2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)

2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid) | 979-8-3503-0119-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/CCGRID57682.2023.00053

XFaaS: Cross-platform Orchestration of FaaS Workflows on Hybrid Clouds

Aakash Khochare, Tuhin Khare, Varad Kulkarni and Yogesh Simmhan
Department of Computational and Data Sciences, Indian Institute of Science, Bangalore 560012 INDIA
Email: {aakhochare, tuhinkhare, varadk, simmhan}@IISc.ac.in

***Abstract*—Functions as a Service (FaaS) have gained popularity for programming public clouds due to their simple abstraction, ease of deployment, effortless scaling and granular billing. Cloud providers also offer basic capabilities to compose these functions into workflows. FaaS and FaaS workflow models, however, are proprietary to each cloud provider. This prevents their portability across cloud providers, and requires effort to design workflows that run on different cloud providers or data centers. Such requirements are increasingly important to meet regulatory requirements, leverage cost arbitrage and avoid vendor lock-in. Further, the FaaS execution models are also different, and the overheads of FaaS workflows due to message indirection and cold-starts need custom optimizations for different platforms. In this paper, we propose XFaaS, a cross-platform deployment and orchestration engine for FaaS workflows to operate on multiple clouds. XFaaS allows "zero touch" deployment of functions and workflows across AWS and Azure clouds by automatically generating the necessary code wrappers, cloud queues, and coordinating with the native FaaS engine of the cloud providers. It also uses intelligent function fusion and placement logic to reduce the workflow execution latency in a hybrid cloud while mitigating costs, using performance and billing models specific to the providers based in detailed benchmarks. Our empirical results indicate that fusion offers up to $\approx 75\%$ benefits in latency and $\approx 57\%$ reduction in cost, while placement strategies reduce the latency by $\approx 24\%$, compared to baselines in the best cases.**

I. Introduction

Functions as a Service (FaaS) have gained immense popularity as a means to design elastic and scalable applications on public clouds [1]. FaaS is a micro-services pattern of deploying cloud applications from building-block functions. Here, users provide the logic of a single stateless function to a cloud provider along with its dependencies. This is packaged using a *serverless computing model*, typically encapsulated as a container, deployed on the cloud, and accessed from a service endpoint. Any invocation of this endpoint automatically triggers the creation of a container for the function with its dependencies, if an idle instance does not exist. Once the function executes and the response is returned to the client, the instance is retained for a short timeout period, and if no further requests arrive, it is spun down. Instances are automatically created if the function gets concurrent requests, and shrunk during low demand or idle periods.

Opportunity. The clear benefits of FaaS is driving its rapid adoption in enterprise and science [2]. They offer a simple function abstraction and allow automatic packaging of dependencies for deployment. Scaling to variable demand is effortless, truly leveraging the elasticity of the cloud. They also offer granular billing, where users only pay for the duration for which the function executes. These make the DevOps lifecycle easy to manage. FaaS is offered by all major public cloud providers as a platform service: Amazon Web Services Lambda functions [3], Microsoft Azure Functions [4], Google Cloud Functions [5] and IBM Cloud Functions [6]. Open source projects like OpenFaas [7] and Knative [8] are designed for use in private clouds.

Cloud providers also offer basic capabilities to compose these functions into *workflows* to design practical and complex applications using the FaaS paradigm. FaaS functions can be composed as a dataflow by defining a Directed Acyclic Graph (DAG), either visually or programmatically, and deployed for execution. Internally, these workflows use cloud queues to exchange messages from upstream to downstream functions upon execution. AWS Step Functions [9] and Azure Durable Functions [10] are two such FaaS workflow platforms.

Limitations. That said, there are key gaps in FaaS workflows that *limit their use in hybrid clouds or across data centers*. FaaS and FaaS workflow models are proprietary to each cloud service provider (CSP). This prevents the portability of even the function logic, much less the workflow, across CSPs. This leads to vendor lock-in and prevents leveraging any pricing arbitrage. There are also specialized cloud services that may only be available from certain providers or in some regions, e.g., elastic inferencing accelerators or ARM-based processors, which require the pinning of some functions in the workflow to particular providers or data centers. There is also growing regulatory requirements, like GDPR, around data locality and sovereignty. This mandates the use of data centers in specific regions or CSPs for functions to access certain data. Lastly, the FaaS execution models for the two leading CSPs we investigate, Amazon AWS and Microsoft Azure, are different. This leads to diverse performance characteristics for different workloads. So selecting the right cloud provider for a specific function or for the workflow can be non-trivial.

Challenges. The proprietary nature of FaaS and FaaS workflows requires significant effort to design FaaS workflows that span functions running on different cloud providers or data centers. Even deploying the same function logic to different clouds requires changing the function interfaces, payload objects, configuration files, etc. There are few solutions available to achieve portability even for single functions [11], [12]. Emerging standards like CNCF Serverless Workflows [13] are complex, have low adoption and nascent implementations.

There are also well documented performance limitations of FaaS and FaaS workflows, such as message indirection latency and cold-starts [14], [15]. While research papers attempt to address these bottlenecks using techniques like pilot jobs, function fusion, etc., they require custom optimizations for each cloud platform [16], [17]. The performance variability of FaaS workloads on different clouds also means finding the right way to partition a FaaS workflow across providers is non-trivial. So going from research outcomes to practical performance gains is a challenge.

Contribution. In this paper, we propose **XFaaS**, a cross-platform deployment and orchestration engine for FaaS workflows to operate on multiple clouds. XFaaS allows *"zero-touch" portable deployment of functions and workflows* in hybrid clouds by automatically generating necessary code wrappers for each FaaS provider, creating cloud queues to execute workflows across data centers, and coordinating with the native FaaS engine of a CSP for management and monitoring.

We provide *detailed benchmarks* on FaaS workflow executions on Azure and AWS, and draw insights on their performance for different types of FaaS workloads. This is used to build simple yet practical performance models for these platforms. We leverage this knowledge to propose *intelligent workflow partitioning* and *function fusion techniques* that optimize for lower latency and/or lower cost of deploying FaaS workflows in a hybrid cloud setup. We validate XFaaS for Amazon AWS and Microsoft Azure in this paper, but this can extend to other public, private and edge clouds.

We make the following specific contributions:

1) We describe the architecture of XFaaS, a portable zero-touch framework to auto-generate wrapper code and deploy FaaS and FaaS workflows to hybrid clouds (§ V).
2) We conduct micro-benchmarks to draw insights on the native FaaS platforms from AWS and Azure, and use this to model their performance for diverse workflows (§ IV).
3) We develop workflow partitioning, placement and fusion algorithms that leverage these models to intelligently rewrite the workflow to maximize their performance and reduce costs on multiple clouds (§ VI and VII).
4) We report detailed experiments on two realistic workflows to evaluate the effects of our partitioning planner on the latency. We also use one of these to analyze the impact of function fusion on latency and cost (§ VIII).

Besides these, we discuss related work in § II, offer a background of the FaaS platforms in § III and present our conclusions in § IX. XFaaS is an open source project available on GitHub[1] and provide additional details in the Appendix.

A prior short paper of ours [18] sets out some of the high-level requirements of such FaaS workflows. This paper builds on those early ideas, implements a full system for multi-cloud workflow orchestration, and offers detailed results.

[1]https://github.com/dream-lab/XFaaS

II. Related Work

A. Serverless Workflows in eScience and Enterprises

Serverless workflows have found applications in scientific computing domains like bio-informatics and Satellite image processing [19], [20]. Scientific workflows exhibit stages with a high degree of parallelism, making them ideal candidates for the on-demand scale-in and scale-out enabled by serverless computing. Serverless platforms specific to scientific computing like FuncX [21] have emerged. FuncX supports containerization techniques such as Singularity designed for scientific computing. Its goal is the efficient use of libraries such as MPI that may be installed on scientific computing hardware. Others [22], [23] use a hybrid container plus VM execution approach to balance monetary costs with execution performance. These are platforms can only be deployed in private clouds since such container optimizations are not applicable in a public cloud with proprietary FaaS platform. While these focus on building a platform for optimized execution of scientific workflows using the serverless paradigm, in XFaaS, we focus on leveraging existing public Cloud FaaS Platforms with their associated benefits to enhance the flexibility and portability for developers in a multi-cloud environment, and to improve their performance.

FaaS finds use across diverse enterprise applications with million of requests daily, including web applications, Internet of Things (IoT) backends, Chatbots/Amazon Alexa and IT Automation [1]. IBM describes a vulnerability scanning service for containers that uses a serverless architecture [24].

B. Tooling for Serverless Functions and Workflows

An earlier report [1] highlights the lack of a standard programming model for serverless computing and the vendor lock-in that it may entail as a barrier to adoption. An effort at standardization is Cloud Native Computing Foundation's (CNCF) serverless workflow specification [13] with three runtimes that currently support it – Synapse [25], EventMesh [26] based on Quarkus [27] and Kogito [28] using Kubernetes and MiniKube [29]. These however are designed for containerized use in private clouds. This specification yet to be adopted by public cloud providers for wider impact. XFaaS is designed to leverage public FaaS platforms as they exist, and perform the necessary code generation and deployment across CSPs using a generic specification of functions and workflows defined as a DAG by the user – which can be extended to support the CNCF specification. Several other workflow engines for Kubernetes, such as Knative [8], KubeFlow [30] and Argo [31] can be used to support serverless workflows in a private clouds. In future, XFaaS can be extended to leverage these to span multi-cloud deployments across both public and private clouds.

GoDeploy [11] is a framework for deployment of serverless functions across clouds. User's code is separated into cloud agnostic and cloud specific sections, with the user having to provide handlers for each serverless provider. It also does not support workflow composition from such functions. XFaaS instead auto-generates the wrapper code from just a single cloud-agnostic user function, and supports workflows and their deployments spanning clouds.

Tools such as Terraform [32] and Serverless [33] provide templates to automate FaaS deployments across different providers. However, they do not support code generation and workflows. QuickFaaS [12] is a platform for interoperability between FaaS providers, with objectives similar to ours. However, they too do not for support serverless workflows. While authors of QuickFaaS use entity-relationship patterns to identify a vendor agnostic class structure for Java-based user functions, in practise, there is a lot more code generation that is needed for deploying functions and their workflow compositions onto cloud providers, which XFaaS automates.

C. Optimizations for FaaS and Serverless Workflows

Cold Starts. Cold starts in FaaS are caused by containers being spun up when a bursty workload arrives for a function after a period of quiescence. There have been several solutions proposed for this [34], [35]. Xanadu [15] identifies that cold start overheads cascaded in a workflow execution and therefore have a significant impact on the latency of the execution. They then use branch prediction to speculatively warm up functions along branches of the workflow ahead-of-time. While cold starts impact highly latency-sensitive applications with variable loads, we show later that for FaaS platforms like Azure, the natural variability in inter-function latency outstrips any cold start overheads. However, cold start mitigation techniques are complementary to the optimizations in XFaaS and can be incorporated in future.

Message Indirection. Passing parameters between function invocations in a workflow does not happen point-to-point due to the transient nature of the function containers, and typically requires a cloud queue. This indirection causes a latency overhead between functions, which accumulates in a workflow.

SONIC [36] evaluates three storage-based mechanisms for message passing between functions in a private cloud. They observe that the optimal choice depends on factors like bandwidth, degree of parallelism and data size. FaastLane [17] reduces these overheads by executing functions as threads within a single process, which turns message passing to an in-memory write/read. However, these optimizations are limited to customizable FaaS platforms deployed in private clouds. XFaaS operates with native FaaS platforms on public clouds with little control over these message passing mechanisms.

As we show later, Azure's Durable Functions suffers from overheads during inter-function coordination. Microsoft's Natherite [16] improves the efficiency of inter-function message passing by using EventHubs and Azure Storage Page Blobs that are partitioned to reduce access bottlenecks. But the intrinsic design of Durable Functions still imposes overheads both due to workflow control messages and data exchanges. XFaaS will benefit naturally from such platform enhancements. Our focus is to identify and leverage such optimizations for workflow-level orchestration to operate efficiently across CSPs. Future such enhancement will require XFaaS to rebuild our performance models used for our placement decisions but otherwise will not fundamentally affect framework.

Function Fusion. FaaS functions are designed to be lightweight. However, having functions that are too small can cause the cold-start overheads to dominate the execution time. But finding the "right size" for a function is difficult, and conflates development design with operations tuning. One option is to combine small adjacent functions in a workflow in a single larger function. This also has the benefit of efficient in-memory communication between these functions rather than through a cloud queue.

WiseFuse [37] and others [38] address these performance limitations by proposing fusion and bundling that couple consecutive stages in a workflow DAG for a user-specified latency objective or budget. However, the cost modelling of WiseFuse is incomplete as it only considers the computation cost of functions in the workflow. In practice, CSPs charge for data transfers and the number of functions executed, which will impact the billing cost. XFaaS uses a simpler greedy heuristic for planning function fusion but complements this with a more realistic cost model that is customized to each CSP. That said, WiseFuse's fusion logic can be improved with a better cost model and be used in XFaaS in future.

III. Background

Here, we briefly introduce the two popular FaaS platforms from Amazon AWS and Microsoft Azure used in this paper.

A. Background: AWS Lambda and Step Functions

AWS Lambda functions [3] is the FaaS platform from Amazon Web Services (AWS), the leading cloud provider according to Gartner [39]. Lambda functions can be natively written in Java, Go, Node.js, C#, Python, etc. using a fixed signature provided by Lambda. Functions can then be packaged as a zip file with dependencies, or as a Dockerfile, and uploaded to AWS. Each Lambda function invocation is executed inside a container. However, the same container may get reused across invocations. Users are required to provide the memory requirement for a Lambda function. The number of vCPUs allocated to the function scales with the memory, e.g., Starting from 128MB it starts with < 1 vCPU and this can rise up to 10 GB of memory with 6 vCPUs. The billing is based on the memory allocated and the duration of execution of a function. Lambda functions may be event driven or use polling, and they can be synchronous or asynchronous. E.g., for functions triggered from an API Gateway, the execution is synchronous and event driven. For Amazon's Simple Queue Service (SQS), Lambda functions are triggered through polling for messages that arrive on a specific queue. Lambda functions must be short running and have a maximum timeout of 15 mins.

AWS Step Functions is a FaaS workflow orchestration platform [9]. It can be used to design and deploy dataflows that use AWS Lambda functions as tasks. The workflow can be composed either visually or using the Amazon State Language (ASL). Messages are passed between tasks as AWS event

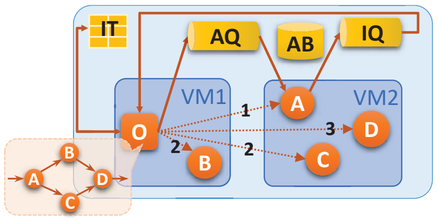


Fig. 1: Sample FaaS workflow deployed using Azure Orchestrator Function.

objects. The exact message-passing mechanism is opaque to the user, though it is likely to be using SQS or EventBridge. The maximum input or output message size for a function in AWS Step Functions is 256 KB.

B. Background: Azure Functions and Orchestrator Functions

Azure Functions is the FaaS offering from Microsoft Azure, the second leading cloud provider according to Gartner [39]. It allows users to develop stateless FaaS, also called *activities*, using .NET, TypeScript, Python, PowerShell, Java, etc. Azure Durable Functions are a "stateful" extension that help compose workflows programmatically using an orchestrator logic. Users write code for an *orchestrator function* (**O** in Fig. 1) that explicitly calls various activity functions (**A–D** in Fig. 1) and passes parameters between them. This mimics the composition, invocation and data flow of a workflow formed from the activities. Azure Durable functions use a *record-replay* mode for orchestration of a workflow execution [40]. The orchestrator creates activity functions and feeds them data using an activity queue (**AQ**) for messages < 64 kB or a Blob store for larger payloads (**AB**), hosted by the Azure storage provider. The orchestrator is woken up by a trigger message that comes on an instance queue (**IQ**) once an activity invocation completes. The orchestrator then replays the history of invocation for this workflow instance from an Azure NoSQL Instance Table (**IT**) to identify the progress done so far, and decides the next activity function to invoke. It then puts the relevant message on that activity queue and updates the replay log table.

Each activity or instance queue is assigned to a single worker VM, and there is internal load balancing across workers when placing messages on different queues. The same worker VM can execute multiple orchestrator or activity instances concurrently. Azure does not require the users to specify a memory requirement for their functions. Each worker VM has a fixed 1.5 GB of memory and 1 vCPU available across all functions it executes. However, the billing is done based on the peak amount of memory actually used by a function and its duration of execution.

IV. A Tale of Two Systems: Performance Analysis and Key Insights

Azure Durable Functions and AWS Step Functions have different performance characteristics for their FaaS workflow behavior. This makes them an interesting study – to offer key insights for DevOps personnel planning manual FaaS workflow deployments, and for us to design performance models to drive automated deployment planning by the XFaaS orchestrator. This complements prior works which have investigated the performance of single functions [41], [42]. However, since cloud providers continuously improve their platforms design, it is necessary to examine their performance characteristics periodically.

We perform detailed micro-benchmarks of diverse functions and small-sized workflows on both Azure Durable Functions and AWS Step Functions to characterize the performance of these two FaaS workflow platforms. Specifically, we run a computer vision workflow, *Object Detection Fanout (ODF)* containing a mix of image analytics functions, such as resize using the OpenCV library and object detection using compact DNN models with fan-out and fan-in. We also run a simple *Linear Chain* workflow with n functions (LC-n), where n varies based on the experiment. Details for these functions and the cloud deployment are provided in § VIII.

1) Azure Durable Functions execute functions faster than AWS Step Functions, but has higher inter-function latencies: Fig. 3 shows the components of the workflow execution times for both LC-8 and ODF when run fully in Azure Durable Functions or AWS Step Functions. Fig. 3 shows the time taken by each function in ODF (dark bars), and the latency between two adjacent functions (light bars), when the workflow is fully run on AWS (orange) or Azure (blue).

As we can see, Azure Durable Functions runs functions faster since a whole VM is shared by all functions and they have access to more compute resources while AWS limits the functions to containers. E.g., in Fig. 2, Azure takes 0.03 sec for the execution of the functions in LC-2 while AWS takes 0.15 sec; this contrast is higher for ODF, with Azure taking 0.2 sec against 0.5 sec for AWS.

However, Azure Durable Functions has higher inter-function times and also a higher variability compared to AWS Step Functions. E.g., in Fig. 3, the error bars for Azure are much higher for its inter-function latencies (light blue bars) compared to AWS (light orange). Here, the design of the Azure orchestrator causes more requests to queues and tables, which has also been identified by Microsoft [16], causing delays. The same orchestrator may also be responsible for multiple workflow instances. On the other hand, AWS Step Functions have a much lower inter-function latency. E.g., Azure takes a total of 4.9 secs for the inter-function latencies in ODF while AWS takes just 3.1 secs.

Despite the client to these workflows executing from VMs present in the same data center, and invoking the HTTP endpoint gateway, Azure exhibits higher initial overheads between the request being triggered to the first function in the workflow being executed, e.g., 0.6 sec for ODF. AWS Step Functions on the other hand has low initial overheads, e.g., 0.2 secs for ODF.

As a result, for workflows with many light-weight functions, Azure is likely to be slower as these inter-function overheads dominate, while AWS will be faster. And for workflows with

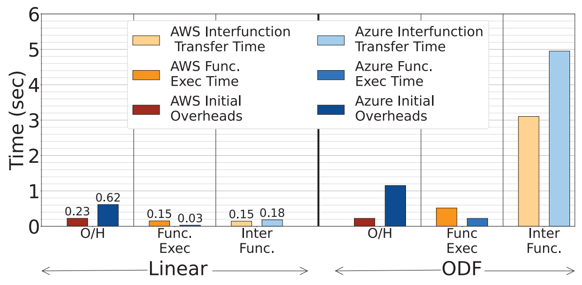


Fig. 2: Time taken for different phases of the workflow: *Initial Overheads*, *Function Execution* and *Inter-function Latency*, for the two workflows on Azure and AWS platforms.

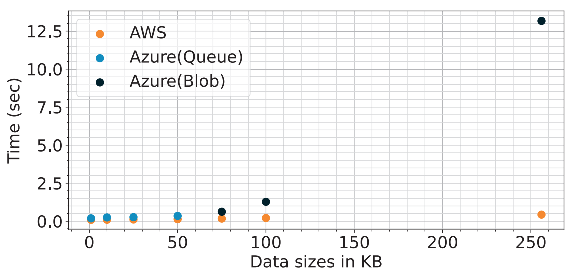


Fig. 4: Scatter plot of inter-function latency against payload size on Azure Durable Functions and AWS Step Functions.

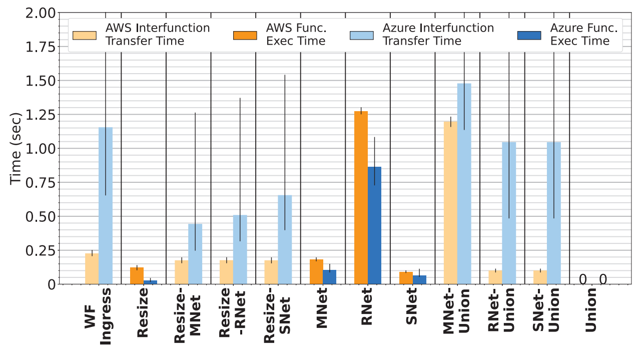


Fig. 3: Latency for different stages of Object Detection Fanout (ODF) workflow execution.

fewer and more resource-intensive functions, Azure is likely to be faster.

2) Azure Durable Functions's message-passing latency grows faster with the payload size than AWS Step Functions, but supports larger payload sizes: Azure Durable Functions and AWS Step Functions internally use their Cloud queues to transfer messages between adjacent functions in the workflow, with the payload sizes limited to 64 kB and 256 kB, respectively. But Azure Durable Functions also allows larger payloads to be sent between functions using their disk-based Blob storage, and this is transparent to the end-user.

We measure the inter-function latency by running the linear chain workflow with different payload sizes being passed between functions. As Fig. 4 shows, the latency for message passing using queues is slower for Azure than AWS for comparable message sizes. For messages larger than 64 kB, Azure shows a higher relative latency and more variability due to disk-based transfers. Hence, while Azure relaxes the message size constraint compared to AWS, it has higher inter-function latencies.

3) AWS Step Functions exhibits consistent cold-start delays while Azure Durable Functions's cold-starts are dwarfed by workflow initialization overheads: A key benefit of FaaS is their automated elasticity, where additional container/VM resources are auto-provisioned by cloud providers when the number of function requests increases, and scaled down to zero when the function is not invoked beyond a timeout duration. This can cause cold-starts, as discussed above. This provisioning time is observed as part of the latency for that function execution. For workflows, such a cold-start accumulates for each downstream function when the workflow is executed afresh after the timeout period.

In AWS Step Functions, every function runs in its own container while in Azure Durable Functions, all functions for a workflow share the same pool of workers (VMs). As a result, it is easier to discern the cold-start overheads in AWS. We deploy the ODF workflow on AWS with the function timeouts set to 5 mins, and invoke the workflow every 4 mins and every 6 mins. The end-to-end latency for the workflow is plotted in Fig. 5 for both these request rates.

When requests arrive every 4 mins, the functions' timeouts have not yet elapsed and the existing (warm) containers execute the new requests within, e.g., 1.5 secs, without additional latency – other than the very first request to the workflow which takes 11.7 secs. At 6 min intervals between requests, the cold-start overhead is seen for every workflow invocation and their constituent functions since all prior containers have spun down after 5 mins. This shows a steadily high latency at 11.8 secs.

We also notice a higher cold-start overhead if the function's deployment size on disk is larger, e.g., taking ≈ 500 ms with a 100 kB package size for the Resize function, and ≈ 2000 ms for a 120 MB package size for the ResNet inference function.

Many sophisticated techniques have been proposed at the OS, container and application levels [34], [43] to handle cold-starts. The deterministic behavior of AWS makes it amenable to some of the application-level mitigation techniques while the high latency and variability of Azure's workflow initialization masks the addition impact of cold starts. That said, the downside of cold-start is limited to highly bursty workloads while also requiring strong latency guarantees for every workflow invocation.

4) AWS Step Functions exhibits consistent elastic scaling: Each warm container in AWS Step Functions can only execute one function at a time. As the number of concurrent function executions increase, we expect the number of containers to grow. The expected number of active containers at a given time is calculated as *Function execution duration* $\times$ *Rate of requests*. Fig. 6 shows the expected number of AWS containers (X axis) using this formula, against the observed number. For

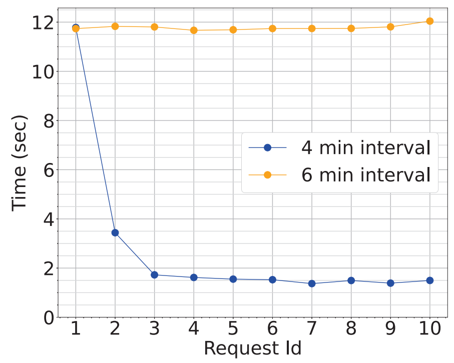


Fig. 5: End to end latency for ODF workflow on AWS Step Functions with timeout of 5 mins, when executed at intervals of every 4 mins and 6 mins.

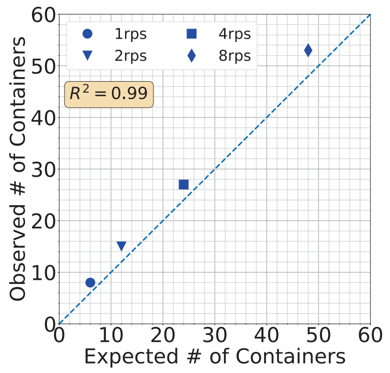


Fig. 6: The number of observed and expected AWS containers at different execution rates of a function.

this experiment, we make requests at $1, 2, 4, 8$ rps for a duration of 5 mins each to a *No-op* function which has an execution duration of 6 secs. As we see, there is a strong correlation ($R^2 = 0.99$) between the two at different invocation rates and this indicates a deterministic scaling behavior.

Discussion. These results indicate that AWS Step Functions in general has lower overheads and more deterministic behavior compared to Azure Durable Functions, while the latter has faster function execution times and supports larger payloads. Both have comparable costs. If forced to pick just a single FaaS platform for workflows, AWS Step Functions comes out ahead. However, there are enough benefits for Azure Durable Functions to prove competitive for some parts of a workflow, which can be leveraged when intelligently splitting and deploying a workflow to span both these clouds. This is also beneficial when data locality constraints and governance policies require running certain functions of the workflow on specific cloud data centers. Lastly, some of the Azure Durable Functions overheads can be mitigated using techniques like function fusion that we discuss in § VI.

Later, we use these and other detailed micro-benchmarks to model the expected execution time of functions on Azure Durable Functions and AWS Step Functions, and the inter-function latencies for given message sizes. These help make intelligent placement, partitioning and fusion decisions.

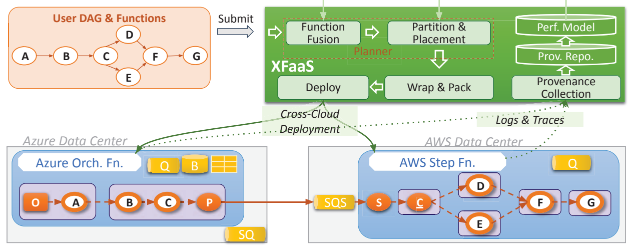


Fig. 7: XFaaS Cross-cloud Deployment Architecture

V. XFaaS Workflow Orchestrator

Fig. 7 shows the architecture of our proposed *XFaaS Workflow Orchestrator*. The core capability of XFaaS is to take a set of user functions defined using a generic function signature along with a workflow defined as a DAG based on these functions, and generate glue functions, wrapper code and packaging for the functions and the DAG specific to a CSP's FaaS and workflow platform, and deploy the DAG with this native FaaS workflow orchestrator. The functions of a workflow may also be placed across different public cloud providers, with all adjacent functions within CSP forming a sub-DAG – how such placement decisions are made intelligently is discussed in future sections. XFaaS handles creating queues, glue functions and other wiring for cross-cloud operations, and deploying the sub-DAGs with the native FaaS workflow platforms of the respective CSPs. Intra-cloud workflow orchestration is managed by the native FaaS provider, and XFaaS is not actively involved at runtime other than for monitoring. This prevents it from becoming a scalability bottleneck.

There is a single instance of the XFaaS platform hosted on any public Cloud VM and this can support any number of workflow deployments. For simplicity, XFaaS is configured with a single set of user credentials and permissions for each CSP, and these credentials are used to deploy all FaaS functions and workflows. Currently, we support Python-based functions due to their popularity in FaaS applications, but a similar approach can be taken for other languages as well.

Next, we describe the steps for deploying a single workflow and its functions within any single CSP using XFaaS.

1) User Functions: To provide compatibility across FaaS providers, XFaaS defines a common and simple function signature for the user to implement their function logic:

$$\textit{function}(\texttt{XFaaSObject}) \rightarrow \texttt{XFaaSObject}$$

Currently, we support HTTP or queue-based means of invocation that is common across FaaS providers. The `XFaaSObject` exposes a `getPayload()` method to return the function's request parameters as a JSON, which arrives over some transport such as HTTP or a cloud queue. In addition, it has metadata fields for performance telemetry tracked by XFaaS. At the end of the user's function logic, they will create a similar `XFaaSObject` whose payload is populated with the JSON for the output from the function.

At deployment-time, XFaaS auto-generates a *runner code* for each function specific to the CSP where it is being deployed. This serves as a wrapper around the user function, and extracts the request payload from the appropriate message transport for the deployment, such as a HTTP Request object or the relevant cloud message object, and places it in the `XFaaSObject` before invoking the user's function. It also populates the metadata fields from the relevant transport object. Similarly, it will access the payload present in the output `XFaaSObject` from the function invocation for downstream processing.

While this signature works for a linear chain of functions, we also support fan-ins (joins) where multiple upstream functions can pass a message to one downstream function. For this, we provide an alternate function signature for users to implement:

$$\texttt{\textit{function}(XFaaSListObject)} \rightarrow \texttt{XFaaSObject}$$

The XFaaS runner code we generate will automatically populate the `XFaaSListObject` with the `XFaaSObjects` received from the upstream edges and make it available to the user logic through the `getPayloads()` method.

Each function is self-contained within its own sub-directory, and this contains the Python entry point file implementing the function signature, any dependent data files, model files, code and/or config files, and a file with the list of external Python packages to import when instantiating a deployment. As part of their execution logic, the user function may access such files that are deployed. The FaaS deployments of CSPs host these files in diverse base folders within which the user functions run, e.g., a relative path in AWS Step Functions or in a path under the root folder for Azure Durable Functions. To make access to these user files transparent, the `XFaaSObject` and `XFaaSListObject` expose a `getBasepath()` method which returns the custom directory path for the CSP where the dependencies are hosted.

A sample user function defined in XFaaS is given below:

```python
def function(inputXFaaSObject):
    # Implement user logic here
    # access inputXFaaSObject.getPayload()
    # access inputXFaaSObject.getBasepath() -> data
    files
    # process payload
    # place output in outputXFaaSObject.setPayload()
    return outputXFaaSObject
```

2) Workflow Description: The workflow can be defined in XFaaS as a DAG specified in JSON. It contains a `WorkflowName`, `Nodes` representing functions, and `Edges` denoting the dataflow dependency between functions. Each `Node` specifies a user-friendly `NodeName`, a globally-unique `NodeId`, a local `Path` to the user code for that function's deployment, the `EntryPoint` Python file that implements the user function signature, the expected peak memory usage for the function as `MemoryInMB`, and optionally the *CSP* where the function must be pinned to. The `Edges` are an adjacency list from a source node to sink nodes. The default dataflow convention is for fan-outs/forks (nodes with > 1 out-edges) to duplicate their `XFaaSObject` output from an execution along all out-edges to sink nodes, and for fan-ins/joins (nodes with > 1 in-edges) to have the upstream task output to be assembled in a `XFaaSListObject` list and provided to the sink node for execution.

A sample LC-2 DAG can be defined in XFaaS as below:

```json
{
  "WorkflowName": "XFaaSUserDagName",
  "Nodes": [
    {
      "NodeName": "func1",
      "Path": "examples/complex-dag-1/src/func1",
      "EntryPoint": "func1.py",
      "MemoryInMB": 256,
      "NodeId": 1
    },
    {
      "NodeName": "func2",
      "Path": "examples/complex-dag-1/src/func2",
      "EntryPoint": "func2.py",
      "MemoryInMB": 512,
      "NodeId": 2
    }
  ],
  "Edges": [
    {
      "func1": ["func2"]
    }
  ]
}
```

3) Packaging and Deployment: When deploying a workflow and its dependent functions to a CSP, the user provides JSON of the DAG and a zipped file containing the sub-folders hosting the user functions and their dependencies, as listed above. The XFaaS platform exposes a REST endpoint where the users can submit this DAG JSON and the functions zip file. The XFaaS Planner analyzes the DAG definition and function descriptions to determine a *deployment plan* based on the latency, cost and function pinning for the workflow. This decides the placement of each function on specific CSPs. The planner may rewrite the DAG structure as part of function fusion, and decide to split the DAG into two parts with each running on a different CSP. These are described in the next sections.

The output of the planner is one or more (sub)DAGs along with the CSP on which each has to be deployed on. For each DAG, XFaaS *generates automatic code* for the runners that wrap functions in the DAG that is specific to the CSP where it will run. This includes code for marshalling and unmarshalling of the native FaaS platform function inputs to the XFaaS input object type to be consumed by the user function, and generating code for provenance and telemetry. The user implemented function and it's dependent Python modules are converted into a single Python file using the *stickytape* tool [44]. There is also workflow level code that is generated for each CSP. This can be ASL to specify AWS step functions or Python code generated for the Azure orchestrator to call the relevant functions in the DAG.

Post the code generation, the native Command Line Interface (CLI) commands of the relevant CSP are invoked to *deploy the functions and the DAG(s)* using their native FaaS platform. Once deployed and available, XFaaS returns the unique HTTP endpoint created for that DAG by the native FaaS workflow for clients to submit requests to. Subsequently, the XFaaS platform is not actively involved in orchestrating the execution of each invocation of the deployed workflow. This is managed completely by the native CSP FaaS workflow platform.

In the case of workflow plans that *span multiple clouds*, there are a few additional steps. We create a cloud queue for each edge-cut in the DAG that spans two CSPs. This queue is created in the CSP hosting the sink function, and used by the source function to send its execution output to the sink. To make this transparent to the users, we automatically create code for a *Push-to-Queue Function* that takes the output of the upstream function and puts it on the cloud queue, and a *Queue Triggered Function* that subscribes to this queue and sends the received payload to the downstream function. These two functions are respectively added to the sub-DAGs deployed to the two CSPs.

Along with the function response payload being passed between functions in the workflow, we also attach *telemetry metadata* to the `XFaaSObject` object as we move through the DAG execution. This contains the workflow invocation ID, the workflow start timestamp, the payload timestamp at the source and sink functions, and the start and end timestamps for every function execution. This metadata is also published to a *provenance store*, currently maintained using a DynamoDB NoSQL store. The telemetry metadata is published as messages to a cloud queue at the end of the workflow's last function's execution by the runner code we generate. An XFaaS service subscribes to this queue and inserts the message into the Workflow Invocation Table of the NoSQL store. This NoSQL store also maintains details of the user-submitted DAG, the refactored DAG after function fusion (discussed next), and deployment details of the DAG and its sub-DAG(s) on one or more CSPs. This metadata is useful for performance monitoring and debugging of the workflow and its functions, and also helps develop the performance models that we use later.

VI. Function Fusion Planner

A user may submit a workflow where each function is an atomic unit of logic that is easy to implement, modular and tractable. However, the complex billing and platform internals of FaaS workflow platforms can cause a user provided DAG structure to be sub-optimal for the CSP platform – on cost and/or latency for executing the workflow. We adopt function fusion in XFaaS to mitigate these effects. *The goal of function fusion is to minimize function latency without exceeding a user-defined cost budget.* The fusion planner greedily and incrementally attempts to fuse adjacent nodes in the user-provided DAG such that each fusion either reduces the estimated latency or the estimated cost for the workflow execution. Once the fusion plan is decided, we generate automated code at deployment time for each *fusion function* that sequentially executes the series of component functions being fused. This single fusion function replaces the fused functions in the DAG structure.

1) Impact of Fusion on Latency: Fusing *linear function chains* on the critical path of the DAG will reduce the latency of the workflow by avoiding inter-function data transfer overheads. However, since fused functions are executed sequentially by the generated fusion function, the fusing of task-parallel functions in the DAG can increase the execution latency due to reduced task concurrency. The magnitude of latency gains depends on the ratio of computation to data transfer overheads of the functions being fused.

2) Impact of Fusion on Cost: When functions are fused, the memory requirement of the fusion function will be the maximum among the memory requirement of the fused functions. This, coupled with the longer execution time of the fusion function relative to its individual constituent functions, can lead to an increase in the cost for execution. However, fusion also leads to a decrease in the number of functions in the DAG, leading to fewer steps in the case of AWS Step Functions and a decrease in the storage requests in Azure Durable Functions. This can cause the overall cost of the fused DAG to decrease since fewer control operations are billed.

3) Planning Algorithm: The nuanced interplay between latency and cost implies that the decision of which functions to fuse cannot be made naïvely. Alg. 1 shows the pseudo-code for our proposed *function fusion algorithm*. The algorithm iteratively and greedily returns the fusion candidate that minimizes the estimated latency without exceeding a user-defined cost threshold.

It accepts as input the current DAG G, which may already have fused nodes, the original user-submitted DAG G^{user}, and the cost threshold θ. Here, θ is the maximum factor by which the fused DAG can exceed the estimated billing cost compared to the original user submitted DAG. We define $\Theta = \theta \times \textsc{BilledCost}(G^{user})$ as the maximum allowed billing budget after fusion. We start by enumerating all *function fusion candidates*, which are nodes that are either a part of a linear chain, or the nodes that lie between a fan-in and fan-out node in the DAG. A fan-out node is a node with an in-degree of 1 and an out-degree > 1, while a fan-in node has an out-degree of 1 and an in-degree of > 1. $\textsc{GetFusionCandidates}$(G) (Alg. 1, Line 17) returns all such fusion candidates[2].

For each fusion candidate f, we evaluate its improvement in workflow latency as, δ_t^f = (Latency before fusion − Latency upon fusion). We also calculate the increase in cost due to this fusion candidate as, δ_c^f = (Cost with fusion − Cost without fusion). The time and cost estimates come from

[2]Currently, only DAGs with a matching pair of fan-in and corresponding fan-out nodes are supported.

Algorithm 1 Function Fusion Planning

```
function FUSE_GRAPH($G$,$G^{user}$, $\Theta$)
    $candidates \leftarrow$ GETFUSIONCANDIDATES(G)
    for $f \in candidates$ do
        $\delta_t^f \leftarrow$ (Latency with fusion $f$ − Latency without fusion)
        $\delta_c^f \leftarrow$ (Cost with fusion $f$ − Cost without fusion)
        $f.score \leftarrow \frac{\delta_t^f}{\delta_c^f}$
    end for
    $sorted \leftarrow$ SORTDESCENDINGBYSCORE($candidates$)
    for $candidate \in sorted$ do
        $fusedG \leftarrow$ FUSEGRAPH($G, candidate$)
        if BILLEDCOST($fusedG$) $< \Theta$ then
            return $fusedG$
        end if
    end for
end function

function GETFUSIONCANDIDATES(G)
    $candidates \leftarrow$ [ ]
    for $(u, v) \in$ DFSTRAVERSAL($G$) do
        if $type(u, v) = LINEAR$ then
            $candidates \leftarrow (u, v)$
        end if
        if $type(u, v) = FANOUT$ then
            $stack.push(u)$
        end if
        if $type(u, v) = FANIN$ then
            $u' \leftarrow stack.pop()$
            $candidates \leftarrow (u', v)$
        end if
    end for
    return $candidates$
end function
```

simple regression models discussed next. These fusion candidates are then sorted in descending order by their *score*, given as the ratio of the time benefit over the cost increase. The candidate with the largest score that does not exceed Θ is selected (Alg. 1, Line 11). If no candidate is returned, the planner stops.

An example of function fusion in action is shown in Fig. 8. The four fusion candidates for the DAG are enumerated. Here, *Candidate 1* has been selected for fusion based on the score and threshold. Nodes A, B, C, and D, which form *Candidate 1*, have been fused into a fusion node *FC1*. An edge has been added from *FC1* to node E. After adding the synthetic node, the structure of the graph has been changed, and we apply the fusion algorithm again on this DAG, until we are not able to find any suitable candidate.

The *time complexity* for evaluating function fusion for a DAG with V nodes and E edges is as follows. Enumerating all fusion candidates takes $\mathcal{O}(E)$, followed by finding the shortest path which takes $\mathcal{O}(V + VlogE)$. Finding the best fusion candidate has a time of $\mathcal{O}(V - 1)$, since we have at most $V - 1$ candidates. So the complexity for fusing one candidate is $\mathcal{O}(E + (V + ElogV) + V)$. This can repeat a maximum of $V - 1$ iterations. This gives a worst case time complexity of $\mathcal{O}(EV + V^2 + EVlogV)$. While this may appear large, given that the vertex and edge counts for DAGs are modest at

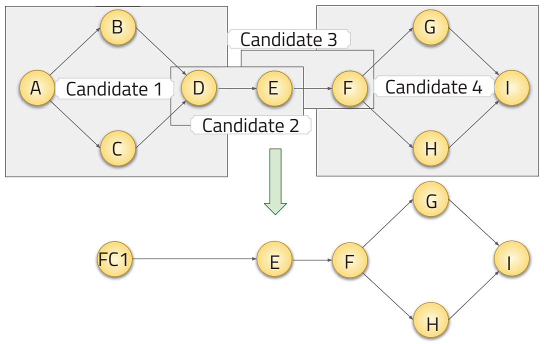


Fig. 8: Identifying fusion candidates, estimating benefits and applying fusion to get the updated DAG. $FC1$ is the fusion function resulting from the fusing of A, B, C and D.

< 100, the wallclock time taken for this fusion logic to run is < 1 *sec*.

4) Latency and Cost Models: The latency and cost estimates come from simple *performance and cost models* we develop based on our micro-benchmarks in § IV. In an operational setting, we can leverage the performance monitoring by the provenance store to build such models dynamically for new workflows. While the accuracy of these models is not high, these are *good enough to select between several choices* for fusion, and later, partitioning and placement.

The function execution duration t is maintained in the Provenance Store either from prior executions of the function or from benchmarks. The Provenance Store also contains the output message size μ for each function. For the *inter-function latency* on AWS Step Functions, if the message size is < 64 KB, we use a constant value of 0.115 sec, and if the message size is ≥ 64 KB, we do a linear regression fit over the message size μ to get the latency as $0.0014 \times \mu + 0.071$, with $R^2 = 0.84$. For the inter-function latency on Azure Durable Functions, we fit the equation $0.12 \times v \times r + 0.418$, where v is the number of nodes in the DAG and r is the request rate (in requests per second). The inter-function latency for Azure Durable Functions depends on the number of concurrent request to the storage tables. $2 \times v \times r$ is an estimate for these concurrent requests. The number of nodes v are available in the user DAG and a service can be used to monitor the request rate. The total execution time for the DAG is given by the sum of the function execution duration and the inter-function latency of the functions that fall on the *critical path* in the DAG.

The *billing cost* for each function invocation using AWS Step Functions in US ¢ is given by $m \times t \times 0.00166667 + 0.00285$, where m is the memory usage of the function in KB and t is the execution duration of the function. ¢0.00285 is the cost for each step transition (function execution). The corresponding cost model for each invocation of an Azure Durable Function is $m \times t \times 0.0016 + 0.000786$. ¢0.000786 is the estimated storage cost for messages transferred.

5) Implementation: We run the fusion algorithm on the user provided DAG to get the new DAG. If it contains a fusion node, we *generate code for a proxy function* for it, which will execute the user-functions of the relevant fused tasks sequentially. The module dependencies for the fused functions are merged together by concatenating their package dependencies. Dependencies for the fused functions are placed under separate sub-directories and the runner template ensures that the *basepath* are appropriately populated for these functions. The memory required for the fusion node is set as the maximum of the memory requirements of the fused nodes. A sample auto-generated fusion function code to fuse *TaskA* and *TaskB* functions is shown below.

```python
def function(inputXFaaSObject):
    # Auto generated code. Do not edit.
    xjaq = TaskA.function(inputXFaaSObject)
    xjaq.set_basepath(basepath_modify(
    inputXFaaSObject.get_basepath(), "TaskA", "TaskB
    "))
    awch = TaskB.function(xjaq)
    return awch
```

VII. Partitioning and Placement Planner

When the user submits a DAG, the XFaaS *partition and placement planner* optionally partitions the DAG and decides the placement of the functions of the DAG (or the sub-DAGs, if partitioned) in the appropriate CSP. XFaaS currently supports partitioning the DAG into two parts, though this can be extended to more than two partitions, across additional CSPs or different data centers of the same CSP.

The input to the partitioner is the user DAG, the estimated execution latencies for each function, and the inter-function or inter-cloud transfer latencies. These values are taken from the cost and latency models above. Users may also provide *constraints* to pin specific functions to particular CSPs, e.g., due to regulatory reasons. Based on these, the partitioner picks the best possible partition point along in the DAG with the CSPs where each part should be placed. The partitioner may also return a single CSP where the entire DAG has to be placed. *The goal of the partitioner is to partition and place the DAG such that the end to end latency for the workflow is minimized.*

Alg. 2 shows the partitioning and placement algorithm. Given a user DAG and constraints on placement, the algorithm starts by initializing the expected latency to be the smaller of the latency from placing the DAG fully in Azure or in AWS. Then, it finds all *valid partitions* for the DAG. A valid partitioning point is after a vertex that is not a part of any fan-in and fan-out pairs [3]. For each such partitioning, the algorithm iteratively tries placing the two sub-DAGs on the two service providers we consider, as long as the user-defined constraints permit such a placement. For each valid partition, a *local best* placement is identified such that the latency is smaller than the previous best latency, and iteratively, we arrive at a *global best* placement.

The time complexity for this algorithm is given by enumerating all partition points ($\mathcal{O}(V)$); calculating latency using the shortest path for both the sub-DAGs ($\mathcal{O}(V+ElogV)$); picking the best sub-DAG out of two ($\mathcal{O}(1)$); to give a total complexity when iterating over all V partition points as $\mathcal{O}(V^2+EVlogV)$. This again is practically manageable for even DAGs with 100s of vertices and edges within 1 sec.

[3]Currently, XFaaS cannot process task-parallel functions that are across multi-clouds

Algorithm 2 Partitioning and Placement Algorithm

```
1:  function PartitionGraph($G^{user}$, $Constraints$)
2:      $l_1 \leftarrow$ GetLatency($AWS\_Only, G^{user}$)
3:      $l_2 \leftarrow$ GetLatency($Azure\_Only, G^{user}$)
4:      $global\_best =$ PickLocalBest($l_1, l_2$)
5:      $best\_partition\_point \leftarrow \varnothing$
6:      $valid\_parts \leftarrow$ GetValidPartitions($G^{user}, Constraints$)
7:      for $p \in valid\_parts$ do
8:          $l_1 \leftarrow$ GetLatency($AWS\_to\_Azure, p$)
9:          $l_2 \leftarrow$ GetLatency($Azure\_to\_AWS, p$)
10:         $local\_best =$ PickLocalBest($l_1, l_2$)
11:         if $local\_best < global\_best$ then
12:             $global\_best \leftarrow local\_best$
13:             $best\_partition\_point \leftarrow p$
14:         end if
15:     end for
16:     return $global\_best, best\_partition\_point$
17: end function
```

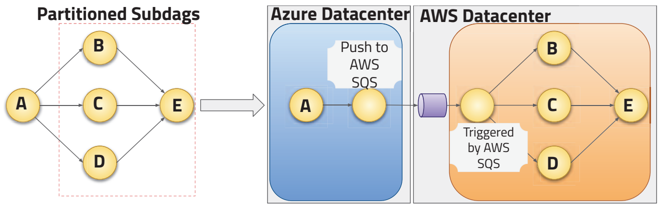


Fig. 9: Partitioning and deploying a user DAG

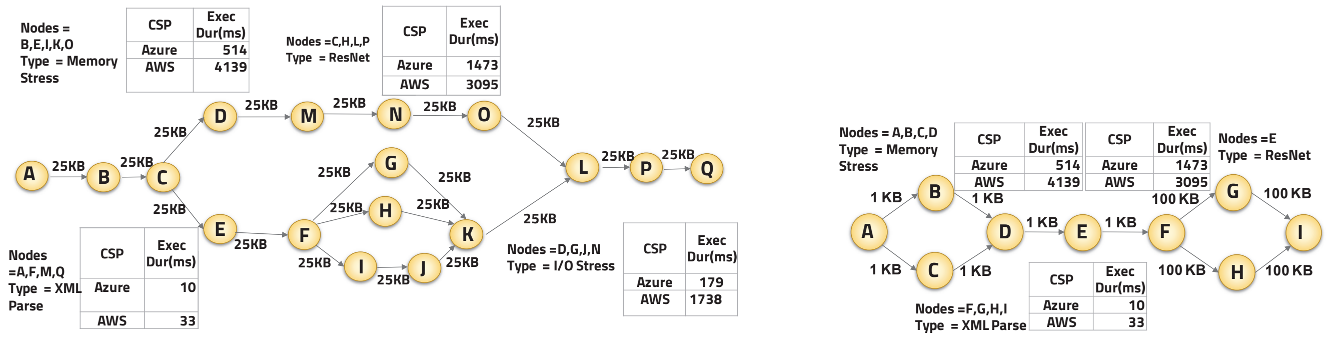
After the planner returns an option split point and the (sub)DAGs, we generate the relevant code and create any queue resources required to operate across clouds. If the DAG is partitioned, then boundary functions like *Push-to-Queue* and *Queue Triggered* functions will also be created, as discussed before. XFaaS abstracts away this complexity from the user.

Fig. 9 shows how a partitioned DAG is deployed across multi-cloud. Here, the planner has decided to place node A in Azure and nodes B, C, D and E on AWS. Two new *Push-to-Queue* and *Queue Triggered* functions have been added to the sub-DAG. Also, an SQS queue has been created in AWS that receives messages from Push-to-Queue and triggers the Queue Triggered function.

VIII. Experiments and Results

We evaluate the impact of partitioning and fusion on two DAGS [45]: *Smart Grid* (Fig. 10a and *Dumbbell* (Fig. 10b). The functions (nodes) are of four types: memory stress, ResNet inferencing (floating point compute), XML parsing (integer compute) or I/O stress. The execution times for these are given in the tables inset in Fig. 10. The edges are annotated with the data transfer size.

Our experiments are conducted in the Azure (Central-India) and AWS (ap-south-1) data centers. For both the clouds the client is placed a single VM within that cloud. For Azure



Nodes = B,E,I,K,O Type = Memory Stress

CSP	Exec Dur(ms)
Azure	514
AWS	4139

Nodes =C,H,L,P Type = ResNet

CSP	Exec Dur(ms)
Azure	1473
AWS	3095

Nodes =A,F,M,Q Type = XML Parse

CSP	Exec Dur(ms)
Azure	10
AWS	33

Nodes =D,G,J,N Type = I/O Stress

CSP	Exec Dur(ms)
Azure	179
AWS	1738

(a) Smart Grid DAG

Nodes = A,B,C,D Type = Memory Stress

CSP	Exec Dur(ms)
Azure	514
AWS	4139

Nodes =E Type = ResNet

CSP	Exec Dur(ms)
Azure	1473
AWS	3095

Nodes =F,G,H,I Type = XML Parse

CSP	Exec Dur(ms)
Azure	10
AWS	33

(b) Dumbbell DAG

Fig. 10: DAGs of the two FaaS workflows used in our experiments.

Durable Functions the VM is the Standard D8s v3 type with 8 vCPUs and 32GB RAM, while for AWS Step Functions we use an EC2 instance of C5.4X Large type with 16 vCPUs and 32GB RAM. We execute these workflows from a client in the same data center at a rate of 1 request per second, unless noted otherwise.

A. *Impact of XFaaS Partitioning and Placement*

1) Multi-Cloud execution of workflows can yield lower latencies than executing in a single cloud: Fig. 11b shows the median observed latency (in seconds) on the Y axis for each valid partition point of the Dumbbell DAG. The X axis shows the edges which are candidate partition points in the DAG. When the partition spans multiple edges, eg., a partition across edges, C-D and C-D, we use C-* to represent that partition point. The left bar (light green) gives the latency when the sub-DAG to the left of the function is placed on AWS and the right sub-DAG on Azure, and the right bar (dark green) means that the left sub-DAG is on Azure and the right on AWS. The last two bars are the latencies when the entire DAG is on AWS (orange) or Azure (blue).

We observe that the median latency for the edge D-E Azure to AWS is 24% lower than running the entire Dumbbell workflow on Azure: 7.3 secs vs. 9.6 secs, and 57% lower than running the entire workflow on AWS: 7.3 secs vs. 17.1 secs. We attribute this behaviour to the $\approx 8\times$ slower execution of Memory Stress (Functions A, B, C and D) on AWS in comparison to Azure. Partitioning the workflow helps leverage the faster execution on Azure before switching to AWS for better inter-function latency for the large message exchanges between Functions F, G, H and I, which perform XML Parsing.

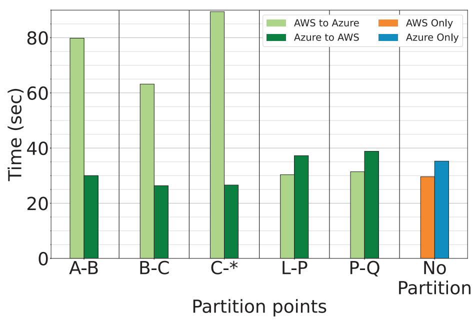
Fig. 11a shows the median observed latency (in seconds) on the Y axis for each valid partition point of the Smart Grid DAG. Even here, we observe that the multi-cloud execution C-* Azure to AWS has 13.3% lower latency than running entirely on Azure and 25.7% lower latency than running entirely on AWS.

2) The partitioner can identify multi-cloud splits that yields a low latency: While we have empirically observed that deploying a workflow across multiple clouds can reduce the latency of execution, we cannot realistically evaluate all possible partition points and CSP combinations. Hence, we rely on the partitioner to pick a multi-cloud configuration such that the latency is minimized. For the Dumbbell DAG, the partitioner selects E-F Azure to AWS. This configuration has the third lowest latency as seen in Fig. 11b, 8.27 secs, but this is very close to the actual lowest which is 7.30 secs. This is still much better than splitting at the other spots. Analytically, the partitioner chooses this split point to leverage the faster execution on Azure for function E (ResNet), which takes 1.47 secs on Azure vs. 3.09 secs on AWS. For the Smart Grid DAG, the partitioner selects C-* Azure to AWS. This configuration has among the lowest latency in Fig. 11a, and comparable to A-B Azure to AWS and B-C Azure to AWS. This again uses the faster compute of Azure for function B (ResNet), before moving to AWS for the parallel sub-DAG with many edges, which cause higher inter-function latencies in Azure.

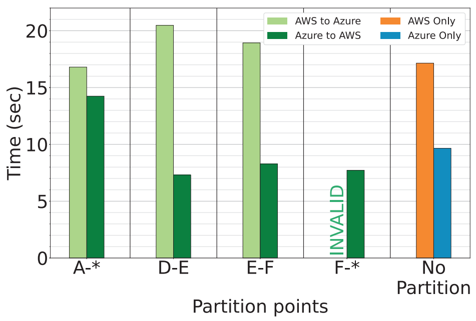
3) Partitioner respects user constraints: The partitioner also takes into account constraints like user-pinned nodes to a particular CSP, or the intra or inter cloud payload size limits, like < 64 Kb ingress into Azure, and < 256 KB intra-cloud payloads in AWS. In Fig. 11b, the partition edge F-* AWS to Azure is INVALID because of the 64KB input constraint in Azure Durable Functions. To verify this feature, we specify pinning constraints on node B and G of the Dumbbell DAG to AWS, as part of the DAG. This gives only two remaining options for the partitioner, A-* Azure to AWS, or run the entire DAG on AWS. Our partitioner picks the former. This can be explained from Fig. 11b, where the dark green bar shows A-* Azure to AWS having a median latency of ≈ 14 secs, as compared to ≈ 17 secs for the orange bar denoting that everything runs on AWS. Our partitioner is able to rightly estimate the expected time and choose the best partition point, after applying the constraints.

B. *Analysis of Function Fusion*

Here, we evaluate the effects of function fusion on the Smart Grid DAG. We do not perform partitioning over here, and evaluate the benefits of fusion when running the DAG fully on Azure or on AWS.

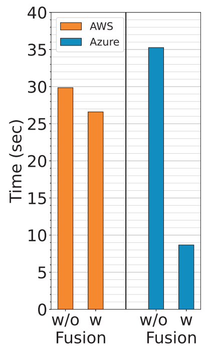


(a) Smart Grid DAG

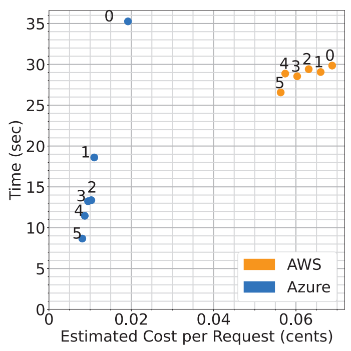


(b) Dumbbell DAG

Fig. 11: Impact of effective partitioning on the workflow's E2E latency.



(a) Impact of fusion on latency



(b) Time vs Cost estimates for different rounds of fusion

Fig. 12: Evaluating fusion on Smart Grid DAG

1) Function fusion discovers lower latency configurations: Fig. 12a shows the bar plots for the median latencies for the Smart Grid DAG executing completely on AWS (orange) and Azure (blue) , but with fusion disabled (left bar) or enabled (right bar). On AWS Step Functions, we observe a small reduction in latency from 29.85 secs to 26.59 secs when fusion logic is enabled. This is due to the higher function execution times and lower inter-function times for AWS estimated by our fusion logic. For AWS, the fusion logic fuses nodes $\{A, B, C\}$ into one node, $\{E, F\}$ into another node, and $\{L, P, Q\}$ into a third node. All these nodes are linear nodes and fall on the critical path. Overall, fusion for AWS reduces only 5 edges, and this gives a smaller reduction in latency. For Azure, interestingly, the fusion logic fuses all 17 nodes into a single node. Here, the inter-function latency dominates the execution times and hence it is better to fuse all nodes together. *Post fusion, Azure Durable Functions exhibits a lower latency than AWS Step Functions* 8.67 *secs vs.* 26.59 *secs.*

2) Fusion may also reduce the cost: Fig. 12b shows a scatter plot for the observed latency in seconds vs. the estimated cost in cents, for the decisions made by the fusion logic on the Smart Grid DAG for both AWS and Azure deployments. The number labels indicate the series of incremental decisions taken by the fusion agorithm and the corresponding time and cost for these. Here, we observe a reduction in both latency and the estimated cost. For Azure we see a drop in the cost from ¢0.019 to ¢0.0081 per execution, and for AWS, we see a drop from ¢0.068 to ¢0.056. Since users are billed either directly or indirectly for the number of nodes in the workflow, reducing the number of nodes leads to a reduction in cost for this combination of function execution time and memory used.

IX. Conclusions

In this paper, we identified the opportunities, limitations and challenges presented by the execution of FaaS workflows on multiple public cloud service providers. We conduct a review of their performance in order to gain key insights, and also build performance models that help us estimate the execution times and inter-function latencies. At the simplest level, XFaaS allows automated packaging and deployment of portable functions and DAGs across one or more clouds. As an added capability, it intelligently applies optimizations like function fusion and deciding partition split points to reduce the workflow execution latency while mitigating increases in costs. These are validated through realistic workloads executing on Azure and AWS, and our optimizations provide practical improvements in latency and/or cost on real clouds. They help developers reduce the CSP-specific FaaS code they write, and automate decisions for the operations team to pick what to run where.

As part of future work, we plan on incorporating more cloud service providers, support edge devices, and also expose data external dependencies like accessing NoSQL databases etc. to guide the placement decisions. We also plan to combine fusion and partitioning to give more optimization choices.

Acknowledgments

This work was supported through an open research grant towards the IBM-IISc Hybrid Cloud Lab (IIHCL) from IBM India Research Lab (IRL). The authors would like to thank Sameep Mehta, Arvind Agarwal and Balaji Ganesan from IRL for discussions that helped shape this paper.

References

[1] G. C. Fox, Vatche Ishakian, V. Muthusamy, and A. Slominski, "Status of serverless computing and function-as-a-service (faas) in industry and research," International Workshop on Serverless Computing (WoSC), Tech. Rep., 2017.

[2] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The rise of serverless computing," *Communications of the ACM*, vol. 62, no. 12, pp. 44–54, 2019.

[3] AWS, "Aws lambda," 2023, https://aws.amazon.com/lambda/.

[4] M. Azure, "Azure functions," 2023, https://azure.microsoft.com/en-in/products/functions/.

[5] Google, "Cloud functions," 2023, https://cloud.google.com/functions.

[6] IBM, "Ibm cloud functions," 2023, https://www.ibm.com/cloud/functions.

[7] OpenFaas, "Openfaas: Serverless functions, made simple," 2023, https://www.openfaas.com/.

[8] Knative, "Serverless containers in kubernetes environments," 2023, https://knative.dev/docs/.

[9] AWS, "Aws step functions," 2023, https://aws.amazon.com/step-functions/.

[10] M. Azure, "What are durable functions?" 2023, https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview.

[11] S. Ristov, S. Brandacher, M. Felderer, and R. Breu, "Godeploy: Portable deployment of serverless functions in federated faas," in *IEEE Cloud Summit*, 2022, pp. 38–43.

[12] P. Rodrigues, F. Freitas, and J. Simão, "Quickfaas: Providing portability and interoperability between faas platforms," *MPDI Future Internet*, vol. 14, no. 12, p. 360, 2022.

[13] CNCF, "Serverless workflow," 2023, https://serverlessworkflow.github.io/.

[14] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot, "Benchmarking, analysis, and optimization of serverless function snapshots," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021, pp. 559–572.

[15] N. Daw, U. Bellur, and P. Kulkarni, "Xanadu: Mitigating cascading cold starts in serverless function chain deployments," in *ACM/IFIP International Middleware Conference*, 2020, pp. 356–370.

[16] S. Burckhardt, B. Chandramouli, C. Gillum, D. Justo, K. Kallas, C. McMahon, C. S. Meiklejohn, and X. Zhu, "Netherite: Efficient execution of serverless workflows," *Proceedings of the VLDB Endowment*, vol. 15, no. 8, pp. 1591–1604, 2022.

[17] S. Kotni, A. Nayak, V. Ganapathy, and A. Basu, "Faastlane: Accelerating Function-as-a-Service workflows," in *USENIX Annual Technical Conference (ATC)*, 2021, pp. 805–820.

[18] A. Khochare, Y. Simmhan, S. Mehta, and A. Agarwal, "Toward scientific workflows in a serverless world," in *2022 IEEE 18th International Conference on e-Science (e-Science)*, 2022, pp. 399–400.

[19] R. Crespo-Cepeda, G. Agapito, J. L. Vazquez-Poletti, and M. Cannataro, "Challenges and opportunities of amazon serverless lambda services in bioinformatics," in *ACM International Conference on Bioinformatics, Computational Biology and Health Informatics*, 2019, pp. 663–668.

[20] A. John, K. Ausmees, K. Muenzen, C. Kuhn, and A. Tan, "Sweep: accelerating scientific research through scalable serverless workflows," in *IEEE/ACM International Conference on Utility and Cloud Computing Companion*, 2019, pp. 43–50.

[21] R. Chard, Y. Babuji, Z. Li, T. Skluzacek, A. Woodard, B. Blaiszik, I. Foster, and K. Chard, "Funcx: A federated function serving fabric for science," in *International symposium on high-performance parallel and distributed computing (HPDC)*, 2020, pp. 65–76.

[22] M. Malawsk, "Towards serverless execution of scientific workflows – hyperflow case study," in *Workflows in Support of Large-Scale Science (WORKS) Workshop*, 2016.

[23] R. B. Roy, T. Patel, V. Gadepally, and D. Tiwari, "Mashup: making serverless computing useful for hpc workflows via hybrid execution," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2022, pp. 46–60.

[24] N. Bila, P. Dettori, A. Kanso, Y. Watanabe, and A. Youssef, "Leveraging the serverless architecture for securing linux containers," in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, 2017, pp. 401–404.

[25] CNCF, "Synapse," 2023, https://github.com/serverlessworkflow/synapse.

[26] Eventmesh, "Apache eventmesh," 2023, https://eventmesh.apache.org/.

[27] Quarkus, "Quarkus funqy portable java api," 2022, https://quarkus.io/guides/funqy.

[28] Kogito, "Kogito serverless workflow guides," 2023, https://kiegroup.github.io/kogito-docs/serverlessworkflow/latest/index.html.

[29] Minikube, "Minikube," 2023, https://minikube.sigs.k8s.io/docs/start/.

[30] KubeFlow, "The machine learning toolkit for kubernetes," 2023, https://www.kubeflow.org/.

[31] Argo, "Argo workflows - the workflow engine for kubernetes," 2023, https://argoproj.github.io/argo-workflows/.

[32] Terraform, 2022, https://www.terraform.io/.

[33] Serverless, 2022, https://www.serverless.com/.

[34] A. Fuerst and P. Sharma, "Faascache: Keeping serverless computing alive with greedy-dual caching," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021, p. 386–400.

[35] Z. Li, L. Guo, Q. Chen, J. Cheng, C. Xu, D. Zeng, Z. Song, T. Ma, Y. Yang, C. Li *et al.*, "Help rather than recycle: Alleviating cold startup in serverless computing through {Inter-Function} container sharing," in *USENIX Annual Technical Conference (ATC)*, 2022, pp. 69–84.

[36] A. Mahgoub, L. Wang, K. Shankar, Y. Zhang, H. Tian, S. Mitra, Y. Peng, H. Wang, A. Klimovic, H. Yang *et al.*, "{SONIC}: Application-aware data passing for chained serverless applications," in *USENIX Annual Technical Conference (ATC)*, 2021, pp. 285–301.

[37] A. Mahgoub, E. B. Yi, K. Shankar, E. Minocha, S. Elnikety, S. Bagchi, and S. Chaterji, "Wisefuse: Workload characterization and dag transformation for serverless workflows," *ACM Measurement and Analysis of Computing Systems*, vol. 6, no. 2, pp. 1–28, 2022.

[38] A. Mahgoub, E. B. Yi, K. Shankar, S. Elnikety, S. Chaterji, and S. Bagchi, "{ORION} and the three rights: Sizing, bundling, and prewarming for serverless {DAGs}," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022, pp. 303–320.

[39] B. Raj, S. Dennis, J. Kevin, W. David, and B. Miguel, Angel, "Magic quadrant for cloud infrastructure and platform services," Gartner, Tech. Rep., 2022.

[40] S. Burckhardt, C. Gillum, D. Justo, K. Kallas, C. McMahon, and C. S. Meiklejohn, "Durable functions: semantics for stateful serverless," *ACM Programming Languages*, vol. 5, no. OOPSLA, pp. 1–27, 2021.

[41] P. Maissen, P. Felber, P. Kropf, and V. Schiavoni, "Faasdom: A benchmark suite for serverless computing," in *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems*, 2020, pp. 73–84.

[42] N. Mahmoudi and H. Khazaei, "Performance modeling of serverless computing platforms," *IEEE Transactions on Cloud Computing*, vol. 10, no. 4, pp. 2834–2847, 2020.

[43] X. Liu, J. Wen, Z. Chen, D. Li, J. Chen, Y. Liu, H. Wang, and X. Jin, "Faaslight: General application-level cold-start latency optimization for function-as-a-service in serverless computing," *ACM Transactions on Software Engineering and Methodology*, 2023.

[44] M. Williamson, "stickytape: Convert python packages into a single script," 2018, https://github.com/mwilliamson/stickytape.

[45] A. Shukla and Y. Simmhan, "Model-driven scheduling for distributed stream processing systems," *Journal of Parallel and Distributed Computing*, vol. 117, p. 98–114, 2018.

APPENDIX
ARTIFACT REPRODUCIBLITY

XFaaS allows "zero touch" deployment of Serverless functions and workflows across AWS and Azure by automatically generating necessary code wrappers, cloud queues, and coordinating with the native FaaS engine of a cloud provider. In, XFaaS we make the following contributions:

- **For Single Cloud:** Deploy common function logic as workflows in AWS Stepfunctions and Azure Durable Functions.
- **For Single Cloud Fusion:** Based on prior benchmarks, deploy modified workflows with *function fusion* in AWS and Azure.
- **For Multi Cloud:** Based on prior benchmarks, deploy a workflow *partitioned* across Azure and AWS.

In this appendix, we provide instruction to verify the functionality of these 3 contributions of our work using the SmartGridDAG from the main paper (Fig 10 (a)). The code is available at https://github.com/dream-lab/XFaaS under the CCGRID2023 branch.

A. Setup

Pre-requisites: Linux desktop with docker installed and Internet connection.

We provide a docker container that has all the XFaaS dependencies pre-installed. Line 1 in Listing 1 clones the github repository so that we can build the docker container. Line 3 in Listing 1 builds the container using the Dockerfile provided with the submission. Line 4 starts a docker container in the background and line 5 provides a bash terminal inside the container. Lines 7 and 8 provide steps needed for logging into the cli of Azure and AWS respectively. Line 9 clones the XFaaS repository into the root directory and line 11 installs the necessary python3 requirements into the docker container. With that the docker environment setup is complete. Note that this step may have to be repeated if the docker container is stopped.

Verification: You can check if the docker container is running using `docker ps | grep xfaas` on the host Linux desktop.

```
1  git clone https://github.com/dream-lab/XFaaS.git
2  cd XFaaS
3  docker build -t xfaas:1.0 .
4  docker run -d --name xfaas-container xfaas:1.0
5  docker exec -it xfaas-container bash
6  <--docker exec should bring you inside the docker
       container-->
7  az login -u <username> -p <password>
8  aws configure
9  git clone https://github.com/dream-lab/XFaaS.git
10 cd XFaaS
11 ./setup.sh
```

Listing 1: Setting up Docker container for the experiments

There are 5 folders under the *serwo/examples* directory each corresponding to an experiment. This is purely for convenience and users can verify that the code under all directories is exactly the same by running diff. Eg: Listing 2.

```
diff smart-grid-singlecloud-aws/src/
    iostress_512wr_25KB/func.py smart-grid-
    singlecloud-azure/src/iostress_512wr_25KB/func.
    py
```

Listing 2: *diff* across folders

The XFaaS directory structure is as follows:

```
XFaaS
├─ serwo
│   └─ examples
│       ├─ smart-grid-singlecloud-aws
│       ├─ smart-grid-singlecloud-azure
│       ├─ smart-grid-fusion-aws
│       ├─ smart-grid-fusion-azure
│       └─ smart-grid-multicloud
├─ run.py
├─ run_client.py
└─ plot_timeline.py
```

We provide a client (`python3 run-client.py --exp-name <experiment-name>`) that uses *jmeter* to make HTTP requests to the deployed workflows for 10 mins at 1 request per second. And a plotting script (`python3 plot-timeline.py <experiment-name>`) that plots the end-to-end latency of these requests and saves it as a pdf (<experiment-name_e2e_timings.pdf>).

B. Single Cloud

Pre-requisites: A Running XFaaS docker container In this group of experiments, we will deploy workflows to AWS Step Functions and Azure Durable Functions.

1) AWS Deployment: Command 1 shows the command to be executed (experiment name: *smart-grid-singlecloud-aws*) – single cloud deployment of the workflow to AWS Step Functions. The command may take a few minutes to complete. Once the workflow has been deployed, we encourage users to navigate to `/XFaaS/serwo/examples/smart-grid-singlecloud-aws/build/workflow/aws` and explore the auto-generated directories and code specific to AWS Step Function deployment. Users can also use the `aws stepfunctions list-state-machines` command and verify that a new Step Function has been created.

```
cd /XFaaS
python3 run.py examples/smart-grid-singlecloud-aws
    dag-description.json --single-cloud aws
```

Command 1: Single Cloud AWS

2) Azure Deployment: Command 2 shows the command to be executed (experiment name: *smart-grid-singlecloud-azure*) – single cloud deployment of the workflow to Azure Durable Functions. Once it executes, we encourage the users to explore the auto-generated directories and code specific to Azure Durable Functions in the build directory `/XFaaS/serwo/examples/smart-grid-singlecloud-azure/build/workflow/azure`.

```
cd /XFaaS
python3 run.py examples/smart-grid-singlecloud-azure
     dag-description.json --single-cloud azure
```

Command 2: Single Cloud Azure

Verification: In the Single Cloud experiments, we verify that common code can be deployed to AWS Step Functions and Azure Durable Functions, with XFaaS auto-generating the code in the build directory and deploying it. For both the experiments, users can make requests to the deployed workflow using the provided client (`run_client.py`) and plot the results (`plot_timeline.py`) using the commands mentioned in the Setup section.

C. Single Cloud with Fusion

Pre-requisites: A Running XFaaS docker container

1) AWS Deployment with Function Fusion:: Command 3 enables and deploys function fusion for AWS Step Functions (experiment name: *smart-grid-fusion-aws*).

```
cd /XFaaS
python3 run.py examples/smart-grid-fusion-aws dag-
    description.json --fusion aws
```

Command 3: Single Cloud AWS Fusion

Once the workflow has been deployed, we encourage the users to navigate to `/XFaaS/serwo/examples/smart-grid-fusion-aws/build/workflow/aws` and explore the auto-generated directories and code specific to AWS Step Function deployment. The fused function code is available under `serwo/examples/smart-grid-fusion-aws/src-fused-AWS`

Command 4 enable and deploy function fusion for Azure Durable Functions (experiment name: *smart-grid-fusion-azure*).

```
cd /XFaaS
python3 run.py examples/smart-grid-fusion-azure dag-
    description.json --fusion azure
```

Command 4: Single Cloud Azure Fusion

Once the workflow has been deployed, we encourage the users to navigate to `/XFaaS/serwo/examples/smart-grid-fusion-azure/build/workflow/azure` and explore the auto-generated directories and code specific to Azure Durable Functions. The fused function code is available under `serwo/examples/smart-grid-fusion-azure/src-fused-Azure`

Verification: In the Single Cloud with Fusion experiments, we verify that fused code can be deployed to AWS Step Functions and Azure Durable Functions, with XFaaS auto-generating the code in the build directory and deploying it. For both the experiments, users can make requests to the deployed workflow using the provided client (`run_client.py`) and plot the results (`plot_timeline.py`) using the commands mentioned in the Setup section.

D. Multi-Cloud partitioning

Pre-requisites: A Running XFaaS docker container

Command 5 enables and deploys a partitioned workflow across AWS and Azure (experiment name: *smart-grid-multi-cloud*).

```
cd /XFaaS
python3 run.py examples/smart-grid-multicloud dag-
    description.json --partition
```

Command 5: Multi-Cloud Partitioned

Verification: Users can verify the auto-generation of code wrappers for partitioning by navigating to the build directory. Here, the DAG is partitioned at Task B In the Azure build directory, only TaskA and TaskB are auto-generated along with a few other helper functions that perform the Egress. In the AWS build folder, code for TasksC to TaskQ is auto-generated along with the function to Collect Logs. An SQS queue is auto-generated and deployed for the inter-cloud message passing. Users can find the name of the generated queue under `serwo/examples/smart-grid-multicloud/build/workflow/resources/aws-cloudformation-outputs.json`. The *OutputValue* of the *"OutputKey": "SQSResource"* is the URI of the SQS queue. Using the `aws sqs list-queues`, users can verify that the queue has been deployed in AWS.