Peeking Behind the Serverless Implementations and Deployments of the Montage Workflow

Simon Triendl
csav3166@student.uibk.ac.at
University of Innsbruck
Department of Computer Science
Innsbruck, Tyrol, Austria

Sashko Ristov
sashko.ristov@uibk.ac.at
University of Innsbruck
Department of Computer Science
Innsbruck, Tyrol, Austria

ABSTRACT

The development of serverless scientific workflows is a complex and tedious procedure and opens several challenges in how to compose workflow processing steps as serverless functions and how much memory to assign to each serverless function, which affects not only the computing resources, but also the networking communication to the cloud storage. Merging multiple processing steps into a single serverless function (fusion) reduces the number of invocations, but restricts the developer to assign the maximum required memory of all fused processing steps, which may increase the overall costs.

In this paper, we address the aforementioned challenges for the widely used Montage workflow. We created three different *workflow implementations* (*fine*, *medium*, and *coarse*) for two cloud providers AWS and GCP and deployed workflow functions with different memory assignments 135 MB, 512 MB, and 1 GB (*function deployments*). Our experiments show that many Montage functions run cheaper and faster with more memory on both providers. Consequently, selecting the most cost-effective memory configuration, as opposed to the minimal memory, resulted in a reduction of the makespan by 67.27 % on AWS and 10.93 % on GCP. Applying the same to workflow implementations with fewer functions (coarse) led to a further reduction in the makespan by 24.98 % on AWS and 12.96 % on GCP, while simultaneously reducing the total cost by 5.33 % and 1.99 %, respectively. Surprisingly, the fastest implementation was the medium implementation executed on AWS.

CCS CONCEPTS

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

KEYWORDS

cost, FaaS, performance, serverless, workflows.

ACM Reference Format:
Simon Triendl and Sashko Ristov. 2024. Peeking Behind the Serverless Implementations and Deployments of the Montage Workflow. In *Companion of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24 Companion), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3629527.3651420

ICPE '24 Companion, May 7–11, 2024, London, United Kingdom

ACM ISBN 979-8-4007-0445-1/24/05.
https://doi.org/10.1145/3629527.3651420

1 INTRODUCTION

Serverless computing is a scalable and cost-effective execution environment for *data-intensive applications*, such as scientific workflow applications, whose complexity and scalability grow [18, 23]. Domain experts code processing steps as serverless functions and orchestrate them into *serverless workflows* [2, 7, 9, 21, 31, 38, 45, 50]. Such workflows include complex applications from astrophysics (e.g., Montage [5]), bioinformatics (e.g., BWA [27]), earthquake simulations (e.g., Cybershake [29]), and many more.

Although the heterogeneous nature of federated clouds [3, 28, 34, 40, 44, 49] brings many benefits in terms of cost [14], execution time [34, 38, 43], and scalability [22, 41], it raises several challenges. First, developers may map the workflow processing steps in a *fine-grained* manner, where each method of the workflow tasks may be deployed as a separate serverless function or a *coarse-grained* approach, in which multiple processing steps are merged in a separate serverless function. While the former approach offers the highest level of granularity and usually minimizes the cost, the latter usually improves the performance as it minimizes data transfers and uses more memory, which often leads to higher costs. We refer to each of these mappings as a *workflow implementation*. Once the functions of the workflow are coded, they are deployed on the selected provider and assigned with a specific amount of memory, often known as *function deployment*.

In this paper, we peek behind the different Montage workflow implementations and various function deployments to investigate how they affect the overall cost and performance. We used two cloud providers AWS and GCP and derived interesting conclusions. The evaluation showed that many workflow functions benefit both in terms of cost and performance when assigned with more memory. Moreover, the coarse implementations additionally reduce the overall cost while improving performance. Surprisingly, the medium implementation of Montage achieved the fastest makespan on AWS. While the workflow community [12] recommends that a workflow orchestrator should make intelligent decisions about the placement of workflow tasks across different sites in the continuum, our results show that the workflow orchestrator should consider fusion and fission of the workflow tasks, as well as the assigned memory to each newly created task.

The remaining part of the paper is organized in five sections. We first present details for the Montage workflow and its processing steps in Section 2. Further on, Section 3 presents various implementation and deployment challenges and how they affect performance and cost. In Section 4, we evaluate several deployments of the Montage workflow functions and determine the cheapest and fastest implementations on two providers AWS and GCP. In Section 5, we

position our work compared to the related work and in Section 6, we conclude the paper and present our plans for future work.

2 INTRODUCTION TO THE MONTAGE WORKFLOW

Montage [5], created by the NASA/IPAC Infrared Science Archive, is an open-source toolkit to assemble astronomical images into custom mosaics. It consists of multiple independent modules indended to be used in a choreography that exploits the parallelization of several processing steps. This orchestration of modules is commonly referred to as *Montage workflow*. The Montage workflow is widely used by the scientist due to its high computation and communication complexity [1, 2, 13, 23, 31–33].

At a high level, the processing steps to compute a mosaic involve initially calculating the geometry of the mosaic, followed by re-projecting the input images, stored in Flexible Image Transport System (FITS) format. Subsequently, background radiation correction is applied to ensure uniformity across the mosaic. Finally, the re-projected and background-corrected images are co-added to generate the mosaic.

The Montage toolkit implements all of these computations with a higher granularity. Additionally, certain stages necessitate the dynamic generation of input variables for subsequent stages. Therefore, a naive fine-grained implementation comprises 19 stages.

(0)-(1) prepare `mProjectPP I-II`: Prepares the parameters for the `mProjectPP` instances.
(2) `mProjectPP`: Performs parallel fast re-projection of the input FITS files according to the region header file.
(3)-(6) prepare `mDiffFit I-IV`: Extracts the header information from the input FITS files and computes a list of overlapping images. Based on the overlaps, the parameters for the `mDiffFit` and `mFitPlane` instances are created.
(7) `mDiff`: Performs parallel image difference between a pair of re-projected images and creates a new FITS file.
(8) `mFitPlane`: Parallelly fits a plane to the FITS files generated by `mDiff`.
(9) prepare `mConcatFit`: Prepares the parameters for `mConcatFit`.
(10) `mConcatFit`: Merges the multiple plane fit parameter into a single file for `mBgModel`.
(11) `mBgModel`: Creates a background radiation model to determine a set of corrections stored in a table.
(12)-(13) prepare `mBackground I-II`: Prepares the corrections table and the parameters for the `mBackground` instances.
(14) `mBackground`: Performs parallel corrections on the re-projected input FITS files.
(15) `mImgTbl`: Extracts the header information from the corrected FITS files and stores them in a table.
(16) `mAdd`: Co-add the corrected FITS files according to the header region file.
(17) `mShrink`: Reduces the size of the co-added FITS file.
(18) `mViewer`: Converts the shrunk FITS file to a PNG or JPEG format.

3 IMPLEMENTATION AND DEPLOYMENT CHALLENGES

In this section, we discuss the challenges that developers face when they decide which implementation of a workflow to use and which function deployments of that implementation.

3.1 Deployment challenges

Cloud providers offer fine-grained configuration of RAM memory to their serverless functions. Developers can deploy their functions starting from 128 MB up to tens of gigabytes, depending on the provider. However, providers use different approaches for the CPU. AWS claims that the CPU scales linearly as the assigned memory. Azure configures the memory dynamically based on the need, while GCP users can configure the CPU, as well, but are additionally charged for the CPU resources.

Function deployments with more memory usually achieve better performance [24] based on the Gustafson's Law [16]. Moreover, function deployments on AWS with more than 1.5 GB achieve the maximum bandwidth of the underlying virtual machines [47], thereby transferring data to the collocated storage faster than the function deployments with lower memory [26, 48]. Still, the speedup when increasing the resources are limited to the linear speedup, often reaching the sublinear speedup.

3.2 Implementation challenges

The cost and performance of a workflow is affected by its implementations. We showed earlier that Montage comprises 19 processing steps, some of which are nested in a parallel loop. If all processing steps are merged in a single function, then the invocation overhead will be minimized as the workflow calls a single function [35], but in that case, the developers lose parallelism. Therefore, fusion of the processing steps should be considered mainly for a sequence of sequential processing steps. Unfortunately, providers limit the size of the function codes, which additionally restricts the decision how many processing steps to merge and deploy in a single function. Another challenge in the fusion of multiple processing steps is the amount of memory that should be assigned to the equivalent function. For instance, if a processing step ps_1 needs 128 MB and the subsequent processing step ps_2 requires 2 GB, then the fused function f_{12} also needs 2 GB, which additionally may increase the costs since the processing step ps_1 is assigned with more memory than required.

4 EXPERIMENTAL DESIGN

In this section, we introduce our implementation and assess how different function deployments and workflow implementations influence the cost and performance of the Montage workflow on the two public cloud providers AWS and GCP. Our implementation is written in Java, utilizing JDK version 17. We encapsulated each Montage toolkit executable within a method, offering a high-level interface for the parameters. Upon method invocation, the executable is executed in a process. We utilized Montage version 6.0 for this study.

We devised three workflow implementations, denoted as the *fine*, *medium*, and *coarse* implementations, as presented in Fig. 1.

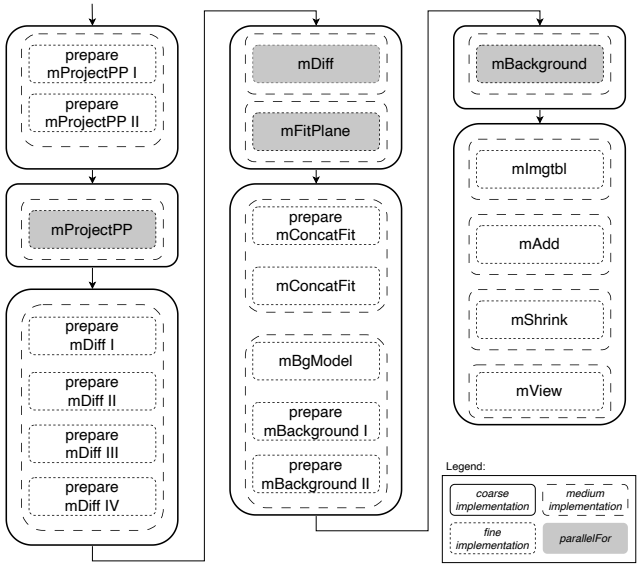

Figure 1: Composition of *coarse*, *medium* and *fine* Montage workflow implementations, depicted with solid, dashed, and dotted rectangles, respectively.

The fine implementation (dotted boxes in Fig. 1) focuses on dividing the Montage computations with the highest granularity, achieved by segregating each method invocation into individual functions. The medium implementation (dashed boxes in Fig. 1) reflects the employed workflow[17] that can be executed with the xAFCL serverless workflow management system [38]. Lastly, the coarse implementation (solid boxes in Fig. 1) targets a minimal granularity approach, also used by Deelman [13] and Berriman [5]. All three implementations and their compositions run the same work and provide the same output, but their computations are grouped differently in serverless functions, leading to 19, 12, and 7 functions for the fine, medium, and coarse implementations, respectively. Note that the functions marked with gray boxes are executed multiple times using data parallelism. Also, there are other implementations of Montage that use 9 functions [31, 32], which we did not consider since they are very similar to the coarse implementation.

4.1 Experiment setup

We deployed the functions of all three workflow implementations on GCP as 2nd generation functions in the `europe-west1` region, and on AWS in the `eu-central-1` region, as the closest regions to University of Innsbruck to minimize the invocation overhead [35]. Each function was deployed with 135 MB, 512 MB, and 1 GB of memory. The usage of 135 MB was necessary on GCP due to its minimal memory assignment being 135 MiB, roughly equivalent to 135 MB. Each function was configured with the maximum timeout duration to prevent premature failure. The storage, which were respectively hosted on Amazon S3 and Google Cloud Storage, were collocated in the same region for both platforms, based on the recent reports to collocate the functions closer to the data [42, 43].

To conduct these experiments we used a custom workflow execution engine written in Python. Parallelization was achieved using a thread pool, with the number of workers regulating the concurrency of the execution. Due to account limitations on both providers, we set the concurrency level to 10 and the block size to 1. This constraint does not affect the results, as the measured execution time reflects the function's internal wall clock time. Thus, factors such as invocation latency and cold starts do not influence the measurements.

Each workflow implementation, along with its corresponding functions deployments, underwent 5 executions like in other recent works [10, 17, 35], and the median execution time of each function was considered. For functions invoked in parallel, the median execution time of all parallel instances was calculated and then extrapolated to determine the total runtime. We choose the median over the mean to mitigate the impact of outliers.

The runtime cost is calculated based on publicly available pricing information from AWS[1] and GCP[2]. Because the storage is collocated, file transfer expenses are confined to the number of file accesses (downloads and uploads). Both AWS[3] and GCP[4] adhere to the same pricing model in this regard.

4.2 Memory impact

We compiled the runtime and its associated cost, excluding file transfer expenses, for each function and function deployment within the fine workflow implementation into a table. Our objective is to identify deployments that minimize runtime costs. To make the detection easier, we computed the cost for 1 million invocations and highlighted the cheapest deployments in bold. Dash symbols indicate that the deployment's memory was insufficient to execute the function.

4.2.1 AWS. In Table 1 we depicted the results for AWS. Observing the runtimes of the functions reveals that deployments with higher memory result in a faster execution. While the cost scales linearly with both runtime and memory assignment, some functions experience a super-linear speedup due to increased memory assignment [39]. Consequently, deployments with higher memory become both more cost-effective and faster compared to their lower-spec counterparts. Functions 2, 7-8, 11, and 14 highlight that this phenomenon is not universal but rather dependent on the behavior of each specific function. Notable are the functions 0, 1, 6, 12-13, 15, and 17, whose deployments with even 1 GB dominate the other deployments with less memory, both in terms of runtime and cost. Function 6 experiences the highest impact running at just 0.77 % of the time and accounting for 5.76 % of the cost compared to its deployment with 135 MB. When comparing the cost-optimal function deployments to deployments with minimal feasible memory, we observe a reduction in makespan from 434.96 s to 142.33 s, representing a decrease of 67.27 %. Simultaneously, the cost decreased from $11.57 to $11.19, a reduction of 3.28 %.

[1] https://aws.amazon.com/lambda/pricing/
[2] https://cloud.google.com/run/pricing
[3] https://aws.amazon.com/s3/pricing/
[4] https://cloud.google.com/storage/pricing

Table 1: Runtime and cost for different deployments of the *fine* workflow implementation on AWS.

f	*Runtime (s)*			*Cost ($) for 1M invocations*		
	135MB	*512MB*	*1024MB*	*135MB*	*512MB*	*1024MB*
0	0.285	0.055	**0.005**	0.63	0.45	**0.09**
1	0.378	0.082	**0.005**	0.837	0.68	**0.08**
2	635.943	**165.571**	84.685	1409.00	**1382.79**	1412.97
3	-	18.054	**7.152**	-	150.78	**119.33**
4	-	7.559	**2.187**	-	63.13	**36.49**
5	-	4.859	**0.921**	-	40.58	**15.36**
6	11.462	2.830	**0.088**	25.39	23.63	**1.46**
7	**2409.051**	653.139	334.195	**5337.54**	5454.82	5576.06
8	613.759	**159.886**	80.283	1359.85	**1335.32**	1339.52
9	-	4.440	**0.672**	-	37.07	**11.20**
10	-	11.178	**5.328**	-	93.35	**88.89**
11	**17.563**	9.893	3.209	**38.91**	82.62	53.543
12	12.978	3.133	**0.124**	28.75	26.16	**2.06**
13	0.514	0.115	**0.005**	1.14	0.96	**0.08**
14	-	**202.668**	101.904	-	**1692.61**	1700.26
15	116.177	30.065	**13.166**	257.40	251.09	**219.66**
16	-	84.068	**38.166**	-	702.11	**636.79**
17	87.483	22.823	**9.570**	193.82	190.60	**159.67**
18	-	17.204	**6.994**	-	143.68	**116.69**

Table 2: Runtime and cost of different deployments of the *fine* workflow implementation on GCP.

f	*Runtime (s)*			*Cost ($) for 1M invocations*		
	135MB	*512MB*	*1024MB*	*135MB*	*512MB*	*1024MB*
0	**0.04**	0.005	0.006	**0.43**	0.91	1.63
1	**0.085**	0.007	0.005	**0.43**	0.91	1.63
2	-	210.939	**115.112**	-	1956.21	**1916.01**
3	-	**8.151**	5.672	-	**75.31**	93.34
4	-	**4.568**	3.57	-	**42.24**	58.95
5	-	**1.878**	1.324	-	**17.45**	22.92
6	1.302	**0.190**	0.235	6.05	**1.83**	4.91
7	-	**247.796**	158.587	-	**2330.92**	2770.85
8	-	**84.89**	54.162	-	**906.47**	923.61
9	**1.886**	1.683	1.059	**8.21**	15.61	18.01
10	-	**25.667**	18.254	-	**236.03**	299.68
11	-	**4.516**	2.811	-	**42.24**	47.49
12	1.384	**0.361**	0.304	6.05	**3.67**	6.55
13	**0.03**	0.005	0.005	**0.43**	0.91	1.63
14	-	**214.309**	126.167	-	**1983.76**	2112.52
15	-	**8.863**	6.334	-	**81.73**	104.80
16	-	-	**38.58**	-	-	**632.12**
17	-	23.909	**13.083**	-	220.41	**214.52**
18	-	**10.436**	6.294	-	**96.43**	103.17

4.2.2 GCP. Table 2 presents the results for GCP, which are comparable to, but not as pronounced as for AWS. Only functions 2, 12 and 17 demonstrate cheaper execution with higher memory deployments. Note that the functions 12 and 17 do not experience this effect on AWS. It is noteworthy that GCP rounds the execution time to the nearest 100 ms increment, which mitigates the speed-up effect for short-running functions such as 0, 1, and 13. Furthermore, GCP appears to be more restrictive with memory, as evidenced by several functions that have sufficient memory on AWS but fail on GCP. This is particularly apparent for function 16, which requires a minimum of 1 GB to execute. One possible reason for this discrepancy may be that the ephemeral storage, used by the Montage executables and their input files, scales with assigned memory on GCP. In contrast on AWS, it is set by default to 512 MB, which provides sufficient storage for most functions. While we still observe a reduction in makespan comparing the cost-optimal deployments with the minimal feasible memory deployments from 147.82 s to 131.67 s, equivalent to a 10.93 % reduction, the improvement is considerably smaller compared to AWS. When we apply the same to the runtime cost, we observe a reduction from $8.64 to $8.59, yielding a negligible 0.05 % decrease.

4.3 Fine vs. Medium vs. Coarse

We further applied the same approach as described in Section 4.2 to the medium and coarse implementations of the Montage workflow.

4.3.1 Cost-optimal memory configuration. The optimal function deployments for cost are presented in Table 3, which indicates that the optimal memory configuration for fused functions is not necessarily equivalent to that of separated functions. Two such examples can be seen when comparing AWS functions 7 and 8 in the medium and coarse implementations. While the separated functions worked most cost-effectively with 512 MB, the merged function is optimal with 135 MB. Conversely, for GCP, the exact same functions work optimally at 512 MB when separated, but when fused, they are optimal at 1 GB. Another notable example is AWS function 7 in both the fine and medium implementations. Despite their equivalence, the optimal deployment shifts from 135 MB in the fine implementation to 512 MB in the medium implementation. This discrepancy may be attributed to variations in the performance of the cloud provider. A similar effect is observed with GCP functions 15 and 18 in the fine and medium implementations. Increasing the number of iterations may help mitigate such results in the future.

4.3.2 Makespan analysis. To assess the overall impact of the cost-effective function deployments on the workflow implementations, we computed the makespan of each workflow along with its associated cost. Assuming no account-specific concurrency limit, we determined the makespan with maximum concurrency for each parallel function (2, 7, 8, and 14). Consequently, the makespan represents the total sum of all non-parallel function runtimes and the maximum runtime of any parallel instances. The different workflow implementations not only affect the optimal function deployments but also influence the number of files that need to be transferred. Fusing multiple sequential functions into one reduces file transfers, thereby affecting the runtime and cost. Hence, we computed costs both with and without factoring in file transfers. The makespan results are visually presented in Figure 2, while the cost is illustrated in Figure 3. When analyzing the makespan on AWS, the medium implementation proves to be the fastest, completing in 106.77 s, which is 93.08 % of the time taken by the coarse implementation (114.70 s), and 75.02 % of the time taken by the fine implementation (143.33 s). This result is supported by Table 3, where it is evident that the medium implementation utilizes a function deployment with either equal or higher memory assignment compared to the

Table 3: Cost optimal memory configurations in MB.

f	Instances	AWS			GCP		
		fine	*medium*	*coarse*	*fine*	*medium*	*coarse*
0	1	1024	1024	512	135	135	135
1	1	1024			135		
2	30	512	512	512	1024	512	512
3	1	1024	1024	512	512	512	512
4	1	1024			512		
5	1	1024			512		
6	1	1024			512		
7	141	135	512	135	512	512	1024
8	141	512	512		512	512	
9	1	1024	1024	512	135	512	512
10	1	1024			512		
11	1	135	1024		512	1024	
12	1	1024			512		
13	1	1024			135		
14	30	512	512	512	512	512	512
15	1	1024	1024	1024	512	1024	1024
16	1	1024	1024		1024	1024	
17	1	1024	1024		1024	1024	
18	1	1024	1024		512	1024	

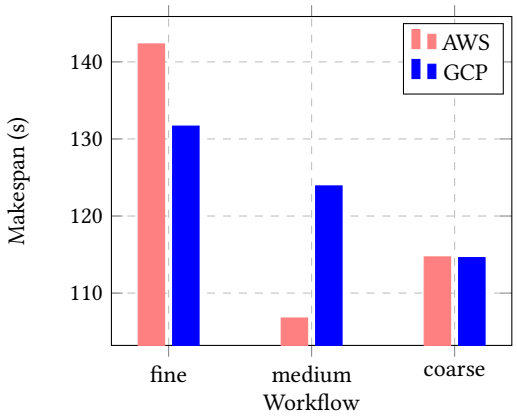

Figure 2: Makespan of cost-optimal Montage implementations on AWS and GCP.

other implementations. The reason for the coarse implementation being the second fastest may be attributed to the reduced number of file transfers, coupled with the utilization of function deployments with either equal or less memory compared to the medium implementation. The longer makespan in the fine implementation can be explained by examining function 7, which has a deployment of 135 MB and its associated runtime in Table 1. Although executed in parallel with 141 instances, its presence significantly impacts the critical path, thus extending the makespan.

Contrary to AWS, on GCP, the coarse implementation executes the fastest, completing in 114.61 s, which is 92.49 % of the time taken by the medium implementation (123.91 s) and 87.04 % of the time taken by the fine implementation (131.67 s). While there are minimal changes in the most cost-effective deployments between the coarse and medium implementations, as depicted in Table 3,

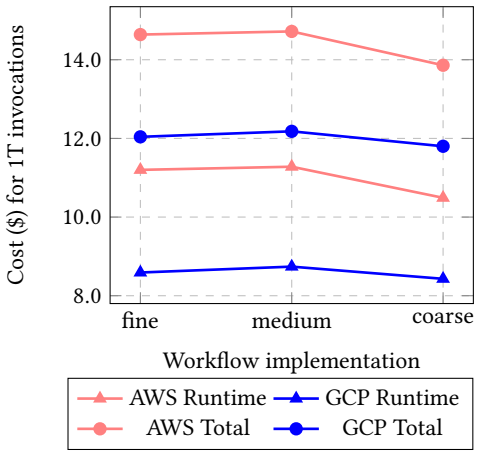

Figure 3: Runtime cost and total cost for 1000 invocations on AWS and GCP.

their order may primarily be attributed to the reduced number of file transfers in the coarse implementation. The discrepancy between the medium and the fine implementation may result from functions 9, 13, 15, and 18, each utilizing only a quarter of the memory compared to the other implementations.

4.3.3 Analize the cost. When evaluating runtime cost, the coarse implementation stands out as the most economical option followed by the fine and the medium implementations on both providers. The primary contributing factor is once again the number of file transfers, which allows the coarse implementation to utilize function deployments with equal or less memory compared to the other implementations, without decisively increasing the makespan. On AWS, the coarse implementation is 6.71 % cheaper than the fine, whereas, on GCP, the difference is less pronounced, with the coarse implementation only saving 1.90 % in costs compared to the fine implementation. We suspect that function 7-8 primarily contribute to this discrepancy, as AWS employs a 135 MB deployment compared to GCP's 1 GB and the function is executed on 141 instances in parallel. Examining the file transfer costs demonstrates that the coarse implementation saves 2.25 % compared to the fine and 2.02 % compared to the medium implementation. Since both AWS and GCP utilize identical pricing structures, this finding is applicable to both service providers. Evaluating the total cost reveals a 5.33 % saving on AWS when comparing the fine to the coarse and a 1.99 % cost saving on GCP.

Overall, the coarse implementation emerges as the most economical choice on both providers. On GCP, it even achieves Pareto optimality, dominating both the makespan and cost. Particularly for data-intensive workflows like Montage, bundling computations into single serverless functions, thereby reducing the necessity for file transfers, substantially enhances cost efficiency. However, the fact that the fine implementation is cheaper than the medium on both providers indicates the potential for separation with a higher granularity at the expense of a longer makespan.

4.4 Threats to validity

Whenever conducting experiments using public serverless providers it is important to highlight their opacity with respect to their hardware and software stack. This heterogeneity may lead to disparities in performance, as noted by Maissen *et al.* [30] and Copik *et al.* [11]. Moreover, Container-as-a-Service (CaaS) is reported to be a cheaper platform than FaaS for long running workflows and a hybrid approach with FaaS may circumvent service-specific limitations [8]. However, we used FaaS only since we evaluated a short-running version of montage, which mainly benefits from FaaS. Another contributing factor is the timeliness of the evaluation, as the cloud provider experiences load at different points in time, which affects performance, as reported by Kelly *et al.* [25]. Furthermore, our approach relies on the cost-performance ratio currently offered by the providers in the selected regions. It's worth mentioning that both AWS and GCP offer varied pricing for their services depending on the region. Therefore, repeating these experiments in different regions will likely yield divergent results. Finally, we ran five repetitions, which may be insufficient for long-running cloud applications, which can experience high variability in network traffic [46]. However, all evaluated Montage workflow implementations are running for a few minutes.

5 RELATED WORK

In this section we discuss various observations reported by the researchers specifically for the Montage workflow and how serverless functions, in general, are affected by various deployments of the functions with different memory.

5.1 Observations for the Montage workflow

Early research was conducted by Jackson *et al.* [20], who evaluated the applicability of scientific computing on AWS EC2. Humphrey *et al.* [19] investigated a hybrid architecture, using in-house computational resources alongside virtual machines on Microsoft Azure. Juve *et al.* [23] characterized various I/O reads and writes, peak memory, and CPU utilization of all Montage tasks. Balis [4] implemented Montage using the Hyperflow's data-flow approach and a high level of abstraction to run the workflow independently of the underlying runtime environment. However, all the aforementioned works used virtual machines as resources, in which the memory is not fully assigned to the workflow tasks, but rather it is managed by the guest operating system.

Hyperflow [31] evaluated that serverless implementations of Montage are cheaper than the classical serverfull implementations deployed on AWS EC2 virtual machines. However, the authors also reported high variability in performance for the functions that are nested in a parallel loop. Hautz *et al.* [17] reported that functions run the computing part and download data faster using AWS Lambda and S3, while GCP functions upload data faster on GCP cloud storage. The authors used the implementation in the Abstract Function Choreography Language [37] and executed the workflow with the xAFCL [38] serverless workflow management system. We used this implementation of the workflow in our evaluation as the medium implementation.

5.2 Observations for FaaS deployments

Jonas *et al.* [22] reported huge delays when invoking more functions simultaneously. Ristov *et al.* [38] reported that the overall round trip time is increased. Moreover, the spawn start affects the performance of the functions [36].

FaaSt [34] analyzed the performance of various cloud providers to the same data storage for BWA. Other researchers reported a speedup when collocating the functions closer to the data rather than moving data to the function [42, 43]. We followed the latter approach and always collocated the functions together with the storage within the same region.

SimLess [35] introduced a deterministic model for the overall round trip time of serverless functions that are deployed across multiple regions of the cloud provider. The model estimates the overall round trip time in one region from the executions in another region of the same function. SizeLess [15] used an ML approach to estimate the function execution time by learning from running the same function with 512 MB. However, both approaches consider a fixed function and workflow setup, while SimLess analyzed the same implementation of the Montage, but still with a different problem size.

6 CONCLUSION

In this paper, we conducted a series of experiments to investigate how different serverless implementations of the Montage workflow with various function deployments affect its makespan and cost. Our investigation led to two important observations.

We first observed a superlinear speedup for some functions when assigning them with more memory, which yielded that they dominated their compatriots with less memory, both for the performance and cost. This insight was mainly observed on AWS due to the fine-grained pricing model down to 1 ms compared to GCP's coarse-grained of 100 ms. With this simple method, we were able to reduce the overall makespan and cost by 67.27 % and 3.28 % on AWS, respectively. On GCP, a smaller improvement was observed with a 10.93 % reduction in makespan and a marginal reduction of 0.05 % in cost. Secondly, the coarse implementation improves both performance and cost by 24.98 % and 5.33 % on AWS, and 12.96 % and 1.99 % on GCP. Surprisingly, on AWS, the medium implementation achieved the smallest makespan, further reducing the time by 6.92 % compared to the coarse implementation. However, this improvement came at a disproportionately higher cost.

We believe that these results will be very valuable for the research community. We will further extend our work in several directions. We will first investigate other serverless workflows [6, 17] and with more fine-grained memory setups. Further on, we will develop a multi-objective scheduler that will determine the optimal setup for the workflows across different providers, especially since our evaluation reported that none of the evaluated providers dominates the other for all workflow functions. Finally, based on the decisions of the scheduler, we will automatize the process of fusion of workflow functions, their packaging, and deployment.

ACKNOWLEDGEMENT

This research received funding from Land Tirol, under the contract F.35499.

REFERENCES

[1] Ali Al-Haboobi and Gabor Kecskemeti. 2021. Improving Existing WMS for Reduced Makespan of Workflows with Lambda. In *Euro-Par 2020: Parallel Processing Workshops: Euro-Par 2020 International Workshops, Warsaw, Poland, August 24–25, 2020, Revised Selected Papers 26*. Springer, 261–272.

[2] Aitor Arjona, Pedro García López, Josep Sampé, Aleksander Slominski, and Lionel Villard. 2021. Triggerflow: Trigger-based orchestration of serverless workflows. *Future Generation Computer Systems* 124 (2021), 215–229. https://doi.org/10.1016/j.future.2021.06.004

[3] Y. Babuji, J. Bryan, R. Chard, K. Chard, I. Foster, B. Galewsky, D. S. Katz, and Z. Li. 2021. Federated Function as a Service for eScience. In *2021 IEEE 17th International Conference on eScience (eScience)*. IEEE Computer Society, Los Alamitos, CA, USA, 251–252. https://doi.org/10.1109/eScience51609.2021.00046

[4] Bartosz Balis. 2016. HyperFlow: A model of computation, programming approach and enactment engine for complex distributed workflows. *Fut. Gen. Comp. Syst.* 55 (2016), 147 – 162. https://doi.org/10.1016/j.future.2015.08.015

[5] G Bruce Berriman, Ewa Deelman, John C Good, Joseph C Jacob, Daniel S Katz, Carl Kesselman, Anastasia C Laity, Thomas A Prince, Gurmeet Singh, and Mei-Hu Su. 2004. Montage: a grid-enabled engine for delivering custom science-grade mosaics on demand. In *Optimizing scientific return for astronomy through information technologies*, Vol. 5493. SPIE, 221–232.

[6] Tekin Bicer, Xiaodong Yu, Daniel J. Ching, Ryan Chard, Mathew J. Cherukara, Bogdan Nicolae, Rajkumar Kettimuthu, and Ian T. Foster. 2022. High-Performance Ptychographic Reconstruction with Federated Facilities. In *Driving Scientific and Engineering Discoveries Through the Integration of Experiment, Big Data, and Modeling and Simulation*, Jeffrey Nichols, Arthur 'Barney' Maccabe, James Nutaro, Swaroop Pophale, Pravallika Devineni, Theresa Ahearn, and Becky Verastegui (Eds.). Springer International Publishing, Cham, 173–189.

[7] Sebastian Burckhardt, Badrish Chandramouli, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, Christopher S. Meiklejohn, and Xiangfeng Zhu. 2022. Netherite: Efficient Execution of Serverless Workflows. *Proc. VLDB Endow.* 15, 8 (apr 2022), 1591–1604. https://doi.org/10.14778/3529337.3529344

[8] Krzysztof Burkat, Maciej Pawlik, Bartosz Balis, Maciej Malawski, Karan Vahi, Mats Rynge, Rafael Ferreira da Silva, and Ewa Deelman. 2021. Serverless Containers – Rising Viable Approach to Scientific Workflows. In *2021 IEEE 17th International Conference on eScience (eScience)*. 40–49. https://doi.org/10.1109/eScience51609.2021.00014

[9] Benjamin Carver, Jingyuan Zhang, Ao Wang, Ali Anwar, Panruo Wu, and Yue Cheng. 2020. Wukong: A Scalable and Locality-Enhanced Framework for Serverless Parallel Computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC '20)*. ACM, 1–15. https://doi.org/10.1145/3419111.3421286

[10] Henri Casanova, Rafael Ferreira da Silva, Ryan Tanaka, Suraj Pandey, Gautam Jethwani, William Koch, Spencer Albrecht, James Oeth, and Frédéric Suter. 2020. Developing accurate and scalable simulators of production workflow management systems with wrench. *Future Generation Computer Systems* 112 (2020), 162–175.

[11] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. 2021. SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing. In *Proceedings of the 22nd International Middleware Conference* (Québec city, Canada) *(Middleware '21)*. Association for Computing Machinery, New York, NY, USA, 64–78. https://doi.org/10.1145/3464298.3476133

[12] Rafael Ferreira Da Silva, Rosa M. Badia, Venkat Bala, Debbie Bard, Peer-Timo Bremer, Ian Buckley, Silvina Caino-Lores, Kyle Chard, Carole Goble, Shantenu Jha, Daniel S. Katz, Daniel Laney, Manish Parashar, Frederic Suter, Nick Tyler, Thomas Uram, Ilkay Altintas, Stefan Andersson, William Arndt, Juan Aznar, Jonathan Bader, Bartosz Balis, Chris Blanton, Kelly Rosa Braghetto, Aharon Brodutch, Paul Brunk, Henri Casanova, Alba Cervera Lierta, Justin Chigu, Taina Coleman, Nick Collier, Iacopo Colonnelli, Frederik Coppens, Michael Crusoe, Will Cunningham, Bruno De Paula Kinoshita, Paolo Di Tommaso, Charles Doutriaux, Matthew Downton, Wael Elwasif, Bjoern Enders, Chris Erdmann, Thomas Fahringer, Ludmilla Figueiredo, Rosa Filgueira, Martin Foltin, Anne Fouilloux, Luiz Gadelha, Andy Gallo, Artur Garcia Saez, Daniel Garijo, Roman Gerlach, Ryan Grant, Samuel Grayson, Patricia Grubel, Johan Gustafsson, Valerie Hayot-Sasson, Oscar Hernandez, Marcus Hilbrich, AnnMary Justine, Ian Laflotte, Fabian Lehmann, Andre Luckow, Jakob Luettgau, Ketan Maheshwari, Motohiko Matsuda, Doriana Medic, Pete Mendygral, Marek Michalewicz, Jorji Nonaka, Maciej Pawlik, Loic Pottier, Line Pouchard, Mathias Putz, Santosh Kumar Radha, Lavanya Ramakrishnan, Sashko Ristov, Paul Romano, Daniel Rosendo, Martin Ruefenacht, Katarzyna Rycerz, Nishant Saurabh, Volodymyr Savchenko, Martin Schulz, Christine Simpson, Raul Sirvent, Tyler Skluzacek, Stian Soiland-Reyes, Renan Souza, Sreenivas Rangan Sukumar, Ziheng Sun, Alan Sussman, Douglas Thain, Mikhail Titov, Benjamin Tovar, Aalap Tripathy, Matteo Turilli, Bartosz Tuznik, Hubertus Van Dam, Aurelio Vivas, Logan Ward, Patrick Widener, Sean Wilkinson, Justyna Zawalska, and Mahnoor Zulfiqar. 2023. Workflows Community Summit 2022: A Roadmap Revolution. https://doi.org/10.5281/ZENODO.7750670

[13] Ewa Deelman, Gurmeet Singh, Miron Livny, Bruce Berriman, and John Good. 2008. The cost of doing science on the cloud: The Montage example. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. 1–12. https://doi.org/10.1109/SC.2008.5217932

[14] Juan J. Durillo, Radu Prodan, and Jorge G. Barbosa. 2015. Pareto tradeoff scheduling of workflows on federated commercial Clouds. *Simulation Modelling Practice and Theory* 58 (2015), 95–111. https://doi.org/10.1016/j.simpat.2015.07.001

[15] Simon Eismann, Long Bui, Johannes Grohmann, Cristina L Abad, Nikolas Herbst, and Samuel Kounev. 2021. Sizeless: Predicting the optimal size of serverless functions. *arXiv preprint arXiv:2010.15162* (2021).

[16] John L. Gustafson. 1988. Reevaluating Amdahl's Law. *Commun. ACM* 31, 5 (May 1988), 532–533. https://doi.org/10.1145/42411.42415

[17] Mika Hautz, Sashko Ristov, and Michael Felderer. 2023. Characterizing AFCL Serverless Scientific Workflows in Federated FaaS. In *International Workshop on Serverless Computing (WoSC '23)*. ACM, Bologna, Italy, 24–29. https://doi.org/10.1145/3631295.3631397

[18] Michael T Heath. 2018. *Scientific Computing: An Introductory Survey, Revised Second Edition*. SIAM.

[19] Marty Humphrey, Zach Hill, Catharine van Ingen, Keith Jackson, and Youngryel Ryu. 2011. Assessing the Value of Cloudbursting: A Case Study of Satellite Image Processing on Windows Azure. In *2011 IEEE Seventh International Conference on eScience*. 126–133. https://doi.org/10.1109/eScience.2011.26

[20] Keith R. Jackson, Krishna Muriki, Lavanya Ramakrishnan, Karl J. Runge, and Rollin C. Thomas. 2011. Performance and cost analysis of the Supernova factory on the Amazon AWS cloud. *Sci. Program.* 19, 2–3 (apr 2011), 107–119. https://doi.org/10.1155/2011/498542

[21] Aji John, Kristiina Ausmees, Kathleen Muenzen, Catherine Kuhn, and Amanda Tan. 2019. SWEEP: Accelerating Scientific Research Through Scalable Serverless Workflows. In *IEEE/ACM International Conference UCC Companion*. ACM, New Zealand, 43–50.

[22] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the cloud: Distributed computing for the 99%. In *Symposium on Cloud Computing*. 445–451.

[23] Gideon Juve, Ann Chervenak, Ewa Deelman, Shishir Bharathi, Gaurang Mehta, and Karan Vahi. 2013. Characterizing and Profiling Scientific Workflows. *Fut. Gen. Comp. Syst.* 29, 3 (March 2013), 682–692. https://doi.org/10.1016/j.future.2012.08.015

[24] Daniel Kelly, Frank Glavin, and Enda Barrett. 2020. Serverless Computing: Behind the Scenes of Major Platforms. In *IEEE International Conference on Cloud Computing (CLOUD)*. 304–312. https://doi.org/10.1109/CLOUD49709.2020.00050

[25] Daniel Kelly, Frank Glavin, and Enda Barrett. 2020. Serverless Computing: Behind the Scenes of Major Platforms. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*. 304–312. https://doi.org/10.1109/CLOUD49709.2020.00050

[26] 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

[27] Heng Li and Richard Durbin. 2010. Fast and accurate long-read alignment with Burrows–Wheeler transform. *Bioinformatics* 26, 5 (2010), 589–595.

[28] Zhuozhao Li, Ryan Chard, Yadu Babuji, Ben Galewsky, Tyler J. Skluzacek, Kirill Nagaitsev, Anna Woodard, Ben Blaiszik, Josh Bryan, Daniel S. Katz, Ian Foster, and Kyle Chard. 2022. funcX: Federated Function as a Service for Science. *IEEE Trans. on Parallel and Distributed Systems* 33, 12 (2022), 4948–4963. https://doi.org/10.1109/TPDS.2022.3208767

[29] Philip Maechling, Ewa Deelman, Li Zhao, Robert Graves, Gaurang Mehta, Nitin Gupta, John Mehringer, Carl Kesselman, Scott Callaghan, David Okaya, Hunter Francoeur, Vipin Gupta, Yifeng Cui, Karan Vahi, Thomas Jordan, and Edward Field. 2007. *SCEC CyberShake Workflows—Automating Probabilistic Seismic Hazard Analysis Calculations*. Springer London, London, 143–163. https://doi.org/10.1007/978-1-84628-757-2_10

[30] Pascal Maissen, Pascal Felber, Peter Kropf, and Valerio Schiavoni. 2020. FaaSdom: A Benchmark Suite for Serverless Computing. In *Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems* (Montreal, Quebec, Canada) *(DEBS '20)*. Association for Computing Machinery, New York, NY, USA, 73–84. https://doi.org/10.1145/3401025.3401738

[31] Maciej Malawski, Adam Gajek, Adam Zima, Bartosz Balis, and Kamil Figiela. 2020. Serverless execution of scientific workflows: Experiments with HyperFlow, AWS Lambda and Google Cloud Functions. *Future Generation Computer Systems* 110 (2020), 502–514. https://doi.org/10.1016/j.future.2017.10.029

[32] Roland Mathá, Sasko Ristov, Thomas Fahringer, and Radu Prodan. 2020. Simplified Workflow Simulation on Clouds based on Computation and Communication Noisiness. *IEEE Transactions on Parallel and Distributed Systems* 31, 7 (2020), 1559–1574. https://doi.org/10.1109/TPDS.2020.2967662

[33] Maciej Pawlik, Pawel Banach, and Maciej Malawski. 2020. Adaptation of workflow application scheduling algorithm to serverless infrastructure. In *Euro-Par 2019: Parallel Processing Workshops: Euro-Par 2019 International Workshops, Göttingen, Germany, August 26–30, 2019, Revised Selected Papers 25*. Springer, 345–356.

[34] Sashko Ristov and Philipp Gritsch. 2022. FaaSt: Optimize makespan of serverless workflows in federated commercial FaaS. In *2022 IEEE International Conference on*

Cluster Computing (CLUSTER '22). IEEE, Heidelberg, Germany, 182–194. https://doi.org/10.1109/CLUSTER51413.2022.00032

[35] Sashko Ristov, Mika Hautz, Christian Hollaus, and Radu Prodan. 2022. SimLess: Simulate Serverless Workflows and Their Twins and Siblings in Federated FaaS. In *2022 ACM SoCC '22: ACM Symposium on Cloud Computing (SoCC '22)*. ACM, San Francisco, CA, USA, 323–339. https://doi.org/10.1145/3542929.3563478

[36] Sashko Ristov, Christian Hollaus, and Mika Hautz. 2022. Colder Than the Warm Start and Warmer Than the Cold Start! Experience the Spawn Start in FaaS Providers. In *Workshop on Advanced Tools, Programming Languages, and PLatforms for Implementing and Evaluating Algorithms for Distributed Systems (APPLIED '22)*. ACM, Salerno, Italy, 35–39. https://doi.org/10.1145/3524053.3542751

[37] Sasko Ristov, Stefan Pedratscher, and Thomas Fahringer. 2021. AFCL: An Abstract Function Choreography Language for serverless workflow specification. *Fut. Gen. Comp. Syst.* 114 (2021), 368 – 382.

[38] Sasko Ristov, Stefan Pedratscher, and Thomas Fahringer. 2023. xAFCL: Run Scalable Function Choreographies Across Multiple FaaS Systems. *IEEE Transactions on Services Computing* 16, 1 (2023), 711–723. https://doi.org/10.1109/TSC.2021.3128137

[39] Sasko Ristov, Radu Prodan, Marjan Gusev, and Karolj Skala. 2016. Superlinear speedup in HPC systems: Why and when?. In *2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*. 889–898.

[40] Josep Sampe, Pedro Garcia-Lopez, Marc Sanchez-Artigas, Gil Vernik, Pol Roca-Llaberia, and Aitor Arjona. 2021. Toward Multicloud Access Transparency in Serverless Computing. *IEEE Soft.* 38, 1 (2021), 68–74. https://doi.org/10.1109/MS.2020.3029994

[41] J. Sampe, M. Sanchez-Artigas, G. Vernik, I. Yehekzel, and P. Garcia-Lopez. 2021. Outsourcing Data Processing Jobs with Lithops. *IEEE Transactions on Cloud Computing* (Nov. 2021), 1–1. https://doi.org/10.1109/TCC.2021.3129000

[42] Biswajeet Sethi, Sourav Kanti Addya, Jay Bhutada, and Soumya K. Ghosh. 2023. Shipping Code towards Data in an Inter-Region Serverless Environment to Leverage Latency. *J. Supercomput.* 79, 10 (mar 2023), 11585–11610. https://doi.org/10.1007/s11227-023-05104-7

[43] Christopher Peter Smith, Anshul Jindal, Mohak Chadha, Michael Gerndt, and Shajulin Benedict. 2022. FaDO: FaaS Functions and Data Orchestrator for Multiple Serverless Edge-Cloud Clusters. In *2022 IEEE 6th International Conference on Fog and Edge Computing (ICFEC)*. 17–25. https://doi.org/10.1109/ICFEC54809.2022.00010

[44] Ion Stoica and Scott Shenker. 2021. From Cloud Computing to Sky Computing. In *Workshop on Hot Topics in Operating Systems (HotOS '21)*. ACM, Ann Arbor, Michigan, 26–32. https://doi.org/10.1145/3458336.3465301

[45] Ali Tariq, Austin Pahl, Sharat Nimmagadda, Eric Rozner, and Siddharth Lanka. 2020. Sequoia: Enabling Quality-of-Service in Serverless Computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC '20)*. ACM, Virtual Event, USA, 311–327. https://doi.org/10.1145/3419111.3421306

[46] Alexandru Uta, Alexandru Custura, Dmitry Duplyakin, Ivo Jimenez, Jan Rellermeyer, Carlos Maltzahn, Robert Ricci, and Alexandru Iosup. 2020. Is Big Data Performance Reproducible in Modern Cloud Networks?. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 513–527. https://www.usenix.org/conference/nsdi20/presentation/uta

[47] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. 2020. INFINICACHE: exploiting ephemeral serverless functions to build a cost-effective memory cache. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20)*. USENIX Association, Santa Clara, CA, USA, 267–282.

[48] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking behind the Curtains of Serverless Platforms. In *USENIX Annual Technical Conference*. Boston, MA, USA, 133–145.

[49] Zongheng Yang, Zhanghao Wu, Michael Luo, Wei-Lin Chiang, Romil Bhardwaj, Woosuk Kwon, Siyuan Zhuang, Frank Sifei Luan, Gautam Mittal, Scott Shenker, and Ion Stoica. 2023. SkyPilot: An Intercloud Broker for Sky Computing. In *Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX, Boston, MA, 437–455.

[50] Minchen Yu, Tingjia Cao, Wei Wang, and Ruichuan Chen. 2021. Restructuring Serverless Computing with Data-Centric Function Orchestration. *arXiv preprint arXiv:2109.13492* (2021).