Academic Session: Fall 2022-Spring 2023

Non Thesis Masters Project Report
Performance Analysis in Storage Engines for Serverless Workflow Execution with Durable Functions

Saurabh Vijay Surana*
CSCI 7200 Masters Project
School of Computing
University of Georgia at Athens
(Dated: December 15, 2022)

The proposal idea is taken from publications in [1] which have a collection of ongoing research at Microsoft R&D and collaborations with Carnegie Mellon and University of Pennsylvania. This is an applied research project where majority of the time will be used to research and test the implementation of the tool called as Netherite [2] as well as build an application featuring real world API and load test it with thousands of user request for analyzing the response time. The project will run under Microsoft's Azure Function service for serverless application deployments.

I. STATEMENT OF THE PROBLEM

Applied research project into storage solutions efficient serverless workflow execution aims to find out how the tool **Netherite** *optimizes and guarantees dependable scalable execution on unpredictable serverless workers* by testing saving of application function states in an elastic environment with a storage providers as its performance bottleneck when facing multiple thousand concurrent requests and a limited capacity persistence layer.

II. INTRODUCTION & BACKGROUND

The backbone of the recent internet is shifting to huge data and computing houses which are capable of hosting everything on the web in the cloud. Cloud has become the standard for deploying solutions quickly where the host of the site is unaware and worry-free of the physical demands of running a setup to support the application. The host is only served with a utility financial bill for the resource consumption which are highly dynamic that is load based and broken down to the milliseconds and bytes of the CPU and memory resource utilization respectively. Serverless microservices is an approach to building distributed applications where each component is a standalone, self contained service that runs in its own process and communicates with other components through well-defined API's. *Serverless compute services allows developers to run functions without provisioning or managing servers. It is a cont-effective solution for running event-driven and asynchronous workloads.*

Microsoft's **Azure Functions** [3] provides this functionality for developers to deploys such pieces of code where the main focus of the developer is the implementation of the function logically while being carefree of the nitty gritty of its deployment. Azure Functions are billed only for the count of triggers and the resource utilization during the execution of such triggers.

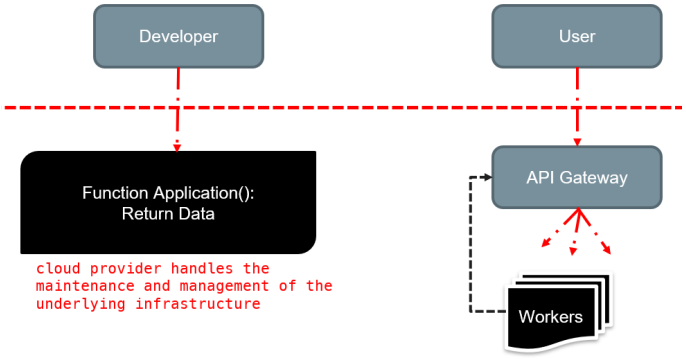
Modern application have the need of being accessible from any point on the globe and thus make use of content delivery networks or CDN's for providing data regionally with minimal latency. The research project is going to focus on the implementation of cloud service architectures where developers and businesses host their applications with minimal deployment effort. Different applications have different needs and the cloud service providers are aware of these requirements which leads to the development of several **architectural patterns** to manage the workflow execution.

Serverless is a recent buzzword which provides Functions as a Service or FaaS. The term serverless makes more sense when seen with the context of stateless and state functions which are basically pure and impure functions depending on the input injection required to run them. A pure function is solely dependent on the input supplied to it whereas impure functions have additional dependencies which we refer to as the current state of the function which changes given the input. Stateless functions are horizontally scalable as we can spin new instances of these functions depending on the user load decrease when not required. On the other hand stateful functions require persistent storage for at least intermediate computations whose scaling is vertical in nature with the addition of multiple copies of same resource which is a very inefficient solution. Thus initially serverless methodology was only used for stateless functions but that cant make the entire application serverless. To address this issue several frameworks were developed for stateful serverless application execution which saves the states of such functions at different checkpoints so we can resume their execution in case of failures, non-availability of resources.

One of the way of implementing these serverless stateful functions is through **Durable Functions** or DF(Azure

* saurabhvijay.surana@uga.edu

terminology) [4]. These functions enable developers to break jobs into orchestrations of parallel workflows from the native language itself. These orchestration store the states as shared objects in *entities* which can be called upon later for the stateful job execution. Thus using this serverless architecture cloud service providers are better able to manage resources in an elastic environment where larger than memory stateful applications can be run with the use of FASTER [5].

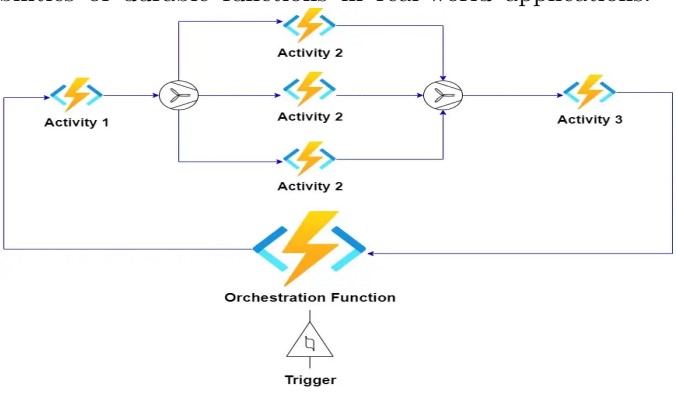


How Serverless eases the software development DevOps

III. ARCHITECTURAL PATTERNS

A. Durable Functions

Durable functions are a type of Azure Function that provides the ability to write stateful functions in a serverless environment. These functions allow developers to create long-running, stateful logic in a reliable and scalable way. One common application pattern for durable functions is orchestrator functions. These **functions** are used to coordinate the execution of multiple sub-functions and manage their state. For example, an orchestrator function could be used to implement a workflow where multiple tasks need to be performed in a specific order, and the orchestrator function is responsible for triggering each task and managing the overall state of the workflow. Another application pattern for durable functions is fan-out/fan-in. This pattern is used when a large amount of work needs to be performed in parallel, and the results of that work need to be combined. In this pattern, the orchestrator function triggers multiple sub-functions to perform the work in parallel, and then waits for all of the results to be returned before combining them and returning the final result. Durable functions can also be used to implement human-in-the-loop processes. In this pattern, a workflow is triggered and performs some initial processing, but then waits for human input before continuing. The orchestrator function can wait for a signal from the human user (such as a webhook call) before continuing the workflow. In conclusion, durable functions provide a powerful way to implement stateful, long-running logic in a serverless environment. The use of orchestrator functions, fan-out/fan-in, and human-in-the-loop patterns are common ways to leverage the capabilities of durable functions in real-world applications.



B. Test Application: Book Processor

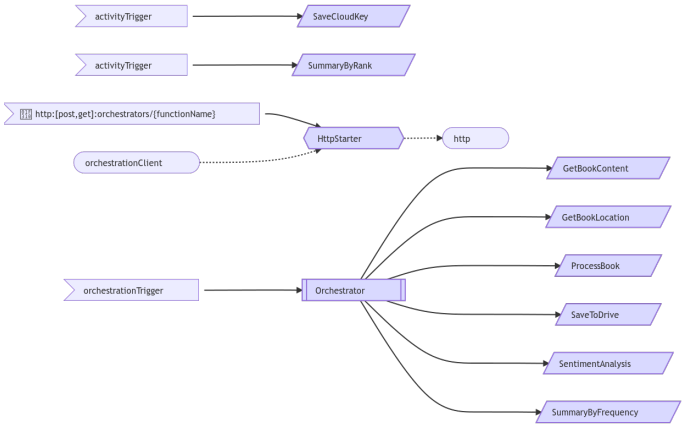
The pattern demonstrated above is like a simplistic real world application where the Trigger(an HTTP request, Timer Events, Human Intervention) would create an instance of the Orchestrator function in the cloud assigned to an available worker or else wait in the queue of Azure EventsHub. After the Instance is created it starts by calling the activity execution defined by the application patterns in developer code with synchronous and asynchronous calls and the orchestrator executes them as such and collects the results and returns the output. Below is the example of the orchestrator code for the demo testing application.

```
1
2   import * as df from "durable-functions"
3   const orchestrator = df.orchestrator(function*(context) {
4       let bookPath: string = yield context.df.callActivity(GetBookLocation);
5       if (bookPath === undefined)
6           return `Could not find book in the directory or directory was empty!`
7       let book: string = yield context.df.callActivity(GetBookContent);
8       if (book === undefined) {
9           return `Unable to read book.`;
10      } else if (book === NEXT_BOOK) {
11          return `Book found in incompatible language.`;
12      } else {
13          context.bindingData.bookContent = book;
14      }
15      let processedBook = yield context.df.callActivity(ProcessBook);
16      context.bindingData.bookContent = processedBook;
17      let uploadToCloud: any = null;
18      if (processedBook !== undefined) {
19          let _a = context.df.callActivity(SummaryByFrequency);
20          let _b = context.df.callActivity(SentimentAnalysis);
21          context.bindingData.bookSummary = yield context.df.Task.all([_a, _b]);
22          uploadToCloud = yield context.df.callActivity(CONSTANTS.SaveToDrive);
23      }
24      return uploadToCloud;
25  });
```

C. Technical Implementation Details

The project was compiled in Typesafe Javascript (Typescript) on a Node LTS 16.0 server on a windows machine. The orchestrator function is implemented using **generator functions** in javascript. In JavaScript,

generator functions are functions that can be paused and resumed at any time. This is useful because it allows the function to return intermediate results, instead of waiting until the entire computation is finished before returning the final result. Generator functions are created using the function* syntax, and they are typically used in conjunction with the yield keyword to pause and resume the function.
Thus the generator function is the perfect solution for running orchestrations and are invoked when the activities complete their executions. The cloud service provider charges the host for the function call invocations along with CPU and memory usage thus it is up to the developer to optimize their code for cost effective deployments.

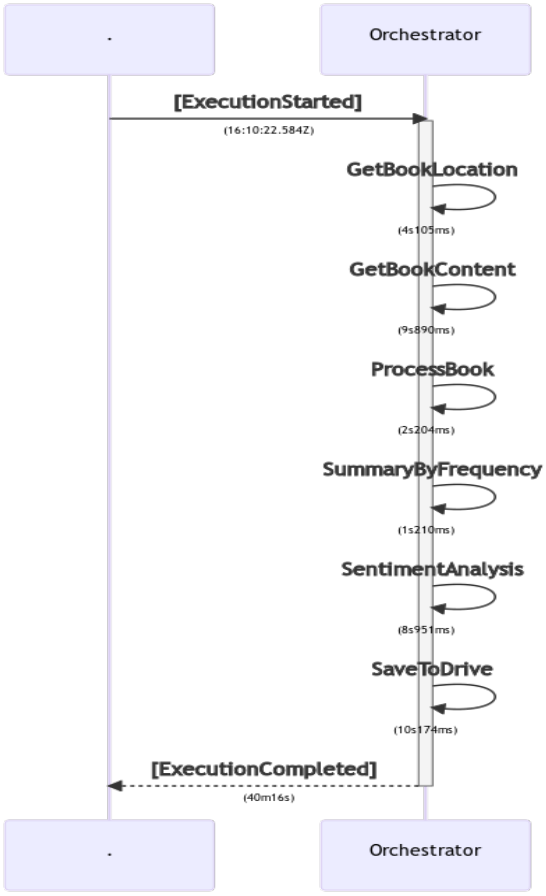


Orchestration, Functions and Triggers for gutenberg book processor

IV. JUSTIFICATION

Serverless Architecture provide developers with the freedom of getting their application out in the market faster on a host with high performance and scalability support with low cost of ownership as they pay only for the usage costs while having the security benefits from general attacks as the Host's design focuses on isolation and fault tolerance for running each such application. All these benefits are available simply with the addition of the developers functions to the cloud.

Originally such applications where only supported for the stateless functions which are idempotent in nature; they do not require a predefined state to run an incoming input every time the function is used where the state updates every time it is run. **Stateful serverless** has also emerged as a strong contender to replace the existing cloud architectures with the addition of *Orchestrations, Entities, Actors and Activities, Critical Sections* for enabling developers to implement long running computations and workflows with persistence at various checkpoints of a functions life cycle.
In traditional enterprise development cycle the developer of the application has to communicate with DevOps team for the particulars of the deployment and the expected resource requirements of the applications and the DevOps then accordingly parameterize's the allocation of resources. Complete serverless architecture which was initially thought of as difficult to implement due to quick rise in the complexity of managing scalable applications in the future without vertically scaling which is costly from a business perspective. Durable Functions solve these shortcomings of complete serverless architectures by giving developers the flexibility of writing the workflows in their native development language called as orchestrations. These orchestrations are still prone to the I/O bottleneck of the database servers which are taken care by Netherite. //



Flow of Orchestration in Durable Function

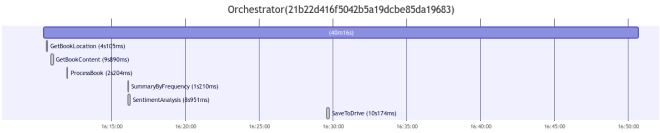


FIG. 1. Gantt Chart for flow of API

V. THEORETICAL FRAMEWORK

The semantics of running these durable functions are defined in a high level mathematical representations in [6]. For the purpose of this research these models will be used to write logical new abstractions according to the theorems defined. Most of the work will be in form of active development and analysis with various workloads and implementations in common enterprise applications such as compute heavy mathematical solvers for scientific research, streaming platforms which use heavy video encoding and decoding engines for distributing content.

VI. AZURE PORTAL CONFIGURATIONS

A. Resource Groups

I availed the resource groups created under University of Georgia (groups.uga.edu) with a $200 account credit. In Microsoft Azure, a resource group is a logical collection of resources that share the same lifecycle, permissions, and policies. You can use resource groups to manage resources such as web apps, databases, and storage accounts.

B. Plans

There are four pricing tiers for Azure Functions:

- The Consumption Plan, which charges based on the number of executions and the compute time required to run your functions.
- The Premium Plan, which provides enhanced performance and scaling capabilities for your functions, along with a fixed monthly cost.
- The Dedicated (App Service) Plan, which lets you run your Azure Functions on dedicated VMs that are managed by App Service.
- The Elastic Premium Plan, which provides the same features as the Premium Plan, but with the ability to automatically scale your functions based on demand.

I ran my functions and load testing in Consumption plan as well as the Elastic Premium Plan EP1 with base setup with and without Netherite.

C. Load Testing Scripts

```
import time
import logging
import asyncio
from aiohttp import ClientSession, ClientResponseError
logging.getLogger().setLevel(logging.INFO)
# API_URL_CONSUMPTION =
# "https://mastersprojectss49293.azurewebsites.net/api/orchestrators/Orchestrator"
# API_URL_PREMIUM =
# "https://gutenberg-df.azurewebsites.net/api/orchestrators/Orchestrator"
API_URL_NETHERITE =
"https://ugamastersss49293.azurewebsites.net/api/orchestrators/Orchestrator"
async def fetch(session, url):
    try:
        async with session.get(url, timeout=1000) as response:
            resp = response.read()
    except ClientResponseError as e:
        logging.warning(e.code)
    except asyncio.TimeoutError:
        logging.warning("Timeout")
    except Exception as e:
        logging.warning(e)
    else:
        return resp
    return
```

Python Scripts using asyncio for Asycnhronous HTTP Requests

```
async def fetch_async(loop, r):
    url = API_URL
    tasks = []
    async with ClientSession() as session:
        for i in range(r):
            task = asyncio.ensure_future(fetch(session, url))
            tasks.append(task)
        responses = await asyncio.gather(*tasks)
    return responses
if __name__ == '__main__':
    for ntimes in [100, 200, 300, 400, 500, 600, 700]:
        start_time = time.time()
        loop = asyncio.get_event_loop()
        future = asyncio.ensure_future(fetch_async(loop, ntimes))
        loop.run_until_complete(future)
        responses = future.result()
        logging.info('Fetch %s urls takes %s seconds',
        ntimes, str(time.time() - start_time))
        logging.info('{} urls were read successfully'
        .format(len(responses)))
```

The requests were made in batches of 100, 200, 300, 400 ,500, 600 with a time distance of random 1 $\tilde{2}$ seconds.

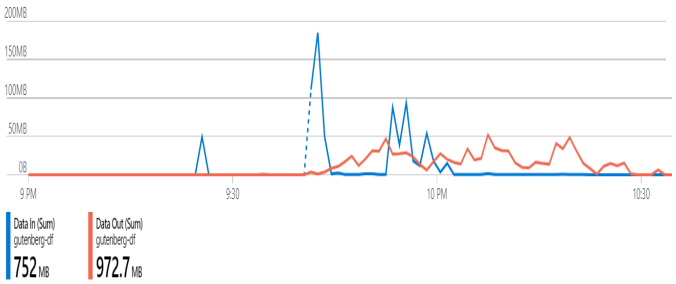
VII. RESULTS

A performance boost of 2x was observed running the application with Netherite.

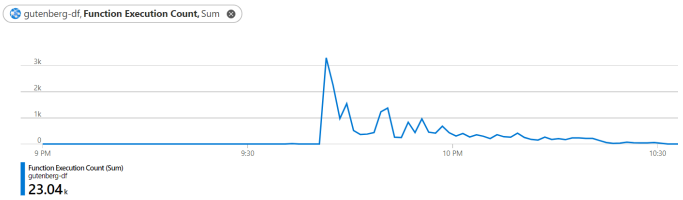
- Consumption Plan:2 hours for 2k requests
- EP1 without Netherite:1 hours for 2k requests
- EP1 with Netherite:0.5 hours for 2k requests

The last step of the orchestration involves saving the processed book output to google cloud storage bucket with no bandwidth limits. I was able to access a $ 300 credits for this with a new account creation. Data flow is good metric for performance analysis as every http request would require the application to fetch roughly 1 mb of data on average and output 1 mb of data on average to an external google cloud server and all these

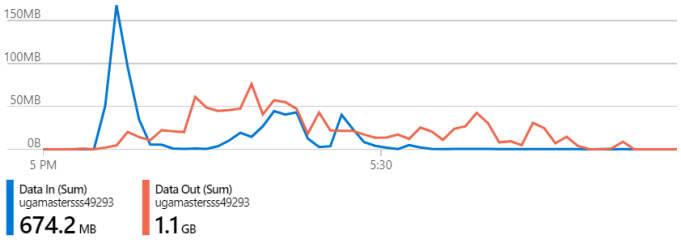
processes have to performed synchronously and causally dependent on each other as seen in the orchestrator function.



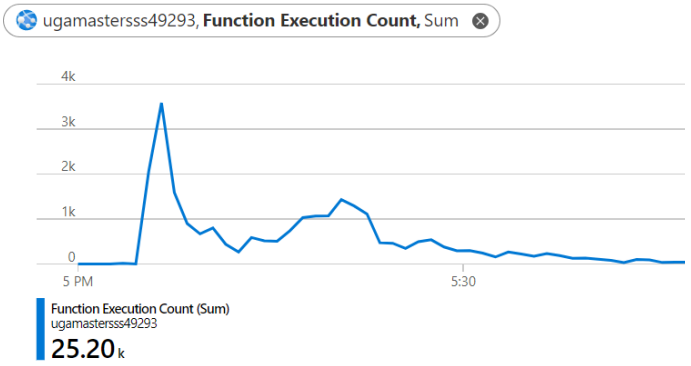
Data Flow without netherite



Function execution count without netherite



Data Flow with netherite



Function execution count with netherite

As evident with the results the functions were both run in same way and with netherite the execution time was reduced to half.

The link to the github repository is available at [7]

[1] Microsoft rise, `https://www.microsoft.com/en-us/research/group/research-software-engineering-rise/`, accessed: 2022-09-26.

[2] Netherite tool repository, `https://github.com/microsoft/durabletask-netherite`, accessed: 2022-09-26.

[3] Azure functions, `https://azure.microsoft.com/en-us/products/functions/#overview`, accessed: 2022-09-26.

[4] Durable functions, `https://learn.microsoft.com/en-us/azure/azure-functions/durable/`, accessed: 2022-09-26.

[5] B. Chandramouli, G. Prasaad, D. Kossmann, J. Levandoski, J. Hunter, and M. Barnett, Faster: A concurrent

key-value store with in-place updates, in *2018 ACM SIGMOD International Conference on Management of Data (SIGMOD '18), Houston, TX, USA* (ACM, 2018).

[6] S. Burckhardt, C. Gillum, D. Justo, K. Kallas, C. McMahon, and C. S. Meiklejohn, Durable functions: Semantics for stateful serverless, Proc. ACM Program. Lang. **5**, 10.1145/3485510 (2021).

[7] Github repository, `https://github.com/l1337xto/gutenberg-df`, accessed: 2022-12-15.