

Advancing Serverless Workflow Efficiency: Integrating Functional Programming Constructs and DAG-Based Execution

Nimród Földvári^a and Florin Crăciun^b

Faculty of Mathematics and Computer Science, Babes-Bolyai University, Cluj-Napoca, Romania
{harold.foldvari, mihai.craciun}@ubbcluj.ro

Keywords: Serverless Computing, Workflow Optimization, Directed Acyclic Graphs, Partial Execution, Function Orchestration, Cold Start.

Abstract: Serverless computing, also known as the Function-as-a-Service (FaaS) paradigm, has become a cornerstone of modern cloud-based applications, enabling developers to build and execute workflows by composing serverless functions. However, current serverless platforms are limited by constrained orchestration and function composition capabilities, which reduce their expressiveness and performance. To address these limitations, this paper introduces three enhancements to Apache OpenWhisk: native support for currying, continuation, and Directed Acyclic Graphs (DAGs). These enhancements are designed to improve the expressiveness of serverless workflows, reduce latency, and enable partial execution of dynamic workflows as soon as data becomes available.

1 INTRODUCTION

Serverless computing has become a popular paradigm for building and deploying native cloud applications. With serverless, developers can focus solely on writing code without worrying about provisioning or managing servers (Jonas et al., 2019). Cloud providers such as AWS Lambda, Azure Functions, and Google Cloud Functions handle all the infrastructure management and scaling automatically.

One of the key advantages of serverless computing is its cost effectiveness. Users only pay for the computation time their code is running, down to the millisecond. However, serverless architecture also faces notable performance bottlenecks, the *cold start* problem being the most significant.

When a serverless function is invoked for the first time after a period of inactivity, or when the cloud provider scales out new instances, there is a delay in launching the execution environment. This delay, known as a *cold start*, can range from a few milliseconds to several seconds, depending on various factors such as the language runtime, function package size, and the chosen cloud provider platform (Lin and Glikson, 2019).


The cold start latency can become a significant


performance bottleneck in complex serverless workflows involving multiple chained functions (Shahrad et al., 2020). If a workflow consists of multiple functions, each experiencing a *cold start*, the cumulative latency can be substantial, leading to a poor user experience (Jonas et al., 2019). Equation (1) shows the total duration of such a serverless workflow, where n is the number of functions in the workflow, Cold Start_i is the cold start latency for the i^{th} function, and Exec Time_i is the execution time of the i^{th} function.

$$T_{\text{workflow duration}} = \sum_{i=1}^n \text{Cold Start}_i + \sum_{i=1}^n \text{Exec Time}_i \quad (1)$$

Addressing the cold start problem is crucial for ensuring reliable and efficient serverless workflows, as it can significantly impact the overall performance and user experience of serverless applications. It is essential to find other orchestration possibilities beyond the existing solutions, with the aim of improving serverless workflows and reducing latency time.

We present an expressive orchestration language for controlling function scheduling and composition. To maximize scheduling optimization opportunities, we propose that this language be interpreted directly by OpenWhisk's scheduler. Additionally, we introduce a function composition style inspired by concepts such as currying and continuation — functional

^a  <https://orcid.org/0009-0008-3531-0478>

^b  <https://orcid.org/0000-0003-0335-8369>

constructs designed to be intuitive and familiar to developers.

The rest of this paper is organized into the following sections. In 2, we discuss challenges in serverless workflow orchestration, other approaches to orchestration and function composition, and present our motivation for enhancement in Apache OpenWhisk. In 3 we describe our enhancements, explaining the necessity of their introduction and illustrating them with examples. In 4, we present how our approach fulfills the serverless trilemma (Baldini et al., 2017b) and a preliminary evaluation of our prototype, demonstrating its benefits and performance improvements. In 5, we discuss limitations and future research directions, and finally conclude in 6.

2 BACKGROUND AND RELATED WORK

Serverless computing has gained traction as a paradigm that abstracts infrastructure management, allowing developers to focus on application logic. However, existing serverless platforms face limitations in workflow orchestration, particularly in modularity, latency optimization, and managing complex dependencies. Addressing these shortcomings requires introducing features such as **currying, continuation, and Directed Acyclic Graph (DAG) execution**. The DAG should be described using a domain-specific language (DSL) that allows a programmer to define the relationships between the inputs and outputs of multiple serverless functions in a workflow. This DSL would be interpreted directly by the serverless platform. Additionally, native support for currying and continuation is also lacking in current solutions. This section explores relevant works and their limitations, highlighting the need for these enhancements.

2.1 Challenges in Serverless Workflow Orchestration

Traditional serverless platforms primarily rely on **control-flow orchestration**, where workflows are executed sequentially based on predefined dependencies (Baldini et al., 2017b). While straightforward, this approach struggles with complex workflows and lacks constructs for modularization, reuse, and dynamic task suspension and resumption, leading to inefficiencies in parallel task execution and increased latency.

2.2 Advancements in Serverless Orchestration

Recent research efforts, such as **DataFlower**, **Pheromone**, **Triggerflow**, **DFlow**, **DAGit**, and **Wukong**, have introduced innovative paradigms to overcome the limitations of traditional serverless orchestration. However, these approaches exhibit trade-offs in expressiveness, dependency management, and workflow composition, particularly in their support for DAG execution, currying, continuation, and language flexibility—key features of our proposed solution. Below, we analyze key platforms and frameworks, highlighting their strengths and limitations in relation to our proposed enhancements (DAGs, currying, and continuation). Table 1 summarizes this comparison.

DataFlower (Li et al., 2024) adopts a decentralized data-flow approach, where workflows are represented as dynamic data-flow graphs. It decouples computation and communication using **Function Logic Units (FLUs)** and **Data Logic Units (DLUs)**, enabling overlapping computation and data transfer to improve throughput. This model is particularly effective for large-scale workflows, where overlapping operations can significantly reduce latency. However, while DataFlower’s decentralized architecture improves parallelism, it introduces complexity due to the need for separate Function Logic Units (FLUs) and Data Logic Units (DLUs), which require explicit coordination between computation and communication components. Notably, it lacks native support for functional constructs like currying, limiting incremental processing of partial data.

Pheromone (Yu et al., 2022), on the other hand, introduces a data-centric approach by using *data buckets* to trigger functions based on data availability. This paradigm simplifies orchestration and reduces latency through shared-memory object stores for rapid data exchange. However, while Pheromone significantly enhances performance with its data-centric design, it currently supports only C++ functions, limiting language flexibility, and its optimization for data-centric workflows may reduce effectiveness for purely function-driven applications. Continuation mechanisms are absent, forcing workflows to remain active during idle periods.

Triggerflow (García-López et al., 2020) presents an extensible trigger-based orchestration architecture for serverless workflows. It enables the construction of reactive orchestrators like state machines and directed acyclic graphs (DAGs). Triggerflow focuses on event-driven mechanisms and enhances orchestration efficiency by allowing reactive triggers to optimize la-

Table 1: Feature comparison of serverless orchestration platforms.

Platform	DAG Support	Currying	Continuation	Language Flexibility	Mitigates Double Spending
DataFlower	Partial	No	No	Medium	Partial
Pheromone	No	No	No	Low (C++ only)	No
Triggerflow	Yes	No	No	High	Partial
DFlow	Partial	No	No	High	Partial
DAGit	Native	No	No	Medium	No
Wukong	No	No	No	High	Partial
Proposed Solution	Native	Yes	Yes	High	Yes

tency and resource utilization. Despite its flexibility, it does not support partial execution or incremental data processing, leading to inefficiencies in workflows with dynamic dependencies.

DFlow (Shi et al., 2023) introduces a dataflow-based serverless workflow system aimed at reducing end-to-end execution time by enabling partial execution of tasks based on data availability. DFlow emphasizes dynamic scheduling and parallel task invocation, aligning with the goals of currying and continuation by allowing functions to execute incrementally based on data readiness. However, DFlow focuses on dataflow orchestration rather than leveraging functional programming constructs like currying and continuation for modularizing and reusing workflow components.

DAGit (Jana et al., 2023) is an open-source platform designed to streamline serverless application development through native support for Directed Acyclic Graphs (DAGs). It provides a declarative framework for defining workflows as DAGs, enabling developers to express complex dependencies, parallel execution, and conditional logic. Unlike platforms that rely on external orchestration layers, DAGit integrates DAG execution directly into its runtime, optimizing task scheduling and reducing overhead.

Wukong (Fouladi et al., 2019) addresses serverless workflow inefficiencies by optimizing parallelism and locality-aware scheduling. Its architecture maps workflows to distributed systems, leveraging fine-grained task partitioning and data locality to minimize latency. Wukong achieves significant performance improvements for parallel workloads, such as linear algebra operations, by reducing cross-node communication and prioritizing colocated tasks.

A critical gap in existing platforms is their inefficiency in handling *double spending*, where workflows incur costs from idle resources (e.g., waiting for dependencies) or redundant invocations, as discussed in (Kratzke, 2021; Baldini et al., 2017a). For instance, DAGit’s inability to suspend workflows during delays results in unnecessary billing, while Pheromone’s

lack of continuation forces full recomputation after interruptions.

Our proposal mitigates this through:

- **Currying:** Enables incremental execution, reducing redundant calls by processing available data immediately.
- **Continuation:** Suspends workflows during idle periods, freeing resources and avoiding costs for inactive states.
- **DAG Execution:** Optimizes parallelism and resource reuse, minimizing idle time.

By addressing these gaps, our enhancements position Apache OpenWhisk as a versatile platform capable of supporting modern, cost-efficient workflows while advancing the state of the art in serverless computing.

2.3 Motivation for Enhancements in Apache OpenWhisk

Apache OpenWhisk was chosen as it is an open-source platform, which allows for modifying the source code to implement these enhancements. The insights from platforms like DataFlower, Pheromone, and other research efforts underscore the need for foundational improvements in serverless platforms. The integration of **currying** would enable developers to modularize workflows, reducing redundancy and promoting reuse. Additionally, currying would help reduce latency by allowing partial execution of available data, improving efficiency and responsiveness in complex workflows. Incorporating **continuation** mechanisms would allow workflows to pause and resume as needed, minimizing latency caused by dependencies or delays. Finally, the addition of **DAG-based execution** would provide a powerful framework for representing and managing complex workflows, allowing parallel execution, partial recomputation, and optimized resource utilization.

By addressing these gaps, Apache OpenWhisk has the potential to evolve into a more expressive,

efficient, and flexible serverless platform, bridging the limitations of current approaches and positioning itself as a leading platform for complex serverless workflows with higher performance and modularity.

3 PROPOSED SOLUTION

Serverless workflows often face inefficiencies due to rigid execution models that require complete readiness of inputs before invoking functions. To address these challenges, we propose three interrelated enhancements that leverage partial execution principles to improve performance, resource utilization, and adaptability. Each enhancement targets a specific limitation in current serverless systems, collectively providing a robust solution for dynamic and efficient workflow orchestration.

3.1 Support Native Directed Acyclic Graphs (DAGs) Execution

Introducing native support for Directed Acyclic Graph (DAG) execution transforms the capabilities of serverless computing platforms by enabling direct orchestration of serverless functions within the core execution framework. This approach enhances the ability to express and manage complex workflows, where tasks have intricate data dependencies and can be executed in parallel without external orchestration layers.

The architecture for supporting native DAG execution involves modifying the Apache OpenWhisk controller to interpret a domain-specific language (DSL) designed for DAG orchestration. The controller processes the DSL input, constructing a DAG where nodes represent serverless functions and edges represent data dependencies. These DAG definitions and function states are stored in a CouchDB database, ensuring persistence and traceability throughout the execution process. The controller dynamically schedules functions based on data availability, minimizing latency and optimizing parallelism.

The design emphasizes a functional approach to specifying complex workflows, enabling developers to describe computations as dataflow graphs. This design choice supports parallelism and efficient scheduling without requiring developers to manage explicit synchronization. The DSL provides intuitive constructs for defining parallel maps and sequential reductions, allowing the system to infer dependencies and schedule functions accordingly. Each function invocation is treated as an independent action, preserving serverless principles such as statelessness and event-driven execution.

The implementation extends OpenWhisk's core by introducing a parser and executor for the DAG DSL. The parser translates the DSL into a JSON representation suitable for storage and processing within the OpenWhisk controller. During execution, the interpreter asynchronously triggers serverless functions as their input data becomes available, ensuring efficient parallel execution and reducing idle time. The minimal code footprint for this extension ensures compatibility with existing OpenWhisk features while introducing powerful orchestration capabilities.

The primary benefits of this solution include dynamic parallelism, efficient resource utilization, and simplified developer experience. By natively supporting DAG execution, OpenWhisk enables workloads that require complex data dependencies, such as machine learning pipelines and large-scale matrix operations. Developers can focus on the computation logic, while the platform manages concurrency and scheduling.

A practical example of the Cholesky decomposition using the DAG DSL is shown in Listing 1. This Python-based DAG representation demonstrates how complex matrix operations can be parallelized effectively using the native DAG execution model. Each function call and dependency is captured as a node in the DAG, and computations run as soon as inputs are ready.

Listing 1: DAG DSL program for Cholesky Decomposition.

```
def cholesky(A, B, N):
    nblocks = ceil(N/B)
    def for_j(j):
        L[j][j] = chol(A[j][j])
        par_for(i in range(j, j + nblocks)):
            L[i][j] = mul(inv(L[j][j]),
                          A[i][j])
    def for_k(k, j):
        par_for(l in range(k, nblocks)):
            A[k][l] = sub(A[k][l],
                        mul(T(L[k][j]), L[l][j]))
        par_for(k in range(j, j + nblocks)):
            for_k(k, j)
    seq_fold(L = A, j in range(0,
                                nblocks)) (for_j(j))
    return L
```

This enhancement represents a step forward in serverless architecture design by integrating complex orchestration capabilities directly into the execution engine. Developers gain a powerful tool for parallel computing, while the underlying platform ensures efficient, cost-effective, and scalable execution of serverless functions. The convergence of serverless computing with native DAG execution opens new horizons for scientific computing, data engineering, and real-time analytics, providing a powerful mech-

anism for modern cloud-native application development.

3.2 Support Native Currying

Currying, a functional programming technique, transforms a function with multiple arguments into a series of functions, each consuming one argument at a time (Hutton, 2016). By adapting this concept, serverless workflows can benefit from incremental execution. By allowing functions to be broken into a sequence of partial applications, currying enables serverless workflows to handle partially available data more efficiently. This flexibility is especially valuable in dynamic workflows where data dependencies might be resolved incrementally or asynchronously. Instead of waiting for all inputs to become available, a curried function can process available inputs immediately and defer the computation of the remaining logic until the required data arrives. This approach minimizes idle time and enhances responsiveness, particularly in workflows with interdependent tasks that do not all resolve simultaneously.

To illustrate the benefits of currying in a serverless context, consider a real-time video analysis pipeline involving multiple stages: frame extraction, object detection, and metadata tagging. In a traditional workflow, each stage must wait for the previous one to fully complete before initiating its processing. For example, object detection cannot begin until all frames have been extracted, and metadata tagging must wait for all object detection results. This sequential dependency unnecessarily increases latency and underutilizes the dynamic invocation capabilities of serverless platforms. Listing 2 highlights this limitation.

Listing 2: Traditional serverless workflow of video analysis.

```
def process_video(video_url):
    frames = extract_frames(video_url)
    objects = detect_objects(frames)
    metadata = tag_metadata(objects)
    return metadata
```

In this approach, each function, such as `detect_objects` and `tag_metadata`, waits for the complete output of the preceding function, `extract_frames`, before starting execution. This creates a bottleneck, particularly in workflows processing large volumes of data, where intermediate results could be used immediately.

With currying, the workflow can be restructured to process data incrementally and trigger subsequent stages dynamically. As shown in Listing 3, partial applications allow each function to operate on the available data as soon as it is produced.

Listing 3: Curried serverless workflow of video analysis.

```
def extract_frames_partial(video_url,
                           callback):
    for frame in
        extract_frames_generator(video_url):
        callback(frame)

def detect_objects_partial(frame, callback):
    objects = detect_objects(frame)
    callback(objects)

def tag_metadata_partial(objects, callback):
    metadata = tag_metadata(objects)
    callback(metadata)

def process_video_with_currying(video_url):
    extract_frames_partial(video_url, lambda
        frame:
        detect_objects_partial(frame, lambda
            objects:
            tag_metadata_partial(objects,
                                lambda metadata:
                                store_metadata(metadata)))
```

In this curried workflow, each frame extracted from the video triggers object detection immediately, and metadata tagging begins as soon as object detection results are available. This eliminates the need for later stages to wait for the entire dataset to be processed in earlier stages, reducing latency. By enabling partial execution, currying enhances the responsiveness of the pipeline and optimizes resource utilization, as downstream stages are invoked only when their input data is ready.

Furthermore, currying enriches the expressiveness of serverless workflows by making them more composable. For instance, the object detection and metadata tagging functions in this example can be reused in other workflows with minimal modification, as their partial application makes them flexible and adaptable. Developers can construct modular workflows that are easier to extend, reuse, and maintain, reducing overall development effort.

This adoption of currying in serverless computing aligns with the goal of enabling expressive, efficient, and latency-optimized workflows. By processing available data incrementally and constructing workflows as a series of lightweight, composable steps, currying addresses the inherent limitations of traditional serverless orchestration while leveraging functional programming principles to enhance cloud computing platforms.

3.3 Support Native Continuation

Continuation, rooted in functional programming (Appel, 1992), enables workflows to suspend and re-

sume dynamically, addressing inefficiencies in long-running serverless tasks. By pausing execution during delays (e.g., waiting for external inputs) and resuming only when dependencies resolve, it eliminates idle resource consumption — critical for cost-sensitive and scalable workflows.

For instance, in a document processing pipeline (text extraction → sentiment analysis → manual annotation), traditional workflows idle between stages, billing continuously. Continuation suspends execution after sentiment analysis, freeing resources until annotations arrive. This avoids wasteful billing and preserves scalability without altering workflow logic.

Listing 4 demonstrates the limitations of a traditional serverless workflow.

Listing 4: Traditional serverless workflow of document processing.

```
def process_document(doc_url):
    text = extract_text(doc_url)
    sentiment = analyze_sentiment(text)
    annotations =
        get_user_annotations(sentiment)
    return apply_annotations(annotations,
        sentiment)
```

In this approach, the function `get_user_annotations` requires external input from the user, causing the workflow to remain idle while waiting. The entire workflow remains active, leading to inefficient resource utilization and potential timeouts for long-running tasks.

With native continuation, the workflow can suspend itself after completing sentiment analysis and wait for the user's annotations to resume. This ensures that resources are only allocated during active computation phases. Listing 5 illustrates how continuation can be applied.

Listing 5: Continuation-enabled serverless workflow of document processing.

```
def process_document_continuation(doc_url):
    text = extract_text(doc_url)
    sentiment = analyze_sentiment(text)
    suspend_workflow(lambda:
        get_user_annotations_async(sentiment,
            lambda annotations:
                resume_workflow(lambda:
                    apply_annotations(annotations,
                        sentiment))))
```

In this continuation-enabled workflow, the `suspend_workflow` function pauses execution after sentiment analysis. Once user annotations are provided asynchronously, the workflow resumes execution at the `resume_workflow` point, completing the remaining tasks. This ensures that resources are used only when needed and avoids unnecessary idling or resource contention.

The benefits of adopting continuation in serverless workflows extend beyond resource optimization. By enabling dynamic pausing and resumption, continuation allows workflows to handle unpredictable or asynchronous dependencies more gracefully. Developers can design workflows that adapt to real-time conditions, making them more robust and responsive to external inputs. Additionally, continuation reduces the likelihood of workflow timeouts, which is particularly valuable for long-running tasks that rely on human input or external data sources.

By integrating native continuation into serverless platforms, workflows become more efficient and scalable, addressing the limitations of traditional execution models. Continuation aligns with the principles of event-driven architecture and asynchronous programming, enhancing the capability of serverless computing to support complex, real-world workflows.

4 EVALUATION

4.1 Serverless Trilemma Fulfillment

Baldini et al. (Baldini et al., 2017c) define the *serverless trilemma*, which outlines three core constraints for effective function composition: (1) **no double-billing**, (2) **black-box usage**, and (3) **substitution principle**. Our enhancements—native DAG execution, currying, and continuation—collectively resolve these constraints:

- **Native DAG Execution:** By decoupling function calls and treating DAGs as atomic actions, it eliminates double-billing (no waiting charges), enforces black-box usage (no source code exposure), and ensures substitutability (DAGs behave as single functions).
- **Currying:** Partial function application processes incremental data without redundant billing. Curried functions retain black-box integrity (no internal code access) and act as substitutable units (same input/output behavior as non-curried functions).
- **Continuation:** Suspends workflows during idle periods to prevent billing, preserves black-box constraints (execution resumption requires no code modification), and maintains substitutability (paused workflows behave as single logical units).

Thus, native DAG execution, currying, and continuation collectively provide a strong foundation for serverless function composition while fully satisfying the serverless trilemma.

4.2 Preliminary Evaluation

A preliminary prototype has been developed and remains under active development. This prototype includes a patch of 800 lines of code integrated into the Apache OpenWhisk core. The codebase modification focuses on extending the reactive core to support native DAG execution and functional programming constructs directly within the controller.

This preliminary version has successfully demonstrated the fundamental principles of DAG execution. The integration enables dynamic orchestration of serverless workflows, reducing latency and enhancing modularity, aligning with the theoretical goals outlined in the design phase.

A preliminary evaluation of our prototype has been conducted, demonstrating improvements over traditional serverless execution models. Our prototype successfully completed identical Cholesky decompositions of various sizes and block numbers with an average reduction in completion time of 28% compared to *numpywren* (Shankar et al., 2018). The evaluation also showed superior parallelization and resource utilization, particularly for workloads involving matrix decompositions where our prototype processed blocks in parallel, reducing idle time by 32%.

The results indicated higher worker utilization and better parallelization with our prototype, leading to improved efficiency. CPU utilization levels on the virtual machines were measured, with our prototype achieving a 27% higher saturation compared to *numpywren*. Network overhead was reduced when passing data between functions due to the optimized execution model, showing a 30% reduction in data transfer time.

The support for currying and continuation in our prototype is expected to provide key benefits. However, these features are not currently supported in the current implementation. Once integrated, they are anticipated to offer improvements such as enabling partial function execution for modular workflows and facilitating long-running computations through pausing and resumption.

As the implementation matures, further testing and optimization are required. The current focus is on ensuring compatibility with the existing OpenWhisk components and validating the performance benefits in real-world scenarios. Early tests indicated promising results, with improved parallelization and reduced cold start impact compared with traditional function chaining models.

5 FUTURE WORK

Future work on the current prototype holds significant potential. Evaluation across diverse real-world use cases is essential to validate our prototype scalability and efficiency. Future work could focus on the following directions:

First, DAG compilation currently occurs within the OpenWhisk controller. A key improvement would be shifting this compilation process to the CLI, enabling proactive type-checking and immediate error detection during DAG creation. By preemptively compiling a structured representation of function dependencies, the system could facilitate advanced analytics, including graph and edge analyses for workflow optimization.

Second, extending the scope of these features to support more complex scenarios, such as workflows with dynamic or conditional dependencies, would significantly enhance their usability. For instance, enabling conditional branching within DAGs or natively supporting higher-order functions would make the platform more versatile.

Third, the integration of intelligent scheduling algorithms for DAG execution could further optimize resource utilization and reduce latency. By leveraging reinforcement learning or heuristic-based approaches, the system could dynamically predict resource requirements, adaptively schedule tasks, and minimize execution latency. Such improvements could lead to more efficient resource utilization, reducing operational costs while improving performance.

Fourth, one avenue for improvement is the development of a graphical domain-specific language (DSL) editor, allowing users to visually design and manage DAG workflows. This enhancement would improve usability, reduce complexity, and provide better observability into function execution, dependencies, and performance metrics.

Finally, comprehensive evaluation of the prototype in real-time scenarios is critical. This includes benchmarking execution time, worker utilization, and resource efficiency for workflows like Cholesky decomposition across varying matrix sizes and block configurations. Metrics such as the number of workers executing concurrently, CPU utilization levels, network overhead between function invocations, and database access latency must be measured to assess scalability and identify bottlenecks. Additionally, comparative benchmarking against state-of-the-art orchestrations (e.g., AWS Step Functions, Google Cloud Workflows) under diverse workloads — including machine learning pipelines, IoT data processing, and large-scale computational tasks — will help

assess performance advantages and guide further refinements.

By addressing these gaps, the prototype can evolve into a robust, production-ready solution, bridging the divide between theoretical enhancements and practical applicability in modern serverless ecosystems.

6 CONCLUSION

This paper has proposed enhancements to Apache OpenWhisk to address the limitations of existing serverless platforms in workflow orchestration and execution. The platform can handle dynamic, asynchronous, and interdependent tasks more efficiently by introducing native support for currying, continuation, and DAG execution.

Currying allows workflows to process partially available data incrementally, reducing latency and enhancing modularity. Continuation enables workflows to suspend and resume dynamically, optimize resource utilization, and improve scalability for long-running tasks. DAG execution transforms workflow orchestration by enabling parallel execution, dynamic scheduling, and efficient handling of complex dependencies. Together, these features align with the principles of serverless computing, which offers greater expressiveness, efficiency, and scalability.

These enhancements position Apache OpenWhisk as a more powerful serverless platform capable of supporting modern, complex workflows. By reducing operational overhead and improving resource utilization, they bring serverless computing closer to its promise of seamless scalability and efficiency, empowering developers to focus on innovation while the platform handles the complexities of execution.

REFERENCES

- Appel, A. W. (1992). *Compiling with Continuations*. Cambridge University Press.
- Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A., and Suter, P. (2017a). Serverless computing: Current trends and open problems.
- Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S. J., Ishakian, V., Muthusamy, V., Rabbah, R. M., Suter, P., and Tardieu, O. (2017b). Serverless computing: Current trends and open problems. *Research Advances in Cloud Computing*, pages 1–20.
- Baldini, I., Cheng, P., Fink, S. J., Mitchell, N., Muthusamy, V., Rabbah, R., Suter, P., and Tardieu, O. (2017c). The serverless trilemma: function composition for serverless computing. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software - Onward! 2017*, pages 89–103. ACM Press.
- Fouladi, S., Mitchell, E. J., Wei, H., Devanbu, P., and Zhao, J. (2019). Wukong: A scalable and locality-enhanced framework for serverless parallel computing. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '19)*, pages 1–12, Santa Cruz, California, USA. ACM.
- García-López, P., Arjona, A., Sampé, J., Slominski, A., and Villard, L. (2020). Triggerflow: Trigger-based orchestration of serverless workflows. *arXiv preprint arXiv:2006.08654*.
- Hutton, G. (2016). *Programming in Haskell*, chapter 5, pages 73–77. Cambridge University Press. Discusses the concept of currying and its applications in functional programming.
- Jana, A., Kulkarni, P., and Bellur, U. (2023). DAGit: A Platform For Enabling Serverless Applications. In *Proceedings of the 2023 IEEE 30th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 367–376.
- Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C.-C., Yadwadkar, N., Gonzalez, J. E., Popa, R. A., Stoica, I., and Patterson, D. A. (2019). Cloud programming simplified: A Berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*.
- Kratzke, N. (2021). *Cloud Computing*. Hanser Verlag, 2nd edition. Section 12.6.1.
- Li, Z., Xu, C., Chen, Q., Zhao, J., Chen, C., and Guo, M. (2024). Dataflow: Exploiting the data-flow paradigm for serverless workflow orchestration. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4, ASPLOS '23*, page 57–72, New York, NY, USA. Association for Computing Machinery.
- Lin, P.-M. and Glikson, A. (2019). Mitigating cold starts in serverless platforms: A pool-based approach.
- Shahrad, M., Fonseca, R., Íñigo Goiri, Chaudhry, G., Batum, P., Cooke, J., Laureano, E., Tresness, C., Russinovich, M., and Bianchini, R. (2020). Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider.
- Shankar, V., Krauth, K., Pu, Q., Jonas, E., Venkataraman, S., Stoica, I., Recht, B., and Ragan-Kelley, J. (2018). numpywren: serverless linear algebra. *arXiv preprint arXiv:1810.09679*.
- Shi, X., Li, C., Li, Z., Liu, Z., Sheng, D., Chen, Q., Leng, J., and Guo, M. (2023). Dflow: Efficient dataflow-based invocation workflow execution for function-as-a-service. *arXiv preprint arXiv:2306.11043*.
- Yu, M., Cao, T., Wang, W., and Chen, R. (2022). Following the data, not the function: Rethinking function orchestration in serverless computing.