# Flexecutor: Out-of-the-Box Smart Provisioning for Serverless Workflows

Enrique Molina-Giménez
enrique.molina@urv.cat
Universitat Rovira i Virgili
Tarragona, Spain

Daniel Barcelona-Pons
daniel.barcelona@urv.cat
Universitat Rovira i Virgili
Tarragona, Spain

Octavio H. Iacoponelli
octaviohoracio.iacoponelli@urv.cat
Universitat Rovira i Virgili
Tarragona, Spain

Pedro García-López
pedro.garcia@urv.cat
Universitat Rovira i Virgili
Tarragona, Spain

## Abstract

Serverless computing enables applications to scale dynamically without manual server management, with users paying only for the actual compute resources consumed. In the context of parallel workload execution, the problem of determining how many resources to provision and of what size has been extensively studied in recent years. However, existing approaches often present usability challenges, such as ad-hoc implementations or tight integration with a single cloud provider, which hinder the efficient deployment of new applications. In this work, we introduce Flexecutor, a Python library built on top of Lithops that unifies state-of-the-art serverless provisioners into a reusable and easy-to-use framework. Flexecutor abstracts away the underlying complexity, providing practitioners with a simple interface to deploy, manage, and optimize serverless workflows across multiple FaaS platforms. With Flexecutor, the deployment effort for coding serverless workflows is significantly reduced, while enabling the seamless integration of new smart provisioning strategies.

## CCS Concepts

• **Computer systems organization → Cloud computing**.

## Keywords

Serverless computing, Function-as-a-Service (FaaS), Serverless workflows, Resource provisioning, Parallel workloads, Scalability, Smart provisioning

**ACM Reference Format:**
Enrique Molina-Giménez, Daniel Barcelona-Pons, Octavio H. Iacoponelli, and Pedro García-López. 2025. Flexecutor: Out-of-the-Box Smart Provisioning for Serverless Workflows. In *11th International Workshop on Serverless Computing (WoSC11 '25), December 15–19, 2025, Nashville, TN, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3774899.3775013

## 1 Introduction

Serverless computing, and more specifically the Function-as-a-Service (FaaS) model, delivered to practitioners a new computational paradigm in which they could abstract away server management, delegating to the cloud provider the provisioning and operation of servers, used only for the time actually needed. The advantages of FaaS over traditional computing are several: (1) no infrastructure management, (2) instantaneous provisioning and very high elasticity [9], and (3) a pay-as-you-go model, where costs are incurred only for the actual compute time consumed.

FaaS services offered by cloud providers have been mostly adopted for web application deployment and event-based computations [23], typically in the form of short-lived activations. However, numerous efforts in the scientific literature (see Section 2), as well as some commercial solutions, leverage the computational power — and the very high scalability provided by FaaS— as a workhorse for the distributed execution of parallelizable workloads. In this work, we establish the term *serverless workflows* to refer to such tasks: parallel computations that exploit the elasticity of FaaS to launch massive-scale executions. The framework of reference for executing serverless workflows is Lithops [21], a Python-based parallel computing framework that offers a unified API to launch workloads in parallel across different FaaS providers: AWS Lambda, Azure Functions, GCP Functions and more.

Although the use of serverless computing for launching parallel workloads is highly attractive (scaling from zero resources to large-scale in very short times, and paying only for compute time) compared to the serverful approach [10], this does not imply that the serverless model is, by itself, close to optimal. Resource provisioning for the execution of serverless workflows has been extensively studied by the scientific community [1, 8, 13, 16, 19, 25–27]. While addressed in different ways, the common challenge remains: determining (1) the required number of functions and (2) the optimal function resource configuration according to the practitioner's objectives, which may involve minimizing cost, execution time, or a trade-off between both. To the best of our knowledge, all existing provisioning solutions for serverless workflows face accessibility and usability challenges: some are not open-source [1, 13, 13, 24, 25], which prevents direct usage, while others, although publicly accessible, are ad-hoc implementations that (1) integrate with only a single FaaS provider [8, 16, 26] and/or (2) are not released as reusable libraries for the implementation of new serverless workflows [16].

Flexecutor is the Python library we propose to overcome these limitations in a unified manner. Our solution builds upon Lithops to extend this popular framework with all the necessary directives to simplify workflow management [2, 12, 18] and support the end-to-end optimization of resource provisioning for serverless workflows, fully transparent to the underlying implementation. The Flexecutor API exposes all the required operations to cover the complete life-cycle of the provisioning problem. The library, abstract by design, currently integrates four state-of-the-art serverless smart provisioning solutions (Caerus [25], Ditto [13], Orion [16], and Jolteon [26]), while leaving the door open to seamless integration of future approaches.

The main contribution of this work is to provide practitioners with the necessary tool to optimize their serverless workflows. While until now the development and implementation of serverless workflows has been a complex, time-consuming, and error-prone process, our out-of-the-box library minimizes the efforts required to deploy, configure, and manage these workflows, providing a unified and reusable interface for state-of-the-art serverless smart provisioning solutions.

The structure of this paper is as follows. In Section 2, we review the background behind the solution. Section 3 focus on the usability frictions noticed in the field. Section 4 presents Flexecutor, the unified library that facilitates the usage of serverless smart provisioners. Section 5 introduces a set of test applications integrated into the library. Section 6 evaluates the performance of the integrated serverless smart provisioners, and finally, Section 7 concludes the paper.

## 2 Background

While most serverless usage targets short-lived, independent activations (HTTP requests, event-driven operations) [23], the use of multiple concurrent serverless functions as a computation engine for parallel workloads is relevant and has been thoroughly studied. Serverless workloads (hereafter referred to as *serverless workflows* for convenience) are Directed Acyclic Graphs (DAGs), where each node represents a computational stage composed of a set of concurrent serverless functions, typically operating over a dataset. Communication between stages can occur through different possible backends [4, 5, 15, 22], with object storage being the most common. Lithops [21] is a Python library that, in a cloud-agnostic manner, allows executing serverless workflows on the desired cloud provider with minimal configuration and coding overhead.

However, using Lithops to deploy serverless workflows does not end the practitioner's concerns. Once a serverless workflow is programmed, there are still variables to optimize (see Figure 1): how many functions should be provisioned for a computational stage (horizontal scaling), and what size should each function be (vertical scaling)? The answers to these questions are not unique and depend on the user's objective: *do I want my workflow to execute as fast as possible? Do I want to spend the least amount of money?* Or *do I prefer a trade-off between both?* In short, depending on the user's goal, there exists a serverless resource optimization problem that has been extensively addressed in the scientific literature [1, 8, 13, 16, 19, 24–26]. Commonly, smart provisioners identify the Pareto-optimal curve in the execution time vs. cost plane. A brief technical overview of a subset of smart provisioners is presented in Table 1.

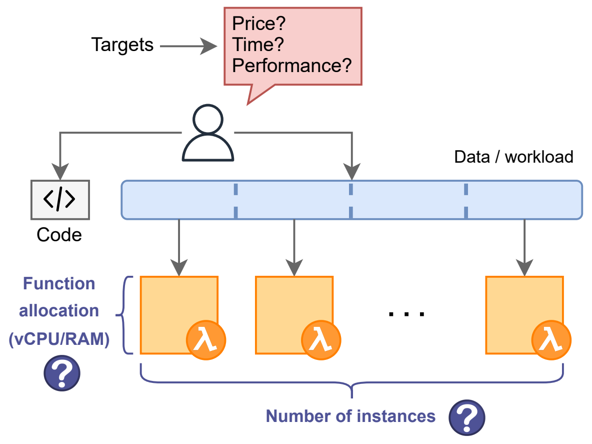

**Figure 1: Visualising the serverless provisioning challenge.**

| Impl. | Description |
|---|---|
| Caerus | Based on the NIMBLE algorithm, it schedules tasks in serverless DAGs at the right moment to minimize cost and completion time. It models fine-grained dependencies between pipelineable and non-pipelineable steps. |
| Ditto | By decoupling parallelism configuration from function placement, the system introduces a new scheduling granularity—"stage groups"—to optimize job completion time and cost. |
| Orion | Represents function latency as distributions rather than single values and combines them with convolution (series) and maximum (parallel). It considers correlations between functions to accurately estimate end-to-end latency in serverless DAGs. |
| Jolteon | Combines white-box and black-box modeling into a stochastic model for serverless workflows, converting the uncertain optimization problem into a deterministic one using Monte Carlo sampling. It finds optimal resource configurations via gradient descent thanks to convexity. |

**Table 1: Summary of some smart provisioners for serverless workflows.**

However, the scope of this work is not to delve into the technical implementation of serverless smart provisioning. The main focus is, after identifying common abstract operations inherent to smart provisioning (see Section 3), to provide a single, easily usable solution for practitioners seeking to optimize their serverless workflows.

## 3 Dissecting the usability problem

**Uncovered gaps in smart provisioning.** Existing serverless smart provisioning frameworks present significant usability challenges, as summarized in Table 2. Most of these tools are research prototypes: they are described and evaluated in papers, but no publicly-accessible code is provided [1, 13, 19, 24, 25]. This makes

| | Open source | Cloud agnostic | DAG mgmt. | Smart provisioning API |
|---|---|---|---|---|
| Caerus [25] | ✗* | - | - | - |
| Ditto [13] | ✗* | - | - | - |
| Locus [19] | ✗ | - | - | - |
| Cose [1] | ✗ | - | - | - |
| Stepconf [24] | ✗ | - | - | - |
| Aquatope [27] | ✓ | ✗ | ✓ | ✓ |
| Orion [16] | ✓* | ✗ | ✗ | ✗ |
| Jolteon [26] | ✓ | ✗ | ✓ | Partially |
| PowerTuning [8] | ✗ | ✗ | ✗ | ✓ |
| Flexecutor | ✓ | ✓ | ✓ | ✓ |

**Re-implemented by Zhang et al. [26]*

**Table 2: Usability and feature coverage of state-of-the-art smart provisioning solutions.**

it impossible to reuse the existing solutions directly, leaving re-implementation as the only option—a process that is often infeasible due to technical and temporal complexity.

Even when source code is available, additional limitations persist. Many tools are tightly coupled to a specific cloud provider [8, 16, 26, 27]; others do not allow native DAG declaration and orchestration [8, 16]; and some offer ad-hoc APIs that do not expose operations aligned with standard smart provisioning practices [16].

These combined limitations motivated the development of Flexecutor, which aims to eliminate the usability barriers present in current open-source smart provisioning solutions.

**Smart provisioning operations must be abstracted.** Across the surveyed tools, a common high-level pattern emerges: the operations available to users are largely shared across different implementations. In other words, despite the underlying technical differences, the systems are composed of the same fundamental operations. Flexecutor abstracts these operations into a single unified interface, which includes:

- *Profile*: Executes the serverless workflow under different user-defined resource configurations and collects execution metrics.
- *Train*: Uses the recorded metrics to build a predictive model (via an implementation-specific learning algorithm) that estimates execution time and cost for a given resource allocation.
- *Predict*: Infers execution metrics from the model for any specified resource configuration.
- *Optimize*: Searches for the optimal configuration according to the user-selected objective, determining the best resource allocation for all stages of the DAG.
- *Execute*: Runs the DAG under the chosen configuration. This operation can be triggered automatically by the system (after *Profile* or *Optimize*) or explicitly by the user.

In summary, prior to this work, available serverless smart provisioning solutions were fragmented in structure and lacking usability, creating high barriers for practical adoption. Flexecutor consolidates these diverse solutions into a single, unified framework, addressing the usability gap and providing a consistent, high-level interface for all core smart provisioning operations.

## 4 Flexecutor: the out-of-the-box solution

As discussed in previous sections, Flexecutor emerges as a response to the usability frictions identified in existing smart provisioners. The solution is implemented as a Python library of approximately 5.5k SLoC, providing essentially a client-side orchestration system for serverless DAGs, native integration with the Lithops compute engine, comprehensive support for smart provisioning, and an extensible design aimed at incorporating new provisioners. To structure the tool description, this section presents it from a structural perspective, the following section from a functional perspective, and finally we conclude with an example illustrating how to program serverless workflows over Flexecutor.

### 4.1 Scaffolding the DAG

The architecture of Flexecutor is based on several Python classes that encapsulate the data structures required for orchestrating and smart provisioning serverless workflows. Figure 2 depicts the key entities of the system. The central piece is the `DAG` entity, which constitutes a Directed Acyclic Graph organizing the different stages that compose a workflow. Each `Stage` is defined by the function code to be executed, called `fn_code`, and by two collections that group its `inputs` and `outputs`. These collections are represented through the `FlexData` entity, which acts as a logical abstraction over objects stored in object storage and their distribution policy (one-to-one, scatter, broadcast, on-the-fly partitioning [3], etc.).

Data access, which in serverless workflows involves constant exchanges of objects between storage and FaaS functions, is thus automated and hidden from the user. To achieve this, each stage has an execution context (`StageContext`), through which the `fn_code` retrieves input paths and defines output paths. The user only needs to implement a function that accepts one parameter which type is `StageContext` and, via using operations `get_input_paths` and `next_output_path`, can read from and write to the storage system. Once this `fn_code` is defined, it is associated with the corresponding stage, and the stage becomes part of the `DAG`.

The full `DAG` is constructed incrementally. After instantiating the initial graph, the various stages are added, and finally, the dependencies that determine execution order are declared. Flexecutor adopts a syntax inspired by Apache Airflow [6], which facilitates understanding and reduces the learning curve for users familiar with workflow orchestration systems. In this way, the user declaratively defines the graph and associated functions, while Flexecutor automatically manages both data transfers and workflow execution on Lithops.

### 4.2 Enabling Smart Provisioning

Once the DAG has been built, the next step is to provide the operations required for serverless smart provisioning. In Section 3, we reviewed the common actions performed by existing smart provisioners—namely `profile`, `train`, `predict`, `optimize`, and `execute`. The `DAGExecutor` class exposes these five methods directly to the programmer, providing a one-to-one correspondence.

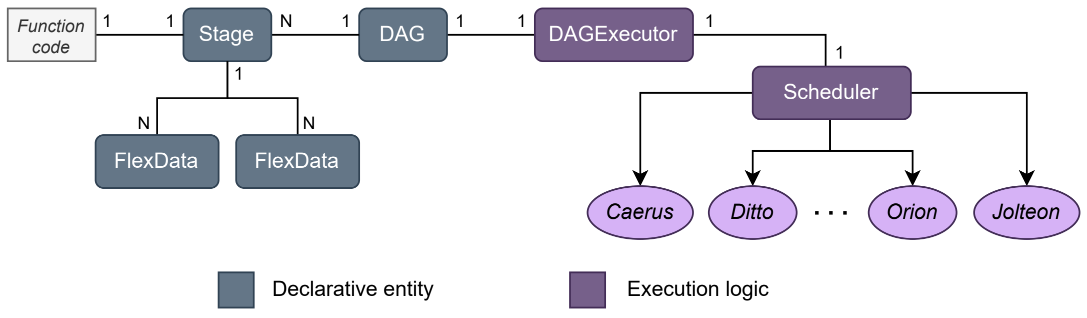

**Figure 2: High-level diagram representing class structure in Flexecutor.**

Figure 3 helps illustrate the different smart provisioning directives and their lifecycle during the provisioning process.

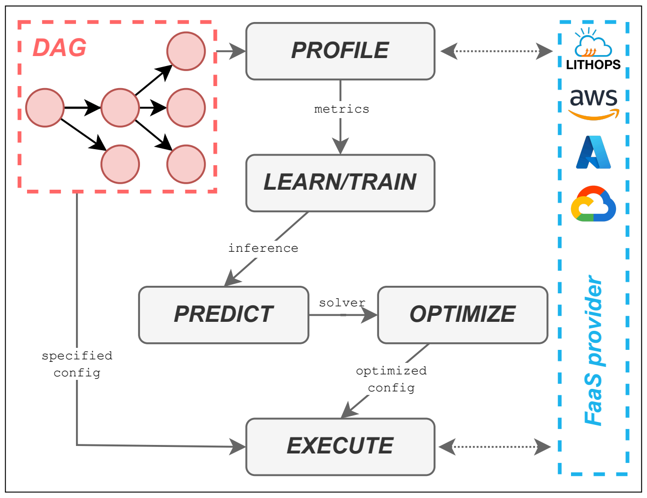

**Figure 3: Flexecutor operative lifeycle.**

Some of these methods, such as `profile` and `execute`, are common to all provisioners, as they merely execute the serverless workflow on the target service (and collect metrics). To enable this functionality, `DAGExecutor` internally wraps Lithops, mapping each stage of the workflow to a Lithops `map()` operation while orchestrating the DAG.

Other operations, namely `train`, `predict`, and `optimize`, are specific to the selected smart provisioner (`Scheduler`). The user must indicate which `Scheduler` to associate with `DAGExecutor` (choosing among four options: `Caerus`, `Ditto`, `Orion`, or `Jolteon`) and provide the `Scheduler` hyperparameters, such as optimization targets (minimum cost, maximum performance) and any scheduler-specific parameters.

Once configured, these operations are executed transparently to the user. This seamless behavior is made possible by the abstract `Scheduler` structure, which not only ensures consistent execution across different provisioners but also allows future integrations with new smart provisioners without requiring changes to the core system.

```python
if __name__ == "__main__":

    # Parallel function definition
    def word_count(ctx: StageContext):
        txt_paths = ctx.get_input_paths("input")
        for input in inputs:
            with open(input, "r") as f:
                content = f.read()
            count = len(content.split())
            count_path = ctx.next_output_path("output")
            with open(count_path, "w") as f:
                f.write(str(count))

    @flexorchestrator()
    def main():
        dag = DAG("hellodag")
        # Configure the DAG stages
        stage1 = Stage(
            "map",
            func=word_count,
            inputs=FlexData(prefix="input"),
            outputs=FlexData(prefix="output"),
        )
        stage2 = Stage(...)
        # DAG dependency definition
        stage1 >> stage2
        dag.add_stages([stage1, stage2])
        # Instance executor and perform operations
        executor = DAGExecutor(dag, Jolteon(...))
        executor.profile([ConfigSpace(...), ...])
        executor.train()
        executor.optimize()
        executor.shutdown()

    main()
```

**Listing 1: Example DAG: counting words.**

## 4.3 Example: Word Counter App

To illustrate the use of Flexecutor, we present a simple serverless workflow that counts the number of words in a set of text files (Listing 1). The workflow is organized as a `DAG` with multiple `Stage` (map-reduce word counter), although for brevity we only reference the first stage.

The first stage processes input `.txt` files and produces output files containing the word counts, transparently using the `FlexData` abstraction. `word_count` is the function that iterates over the input files, reads their content, calculates the number of words, and writes the result to the corresponding output path using `get_input_paths` and `next_output_path` methods of `StageContext`. Once declared, the `Stage` is added to the `DAG`.

Once the DAG is fully defined, an instance of `DAGExecutor` takes care of its execution. It receives the instance of the `Scheduler` specified by the user (`Jolteon` in this case). With this setup, the smart provisioning operations (`profile`, `train`, `optimize`, etc.) are ready to be executed.

This example demonstrates the key features of Flexecutor: declarative DAG construction, automatic data handling, seamless integration with Lithops, and transparent orchestration of smart provisioner operations. Although simple, it is representative of the framework's ability to structure and optimize the execution of serverless workflows.

## 5 Example workloads

Flexecutor library is not only delivered as a solution to the usability issues discussed in this work, but it also comes with a set of example applications provided for learning, development, and testing purposes. Below, we give a brief overview of the mission and structure of each example serverless workflow:

***Titanic app:*** its particularity is that it is a monostage app. An arbitrary number of parallel functions train a random forest classifier on the popular Titanic dataset [14]. The goal of this training is to predict the survival of the ship's passengers.

***Video app:*** based on Pocket [15], it is responsible for splitting videos, extracting frames, preprocessing them, and classifying them in four different stages. As a particular feature, some frames are preprocessed before being analyzed by the YOLO [20] model, while others are sent directly to classification. Each run processes one-minute videos in the Music and News categories, covering a mix of content for evaluation.

***Machine learning app:*** This ML workflow, inspired by Cirrus [7], operates through four consecutive stages: reducing dimensionality with PCA, training models, merging them, and testing. The training phase leverages the LightGBM library to run several parallel processes that generate multiple decision trees, which are later merged into a single random forest.

***Radio interferometry app:*** This large-scale serverless workflow use case, developed within the EXTRACT EU project [11], handles large volumes of radio astronomy data from NenuFar telescopes. Featured as a showcase in Flexecutor, it executes rebinning of Measurement Set (MS) files [17], precise signal calibration, and sophisticated subtraction to eliminate noise and interference. Once calibrated, the pipeline produces high-fidelity images of the observed sky regions.

## 6 Evaluation

**Smart provisioners integration.** Flexecutor integrates four smart provisioners—Caerus, Ditto, Orion, and Jolteon—into a unified framework. While a direct comparison of code complexity before (*without Flexecutor*) versus after (*with Flexecutor*) may seem tempting, it is important to keep in mind that the original baselines were released as research prototypes. They provide the heuristics needed for provisioning optimization according to each method, but were never intended as ready-to-use tools.

Flexecutor takes care of the underlying framework code, saving users from implementing hundreds of lines of low-level logic. At repository checkout `34f755d`, the difference is clear: Caerus is 261 SLoC, Ditto 476 SLoC, Orion 508 SLoC, and Jolteon 855 SLoC. This abstraction drastically lowers the implementation effort while still giving access to state-of-the-art provisioning strategies.

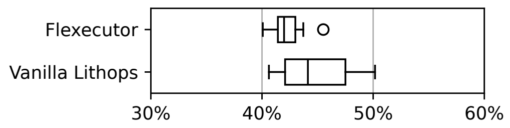

**Figure 4: Orchestation time (%) respecting total execution time in video app.**

We also tested Flexecutor with video processing and machine learning workloads (see Section 5). Using the same setup described for Jolteon in Section 6.1 of Zhang et al. [26], we reproduced the results shown in Figures 9b and 9c of the same work with no meaningful differences, showing that Flexecutor preserves the performance of the original smart provisioners while making them much easier to use.

**DAG orchestration and data management systems.** Flexecutor builds on top of Lithops to provide a system that enables both (1) seamless DAG declaration, in the style of Apache Airflow [6], and (2) explicit data dependency management, with `FlexData` inputs and outputs defined at each stage of the serverless workflow. Vanilla Lithops offers none of these abstractions, leaving users to implement them manually. To quantify the impact of Flexecutor, we evaluate both the simplification it provides and any runtime overhead introduced in DAG orchestration.

| Application | Lithops SLoC | Flexecutor SLoC | SLoC savings (%) |
|---|---|---|---|
| ML app | 322 | 278 | 13.66% |
| Titanic app | 80 | 63 | 21.25% |
| Video app | 247 | 180 | 27.13% |

**Table 3: SLoC savings and runtime overhead (Flexecutor vs vanilla Lithops) for different applications.**

In Table 3, we show the reduction in SLoC achieved with Flexecutor, with savings ranging from 13.66% to 27.13% across the applications presented. Regarding the evaluation of DAG orchestration performance, Figure 4 compares vanilla Lithops (without DAG awareness) with Flexecutor, based on 10 runs of the video application for each approach. On the X-axis, we show the orchestration time (i.e., the total execution time minus the periods in which the client is blocked by FaaS invocations). We observe that introducing the DAG orchestration system does not impose any overhead on application performance.

## 7 Conclusions

Throughout this work, we have shown that resource provisioning for serverless workflows has been a long-standing challenge, addressed by a wide range of heuristics and optimization strategies in the literature. While these approaches offer valuable techniques for fine-tuning resource allocation, they often remain narrowly focused on the complexity of the provisioning problem itself. Crucially, they tend to overlook the need for a fully-fledged solution that embraces the entire lifecycle of serverless workflows — from coding and orchestration to execution and optimization.

With Flexecutor, we address these limitations by introducing an open-source, cloud-agnostic framework that brings together all the core operations required to work with serverless workflows. Flexecutor not only abstracts away smart provisioning but also provides end-to-end orchestration capabilities, empowering developers and researchers alike to experiment, deploy, and scale workflows with minimal friction.

Code is publicly available at https://github.com/CLOUDLAB-URV/flexecutor. We actively encourage the community to both use the tool and extend it with new features. Thanks to its modular and extensible design, Flexecutor paves the way towards future integrations with additional function schedulers and advanced provisioning strategies, always with the ultimate goal of lowering the barriers to working effectively with serverless workflows.

In this sense, Flexecutor represents not just another optimization method, but a comprehensive, extensible, and practical step forward — bridging the gap between theoretical approaches and real-world deployments.

## Acknowledgments

We are grateful to the WoSC11 reviewers for their careful reading and valuable suggestions. This work has been partially funded by the European Union through Horizon Europe NEARDATA (101092644), CLOUDSTARS (101086248), EXTRACT (101093110) projects and by the Spanish Ministry of Economic Affairs and Digital Transformation and the European Union-NextGenerationEU (frameworks PRTR and the MRR), through the CLOUDLESS UNICO I+D CLOUD 2022 project.

## References

[1] Nabeel Akhtar, Ali Raza, Vatche Ishakian, and Ibrahim Matta. 2020. COSE: Configuring Serverless Functions using Statistical Learning. In *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*. 129–138. doi:10.1109/INFOCOM41043.2020.9155363

[2] Amazon Web Services. 2025. AWS Step Functions. https://docs.aws.amazon.com/step-functions/. Accessed: 2025-07-29.

[3] Aitor Arjona, Pedro García-López, and Daniel Barcelona-Pons. 2024. Dataplug: Unlocking extreme data analytics with on-the-fly dynamic partitioning of unstructured data. In *2024 IEEE 24th International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. 567–576. doi:10.1109/CCGrid59990.2024.00069

[4] Daniel Barcelona-Pons, Aitor Arjona, Pedro García-López, Enrique Molina-Giménez, and Stepan Klymonchuk. 2024. FaaS Is Not Enough: Serverless Handling of Burst-Parallel Jobs. arXiv:2407.14331 [cs.DC] https://arxiv.org/abs/2407.14331

[5] Daniel Barcelona-Pons, Pierre Sutra, Marc Sánchez-Artigas, Gerard París, and Pedro García-López. 2022. Stateful Serverless Computing with Crucial. *ACM Trans. Softw. Eng. Methodol.* 31, 3, Article 39 (March 2022), 38 pages. doi:10.1145/3490386

[6] Maxime Beauchemin et al. 2015. Apache Airflow: A platform to programmatically author, schedule, and monitor workflows. https://airflow.apache.org.

[7] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. 2019. Cirrus: a Serverless Framework for End-to-end ML Workflows. In *Proceedings of the ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 13–24. doi:10.1145/3357223.3362711

[8] Alex Casalboni. 2024. aws-lambda-power-tuning. https://github.com/alexcasalboni/aws-lambda-power-tuning. Accessed: 2025-07-29.

[9] Gerard Finol, Gerard París, Pedro García López, and Marc Sánchez Artigas. 2024. Exploiting inherent elasticity of serverless in algorithms with unbalanced and irregular workloads. *J. Parallel Distributed Comput.* 190 (2024), 104891. doi:10.1016/J.JPDC.2024.104891

[10] Pedro García-López, Marc Sánchez-Artigas, Simon Shillaker, Peter Pietzuch, David Breitgand, Gil Vernik, Pierre Sutra, Tristan Tarrant, Ana Juan-Ferrer, and Gerard París. 2022. *Trade-Offs and Challenges of Serverless Data Analytics*. Springer International Publishing, Cham, 41–61. doi:10.1007/978-3-030-78307-5_3

[11] Janine Gehrig. 2023. Horizon Europe project EXTRACT kicks off with a holistic approach to extreme data across the compute continuum. doi:10.5281/zenodo.8101639 Press release announcing the launch of the EU-funded EXTRACT project.

[12] Google Cloud. 2025. Google Cloud Workflows. https://cloud.google.com/workflows. Accessed: 2025-07-29.

[13] Chao Jin, Zili Zhang, Xingyu Xiang, Songyun Zou, Gang Huang, Xuanzhe Liu, and Xin Jin. 2023. Ditto: Efficient Serverless Analytics with Elastic Parallelism. In *Proceedings of the ACM SIGCOMM 2023 Conference* (New York, NY, USA) *(ACM SIGCOMM '23)*. Association for Computing Machinery, New York, NY, USA, 406–419. doi:10.1145/3603269.3604816

[14] Kaggle. 2012. Titanic: Machine Learning from Disaster. https://www.kaggle.com/c/titanic. Accessed: 2025-09-12.

[15] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 427–444. https://www.usenix.org/conference/osdi18/presentation/klimovic

[16] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. 2022. ORION and the Three Rights: Sizing, Bundling, and Prewarming for Serverless DAGs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 303–320. https://www.usenix.org/conference/osdi22/presentation/mahgoub

[17] J. P. McMullin, B. Waters, D. Schiebel, W. Young, and K. Golap. 2007. *CASA Architecture and Applications*. https://casa.nrao.edu/ Astronomical Data Analysis Software and Systems XVI, ASP Conference Series, Vol. 376, p.127.

[18] Microsoft Corporation. 2025. Azure Logic Apps. https://azure.microsoft.com/en-us/products/logic-apps. Accessed: 2025-07-29.

[19] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 193–206. https://www.usenix.org/conference/nsdi19/presentation/pu

[20] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. You Only Look Once: Unified, Real-Time Object Detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 779–788. doi:10.1109/CVPR.2016.91

[21] Josep Sampé, Marc Sánchez-Artigas, Gil Vernik, Ido Yehekzel, and Pedro García-López. 2023. Outsourcing Data Processing Jobs With Lithops. *IEEE Transactions on Cloud Computing* 11, 1 (2023), 1026–1037. doi:10.1109/TCC.2021.3129000

[22] Marc Sánchez-Artigas, Germán T. Eizaguirre, Gil Vernik, Lachlan Stuart, and Pedro García-López. 2020. Primula: a Practical Shuffle/Sort Operator for Serverless Computing. In *Proceedings of the 21st International Middleware Conference Industrial Track* (Delft, Netherlands) *(Middleware '20)*. Association for Computing Machinery, New York, NY, USA, 31–37. doi:10.1145/3429357.3430522

[23] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the wild: characterizing and optimizing the serverless workload at a large cloud provider. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC'20)*. USENIX Association, USA, Article 14, 14 pages.

[24] Zhaojie Wen, Yishuo Wang, and Fangming Liu. 2022. StepConf: SLO-Aware Dynamic Resource Configuration for Serverless Function Workflows. In *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications*. 1868–1877. doi:10.1109/INFOCOM48880.2022.9796962

[25] Hong Zhang, Yupeng Tang, Anurag Khandelwal, Jingrong Chen, and Ion Stoica. 2021. Caerus: NIMBLE Task Scheduling for Serverless Analytics. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 653–669. https://www.usenix.org/conference/nsdi21/presentation/zhang-hong

[26] Zili Zhang, Chao Jin, and Xin Jin. 2024. Jolteon: Unleashing the Promise of Serverless for Serverless Workflows. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 167–183. https://www.usenix.org/conference/nsdi24/presentation/zhang-zili-jolteon

[27] Zhuangzhuang Zhou, Yanqi Zhang, and Christina Delimitrou. 2022. AQUATOPE: QoS-and-Uncertainty-Aware Resource Management for Multi-stage Serverless Workflows. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 1–14. doi:10.1145/3567955.3567960