instances (i.e., *billable instances*) per unit of time. Therefore, optimizing the number of instances is crucial for creating efficient and cost-effective applications.

FaaS leaves resource allocation mainly under the control of cloud providers, who take care of adapting applications at runtime to eliminate queuing times and allow functions to quickly scale in response to fluctuating workloads [2]. While this model captivates developers with the promise of effortless, highly scalable and cost effective deployments, the behavior of FaaS applications still depends on different configuration knobs. Currently, these knobs are manually tuned by application designers, who must specify suitable amounts for memory and CPU cores to reserve for each function instance, trading off performance with cost.

Relying on the manual configuration of these parameters casts a shadow over the real economic benefits of FaaS [3], [4]. Approaches to assist designers in choosing the right "function size" have been proposed (e.g., COSE [5], Sizeless [6]). However, choosing the best cost-performance trade-off for a given application ultimately remains under designers' responsibility, and the following challenges are still open:

1) Performance and cost trade-off heavily depends on complex systems configurations.
2) There are limitations in existing automatic configurators that lead to performance degradation or unnecessary operational costs in many scenarios.

To enhance FaaS resource efficiency, *second-generation* functions have recently been introduced, enabling multiple invocations to run concurrently within the same function instance.[4] While this approach reduces the resources provisioned for a given workload, it introduces a new configurable parameter, i.e., *the concurrency limit*, representing the maximum number of concurrent invocations per instance. This parameter significantly affects performance: setting it too high can cause resource contention and performance degradation,

[1]https://aws.amazon.com/it/lambda/
[2]https://cloud.google.com/functions/
[3]https://azure.microsoft.com/en-us/products/functions/
[4]https://cloud.google.com/functions/docs/concepts/version-comparison

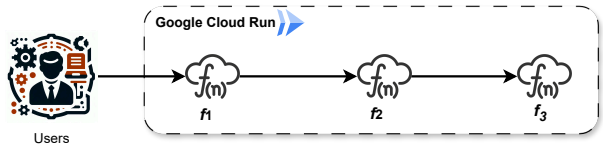

Fig. 1: Overview of the three-tier FaaS motivating example.

while setting it too low undermines the cost efficiency of consolidating requests within a single instance.

To assist developers, Google Cloud Run (GCR) offers an automatic instance scheduler for second-generation functions. It is a self-adaptive mechanism that, before spawning new instances, attempts to accommodate newly arrived requests into the existing ones. A new instance is only created if the concurrency limit is reached, or the CPU utilization of a function exceeds a platform-defined threshold.[5] In the following, we demonstrate how this adaptive instance scheduler may lead to potential cost savings but with significant performance degradation if developers misconfigure the concurrency limit.

To address these challenges, we present *WasteLess*, a novel optimal resource provisioning framework for second generation self-adaptive FaaS applications.

> Using WasteLess, FaaS developers can automatically assign the amount of memory, CPU cores, and concurrency limit of each function to jointly minimize response time and the allocation costs.

WasteLess consists of three main components: i) a stochastic model predicting the performance of FaaS applications (response time and throughput) based on hardware/software configurations; ii) an efficient optimization process to configure functions for minimal average response time at minimal cost; iii) a memory configurator that assigns appropriate memory to each function based on its concurrency limit. The memory configurator is particularly crucial, as increasing concurrent users requires proportionate memory allocation to prevent performance issues or out-of-memory errors. These aspects are further detailed in Section IV.

Notably, WasteLess needs to compute optimal configurations *only once*, avoiding the need for costly online optimizations. It achieves this by integrating with and guiding the GCR instance scheduler to achieve a better performance-cost trade-off. To accomplish this, WasteLess calculates the concurrency limit for each function based on the ratio between the actual concurrency level, as estimated by the layered queuing network (LQN) [7] of the FaaS system, an the the minimum number of cores required to minimize the average system response time (i.e., the solution of WasteLess's optimization phase). This ratio remains constant as the number of users and functions instances varies; therefore it can be effectively used in combination with the traditional GCR instance scheduler (see Section IV for a more detailed explanation)

[5]It is important to note that Google, while mentioning CPU utilization as a scaling factor, doesn't publicly quantify this threshold: https://cloud.google.com/run/docs/configuring/services/cpu, https://cloud.google.com/run/docs/about-instance-autoscaling

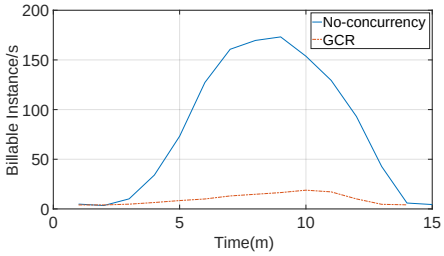

Fig. 2: Motivating example results.

TABLE I: Average end-to-end response time of the motivating example under two different concurrency limit strategies: No-concurrency and GCR.

	No-concurrency	GCR
Average response time (s)	2.73	15.71

***Motivating Example**:* To illustrate the motivation behind our proposal, consider this simple example (see Section V for a deeper numerical evaluation of WasteLess). Figure 1 depicts a three-tier FaaS application deployed on GCR, consisting of three CPU-bound functions. Each function synchronously calls the next, waiting for the response before replying to its own caller. While waiting, the caller does not use CPU resources. Thus, creating additional instances of the caller function results in unnecessary resource allocation. Since, among various factors (e.g., number of invocations, maximum runtime, storage [4]), GCR bills based on the number of active function instances, users incur unnecessarily higher costs. Adjusting the concurrency limit of the caller function would suffice. However, setting concurrency limit too high can lead to resource contention, degrading overall performance.

We evaluated the application under two different concurrency limit allocation strategies: no-concurrency, and GCR instance scheduler with the default value for the concurrency limit, i.e, 80,[6]. With "no-concurrency" we refer to the *first-generation* of serverless functions, allowing the execution of only one request per function instance; this is equivalent to setting the concurrency limit to 1 in second-generation functions. Thus, in the no-concurrency scenario, all resources are reserved for a single request. As a final remark, we highlight that in this simple example we configured each function with an amount of memory suitable to support the maximum concurrency limit set across all evaluated strategies.

Figure 2 shows the number of billable instances under a sinusoidal load of requests, while Table I reports the average end-to-end response time under the different concurrency limit allocation strategies. The results highlight the significant impact of concurrency on the performance/cost trade-off in FaaS systems. Indeed, GCR instance scheduler may leads to a substantial reduction in the total number of used billable

[6]https://cloud.google.com/run/docs/about-concurrency

instances (i.e., around 8 times smaller), but at the expense of an excessive deterioration in the application's response time, around six times higher on average, due to an excessive concurrency limit. This simple example reinforces our motivations:

- Effective FaaS configuration still relies on user input
- Automated FaaS configurators may result in significant performance degradation if not properly set up.

To further strengthen our proposal, in Section V we compare the cost and performance improvements produced by WasteLess with a recent research approach, namely ProPack [8], specifically designed for packing multiple function invocations within the same instance. Its main purpose is to enable concurrency for FaaS functions at the application level, even without support from the underlying platform. Although it has proven effective in determining the concurrency limit of isolated functions, its self-adaptation capabilities are limited since it leaves the sizing of the function's CPU and memory still under the user's responsibility.

Our contribution. The optimization process at the core of WasteLess is driven by the solution of a LQN model [7], capable of representing the key performance characteristics of a FaaS system (i.e., throughput, utilization, response time and memory) as a function of its configuration parameters (e.g., cores and concurrency limit).

LQNs are a class of stochastic processes that have been widely used for modeling applications of different domains [9], [10], [11], [12] including FaaS [13]. As minor technical contribution, within WasteLess we extend the usage of LQN to estimate the memory requirements of each function. To achieve this, we establish a relation between the number of active users in each function, computed by an LQN, and the measured memory usage. This relation is then used to predict the memory required by a function when deployed with the optimal concurrency limit. We model the relationship between memory and the number of active users using polynomial regression, computed during the initial profiling phase of WasteLess (see Section IV for more details).

Unfortunately, using LQNs within some optimization process makes the problem highly nonlinear, hence very challenging to solve [14]. WasteLess overcomes this issue by using an analytical representation of the LQN called the *fluid approximation* [15], an established technique for Markov chains that has been employed in a variety of applications [11], [12], [16], [17], [18], [19]. Fluid LQNs are particularly suitable for complex systems with many elements and users, such as serverless or microservices systems, where they have been successfully applied [12], [20].

The fluid approximation represents an LQN using a compact system of coupled ordinary differential equations (ODEs). By encoding this LQN analytic representation as constraints of a differentiable nonlinear optimization problem, WasteLess can efficiently devise hardware (i.e., memory and CPU) and software (i.e., concurrency limit) configurations that minimize application average response time at the minimum cost.

We validated WasteLess by considering a FaaS porting of *AcmeAir*, a well-known benchmark taken from the literature [10], [12], [21], [22]. The validation demonstrates the ability of WasteLess to deploy FaaS applications with an average cost saving of 40% compared to its non-concurrent counterpart, along with a 40%–55% latency reduction compared to the commercial GCR instance scheduler and ProPack, respectively.

Our contributions can be summarized as follows:

- We present WasteLess, a resource provisioning framework for FaaS applications, which jointly optimizes CPU, memory and functions concurrency limit.
- We devise a LQN fluid approximation encoding for the accurate yet efficient performance modeling and optimization of FaaS applications.
- We integrate WasteLess with GCR and compare it against its instance scheduler provided as a commercial solution by Google, as well as against ProPack which is the reference baseline from the literature.

The remainder of this paper is organized as follows. We review related works in Sec. II. We discuss the adopted modeling framework in Sec. III while WasteLess's design is described in Sec. IV. In Sec. V we present the experimental results. In Sec.VI we discuss the potential threats to validity and conclude in Sec. VII.

II. Related Work

Very close to our works, Lin et al. consider the problem of modeling and optimizing performance and cost of FaaS applications for the mean [23] and the distribution [24] of response times using an extension of stochastic Petri nets (SPN), with the objective to optimize memory allocation. WasteLess optimizes of additional parameters i.e., CPU cores and concurrency limit besides memory.

The focus on memory optimization is a common trait of other model-based approaches : COSE [5], based on Bayesian optimization; Sizeless [6], which uses a multi-target regression neural network; and Costless [25] which proposes an algorithm that optimizes the cost of serverless applications by strategically fusing, placing, and allocating memory to functions within a workflow. Still in the memory-only optimization space are Orion [26] and then extended in WiseFuse [27] which optimize with respect to response time distributions.

Roy et al. [8] present ProPack, a solution that "packs" multiple functions instances into a single container and executes them concurrently without an explicit support of the underlying FaaS framework. To choose the optimal concurrency limit, ProPack constructs an approximate model of expected response time and cost for varying concurrency limit of a single function. However, it is affected by two major limitations: i) it neglects the topology of the entire application, a crucial aspect considered in our work, ii) it leves the CPU and memory sizing of FaaS functions still under designers' responsibility.

For an extensive numerical comparison of WasteLess against ProPack see Sec. V.

To our knowledge, in the context of FaaS LQNs have been exploited only in COCOA [28]. Differently from our work, the authors consider a capacity planning problem, where they optimize the amount of resources allocated to the whole FaaS system (i.e., number of CPUs and memory) and the "idle time" of each function (i.e., the maximum time an idle container of a function is kept alive before being terminated). In the cloud scenario we target, these parameters are not exposed to us, and we instead focus on (second-generation) function-level resource configuration, including concurrency limit.

While function sizing has been regarded as a key issue in the adoption of cloud-based FaaS, it is worth remarking that other performance issues – which are out of the scope of this work – have received significant attention from researchers, including, e.g., auto-scaling [2], [29], [30], scheduling [2], [31], [32], [33], and cold start reduction and mitigation [34], [35], [36], [37], [38]. Our approach is complementary to these.

Concurrency tuning for FaaS applications resembles soft resource allocation for microservices (e.g., [39], [40]). However, the two problems also show important differences, as (i) FaaS applications are subject to particular dynamics (e.g., cold starts) not encountered in microservice applications, and (ii) FaaS is characterized by its own pricing model, which impacts resource allocation optimization.

III. Layered Queueing Networks for FaaS Applications

In this section, we introduce the performance model we rely on and its analytical representation.

A. Layered Queuing Networks (LQNs)

LQNs extend traditional queuing networks with multiple layers, resource types, and requests [7], [41]. Figure 3 shows the LQN model of the three-tier system from Figure 1.

Tasks, depicted as parallelograms, represent service points (e.g., $\texttt{F}_1$ in Figure 3). The replication factor within angular brackets indicates the number of task replicas, while task multiplicity within curly brackets denotes the number of software threads per replica. Each task maps to a processor (depicted as a circle), representing the hardware executing each thread. Processor multiplicity (number of cores) is shown within angular brackets.

Within our framework, each task maps to a specific FaaS function as shown in Figure 3. Tasks $\texttt{F}_1$, $\texttt{F}_2$, and $\texttt{F}_3$ represent functions f_1, f_2, and f_3, respectively. The function CPU configuration (i.e., number of cores) is reflected in the LQN by the processor multiplicities c_i with $i \in \{1, 2, 3\}$. The concurrency limit is captured by the thread pool size m_i, while the number of parallel instances activated for the function is modeled by the replication factor r_i.

Each LQN model includes a reference task to model users' behavior, such as the `Client` task in Figure 3. The thread-pool size of this task (m_c) represents the number of users and is linked to a processor with equivalent parallelism. This reference task simulates users repeatedly running services separated by (exponentially distributed) delays.

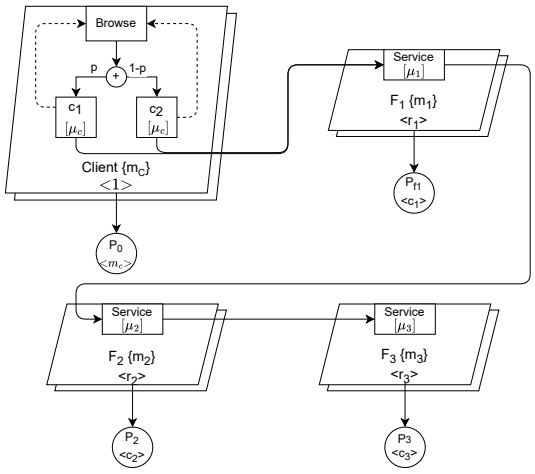

Fig. 3: Running example – LQN model of the three-tier system of Figure 1.

Entries (small rectangles within tasks) represent service types, specified by directed acyclic graphs (DAG) of activities and control nodes. Each activity (rectangle) is characterized by a service time (in square brackets), indicating average processor demand.

Without loss of generality, we assume FaaS functions to have a single service endpoint, represented by a single entry named `Service`. In the example, each function has a simple behavior: it sends a request to the nested layer and waits for a response. Therefore, each entry has a single activity with its average service time specified beneath it (see μ_1, μ_2, and μ_3 in Figure 3). To illustrate an entry behavior with multiple activities, we include a probabilistic choice between two activities within the client logic (the DAG of the `Browse` entry of the `Client` task), functionally equivalent to the example in Figure 1.

Finally, function interactions (e.g., HTTP requests) are represented by arcs connecting tasks, from any activity of the calling entry to the root activity of the called one. These interactions can be synchronous (closed arrowheads) or asynchronous (open arrowheads), with the average number of requests per invocation indicated by a number in parentheses (defaulting to 1 if omitted).

B. LQN Fluid Approximation

The fluid approximation is a dynamical model represented by coupled ordinary differential equations (ODEs) that capture the average behavior of the LQN over time [15], based on a fundamental result from the theory of stochastic processes [42]. Such approximation can be derived automatically starting from a syntactic description of the LQN model [43].

Using the fluid approximation, we simplify the model by treating a FaaS function deployed on r instances as a function on a single instance with a thread pool size and number of

cores scaled by r. This reduction is exact when the scheduling mechanism between instances is work-conserving [44].

With reference to the example in Fig. 3, we show the equations defining variables $\dot{x}_{\{\cdot\}}$, which represent the time derivative (i.e., the rate of change) of the number of users at a specific model location over time. For instance, for f_1 we get

$$\dot{x}_{f_{1_e}} = T_{c_1}(x) + T_{c_2}(x) - T_{f_1}(x) \tag{1}$$

These derivatives are combinations of incoming and outgoing user flows, representing movements within the model. For example, in (1), $T_{c_1}(x)$, $T_{c_2}(x)$ and $T_{f_1}(x)$ denote the rates at which users submit requests to F_1 and the rate at which F_1 processes its requests, respectively. For example, these rates can be formally defined as follows:

$$T_{c_1}(x) = \mu_{c_1} x_{c_1} \tag{2}$$

$$T_{c_2}(x) = \mu_{c_2} x_{c_2} \tag{3}$$

$$T_{f_1}(x) = \mu_{f_1} \min\{x_{f_1}, c_{f_1}\} \tag{4}$$

The fluid approximation interprets original integer quantities in the model, like the number of users in queue or the number of cores assigned to a task, as continuous variables that change over time (since it estimates their mean value). Roughly speaking, flooring (or ceiling) such quantities becomes asymptotically exact as the number of users increases [42].

Performance Indices Computation: To derive optimal configurations, WasteLess relies on the steady-state solution of a fluid LQN, i.e., a classing setting for performance evaluation indicating situation that is away from the initial conditions of the model, formally obtained by setting $\dot{x} = 0$, i.e., setting the derivative to zero and imposing a constant solution. Let x^* be such solution. We now illustrate how to compute throughput, response times, and the total number of active clients in our running example using queuing theory and the *Little's law* [45].

The throughputs for tasks `Client` and F_1, F_2, and F_3 are given by:

$$\mathcal{T}_{Cl}=T_{c_1}(x^*)+T_{c_2}(x^*) \tag{5}$$

$$\mathcal{T}_{f_1}=T_{f_1}(x^*) \tag{6}$$

$$\mathcal{T}_{f_2}=T_{f_2}(x^*) \tag{7}$$

$$\mathcal{T}_{f_3}=T_{f_3}(x^*) \tag{8}$$

The sum in (5) is justified by the fact that activities c_1 and c_2 contribute concurrently to the processing rate of each user. The respective total number of active users are given by:

$$\mathcal{M}_{Cl}=x^*_{c_1}+x^*_{c_2}+x^*_{c\to f_1} \tag{9}$$

$$\mathcal{M}_{f_1}=x^*_{f_{1_e}}+x^*_{f_1\to f_2} \tag{10}$$

$$\mathcal{M}_{f_2}=x^*_{f_{1_e}}+x^*_{f_1\to f_2} \tag{11}$$

$$\mathcal{M}_{f_3}=x^*_{f_{1_e}} \tag{12}$$

Essentially, for each task we sum all the users currently using the CPU at a given moment in time, plus those waiting to receive responses from the nested layers. Then, using the quantities introduced so far the respective average response times can be computed as:

$$\mathcal{R}_{Client}=\mathcal{M}_{Cl}/\mathcal{T}_{Cl} \tag{13}$$

$$\mathcal{R}_{f_1}=\mathcal{M}_{f_1}/\mathcal{T}_{f_1} \tag{14}$$

$$\mathcal{R}_{f_2}=\mathcal{M}_{f_2}/\mathcal{T}_{f_2} \tag{15}$$

$$\mathcal{R}_{f_3}=\mathcal{M}_{f_3}/\mathcal{T}_{f_3} \tag{16}$$

where we derived equations (13)–(16) by applying Little's law to the previously introduced steady-state performance indices.

IV. WasteLess

This section provides a comprehensive discussion of WasteLess. Here we present it in the context of a generic abstract FaaS provider that has the following features:

1) it allows us to monitor CPU utilization, memory consumption, concurrency limit and actual concurrency level, and response time, for each function;
2) it exposes an API that can configure CPU core count, memory allocation, and concurrency limit per function.

An overview of WasteLess is illustrated in Figure 4. As input, WasteLess takes a FaaS application comprising a set of functions that communicate via HTTP messages.

WasteLess operates in three distinct phases, each depicted by a dashed rounded box in Figure 4. The initial phase serves two primary purposes, both originating from the same profiling campaign: (i) constructing a well-calibrated LQN model of the FaaS application, enabling performance prediction across diverse configurations; (ii) establishing a mapping between the concurrency limit and memory footprint of each function.

In the second phase, utilizing the fluid representation of the LQN (see Section III-B), a differentiable nonlinear optimization problem is formulated. This problem incorporates the fluid LQN's ODEs as constraints, with decision variables representing the LQN state and the processor multiplicities of each function. Solving this problem yields the ratio between the optimal concurrency limit and the number of cores.

In the final phase, WasteLess translates this ratio into concrete configurations for adjusting the deployment of each function within the FaaS system. This phase consists of two steps: firstly, the optimization result directly determines the core count and concurrency limit assigned to each function. Secondly, the optimal number of active users is utilized to compute the memory required by each function to support the concurrent configuration. The subsequent sections elaborate on these phases in detail.

❶ *LQN Model Creation & Concurrency/Memory Mapping*

This phase aims to discover the LQN model of the input FaaS application and establish a link between function concurrency limitand memory usage. Both require initial profiling to gather information. Details on LQN model creation, concurrency/memory mapping, and profiling are reported below.

1a *Profiling*

Here, we gather essential information to construct an LQN model of the FaaS application and a model mapping the concurrency limit of each function to its memory consumption.

To achieve this, we assume the ability to exercise the application under study by generating a swarm of users submitting requests. We utilize standard profiling and load testing tools for this purpose. It is important to note that the profiling protocol varies depending on the intended purpose of the collected data, i.e., LQN model creation or mapping the concurrency limit

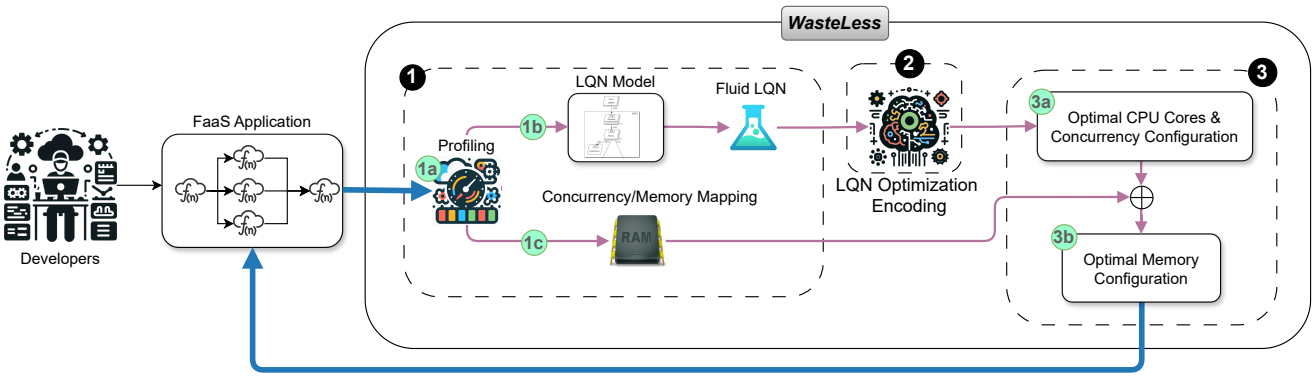

Fig. 4: Overview of WasteLess.

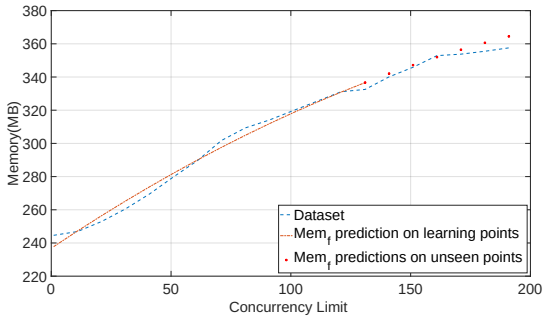

Fig. 5: Example of WasteLess Concurrency/Memory Mapping. The x-axis denotes the concurrency limit of a function while the y-axis is the required memory.

over memory. We will elaborate on these variations as we introduce the procedures for model creation.

1b *LQN Model Creation*

The LQN model is constructed in two distinct phases. Initially, the model's topology is derived by analyzing workflow artifacts, as detailed in [20]. These artifacts outline the application's composition in terms of serverless functions and their interconnections. A workflow specification serves as a blueprint, dictating how different functions interact to achieve a desired outcome. Workflows are typically represented as DAGs, where nodes symbolize serverless functions and edges represent dependencies among them, often manifested as HTTP requests. Following the approach in [20], the type of connection (synchronous or asynchronous) and the presence of branching behavior can be extracted from these artifacts. This DAG representation readily translates into an LQN structure, as the one reported in Figure 3. However, at this stage, the LQN parameters, i.e., the processing rates (μ_i) and routing probabilities (p), remain undetermined. Below we detail the systematic profiling procedure employed for their calibration. This procedure involves executing the FaaS application under a controlled single-user load and measuring the CPU time consumed by each function. This metric was chosen due to its direct correspondence to service time in the LQN model, simplifying data collection and reducing profiling duration. Routing probabilities are derived by annotating the control flow of each function to track the number of executions of different paths, following an approach similar to that in [11]. Probabilities are then estimated as the ratio of executions of a given path to the total number of executions.

Since at this stage the optimal FaaS configurations is not known, each function is configured to utilize 1 core and a concurrency limit of 1, allowing the serverless platform to automatically scale the number of function instances. Memory configuration primarily requires ensuring that total consumption remains within an acceptable range, such as 80% of total memory, to prevent disruptive events like out-of-memory exceptions. Memory settings can be adjusted through trial and error while monitoring for such exceptions.

To accurately capture the system's inherent stochastic behavior, the profiling session continues until a predetermined level of statistical convergence is achieved for the measured quantities. The batch means algorithm [46] is employed, with a stopping condition where the 95% confidence interval width of the mean CPU time falls below 1% of its mean value.

Finally, following a procedure similar to that in Section III-B, the set of ODEs describing the LQN can be derived and used to define the structure of the WasteLess optimization problem. To simplify this task, we developed a tool included in our replication package [47].

1c *Concurrency/Memory Mapping*

The objective here is to establish the analytic relation $Mem_f(u) : \mathbb{R}^+ \rightarrow \mathbb{R}^+$, which links the concurrent users u on each function to its memory usage. This relationship guides the memory sizing for each function based on the actual concurrency level determined by WasteLess. This step is crucial as WasteLess may optimize concurrency limit, potentially exceeding the memory allocated in a non-concurrent deployment, causing performance issues.

To derive Mem_f, we conduct a profiling process where the application is run under increasing loads, monitoring memory usage. Each function is configured with a single instance and high concurrency limit to induce memory variation. Collected

data is then fitted into a polynomial function correlating the concurrency limit to memory usage for each function.

As a simple example, Figure 5 shows the evaluation of a possible instance of Mem_f synthesized to predict the memory usage of a generic CPU-bound serverless function. As a model we used a second-degree polynomial trained on a subset of the measured dataset consisting of 14 (concurrency limit, memory) pairs out of 20 available. We performed the fitting using the *polyfit* function natively available in MATLAB. Figure 5 shows three curves: all the measured (concurrency limit, memory) pairs (dashed line), the memory usage predictions computed on the points used for training (dotted line), and the memory usage predictions calculated on concurrency limit not used for training. The results demonstrate the effectiveness of this method. We leave the exploration of alternative methods for discovering this mapping as future work.

❷ *WasteLess Optimization Problem*

The goal of this section is to introduce the optimization problem employed by WasteLess to determine optimal configurations for each function. The structure of this optimization problem is outlined below.

$$\min_{c} \quad \psi\mathcal{R}(c,x,\theta) + (1-\psi)\sum_{p\in Proc} c_p \tag{17}$$

s.t.

$$\dot{x} = 0 \tag{18}$$

$$\sum_{l} x_l = W \tag{19}$$

where $Proc$ is the sets of all processors bound to the tasks (i.e., functions) of the LQN and $c = (c_p)_{p\in Proc}$ is the vector of processors' multiplicities; x is the vector of queue lengths at any location of the LQN model, with the queue length at generic location l being denoted by x_l; θ is the vector of LQN parameters, i.e., the service times and branch probabilities.

The objective function (17) expresses the goal of the optimization as a combination of two conflicting sub-objectives: the minimization of the average response time $\mathcal{R}(c,x,\theta)$ and the minimization of the allocated resources $\sum_{p\in Proc} c_p$ where $\mathcal{R}$ has a structure similar to the Equations (13)–(16), but applied to a generic LQN model.

Each sub-objective is scaled according to the weight $0{\leq}\psi{\leq}1$. Although tuning this weight has been extensively studied for multi-objective optimization [48], there is no universal choice as it heavily depends on the application. In our experimentation, we set $\psi = 0.5$.

Equations (18)–(19) encode the steady-state solution the fluid LQN model into the optimization problem. Specifically, Equation (18) enforces the steady state of the ODEs, while Equation (19) ensures that the total number of users in the model is equal to W. Also in this case, Equations (18) have a structure similar to the Equations (1)–(4), but applied to a more general LQN model.

Solving the optimization problem (17)–(19) yields two key pieces of information: c^*, representing the minimum number of cores to assign to each function to minimize its response time, and $\dot{x}^*$, the optimal steady-state solution for all quantities tracked by the fluid LQN. From these, similarly to (9)–(12), we can obtain $\mathcal{M}^* = (m_i)_{i\in\text{Task}}$ denoting the number of requests executing or waiting for potential nested calls on each Task of the LQN and consequently function of the FaaS system.

It is important to note that it is not possible to directly use $\mathcal{M}^*$ for determining the concurrency limit for each function, as these values depend on the number of users considered while solving the optimization problem and would therefore need to be recalculated each time. To overcome this issue, WasteLess relies on the ratio

$$b_i^* = \frac{m_i^*}{c_i^*} \quad \forall i \in Task \tag{20}$$

which is instead constant regardless of the number of functions' instances and the number of users interacting with the FaaS system.

❸ *WasteLess FaaS Configurator*

3a *CPU & Concurrency Configuration*

As a first step, WasteLess calculates the concurrency limit relying on Equation (20) that expresses the optimal balance between the number of actual concurrent requests of a function and the corresponding number of cores. For doing so, it considers c_i^* as the base core allocation of each function and chooses a concurrency limit equal to $\hat{m}_i = \lceil b_i^* c_i^* \rceil$, i.e., it rounds up to the nearest integer as the function concurrency limit can only be set to an integer value.

3b *Memory Configuration*

Central in this phase is the function Mem synthesized in point 1c of Section 4, and the concurrency limit $\hat{m}_i$ just calculated for each function. By leveraging these two quantities, we allocate to each function a memory level equal to $Mem(\hat{m}_i)$ so that it can support the presence of $\hat{m}_i$ concurrent requests without affecting performance.

V. Experimental Evaluation

In this section, we present a series of experiments evaluating the impact of WasteLess on the performance and cost of a representative FaaS application deployed on GCR using second-generation serverless functions.

WasteLess is implemented as a combination of Matlab and Python modules. For profiling infrastructure, we leveraged the standard GCR monitoring tools[7] to collect data from the executing system. Additionally, we relied on Google Cloud Logging[8] to track the CPU time consumption of individual functions. The whole experimental infrastructure and the associated resources are available at [47].

To conduct a comprehensive experimentation, we compare the effectiveness of configurations suggested by WasteLess against those produced by the vanilla GCR instance scheduler and ProPack [8]. As explained in Sec. II, ProPack is a recent

[7]https://cloud.google.com/monitoring
[8]https://cloud.google.com/logging

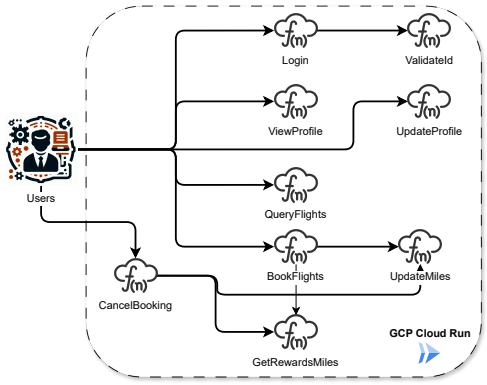

Fig. 6: Topology of AcmeAir's serverless porting

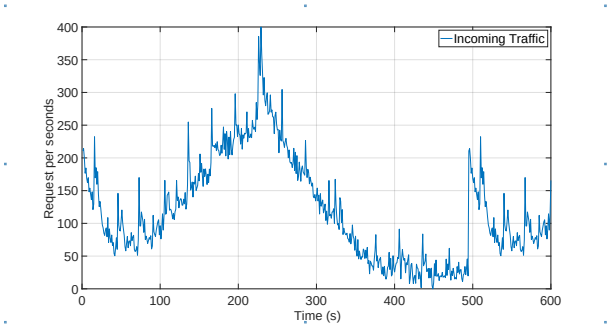

Fig. 7: Twitter traffic trace used in the experiments [50].

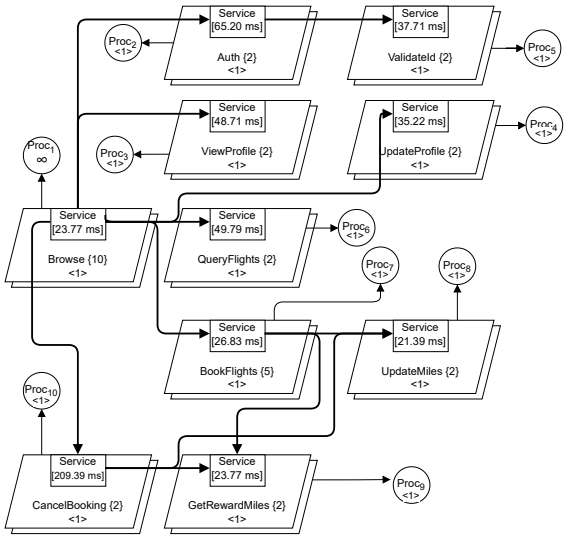

Fig. 8: LQN model of an analyzed variant of AcmeAir.

research approach that strives to improve the efficiency and cost-effectiveness of FaaS applications characterized by high levels of concurrency. Considering that ProPack is implemented on top of a custom prototype, here we integrated ProPack-suggested configurations with GCR following a similar strategy used to integrate WasteLess with GCR.

A. Experiment Setup

We consider *AcmeAir* [49], a well-known benchmark of microservice-based application ([10], [12], [21], [22]). We ported the original project to a FaaS implementation, which is available in the replication package [47].

The architecture is depicted in Figure 6. Each microservice is implemented as a serverless function in GCR, invoked via HTTP requests. The application is thus composed of nine functions providing the following endpoints to its users: *Auth*, *ValidateId*, *ViewProfile*, *UpdateProfile*, *QueryFlights*, *BookFlights*, *UpdateMiles*, *GetRewardMiles*, and *CancelBooking*. We replaced the original business logic of each function with a CPU-intensive load, whose resource demand can be easily configured. Thus, we could generate different AcmeAir variants, each characterized by different resource demands.

We generated 30 variants of AcmeAir, where the CPU demand of each function is uniformly sampled at random from the interval $[0.02, 3]$ (in s). By doing so, we generated applications with a mix of fast and slow functions to highlight the challenges and the potential benefits of concurrent function execution. For each variant, we also generated and calibrated the associated LQN model to be used by WasteLess.

For workload generation, we relied on the well-known load testing tool *Locust* considering a real traffic trace provided by Twitter (see, e.g., [50]). We downscaled this trace in the range from 0 to 400 requests per second without altering its shape. The resulting load shape is reported in Figure 7. Following the classic implementation of AcmeAir [10], [12], the logic for each user was chosen to solicit all the system functions.

We used the Google Cloud command-line interface, namely *gcloud*, to update the number of cores and memory as determined by WasteLess. Furthermore, gcloud is also used at runtime to set the concurrency limit as computed by WasteLess and ProPack when needed.

B. LQN Model Derivation

Figure 8 shows the LQN model of AcmeAir derived following point 1b in Section IV. Each function is assigned to an LQN task with a single entry corresponding to its HTTP endpoint. Function communications are modeled as synchronous calls between LQN entries, and the logic of the task *Client* represents user behavior. Since each AcmeAir variant differs in function execution speed, the service rates for each model entry vary accordingly. For brevity, Figure 8 presents the parameterization of a particular instance; detailed information on all variants is available in the replication package [47]. To validate each LQN model, we solved the corresponding fluid LQN using MATLAB and compared the results with the average steady-state end-to-end response time and throughput measured in the corresponding AcmeAir instance. Specifically, the application was tested under workloads ranging from 2 to 180 users, representing validation conditions distinct from the single-user workload used for model calibration (cf. point 1b in Section IV).

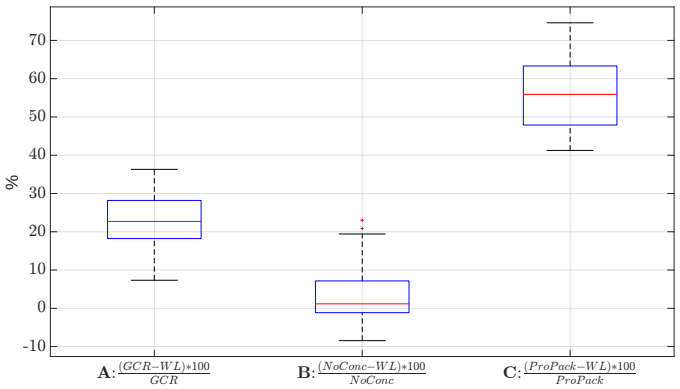

Fig. 9: Percentage difference between the average end-to-end response times measured when using the GCR instance scheduler (GCR) and WasteLess (WL) (boxplot **A**), with no-concurrency (NoConc) and WasteLess (boxplot **B**), and the one with ProPack and WasteLess (boxplot **C**).

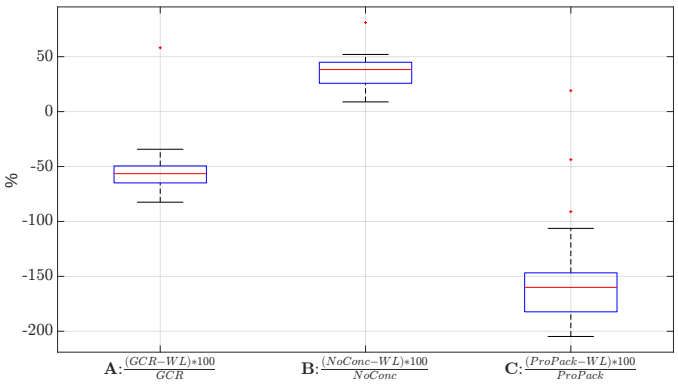

Fig. 10: Percentage difference between the billable instances measured when using the GCR concurrency tuner and WasteLess (**A**), the one measured with no-concurrency and WasteLess (**B**), and the one measured with ProPack generated concurrency and WasteLess (**C**).

For the LQN reported in Figure 8, model accuracy is high: the error on response time is less than 10% on average, while throughput prediction error is less than 5%. Comparable accuracy scores also apply to all other AcmeAir variants.

C. Results

We discuss the results of our experimentation specifically designed to address the following research questions:

RQ1 Do WasteLess computed configurations effectively minimize FaaS applications' average response time?
RQ2 Does the utilization of WasteLess result in a tangible decrease in operational costs on the cloud?

To answer these questions, we conducted a series of experiments, each involving one of the 30 variants of AcmeAir under four different scenarios of concurrency limit configuration: (i) no-concurrency; (ii) GCR instance scheduler with default concurrency limit (i.e., 80); (iii) WasteLess-suggested concurrency limit; (iv) ProPack-suggested concurrency limit. For a fair evaluation, in all four cases the functions were set up with the same CPU core and memory allocations, as suggested by WasteLess, since both GCR and ProPack do not offer methods for automatic CPU cores and memory configurations.

To assess research question **RQ1**, Figure 9 shows the statistics of three crucial metrics:

- **A**: the percentage difference between the average end-to-end response time measured when using the GCR instance scheduler with default concurrency limit, and when using WasteLess;
- **B**: the percentage difference between average end-to-end response time measured with no-concurrency, and when using WasteLess;
- **C**: the percentage difference between average end-to-end response time measured with ProPack-suggested concurrency limit, and when using WasteLess.

Our analysis shows that WasteLess achieves significantly lower end-to-end response times compared to the GCR instance scheduler and ProPack. Specifically, WasteLess reduces the average response time by 7% to 35% (median 23%) compared to GCR, and by 40% to 74% (median 55%) compared to ProPack. The Kolmogorov-Smirnov test [51] confirms the statistical significance of these differences. For GCR configurations versus WasteLess, the null hypothesis of equal response time distributions was rejected with a 95% confidence level and a p-value of 0.0113. Similarly, comparisons between ProPack-suggested configurations and WasteLess yielded a p-value of $5.5870e^{-8}$, demonstrating a high statistical significance.

Comparing the average end-to-end response times of WasteLess with those measured under no-concurrency, we observe percentage differences between approximately -8% and 20%, indicating that no-concurrency is not always the optimal choice performance-wise. The Kolmogorov-Smirnov test does not reject the hypothesis of equality for the average response time distributions obtained with WasteLess and the no-concurrency approach, with a 95% confidence level and a p-value of 0.9360, indicating that the difference between the two distributions is not statistically significant. This is a notable result, as WasteLess is the only approach among those evaluated that enables the use of concurrent functions without significant performance degradation compared to the no-concurrency scenario (i.e., the one with no contention).

These results demonstrate that the configurations suggested by WasteLess yield response times statistically indistinguishable from the no-concurrency case, thus positively answering to **RQ1**.

Figure 10 illustrates the percentage difference in total billable instances, and consequently the monetary cost, between configurations suggested by WasteLess and those obtained

TABLE II: Statistics of the concurrency limit computed by WasteLess and ProPack across all Acmeair variants. For the GCR instance scheduler with default parameter and no-concurrency these values are set to 80 and 1, respectively.

	50^{th}	95^{th}	99^{th}
WasteLess	2	19	145
ProPack	38	100	100

using: (**A**) the GCR instance scheduler, (**B**) no-concurrency, and (**C**) ProPack.

As expected, both ProPack and the GCR instance scheduler aggressively utilize concurrency limit, achieving the lowest expenses with median cost reductions of approximately $\sim 160\%$ and $\sim 56\%$ compared to WasteLess, respectively. In contrast, the no-concurrency configuration employs the highest number of function instances, resulting in significantly higher costs, around 40% more on average, both with statistical significance.

These results clearly show that WasteLess achieves significant savings compared to the no-concurrency scenario while maintaining comparable performance, thus positively answering **RQ2**.

Table II presents statistics on the concurrency limit values configured by WasteLess and ProPack during the experimental campaign, compared to the GCR default concurrency limit. The results show that in over 95% of the randomly generated application instances, WasteLess set concurrency limit no higher than 19, while ProPack configured significantly higher values. This highlights the suboptimality of both the GCR default and ProPack, explaining the severe response time degradation caused by excessive resource contention within function instances.

The results confirm that concurrent function execution can substantially reduce cloud costs by limiting the number of spawned instances, but it demands precise configuration to maintain performance. WasteLess excels in this regard, achieving a 40% cost reduction without compromising performance, unlike GCR's scheduler with default concurrency limit and ProPack.

VI. Threats to validity

LQN model and configurations optimality. The main threat to our work's validity is the potential inaccuracy of the LQN model, which does not currently capture all the involved system dynamics (e.g., cold starts, scaling latency). As WasteLess computes optimal configurations with respect to the derived LQN, we cannot claim in advance that they will remain optimal when applied to the real system. To mitigate this, we validated our methodology in a production-ready environment, GCR, showing that our model accurately predicts application performance under various conditions.

Overhead of WasteLess. The optimizations provided by WasteLess incur some additional computation for its execution. However, in our experiments, WasteLess's execution times were on the order of milliseconds, making them negligible compared to the overall application runtime. The most time-intensive task in our approach is function profiling, but WasteLess requires significantly fewer profiled samples compared to state-of-the-art alternatives like [8].

Single application with CPU-intensive workloads. Our validation leverages a specific topology (i.e., AcmeAir), a well-known case study with complex function interactions. To demonstrate how concurrency limit impacts the performance and cost of a FaaS system, we created 30 variants with different mixes of fast and slow functions to illustrate both optimal and suboptimal configurations. While the variants focus on CPU-intensive functions, we plan to extend the WasteLess model to account for memory and I/O resource contention in the future.

Single FaaS provider. WasteLess has been integrated and evaluated exclusively on GCR, chosen because, at the time of writing, it is the only major FaaS offering with native support for second generation severless functions. However, our approach is easily portable to other platforms, as the underlying LQN model is quite general. Additionally, the mechanism used by ProPack could be used to integrate WasteLess into FaaS platforms lacking built-in concurrency support.

Hybrid deployments. Our approach targets applications built entirely on the FaaS model. However, many real-world applications combine serverless functions with other technologies, such as containers. While our method may not directly apply to these hybrid systems as a whole, it could still be used to create individual predictive models for each function. We anticipate some sub-optimality in the results in such cases but leave a systematic investigation for future work.

VII. Conclusion

We presented WasteLess, a framework that optimizes CPU, memory, and concurrency limit configurations for serverless functions. By leveraging performance models, WasteLess identifies optimal settings to minimize response time at the lowest cost. Integrated with Google Cloud Run (GCR), WasteLess was evaluated against the default GCR configurations and those suggested by ProPack [8]. WasteLess matched the performance of the no-concurrency baseline while reducing costs by an average of 40%. In contrast, both the vanilla GCR scheduler and ProPack significantly impacted performance.

As future work, we aim to automate the performance modeling process by integrating tools that generate LQN models directly from functions' source code. We will also expand WasteLess to consider additional cost factors, such as invocation count, maximum runtime, and storage, to enhance its optimization capabilities.

Acknowledgment

Work partially supported by the project SERICS (SEcurity and RIghts in the CyberSpace), PE0000014, PNRR M4C2 I. 1.3 funded by European Union Next Generation EU

REFERENCES

[1] S. Kounev, N. Herbst, C. L. Abad, A. Iosup, I. Foster, P. Shenoy, O. Rana, and A. A. Chien, "Serverless computing: What it is, and what it is not?" *Communications of the ACM*, vol. 66, no. 9, pp. 80–92, 2023.

[2] L. Baresi, D. Y. X. Hu, G. Quattrocchi, and L. Terracciano, "NEPTUNE: network- and gpu-aware management of serverless functions at the edge," in *Proc. of Int'l Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS '22*. ACM/IEEE, 2022, pp. 144–155.

[3] A. Eivy and J. Weinman, "Be wary of the economics of" serverless" cloud computing," *IEEE Cloud Computing*, vol. 4, no. 2, pp. 6–12, 2017.

[4] F. Liu and Y. Niu, "Demystifying the cost of serverless computing: Towards a win-win deal," *IEEE Transactions on Parallel and Distributed Systems*, 2023.

[5] N. Akhtar, A. Raza, V. Ishakian, and I. Matta, "COSE: configuring serverless functions using statistical learning," in *Proc. of 39th IEEE Conference on Computer Communications, INFOCOM '20*, 2020, pp. 129–138.

[6] S. Eismann, L. Bui, J. Grohmann, C. L. Abad, N. Herbst, and S. Kounev, "Sizeless: predicting the optimal size of serverless functions," in *Middleware '21: 22nd International Middleware Conference, Québec City, Canada, December 6 - 10, 2021*, K. Zhang, A. Gherbi, N. Venkatasubramanian, and L. Veiga, Eds. ACM, 2021, pp. 248–259.

[7] G. Franks, T. Al-Omari, M. Woodside, O. Das, and S. Derisavi, "Enhanced modeling and solution of layered queueing networks," *IEEE Transactions on Software Engineering*, vol. 35, no. 2, pp. 148–161, 2008.

[8] R. B. Roy, T. Patel, R. Liew, Y. N. Babuji, R. Chard, and D. Tiwari, "Propack: Executing concurrent serverless functions faster and cheaper," in *Proc. of the 32nd International Symposium on High-Performance Parallel and Distributed Computing, HPDC '23*. ACM, 2023, pp. 211–224.

[9] G. Maddodi, S. Jansen, and M. Overeem, "Aggregate architecture simulation in event-sourcing applications using layered queuing networks," in *Proc. ACM/SPEC Int. Conf. Perf. Eng.*, 2020, pp. 238–245.

[10] T. Inagaki, Y. Ueda, M. Ohara, S. Choochotkaew, M. Amaral, S. Trent, T. Chiba, and Q. Zhang, "Detecting layered bottlenecks in microservices," in *IEEE 15th Int. Conf. Cloud Comput.*, 2022, pp. 385–396.

[11] G. Garbi, E. Incerto, and M. Tribastone, "μP: A development framework for predicting performance of microservices by design," in *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*. IEEE, 2023, pp. 178–188.

[12] E. Incerto, R. Pizziol, and M. Tribastone, "μOpt: An efficient optimal autoscaler for microservice applications," in *2023 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE, 2023, pp. 67–76.

[13] R. Wang, G. Casale, and A. Filieri, "Enhancing performance modeling of serverless functions via static analysis," in *Int. Conf. Service-Oriented Comput.* Springer, 2022, pp. 71–88.

[14] A. U. Gias, G. Casale, and M. Woodside, "Atom: Model-driven autoscaling for microservices," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019, pp. 1994–2004.

[15] M. Tribastone, "A fluid model for layered queueing networks," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 744–756, 2012.

[16] J. Ruuskanen, T. Berner, K.-E. Arzen, and A. Cervin, "Improving the mean-field fluid model of processor sharing queueing networks for dynamic performance models in cloud computing," *ACM SIGMETRICS Perf. Eval. Rev.*, vol. 49, no. 3, pp. 69–70, 2022.

[17] G. Garbi, E. Incerto, and M. Tribastone, "Learning queuing networks by recurrent neural networks," in *Proc. ACM/SPEC Int. Conf. Perf. Eng.*, 2020, pp. 56–66.

[18] L. Zhu, G. Casale, and I. Perez, "Fluid approximation of closed queueing networks with discriminatory processor sharing," *Perf. Eval.*, vol. 139, p. 102094, 2020.

[19] M. Woodside, "Heuristic derivation of a fluid model from a layered queueing network," in *Companion of ACM/SPEC Int. Conf. Perf. Eng.*, 2023, pp. 389–395.

[20] R. Wang, G. Casale, and A. Filieri, "Enhancing performance modeling of serverless functions via static analysis," in *Service-Oriented Computing*. Cham: Springer Nature Switzerland, 2022, pp. 71–88.

[21] C. M. Aderaldo, N. C. Mendonça, C. Pahl, and P. Jamshidi, "Benchmark requirements for microservices architecture research," in *IEEE/ACM 1st Int. Workshop Establishing Community-Wide Infrastructure Architecture-Based Softw. Eng.*, 2017, pp. 8–13.

[22] G. Blinowski, A. Ojdowska, and A. Przybyłek, "Monolithic vs. microservice architecture: A performance and scalability evaluation," *IEEE Access*, vol. 10, pp. 20 357–20 374, 2022.

[23] C. Lin and H. Khazaei, "Modeling and optimization of performance and cost of serverless applications," *IEEE Trans. Parallel Distributed Syst.*, vol. 32, no. 3, pp. 615–632, 2021.

[24] C. Lin, N. Mahmoudi, C. Fan, and H. Khazaei, "Fine-grained performance and cost modeling and optimization for faas applications," *IEEE Trans. Parallel Distributed Syst.*, vol. 34, no. 1, pp. 180–194, 2023.

[25] T. Elgamal, "Costless: Optimizing cost of serverless computing through function fusion and placement," in *Proc. of 2018 IEEE/ACM Symposium on Edge Computing, SEC '18*, 2018, pp. 300–312.

[26] A. Mahgoub, E. B. Yi, K. Shankar, S. Elnikety, S. Chaterji, and S. Bagchi, "ORION and the three rights: Sizing, bundling, and prewarming for serverless dags," in *Proc. of 16th USENIX Symposium on Operating Systems Design and Implementation, OSDI '22*, 2022, pp. 303–320.

[27] A. Mahgoub, E. B. Yi, K. Shankar, E. Minocha, S. Elnikety, S. Bagchi, and S. Chaterji, "WISEFUSE: workload characterization and DAG transformation for serverless workflows," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 6, no. 2, pp. 26:1–26:28, 2022.

[28] A. U. Gias and G. Casale, "COCOA: cold start aware capacity planning for function-as-a-service platforms," in *28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS 2020, Nice, France, November 17-19, 2020*. IEEE, 2020, pp. 1–8.

[29] H. Qian, Q. Wen, L. Sun, J. Gu, Q. Niu, and Z. Tang, "Robustscaler: Qos-aware autoscaling for complex workloads," in *Proc. of 38th IEEE International Conference on Data Engineering, ICDE '22*, 2022, pp. 2762–2775.

[30] X. Li, P. Kang, J. Molone, W. Wang, and P. Lama, "Kneescale: Efficient resource scaling for serverless computing at the edge," in *Proc. of 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing, CCGrid '22*, 2022, pp. 180–189.

[31] A. Das, A. Leaf, C. A. Varela, and S. Patterson, "Skedulix: Hybrid cloud scheduling for cost-efficient execution of serverless applications," in *Proc. of 13th IEEE Int'l Conference on Cloud Computing, CLOUD '20*, 2020, pp. 609–618.

[32] A. Tariq, A. Pahl, S. Nimmagadda, E. Rozner, and S. Lanka, "Sequoia: enabling quality-of-service in serverless computing," in *Proc. of ACM Symposium on Cloud Computing, SoCC '20*, 2020, pp. 311–327.

[33] G. R. Russo, A. Milani, S. Iannucci, and V. Cardellini, "Towards qos-aware function composition scheduling in apache openwhisk," in *Proc. of 1st Workshop on Serverless Computing for Pervasive Cloud-Edge-Device Systems and Services, *LESS '22*. IEEE, 2022.

[34] Z. Li, L. Guo, Q. Chen, J. Cheng, C. Xu, D. Zeng, Z. Song, T. Ma, Y. Yang, C. Li, and M. Guo, "Help rather than recycle: Alleviating cold startup in serverless computing through inter-function container sharing," in *Proc. of 2022 USENIX Annual Technical Conference, ATC '22*, 2022, pp. 69–84.

[35] A., S. Chang, H. Tian, H. Wang, H. Yang, H. Li, R. Du, and Y. Cheng, "Faasnet: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute," in *Proc. of 2021 USENIX Annual Technical Conference, ATC '21*, 2021, pp. 443–457.

[36] J. Shen, T. Yang, Y. Su, Y. Zhou, and M. R. Lyu, "Defuse: A dependency-guided function scheduler to mitigate cold starts on faas platforms," in *Proc. of 41st IEEE International Conference on Distributed Computing Systems, ICDCS '21*, 2021, pp. 194–204.

[37] P. Silva, D. Fireman, and T. E. Pereira, "Prebaking functions to warm the serverless cold start," in *Proc. of 21st Int'l Middleware Conference, Middleware '20*. ACM, 2020, pp. 1–13.

[38] N. Daw, U. Bellur, and P. Kulkarni, "Xanadu: Mitigating cascading cold starts in serverless function chain deployments," in *Proc. of 21st Int'l Middleware Conference, Middleware '20*. ACM, 2020, pp. 356–370.

[39] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "FIRM: an intelligent fine-grained resource management framework for slo-oriented microservices," in *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 2020, pp. 805–825.

[40] J. Liu, S. Zhang, and Q. Wang, "conadapter: Reinforcement learning-based fast concurrency adaptation for microservices in cloud," in *Proceedings of the 2023 ACM Symposium on Cloud Computing*, ser. SoCC '23, 2023, p. 427–442.

[41] M. Tribastone, P. Mayer, and M. Wirsing, "Performance prediction of service-oriented systems with layered queueing networks," in *Leveraging Appl. of Formal Methods, Verification, and Validation: 4th Int. Symp. Proc., Part II 4*. Springer, 2010, pp. 51–65.

[42] T. G. Kurtz, "Solutions of ordinary differential equations as limits of pure jump markov processes," *J. Appl. Probability*, vol. 7, no. 1, pp. 49–58, 1970.

[43] G. Casale, "Integrated performance evaluation of extended queueing network models with line," in *2020 Winter Simulation Conference (WSC)*. IEEE, 2020, pp. 2377–2388.

[44] V. Gupta, M. H. Balter, K. Sigman, and W. Whitt, "Analysis of join-the-shortest-queue routing for web server farms," *Performance Evaluation*, vol. 64, no. 9-12, pp. 1062–1081, 2007.

[45] J. D. Little and S. C. Graves, "Little's law," *Building intuition: insights from basic operations management models and principles*, pp. 81–100, 2008.

[46] G. S. Fishman and L. S. Yarberry, "An implementation of the batch means method," *INFORMS Journal on Computing*, vol. 9, no. 3, pp. 296–310, 1997.

[47] E. Incerto, R. Pizziol, G. R. Russo, and M. Tribastone, "WasteLess replication package," Oct. 2024. [Online]. Available: https://zenodo.org/doi/10.5281/zenodo.11519692

[48] R. T. Marler and J. S. Arora, "The weighted sum method for multi-objective optimization: new insights," *Structural and Multidisciplinary Optimization*, vol. 41, no. 6, pp. 853–862, 2010.

[49] D. Tollefson and A. Spyker, "AcmeAir: A Sample Cloud Native Application," https://github.com/AcmeAir/AcmeAir, 2015.

[50] G. Quattrocchi, E. Incerto, R. Pinciroli, C. Trubiani, and L. Baresi, "Autoscaling solutions for cloud applications under dynamic workloads," *IEEE Transactions on Services Computing*, pp. 1–17, 2024.

[51] V. W. Berger and Y. Zhou, "Kolmogorov–smirnov test: Overview," *Wiley statsref: Statistics reference online*, 2014.