Experimental Study on Microservices Orchestration with Amazon Step Function

1st Siyang Zhang
Khoury College of Computer Sciences
Northeastern University
Vancouver, BC, Canada
zhang.siyan@northeastern.edu

2nd Chen Qiu
Khoury College of Computer Sciences
Northeastern University
Vancouver, BC, Canada
qiu.chen1@northeastern.edu

3rd Huaxing Wang
Khoury College of Computer Sciences
Northeastern University
Vancouver, BC, Canada
wang.huax@northeastern.edu

4th Imdadul Al Imraan
Khoury College of Computer Sciences
Northeastern University
Vancouver, BC, Canada
imraan.i@northeastern.edu

5th Michal Aibin
Khoury College of Computer Sciences
Northeastern University
Vancouver, BC, Canada
m.aibin@northeastern.edu

***Abstract*—Serverless computing has become widespread in today's tech industries. Compared to traditional cloud computing, it allows developers to build and run applications without managing servers. The Function-as-a-Service(FaaS) as an example of this topic usually requires orchestration to be applied for the application. AWS Step Function is one example. Although FaaS is not the only option for building a service, its orchestration feature provides convenience to developers. However, a major concern is how we know this is the best option to choose compared to other approaches like microservices and event-driven applications. To shed light on this matter, we evaluate the performance of applications built by FaaS orchestrator AWS Step Functions(ASF) for both Express and Standard workflow, event-driven. These four types of applications aim to implement an order booking service with different architectures. Our results evaluate the performance of these four types of workflows, from the experiments we can find the specialty in performance. In Express workflow, the results show that it's suitable for short-term, event-based applications with multiple states, and the performance is greatly affected by Lambda functions within a workflow. As for Standard Workflow, it's suitable for long-running, durable application workflow, its API could offer more detailed information for logging. For the Event-Driven approach, it is easier to implement but not as good as step function in terms of performance and cost, also it is hard to track stage and handle errors.**

***Index Terms*—Microservice Orchestration, Workflow, Serverless, Amazon Web Services, AWS Step Function, Event-Driven AWS**

I. Introduction

Serverless computing, and in particular, the "Function as a Service" (FaaS) paradigm, has taken the industry by storm. AWS(Amazon Web Services) Step Function [1] is the most mature and performant project in the market: According to the validation, ASF appears to be the most efficient service for both short and long-running orchestrations. However, there is a lack of research on the comparison of the pros and cons with other more traditional approaches, like microservice and event-driven applications. Even though a serverless solution is cost-effective and eases resource management, these unclarities could be hindering its wide adoption by potential users. After we dig into the current research papers, we find that there are previous experiments that checked the validity of the proposed models by a limited number of applications deployed on AWS, the research mostly ranges in AWS Functions itself, it lacks comparison with external application architectures. There could be many different cases of implementing FasS services. Thus, our research aims to fill the gap in the range of performance reviews between ASF and other workflow approaches.

To complete our research topic and explore and come up with a solution to our problem, we need to build and deploy an application based on ASF. For the testing application, we planned to come up with a more centralized app that contains enough serverless functions to make sure the testing service itself has enough variables to provide the metrics we are going to need. These potential microservices could be a booking order service, a payment verification service, and a delivery service. Of course, to better optimize our results, we need to carefully design and compare the cost results of different apps and services.

This paper focuses on examining Step Function's [2] role as the orchestrator to manage the workflow and organize multiple AWS services and functions based on demand, especially combined with microservices deployment in the first place. As well as the performance comparison with other approaches including event-driven and microservices.

For this paper, section two is literature review on related works, we reviewed articles about the Microservices Workflow and Serverless Framework and Performance and cost of Serverless Applications. Section three is about the problem statement, we discussed problem background, our own research approach, our objectives and the innovation. Section four is about the application we built, experiment setup, experiment parameters. Section five is experiment and section six is experiment results. Section seven is conclusion.

II. Literature Review on Related Works

A. Microservices Workflow and Serverless Framework

According to the study [3] on architecture patterns for microservices, where it uses a catalog of microservice architecture patterns to analyze the advantages and disadvantages of different patterns based on implementations, it finds that the structure-oriented orchestration/coordination and storage pattern is used to address the concerns on the interaction among components and data management, while the deployment pattern takes advantages of the container or VM deployment to link microservices with the deployment strategies, contributing to the research focus on continuous development, integration, and deployment as DevOps framework. This paper also leaves some open issues such as the comparison study between SOA and microservices on different views and the problems raised especially when trying to employ microservices in a large growing system, which enlightens our study on using Step Function to manage microservices workflows [3].

Another review study [4] on serverless frameworks performs evaluating multiple serverless frameworks based on their performance from software development phases, where it finds two development directions on current frameworks, the first is the necessity of serverless framework to integrate with cloud application management framework to manage applications; the second trend involves the improvement on current serverless frameworks to be more independent of a cloud platform where model-driven application management may help, but it requires the assistance of abstraction mechanism and language to meet the demands. Next, it reveals a series of challenges faced during software development, which inspires our study on how this ideal abstraction framework should be like to be independent of current AWS technologies and Amazon state language.

This paper [5] proposes an Abstract Function Choreography Language(AFCL) for serverless workflow specification to avoid vendor lock-in and limited support for important data flow and control-flow constructs. AFCL is a YAML- based language that supports a rich set of constructs to express advanced control-flow and data-flow.

There are also other similar function orchestration services or function choreographers offered by other cloud providers such as Composer by IBM Cloud® built on Apache OpenWhisk [6] [7], Workflows by Google Cloud [8] and Durable Functions by Microsoft Azure [9]. The survey named "Secure FaaS orchestration in the fog: how far are we?" [10] discussed and compared all those function orchestration services in detail.

B. Performance and cost of Serverless Applications

Research [11] has been done to investigate the benefit of using the AWS step function in terms of cost and performance. The paper introduces experiments to test the 1. effect of lambda function on AWS step functions, the test contains different allocated memory sizes and startup types (warm start and cold start); 2. State transition duration in workflow, there are three parts needed to be measured, the state transition duration, number of states, and execution duration ;3. For the load part, the experiment focuses on the payload size which is the amount of data the AWS step function will be processing; The results show that [11] [12]: cold start latency decreases when lambda memory size increases; AWS step function leads to a lower cost due to less state transition times in express workflow. But there are still some gaps we need to fill [11]: 1. the research did not address the performance difference between using AWS step function application and traditional event based application, or code based application. 2. The application used in the experiment is a data processing pipeline, a more common application (e-commerce) should be tested in terms of the performance and cost.

A blog post from dashbird [13] compared different cheaper function orchestration services provided by AWS themselves such as EventBridge, DynamoDB Streams, and Lambdas. It also mentioned various shortcomings of those services. One major piece of information they missed in the blog post is the performance comparisons of those services.

C. Summary

From our literature review, we find that the current gap is located in the price and cost model of AWS Step Functions, the accuracy and prediction correctness still have space to improve. The lack of granular billing and execution details offered by the providers makes the development and evaluation of serverless applications challenging for AWS users. Therefore, we want to do research on this topic and try to come up with an optimization for the current cost billing model, to provide more information related to billing and orchestrator options for developers.

III. Problem Statement

A. Problem background

For developers, what kind of architecture they chose to build their services depends on many aspects and metrics. Event based applications, API gateway and microservice orchestration are different ways to coordinate actions across a distributed system. Each has pros and cons and is appropriate in different circumstances.

As for event-based applications, there are several advantages: 1) Decentralization: It doesn't need a central authority to keep an eye on an order flow process. It's implicit workflow connections enable independent team collaboration. 2) Performance: Since it has no overhead of using any centralized orchestrator and contains mostly low-level AWS services, its performance is quite satisfactory for most developers. However, there are some cons at the same time, namely:- a) No effective way to monitor the process state. b) Failure handling needs extra efforts to re-heal a failed service due to the implicit connection.

AWS Step Functions(ASF) seems like a cure for event-based applications, but what are the trade-offs of using ASF orchestrator and how much exactly will developer benefit from using orchestrator are still not addressed.

There is little research talking about the performance difference between AWS step function and other approach application. Users often have a hard time figuring out which application approach they should use to maximize their benefit. As either AWS step function or Event-based application or Code-based application will use AWS lambda services (some will also be using the AWS SNS), but the performance and outcome would be different, for AWS step function, the state transition, number of states and execution duration will be the factor affecting the performance.

B. Our research approach

To dive deep into the difference between the four service architectures (ASF express, ASF standard, Event-based, Code-based), we are going to build a typical e-commerce service application as shown in the Figure 1 for each of the architectures. For ASF applications, we also have a detailed comparison between Standard and Express workflows.

Once the applications are ready, we can run many different tests based on the input load, and processing data types to get the metrics we need to optimize the cost model. One major task is to take full advantage of the retry handling of ASF, to find out how much performance it can improve compared to the Native AWS(event-driven) application and Code-based application.

After we have collected enough data, we will try to improve the optimizing algorithms in one of our literature reviews. By continuing their work and coming out with our own optimization, hopefully, our research can fill the gap and solve this issue.

Besides, the former experimental study on the Amazon step function is quite limited, we extend their research on the Amazon step function's performance on data pipeline processing to the microservices orchestration area. We also compare the workflow orchestration application implemented by the Amazon step function with event-based architecture to evaluate the performance of the Amazon step function in a more comprehensive way.

C. Set the aims and objectives

The limitation of the research we discovered in the previous literature review is that some researchers have come up with an approach to improve the cost prediction accuracy. Still, the experiment is based on certain types of applications, so it may not fit all cases. Moreover, the research doesn't provide insights on how to trade off between a limited budget and performance. So the innovation part of our research project is that we are going to combine the comparison of both cost and performance, by optimizing the cost model, we can provide developers with enough data to help them make better choices about their service architecture.

To shed some light on ASF cost models with respect to their performance and cost, we want to conduct a series of experiments using a serverless data processing pipeline application developed as both ASF Standard and Express workflows. By comparing these two different workflows, we hope to figure out the pros and cons of these workflow types and optimize the cost model regarding performance as well. Our final objective is to find out the most cost-effective plan to use AWS step functions. In order to reach the final goal, we will have several sub-goals. First, we need to build an application using the AWS step function and AWS lambda function to conduct the test, the application could take different payloads and mimic real situations such as invalid input data, and error handling. Next, we will start the experiment based on the different parameter inputs. Finally, we will generate the report and conclude the most cost-effective plan for using AWS step functions.

D. Innovation

The innovation part of our research is that our project compares four different types of workflow applications, which aims to implement the same e-commerce service. There are Step Functions in Express workflow, and in Standard workflow, we also have an event-driven application using AWS service and a code-based microservice deployed on EC2. In our literature review, the previous research only focused on comparing step functions so its research doesn't include other types of applications. Thus, we can get the metrics from more different aspects and can dive deep more thoroughly. The second part of innovation is that we will compare the difference in cost and performance among these applications, and analyze the reason which causes these differences.

IV. Experiment Design

A. Application

The application we designed in this research is a typical E-commerce application, which involves three shopping stages: 1) Order Validation, 2) Charging Service, and 3) Shipment Service. The application will be built in four different ways: 1) ASF standard workflow, 2) ASF express workflow, 3)Native AWS (Event-Based), 4) API Gateway pattern (Code-Based).

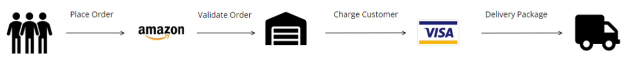

Fig. 1: Application Diagram

B. Experiment Setup

1) AWS Step Function application: Figure 2 displays the architecture of AWS Step Function application Workflow Types: Amazon Step Function has two workflow types: Standard and Express, whose details will be explained within Experiments section below. Both workflow execution types will be implemented and examined in our experiments.

	Application Type	Parameters	Workflow Type	Total Run Times
1	Event Driven	Concurrent Payload		1-1000
		Execution Duration		1-1000
2	Step Function	State Transition Duration	Standard & Express	25
3		Execution Duration	Standard & Express	25
4		Number of States	Standard & Express	25
5		Concurrent Payload	Standard & Express	1-1000
6	Microservice	Concurrent Payload		1-1000

TABLE I: Experiment Setup Parameters

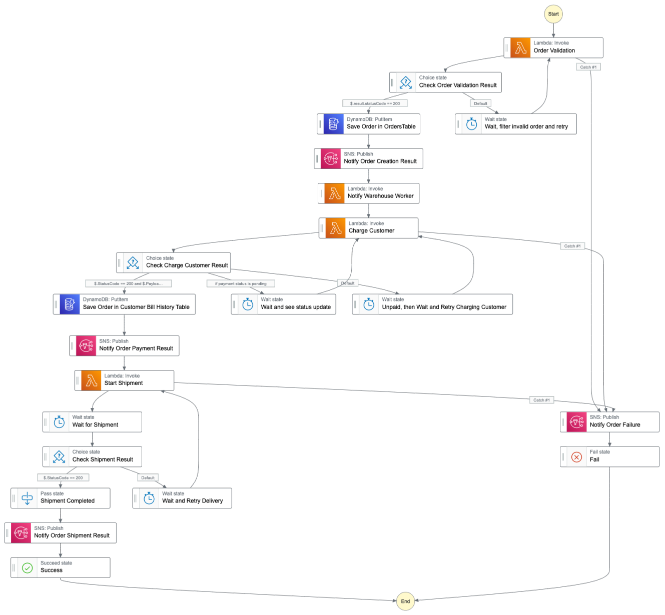

Fig. 2: Step Function Application Diagram

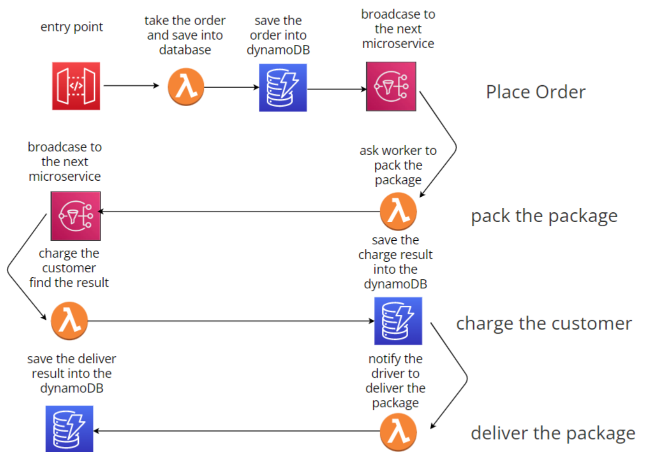
entry point
take the order and save into database
save the order into dynamoDB
broadcase to the next microservice
Place Order
ask worker to pack the package
broadcase to the next microservice
pack the package
save the charge result into the dynamoDB
charge the customer find the result
charge the customer
save the deliver result into the dynamoDB
notify the driver to deliver the package
deliver the package

Fig. 3: AWS microservice workflow diagram

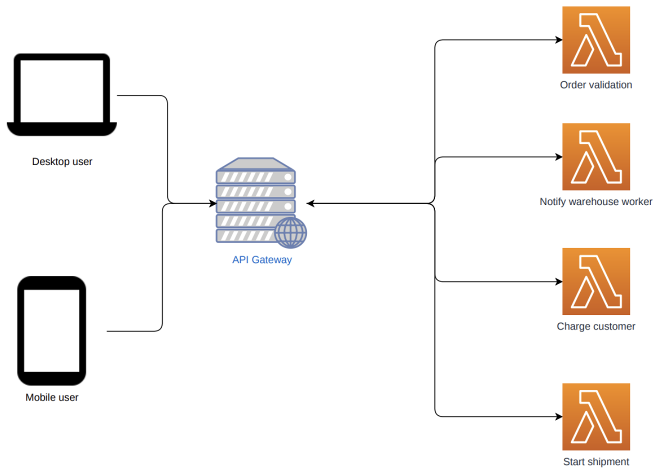

Fig. 4: API Gateway

2) Event-based application: To build the application in Event-based manner, we will need mainly three AWS services: lambda function, SNS and DynamoDB. Lambda function is the core service in each stage, it will process the order for different purposes. SNS is used to broadcast to next microservice stage, it will pass the JSON payload to the next lambda function. DynamoDB is our database which is used to store all the information at each stage. See Figure 3 for details.

3) Code-based application: We have an API gateway written in Golang that will work as an orchestrator. An API gateway is an API management tool that sits between a client and a collection of backend services Figure 4. An API gateway acts as a reverse proxy to accept all application programming interface (API) calls, aggregate the various services required to fulfill them, and return the appropriate result [20].

This API gateway is deployed on Amazon EC2 instance t4g.small which has 2 vCPUs and 2GiB memory. This setup will be tested against a serverless workflow orchestrator namely Amazon Step Function.

C. Experiment parameters

Table I summarizes the six parameters to consider in the experiments discussed below, organized in three distinct categories: parameters related to the effect of lambdas on ASF execution (memory size and startup type), related to the workflow itself (state transition duration, number of states, total execution duration), and finally related to the input load of the application.

1) lambdas: cold starts and warm starts make difference for lambda and ASF performance, since we can not guarantee the lambdas in the warm status period, we will test our workflow in both cold start and warm start.

We could also configure the memory size of the lambda function, which has the range from 128MB to 10,240MB, based on our JSON payload and the operations in the lambda functions, for our research we will set all the Lambda memory size to be the same for better comparison.

2) Workflow: Step Function supports two workflows: Express workflow [1] is ideal for high-volume, event-processing workloads, it has at-least-once asynchronous and at-most-once synchronous execution models. In addition, Express Workflow supports a high-volume execution start rate, nearly unlimited state transitions, and unlimited execution history. Express execution is limited to 5 minutes' maximum duration. Regardless of which workflow is used, ASF execution is affected by the Lambda cold start. When a workflow is first started, all associated Lambdas functions will start with a configured delay, while subsequent workflow execution will reuse the warm-started Lambdas functions. As for the billing model, express workflow is billed by the number of executions, duration of execution, and memory consumed.

Standard Workflows are ideal for long-running, durable, and auditable workflows. They can run for up to a year and you can retrieve the full execution history using the Step Functions API, up to 90 days after your execution completes. Standard Workflows employ an exactly-once model, where your tasks and states are never executed more than once unless you have specified Retry behavior in ASL. This makes them suited to orchestrating non-idempotent actions, such as starting an Amazon EMR cluster or processing payments. Standard Workflows executions are billed according to the number of state transitions processed. For reason that Standard workflows are charged by the total number of state transitions across all state machines, including retries, and as such the number of state transitions should be taken into account for our experiments, we need to investigate how ASF standard workflow behaves when input load changes.

Payload: To evaluate the performance and stress limit, by load size of the initial input of standard workflow

V. Experiments

A. AWS Step Function Workflow

We design the applications to find out the effect of workflow on the cost and performance of Amazon Step Function, the parameters are all listed in Table I. Below is the explanation for each parameters:

Number of States: State can take input from the previous state, perform the task then pass output to next state [19]. In this experiment, we adjust the number of states by merging DynamoDB task states, Simple Notification Service task states, Choice states, and Wait states into Lambda Function task states, while maintaining the same function of the original application, to check the number of states' effect on the performance of ASF workflow.

1) 17 states: Figure 2 displays the original application without adjusting states, including 4 Lambda function task states, 3 DynamoDB task states, 3 SNS task states, 3 Choice states, 1 Wait state, and necessary start, end and success states.
2) 14 states: The workflow deletes 3 choice states and the rest states remain the same.
3) 9 states: The workflow merges 3 DynamoDB task states and 1 Wait state with the corresponding Lambda function within the same component, and the rest are unchanged.
4) 6 states: The workflow merges 3 SNS task states with corresponding Lambda function, the rest are unchanged.

State Transition Time: to evaluate the effects of transition time between states on the cost and performance of ASF workflows. Log data collected from ASF execution logs will be used to measure the execution time for each state and the transition time between states.

Execution Time: to check the execution time's effect on the ASF workflow' cost and performance by comparing Express workflow and Standard workflow.

B. Event-based application

1) build up AWS microservice: As we discussed before, the traditional AWS services will be used to create the application. For the validate order part, the lambda function will first check the input payload (JSON) file, and then check if we have enough inventory left in warehouse or not, this is done by checking with the inventory DynamoDB. Then we will check if the item can be delivered or not, we set certain area that can not be reached. Next, we will update the payload data and save the updated information into the DynamoDB and than process to the next stage. SNS service will be used to broadcast the payload to the charge customer service. In the charge customer service, similarly, we will check the input data first, and then start the charging customer process, and update the payload data, save the updated data into the database and send to the next service via SNS. The last stage is deliver the package and save the final result into DynamoDB to finish the order.

2) performance test: During the busy season (Black Friday), all E-commerce platform will be facing a huge amount of order passing to their app. One of the import performance test is to test how quick the application can process the huge load of data. In order to test this performance, we randomly create different amount of orders (1,100...800,900,1000) to mimic the amount of order per second the application will process. The way to trigger up to 1000 order at once is to use multithreading programming in the lambda function, each thread will be response for taking one order (we can omit the time to create new thread), AWS lambda function will allow up to 1000 concurrent load running at same time. In fact we don't have to take up all 1000 concurrent load resource as the first few orders will be finished before taking more resources. We will start to run all the data to four different approach and get results.

3) cost test: Different application approaches have different cost model, we need to first dive into each one of them in order to have a common metrics to test. For the AWS step function - express, the cost is based on the function duration; it has three threshold: \$0.06 per GB-hour for the first 1,000 hours GB-hours \$0.03 per GB-hour for the next 4,000 hours GB-hours \$0.01642 per GB-hour beyond that. For the AWS step function - standard, the cost is based on state transitions; \$0.025 per 1,000 state transitions. For the Event-based application, the cost is based on the lambda function execution duration, SNS notification times and DynamoDB read and write requests, so the total cost would be measured in a execution duration with the a certain cost of DynamoDB and SNS. To complete the cost between all application approach, the execution duration would be the variable, and the duration we used in the performance test will also be used in this test.

C. *Code-based application*

In this approach, we have an API gateway written in Golang Echo framework. Echo is a high performance, extensible, minimalist Go web framework [21].

This API gateway accepts client requests and calls the same lambdas as ASF workflows, gather results and send response back to clients. We use AWS SDK for Go v2 integrate AWS services with our API gateway [22]. This API gateway implements the same workflows as ASF described in Figure 2. Full source code is available in the github repistory [23].

VI. Experiment Results

A. *Results of Step Function Workflow Experiment*

Number of States: Figure 5 displays the results of the experiment where we adjust the number of states on Express workflow to test its effect on total running time, and findings are listed below:

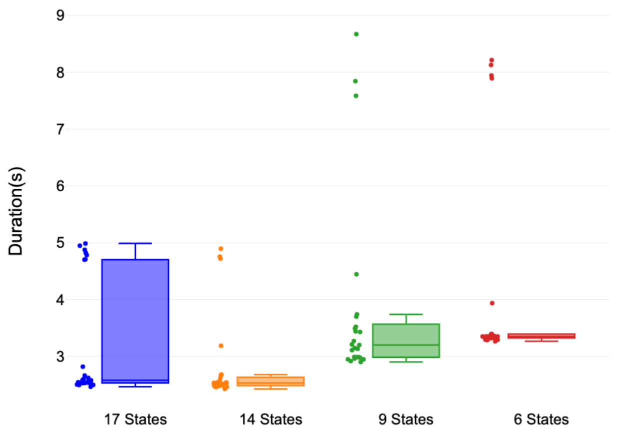

Fig. 5: Number of States Effect on Express Workflow Execution Duration

Figure 6 displays the results of the experiment where we adjust the number of states on Standard workflow to test its effect on total running time, and findings are listed below:

The average running time of the 17-States workflow is 3.1996s, which is slightly slower than the 14-States workflow whose average running time is 2.82488s. It's reasonable since the subtle difference lies in the Choice states, because Express workflow doesn't store state transition information between

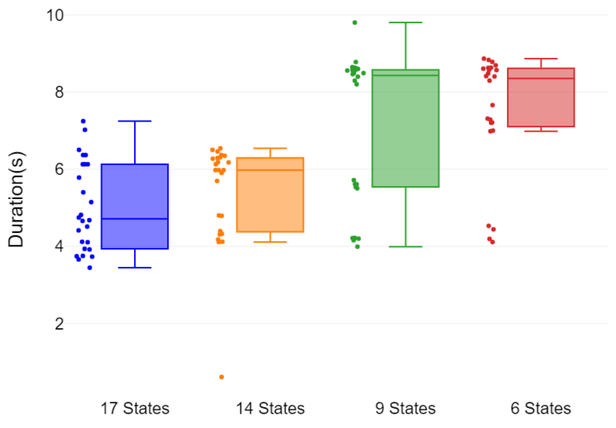

Fig. 6: Number of States' Effect on Standard Workflow Execution Time

states, therefore adjusting the number of Choice states doesn't have much influence on the total execution duration while only contributing to limited improvement in performance.

The average running time of 9-States workflow and 6-States workflow is 3.8224s and 4.11444s respectively, which should be smaller than workflows with more states since merging Task states within the Lambda function reduces the number of states within a workflow, which should save State Transition Time. One potential explanation is that Step Function orchestrates services within workflow more efficiently and reduces the cost of information passing among states, compared with calling other AWS services within the Lambda function.

All four workflow's data layout displays grouping behavior: 17-States workflow's data lies between 2.5s to 3s and 4.5s to 5s, 14-States workflow's data also resides between 2.5s to 3s and 4.5s to 5s ranges with slightly better performance. For 9-States workflow, even though the grouping is not that obvious, it's still clear that most execution time is within the range of 2.5s to 4s while outliers are more than 7s. A similar pattern also exists in 6-States workflow where most executions are within 3s to 4s range while outliers are around 8s. The pattern where data points from all four workflows lie in two major ranges demonstrates the effect of the Lambda function's warm/cold start on execution time.

For the Standard workflow, the average running time of the 17-States workflow is 5.061s, while in 14 states it lasts for 5.3887s. These two numbers are pretty close to each other. The explanation is that choice state will prolong the state transition time, but in practice, this transition time runs fast inside the Standard workflow.

From 14 states to 9 states, the average running time increased to 7.32609s even though the number of states decreased. Theoretically, the running time should be shorter with fewer states. In our experiments, the functions of the states removed are all added to the Lambda function. The possible reason is invoking other AWS services could be more efficient than calling them inside the lambda. We can see from the diagram the pattern for Standard workflow is the fewer the number of states, the more time it runs. The range of each type could differ17-States workflow's run time ranges from 3.5s to 7.5s, and 14-States workflow runs between 4s to 6.5s, there's a subtle increase in running time. 9 states ranges from 4s to 10s while half the data resides around 8.5s. 6 states' run times is also following this pattern.

State Transition Time:

The experiment on State Transition Duration collects the total transition duration for one single execution, and we perform the experiment 25 times to draw Figure 7, Most Express Workflow's state transition time lies in the range from 10ms to 14ms, significantly less than Standard workflow's 947ms average state transition time. The difference lies in the feature of express workflow that it doesn't store state transition information, thus saving the transition time between states.

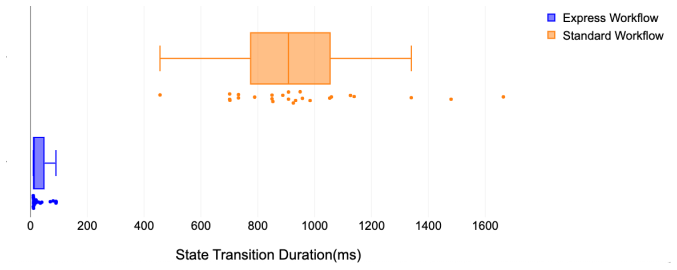

Fig. 7: State Transition Duration between Standard and Express Workflow

Total Execution Time:

According to Figure 8, Express workflow is executed 25 times and the average execution time of Express workflow is 3.199s which is around 37% faster than Standard workflow's 5.061s. Since our application's input is a JSON file to simulate an event call, and each invocation on the application with the given input is fast-complete, besides the idempotent action within States like DynamoDB put operation suits Express workflow's at-least-once model. All the above points align with Express workflow's feature more closely thus explaining the performance difference.

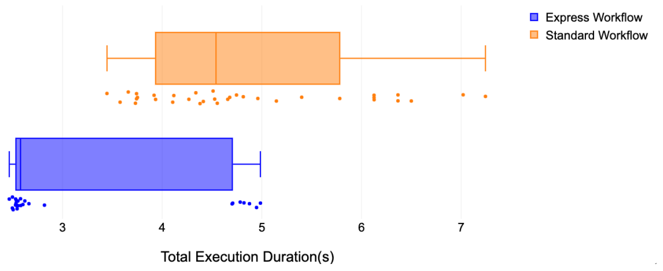

Fig. 8: Total Execution Duration between and Express Workflow

B. *Results of Experiments Cross Four Applications*

Performance Result:

Fig 9 shows the result of our four application total run time in different amount of load. From 1 to 100, all four applications showed the big jump and starts to decrease from 200 to 400, after 400 and 500, Event-based application starts to increase dramatically, Two step function grow steadily. As we can see, from 100 to 200, all four application have a clearly drop, this is most because of the lambda function the lambda function processing is shifting from cold start to warm start, after 400 orders, number of lambda function execution will become the bottleneck of Event-based and Code-based approach, in this case step function has a better performance. Within the step functions, express is better than the standard one.

Cost Result: Fig 10 shows the cost comparison. For the AWS step function standard, the cost is based on state transition, if we conservatively assume that the state transition is 100, then the cost of the application is constant \$0.025/10=\$0.0025. For the AWS step function express, the cost is \$0.0000166667 per second, so we could draw out a line for the express function. For the Event-Based approach, the cost is from three part, lambda(\$0.0000166667 per second), SNS(\$0.5 per million notification), DynamoDB(\$1.5 per million read and write pair), similarly we can draw out the graph as well.

For the API gateway setup, t4g.small EC2 instance is used in our experiment. This instance costs \$0.0168 hourly when rent on demand basis. We use this hourly rate to calculate the cost for different durations.

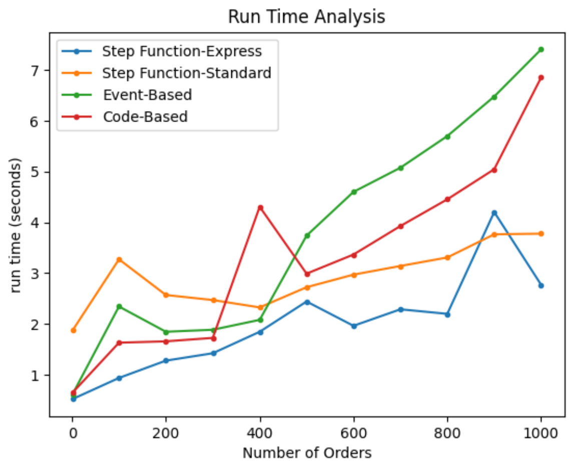

Fig. 9: Total Run Time Analysis

VII. Conclusion

In the experiments between Express and Standard workflow, we find that: adjusting the number of Choice states only contribute subtle improvement in performance, service orchestration with Step Function performs better than invocation of other AWS services within Lambda function, Express

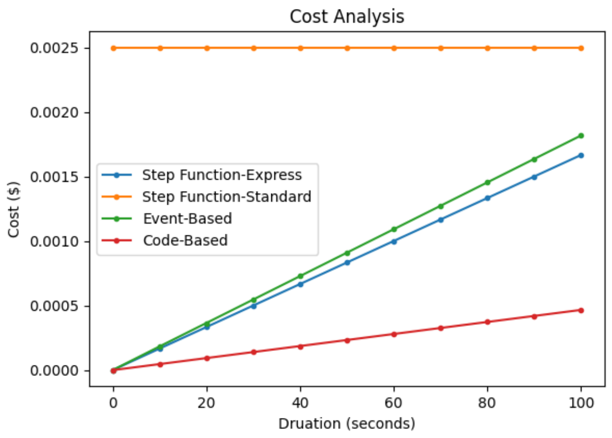

Fig. 10: Cost Analysis

ASF's performance is significantly influenced by the Lambda function' code/warm start within workflow states, Express workflow's feature on not storing state transition information saves state transition time between states compared with Standard workflow. API gateway architecture is the most cost efficient in terms of cost and can be ported across different cloud platforms with ease. Simple cost calculation of EC2 instance compared to ASF also helps developers predict the cost of the services ahead of time since the costing is not tightly coupled with the number of requests like others.

Short-term, event-based application with idempotent operation is more suitable to implement with Express workflow. In Standard workflow, reducing the non-service state only has little influence on the performance efficiency. The cost of Standard workflow is also easy to track and predict. For the Lambda Function state within, the complexity of the function plays a major role regarding the performance. Event-Based application is easy to implement, however, as we discussed before, both performance and cost are not as good as the step function. Most importantly, it is hard to trace where the stage is, and the error handling ability is also the shortcoming of this approach.

For future work on the Express workflow experiment, currently, there is no available backend API to directly export execution logs, therefore further research could add tools to enable automatic export execution logs. Besides, the current application is more aligned with Express workflow's feature, further research could be executed on applications that fit Standard workflow more closely, to see its performance and cost. As for the Standard workflow, we could use the standard API to logging more detailed data to have a further dive into the detail of each experiments we implemented. Also, the current application we are testing contains only limited AWS resources, there are other popular AWS services could have relatively obvious influence on the Step Function. In the future they could also be included as part of our applications.

For future work on the Event-Based application, we could use another DynamoDB+SNS specifically to record the status of each stage, which is once we finish a step we broadcast the status for that step whether it is successful or failed, this would trace where the order stage is. Besides, we could write another sets of lambda function to handle the error encountering during the process, but this can be complicated as lambda function will likely spin in a loop which costs unnecessary resources.

References

[1] AWS Step Functions Developer Guide, URL:https://docs.aws.amazon.com/step-functions/latest/dg/use-cases-orchestration.html (visited on 09/21/2022).

[2] Mukherjee, Sourav. "Benefits of AWS in modern cloud." arXiv preprint arXiv:1903.03219 (2019).

[3] Taibi, D, V Lenarduzzi, and Claus Pahl. "Architectural Patterns for Microservices: A Systematic Mapping Study." CLOSER 2018: Proceedings of the 8th International Conference on Cloud Computing and Services Science; Funchal, Madeira, Portugal, 19-21 March 2018 (2018).

[4] K. Kritikos and P. Skrzypek, "A Review of Serverless Frameworks," 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), 2018, pp. 161-168, doi: 10.1109/UCC-Companion.2018.00051.

[5] Sasko Ristov, Stefan Pedratscher, Thomas Fahringer: AFCL: An Abstract Function Choreography Language for serverless workflow specification, URL: https://doi.org/10.1016/J.FUTURE.2020.08.012

[6] Introducing serverless composition for IBM Cloud functions (no date) IBM. Available at: http://www.ibm.com/cloud/blog/serverless-composition-ibm-cloud-functions (Accessed: November 30, 2022).

[7] Ibm-Functions (no date) IBM-functions/composer: Composer is a new programming model for composing cloud functions built on Apache OpenWhisk., GitHub. Available at: https://github.com/ibm-functions/composer (Accessed: November 30, 2022).

[8] Ibm-Functions (no date) IBM-functions/composer: Composer is a new programming model for composing cloud functions built on Apache OpenWhisk., GitHub. Available at: https://github.com/ibm-functions/composer (Accessed: November 30, 2022).

[9] Cgillum Durable functions overview azure, Durable Functions Overview Azure — Microsoft Learn. Available at: https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview?tabs=csharp (Accessed: November 30, 2022).

[10] Alessandro Bocci, Stefano Forti, Gian-Luigi Ferrari, and Antonio Brogi. 2021. Secure FaaS orchestration in the fog: how far are we? Computing (2021), 1–32.

[11] Philipp Leitner, Erik Wittern, Josef Spillner, Waldemar Hummer, A mixed-method empirical study of Function-as-a-Service software development in industrial practice, Journal of Systems and Software, Volume 149, 2019, Pages 340-359, ISSN 0164-1212, https://doi.org/10.1016/j.jss.2018.12.013.

[12] Varia J. Best practices in architecting cloud applications in the AWS cloud. Cloud Computing: Principles and Paradigms. 2011;18:459-90.

[13] Lin WT, Krintz C, Wolski R, Zhang M, Cai X, Li T, Xu W. Tracking causal order in aws lambda applications. In2018 IEEE international conference on cloud engineering (IC2E) 2018 Apr 17 (pp. 50-60). IEEE.

[14] Mathew A, Andrikopoulos V, Blaauw FJ. Exploring the cost and performance benefits of AWS step functions using a data processing pipeline. InProceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing 2021 Dec 6 (pp. 1-10).

[15] Quinn S, Cordingly R, Lloyd W. Implications of alternative serverless application control flow methods. InProceedings of the Seventh International Workshop on Serverless Computing (WoSC7) 2021 2021 Dec 6 (pp. 17-22).

[16] Cutting Step-Functions Costs on Enterprise-Scale Workflows, URL: https://dashbird.io/blog/cutting-step-functions-costs-enterprise/

[17] Geoffrey C. Fox, Vatche Ishakian, Vinod Muthusamy, Aleksander Slominski. Status of Serverless Computing and Function-as-a-Service(FaaS) in Industry and Research. 2017 Aug.

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

[19] Open University Press. (1975). Step functions. Amazon. Retrieved November 30, 2022, from https://docs.aws.amazon.com/step-functions/latest/dg/concepts-states.html

[20] What does an API gateway do? (n.d.). https://www.redhat.com/en/topics/api/what-does-an-api-gateway-do
[21] Echo - High performance, minimalist Go web framework. (n.d.). Echo - High Performance, Minimalist Go Web Framework. Retrieved December 2, 2022, from https://echo.labstack.com/
[22] AWS SDK for Go v2 GitHub - aws/aws-sdk-go-v2: AWS SDK for the Go programming language. (n.d.-b). GitHub. https://github.com/aws/aws-sdk-go-v2
[23] Imraan, A. (n.d.). GitHub - imraan-go/aws-step-api-gateway. GitHub. https://github.com/imraan-go/aws-step-api-gateway